Methodische Entwicklung und rollenbasierte Integration von Komponentenframeworks

Marc Sihling

Technische Universität München
Institut für Informatik

Methodische Entwicklung und rollenbasierte Integration von Komponentenframeworks

Marc Sihling

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.

Vorsitzende: Univ.-Prof. Gudrun J. Klinker, Ph.D

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Manfred Broy
2. Univ.-Prof. Dr. Wolfgang Pree, Universität Konstanz

Die Dissertation wurde am 30.08.2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 09.11.2001 angenommen.

Kurzfassung

Ziel der Arbeit ist die flexible, modellbasierte Konstruktion von Softwaresystemen durch die Integration von Komponentenframeworks. Hierfür wird ein Verständnis der grundlegenden Konzepte entwickelt und ein geeigneter Integrationsmechanismus anhand eines durchgängigen Beispiels vorgestellt. Dabei erfolgt die Zusammenführung unterschiedlicher Aspekte einer Anwendung anhand konzeptueller, rollenbasierter Modelle und der analogen Komposition entsprechender Komponentenframeworks. Geeignete Beschreibungstechniken sowie ein Vorgehensmodell basierend auf Prozeßmustern stellen die Bausteine einer Methodik dar, die die Anwendbarkeit der eingeführten Konzepte gewährleistet.

Danksagung

Diese Arbeit reflektiert die vielfältigen Erfahrungen, Ergebnisse und Diskussionen des Teilprojekts „Methodik der bausteinorientierten Softwareentwicklung“ des Forschungsverbunds FORSOFT. In diesem Zusammenhang möchte ich mich vor allem bei dem A1-Team, namentlich Klaus Bergner, Andreas Rausch und Alexander Vilbig bedanken. Für hilfreiche Kommentare zu frühen Versionen dieser Arbeit gebührt Dank meinen Kollegen Katharina Spies und Bernhard Rumpe.

Bei Prof. Dr. Manfred Broy möchte ich mich für die vielfältige Unterstützung im Vorfeld und bei der Erstellung dieser Arbeit bedanken. Für die Übernahme des zweiten Gutachtens und für viele interessante Anregungen bin ich Prof. Dr. Wolfgang Pree von der Universtität Linz zu Dank verbunden.

Nicht zuletzt bedanke ich mich bei allen meinen Bekannten, meinen Eltern und vor allem bei Sabine für die Geduld, die sie während der Erstellung dieser Arbeit mit mir hatten.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Lösungsansatz 3

1.1.1 Frameworkintegration 4

1.1.2 Methodik 6

1.2 Ergebnisse der Arbeit 7

1.3 Verwandte Arbeiten 8

1.4 Aufbau der Arbeit 11

2 Grundlagen **13**

2.1 Komponenten 14

2.2 Softwarearchitektur 17

2.3 Frameworks 19

2.4 Rollen 20

3 Formale Modellierung **23**

3.1 Anforderungen 26

3.2 Einreichung zur OOPSLA (ein Beispiel) 27

3.3 Fachmodelle 28

3.3.1 Der Anwendungsbereich 29

3.3.2 Rollen 30
3.3.3 Assoziationen 32
3.3.4 Abhängigkeiten 33
3.3.5 Fachmodell 37
3.3.6 Ausblick: Rollen für Assoziationen 39
3.3.7 Zusammenführung von Modellen 40
3.3.8 Flache Integration 44
3.3.9 Hierarchische Integration 49
3.3.10 Beispiel 53
3.3.11 Zusammenfassung 56
3.4 Modellierung komponentenbasierter Systeme 57
3.4.1 Grundlagen 58
3.4.2 Zustand 59
3.4.3 Schnittstellen und Komponenten 60
3.4.4 Unterspezifikation 67
3.4.5 Zusammenfassung 68
3.5 Komponentenframeworks und Fachkomponenten 68
3.5.1 Vom Fachmodell zum Systemmodell 69
3.5.2 Zusammenspiel 70
3.5.3 Erweiterung des Systemmodells 72
3.5.4 Typkomposition 74
3.5.5 Subtyping 76
3.5.6 Komposition im Detail 80
3.5.7 Abbildung fachlicher Abhängigkeiten 88
3.5.8 Beispiel 90
3.5.9 Zusammenfassung 97

3.6 Erweiterung: Rollenkomponenten 99
3.6.1 Das „Zwiebelschalenmodell“ 100
3.6.2 Integration von Rollenkomponenten 101
3.6.3 Abbildung von Abstraktionsbeziehungen 103
3.6.4 Zusammenfassung 105
3.7 Zusammenfassung und Ausblick 106

4 Technische Umsetzung 109
4.1 Der Weg zur Implementierung 110
4.2 Komponententechnologien 113
4.3 Kernpunkte einer Umsetzung 116
4.3.1 Komponentenparadigma 116
4.3.2 Namensräume 119
4.3.3 Kanäle 120
4.3.4 Schnittstellen 122
4.3.5 Rollen 123
4.4 Erweiterung des CORBA Komponentenmodells 125
4.4.1 Komponententypen 125
4.4.2 Standardrolle und Verbindungsaufbau 129
4.4.3 Abhängigkeiten 131
4.4.4 Rollenkomponenten 133
4.4.5 Mediatoren 135
4.4.6 Component Implementation Framework 136
4.5 Implementierung 137
4.5.1 Infrastruktur für Rollenkomponenten 137
4.5.2 Kernkomponenten 142
4.6 Werkzeugunterstützung 143

4.7 Zusammenfassung 145

5 Frameworkdokumentation mit der UML 147

5.1 Grundzüge der UML 150

5.2 Erweiterungsmechanismus 151

5.3 Component Profile 153

5.4 Role Profile 158

5.5 Framework Profile 164

5.6 Zusammenfassung 165

6 Prozeßmusterorientiertes Vorgehensmodell 167

6.1 Vorgehensmodelle 168

6.1.1 Komponentenbasierte Softwareentwicklung 169

6.1.2 Entwicklung und Anwendung von Komponentenframeworks . . . 172

6.1.3 Allgemeine Richtlinien 176

6.2 Einführung in Prozeßmuster 177

6.3 Auswahl einiger Prozeßmuster 181

6.3.1 Entkopplung von Rollen 181

6.3.2 Zerlegung von Kollaborationen 183

6.3.3 Entwurf von Rollenkomponenten 184

6.4 Zusammenfassung 187

7 Zusammenfassung und Ausblick 189

Literaturverzeichnis 196

Kapitel 1

Einleitung

> The engineering method: „*Observe existing solutions, propose better solutions, build or develop, measure and analyze, repeat until no further improvements are possible*“ [Adr93].

Es ist seit längerer Zeit zu beobachten, daß der Anteil an Software und damit ihre Bedeutung im Alltag kontinuierlich ansteigt. Mit dem Ansatz, die Entwicklung von Software analog zu den „klassischen“ Ingenieursdisziplinen zu systematisieren und zu organisieren, wird vor allem der stetig steigenden Größe und Komplexität moderner Softwaresysteme begegnet. So ist ein weiter Bereich der Disziplin Software-Engineering getrieben von der Vision, Softwareprodukte — ähnlich wie in der Architektur oder im Maschinenbau — aus vorgefertigten Modulen zu fertigen und auf diese Weise insbesondere die gegenläufigen Kräfte zwischen Kosten, Qualität und Zeit besser auszubalancieren.

Der Begriff der Komponente markiert derzeit den letzten Meilenstein in einer Reihe teils sehr unterschiedlicher Modulkonzepte für die Entwicklung von Software. Und aufgrund eher enttäuschender Erfahrungen mit früheren Ansätzen hinsichtlich wünschenswerter Eigenschaften wie zum Beispiel Wiederverwendbarkeit aber auch Praxistauglichkeit waren in den vergangenen Jahren die Erwartungen an das Konzept „Komponente“ vergleichsweise hoch. Doch die Ideen, auf denen Komponenten basieren, entstanden zu großen

Teilen aus einer technischen Motivation, so daß teilweise noch bis heute über die zugrundeliegenden Konzepte und ihre methodische Anwendung unterschiedliche Vorstellungen existieren. Beispielsweise herrscht Uneinigkeit darüber, wie eine Anwendung geeignet in einzelne Module zerlegt werden kann, um den Nutzen der Einzelteile in anderen Anwendungen zu maximieren.

Die Etablierung eines gemeinsamen, präzisen Verständnisses über solche und verwandte Themenstellungen erscheint jedoch essentiell, um Komponenten zu erstellen, die auf „Komponentenmärkten“ gehandelt werden und mit vertretbarem Aufwand so angepaßt werden können, daß sie in unterschiedlichen Anwendungen Verwendung finden. Heutzutage existieren bereits einige Komponenten, die auf breiter Basis und erfolgreich wiederverwendet werden. Es handelt sich hierbei meist um eine Kapselung bestimmter Funktionalitäten (beispielsweise für die dauerhafte Speicherung von Daten in einer Datenbankkomponente). In den weitaus überwiegenden Fällen können jedoch Verantwortlichkeiten für Funktionalitäten nicht eindeutig einer Komponente zugewiesen werden, sondern finden sich stattdessen im Zusammenspiel mehrerer unterschiedlicher Komponenten. Und gerade das Wissen um dieses Zusammenspiel ist ein essentieller Bestandteil des Designs einer Anwendung und wert, auch in anderen Umgebungen wiederverwendet zu werden. Wünschenswert ist daher eine Abkehr von der bisherigen „introvertierten“ Komponentenentwicklung hin zu einer ganzheitlichen Sicht, die das Zusammenwirken einer Komponente mit den weiteren Bestandteilen einer Anwendung umfassender berücksichtigt.

So fand in den vergangenen Jahren gerade im Zusammenhang mit der komponentenbasierten Softwareentwicklung das Konzept der „Softwarearchitektur“ als eigenständige Disziplin der Softwaretechnik besondere Aufmerksamkeit (siehe z.B. [Arc97, GS94]). Die Architektur eines Softwaresystems organisiert die Komponenten einer Anwendung und legt insbesondere fest, wie die Verantwortlichkeiten verteilt sind. Diese fundamentalen Vorgaben wirken sich oft deutlich auf unterschiedliche Qualitätskriterien eines Softwaresystems aus, wie beispielsweise die Systemleistung aber auch die Erweiterbarkeit. Zudem vermittelt die Architektur eines Systems ein Verständnis über diverse Konzepte der Anwendung und ihr Zusammenwirken. Aus der entsprechenden Dokumentation kann ein Entwickler herauslesen, auf welche Art und Weise die Anwendung funktioniert und wie sie geeignet erweitert werden kann.

In diesem Zusammenhang erklärt sich das große Interesse an Frameworks, mit denen basierend auf einer bewährten Softwarearchitektur einem Entwickler eine „halbfertige“ Anwendung angeboten wird. Diese kann in wenigen, einfachen Schritten an konkrete Gegebenheiten anpasst werden und somit in kurzer Zeit zu einem lauffähigen Anwendungssystem führen. Erfahrungen mit Frameworks zeigen jedoch, daß diesem Wunsch nach geringem Anpassungsaufwand mit wachsender Größe des Frameworks nur schwerlich entsprochen werden kann [FSJ99, Mat00]. Je mehr Funktionalität in das Framework

eingebaut wird und je mehr Anwendungsbereiche mit diesem Framework abgedeckt werden sollen, desto rasanter steigt die Komplexität und somit der Aufwand für Einarbeitung und Anpassung beim Anwender des Frameworks.

Ein prominentes Beispiel für ein hochkomplexes Framework ist IBMs *San Francisco* [SAN], das für die Domäne der betrieblichen Informationssysteme entwickelt wurde. Potentiellen Anwendern werden umfassende Schulungen und eine Fülle themenbezogener Bücher angeboten, um den ganz beträchtlichen Lernaufwand auf ein Minimum zu reduzieren. Aber auch viele andere Frameworks zeigen das Phänomen eines kontinuierlichen Wachstums, da der Funktionsumfang laufend erweitert wird und die Implementierung zu monolithisch aufgebaut ist, um flexible Erweiterungen zu ermöglichen. Die Situation verschlimmert sich zudem durch die Tatsache, daß Frameworkentwickler häufig Funktionalitäten einbinden müssen, die nicht direkter Bestandteil ihres eigentlichen Fachwissens sind. Ein im Steuerrecht versierter Entwickler ist beispielsweise zwar geeignet für die Entwicklung eines Frameworks für die Buchhaltungsdomäne, gleichzeitig aber vielleicht nicht Profi hinsichtlich benötigter Persistenzmechanismen.

Aus diesen Erfahrungen heraus entsteht der Wunsch, ein Anwendungssystem als Ergebnis einer Komposition und Instantiierung unterschiedlicher Frameworkmodule aufzufassen, wobei jedes Frameworkmodul eine bestimmte Funktionalität (oder eine „Sicht“) der Anwendung adressiert und repräsentiert. Auf diese Weise wird die Komplexität reduziert und die Handhabung vereinfacht. Die einzelnen Frameworkmodule — im weiteren „Komponentenframeworks“ genannt[1] — können individuell entwickelt, verstanden und wiederverwendet werden. In den folgenden Kapiteln zeigen wir insbesondere Komponentenframeworks als ergänzendes Modulkonzept zu dem der Komponente, das mit diesem auf ideale Weise harmoniert.

1.1 Lösungsansatz

Im Rahmen dieser Dissertation wird ein zweistufiger Ansatz gewählt, um der im letzten Abschnitt beschriebenen Problematik starrer, monolithischer Frameworks zu begegnen und der Softwareentwicklung mit Komponentenframeworks zu einer breiteren Anwendung zu verhelfen. Auf der einen Seite wird ein sehr flexibles Konzept *integrierbarer Komponentenframeworks* entwickelt und ihre Verwendung anhand geeigneter Beispiele

[1] Ist durch den Zusammenhang offensichtlich, daß es um Komponentenframeworks geht, so werden wir im folgenden auch vereinfachend von „Frameworks“ sprechen.

demonstriert. Auf der anderen Seite verhilft eine methodische Fundierung bei der Beherrschung der einhergehenden Komplexität und steigert somit Verständnis und Anwendbarkeit. Im folgenden werden diese beiden Aspekte detailliert vorgestellt und diskutiert.

1.1.1 Frameworkintegration

Die Zerlegung eines Frameworks in einzelne Module erscheint nur unter der Voraussetzung sinnvoll, daß diese Module unabhängig voneinander entwickelt und wiederverwendet werden können. Je eigenständiger solche spezifischen Komponentenframeworks jedoch ausgerichtet sind, umso aufwendiger wird sich eine Integration mit anderen Komponentenframeworks zu einem späteren Zeitpunkt gestalten. So wie die Wiederverwendung von Komponenten heutzutage oft mit der Entwicklung von Adaptoren zur Anpassung der Schnittstelle verbunden ist, kann die Integration mit einem erheblichen Aufwand verbunden sein, sofern die einzelnen Komponentenframeworks nicht entsprechend miteinander harmonieren.

Ein wichtiger Teil dieser Arbeit beschäftigt sich daher mit der passenden Ausgestaltung von Komponentenframeworks, so daß sie unter vertretbarem Aufwand zu einem Gesamtframework zusammengefügt und letztlich geeignet instantiiert werden können. Ein bedeutender Bestandteil ist die durch ein Komponentenframework definierte Sicht auf ein Softwaresystem, die wir mit Hilfe eines rollenbasierten Modells repräsentieren (vgl. [RWL96]). Dieser Modellierung kommt in diesem Ansatz eine zentrale Bedeutung zu, da sie durchgängig den Entwicklungsprozeß eines Komponentenframeworks begleitet:

- Als Analysedokument repräsentiert ein Modell das zugrundeliegende Verständnis des Anwendungsbereichs und enthält über den Begriff der Rolle eine geeignete Abstraktion der jeweiligen Konzepte und Zusammenhänge.
- Die Integration unterschiedlicher Komponentenframeworks erfolgt anhand gemeinsamer Entitäten der beiden Modelle. Insbesondere kann (in den meisten Fällen) die Komposition von Komponentenframeworks auf die Komposition von Rollen zurückgeführt werden.
- Die Instantiierung eines Komponentenframeworks erfolgt über die Einbindung konkreter Komponenten, die unmittelbar den modellierten Entitäten des Anwendungsbereichs entsprechen.

Das Modell kann somit als eine besondere Art „Schnittstelle“ des Komponentenframeworks aufgefaßt werden, über die sowohl die Integration als auch die Instantiierung vorgenommen werden. Zudem argumentieren wir, daß nur die explizite Vorbereitung eines Komponentenframeworks auf eine spätere Integration zu einem System von Komponentenframeworks führt, die mit vertretbarem Aufwand zu Anwendungen zusammengefügt werden können. Eine zentrale Modellierung des Anwendungsbereichs erscheint hierbei ideal, um einzelne Komponentenframeworks aufeinander abzustimmen und eine homogene Anwendung zu erzielen.

Oft sind die Ansätze der komponentenbasierten Softwareentwicklung von der Idee begleitet, daß ein Entwickler fertige Komponenten über geeignete Komponentenmärkte einkauft, mit selbstentwickelten Komponenten ergänzt und schließlich alle diese Bestandteile — beispielsweise mit Hilfe geeigneter graphischer Werkzeuge — zu einer kompletten, lauffähigen Anwendung verschaltet. Demgegenüber sehen wir in dieser Arbeit Komponentenframeworks als die wiederverwendbaren Module eines Entwicklungsprozesses an, die — ausreichend dokumentiert — auch gehandelt werden können. Komponenten selbst reflektieren deutlich die Entitäten des Geschäftsmodells und werden eher vom Anwender eines Komponentenframeworks entwickelt, um dieses zu instantiieren. In Abbildung 1.1 ist eine Reihe von Komponentenframeworks schematisch dargestellt, die eine Menge von Komponenten strukturiert und umgekehrt organisieren die Komponenten das Zusammenspiel der unterschiedlichen Komponentenframeworks.

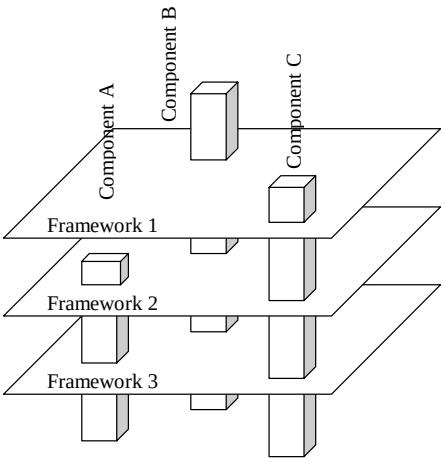


Abbildung 1.1: Integration von Komponentenframeworks

Zusammenfassend sehen wir die Vorteile eines solchen Ansatzes in den folgenden Punkten:

- Die Konzentration einzelner Komponentenframeworks auf jeweils ganz bestimmte Verantwortlichkeiten macht Frameworks zu überschaubaren, verständlichen und somit handhabbaren Einheiten.
- Der Anwender ist nicht länger durch den Einsatz eines Frameworks an einen bestimmten Hersteller gebunden.
- Die gesteigerte Flexibilität erlaubt auch das nachträgliche Hinzufügen oder Entfernen einzelner Frameworks.
- Die Architektur, die ein Framework dem System aufprägt, ist erheblich besser auf die jeweilige Funktionalität abgestimmt.

Der Nachteil dieses Ansatzes ergibt sich im Vergleich mit gängigen Formen der Softwareentwicklung: So ist es Praxis, in den frühen Phasen des Entwicklungsprozesses unterschiedliche Sichten auf ein System zu definieren, um durch geeignete Abstraktionen die Komplexität zu reduzieren und somit beherrschbar zu gestalten. Solche Sichten werden meistens mit der Hilfe graphischer Beschreibungstechniken definiert und in Form eigenständiger Dokumente festgehalten. Diese Sichten werden gleichzeitig weiterentwickelt und durch eine Reihe von Abhängigkeiten beeinflussen Designentscheidungen in einer Sicht die Situation in anderen Sichten. Es findet somit eine koordinierte Entwicklung statt, die zu den festgelegten Meilensteinen und am Ende der Entwicklung in einem konsistenten System resultiert.

In einem Szenario, in dem eine Reihe von Komponentenframeworks zum Einsatz gelangen, findet die Weiterentwicklung der einzelnen Sichten unabhängig von anderen Sichten statt. Eine Schwierigkeit besteht somit in der Formulierung von Abhängigkeiten des Frameworks zu dem Rest des Systems (dem „Kontext“), das zum Zeitpunkt der Frameworkentwicklung meist noch nicht bekannt ist. Zumindest spricht es für die Qualität eines Frameworks, wenn die Anzahl und Komplexität solcher Abhängigkeiten minimal sind. In diesem Zusammenhang ist die Betrachtung existierender Systeme interessant und die Frage, wie geeignete Sichten extrahiert und in Form von Frameworks festgehalten werden können, so daß möglichst wenig Abhängigkeiten auftreten.

1.1.2 Methodik

Der deutlich größeren Flexibilität beim Einsatz integrierbarer Komponentenframeworks (siehe letzter Abschnitt) steht eine vergleichsweise hohe Komplexität bei der Erstellung

und Handhabung dieser Frameworks gegenüber. Dies erscheint besonders schwerwiegend, da selbst der Einsatz „traditioneller“, objektorientierter Frameworks häufig an einem Mangel methodischer Unterstützung scheitert [MN96, SBF96]. Ist eine ungefähr zwanzig Jahre alte Disziplin noch nicht ausgereift in ihrer Anwendung, so kann die Einführung neuer Konzepte nur im Zusammenhang mit Überlegungen zu ihrer effektiven Verwendung sinnvoll sein.

Ein wichtiges Ziel der Arbeit besteht daher in der Konzeption einer Methodik für die Softwareentwicklung mit Komponentenframeworks. Damit werden sich vor allem die folgenden Fragestellungen beantworten lassen:

- Auf welche Weise können Komponentenframeworks spezifiziert und dokumentiert werden? Wie werden die Erweiterungsmöglichkeiten erfaßt? Wie die Abhängigkeiten zu anderen Frameworks?
- Was macht die Qualität eines Frameworks aus? Wie können Komponentenframeworks aus bestehenden Systemen so extrahiert werden, daß sich ihre Wiederverwendbarkeit maximiert?
- Auf welche Weise werden verschiedene Frameworks zu einem lauffähigen System zusammengestellt? Unter welchen Umständen lassen sich auf solche Weise integrierte Systeme nachträglich noch verändern?

Um diese vielfältigen Fragestellungen beantworten zu können, wird eine geeignete Methodik vorgestellt, die auf den Ergebnissen unserer Arbeiten im Forschungsprojekt FORSOFT A1 basiert. Hierbei werden graphische Beschreibungstechniken ebenso konzipiert wie ein flexibles Vorgehensmodell basierend auf sogenannten „Prozeßmustern“. Insbesondere die Komposition von Komponentenframeworks auf der Ebene von Analyse, Design und Realisierung wird auf diese Weise handhabbar.

1.2 Ergebnisse der Arbeit

Die wichtigsten Ergebnisse dieser Arbeit fassen wir wie folgt zusammen:

1. Komponentenframeworks werden als ein geeignetes Konzept vorgestellt, um nicht nur bewährtes Designwissen, sondern auch zugehörige Implementierungsfragmente wiederzuverwenden.

2. Wir führen ein Verständnis von Komponentenframeworks als Module der Softwareentwicklung ein, die flexibel kombiniert und integriert werden können.

3. Der vorgestellte Ansatz propagiert eine durchgängige Modellierung der Anwendung von der Analyse bis zur Implementierung, fördert somit das Verständnis der Zusammenhänge und ermöglicht die Integration unterschiedlicher Komponentenframeworks.

4. Das Konzept der Rolle wird als grundlegender Abstraktionsmechanismus eingeführt und in seiner Bedeutung klar von der Vorstellung über Schnittstellen abgegrenzt. Die Zusammenführung von Rollen wird auf die Komposition von Komponententypen zurückgeführt, die im Detail diskutiert und demonstriert wird.

5. Ein formales Sytemmodell präzisiert die eingeführten Konzepte und setzt sie zueinander in Beziehung. Insbesondere stellen wir ein nachrichtenbasiertes Ablaufmodell für komponentenbasierte Anwendungssysteme vor.

6. Die technische Realisierung von Komponentenframeworks wird am Beispiel des „CORBA Component Model“ vorgestellt und einhergehende Problemstellungen diskutiert.

7. Aufbauend auf einer bedarfsgetriebenen Erweiterung der UML wird ein Grundlage geschaffen für eine Methodik zur Softwareentwicklung mit Komponenten und Komponentenframeworks. Der eingesetzte Mechanismus über sogenannte „Prozeßmuster“ garantiert hohe Flexibilität und Effizienz.

1.3 Verwandte Arbeiten

Die vorliegende Arbeit stellt eine Fortsetzung der Forschungen zur komponentenbasierten Softwareentwicklung im Projekt A1 am Lehrstuhl von Prof. Broy dar. Sie wurde an vielen Stellen beeinflusst durch themenverwandte Arbeiten und Diskussionen mit Kollegen. Die folgende Übersicht bietet eine Auflistung verwandter Arbeiten, nach thematischen Schwerpunkten gruppiert:

Komponentenframeworks: Viele Überlegungen und Anregungen zu unterschiedlichen Themen im Umfeld von Komponenten und Komponentenframeworks finden sich in den Arbeiten von Szyperski (siehe z.B. [Szy00]). Die in dieser Arbeit vorgestellten

Konzepte zu Komponentenframeworks — gerade auch hinsichtlich der methodischen Handhabung — basieren auf den umfassenden Arbeiten zu objektorientierten Frameworks beispielsweise von Pree [Pre97a], Fayad [FSJ99] und Lewandowski [Lew98]. Eine Diskussion der Problemstellungen bei der modularen Gestaltung von Frameworks mit anschließender Integration findet sich in einigen aktuellen Arbeiten, wie beispielsweise [Mat00] oder [MBF00].

Separation of Concern: Seit geraumer Zeit bestehen Ansätze, wie beispielsweise das „subject-oriented programming“ [HO93], mit deren Hilfe die isolierte Betrachtung einzelner „Sichten“ auf eine Anwendung möglich ist. Doch erst mit der sehr pragmatischen Herangehensweise des „aspect-oriented programming“ (kurz: AOP) hat dieser Forschungsbereich wieder neuerliches Interesse erfahren. Mit AOP können mit recht einfachen Werkzeugen hauptsächlich technische Aspekte (z.B. ein Persistenzmechanismus), die orthogonal zu der eigentlichen Anwendungsstruktur organisiert sind, mit bestehendem Quelltext „verwoben“ werden (siehe [KLM+97]). Durch die implementierungsnahe Behandlung von Aspekten ist dieser Ansatz nur bedingt brauchbar für die Zielsetzung dieser Arbeit, in der wir bereits sehr frühzeitig im Entwicklungsprozeß unterschiedliche Aspekte identifizieren, die zudem nicht zwangsläufig auf technische Funktionalitäten hinauslaufen (siehe auch [CHOT99]). Dennoch sind viele Ideen aus diesen und vergleichbaren Arbeiten (z.B. [ML98]) in die hier vorgestellten Lösungsansätze eingeflossen.

Rollenmodelle: Die Zusammenführung konzeptueller Modelle basierend auf der Komposition einzelner Rollen ist ein essentieller Bestandteil der OORam-Methode [RWL96]. In der Dissertation von Riehle [Rie00] werden die Rollenmodelle aus der OORam-Methode erweitert und für die Spezifikation von Frameworks eingesetzt. In beiden Arbeiten werden Probleme der Verhaltensspezifikation und ihre Auswirkungen auf die Implementierung nur am Rande behandelt. Insbesondere wird davon ausgegangen, daß die zu integrierenden Rollenmodelle auf demselben Abstraktionsniveau angesiedelt sind.

Verhaltenskomposition: Die Zusammenführung von Rollen erfordert eine Komposition der jeweils zugeordneten Verhaltensspezifikationen. Für die nötigen Mechanismen hierzu finden sich theoretische Grundlagen z.B. in [LW94]. In diesem Zusammenhang sind ebenfalls Forschungsarbeiten zu dem Themenfeld „Feature Interaction“ interessant, die in den vergangenen Jahren vor allem in den Bereichen der Telekommunikation und Multimedia-Anwendungen entstanden. Ein Schwerpunkt liegt in der Behandlung von gegenseitigen Beeinflussungen zwischen unterschiedlichen Rollen, den „features“ eines Objekts. In den Arbeiten von Prehofer (z.B. [Pre97b]) wird gezeigt, wie durch die Einführung sogenannter „lifter“ Abhängigkeiten zwischen Features spezifiziert und bei der Kombination von Features

eingesetzt werden können. Der vorgeschlagene Delegationsmechanismus ist ähnlich zu dem hier verfolgten Ansatz mit Rollenkomponenten, wobei in dieser Arbeit „Feature Interaction“ über spezielle Integrationsschnittstellen erfolgt.

Zentrales Fachmodell: Die Erstellung eines zentralen Modells für die Zusammenführung unterschiedlicher Sichten eines Entwicklungsprozesses wurde weitgehend von der FOCUS-Methode [BS01] beeinflusst. Doch auch im Umfeld klassischer Ingenieursdisziplinen trifft dieser Ansatz auf wachsendes Interesse in Form zentraler Produktmodelle mit ganz ähnlichen Problemstellungen (vgl. z.B. [ART]). Die Idee eines gemeinsamen Fachmodells zur Synchronisation unterschiedlicher Sichten repräsentiert durch entsprechende Komponentenframeworks ist neu.

Softwarearchitektur: Gerade in theoretischen Arbeiten zu Softwarearchitekturen wurde bereits frühzeitig die Notwendigkeit erkannt, die Behandlung von Beziehungen zwischen den Komponenten einer Anwendung von diesen zu entkoppeln und als eigenständige Bestandteile des Architekturentwurfs zu spezifizieren (vgl. *Connectors* in [Sha94]). Während diese Erkenntnisse des Designs im Verlauf der Realisierung jedoch wieder in die Verantwortungsbereiche einzelner Komponenten übergehen, erfolgt in dieser Arbeit eine direkte Abbildung auf Komponentenframeworks, die zu Komponenten gleichberechtigte Instanzen des Anwendungssystems darstellen. Im Vergleich zu den Konnektoren nach Shaw erlauben wir zudem fachliche Funktionalität im Framework, die über die reine Verwaltung von Beziehungen hinausgeht.

Formales Systemmodell: Die Abstützung unterschiedlicher Beschreibungstechniken auf eine formale, semantische Basis entspricht ebenfalls der Tradition von Arbeiten im Umfeld von FOCUS (siehe z.B. [Rum96]). Der mathematische Formalismus wird im Rahmen dieser Arbeit jedoch hauptsächlich für ein solides Verständnis der zugrundeliegenden Konzepte und Ideen verwendet (beispielsweise bei der Modellintegration). Die logische Ergänzung der Konzepte „Schnittstelle“ und „Rolle“ ist neu — die flexible Kombination von Schnittstellen wird auch im Umfeld von Rapide [LKA+95] untersucht.

Methoden der Softwareentwicklung: Die Ideen und Erwartungen der komponentenbasierten Softwareentwicklung erfordern eine intensive Auseinandersetzung mit teils sehr grundlegenden methodischen Fragestellungen [BRSV00, BRSV99]. Neben den Ergebnissen aus dem Forschungsprojekt A1 flossen viele Anregungen aus ähnlichen Ansätzen ein (z.B. [AF98, DW98]). Aber auch Prinzipien allgemeinerer Methoden, wie beispielsweise vom Rational Unified Process (kurz: RUP) [Kru99] finden sich in der ein oder anderen Form wieder.

1.4 Aufbau der Arbeit

Der Aufbau der Arbeit untergliedert sich wie folgt: im nachfolgenden Kapitel 2 erfolgt eine Diskussion über relevante, grundlegende Konzepte und die damit verbundenen Problemstellungen. Hier finden sich überblickartige Einführungen zu Komponenten, Frameworks, Softwarearchitekturen sowie zu einem Rollenbegriff, wie er sich in den vergangenen Jahren in unterschiedlichen Arbeiten herauskristallisiert hat.

Die weiteren Kapitel behandeln die einzelnen Bausteine einer übergreifenden Methodik für die Softwareentwicklung mit Komponentenframeworks. Abbildung 1.2 zeigt diese Bestandteile und wie sie aufeinander aufbauen.

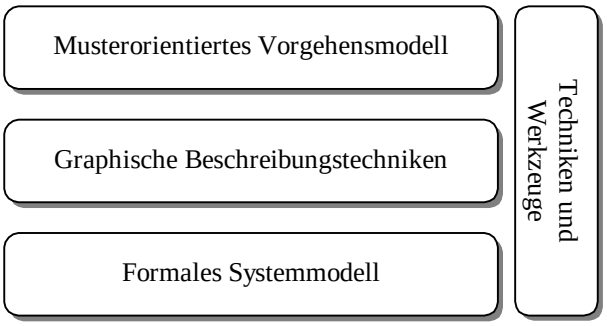


Abbildung 1.2: Bausteine der komponentenbasierten Softwareentwicklung

Kapitel 3 beschäftigt sich mit der Entwicklung eines formalen Systemmodells, anhand dessen im weiteren grundlegende Konzepte definiert werden. So wird die rollenbasierte Modellierung eines Anwendungsbereichs motiviert und vorgestellt. Anhand dieser Ergebnisse können bereits unterschiedliche Problemstellungen der Modellintegration diskutiert und geeignete Lösungsansätze entwickelt werden.

Die praktische Umsetzung dieser Ergebnisse auf eine konkrete Komponentenplattform wird in Kapitel 4 erörtert. Dazu gehören Vergleiche aktueller Komponententechnologien und eine Abschätzung der Praxistauglichkeit der Konzepte dieser Arbeit.

Eine geeignete Erweiterung der Unified Modeling Language (kurz: UML) für die Dokumentation von Komponentenframeworks wird in Kapitel 5 entworfen. Auf eine präzise semantische Fundierung über das formale Systemmodell wird im Gegensatz zu Arbeiten wie [Rum96] nicht genauer eingegangen. Stattdessen werden die gängigen Erweiterungsmechanismen der UML verwendet, um essentielle Konzepte wie Rolle und Framework adäquat repräsentieren zu können.

Basierend auf dem Mechanismus der Prozeßmuster wird anschließend in Kapitel 6 ein flexibles Vorgehensmodell motiviert und skizziert. Einzelne konkrete Prozeßmuster zeigen sinnvolle Herangehensweisen an die Entwicklung und den Einsatz wiederverwendbarer Komponentenframeworks.

Abschliessend fassen wir in Kapitel 7 die Ergebnisse dieser Arbeit zusammen und zeigen Vor- und Nachteile des Ansatzes zur Integration modularer Komponentenframeworks.

Kapitel 2

Grundlagen

Es ist eine der ältesten Wunschvorstellungen im Bereich der Softwareentwicklung, die Erstellung eines Softwaresytems ebenso prägnant und nachvollziehbar zu gestalten wie den Bau eines Hauses oder die Montage eines Autos. Diese Disziplinen sind jedoch weitaus älter und profitieren daher von einem übergreifenden und besseren Verständnis und Erfahrungen hinsichtlich der einzelnen Bestandteile und der Art ihres Zusammenbaus. Obwohl viele Programmiersprachen bereits frühzeitig mit einem Modulkonzept ausgestattet waren (siehe z.B. Übersicht in [Sih95]), besteht bei Entwicklern häufig eine Unsicherheit darüber, wie Systeme „geschnitten“ werden müssen, damit ihre Teile in anderen Systemen sinnvoll wiederverwendet werden können. Daraus resultieren weitere Probleme, wie beispielsweise bei der korrekten Spezifikation solcher Systemteile, ihrer konkreten Anwendung sowie ihrer wirtschaftlichen Verwertbarkeit.

In den letzten Jahren entstand unter dem Begriff „Componentware“ ein Ansatz, der eine Lösung vieler dieser Problemstellungen versprach. Der etwas allgemeine Begriff einer Komponente führt hierbei oft zu Verwirrungen, denn „*everything that can be composed into a composite is a component*“ [Szy00]. An der Fülle unterschiedlicher Begriffsdefinitionen für Komponenten (siehe z.B. [Szy97, Gri98]) läßt sich erkennen, daß auch heute noch kein gemeinsames Verständnis der Basiskonzepte für die komponentenbasierte Softwareentwicklung existiert.

Berechtigterweise stellt sich die Frage nach dem Unterschied zwischen Componentware und objektorientierter Programmierung, da hinter beiden Ansätzen zu großen Teilen dieselbe Motivation steht. Tatsächlich basieren beide Paradigmen auf demselben klassischen Handwerkszeug des Informatikers: *Abstraktion* und der Anwendung des Prinzips *Teile und Herrsche*. Ganz konkrete Unterschiede werden Komponenten oft etwas willkürlich

attributiert — so beispielsweise das Fehlen eines Instanzenkonzepts [Szy00] oder ein betonter Verteilungscharakter [Gri98]. Letztlich hat die Suche nach einem einheitlichen Komponentenbegriff jedoch eine wichtige Diskussion um neue Lösungsansätze für alte Probleme entfacht: Wie können Softwarefragmente erfolgreich eingesetzt werden, um moderne Softwaresysteme mit ihrer stetig wachsenden Komplexität effizient realisieren und warten zu können?

2.1 Komponenten

Vielversprechender als meist unpräzise Begriffsdefinitionen erscheint für das Verständnis des Konzepts „Komponente“ eine Auflistung der Vorteile, die sich durch den Einsatz von Komponenten ergeben sollen. Diese Ziele eines komponentenbasierten Entwicklungsparadigmas sind nicht neu — von der Objektorientierung hatte man sich ganz ähnliche Vorteile erhofft. Und auch die Konzepte, mit denen die Ziele erreicht werden sollen (z.B. *information hiding*) sind bekannt und ihre Bedeutung für die erfolgreiche Wiederverwendung von Software bereits im Bewußtsein der Entwickler. Jedoch hat das Paradigma der objektorientierten Entwicklung die Einhaltung dieser Konzepte nicht in nötigen Maßen zugesichert. So wurde mit dieser Art der Softwareentwicklung nie der Grad an Wiederverwendung erreicht, der nun mit dem „neuen“ Konzept der Softwarekomponente endlich verwirklicht werden soll.

Die folgende Aufzählung ist keineswegs vollständig, reflektiert jedoch eine Vorstellung von Komponenten, wie sie dieser Arbeit zugrundeliegt (siehe auch [Szy97]).

Abstraktion: Durch die Abstraktion unnötiger Details eines Systems kann die Komplexität der Zusammenhänge reduziert und damit das Verständnis gesteigert werden. Die Bildung von Modellen erfordert die Anwendung der Abstraktion und in vielen Fällen entstehen unterschiedliche Ebenen mit mehr oder weniger detaillierter Information, wobei die unterschiedlichen Ebenen sich aufeinander berufen können.

Komponenten bieten einen einfachen Mechanismus zur Abstraktion, indem sie ihren internen Zustand und ihr Verhalten nach „außen“ abkapseln. Während Objekte noch teilweise über Attribute ihren Zustand offenbaren, ist die konkrete Realisierung einer Komponente und damit interne Details vollständig verborgen. Die Spezifikation der Schnittstelle einer Komponente stellt hier eine abstrakte Repräsentation der Komponente dar.

Separation of concern: Systeme, deren Module über ein enges Beziehungsgeflecht miteinander verbunden sind, bieten wenig Möglichkeit zur Wiederverwendung. Von

Komponenten wird dagegen erwartet, daß sie losgelöst von späteren Anwendungssystemen entwickelt, erstellt und getestet werden können. Die Annahmen, die eine Komponente über ihr mögliches Umfeld trifft (und umgekehrt) müssen in einem überschaubaren Rahmen bleiben, um die spätere Integration in ein System nicht durch unnötige Komplexität zu gefährden.

Dieses Ziel ist einfacher zu erreichen, wenn sich für die einzelnen Komponenten eines Systems klare Verantwortlichkeiten finden und zuordnen lassen. Beispiele für erfolgreiche Komponenten finden sich daher auch dort, wo dezidierte Dienste einer Anwendung angeboten werden können: Datenbank, Email-Postfach und ähnliche. Nicht zuletzt hat diese Rollenverteilung in Diensterbringer und Dienstnutzer die technische Infrastrukur im Komponentenumfeld geprägt (z.B. bei CORBA [OH98]).

Flexibilität: Moderne Softwaresysteme erreichen in vielen Fällen eine ausgesprochen hohe Komplexität und Langlebigkeit. Beides führt dazu, daß eine Anwendung — neben der Anpaßbarkeit an konkrete Gegebenheiten des Anwendungsbereichs — auch nach ihrer Fertigstellung veränderbar sein muß, um neuen Anforderungen zu begegnen. Diese Flexibilität kann durch komponentenbasierte Anwendungen einfach erreicht werden: heutige Komponententechologien erlauben das dynamische Hinzufügen oder Ablösen von Komponenten in einem System. Komponenten sind somit die „units of change“ — alle Veränderungen am System müssen sich auf den Austausch von Komponenten reduzieren lassen.

Je höher jedoch die Flexibilität eines Systems zur Laufzeit, desto mehr Konsistenzüberprüfungen müssen vom Zeitpunkt der Übersetzung einer Komponente zum Zeitpunkt der Ausführung verlagert werden. Compiler als Werkzeug zur Überprüfung der Systemkorrektheit müssen daher ergänzt werden durch Konsistenzchecker im laufenden System.

Handelbarkeit: Damit sich die Wiederverwendung von Design- und Implementierungsfragmenten auch für den Ersteller wirtschaftlich lohnt, muß ein Markt entstehen, der Angebot und Nachfrage zusammenführt und einen Handel ermöglicht. Komponenten eignen sich aufgrund ihrer Abgeschlossenheit ausgesprochen gut als handelbare Güter. Durch verständlich und eindeutig spezifizierte Schnittstellen läßt sich die Funktionalität einer Komponente kommunizieren und ihre Rolle im zu erstellenden System bewerten.

In objektorientierten Anwendungen sind einzelne Klassen oft zu klein, um sie aus dem System zu extrahieren und in einem anderen Kontext wiederzuverwenden. Aus diesem Grund haben sich hier ganze Objektverbünde, die objektorientierten Frameworks als wiederverwendbare Module etabliert. Es gibt jedoch eine Reihe von

Gründen, weshalb Frameworks nicht so erfolgreich gehandelt werden, wie es heutzutage bereits bei Komponenten zu beobachten ist (siehe Abschnitt 2.3,[Pre97a]). Komponentenmärkte wie beispielsweise *ComponentSource* [Com] reflektieren den steigenden Bedarf eines wirtschaftlichen Handels mit Komponenten.

Standards: Die Konzepte des Themenfelds Componentware betonen die Komposition unabhängig entwickelter Systembestandteile, insbesondere im Hinblick auf die Kommunikation der einzelnen Komponenten eines Systems untereinander. Komponenten müssen sich untereinander verständigen können und im Rahmen bestimmter Aufgaben miteinander kooperieren. Weiterhin muß jede Komponente auch bestimmten Anforderungen genügen, damit sie vom Systemmanagement verwaltet werden kann. Nur so ist es möglich, Komponenten flexibel einem System hinzuzufügen oder sie nach Bedarf wieder zu entnehmen.

Die Lösung für diese (und viele andere) Probleme findet sich in der Standardisierung z.B. von Technologien, Sprachen und Schnittstellen. Ein Standard für die Kommunikation mit einer Datenbankkomponente (z.B. ODBC) ermöglicht das problemlose Zusammenspiel bei der Realisierung eines Persistenzmechanismus. Die Berufung auf einen gemeinsamen Komponentenstandard (beispielsweise EJB) führt zur einfachen Verwaltung der Systembestandteile und ihres Umfeldes in einem EJB-Container.

Nachdem die Fülle der Erwartungen an die komponentenbasierte Entwicklung anhand obiger Zielsetzungen dargestellt wurde, kann die eigentliche Begriffsdefinition nun etwas knapper ausfallen und nur die wichtigsten Kernaspekte betonen (aus [Szy97]):

> *„A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties“.*

Entscheidend ist hierbei die Tatsache, daß eine Schnittstelle einen „Vertrag“ über die Art und Weise darstellt, wie eine Komponente genutzt werden soll (vgl. [HHG90]). Über solche Verträge ist zudem die Umgebung einer Komponente festgelegt, die eine Voraussetzung für die korrekte Funktionalität der Komponente darstellt. Die in der Begriffsdefinition dargestellte Aussicht auf eine neue Art der Anwendungsentwicklung durch das reine Anpassen und Zusammenfügen geeigneter Softwarekomponenten („*programming in the large*“) wurde zur zentralen Vision hinter dem Begriff Componentware [McI68] und tatsächlich existieren bereits einige Werkzeuge, die eine einfache Komposition von Komponenten über eine graphische Oberfläche erlauben (z.B. Component-X Studio [CXS]).

Ist die Funktion einer jeden Komponente im Verbund ebenso eindeutig wie die Aufgabe einer Tür oder eines Fensters, so nähert sich tatsächlich die Softwareerstellung den klassischen Disziplinen wie Architektur oder Elektrotechnik an. Aus diesem Grund erscheint es naheliegend, Komponenten anhand der Entitäten und Konzepte des Anwendungsbereichs zu modellieren. Beispiele für solche „Business-Komponenten“ wären ein „Kunde“, eine konkretes „Produkt“ oder auch ein „Lagerhaus“. Business-Komponenten, wie sie beispielsweise von der OMG definiert werden, sind eine Antwort auf die Frage, wie man am besten ein Anwendungssystem in einzelne Bestandteile zerlegt, um die Wiederverwendung der einzelnen Teile zu maximieren. Findet sich in einem fremden System eine Business-Komponente „Kunde“, so assoziiert fast jeder ein grundlegendes Verständnis darüber, was die Aufgabe dieser Komponente sein könnte. So wird in [Gri98] die Voraussetzung für die erfolgreiche und umfassende Wiederverwendung einer Komponente darin gesehen, daß sie in ihrer Funktion allgemein bekannt und akzeptiert ist.

2.2 Softwarearchitektur

Der Begriff „Softwarearchitektur“ wurde Anfang der 90er Jahre modern, als Softwaresysteme immer umfassender und komplexer wurden [GS94]. Die Softwarearchitektur eines Systems beschreibt seine Struktur auf einer relativ abstrakten Ebene und vermittelt ein Verständnis über die Art der beteiligten Komponenten und die Menge der Regeln, die ihr Zusammenspiel festlegen. Eine prägnante Definition findet sich bei [BCK98]:

> „*The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them.*“

Obwohl diese Definition auf Softwarekomponenten aufbaut, ist das Konzept Softwarearchitektur nicht zwangsläufig an Komponenten gebunden. Im Rahmen dieser Arbeit ergänzt sich jedoch die Softwarearchitektur ideal mit den Ideen der Componentware und aus diesem Grund findet sich nur an wenigen Stellen der etwas präzisere Begriff „komponentenbasierte Softwarearchitektur“.

Aus der aufgeführten Definition ergibt sich die Softwarearchitektur als eine abstrakte, strukturelle Modellierung eines Systems. Dem gängigen Verständnis entsprechend werden Menge und Art der beteiligten Komponenten festgelegt, ihre Beziehungen untereinander und auf welche Weise sie kommunizieren. Daneben gibt es in vielen Ansätzen weitergehende Informationen, wie beispielsweise bestimmte Einschränkungen, konkrete

Anforderungen oder Regeln der Zusammenstellung und des Zusammenspiels der Komponenten. Daneben werden oft auch Informationen über dynamische Veränderungen am System in Rahmen einer Softwarearchitektur spezifiziert. Dabei handelt es sich um Fragen der Modifikation der Systemkonfiguration, also beispielsweise unter welchen Bedingungen neue Komponenten in ein bestehendes System eingeführt werden. Gerade im Umfeld langlebiger Systeme sind Antworten auf solche Fragen entscheidend, da sie es erlauben, die Evolution einer Anwendung zu beschreiben.

In den vergangenen Jahren hat sich die Architektur eines Softwaresystems als ein zentraler Faktor für den Erfolg eines Projektes herausgestellt [SG95]. Obwohl oft nur in Form einfacher Graphiken skizziert, vermittelt die Softwarearchitektur ein konzeptuelles Verständnis über die Organisation der Komponenten eines Systems. In manchen Fällen reichen schon knappe Angaben wie beispielsweise „web-basiertes Informationssystem mit Drei-Schichten-Architektur“, um einen ersten Eindruck der Richtlinien zu bekommen, nach denen die Struktur des Systems vorgegeben ist. Diese Richtlinien beeinflussen zudem mehr oder weniger stark bestimmte Qualitätseigenschaften des Systems, wie beispielsweise Leistung, Skalierbarkeit oder Wartbarkeit [Bos00].

Letztlich ist die Dokumentation der Softwarearchitektur ein probates Mittel für den Austausch zwischen Entwicklern und Kunden bereits zu einem frühen Zeitpunkt der Systementwicklung. Anhand dieses Grobdesigns kann bereits abgesehen werden, welche Eigenschaften der Anwendung zugeordnet werden können und auf welche Weise bestimmte Anforderungen adressiert werden.

Eine Vielzahl von Forschungsarbeiten beschäftigt sich mit unterschiedlichen Fragestellungen der Softwarearchitektur. Wichtige Themen umfassen beispielsweise die akkurate Analyse und Spezifikation einer Architektur mit Hilfe sogenannter *architectural description languages* (ADLs) oder die Katalogisierung und Darstellung der Prinzipien bewährter Architekturen [SG95]. Bereits frühzeitig entstanden „Architekturstile“, die eine Familie von Systemen prägen und auszeichnen. Bewährte Architekturen sollen in weiteren Systemen wiederverwendet werden und innerhalb der Anwendungen derselben Familie ist zu erwarten, daß eine Wiederverwendung der Architektur aber auch von einzelnen Komponenten besonders einfach ist.

2.3 Frameworks

Die Motivation für den Einsatz von Frameworks bei der Softwareerstellung liegt in der Möglichkeit, in relativ kurzer Zeit mit entsprechend geringem Aufwand qualitativ hochwertige Anwendungen erstellen zu können. Das bedeutet eine Verkürzung des „time-to-market“ bei gesteigerter Produktivität. Unter einem Frameworks verstehen wir dabei eine „halbfertige“ Anwendung, die lediglich an wenigen Stellen an die aktuellen Gegebenheiten angepaßt werden muß, um eine lauffähige Anwendung zu erhalten. Dieser Ansatz ist gerade deshalb so interessant, da es auf diese Weise möglich ist, in einem System bestimmte, kritische Designentscheidungen festzuhalten während andere vorerst noch offen bleiben (vgl. [Szy97]).

In den 90er Jahren entstanden in Forschung und Industrie eine Reihe objektorientierter Frameworks für eine Vielzahl unterschiedlicher Anwendungsbereiche [FPR00]. Als Beispiele seien Frameworks genannt für graphische Benutzerschnittstellen (Java Swing [Swi]) und für Geschäftsanwendungen (IBM’s *San Francisco* [SAN]). Der Einsatz von Frameworks auf einer breiteren Basis krankte bereits zu dieser Zeit an oft mangelnder Qualität, was sich häufig in einem Ungleichgewicht zwischen Flexibilität und Wiederverwendbarkeit bemerkbar macht. Einer Idealvorstellung entsprechen jene Frameworks, die eine klar begrenzte Familie von Anwendungen umfassen und demzufolge nur wenige Möglichkeiten der Anpassung erforderlich machen. Aufgrund dieser gerade bewußt angestrebten Limitierung sind diese Frameworks aufgrund hoher Verständlichkeit einfach einzusetzen. Der entscheidende Nachteil liegt jedoch in der mangelhaften Erweiterbarkeit. So können Systeme, die auf solchen Frameworks aufbauen, nur aufwendig um neue Funktionalitäten erweitert werden, die von den Framework-Designern nicht bereits vorgesehen waren. Auf diese Weise entstanden Frameworks wie *San Francisco* [SAN], die eine breite Fülle an Erweiterungen vorsehen, um eine möglichst weit gefaßte Familie von Anwendungen abzudecken. Die damit zwangsläufig gestiegene Komplexität führt nun dazu, daß der Einsatz des Frameworks deutlich erschwert ist. Im Falle von *San Francisco* wird eine Fülle von Büchern und Lehrgängen angeboten, um die Handhabung dieses umfassenden Rahmenwerks zu vermitteln.

Betrachtet man die Motivation, die zu der Konzeption und Erstellung eines Frameworks führt, so findet sich meist ein ausgeprägtes Domänenwissen. Kenntnisse über die Organisation eines Systems, über die eingesetzten Konzepte und ihr Zusammenspiel sollen über den aktuellen Kontext hinaus in unterschiedlichen Szenarien nutzbar sein. Um dieses sehr spezifische Wissen in Form eines Frameworks festzuhalten, ist häufig die Auseinandersetzung mit verwandten Themen erforderlich, die nicht zwangsläufig im Bereich der „Kernkompetenzen“ des Entwicklers liegen. Diese Situation verschärft sich durch die Bestrebungen weitreichende Funktionalitäten in das Framework mit aufzunehmen.

Aus solchen und ähnlichen Überlegungen entstammt der Wunsch nach mehr Flexibilität bei der Erstellung und Verwendung von Frameworks. Ohne dem Anwender eines Frameworks „alle Tore zu öffnen“, soll es möglich sein, neue Funktionalitäten zu integrieren oder bestehende zu ersetzen. Auf diese Weise entstehen flexible Frameworks, die den sich ständig wandelnden Anforderungen moderner Software-Systeme Rechnung tragen. Gleichzeitig ist es erforderlich, die zwangsläufig gesteigerte Komplexität beherrschbar zu gestalten. So können die folgenden Probleme mit Framtworks identifiziert werden (nach Riehle [Rie00]): Einzelne Klassen erreichen oft eine unüberschaubare Größe und Komplexität und die entsprechenden Schnittstellen sind nicht mehr leicht verständlich. Der Fokus liegt entweder auf den beteiligten Klassen oder auf den Kollaborationen, wodurch der jeweils andere Aspekt vernachlässigt wird. Auch fehlen häufig Mechanismen, um die teils sehr komplexen Abläufe zu strukturieren und verständlich zu gestalten. Die Einführung von „hot-spots“ [Pre97a] ist nur einer von mehreren Ansätzen, um die Anwendung von Frameworks effizienter zu gestalten.

2.4 Rollen

Das Konzept der Rolle ist weit verbreitet beispielsweise bei der Modellierung von Geschäftsprozessen oder Unternehmensstrukturen. Mit einer Rolle wird dabei meistens eine intuitive Vorstellung über einen bestimmten Verantwortungsbereich einer Person (und damit über ihr Verhalten) verbunden sowie die Tatsache, daß einer Person durchaus mehrere Rollen zugeordnet werden können. Durch diese allgemein angenommene Eigenschaft bietet sich die Verwendung des Konzepts Rolle als allgemeiner Abstraktionsmechanismus für viele Bereiche der Modellierung an. Reenskaug beispielsweise sieht das Klassenkonzept aus der objektorientierten Analyse als nicht ausreichend an, um die unterschiedlichen Aspekte eines Objekts in geeigneter Form zu repräsentieren. Die von ihm mitentwickelte OORam-Methode [RWL96] nutzt daher ausgiebig das Konzept der Rolle für die Beschreibung von Objekten.

In den vergangenen Jahren entstand insbesondere in den unterschiedlichen Arbeitsgruppen der „Object Management Group“ der Wunsch nach einem präziseren Verständnis des Konzepts Rolle und seiner Auswirkungen auf die Modellierung. Ergebnisse dieser Bemühungen zeigen sich in einer Reihe von Positionspapieren (z.B. [GW00, Cum00]) sowie in einem verfeinerten Rollenbegriff in der aktuellen Spezifikation der Unified Modeling Language (siehe [UML00] und Kapitel 5).

Jede Definition des Konzepts Rolle basiert auf dem Begriff der Entität, der Rollen zugeordnet werden können. Unter einer Entität verstehen wir ein konkretes „Ding“ des

Anwendungsbereichs, das eindeutig identifizierbar und damit von anderen Entitäten abgrenzbar ist. Prinzipiell erscheint es naheliegend, nur solche Entitäten durch Rollen zu beschreiben, von denen anzunehmen ist, daß sie ein Verhalten aufweisen. In Kapitel 3 werden Rollen dagegen noch allgemeiner als bestimmte Eigenschaften einer Entität verwendet, ohne zwingend festzulegen, ob sich diese Eigenschaften in ihrem Verhalten oder ihrem Zustand niederschlagen. Eine diesem Verständnis entsprechende Definition des Begriffs Rolle findet sich in [Cum00]:

> „*A role is the state and behavior of an entity with respect to a particular context. A role does not exhibit all of the state or behavior of the entity, but the role will exhibit state and/or behavior that is relevant to its participation in the context. The context is generally associated with some endeavor, e.g. an enterprise, a community of interest, a service or a business transaction.*“

Demnach ist eine Rolle stets an ihren Kontext gebunden, wobei eine Entität in unterschiedlichen Kontexten dieselbe Rolle und in demselben Kontext unterschiedliche Rollen ausüben kann. Ein wichtiger Bestandteil der Diskussion um ein Rollenkonzept in der UML ist die Frage, ob Rollen als Typen einer Entität verstanden werden oder in Form beispielsweise von „Rollenobjekten“ (siehe [Cum00, Bäu98]) mit eigener Identität und Zustand versehen sind. Beide Sichtweisen schließen sich gegenseitig nicht aus, beeinflussen jedoch nachhaltig den Umgang mit dem Konzept Rolle, worauf wir in den folgenden Kapiteln noch genauer eingehen werden.

Der Vorteil von Rollen insbesondere hinsichtlich der Integration von unterschiedlichen Aspekten eines Anwendungssystems liegt in der Möglichkeit, nur relevante Eigenschaften von Komponenten zu beschreiben und damit in der Reduktion der Komplexität und der Steigerung der Skalierbarkeit der entstehenden Systeme.

Kapitel 3

Formale Modellierung

Allgemeine Probleme der Integration von Ergebnissen unabhängiger Entwicklungsprozesse ergeben sich aus der Zusammenführung unterschiedlicher Vorstellungen derselben Konzepte, die nicht selten teilweise widersprüchlicher Natur sind. Unabhängig davon, auf welcher Stufe der Entwicklung Artefakte angesiedelt sind, die miteinander kombiniert werden sollen, stets basieren sie auf dem konzeptuellen Verständnis der jeweiligen Entwickler. Selbst bei gemeinsamen Basistechnologien können die Differenzen in den Absichten und Zielen eine Integration erschweren oder oft sogar unmöglich machen. Es erscheint aus diesen Gründen naheliegend, eine gemeinsame Basis zu schaffen, über die unterschiedliche Vorstellungen miteinander in Einklang gebracht werden können. Dabei ist die Möglichkeit zur präzisen Formulierung der Konzepte entscheidend, um detaillierte Aussagen über Gemeinsamkeiten oder Unterschiede in den einzelnen Ansichten treffen zu können. In diesem Kapitel präsentieren wir aus dieser Motivation heraus ein einfaches, formales Metamodell, das die gemeinsame Basis für die Spezifikation und Zusammenführung unterschiedlicher Verständnisse darstellt.

Mit Hilfe der einhergehenden Konzepte kann das Verständnis über ein Anwendungssystem in Form einfacher Modelle präzisiert und spezifiziert werden. Gerade solche Modelle sind inzwischen beispielsweise im Rahmen der *Objektorientierten Analyse* (kurz: OOA) ein verbreitetes Instrument, dessen Nutzen jedoch häufig durch eine mangelnde Präzision der verwendeten Beschreibungstechniken begrenzt ist. In späteren Phasen werden solche Analysemodelle Schritt für Schritt verfeinert zu Implementierungsmodellen, wobei sich an dieser Stelle die Spezifikation von Protokollen (beispielsweise mit Hilfe der Interaktionsdiagramme der UML) bewährt hat, um das Zusammenspiel der einzelnen Objekte zu detaillieren.

Die Zusammenführung unterschiedlicher Verständnisse ist prinzipiell eine Integration der zugrundeliegenden Modelle. Hierbei treten eine ganze Reihe interessanter Fragestellungen auf: Einerseits stellt sich die Frage, an welchen Stellen es Überschneidungen zwischen den Modellen gibt, also wo tatsächlich dasselbe „Konzept“ in beiden Modellen repräsentiert ist. Wenn zwei Modelle denselben Ausschnitt beispielsweise der „realen Welt“ darstellen, so werden wir im weiteren auch von unterschiedlichen *Sichten* sprechen.

Während nun die traditionelle (objektorientierte) Softwareentwicklung auf das Zusammenspiel einzelner Objekte fokussiert, ist das in den nächsten Abschnitten vorgestelle Metamodell weitgehend auf das Verständnis des Zusammenspiels unterschiedlicher Modelle ausgerichtet. Wir erwarten und werden zeigen, daß im Normalfall die Modellintegration zusätzliche Konsistenzbedingungen und damit Querbezüge zwischen den beteiligten Modellen einführt, die sich im Implementierungsmodell in entsprechenden Interaktionen „zwischen“ den Modellen zeigen.

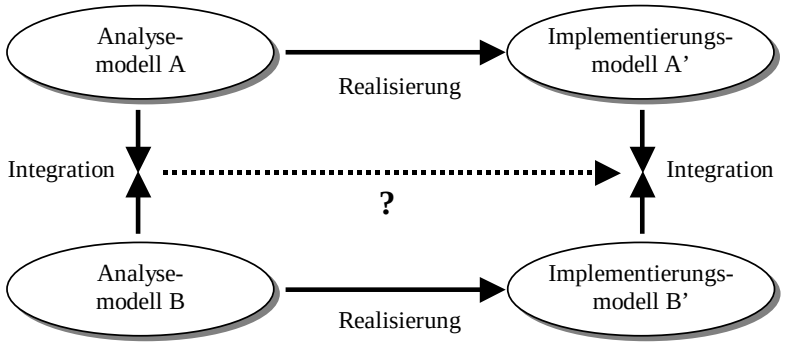


Abbildung 3.1: Modellintegration auf unterschiedlichen Entwicklungsstufen

Gerade die Frage nach dem geeigneten Detaillierungsgrad für die Integration von Modellen stellt einen Schwerpunkt dieses Kapitels dar. Beispielsweise sind einem Entwickler weitgehende Freiheiten gegeben, die Informationen aus einem Analysemodell in ein geeignetes Implementierungsmodell umzusetzen. Die damit einhergehende Problematik ist (vereinfacht) in Abbildung 3.1 dargestellt: Existieren für zwei Analysemodelle zugehörige Implementierungsmodelle, so wäre es — gerade hinsichtlich der Wiederverwendung erbrachter Entwicklungsleistungen — hilfreich zu wissen, wie sich die Integration auf der Ebene der Analysemodelle auf die Implementierungsmodelle auswirkt. Nicht zuletzt wäre es damit möglich, konkrete Bestandteile der Implementierung wiederzuverwenden und den Entwicklungsaufwand zu reduzieren.

In den nachfolgenden Abschnitten zeigen wir, auf welche Weise das Konzept der Komponentenframeworks diese Problemstellungen adressiert und im Besonderen die Wiederverwendung gleichfalls von Analyse- und Designwissen als auch von bestehenden Artefakten

der Implementierung zuläßt. Der zugrundeliegende Ansatz schafft eine enge Kopplung zwischen frühen und späten Phasen eines Entwicklungsprozesses, indem Elemente der Analysephase unmittelbar in die Realisierung übertragen werden.

Um sowohl das abstrakte Verständnis repräsentiert durch ein Analysemodell als auch die Wiederverwendung von konkreten Implementierungsartefakten zu berücksichtigen, ist das formale Metamodell in verschiedene Schichten unterteilt, die in festgelegten Beziehungen zueinander stehen (vgl. Abbildung 3.2).

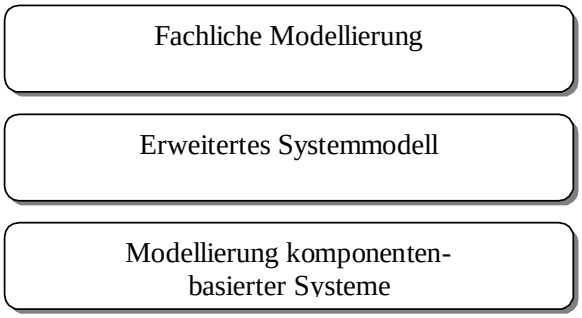


Abbildung 3.2: Aufbau der Modellierung

In der obersten Schicht werden Konzepte eingeführt, die die abstrakte Modellierung eines Anwendungsbereichs ermöglichen. Dieser wird dabei in einzelne Entitäten zerlegt, die über unterschiedliche „fachliche“ Beziehungen verknüpft sind. Weiterhin stehen Mechanismen zur Verfügung, die die Zusammenführung mehrerer fachlicher Modelle ermöglichen. Bereits auf dieser abstrakten Ebenen können die unterschiedlichen Problemstellungen der Modellintegration untersucht und geeignete Lösungsmöglichkeiten aufgezeigt werden.

Die unterste Schicht stellt ein allgemeines Sytemmodell für komponentenbasierte Anwendungen zur Verfügung. Hierbei werden elementare Konzepte wie beispielsweise „Komponente“, „Schnittstelle“ und „Verbindung“ diskutiert und präzisiert. Die Ergebnisse dieses Abschnitts basieren auf früheren Arbeiten zur komponentenbasierten Softwareentwicklung (vgl. [BRS+00]). Die Festlegung auf komponentenbasierte Anwendungssysteme bringt den Vorteil einer leichten Wiederverwendbarkeit und hohen Interoperabilität. Komponenten zeichnen sich unter anderem aus durch klare Schnittstellendefinitionen und ihre pragmatische wirtschaftliche Verwertbarkeit auf Komponentenmarktplätzen (siehe dazu Abschnitt 2.1).

Die Zwischenschicht (siehe Abbildung 3.2) schafft die Verknüpfung zwischen beiden Schichten. An dieser Stelle werden Konstrukte wie Komponentenframeworks und Fachkomponenten eingeführt, um Aspekte der fachlichen Modellierung auf die Bestandteile einer komponentenbasierten Anwendung abzubilden. Durch diese Abbildung stehen

dem Entwickler einerseits verhältnismäßig mächtige Konzepte für die Beschreibung und Handhabung von Softwaresystemen zu Verfügung — andererseits garantiert das sehr allgemeine Komponentensystemmodell die pragmatische Realisierbarkeit mit modernen Komponententechnologien, wie beispielsweise CORBA oder EJB.

In den späteren Abschnitten werden die drei Schichten im Detail motiviert und präsentiert. Den Anfang macht die abstrakte, fachliche Modellierung in Abschnitt 3.3. Anschließend wird die unterste Schicht, die Modellierung komponentenbasierter Systeme vorgestellt (siehe Abschnitt 3.4). Die passende Kopplung beider Schichten ist schließlich Thema von Abschnitt 3.5.

3.1 Anforderungen

In diesem Kapitel wird ein einfacher mathematischer Formalismus eingeführt, mit dessen Hilfe es möglich ist, bestimmte Modelle prägnant zu spezifizieren. Ein wichtiger Vorteil einer formalen Modellierung liegt in der präzisen Definition der zugrundeliegenden Konzepte und ihrer Beziehungen untereinander. Auf diese Weise werden in den nachfolgenden Abschnitten sowohl Aspekte der fachlichen Modellierung als auch elementare Bestandteile komponentenbasierter Anwendungssysteme erfaßt. Insgesamt erwarten wir die folgenden Eigenschaften bei einer formalen Modellierung:

Angemessenheit: Das formale Modell soll zentrale Konzepte aus dem jeweiligen Modellierungsbereich umfassen. Bei der fachlichen Modellierung, die in Abschnitt 3.3 vorgestellt wird, gilt es also, gängige und relevante Praktiken der Analysemodellierung zu erfassen. Ebenso soll das Systemmodell (vgl. Abschnitt 3.4) essentielle und dem Anwender bekannte Konzepte gängiger Komponententechnologien, wie beispielsweise Enterprise Java Beans [EJB] oder CORBA Components [CCM99] repräsentieren. Durch die Verbreitung dieser (Quasi-)Standards sind die verwendeten Techniken weithin bekannt und müssen demzufolge berücksichtigt werden. Gleichfalls ist es nötig, daß sich die formale Modellierung an bekannten theoretischen Arbeiten orientiert, um beispielsweise bewährte Spezifikationstechniken wiederverwenden zu können.

Ausdruckstärke: Als dazu gegenläufiger Einfluß fordern wir, daß die formale Modellierung in der Lage ist, essentielle Aspekte durch ein Minimum relevanter Konzepte auszudrücken. Dies bedeutet vor allem, daß von unwichtigen Details abstrahiert wird, um die Komplexität des Modells nicht unnötig zu steigern.

Klarheit: Aus den beiden vorangegangenen Punkten ergibt sich die Forderung nach Klarheit und Verständlichkeit der eingeführten Konzepte, um die Anwendung praktikabel zu gestalten und die Umsetzung auf konkrete Anwendungssysteme zu vereinfachen.

3.2 Einreichung zur OOPSLA (ein Beispiel)

Anhand des folgenden Beispiels werden in den nächsten Abschnitten die einzelnen Konzepte und ihre Zusammenhänge verständlich motiviert und demonstriert. Das „OOPSLA Conference Registration Problem“ hat seinen Ursprung in einer Arbeit von Høydalsvik und Sindre [HS93] und wird seither oft als Beispiel für unterschiedliche Problemstellungen im Umfeld des Softwareengineering herangezogen. Die hier vorgestellte Beispielanwendung basiert auf dieser Aufgabenstellung und läßt sich wie folgt charakterisieren (vgl. [Sih00]):

> „Nachdem die Organisatoren der OOPSLA den „Call for Papers“ versandt haben, können sich Autorengruppen für einen Beitrag im Rahmen der Konferenz registrieren und haben dann bis zu einem festgelegten Termin die Möglichkeit, ihre Arbeit elektronisch abzugeben. Im Rahmen eines Reviews wird jede (rechtzeitig eingereichte) Arbeit an mindestens drei Personen versandt, von diesen gelesen und kommentiert. Der zweite Reviewer erhält dabei die Einreichung mit den Kommentaren des ersten Reviewers, sobald dieser fertig ist. Gleiches gilt für weitere Reviewer, die die Anmerkungen der jeweils vorangegangenen Gutachters erhalten. Abschließend wird eine Gesamtbewertung der Arbeit erstellt und beispielsweise anhand zuvor festgelegter Grenzwerte über die Annahme entschieden. In jedem Fall wird eine entsprechende Benachrichtigung an die Autorengruppe versandt. Ein Informationssystem soll die Anmeldung der Autorengruppen, das Einreichen der Arbeiten sowie den Reviewprozeß verwalten und steuern.“

Bei genauerer Betrachtung dieses Beispiels fällt auf, daß grob zwei Gruppen von Anwendungsfällen (engl. *use-cases*, siehe [RJB98]) auftreten: einerseits der Einreichungsprozeß und andererseits der Vorgang der Begutachtung. Obwohl beide Gruppen über gemeinsame Daten eng gekoppelt sind, erscheinen sie doch relativ unabhängig voneinander. Und zwar sowohl in zeitlicher Hinsicht (das erste Review kann bereits vor der Abgabe der letzten Arbeit anfangen) als auch hinsichtlich ihrer internen Details (z.B. könnte der eigentliche Reviewprozeß auch anders gestaltet sein).

Aufgrund dieser Unabhängigkeit erscheint es sinnvoll, beide Gruppen möglichst weitgehend voneinander zu entkoppeln, um die jeweiligen Systembestandteile getrennt voneinander entwicklen und auch wiederverwenden zu können. Eine solche „separation of concern“ könnte nun auf der Basis einzelner Softwaremodule vorgenommen werden, was jedoch aufgrund der starken Verflechtungen wenig praktikabel ist. Demgegenüber bietet sich die Aufteilung in kooperierende Frameworks an, die an klar definierten Punkten zusammenwirken und anhand des gemeinsamen Anwendungsbereichs zusammengefügt werden. Die entsprechende Anwendung werden wir in den folgenden Abschnitten Schritt für Schritt entwickeln.

3.3 Fachmodelle

Die Modellbildung ist ein zentraler Aspekt der Anwendungsentwicklung. Modelle stellen eine Abstraktion von Konzepten und Zusammenhängen dar und helfen somit vor allem bei der Bewältigung der einhergehenden Komplexität. Von besonderem Interesse in diesem Kapitel ist das „Fachmodell“, das ein Modell des „Anwendungsbereichs“ (bzw. der „Anwendungsdomäne“) und damit die Grundlage für ein Anwendungssystem darstellt. Die folgenden Abschnitte entwickeln ein Verständnis von Fachmodellen als rollenbasierte Spezifikationen, die eine Menge von verwandten Systemen beschreiben. Die Einschränkung auf Fachmodelle ist einerseits getrieben durch die Annahme, daß zwischen zwei Fachmodellen desselben Anwendungsbereichs zwangsläufig Beziehungen bestehen müssen, die eine Integration dieser Modelle ermöglichen. Solche Abhängigkeiten werden in Abschnitt 3.3.7 vorgestellt und diskutiert. Eine andere Motivation für Fachmodelle ist die Tatsache, daß sie eine Dekomposition des Anwendungsbereichs vornehmen und somit eine ideale Basis für die Untersuchung der Integration unterschiedlicher Strukturen darstellen (siehe Abschnitt 3.3.9). Obwohl eine Integration von Artefakten aus allen Stufen eines Entwicklungsprozesses möglich ist, verfolgen wir im weiteren den Weg, nur Modelle der frühen Phasen zusammenzuführen und Auswirkungen auf den Bereich der Realisierung aus den gewonnenen Erfahrungen abzuleiten. Insbesondere vernachlässigen wir auf diese Weise die Integration technischer Details (z.B. Kommunikationsprotokoll, hardwarespezifische Lösungen), deren Zusammenführung und Abstimmung sich meist sehr aufwendig gestaltet.

Durch den Schwerpunkt auf der Dekomposition eines Anwendungsbereichs ist es bei der fachlichen Modellierung vollkommen ausreichend, sich auf einfache, abstrakte Konzepte zu beschränken, die den Anwendungsbereich in eine Menge von Entitäten unterteilen, die miteinander in Beziehungen stehen. Solche „Entity-Relationship-Modelle“ haben sich in den letzten Jahrzehnten bei der Datenmodellierung bewährt und werden an dieser Stelle

um das Konzept der Rolle angereichert, das prinzipiell einzelnen Entitäten Aussagen über ihr angenommenes Verhalten zuordnet.

3.3.1 Der Anwendungsbereich

Mit dem Begriff „Anwendungsbereich“ ist die Vorstellung eines klar umrissenen Ausschnitts beispielsweise der realen Welt verbunden, der festlegt, was Teil eines Modells dieses Anwendungsbereichs ist, und was nicht. Zudem verbinden wir mit dem Begriff die Annahme, daß alle denkbaren Modelle dieses Ausschnitts sich lediglich durch den gewählten Grad der Abstraktion unterscheiden. Jedes einzelne Modell identifiziert auf dem jeweils gewählten Abstraktionsniveau einzelne *Entitäten*, grenzt diese voneinander ab und setzt sie zueinander in *Beziehung*.

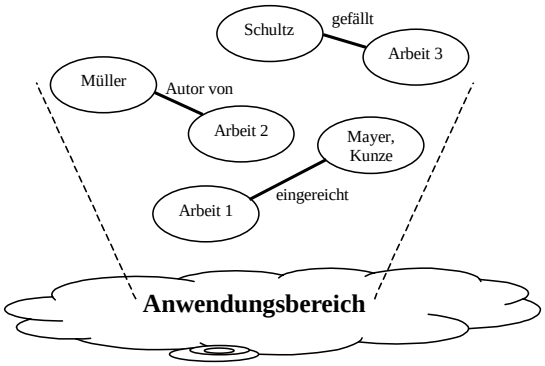


Abbildung 3.3: Entitäten und Beziehungen

Ein einfaches Beispiel ist in Abbildung 3.3 dargestellt. Der gewählte Anwendungsbereich ist hier repräsentiert durch eine Reihe von Autoren und eine Menge von Arbeiten. Diese Entitäten stehen in unterschiedlichen Beziehungen zueinander. Manche Entitäten zeigen (willkürliche) Abstraktionen, so zum Beispiel `Mayer, Kunze`, die auch durch zwei einzelne Entitäten `Mayer` und `Kunze` dargestellt sein könnte. In jeder anderen Modellierung dieses Anwendungsbereichs müssen sich diese Entitäten auch wiederfinden. So taucht `Müller` in einem anderen Modell vielleicht nur als Bestandteil einer Gruppe auf oder findet sich in einem weiteren Modell in ganz anderer Form.

Erläuterung 3.1 (Entitäten und ihre Beziehungen) Die Dekomposition eines Anwendungsbereichs führt auf eine Reihe abgegrenzter Entitäten. Unter Entitäten verstehen wir dabei allgemein bestimmte „Dinge“, von denen es Sinn macht anzunehmen, daß sie ein bestimmtes Verhalten aufweisen. Dies trifft beispielsweise auf die Entität

`Arbeit2` zu, obwohl konkrete Verhaltensaussagen erst in späteren Abschnitten aufgestellt werden können, wenn Entitäten unmittelbar durch Komponenten repräsentiert werden. Zusammen mit bestimmten Beziehungen untereinander ermöglichen Entitäten eine abstrakte Repräsentation des Anwendungsbereichs. Der Zusammenhang zwischen Entität und Beziehung wird über die potentiell unendlichen Mengen $Entity$ und $Relationship$[1] aller entsprechenden Bezeichner definiert:

$$relates \quad : \quad Relationship \rightarrow \wp(Entity)$$

Prinzipiell stellen Beziehungen somit Gruppierungen auf der Menge der Entitäten dar. So faßt die Beziehung `gefällt` aus Abbildung 3.3 die Entitäten `Schultz` und `Arbeit 3` zusammen. Auf gerichtete Beziehungen, wie sie durch `gefällt` bereits nahegelegt werden, soll an dieser Stelle verzichtet werden.

3.3.2 Rollen

Ein essentieller Teil der Modellbildung ist die Abstraktion von konkreten Eigenschaften oder auch Fähigkeiten einzelner Entitäten. Durch diesen Prozeß werden Unterschiede zwischen den Entitäten verdeckt und die Gemeinsamkeiten, die sie aufweisen, herausgearbeitet. Das Konzept der *Rolle* bietet eine praktikable Möglichkeit, solche Gemeinsamkeiten zu erfassen und sie Entitäten zuzuordnen.

Erläuterung 3.2 (Rolle) Eine Rolle stellt eine Abstraktion (und damit einen Ausschnitt) des Verhaltens einer Entität dar. Eine Entität kann mehrere Rollen „spielen“ und mehrere Entitäten können sich entsprechend derselben Rolle verhalten. Mit der unendlichen Menge $Role$ aller Rollen legen wir fest:

$$\begin{aligned} plays \quad &: \quad Entity \rightarrow \wp(Role) \quad \text{wobei} \\ \forall e \in Entity \quad &. \quad plays(e) \neq \emptyset \end{aligned}$$

Während jede Entität mindestens eine Rolle erfüllen muß, kann es Rollen geben, die von keiner Entität gespielt werden. Solche Rollen werden im weiteren als ***abstrakt*** bezeichnet und die Menge $Abstract$ aller abstrakten Rollen ist eine Untermenge aller Rollen: $Abstract \subset Role$.

Rollen stellen somit eine Art Typkonzept für Entitäten dar. Eine Rolle beschreibt eine Menge von Entitäten mit gleicher Charakteristik. So können die Entitäten aus Abbildung 3.3 zum Beispiel unterteilt werden in solche, die die Rolle `Autor` spielen und solche,

[1] Alle eingeführten Bezeichnermengen werden mit englischen Namen versehen, um sie als Teil der formalen Modellierung kenntlich zu machen. $\wp(X)$ bezeichnet die Potenzmenge von X.

die der Rolle `Arbeit` genügen. Anstelle der Auflistung aller Entitäten des Anwendungsbereichs vermittelt die Menge der Rollen ein besseres Verständnis der Zerlegung. Hierbei ist jedoch zu beachten, daß die Menge der Rollen keine Partitionierung der Menge der Entitäten darstellt — die konkrete Zuordnung $plays$ läßt sich hieraus nicht ableiten. Die graphische Darstellung von Rollen ist beispielhaft für die Rollen `Autor` und `Arbeit` in Abbildung 3.4 skizziert. Rollen werden im weiteren stets als Rechtecke mit abgerundeten Ecken symbolisiert (angelehnt an [Rie00]). Das Beispiel zeigt einen wichtigen Konflikt bei der Modellierung, auf den wir in späteren Abschnitten noch genauer eingehen werden: Obwohl manche Arbeiten von mehreren Personen verfasst wurden, legt die Rolle `Autor` nahe, daß es nur einen Autor gibt oder zumindest eine Gruppe von Autoren als eine Person auftritt. In diesem Modell können somit einzelne Autoren einer Arbeit nicht voneinander unterschieden werden. Praktische Probleme ergeben sich beispielsweise dann, wenn eine Person bei mehreren Arbeiten mitgewirkt hat.

Abbildung 3.4: Zwei Rollen

Das Konzept der Rolle ist ideal, um einzelne Aspekte einer Anwendung im Fachmodell zu identifizieren und zu isolieren. Beziehungen zwischen Rollen definieren dabei sowohl den Aspekt selbst als auch das Zusammenwirken unterschiedlicher Aspekte der Anwendung. Solche Beziehungen können grob in drei Kategorien unterteilt werden:

- Beziehungen zwischen Rollen unterschiedlicher Entitäten charakterisieren eine Zusammengehörigkeit von Entitäten und abstrahieren damit von den Beziehungen zwischen Entitäten, wie sie in Abschnitt 3.3.1 eingeführt wurden. Diese Beziehungen werden im weiteren *Assoziationen* genannt.
- Beziehungen zwischen unterschiedlichen Rollen derselben Entität erfassen das Zusammenspiel der verschiedenen Aspekte, in denen eine Entität mitwirkt.
- Beziehungen zwischen Rollen ohne Aussage über Entitäten erlauben weiterreichende Mechanismen wie Vererbung oder auch die hierarchische Komposition zwischen Rollen. Diese beiden Varianten bezeichnen wir allgemein als *Abhängigkeiten*.

Entitäten und Rollen sind in dieser mathematischen Modellierung lediglich als Bezeichner festgehalten. Was beispielsweise eine Rolle zu einer Rolle macht, wird an dieser Stelle

nicht näher festgelegt. Es ist lediglich die intuitive Vorstellung, daß eine Rolle eine bestimmte Eigenschaft ausdrückt, wie diese jedoch konkret geartet ist bleibt noch unklar. So kann eine Rolle sowohl eine Aussage über eine bestimmte Eigenschaft der Entität treffen, oder auch ihr Verhalten spezifizieren. In einem späteren Abschnitt steht mit dem Systemmodell für komponentenbasierte Anwendungen ein Mechanismus zur Verfügung, um Rollen die Bedeutung eines festgelegten Verhaltens zuzuweisen.

3.3.3 Assoziationen

Unter Verwendung des Rollen-Konzepts können nun die Beziehungen zwischen den einzelnen Entitäten auf eine allgemeine Weise definiert werden. Hierzu führen wir den Begriff der *Assoziation* ein und zeigen, daß eine Beziehung zwischen Entitäten als konkrete Ausprägung einer Assoziation[2] verstanden werden kann, falls die beteiligten Entitäten entsprechende Rollen spielen.

Erläuterung 3.3 (Assoziation) Eine Assoziation definiert eine Verknüpfung zwischen zwei oder mehreren Rollen. Jeder Rolle ist hierbei eine bestimmte Kardinalität zugeordnet, die die Anzahl der Entitäten definiert, die im Rahmen einer konkreten Beziehung diese Rolle spielen. Für $\mathbb{N}_\infty := \mathbb{N} \cup \{\infty\}$ gilt

$$associates \quad : \quad (Association \times Role) \rightarrow \mathbb{N}_\infty$$

Wird die Kardinalität einer Rolle mit ∞ belegt, so bedeutet dies, daß die tatsächliche Anzahl an Entitäten dieser Rolle unbegrenzt ist.

Assoziationen manifestieren sich in konkreten Beziehungen zwischen den einzelnen Entitäten, wobei jeder Beziehung stets genau eine Assoziation zugeordnet sei. Dies wird durch die totale Abbildung $corresponds$ festgelegt:

$$\begin{aligned}
corresponds &: Relationship \rightarrow Association \\
maps &: (Relationship \times Role \times \mathbb{N}) \rightarrow Entity, \quad \text{wobei} \\
\forall a \in Relationship, e \in Entity &: e \in relates(a) \Rightarrow \\
\exists r \in Role, n \in \mathbb{N} &: n \leq associates(corresponds(a), r) \wedge \\
&\quad maps(a, r, n) = e \wedge r \in plays(e)
\end{aligned}$$

Durch die Abbildung $maps$ und die nachfolgende Bedingung ist sichergestellt, daß jeder Entität einer Beziehung genau einer entsprechende Rolle ihrer Assoziation zugeordnet ist

[2] Der Zusammenhang zwischen Rolle und Entität sowie zwischen Assoziation und Beziehung entspricht einer Typ-Instanz Beziehung.

(n muß daher eindeutig sein) und diese auch ausgeübt werden kann. Eine Assoziation aus dem Konferenzbeispiel, die einen Autor mit seinen Werken verknüpft, ist in Abbildung 3.5 dargestellt. Konkrete Ausprägungen dieser Assoziation ergeben sich durch das Auffinden geeigneter Beziehungen zwischen Entitäten, die den zwei Rollen `Autor` und `Arbeit` entsprechen. Beispielsweise kämen hierfür die Entitäten `Müller` und `Arbeit 2` in Betracht (vgl. Abbildung 3.3). Die Beziehung `Autor von` ist somit eine konkrete Ausprägung der hier vorgestellten `verfasst`-Assoziation.

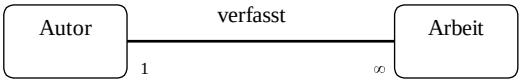


Abbildung 3.5: Die „verfasst“-Assoziation

Im mathematischen Modell könnte diese Assoziation zum Beispiel wie folgt repräsentiert sein:

$$\begin{aligned} associates(verfasst, Autor) &= 1 \\ associates(verfasst, Arbeit) &= \infty \end{aligned}$$

3.3.4 Abhängigkeiten

In den meisten Fällen sind Entitäten mehreren Beziehungen zugeordnet und agieren im Rahmen dieser Beziehungen entsprechend den jeweiligen Rollen. Da eine Rolle eine Annahme über das Verhalten einer Entität repräsentiert, ergibt sich das Verhalten einer Entität aus allen Rollen, die diese zu erfüllen hat. Um eine Integration unterschiedlicher Modelle basierend auf überschneidenden Mengen von Entitäten zu ermöglichen (siehe Abschnitt 3.3.7), sind insbesondere Querbeziehungen zwischen den einzelnen Rollen einer Entität interessant. Diese Abhängigkeiten werden nun klassifiziert und diskutiert.

Erläuterung 3.4 (Abhängigkeit zwischen Rollen) Eine Abhängigkeit repräsentiert das „interne“ Zusammenspiel zwischen mehreren Rollen derselben Entität. Für die Zwecke dieses Modells ist eine Einschränkung auf Abhängigkeiten zwischen einer Rolle und mehreren anderen Rollen ausreichend (1-zu-n Abhängigkeit). Die folgende Abbildung $depends$ definiert die entsprechenden Rollen für die unendliche Menge $Dependency$ aller gerichteten Abhängigkeiten.

$$depends \quad : \quad Dependency \rightarrow Role \times \wp(Role)$$

Im weiteren werden eine Reihe konkreter Ausprägungen von Abhängigkeiten besprochen, die beispielhaft in Abbildung 3.6 dargestellt sind (für eine detaillierte Diskussion der graphischen Darstellung siehe Kapitel 5). Die Rollen der 1-zu-n Beziehung werden dabei stets mit r_1 bis r_n sowie mit p bezeichnet.

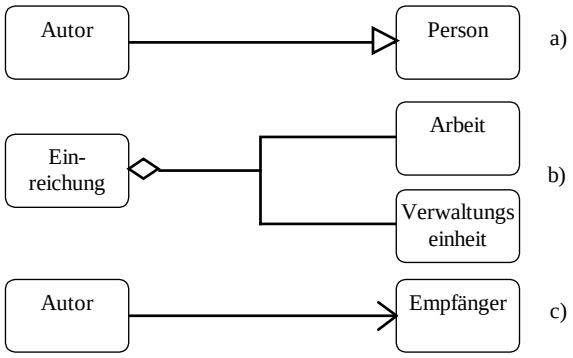


Abbildung 3.6: Abhängigkeiten zwischen Rollen

Erläuterung 3.5 (Generalisierung) Eine Generalisierungsbeziehung zwischen verschiedenen Rollen ist eine Abhängigkeit mit folgender Bedeutung: Jede Rolle $r_i (1 \leq i \leq n)$ kann dort verwendet werden, wo die Rolle p gefordert ist. Werden Rollen als Typen interpretiert, so führt die Generalisierungsbeziehung zu einer Subtyp-Beziehung und nicht zu der klassischen (Code-)Vererbung der Objektorientierung.

Teil a) der Abbildung 3.6 beschreibt die Abhängigkeit zwischen der Rolle `Autor` und der Rolle `Person`. Ein Autor ist eine Person im allgemeinen Sinne, da ihm oder ihr beispielsweise auch ein Name und ein Alter zugeordnet ist. In Fällen, in denen die Rolle `Person` gefragt ist, kann demzufolge eine Entität eingesetzt werden, die die Rolle `Autor` spielt. Zwischen beiden Rollen besteht damit eine Subtyp-Beziehung und die Rolle `Person` wird durch die Rolle `Author` *konkretisiert*. Zudem gilt, daß von einer nicht abstrakten Rolle keine abstrakte Rolle abgeleitet werden kann.

Erläuterung 3.6 (Aggregation) Eine Aggregation faßt mehrere Rollen zu einer neuen Rolle zusammen. Damit entspricht die Aggregation einer „umgedrehten“ Generalisierung: wenn eine Entität die Rolle p spielen kann, so bedeutet das gleichfalls ein Verhalten entsprechend den Rollen r_1 bis r_n. Die Aggregation wird jedoch im weiteren in einer schwächeren Variante verwendet, da wir fordern, daß die aggregierende Rolle das Verhalten der Entität nur soweit einschränkt, wie es durch die aggregierten Rollen impliziert wird.

Abbildung 3.6 zeigt in Teil b), daß eine `Einreichung` hier aus zwei unterschiedlichen Rollen zusammengesetzt ist. Einerseits verhält sie sich (gegenüber ihren Autoren) als `Arbeit` — andererseits stellt sie (für das zugrundeliegende Informationssystem der Konferenz) eine `Verwaltungseinheit` dar, die eine geeignete Organisation aller Beiträge ermöglicht. In Kombination mit der Möglichkeit, Rollen als abstrakt zu definieren, kann auf diese Weise festgelegt werden, daß die Rollen `Arbeit` und `Verwaltungseinheit` nur in Kombination von einer Entität erfüllt werden können.

Erläuterung 3.7 (Anforderung) Durch eine Anforderung ist festgelegt, daß eine Entität die Rollen p nur dann erfüllen kann, wenn sie gleichzeitig auch die Rollen r_1 bis r_n spielt. Auf diese Weise repräsentieren Anforderungen Detailwissen über das interne Zusammenspiel der einzelnen Rollen einer Entität. Obwohl in vielen Fällen solche Informationen die konkrete Realisierung einer Rolle betreffen, lassen sich oft bereits in der fachlichen Modellierung entsprechende Zusammenhänge aufstellen.

Da Autoren laut der Aufgabenstellung in unserem Beispiel nach erfolgtem Review von der Annahme oder Ablehnung ihrer Arbeit informiert werden müssen, ist es erforderlich, daß sie entsprechende Nachrichten empfangen können. In Teil c) von Abbildung 3.6 ist dieser Sachverhalt ausgedrückt durch eine Abhängigkeit zwischen der Rolle `Autor` und der Rolle `Empfänger`.

Erläuterung 3.8 (Komposition) Abhängigkeiten existieren stets zwischen den Rollen einer Entität, die das „äußere“ Erscheinungsbild einer Entität definieren. In manchen Fällen ist es für ein besseres Verständnis einer Rolle p hilfreich, diese in Abhängigkeit weiterer Rollen r_1 bis r_n zu definieren, die nach außen nicht sichtbar sein sollen. Diese *Komposition* betrifft insbesondere den Fall, in dem die neue Rolle eine Erweiterung oder Adaption des Verhaltens der referenzierten Rollen darstellt.

Komposition und Abhängigkeit sind eng miteinander verwandt und werden in der graphischen Darstellung beide mit einem einfachen Pfeil symbolisiert. Der Unterschied zeigt sich in der Anordnung der einzelnen Kästchen, die die Rollen symbolisieren. In der Darstellung einer Komponente liegen die „sichtbaren“ Rollen unmittelbar an der Kante, während andere Rollen im inneren Bereich des Kästchens angeordnet werden.

Die hier vorgestellten Untergruppen von Abhängigkeiten partitionieren die Menge $Dependency$ aller Abhängigkeiten und werden mit den folgenden Mengenbezeichnern abgekürzt: Gen bezeichnet alle Generalisierungen, Agg alle Aggregationen, $Comp$ alle Kompositionen und Req alle Anforderungen. Für die Aggregation aus Abbildung 3.6 gilt beispielsweise:

$$depends(Agg1) = (Einreichung, \{Arbeit, Verwaltungseinheit\})$$

Bei der Integration unterschiedlicher Modelle werden Abhängigkeiten dabei helfen, die diversen Annahmen über das Verhalten einer Entität konsistent zusammenzuführen. Zudem ermöglichen Abhängigkeiten die Reduzierung der Granularität von Rollen und auf diese Weise die Steigerung der Wiederverwendbarkeit von Rollendefinitionen (siehe dazu methodische Überlegungen in Kapitel 6). Konkret bieten sich dem Entwickler zwei Wege, eine Rolle in kleinere Bestandteile zu zerlegen (vgl. Abbildung 3.7).

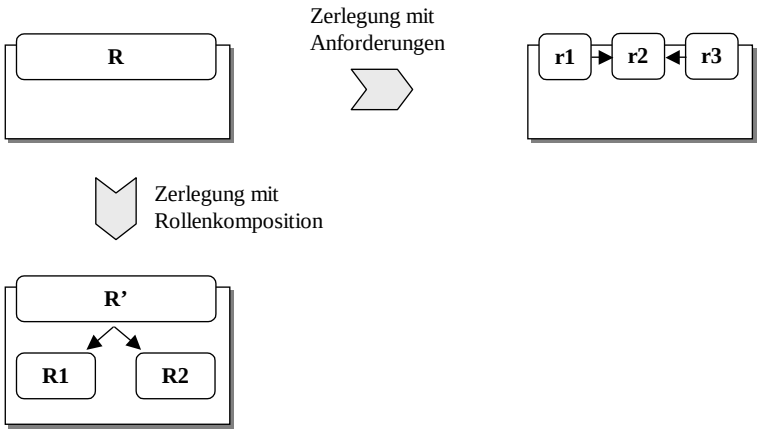


Abbildung 3.7: Zwei Arten der Zerlegung einer Rolle

Einerseits ermöglichen es Anforderungen, eine Rolle in feinere Rollen zu zerteilen und so eine detailliertere Aussage über das Verhalten der Entität zu treffen. Abbildung 3.7 skizziert diesen Vorgang anhand einer Rolle `R`, die in drei weitere Rollen `r1`, `r2` und `r3` zerlegt wurde. Die dabei entstehenden Anforderungen lassen genauere Rückschlüsse auf das Zusammenspiel zu. Diese Zerlegung werden wir später verwenden, um Kollaborationen zwischen Komponenten in Teilabläufe zu zerlegen, die eine Integration der unterschiedlichen Komponentenframeworks vereinfacht.

Andererseits wird ebenfalls gezeigt, wie die Rolle `R` so umgeformt wird, daß sie ihre Funktionalität als *Adapter* weiterer, interner Rollen realisiert, die nach außen hin nicht in Erscheinung treten. Diese Zerteilung mag willkürlich erscheinen, ist dann aber sinnvoll, wenn diese internen Rollen mehrfach genutzt werden können. Im Beispiel aus Abbildung 3.7 tragen die internen Rollen die Bezeichner `R1` und `R2`. Die Rolle `R'` beschreibt die Funktionalität des Adapters, der im Verhalten nach außen mit der Funktionalität von `R` übereinstimmt.

3.3.5 Fachmodell

Mit den Konzepten der letzten Abschnitte kann nun der Begriff „Fachmodell“ präzisiert werden:

Erläuterung 3.9 (Fachmodell) Das Fachmodell ist ein auf den Konzepten, Strukturen und Entitäten des Anwendungsbereichs basierendes Modell. Es repräsentiert ausschließlich anwendungsorientierte Begebenheiten und zeigt jenen Ausschnitt, der durch das Anwendungssystem softwaretechnisch unterstützt werden soll. Ein Fachmodell impliziert eine *zielgerichtete* Dekomposition des Anwendungsbereichs in einzelne Rollen und ihre Beziehungen untereinander. Ein Fachmodell ist repräsentiert durch ein Tupel $(Association, Role, Dependency)$ mit folgender Bedeutung:

Assoziationen definieren ein abstraktes Verständnis des Zusammenwirkens oder der Zusammengehörigkeit von Entitäten. Im Rahmen jeder Assoziation werden dabei Annahmen über die Charakteristik der beteiligten Entitäten in Form von *Rollen* spezifiziert. *Abhängigkeiten* definieren Beziehungen zwischen Rollen entweder ohne eine Aussage über mögliche Zuordnungen zu Entitäten (z.B. bei abstrakten Rollen oder Generalisierungsbeziehungen) oder mit der Forderung, daß die beteiligten Rollen derselben Entität zugeordnet sein müssen.

Insbesondere trifft ein Fachmodell nur im beschränkten Rahmen Aussagen über die tatsächliche Instantiierung, also die Abbildung der Rollen und Assoziationen auf die Entitäten des Anwendungsbereichs (vgl. Abbildung $plays$ in Erläuterung 3.2). So kann bei der Festlegung eines Fachmodells nicht verhindert werden, daß zwei bestimmte Rollen von derselben Entität gespielt werden. Hierin liegt ein Freiraum bei der konkreten Ausprägung eines Fachmodells, der es auch erlaubt, mehrere Rollen derselben Entität zuzuweisen. Soll umgekehrt festgelegt werden, daß zwei Rollen stets von derselben Entität gespielt werden, so muß über eine Aggregationsbeziehung eine neue Rolle definiert werden[3]. Ein Beispiel hierfür findet sich in Abbildung 3.8, in der die Aggregation der Rollen `Arbeit` und `Verwaltungseinheit` in einer vereinfachten graphischen Darstellung gezeigt wird. Beide Notationen werden im Weiteren synonym verwendet. Um Assoziationen zwischen mehr als zwei Rollen besser darstellen zu können, wurden in der Abbildung Ellipsen eingeführt, die den Bezeichner der Assoziation enthalten.

In Abbildung 3.8 ist ein Fachmodell für die Einreichung von Beiträgen zur OOPSLA-Konferenz dargestellt. Konkret werden drei Assoziationen eingeführt (repräsentiert durch Ellipsen), die großteils auf derselben Menge von Rollen definiert sind. Die Assoziationen

[3] Auf ähnliche Weise ließe sich eine Abhängigkeit für den gegenseitigen Ausschluss von Rollen definieren (vgl. [Rie00]). Darauf wurde jedoch verzichtet, da kaum Bedarf für ein solches Konstrukt besteht.

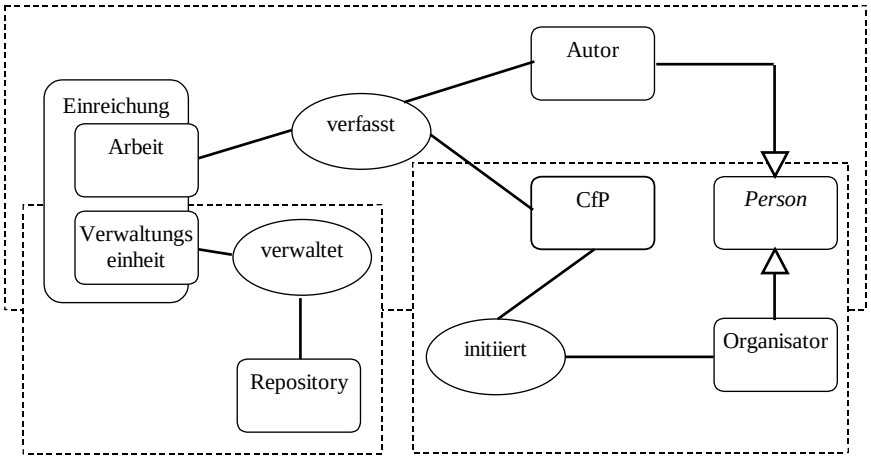


Abbildung 3.8: Einreichung zur Konferenz: Ein Fachmodell

stehen dabei sowohl für Zugehörigkeiten als auch für Abläufe in diesem Modell. So repräsentiert die Assoziation `verfasst` zwar die logische Zusammengehörigkeit zwischen Autoren und ihrer Arbeit, steht andererseits jedoch gleichfalls für den Prozeß, der auf diesen Strukturen operiert. Beispielsweise also das Erhalten des „Call for Papers“(CfP) und das Verfassen einer geeigneten Einreichung.

Wären die Rollen in der aggregierten Rolle `Einreichung` abstrakt, so könnten `Arbeit` und `Verwaltungseinheit` nur gemeinsam einer Entität zugeordnet werden. In der dargestellten Notation kann es dagegen Arbeiten geben, die noch nicht verwaltet werden und Verwaltungseinheiten, die keine Arbeit repräsentieren. Bei einer konkreten Ausprägung dieses Fachmodells ist insbesondere offen, ob jede Rolle von genau einer Entität implementiert wird oder ob es Entitäten gibt, die mehrere Rollen auf sich vereinen. Ein solcher Kandidat wäre beispielsweise eine Entität für den Organisator, der einerseits die Rolle `Organisator` spielt und andererseits selbst als `Autor` einer Einreichung bei der OOPSLA auftritt.

So zeigt Abbildung 3.8 interne Abhängigkeiten zwischen den Rollen derselben Entität (impliziert durch die Aggregation), Assoziationen zwischen Rollen unterschiedlicher Entitäten und Abhängigkeiten zwischen abstrakten und konkreten Rollen. Ein Beispiel für letzteres findet sich in der Generalisierungsbeziehung zwischen der abstrakten Rolle `Person` sowie den Rollen `Autor` und `Organisator`. Wie bereits erwähnt werden in der graphischen Darstellung die Namen abstrakter Rollen kursiv geschrieben.

3.3.6 Ausblick: Rollen für Assoziationen

Eine genaue Betrachtung des letzten Beispiels führt zu der Erkenntnis, daß Ähnlichkeiten zwischen den einzelnen Assoziationen bestehen. Ein `Organisator` ist sicherlich `Autor` und `verfasst` als solcher den *Call for Papers*, der wiederum als `Arbeit` aufgefaßt werden kann.

Eine entsprechende Erweiterung des vorgestellten Modells bestünde darin, auch Assoziationen Rollen spielen zu lassen. In diesem Fall wäre es möglich, die Rollen unterschiedlicher Assoziationen mit den vorgestellten Mechanismen zueinander in Beziehung zu setzen. Für das Beispiel der OOPSLA-Konferenz zeigt Abbildung 3.9 die Vererbung zwischen den zwei Assoziationen `verfasst` und `Ausschreibung`.

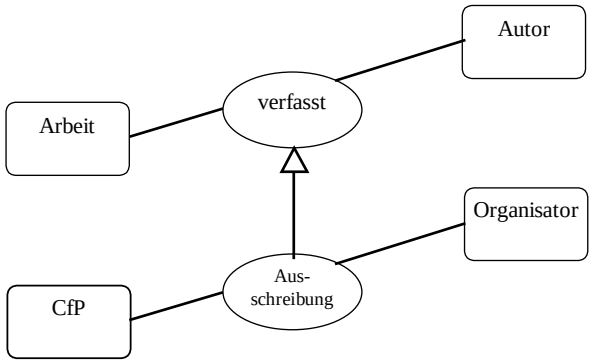


Abbildung 3.9: Generalisierung zwischen Rollen von Assoziationen

Das Ergebnis dieser Überlegung besteht in einer Gleichbehandlung von Entität und Beziehung. In diesem Fall wäre im obigen Beispiel `Ausschreibung` lediglich eine Rolle und es wäre letztlich unbedeutend, ob diese Rolle von einer Entität oder einer Beziehung erfüllt wird. Die entsprechende Beziehung wäre über `Ausschreibung` ebenfalls in der Lage, in einer weiteren Beziehung eine Rolle zu spielen.

Die Unterscheidung von Entität und Beziehung ist allerdings nicht nur für das Verständnis des Anwendungsbereichs entscheidend. Auch methodische Gründe sprechen für die klare Trennung der Konzepte, die eine relativ einfache Integration mehrerer Modelle ermöglicht (siehe den nachfolgenden Abschnitt 3.3.7). Aus diesen Gründen wird im weiteren die konzeptuelle Trennung beibehalten bei einer reduzierten Mächtigkeit hinsichtlich der Spezifikation und Handhabung von Assoziationen.

3.3.7 Zusammenführung von Modellen

Bereits bei der Konzeption eines Fachmodells müssen Teilsichten miteinander kombiniert werden. Im Rahmen eines Use-Case-zentrierten Entwicklungsprozesses stehen einzelne Anwendungsfälle im Vordergrund, die sich auf Beziehungen einer geeigneten Menge von Entitäten auswirken. Die Vorstellung von einer Assoziation als einem minimalen Fachmodell führt bereits zu der Frage nach der Integration einzelner Fachmodelle. In der Darstellung des Fachmodells in Abbildung 3.8 wurden die einzelnen Assoziationen mit ihren jeweiligen Rollen durch gestrichelte Rechtecke kenntlich gemacht. Ein Integration wurde hier beispielsweise durch die Identifizierung gemeinsamer Rollen hergestellt. An einer Stelle wurde die Integration durch die Definition einer neuen Rolle `Einreichung` vorgenommen, die zwei Rollen aus unterschiedlichen Sichten auf eine Entität kombiniert.

Bei der Integration von Fachmodellen ergeben sich eine ganze Reihe von Fragestellungen, wie z.B. „Gibt es Widersprüche zwischen den Modellen?“ oder “Wie werden die einzelnen Modelle aufeinander abgebildet?“. In diesem Abschnitt wird diesen Fragen nachgegangen und dabei werden systematisch die verschiedenen Wege der Integration aufgezeigt und untersucht.

Dazu wird zunächst die mengenbasierte, formale Fundierung der vorangegangenen Abschnitte hinsichtlich einer unendlichen Menge $Model$ von Fachmodellen (resp. ihrer Bezeichner) erweitert. OBdA nehmen wir an, daß die einzelnen Modelle Teilmengen auf den jeweiligen Bezeichnermengen implizieren. Für ein Fachmodell $m \in Model$ schreiben wir daher auch das Tupel $(Role_m, Association_m, Dependency_m)$ und beschränken die jeweiligen Abbildungen wie beispielhaft anhand der Abbildung $relates$ gezeigt (vgl. Abschnitt 3.3.1):

$$relates\,|_m \;\; : \;\; Association_m \rightarrow \wp(Role_m \rightarrow \mathbb{N}_\infty)$$

Die Abbildung $relates\;|_m$ ist eine Einschränkung von $relates$ auf die dem Modell m zugehörigen Assoziationen und Rollen.

Als weitere Vereinfachung betrachten wir im weiteren lediglich die Integration von zwei Fachmodellen — dieser Vorgang läßt sich jedoch einfach auf die Integration mehrerer Fachmodellen verallgemeinern. Prinzipiell lassen sich zwei unterschiedliche Varianten der Integration unterscheiden, die hier kurz skizziert werden, bevor eine detailliertere Diskussion in den nachfolgenden Abschnitten erfolgt.

Übersicht

Für ein besseres Verständnis der Anforderungen an eine Integration von Fachmodellen ist es erforderlich zu bestimmen, zu welchem Zeitpunkt im Lebenszyklus eines Anwendungssystems eine Integration stattfindet. Hierbei sind hauptsächlich die beiden folgenden Varianten von Bedeutung:

Early Integration: Hierunter verstehen wir den klassischen Weg der Anwendungsmodellierung. Während der Modellbildung werden unterschiedliche Sichten auf eine gemeinsame Dekomposition des Anwendungsbereichs abgebildet. Konflikte können und müssen behoben werden, indem das Modell so lange verbessert wird, bis alle Sichten geeignet vereint sind. Sind die einzelnen Sichten bereits durch eigene Fachmodelle repräsentiert, kann bei der Integration die Modellierung entsprechend angepaßt werden, um geeignete Kompromisse umzusetzen.

Late Integration: Eine späte Integration bietet dem Entwickler nicht mehr die Möglichkeit, die zu integrierenden Fachmodelle noch im grösseren Umfang zu verändern. Vielmehr soll ein Fachmodell möglichst in seiner ursprünglichen Form erhalten bleiben — einerseits, um ein Verständnis dieser isolierten Sicht weiterhin zu ermöglichen und andererseits aufgrund möglicherweise bestehender Implementierungen, die sich an der gegebenen Struktur orientieren. Eine Integration solcher Fachmodelle gestaltet sich naturgemäß bedeutend aufwendiger, nicht zuletzt aufgrund der unterschiedlichen Dekompositionen, die durch die einzelnen Fachmodelle impliziert werden.

Anhand des Beispiels der OOPSLA-Konferenz werden nun beide Varianten verdeutlicht. In beiden Fällen versuchen wir unterschiedliche Sichten, wie sie durch die Use-Cases „Einreichung“ und „Review“ gegeben sind, zu einem konsistenten Modell zusammenzuführen. Im Fall einer *Early Integration* existieren für die einzelnen Sichten noch keine zugehörigen Implementierungsartefakte. Die Modelle der beiden Sichten können daher recht problemlos zu einem neuen Fachmodell zusammengeführt werden, das schließlich in weiteren Schritten realisiert wird. Durch die Aufgabenstellung wird beispielsweise verlangt, daß ein Autor einer Einreichung auch als Gutachter einer anderen Arbeit herangezogen werden kann. Das Verständnis der Rollen `Autor` und `Gutachter` ist nun durch eine isolierte Betrachtung der beiden Sichten gegeben und bei der Zusammenführung werden beide Rollen zu einer neuen Rolle aggregiert. Durch die Tatsache, daß sich die einzelnen Verhaltensaspekte beider Rollen gegenseitig beeinflussen können und daß durch die Zusammenführung der Sichten weitere Konsistenzkriterien relevant werden („Ein Gutachter darf nicht seine eigene Arbeit beurteilen“) erfordert die neue Rolle eine entsprechend angepaßte Verhaltensbeschreibung. Schließlich führt der Vorgang der Integration zu einem

konsistenten, einheitlichen Fachmodell, das beide Sichten in sich vereint. Das Wissen über Verhaltensaspekte der einzelnen Rollen kann hierbei nur schwerlich wiederverwendet werden — stattdessen ist es bei der Integration erforderlich, sich genaustens mit dem Verständnis der beiden Sichten auseinanderzusetzen, um die Integration korrekt durchführen zu können.

Sind die den einzelnen Sichten zugeordnente Fachmodelle bereits (teilweise) in eine Realisierung übergeführt, so entspricht die Zusammenführung einer *Late Integration*. Die Erstellung eines gemeinsamen Fachmodells ist in diesem Fall verhältnismäßig aufwendig, je nach den Unterschieden in den beiden Sichten. Die eingesetzten Implementierungsbestandteile müssen zudem so geartet sein, daß sie geeignet adaptiert oder konfiguriert werden können, um zusätzliche Konsistenzbedingungen unterbringen zu können.

Im Rahmen moderner, langlebiger Anwendungssysteme findet sich häufig eine Kombination aus beiden Integrationsformen. In den frühen Phasen der Entwicklung werden die unterschiedlichen Sichten aus den diversen Anwendungsfällen kombiniert. Wenn sich nach der Fertigstellung der Anwendung zusätzliche Anforderungen ergeben, oder bestehende Anforderungen verändern, müssen dementsprechend neue Sichten integriert oder bestehende modifiziert werden.

Eine weitere, prinzipielle Unterscheidung besteht darin, auf welche Weise eine Integration vorgenommen wird und wie das jeweilige Ergebnis aussieht. Wieder können zwei Varianten unterschieden werden:

Modellkoexistenz: In manchen Fällen erscheint es zu aufwendig, mehrere Fachmodelle zu einem integrierten Fachmodell zusammenzufügen. Ein häufige Ursache findet sich beispielsweise in sehr unterschiedlichen Repräsentationen derselben Information in den verschiedenen Fachmodellen. Wenn es zudem nur relativ wenige Berührungspunkte zwischen den Modellen gibt, kann es vorteilhafter sein, beide Modelle nur in Beziehung zueinander zu setzen. Unter Modellkoexistenz verstehen wir die Eigenschaft, daß die Information eines Anwendungsbereichs redundant über mehrere Fachmodelle verteilt ist. Einzelne Koordinationspunkte synchronisieren die Modelle und schaffen so ein konsistentes Gesamtmodell. Obwohl diese Lösung häufig pragmatisch erscheint, führt sie zu einem hohen Grad an Redundanz, einem unter Umständen komplexen Geflecht von Abhängigkeiten und somit letztlich zu einem geringen Grad an Verständlichkeit.

Modellintegration: Die „echte“ Integration von Fachmodellen ergibt wieder ein Fachmodell, das sämtliche Informationen der integrierten Modelle enthält und zu einem Ganzen zusammenstellt. Dieser Ansatz ist sehr gut vergleichbar mit den Bestrebungen beispielsweise in der Automobilindustrie, ein gemeinsames Produktmodell (*common product model*) für alle Teilaspekte des Automobilbaus zu finden und zu

spezifizieren. Dem Nachteil eines hohen Aufwands bei der Modellfindung steht der entscheidende Vorteil gegenüber, daß der Anwendungsbereich konsistent und redundanzfrei modelliert ist.

Die Modellintegration ist ein entscheidender Ansatz dieser Arbeit. Im folgenden werden beide Ansätze detailliert diskutiert und die Entscheidung für die Modellintegration als Favorit wird motiviert und argumentativ unterstützt.

Modellkoexistenz

Bei der Betrachtung zweier unabhängiger Fachmodelle erscheint es sehr wahrscheinlich, daß überlappende Bereiche des Anwendungsfelds auf ganz unterschiedliche Art und Weise repräsentiert sind. Aufgrund einer eigenständigen Entwicklung der Modelle basieren diese jeweils auf einem Verständnis des Anwendungsbereichs, das optimal auf die jeweilige Zielsetzung (z.B. ein funktionaler Aspekt) abgestimmt ist. Bei der Integration dieser Modelle erscheint es daher nicht nur pragmatisch sondern auch für ein weiteres Verständnis vorteilhaft, die einzelnen Modelle so zu belassen wie sie sind.

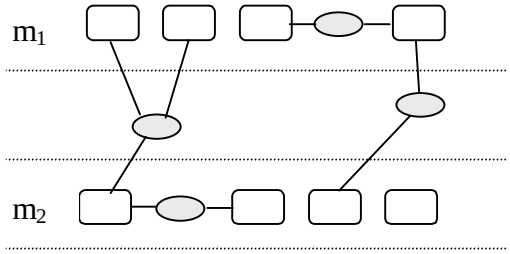


Abbildung 3.10: Fachmodelle und Integrationsmodelle

Bei dieser Variante wird daher versucht, die einzelnen Repräsentationen aufeinander abzubilden, um mögliche Wege der Synchronisation der enthaltenen Information zu finden und zu etablieren. Als Beispiel seien zwei Modelle m_1 und m_2 gegeben, wie in Abbildung 3.10 dargestellt. Die horizontale Aufreihung von Rollen und Assoziationen bezeichnet hierbei jeweils ein Modell, während dazwischen sogenannte *Integrationsmodelle* definiert sind, die konkrete Abhängigkeiten erfassen. Sie basieren auf den bereits existierenden Rollenspezifikationen und definieren geeignete Assoziationen dazwischen.

Der entscheidende Nachteil dieses Lösungsansatzes liegt in den oft mannigfaltigen Abhängigkeiten zwischen den Modellen, die umso komplexer sind, je unterschiedlicher die Informationen in den Modellen repräsentiert sind. Zudem übernehmen die Integrationsmodelle oft nicht nur die technischen Aspekte der Synchronisation sondern enthalten

selbst fachliche Anwendungslogik, was einer klaren Trennung von Fachlichkeit und Technik zuwiderläuft.

Modellintegration

Der vielversprechendere Ansatz ist die tatsächliche Integration der diversen Fachmodelle zu einem neuen, zentralen Fachmodell, das sämtliche Aspekte in sich vereint. Dieses Vorhaben ist sicherlich schwieriger umzusetzen als die Lösung aus dem vorherigen Abschnitt, zwingt jedoch zur detaillierten Auseinandersetzung mit den einzelnen Modellen. Der Vorteil besteht vor allem in der unifizierten Repräsentation des Anwendungsbereichs.

Der Vorgang der Integration kann bei dieser Variante einfach als Verfeinerungsschritt zweier Modelle interpretiert werden. Nicht zuletzt sieht man darin die Vision einer Softwareerstellung (hauptsächlich) durch die Integration von Frameworks, die je einem Fachmodell entsprechen. Aufgrund des Schwerpunkts in dieser Arbeit auf dieser Variante werden wir nun den Begriff der Modellintegration präzisieren.

Erläuterung 3.10 (Modellintegration) Aus der Integration zweier Fachmodelle entsteht ein neues Fachmodell, das die unterschiedlichen Sichten und ihre jeweiligen Beschränkungen beinhaltet. Gegeben sei eine Verfeinerungsrelation $\leq$ auf der Menge der Modelle. Die Integration entspricht einer geeigneten Abbildung $\circ$, für die gilt:

$$\begin{aligned} \circ &: Model \times Model \rightarrow Model \\ \forall m_1, m_2, m_3 \in Model &: m_1 \circ m_2 = m_3 \Rightarrow m_3 \leq m_1 \wedge m_3 \leq m_2 \end{aligned}$$

In der zweiten Zeile ist die intuitive Vorstellung festgehalten, daß der Vorgang der Integration eine Verfeinerung der beiden einzelnen Fachmodelle darstellt und insbesondere die jeweiligen Charakteristika erhält [BS01].

Die eingesetzte Verfeinerungsrelation $\leq$ ist essentiell für die Beurteilung der Korrektheit von Integrationsvorgängen. Für ein besseres Verständnis stellen die nachfolgenden Abschnitte eine systematische Charakterisierung möglicher Integrationsformen vor.

3.3.8 Flache Integration

Eine grundlegende Annahme vieler Ansätze zur Integration unterschiedlicher Anwendungssichten besteht darin, daß diese Sichten auf demselben Abstraktionsniveau angesiedelt sind und somit — übertragen auf die Konzeption der Fachmodelle dieses Kapitels —

gemeinsame Entitäten existieren, über die nur unterschiedliche Aussagen getroffen werden. Ein Beispiel findet sich im Bereich des *aspect-oriented programmings* [KLM+97], bei dem diverse Aspekte einer Klasse zu einer neuen Klasse verwoben werden. Diese „flache“ Integration erfordert bereits die Auseinandersetzung mit Abhängigkeiten zwischen den unterschiedlichen Sichten.

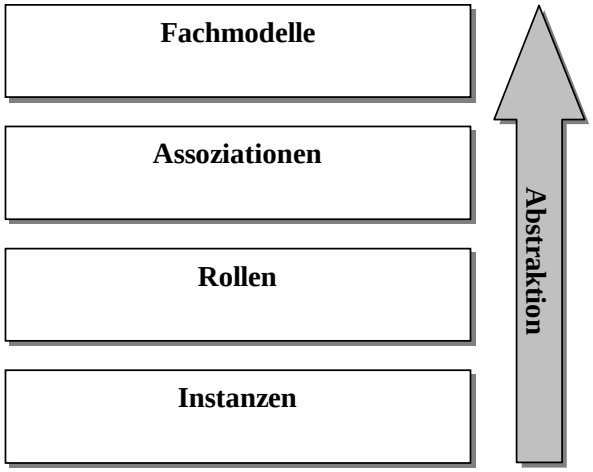


Abbildung 3.11: Abstraktionsstufen der fachlichen Modellierung

Für eine systematische Untersuchung der Modellintegration ordnen wir die Konzepte der fachlichen Modellierung als aufeinander aufbauende Abstraktionsebenen an, wie in Abbildung 3.11 dargestellt. Die Zusammenführung der Modelle kann nun für jede dieser Ebenen getrennt untersucht werden. Hierbei seien die zu integrierenden Modelle mit $m_1, m_2 \in Model$ und das resultierende Modell mit $m_3 \in Model$ bezeichnet.

Entitätenebene: Im einfachsten Fall sind die zu integrierenden Modelle zueinander *orthogonal* in dem Sinne, daß sie zwar dieselben Entitäten betreffen, jedoch sich gegenseitig in ihren Aussagen über die Anwendung nicht beeinflussen. Die Integration erfolgt durch das Zusammenfügen der einzelnen Rollen und Assoziationen durch eine einfache Vereinigung der jeweiligen Mengen.

Lediglich zur Laufzeit gilt es, eine geeignete Instantiierung des neuen Modells, also insbesondere eine Abbildung $plays$ zu finden, die den Entitäten der Anwendung die passenden Rollen aus den ursprünglichen Modellen m_1 und m_2 zuweist (siehe Abbildung 3.12). Der Vorteil dieser Variante ergibt sich aus der hohen Flexibilität bei der Integration, da eine Zusammenführung keine weiteren Entwurfsentscheidungen notwendig macht.

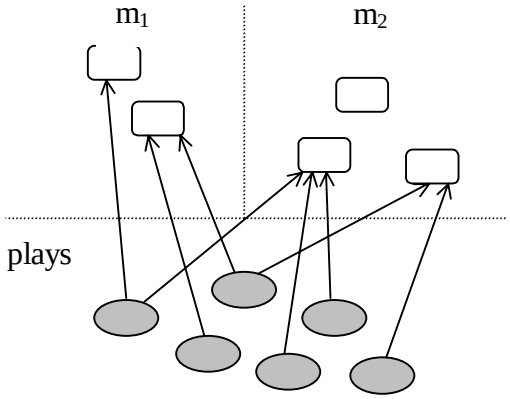


Abbildung 3.12: Fachmodellintegration auf der Entitätenebene

Der Nachteil liegt in mangelnem Verständnis über die korrekte Instantiierung, die nicht durch das Modell vorgegeben wird. Eine solche Festlegung würde die Einführung weiterer Rollen erfordern, um beispielsweise Regelungen der Art „Jeder `Autor` aus Fachmodell m_1 spielt die Rolle `Person` aus Fachmodell m_2“ festzuhalten. Gerade aus diesem Grund erscheint die Integration auf der Entitätenebene nur für Ausnahmefälle geeignet.

Rollenebene: Im Regelfall überschneiden sich die Aussagen, die die zu integrierenden Fachmodelle über den Anwendungsbereich treffen. Diese Eigenschaft läßt sich bereits auf der Ebene der Rollen beobachten, z.B. wenn zwei Rollen aus m_1 und m_2 identische Charakteristika einer Entität beschreiben. Aus solchen und ähnlichen Gründen ist es erforderlich, eine einheitliche Gruppe von Rollen zu finden, die sämtliche interessanten Eigenschaften eindeutig beschreiben. Weiterhin müssen die bestehenden Rollen der Fachmodelle auf diese neuen Rollen abgebildet werden, um bestehende Assoziationen übernehmen zu können.

Die eigentliche Integration der beiden Modelle erfolgt nun prinzipiell auf zwei unterschiedlichen Wegen:

- Adressieren die Assoziationen der einzelnen Modelle dieselben Charakteristika einer Entität, lassen sich also beispielsweise gemeinsame Rollen identifizieren, so korrelieren darüber die beiden Fachmodelle.
- Ähnlich verhält es sich bei der Existenz gemeinsamer Rollen, die über Abhängigkeiten innerhalb einer Entität von anderen Rollen referenziert werden.

- Schließlich kann ein Entwickler aufgrund weiterer Entwurfsentscheidungen die Instantiierung des integrierten Modells reglementieren. Durch die Aggregation zweier Rollen aus m_1 und m_2 ist somit ebenfalls eine Kopplung beider Fachmodelle erreichbar.

Im weiteren werden konkrete Schritte vorgestellt, um die Menge der Rollen des neu entstehenden Fachmodells $m_3 \in Model$ aus den Mengen $Role_{m_1}$ und $Role_{m_2}$ abzuleiten (es gilt: $m_3 = m_1 \circ m_2$). Konkret werden dabei nur jene Rollen untersucht, die voraussichtlich von derselben Entität erfüllt werden müssen.

Schritt 1: Zwischen den Rollen aus $Role_{m_1}$ und $Role_{m_2}$ lassen sich Identitäten ausmachen und zwei solche Rollen unterscheiden sich demnach lediglich in ihren Bezeichnern. Es kann somit beliebig eine der zwei Rollen ausgewählt werden. Mit einer geeigneten Identitätsbeziehung $id : Role \times Role$ ergibt sich somit:

$$\begin{aligned} \forall r_1 \in Role_{m_1}, r_2 \in Role_{m_2} \quad &: \quad r_1 \; id \; r_2 \Rightarrow \\ \exists r_3 \in Role_{m_3} \quad &: \quad r_3 = r_1 \vee r_3 = r_2 \end{aligned}$$

In unserem Beispiel konnte die Rolle `Person` im Umfeld der Autorenschaft einer Arbeit und im Bereich der Erstellung einer Ausschreibung als identisch ausgemacht werden. Die gleiche Benennung der Rollen in den unterschiedlichen Fachmodellen ist dabei nur als Indiz zu werten. Aus diesem Grund umfaßt die Identitätsfunktion weitreichendere Informationen über eine Rolle. In Abschnitt 3.5.3 werden Rollen Komponententypen zugeordnet, so daß sich eine Identität zwischen Rollen ergibt, wenn ihnen derselbe Komponententyp assoziiert wurde.

Schritt 2: Unter Umständen sind die beiden Rollen aus den unterschiedlichen Fachmodellen über eine Generalisierungsbeziehung zusammenzuführen. In dem neu entstehenden Fachmodell ist hierbei die generellere Rolle als abstrakt deklariert. Für die Rollen r_1 und r_2 gilt nun:

$$\begin{aligned} r_1, r_2 \in Role_{m_3} \quad \wedge \quad & \exists d \in Gen_{m_3} : \\ & depends_{m_3}(d) = (r_1, \{r_2\}) \wedge r_2 \in Abstract \\ \vee \quad & depends_{m_3}(d) = (r_2, \{r_1\}) \wedge r_1 \in Abstract \end{aligned}$$

Schritt 3: Sind beide Rollen nicht identisch, markieren jedoch überschneidende Eigenschaften der jeweiligen Entität, so wird der gemeinsame Anteil isoliert und die ursprünglichen Rollen über Aggregationsbeziehungen aus diesem Anteil komponiert. Für die beiden Rollen $r_1 \in Role_{m_1}$, $r_2 \in Role_{m_2}$ gilt nun:

$$\begin{array}{rcl} r_1, r_2 \in Role_{m_3} & \wedge & \exists r', r'', r_3 \in Role_{m_3}. \\ \exists d_1, d_2 \in Agg_{m_3} & : & depends_{m_3}(d_1) = (r_1, \{r', r_3\}) \wedge \\ & & depends_{m_3}(d_2) = (r_2, \{r'', r_3\}) \end{array}$$

Schritt 4: Beschreiben beide Rollen unterschiedliche Eigenschaften der Entität, so können sie vom Entwickler zu einer neuen Rolle zusammengefaßt werden. Dadurch wird insbesondere festgelegt, daß beide Rollen stets gemeinsam von einer Entität gespielt werden müssen.

Zwei Rollen bilden im integrierten Modell eine logische Einheit und sollen deshalb zu einer neuen Rollenspezifikation aggregiert werden. Dadurch wird vor allem festgelegt, daß diese zwei Rollen stets zusammen erfüllt werden müssen. Es gilt:

$$\begin{array}{rcl} r_1, r_2 \in Role_{m_3} & \wedge & \exists d \in Agg_{m_3}, r_3 \in Role_{m_3} : \\ & & depends_{m_3}(d) = (r_3, \{r_1, r_2\}) \wedge r_1, r_2 \in Abstract \end{array}$$

Abbildung 3.8 zeigt die Aggregation der Rollen `Arbeit` und `Verwaltungseinheit` zu einer neuen Rolle `Einreichung`. In dem entstehenden Modell bilden somit beide Rollen eine Einheit. Soll festgelegt sein, daß die Rollen nicht mehr einzeln durch eine Entität erfüllt werden dürfen, so müssen sie noch zusätzlich als abstrakt definiert werden.

Als Ergebnis dieser Schritte entsteht eine neue Menge von Rollen $Role_{m_3}$, die die Charakteristika der beiden ursprünglichen Fachmodelle in sich vereint. Die entsprechenden Assoziationen werden nun einfach direkt übernommen und müssen dabei im Hinblick auf die neuen Rollendefinitionen — wie im Fall der Generalisierung — angepaßt werden.

Assoziationenebene: Nach Erläuterung 3.3 bezeichnet eine Assoziation eine abstrakte Zusammengehörigkeit unterschiedlicher Entitäten. Auch hierbei kann die Integration logische Überschneidungen zwischen den Assoziationen der einzelnen Fachmodelle aufdecken. Diese Redundanz läßt sich — ähnlich zum Vorgehen bei der Rollenintegration — gleichfalls durch die Unterteilung von Assoziationen auflösen. Da unser formales Modell jedoch nicht mächtig genug konzipiert ist, um Beziehungen zwischen Assoziationen zu repräsentieren, werden wir im weiteren diese Varianten der Integration vernachlässigen. Auf der Ebene der Realisierung wird sich diese Entscheidung so auswirken, daß Komponentenframeworks nicht hierarchisch geschachtelt werden. Diese Einschränkung ist im Hinblick auf die rasant ansteigende Komplexität solcher Konstruktionen akzeptabel.

Eine Annahme der letzten Abschnitte bestand darin, daß sich Beziehungen zwischen den zu integrierenden Fachmodellen in Beziehungen zwischen den Rollen der gemeinsamen Gruppe von Entitäten abbilden lassen. Dies ist nur dann möglich, wenn Entitäten auch in beiden Fachmodellen durch entsprechende Rollen repräsentiert sind. Sind Entitäten in einem der Fachmodelle nicht repräsentiert, kann es bei der Integration trotzdem wichtig sein, geeignete Querbezüge einzuführen. Dabei handelt es sich prinzipiell um Assoziationen zwischen Rollen aus beiden Fachmodellen.

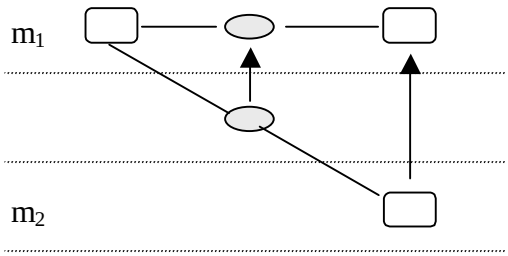


Abbildung 3.13: Vermeidung von Assoziationen zwischen Fachmodellen

Abbildung 3.13 zeigt, daß solche Querbezüge zwischen Fachmodellen aufgelöst werden können, indem eines der Fachmodelle um eine geeignete Abstraktion der spezifischen Entitäten erweitert wird. Dabei kann in vielen Fällen dieselbe Rolle verwendet werden. Andererseits führt dieser Schritt unter Umständen auch zu einer Rolle, die spezifisch für den durch m_1 ausgedrückten Aspekt ist. In diesem Fall müssten die zwei Rollen aus den Modellen m_1 und m_2 nach dem bereits bekannten Verfahren aus den letzten Abschnitten wieder zusammengeführt werden.

3.3.9 Hierarchische Integration

Wie bereits in Abschnitt 3.3.1 erläutert, impliziert jedes Fachmodell eine bestimmte Abstraktion des jeweiligen Anwendungsbereichs. In vorangegangenen Überlegungen zur Integration (vgl. Abschnitt 3.3.7) wurde — wie häufig bei rollenbasierten Spezifikationen [RWL96, Rie00, MH01] — davon ausgegangen, daß es eine (gemeinsame) Dekomposition des Anwendungsbereichs in einzelne Entitäten gibt. Jedes Fachmodell würde hierbei zwar unterschiedliche Sichten auf eine Entität präsentieren — letztlich wäre es jedoch dieselbe Entität.

Wie in der Einleitung bereits angeklungen, ist diese Vereinfachung im Fall unabhängiger Modellentwicklung kaum haltbar. Vielmehr führt die Integration in solchen Fällen

zwangsläufig zu einer Zusammenführung unterschiedlicher Strukturen. Durch die Einschränkung auf Fachmodelle können wir davon ausgehen, daß Entitäten des einen Modells sich in der ein oder anderen Form im anderen Modell wiederfinden, sofern beide Fachmodelle denselben Anwendungsbereich repräsentieren. Nur unter dieser Voraussetzung kann der Lösungsansatz *Modellintegration* dem allgemeineren Ansatz *Modellkoexsistenz* vorgezogen werden (siehe Abschnitt 3.3.7).

Die korrekte Zusammenführung unterschiedlicher Fachmodelle mit ihren jeweiligen Rollen führt damit zu weiteren Abhängigkeiten zwischen den beteiligten Rollen. Neben den Assoziationen zwischen Rollen unterschiedlicher Entitäten desselben Fachmodells und den Abhängigkeiten zwischen unterschiedlichen Rollen derselben Entität führen wir nun eine weitere Beziehung ein, die Rollen unterschiedlicher Entitäten miteinander verknüpft. Um die Auswirkungen dieser Beziehung zwischen Rollen auf die Ebene der Entitäten bestimmten zu können, führen wir eine einfache, partielle Ordnung auf der Menge aller Entitäten ein:

$$\rightsquigarrow \quad : \quad Entity \rightarrow Entity$$

Diese Beziehung repräsentiert auf einfache Weise Aggregationsbeziehungen, wie sie aus dem Bereich der objektorientierten Analyse bekannt sind. Eine Entität e_2 ist somit „Teil“ einer anderen Entität e_1 genau dann, wenn die Relation $e_1 \rightsquigarrow e_2$ gilt.

Gegeben seien nun erneut zwei zu integrierende Fachmodelle m_1 und m_2. Zusammen mit dem bisher untersuchten Fall, bei dem Rollen aus den unterschiedlichen Fachmodellen höchstens von einer gemeinsamen Entität gespielt werden, können ingesamt drei Varianten der Integration unterschieden werden. Dabei seien mit $r_1 \in Role_{m_1}$ und $r_2 \in Role_{m_2}$ die beiden Rollen bezeichnet, die in einer Abstraktionsbeziehung[4] zueinander stehen.

Fall A: Es findet sich eine Entität, auf der sich die Rollen aus beiden Modellen vereinen. Dies war die Voraussetzung der vorangegangenen Abschnitte. Für besagte zwei Rollen r_1 und r_2 gilt in diesem Fall:

$$\exists e \in Entity \quad . \quad r_1, r_2 \in plays_{m_3}(e)$$

Bei der Instantiierung des integrierten Fachmodells findet sich also genau eine Entität, die beide Rollen in sich vereint.

Fall B: Eine der Rollen trifft eine Aussage über eine Repräsentation des Anwendungsbereichs, die die Repräsentation der anderen Rolle umschließt. Weisen wir nun jeder

[4]Um eine begriffliche Verwirrung zu der Aggregation zwischen Rollen zu vermeiden, sprechen wir im Hinblick auf die neu eingeführte Beziehung zwischen Rollen von einer *Abstraktionsbeziehung*.

der Rollen eine eigene Entität bei der Instantiierung des integrierten Fachmodells zu, so muß zwischen diesen Entitäten die folgende Beziehung gelten:

$$\begin{aligned} \exists e_1, e_2 \in Entity \quad . \quad & r_1 \in plays_{m_3}(e_1) \wedge r_2 \in plays_{m_3}(e_2) \\ \wedge \quad & e_1 \rightsquigarrow e_2 \vee e_2 \rightsquigarrow e_1 \end{aligned}$$

Fall C: In diesem Fall gibt es eine Überschneidung der jeweiligen Repräsentationen. Die entsprechenden Entitäten stehen insoweit in Beziehung, als daß es eine gemeinsame weitere Entität gibt, die einen Teil von beiden darstellt.

$$\begin{aligned} \exists e_1, e_2 \in Entity \quad . \quad & r_1 \in plays_{m_3}(e_1) \wedge r_2 \in plays_{m_3}(e_2) \\ \wedge \exists e_3 \in Entity \quad . \quad & e_1 \rightsquigarrow e_3 \wedge e_2 \rightsquigarrow e_3 \end{aligned}$$

Bei der Integration der beiden Fachmodelle aus dem Beispiel der OOPSLA-Konferenz wurden bisher Konflikte aufgrund unterschiedlich gewählter Abstraktionsebenen ausgelassen. Auf diese Weise konnten die jeweiligen Rollen stets sehr direkt analog zu oben erwähntem Fall A miteinander verknüpft werden. Bei der Zusammenführung der Rollen `Autor` und `Gutachter` zeigt sich jedoch ein Unterschied der Modellierung, der eigentlich die Integration in der einfachen Form erschwert. Das Fachmodell für den Review-Prozeß sieht eine Rolle `Gutachter` vor für eine Person, die eine eingereichte Arbeit bewertet und ein entsprechendes Urteil abgibt. In den vergangenen Abschnitten wurde diese Rolle mit der Rolle `Autor` zusammengeführt, da — entsprechend der Aufgabenstellung — Autoren als Gutachter für die OOPSLA herangezogen werden sollen. Hinter der Abstraktion `Autor` kann jedoch in dem Fachmodell zum Einreichungsprozeß eine ganze Gruppe von Autoren stehen, die gemeinsam an einem Werk gearbeitet haben. Die einfache Zusammenführung ergibt nun eine Modellierung, in der *alle* Autoren als Gutachter eingesetzt werden. Auch wenn die Aufgabenstellung an dieser Stelle keine präzisen Vorgaben macht, ist wohl eine dezidierte Auswahl eines Autors als Gutachter wünschenswert.

Bei der Erstellung eines integrierten Fachmodells aus beiden Sichten wird dieser Zusammenhang durch eine Abstraktionsbeziehung hergestellt. Dadurch wird ausgedrückt, daß die Entität der einen Rolle einen Teil der Entität der anderen Rolle darstellt. Im Hinblick auf eine Realisierung wird klar, daß beide Entitäten sich geeignet austauschen müssen, um ihre Rollen erfüllen zu können. So ist es für einen Gutachter essentiell, in Erfahrung bringen zu können, welche Arbeiten von ihm stammen, um die Begutachtung der eigenen Werke ausschließen zu können. Die entsprechende Abstraktionsbeziehung ist in Abbildung 3.14 dargestellt. Der Doppelpfeil ergänzt die bestehenden Notationen von Assoziationen und Abhängigkeiten.

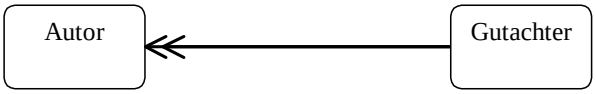


Abbildung 3.14: Abstraktionsbeziehung zwischen Rollen

Im oben aufgeführten Fall C ist es bei der Integration zudem nötig, eine zusätzliche Rolle einzuführen, um die beiden Fachmodelle an dieser Stelle zueinander in Beziehung setzen zu können. Ebenso wie im Fall B sind die aufgestellten Abstraktionsbeziehungen nur ein grober Hinweis auf einen konkreten Abstimmungsbedarf zwischen unterschiedlichen Entitäten des Anwendungsbereichs. Es stellt sich insbesondere die Frage, inwieweit bestehende Bestandteile einer Implementierung für ein integriertes Fachmodell wiederverwendet werden. In späteren Abschnitten werden diesbezüglich genauere Untersuchungen vorgenommen.

Zusammenfassend formulieren wir die zentrale Annahme, die eine Zuordnung von Rollen aus unterschiedlichen Abstraktionen des Anwendungsbereichs ermöglicht:

> *Bei der Integration verschiedener Fachmodelle gibt es stets eine Dekomposition des Anwendungsbereichs, die es erlaubt, eindeutige Zugehörigkeiten von Entitäten zu den definierten Rollen festzulegen.*

Für ein besseres Verständnis der Integration von Fachmodellen mit teils unterschiedlichen Abstraktionen bietet sich ein Schichtenmodell zur Darstellung an, wie es bereits einigen der gezeigten Abbildungen zugrunde liegt. Die Idee besteht dabei darin, die zu integrierenden Fachmodelle übereinander aufzuzeichnen, wobei Rollen derselben Entität an einer gedachten Senkrechten bündig angeordnet werden. Abbildung 3.15 zeigt drei Fachmodelle, die eine jeweils eigene Sicht auf dieselbe Menge der Entitäten repräsentieren. Unterschiedliche Abstraktionen sind in dieser Abbildung teilweise durch gestrichelte Linien deutlich gemacht.

So intuitiv diese Darstellung die Integration von Fachmodellen verdeutlicht, so wichtig erscheint die Betonung des folgenden Sachverhalts: Die Sortierung der einzelnen Schichten bedeutet keinesfalls eine absteigende oder aufsteigende Sortierung der Abstraktionsniveaus. Die Existenz einer solchen Reihenfolge ist überhaupt selten gegeben, da Fachmodelle in unterschiedlichen Abschnitten des Anwendungsbereichs ganz unterschiedliche Abstraktionen vornehmen können. Die Darstellung ist daher als Hilfsmittel zu sehen, um die Zusammenführung von Rollen bei der Modellintegration zu erleichtern und ein geeignetes Verständnis zu vermitteln. Insbesondere lassen sich auf diese Weise Abhängigkeiten zwischen den einzelnen Rollen nur mühsam zeigen.

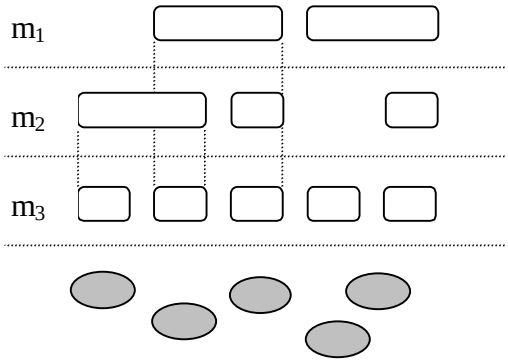


Abbildung 3.15: Fachmodelle in Schichten

3.3.10 Beispiel

In diesem Abschnitt werden die bisherigen Erkenntnisse der Modellintegration anhand der Konferenzapplikation der OOPSLA (vgl. Abschnitt 3.2) angewendet und demonstriert. Im Rahmen der Anforderungen wurden bereits zwei Sichten identifiziert: einerseits agiert die Anwendung als eine besondere Form von Datenbank, in die Dokumente abgelegt und ausgelesen werden können und andererseits wird ein verteilter Ablauf für Reviews von Dokumenten realisiert, der die Auswahl für die Konferenz bestimmt.

Die beiden Fachmodelle für diese Sichten werden unabhängig voneinander spezifiziert und bieten ein Abstraktionsniveau, das die Wiederverwendung der jeweiligen Spezifikation auch in anderen Anwendungen ermöglicht. Konkret bedeutet das beispielsweise, daß die *remote repository*-Sicht beliebige Datenformate speichern kann, auch wenn in der konkreten Ausprägung nur Dokumente (z.B. mit Microsoft Word erstellt) verwaltet werden müssen. Beide Fachmodelle basieren auf unabhängig voneinander analysierten Anwendungsbereichen. Die Modellintegration hat ihr Ziel in einem gemeinsamen Verständnis des Anwendungsbereichs, der insbesondere das Zusammenwirken der beiden Sichten festhält.

Abbildung 3.16 zeigt das Fachmodell der *remote repository*-Sicht. Dabei können über eine geeignete `Kennung` ein oder mehrere `Verwaltungseinheiten` in einem `Repository` gespeichert und bei Bedarf auch wieder ausgelesen werden. Die Verwaltung des Repositories (insbesondere das Einrichten neuer Kennungen) übernimmt ein `Verwalter`. So lassen sich zwei Beziehungen unterscheiden: Einerseits faßt `Liest/Schreibt` die Vorgänge Einstellen, Auslesen und Löschen eines Dokuments zusammen und andererseits bezeichnet `organisiert` die Verwaltung des Repositories. Da der Verwalter den Dokumentenbestand organisiert, ist ihm ebenfalls eine Kennung zugeordnet und die Rolle `Verwalter` ist von `Kennung` abgeleitet.

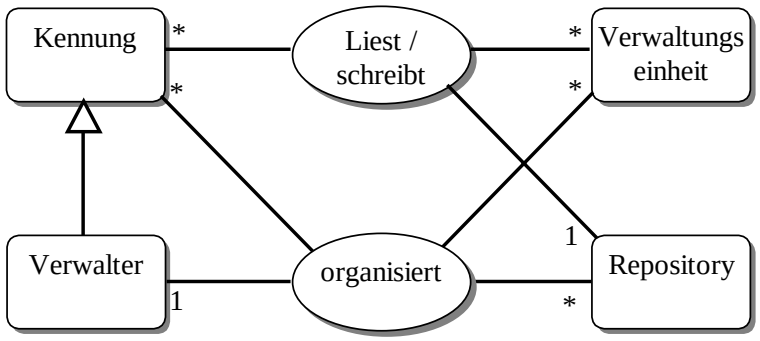


Abbildung 3.16: Fachmodell für ein Repository

Das Fachmodell für die zweite Sicht entspricht dem Verständnis eines allgemeinen Review-Mechanismus. Hierbei sind unterschiedliche Abläufe denkbar, eine einfache Modellierung findet sich in Abbildung 3.17. Eine Reihe von `Gutachtern` bilden sich Meinungen über eine Menge von `Dokumenten` und verfassen entsprechende `Kommentare`. Die Auswertung aller Kommentare erlaubt es, eine Reihenfolge der Dokumente aufzustellen, wobei die Sortierung die zugrundegelegten Qualitätskriterien der Gutachter wiederspiegelt.

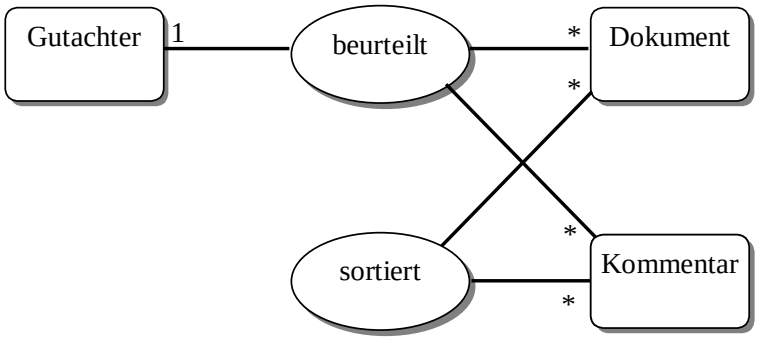


Abbildung 3.17: Fachmodell des Reviewprozesses

Die Integration der beiden Fachmodelle beginnt mit Überlegungen zur Zusammenführung der einzelnen Rollen und Assoziationen. Würden bei den jeweiligen Bezeichnern Konflikte auftreten, müsste zuerst eine geeignete Umbenennung vorgenommen werden. Auch der hier vorgestellte Fall zeigt — entsprechend der Annahme aus Abschnitt 3.3.8 — keine Konflikte zwischen den Assoziationen, so daß im weiteren die Zusammenführung auf der Ebene der Rollen adressiert wird.

Hierbei fallen weitere Entwurfsentscheidungen an: der Entwickler legt zum Beispiel fest, daß jeder `Gutachter` eine `Kennung` benötigt, um auf Dokumente zuzugreifen und diese im Repository wieder abzulegen. Zudem sind im zusammengeführten Modell Dokumente die zu speichernden Daten und so muß zwischen diesen beiden Rollen eine Kopplung hergestellt werden.

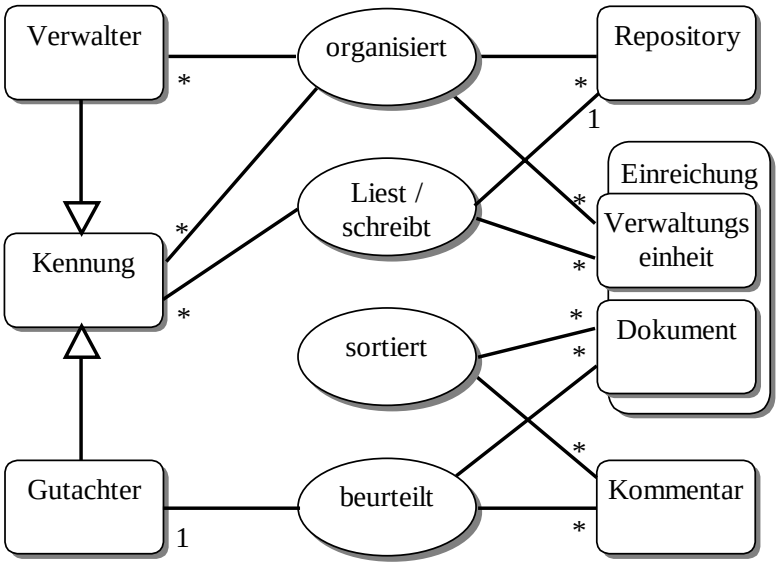


Abbildung 3.18: Integrierte Fachmodelle

Abbildung 3.18 zeigt das integrierte Fachmodell, in dem die Rollen `Verwaltungseinheit` und `Dokument` zu einer neuen Rolle `Einreichung` aggregiert wurden. Die entsprechende Entwurfsentscheidung hätte auch vorweggenommen werden können, wenn die Spezifikation des Fachmodells für den Reviewprozeß bereits die Existenz einer Rolle `Verwaltungseinheit` angenommen hätte, und diese Rolle als Anforderung der Rolle `Dokument` gestaltet hätte. Über solche Querbezüge zwischen den Rollen einer Entität kann somit der Integrationsprozeß vereinfacht werden.

Dieses einfache Beispiel zeigt eine flache Integration, da sich stets gemeinsame Entitäten finden lassen, die die eingeführten Rollen spielen können (vgl. Fall A aus Abschnitt 3.3.9). Ein Konflikt zeigt sich bei einer genaueren Untersuchung der Anwendung des *remote repository*-Fachmodells: Die Rolle `Kennung` repräsentiert hier den Urheber eines Dokuments, insbesondere die Gruppe der Autoren. Die Behandlung aller Autoren als eine Entität erweist sich jedoch als nachteilig, wenn einzelne Autoren auch als Gutachter eingereichter Arbeiten agieren sollen. In diesem Fall entsteht ein Konflikt aus der Repräsentation von Autorengruppen in der einen Sicht und der Repräsentation einzelner Autoren in

der anderen. Über eine Abstraktionsbeziehung läßt sich diese Überbrückung unterschiedlicher Abstraktionsebenen im Modell festhalten (vgl. Abschnitt 3.3.9).

3.3.11 Zusammenfassung

In den vorangegangenen Abschnitten wurde eine abstrakte, fachliche Modellierung von Anwendungsbereichen vorgestellt und diskutiert. Fachmodelle vermitteln ein Verständnis der Konzepte und Zusammenhänge und ermöglichen damit die Untersuchung essentieller Fragestellungen möglicher Anwendungssysteme. Einzelne Fachmodelle sind in dieser Hinsicht vergleichbar mit konzeptuellen (Daten-)Modellen, wie sie z.B. über Entity-Relationship-Modelle repräsentiert werden (vgl. [Vet91, Tha00, RWL96]). Über Fachmodelle ist festgelegt, was ein System leisten soll (und was nicht), sowie die Menge der Eigenschaften, die erfüllt werden müssen. In dieser Zielsetzung zeigt sich auch eine Nähe mit der objektorientierten Analyse (kurz: OOA) [Mey97]. Das Konzept der Rolle ist eng verwandt mit den hier eingesetzten Klassen und beide bieten die Möglichkeit, Abstraktionen des Anwendungsbereichs zu schaffen.

Im Vergleich mit bekannten Prinzipien der OOA wurde in den vergangenenen Abschnitten jedoch nur ein kleiner Schwerpunkt auf die Modellierung von Beziehungen zwischen unterschiedlichen Entitäten gelegt. Stattdessen führten wir das sehr allgemeine Konzept der Assoziation ein, das diverse Entitäten verknüpft ohne detaillierte semantische Aussagen zu treffen. Der Fokus liegt vielmehr auf der Zerlegung einer Entität in mehrere Rollen und der Spezifikation des Zusammenspiels zwischen diesen Rollen. Insbesondere der Schritt von einer klassischen Blackbox-Betrachtung der beteiligten Entitäten hin zu einem detaillierteren Verständnis über den internen Aufbau einer einzelnen Rolle geht über die klassische Analyse hinaus.

Letztlich werden auf diese Weise bereits Details des Modells festgelegt, die für die konkret zu modellierende Sicht noch nicht bedeutsam sind (es also überspezifizieren). Wenn der Entwickler beschreibt, daß die Rolle `Dokument` auf einer Rolle `Verwaltungseinheit` aufbaut, so zeigt sich der Wert dieser Information erst bei der Integration des Fachmodells mit anderen Sichten. Vom Entwickler erfordert die Erstellung solcher „integrationsoffener“ Modelle Weitblick, Erfahrung und die Fähigkeit, spätere Integrationswünsche abschätzen zu können. Nur auf diese Weise gelingt es, den Zeitpunkt der Integration sehr spät nach der Erstellung der Fachmodelle anzusiedeln und die Integration mit einem geringen Bedarf an zusätzlichen Entwurfsentscheidungen durchzuführen.

In den frühen Phasen der Entwicklung eines Anwendungssystems wird nun versucht, bestehende Fachmodelle auf den Anwendungsbereich abzubilden und sie darüber miteinander zu integrieren. Neben dieser Wiederverwendung auf der Spezifikationsseite ist ein

essentielles Ziel die Wiederverwendung von Artefakten der Realisierung. Wie auf den nächsten Seiten gezeigt wird, werden Fachmodelle als „Schnittstelle“ zugehöriger Komponentenframeworks gehandelt und idealerweise erfolgt die Integration auf der Realisierungsebene analog zu den Vorgaben der fachlichen Modellierung.

Im weiteren betrachten wir die Überführung eines Fachmodells in Bestandteile eines Anwendungssystems. Auf dieser Ebene können eine Reihe weitergehende Fragestellungen untersucht werden, beispielsweise im Hinblick auf mögliche Konflikte bei der gleichzeitigen Erfüllung mehrerer Rollen durch eine Entität. Diese Überführung wird als Abbildung zwischen einem Fachmodell und einem Systemmodell (siehe Abschnitt 3.4.1) verstanden. Dabei sind die folgenden Eigenschaften wichtig:

Modellnähe: In der gängigen Praxis der Softwareentwicklung weist die Modellierung als Ergebnis der Analysephase Strukturen auf, die sich im entstehenden Anwendungssystem in dieser Form nicht zwangsläufig wiederfinden (vgl. [BRS97]). Eine solche Freiheit bei der Realisierung führt jedoch dazu, daß die Integration zweier Fachmodelle nur mühsam in den jeweiligen Realisierungen nachgezogen werden kann. Wir fordern aus diesem Grund, daß die Konzepte und Zusammenhänge in der Implementierung erhalten bleiben, insbesondere also eine Nähe zwischen Fachmodell und Systemmodell gegeben ist.

Wiederverwendbarkeit: Die diversen Artefakte einer Realisierung eines Fachmodells müssen als Einheit handelbar, vergleichbar und nutzbar sein.

Integrierbarkeit: Auch unabhängig voneinander entwickelte Module dieser Realisierung müssen unter bestimmten Gegebenheiten miteinander verbunden werden können zu einem lauffähigen Anwendungssystem. Dies erfordert möglicherweise die präzise Definition der einzelnen „Schnittstellen“ zwischen diesen Modulen.

Die bisherigen Ergebnisse repräsentieren die oberste Schicht der Modellierung, wie sie in diesem Kapitel aufgebaut wird. In den folgenden Abschnitten diskutieren wir die unterste Schicht, eine allgemeine Repräsentation komponentenbasierter Systeme. Die verknüpfende Schicht wird anschließend in Abschnitt 3.5 präsentiert.

3.4 Modellierung komponentenbasierter Systeme

Viele Jahre war der Begriff der Komponente geprägt durch eine Uneinigkeit darüber, was die essentiellen Konzepte ausmacht und wie sich eine Komponente vor allem von

der inzwischen gut verstandenden Idee des Objekts unterscheidet. So gibt es eine Fülle verschiedener Begriffsdefinitionen, die von ganz unterschiedlichen Voraussetzungen ausgehen und meist bestimmte Charakteristika betonen (siehe [Szy97]). In diesem Abschnitt stellen wir ein Modell komponentenbasierter Systeme vor, das unser über Jahre entwickeltes Verständnis von Komponenten wiedergibt. Die zugrundeliegende, formale Konzeption ist eine verfeinerte Variante unserer Arbeit im Buch „Foundations of Component-Based Systems“ [BRS+00]. Diese wiederum ist angelehnt an die *FOCUS*-Methodik, die am Lehrstuhl von Prof. Broy entstand und seither kontinuierlich weiterentwickelt wurde (vgl. beispielsweise [BDD+92, BS01]). FOCUS ist spezialisiert auf die Entwicklung verteilter, reaktiver Systeme. Die folgende Erläuterung präzisiert den Begriff des Systemmodells (nach [Rum96]):

Erläuterung 3.11 (Systemmodell) Ein *Anwendungssystem* ist eine konkrete Ausprägung einer Kombination aus Softwareeinheiten, die eine bestimmte Struktur besitzen und ein bestimmtes Verhalten aufweisen. Ein *Systemmodell* ist eine Charakterisierung aller für uns interessanten Anwendungssysteme. Es charakterisiert sowohl das Verhalten als auch die Struktur eines Systems, läßt aber Freiheitsgrade, die durch die konkrete Systementwicklung spezialisiert werden.

Die konkrete Zielsetzung besteht darin, die grundlegenden Konzepte komponentenbasierter Systeme prägnant zu erfassen und zueinander in Beziehung zu setzen.

3.4.1 Grundlagen

Entsprechend den in Abschnitt 3.1 gestellten Anforderungen modellieren wir komponentenbasierte Systeme als eine Menge individueller, interaktiver Instanzen, genannt *Komponenten*. Komponenten weisen *Schnittstellen* zu ihrer Umgebung auf und können über diese Schnittstellen mit anderen Komponenten interagieren. Schnittstellen unterschiedlicher Komponenten sind über *Kanäle* miteinander verbunden und gestatten somit Kommunikation zwischen Komponenten durch den Austausch von Nachrichten. Kanäle erhalten hierbei die Reihenfolge der zu übertragenden Nachrichten und werden zudem als bidirektional definiert.

Das beobachtbare Verhalten einer Komponente ist festgelegt über die Menge der empfangenen und gesendeten Nachrichten. Bestimmte Komponenten mit vergleichbarer Charakterisitik hinsichtlich ihres Verhaltens gruppieren wir zu *Komponententypen* und sagen, die entsprechenden Komponenten seien eine Instanz von diesem Typ. In Abschnitt 3.5 wird gezeigt, daß Komponententyp und Schnittstelle ergänzende Konzepte sind.

Ähnlich zu den Arbeiten in [KRB96] modellieren wir die Dynamik eines Systems anhand einer unendlichen Zeitachse, die aus einer Reihe diskreter Zeitintervalle gleicher Länge besteht. Wir definieren die Menge $\mathbb{N}^+$ als abstrakte Zeitachse und kürzen sie der Einfachheit und Eindeutigkeit halber mit T ab. Das zugrundeliegende Zeitmodell sei synchron — es existiert damit eine globale Uhr, die für alle Teile des Systems verbindlich ist.

Erläuterung 3.12 (Gezeitete Ströme) Eine unendliche Reihung von Elementen einer Menge X heißt unendlicher, *gezeiteter Strom* über X, wenn jedem Zeitintervall aus T genau ein Element der Menge X zugeordnet wird. Jeder gezeitete Strom ist somit ein Element vom Typ $X^T =_{def} T \rightarrow X$. Analog definiert sich ein endlicher, gezeiteter Strom als eine Abbildung über einem Intervall von T.

Für einen Strom $x \in X^T$ bezeichnet die Notation $x \downarrow t$ den endlichen Präfix aus allen Elementen der ersten t Zeitintervalle. Weiterhin wird für die Auswertung eines Stroms x zu einem bestimmten Zeitpunkt t die Abkürzung x^t verwendet. Diese Operatoren lassen sich durch die elementweise Anwendung einfach auf Mengen von Strömen verallgemeinern.

Gezeitete Ströme werden im Weiteren verwendet, um Nachrichtenströme zwischen Komponenten zu repräsentieren. Dazu werden alle Nachrichten eines Zeitintervalls zu einer Nachrichtensequenz zusammengefaßt. Sei M die Menge aller Nachrichten, die zwischen den Komponenten versendet werden können. Mit M^* bezeichnen wir nun die Menge aller endlichen Nachrichtensequenzen und somit ist jeder unendliche, gezeitete Nachrichtenstrom ein Element der Menge $(M^*)^T$. Konkrete Nachrichtensequenzen werden mit spitzen Klammern dargestellt. So ist beispielsweise $\langle a \rangle$ eine Sequenz aus der Nachricht a, $\langle a, b, c \rangle$ die Sequenz aus a, b und c und die leere Sequenz wird durch $\langle\rangle$ repräsentiert. Für die Konkatenation zweier Sequenzen wird die Schreibweise $p \hat{} q$ eingeführt, wobei p und q zwei Nachrichtensequenzen darstellen.

3.4.2 Zustand

Komponenten sind die individuellen, operationellen Einheiten eines Systems. Es macht Sinn anzunehmen, daß Komponenten einen internen, gekapselten Zustand haben, der entsprechend dem der Komponente zugewiesenen Verhalten über die Zeit verändert wird. Soll der Zustand einer Komponente verändert werden, so ist es erforderlich, über eine geeignete Schnittstelle mit der Komponente zu kommunizieren, um sie eine Zustandsänderung durchführen zu lassen. Durch diese Kapselung der internen Repräsentation eines Zustands durch eine geeignete Schnittstelle — ein essentielles Konzept aus der Objektorienierung — ist eine explizite Modellierung des Zustands nicht erforderlich. Stattdessen

kann aus dem Nachrichtenstrom an den Schnittstellen der Komponente auf ihren jeweiligen Zustand geschlossen werden. Einen solchen Präfix eines Nachrichtenstroms bis zu einem festgelegten Zeitpunkt nennen wir im folgenden auch *Historie*.

Ein *Snapshot* eines Systems bezeichnet den vollständigen Zustand des Sytems zu einem bestimmten Zeitpunkt $t \in T$. Die Menge aller Kommunikationshistorien an den Schnittstellen der Komponenten repräsentiert dabei nur unzureichend den Zustand des Gesamtsystems, da nur der interne Zustand der Komponenten berücksichtigt ist. Um das System vollständig zu beschreiben, sind noch weitere Informationen nötig:

1. Nachrichten, die noch auf den Kanälen unterwegs sind. Diese Problematik kann vermieden werden durch die Festlegung, daß Kanäle nicht puffern, die Nachrichten also insbesondere ohne Verzögerung übertragen werden.
2. Menge der Komponenten im System. Während dem Ablauf eines Systems ist zu erwarten, daß neue Komponenten in das System mit aufgenommen werden bzw. bestehende Komponenten das System verlassen.
3. Aktuelle „Verschaltung“ der Komponenten. Die Menge der Kanäle und ebenfalls die jeweils verbundenen Schnittstellen können über die Zeit variieren.

Bei den Punkten 2. und 3. handelt es sich um einen Zustand des Systems, der für alle Komponenten sichtbar ist. Eine Komponente muß in der Lage sein, diesen Zustand zu beeinflussen oder auf jeweilige Veränderungen geeignet zu reagieren. An dieser Stelle stellt sich die Frage, ob eine ähnliche Kapselung dieses Zustands wie beim internen Zustand von Komponenten möglich ist. Beispielsweise besteht eine sehr allgemeiner Ansatz in einer zentralen *Konfigurationskomponente*, die die gesamte Systemkonfiguration verwaltet und Veränderungen kanalisiert (vgl. [DMNS97]). Eine geeignete Lösung dieser Problemstellung ist essentiell, um die Kompositionalität des Systemmodells zu gewährleisten. Vereinfachend berücksichtigen wir im weiteren dabei nur die flexible Veränderung von Kanälen zwischen Komponenten.

3.4.3 Schnittstellen und Komponenten

Jeder Instanz in einem System wird ein eindeutiger Identifikator zugeordnet. Dazu sind entsprechende Bezeichnermengen definiert und zwar $Component$, $Interface$ und $Channel$ für die Menge aller Komponenten, Schnittstellen und Kanäle. Jeder Komponente sind eine Reihe von Schnittstellen zugeordnet und je zwei Schnittstellen können mit einem Kanal verbunden sein:

$$\begin{aligned} assigned &: Interface \rightarrow Component \\ connected &: Channel \rightarrow \{\{i,j\} \mid i,j \in Interface \wedge i \neq j\} \end{aligned}$$

Die Abbildung $assigned$ wird als total definiert, um sicherzustellen, daß es keine Schnittstelle ohne zugeordnete Komponente gibt. Zudem ist festgelegt, daß Kanäle eine Schnittstelle nicht mit sich selbst verbinden dürfen (wohl aber mehrere Schnittstellen derselben Komponente).

Ein entscheidendes Konzept im Rahmen dieser Arbeit ist das der hierarchischen Schachtelung von Komponenten. Die Idee ist hierbei, daß eine Komponente aus mehreren anderen Komponenten bestehen kann und diese damit gegenüber anderen Komponenten abkapselt. Auf diese Weise können Komponentengruppen einfach zu einem Verbund zusammengefaßt werden. Diese Beziehung zwischen Komponenten wird durch die partielle Abbildung $parent$ modelliert, die jeder Komponente ihre Vaterkomponente zuordnet, sofern eine solche existiert.

$$parent \;:\; Component \rightarrow Component$$

Komponenten, für die $parent$ nicht definiert ist, sind auf oberster Ebene angesiedelt und haben demzufolge keine eigene *Vaterkomponente*. Von der Abbildung $parent$ sei gefordert, daß sie über der Menge der Komponenten einen gerichteten, azyklischen Graphen definiert, so daß atypische Konstruktionen (eine Komponente enthält sich selbst) ausgeschlossen sind. Auf die entsprechende formale Festlegung sei hier verzichtet. Entscheidend ist zudem die Auflage, daß nur Komponenten mit derselben Vaterkomponente Kanäle untereinander aufbauen dürfen.

$$\begin{aligned} &\forall cn \in Channel; c,d \in Component; i,j \in Interface : \\ &connected(cn) = \{i,j\} \wedge assigned(i) = c \wedge assigned(j) = d \Rightarrow \\ &parent(c) = parent(d) \end{aligned}$$

Abbildung 3.19 zeigt die Komponente K als Vaterkomponente der Komponenten SK_1, SK_2 und SK_3 und eine Kanal C, der je eine Schnittstelle von SK_1 und SK_2 miteinander verbindet. Die Komponente K kann zudem mit anderen Komponenten derselben Ebene über ihre Schnittstellen s_1 bis s_5 kommunizieren.

Die Funktionaltät der Komponente K wird auf diese Weise durch das Zusammenspiel ihrer Subkomponenten realisiert. Ein einfacher Weg, diesen Zusammenhang auszudrücken, ist eine direkte Abbildung der Schnittstellen der Vaterkomponente auf die Schnittstellen der Söhne.

$$\begin{aligned} maps &: Interface \rightarrow Interface \\ \forall i,j \in Interface &: maps(i) = j \Rightarrow \\ &\quad parent(assigned(j)) = assigned(i) \end{aligned}$$

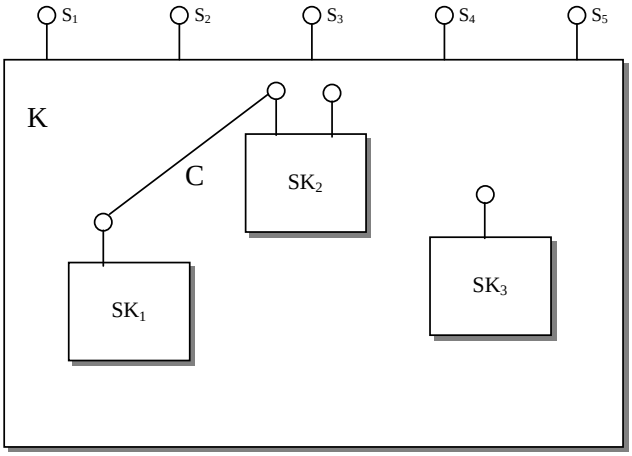


Abbildung 3.19: Ein Komponentendiagramm

Ähnlich zum Ansatz in FOCUS sind Vaterkomponenten primär ein Hilfskonstrukt, um das Zusammenwirken mehrerer Komponenten so zu beschreiben, als wäre es eine einzelne (vgl. [Bro98]). Auf den ersten Blick führt diese Konzeption zu zwei Nachteilen: Einerseits sollen Vaterkomponenten *aktiv* ihre jeweiligen Söhne nutzen, um ihre spezifizierte Funktionalität zu erreichen („Das Ganze stellt mehr als die Summe seiner Teile dar.“). Und andererseits erfordert die oben eingeführte Abbildung zwischen Schnittstellen die Suche nach genau einer verantwortlichen Sohnkomponente, die für eine Schnittstelle des Vaters verantwortlich ist. Dies erscheint als eine eher unhandliche Einschränkung.

Beide Probleme werden durch die Einführung einer *Gluekomponente* adressiert (siehe auch [DW98]). Mit dem Begriff *glue* verbinden wir intuitiv die Vorstellung eines verknüpfenden Stoffes, der bestehende Komponenten zu einer neuen Komponente zusammenführt. Die explizite Kapselung in einer Gluekomponente ermöglicht die genaue Zuordnung der Schnittstellen: jeder hierarchisch zerlegten Komponente ist genau eine Gluekomponente zugeordnet, die mindestens dieselbe Schnittstellentypen aufweist. Über weitere Schnittstellen ist die Gluekomponente — nach Bedarf — mit den Subkomponenten der Vaterkomponente verschaltet. Die Gluekomponente repräsentiert somit den Kern der Vaterkomponente und das Wissen über die korrekte Delegation zu den Subkomponenten.

$$\begin{aligned} glue &: Component \rightarrow Component \\ \forall i, j \in Interface &: maps(i) = j \Rightarrow \\ &\quad glue(assigned(i)) = assigned(j) \end{aligned}$$

Mit der partiellen Abbildung $glue$ können wir einer Komponente eine ausgezeichnete Gluekomponente zuweisen und festlegen, daß alle Schnittstellen der Vaterkomponente auf die Schnittstellen der zugeordneten Gluekomponente abgebildet werden (über $maps$).

In vielen Fällen reicht es zum Verständnis aus, die Gluekomponente nicht explizit zu modellieren, sondern stattdessen anzunehmen, daß die „offenen“ Schnittstellen der beteiligten Komponenten für die Realisierung der Vaterkomponente herangezogen werden. Mit dieser Regelung kann Abbildung 3.19 so interpretiert werden, daß die freien Schnittstellen der Komponenten SK_2 und SK_3 für die Realisierung der Funktionalität der Vaterkomponente eingesetzt werden.

Die bisher eingeführten Konzepte erlauben nun eine präzise Definition von Komponenten, wie sie im weiteren verstanden und genutzt werden:

> **Erläuterung 3.13 (Komponente)** Eine Komponente ist eine individuelle, operationelle Einheit eines Systems mit einem eindeutigen Identifikator. Jeder Komponente sind Schnittstellen zugewiesen, über die sie mit ihrer Umgebung (also mit anderen Komponenten) interagiert. Eine Komponente kann in weitere Komponenten zerlegt werden, die entweder direkt oder über eine Gluekomponente zusammenwirken. Die Gluekomponente realisiert zudem das Verhalten der Vaterkomponente im Zusammenspiel mit den hierzu benötigten Subkomponenten.
>
> Als *blackbox*-Verhalten einer Komponente werden die beobachtbaren Kommunikationsflüsse an den Schnittstellen der Komponente bezeichnet. Demgegenüber beinhaltet das *glassbox*-Verhalten zusätzlich den Nachrichtenverkehr an den Schnittstellen interner Komponenten und Veränderungen des internen Zustands, beispielsweise hinsichtlich der Menge der internen Komponenten bzw. deren Verschaltung.

In jedem Zeitintervall wird über jeden Nachrichtenkanal genau eine Nachrichtensequenz versendet — unter Umständen auch eine leere Sequenz. Komponenten sind *reaktiv*: sie akzeptieren zu jedem Zeitpunkt ankommende Nachrichten und blockieren auf diese Weise keinen Sender von Nachrichten. Schnittstellen agieren hierbei als Multiplexer bzw. Demultiplexer: Ankommende Nachrichtensequenzen auf den diversen Kanälen einer Schnittstelle werden zu einer Sequenz gemischt und ausgehende Nachrichtensequenzen werden auf jeden Kanal kopiert. Auf diese Weise werden individuelle Kommunikationspartner an derselben Schnittstelle nicht unterschieden.

Ein einfaches Beispiel verdeutlicht das Verhalten einer Schnittstelle: werden die drei Nachrichtensequenzen $\langle a \rangle$, $\langle b \rangle$ und $\langle c \rangle$ auf den drei Kanälen einer Schnittstelle empfangen, so wird beispielsweise die Sequenz $\langle b, c, a \rangle$ an ihre Komponente weitergeschickt. Umgekehrt wird eine Nachricht $\langle m, n \rangle$ von der Komponente in dieser Form auf alle verbundenen Kanäle geschrieben.

Mit dem Begriff *Auswertung* einer Schnittstelle zu einem Zeitpunkt $t \in T$ bezeichnen wir die zwei konkreten Nachrichtensequenzen in_i^t und out_i^t, die die Eingabe und Ausgabe einer Schnittstelle ausmachen. Ein Nachrichtenkanal benötigt für den Transfer einer Nachrichtensequenz genau einen Takt und so gilt:

$$\begin{aligned} In &=_{def} Interface \rightarrow (M^*)^T \\ Out &=_{def} Interface \rightarrow (M^*)^T \\ \forall i_1, i_2 \in Interface, cn \in Channel &: connected(cn) = \{i_1, i_2\} \Rightarrow \\ \forall t \in T, \exists p_1, p_2, q_1, q_2 \in M^* &: in_{i_1}^{t+1} = p_1 \hat{} out_{i_2}^t \hat{} q_1 \wedge in_{i_2}^{t+1} = p_2 \hat{} out_{i_1}^t \hat{} q_2 \end{aligned}$$

Durch diese Definition ist festgelegt, daß die Nachrichten erhalten bleiben, ohne auf einen genauen Mechanismus des Zusammenführens in der Schnittstelle einzugehen[5]. Beispielsweise ist keine bestimmte Reihenfolge der einzelnen Nachrichten in der zusammengeführten Nachrichtensequenz festgelegt. Auswertungen einer Schnittstelle bezeichnen wir als *Kommunikationshistorien*. Für die Charakterisierung des Kommunikationsverhaltens einer Komponente, also insbesondere eines konkreten Ablaufs, werden die Kommunikationshistorien der eingehenden Nachrichten an den Schnittstellen auf die der ausgehenden Nachrichten abgebildet. Diese Betrachtung führt zu einer Verhaltensspezifikation in der Blackbox-Sicht einer Komponente. Hierbei sind Veränderungen im internen Aufbau der Komponente nicht von Interesse.

Um das Verhalten einer Komponente nicht nur im Bezug auf konkrete Abläufe, sondern für alle sinnvollen Abläufe zu konkretisieren, führen wir *Komponententypen* ein. Die Menge der sinnvollen Abläufe stellt insbesondere eine Einschränkung der Nachrichtenarten dar, die eine Schnittstelle empfangen oder versenden kann. Schnittstellen agieren deshalb nicht nur als Mittel der Verschaltung sondern gleichfalls als Zugangspunkt für Funktionalitäten der Komponente. Die im Umfeld von Komponententechnologien gängigen Sprachen für die Definition von Schnittstellen (z.B. CORBA IDL [OH98]) erlauben lediglich die Festlegung syntaktischer Eigenschaften. Obwohl heutzutage die Notwendigkeit für mit semantischen Informationen angereicherte Schnittstellen erkannt wurde [HHG90], bietet dieses traditionelle Schnittstellenverständnis die Möglichkeit zur eleganten Trennung von Syntax und Semantik. So führen wir in diesem Modell *Schnittstellentypen* ein, um lediglich die Art der zu empfangenden oder zu versendenden Nachrichten festzulegen. Diese Schnittstellentypen werden von Komponententypen referenziert, um das Verhalten einer Komponente zu definieren.

Ausgehend von der Menge $IType$ aller Schnittstellentypen und $CType$ aller Komponententypen weisen wir Schnittstellen und Komponenten jeweils genau einen Typ zu:

$$itypeOf \quad : \quad Interface \rightarrow IType$$

[5] Die Schreibweise in_i^t ist eine Abkürzung für $in(i,t)$

$$\begin{aligned} ctypeOf &: Component \rightarrow CType \\ inMsg, outMsg &: IType \rightarrow \wp(M) \\ uses &: CType \rightarrow \wp(IType) \end{aligned}$$

Die Abbildungen $inMsg$ und $outMsg$ definieren die Arten von Nachrichten, die eine Schnittstelle eines Typs empfangen bzw. versenden kann. Über $uses$ sind einem Komponententyp unterschiedliche Schnittstellentypen zugeordnet, über die sich das Verhalten definiert. Obwohl es moderne Komponententechnologien, wie das CORBA Komponentenmodell [CCM99] ermöglichen, mehrere Schnittstellen desselben Typs einer Komponente zuzuordnen, beschränken wir das Modell auf maximal eine solche Schnittstelle, vor allem um im weiteren einen einfacheren Umgang mit dem mathematischen Modell zu ermöglichen.

Das Blackbox-Verhalten einer Komponente ergibt sich nun durch eine Relation der eingehenden und ausgehenden Nachrichtenströme:

$$\begin{aligned} compBeh_{bb} &: CType \rightarrow (\mathbb{N} \rightarrow (IType \rightarrow (M^*)^T \times (M^*)^T)), \quad \text{wobei} \\ &\forall ct \in CType, it \in IType, b \in compBeh_{bb}(ct), n \in \mathbb{N} : \\ &b(n)(i) \in (inMsg(itypeOf(i))^T \times (outMsg(itypeOf(i))^T) \end{aligned}$$

Ein Komponententyp besteht aus allen Varianten an Ein- und Ausgabehistorien an den jeweiligen Schnittstellen, die durch ihren Schnittstellentyp bezeichnet sind. Für eine leichtere Handhabung werden die einzelnen Varianten anhand der Menge $\mathbb{N}$ aufgezählt. Durch die Relation zwischen den Historien ist ein nichtdeterministisches Verhalten der Komponenten möglich. Die nachfolgende Bedingung sichert zu, daß die Kommunikationshistorien, die das Verhalten definieren, konform zu den Schnittstellentypen der Komponente definiert sein müssen (also nur erlaubte Nachrichtenarten enthalten).

Ist nun ein konkreter Ablauf eines Komponentensystems durch $in \in In$ und $out \in Out$ gegeben, so kann mit der folgenden Bedingung überprüft werden, ob sich die einzelnen Komponenten ihrem jeweiligen Komponententyp entsprechend verhalten.

$$\begin{aligned} &\forall c \in Component, i \in Interface, n \in \mathbb{N}, assigned(i) = c : \\ &(in_i, out_i) \in compBeh_{bb}(n)(ctypeOf(c)) \end{aligned}$$

Die Betrachtung des internen Aufbaus einer Komponente und möglicher Veränderungen desselben führt zu der *Glassbox*-Sicht. Hier muß die Spezifikation eines Komponententyps zusätzliche Aussagen über die Menge der enthaltenen Komponenten treffen und bestimmen, wie das Zusammenspiel der einzelnen Komponenten stattzufinden hat. Die einzelnen Bestandteile sind wie folgt charakterisiert:

- Im Komponententyp werden weitere Komponententypen referenziert, um den internen Aufbau zu charakterisieren. Um das Modell zu vereinfachen, ist an dieser Stelle — ebenso wie zuvor bei den Schnittstellen — festgelegt, daß von jedem Komponententyp nur eine Instanz in einer Komponente existieren darf. Das dynamische Ein- bzw. Ausgliedern von Komponenten wäre zwar eine wünschenswerte Möglichkeit, trägt aber nur unmaßgeblich zu den hier entwickelten Konzepten bei.

$$composedOf \quad : \quad CType \rightarrow \wp(CType)$$

- Die gekapselten Komponenten einer Vaterkomponente können nach Bedarf Kanäle zueinander auf- oder auch wieder abbauen. Aufgrund der hierarchischen Kapselung durch Komponenten des Systems wird dieser Zustand der Systemkonfiguration aufgeteilt auf die durch die jeweiligen Vaterkomponenten vorgegebenen Bereiche. Dies ist vor allem möglich, da eine Komponente nur Kanäle zu anderen Komponenten ihrer Vaterkomponente aufbauen darf. Die Verantwortung für die Verschaltung ist aus pragmatischen Gründen bei der Gluekomponente angesiedelt und so müssen Subkomponenten entsprechende Wünsche zur Veränderung der Verbindungen in eine Kommunikation mit der jeweiligen Gluekomponente übertragen.

- Die Abbildung von Schnittstellen der Vaterkomponente auf die Schnittstellen der jeweiligen Gluekomponente erfolgt ebenfalls anhand von Schnittstellentypen. Eine weitere besondere Behandlung der Gluekomponente ist hierbei jedoch nicht notwendig.

Das Glassbox-Verhalten besteht damit aus einer Verknüpfung der sinnvollen Abläufe der einzelnen Subkomponenten sowie der Vaterkomponente entsprechend den Vorgaben aus dem Blackbox-Verhalten. Um die Menge der Kanäle zwischen den Komponenten zur Laufzeit zu modellieren, führen wir eine Verknüpfung der jeweiligen Schnittstellentypen ein:

$$Connection \quad =_{def} \quad (CType \times IType) \times (CType \times IType)$$

Die Verwendung von $Connection$ ist nur sinnvoll bei einer Einschränkung von $CType$ auf die Komponententypen der Subkomponenten einer Komponente. So bezeichnet beispielsweise $cn \in Connection \mid_{composedOf(c)}$ einen bestimmtes Verbindungsgeflecht zwischen den einzelnen Schnittstellen der Subkomponenten der Komponente c. Um nun die Veränderungen in der Menge der Kanäle zu erfassen, werden eine Reihe solcher Snapshots in zeitlicher Reihenfolge zusammengestellt und mit $Connection^T$ bezeichnet. Ein Glassbox-Komponententyp kann nun definiert werden als die Zusammenstellung einer Reihe von Blackbox-Komponententypen (inklusive dem der Vaterkomponente) und einer solchen Historie der Verbindungen:

$$compBeh_{gb} \quad : \quad CType \rightarrow \wp((CType \rightarrow \mathbb{N}) \times Connection^T)$$

Diese Definition weist einem Komponententyp eine Menge möglicher Abläufe zu, wobei über die natürlichen Zahlen die Abläufe der jeweiligen Blackbox-Komponententypen referenziert werden. Diese Definition erfordert eine ganze Reihe von Konsistenzkriterien insbesondere im Hinblick zu den anderen bestehenden Definitionen. Auf die formale Festlegung sei an dieser Stelle verzichtet — stattdessen führen wir die Einschränkungen hier nur stichpunktartig auf: die Abbildung $compBeh_{gb}$ muß eingeschränkt sein auf die Komponententypen der Vaterkomponente und der Subkomponenten. Zudem muß der Ablauf der Vaterkomponente konform sein zu den Abläufen der Gluekomponente an übereinstimmenden Schnittstellentypen. Nicht zuletzt verbleibt die Sicherstellung zwischen Typ- und Instanzebene: die Verbindungen zwischen Schnittstellen (entsprechend der Abbildung $connected$) müssen den Vorgaben aus dem Glassbox-Komponententyp entsprechen.

Soll im Modell die Menge der Subkomponenten nicht starr festgelegt sein, sondern gleichfalls über die Laufzeit variabel gestaltet werden, so liesse sich der Glassbox-Komponententyp auf eine zu den flexiblen Verbindungen analoge Weise erweitern.

3.4.4 Unterspezifikation

Insbesondere für die Behandlung von Rollen in späteren Abschnitten ist es wichtig, daß über Komponententypen nur bestimmte Aspekte des Verhaltens einer Komponente spezifiziert werden. Diese Eigenschaft, daß über den Komponententyp keine vollständige Beschreibung des Komponentenverhaltens vorliegt, bezeichnen wir als *Unterspezifikation*. Der einzige Mechanismus an dieser Stelle, um nur Mindestanforderungen an das Verhalten zu stellen, ist die Einführung von Nichtdeterminismus bei der Abbildung der Kommunikationshistorien (siehe dazu auch die Typ-Instanz-Beziehung im vergangenen Abschnitt). Aus diesem Grund müssen beispielsweise alle möglichen Abläufe an einer Schnittstelle angegeben werden, falls die Kommunikationsflüsse an dieser Schnittstelle nicht relevant im Rahmen eines Komponententyps sind.

Allein aus pragmatischen Gründen bietet es sich an, für die Spezifikation von Komponententypen einfachere Hilfsmittel zu verwenden, die sich unmittelbar auf Kommunikationshistorien abbilden lassen. Die Auswahl hierbei ist vielfältig: Formale Beschreibungstechniken wie z.B. Prädikatenlogik bieten sich ebenso an wie graphische Notationen, beispielsweise in Form von *Interaktionsdiagrammen* [BRS97]. Ein Beispiel für die Umwandlung von Diagrammen, die das Kommunikationsverhalten einer Komponente beschreiben, in die formale Repräsentation durch den zugehörigen Komponententyp findet sich in [BRS$^+$00].

3.4.5 Zusammenfassung

In den vergangenen Abschnitten wurde ein einfaches Systemmodell basierend auf interagierenden Komponenten vorgestellt. Komponenten weisen hierbei eine Reihe von Schnittstellen auf, über die sie mit anderen Komponenten verbunden sein und kommunizieren können. Zudem wurde eine hierarchische Komposition von Komponenten eingeführt, die insbesondere zu Restriktionen hinsichtlich der Verbindung zwischen Komponenten führt. Eine Unterscheidung zwischen Instanz- und Typebene wurde eingeführt, wobei Schnittstellentypen syntaktische Aussagen über den Nachrichtenstrom treffen, während Komponententypen das Verhalten einer Komponente spezifizieren und sich dabei auf Schnittstellentypen abstützen. An strukturellen Veränderungen einer Anwendung wurde in der Modellierung lediglich der Auf- oder Abbau von Komponenten berücksichtigt. Weitere Varianten, wie beispielsweise das Hinzufügen oder Entfernen von Subkomponenten, können jedoch verhältnismäßig einfach in das Modell mit aufgenommen werden.

Der Schwerpunkt liegt auf der Spezifikation des Verhaltens einer Komponente — hierfür wurden zwei unterschiedliche Sichtweisen diskutiert. Das von „außen“ beobachtbare Verhalten einer Komponente wird beschrieben durch den Blackbox-Komponententyp und stellt eine Art Vertrag (engl: contract) zwischen dem Hersteller und dem Anwender einer Komponente dar. Der Glassbox-Komponententyp kombiniert das Blackbox-Verhalten einer Komponente mit dem Blackbox-Verhalten ihrer Subkomponenten und legt insbesondere die möglichen strukturellen Veränderungen fest. Im vorliegenden Modell wurde dabei nur die Verschaltung zwischen den einzelnen Subkomponenten berücksichtigt.

Das vorgestellte Systemmodell vereint die gängigsten Konzepte existierender Komponentenplattformen und sichert auf diese Weise eine einfache Übertragbarkeit in eine technische Realisierung (vgl. Kapitel 4). Einzig die hierarchische Komposition wird durch bestehende Technologien wie z.B. das CORBA Komponentenmodell [CCM99] oder EJB [EJB] nicht unterstützt. Letztlich kann diese Strukturierung auch als ein Hilfsmittel der Spezifikation verstanden werden, dessen Auswirkungen auf eine „flache“ Komponentenmenge mit anderen Mechanismen nachvollzogen werden können.

3.5 Komponentenframeworks und Fachkomponenten

In den folgenden Abschnitten wird die mittlere Schicht der formalen Modellierung aus Abbildung 3.2 motiviert und präsentiert. Eine Reihe neuer Konzepte, wie beispielsweise das der Fachkomponenten und der Komponentenframeworks werden eingeführt, um insbesondere eine stringente Abbildung der fachlichen Modellierung (aus Abschnitt 3.3) auf

ein komponentenbasiertes Anwendungssystem (entsprechend der Erkenntnisse aus Abschnitt 3.4) zu erreichen. Dabei wird die Notwendigkeit eines flexiblen Typsystems für Komponenten verdeutlicht und die Komposition von Rollen in Form von Komponententypen im Detail diskutiert. Weiterhin wird die Softwarearchitektur als eine mehrschichtige Strukturierung eines Anwendungssystems verstanden, deren einzelne Bestandteile anhand der beteiligten Komponentenframeworks ausgerichtet sind. Auf diese Weise ergibt sich eine flexible Modularisierung, die die Auswahl einer optimalen Organisation für die jeweils beteiligten Sichten ermöglicht.

3.5.1 Vom Fachmodell zum Systemmodell

Es ist gängige Praxis in der objektorientierten Softwareentwicklung, bei der Konzeption der Implementierung die Vorgaben des Analysemodells nur als Richtlinie zu werten und je nach Bedarf die vorgegebenen Klassen zu übernehmen oder in eine Reihe geeigneterer Klassen zu transformieren (vgl. [BRS97]). Die Ausnutzung dieser Freiheit bei der Realisierung führt zwar zu einer optimierten Implementierung, jedoch kann in einem solchen Fall das Analysemodell nicht mehr für ein Verständnis der Anwendung herangezogen werden. Um einen solchen „Modellbruch“ bei der Umsetzung eines Fachmodells in ein Systemmodell zu vermeiden, müssen die Entitäten des Fachmodells unmittelbar auf Komponenten des Systemmodells abgebildet werden. Obwohl dieses Vorgehen in vielen Situationen eine Einschränkung der üblichen Freiheiten bei der Implementierung darstellt, erkauft sich der Entwickler damit eine Reihe von Vorteilen: Einerseits bleibt das durch das Fachmodell aufgebaute, abstrakte Verständnis der Zusammenhänge erhalten und andererseits ermöglicht diese direkte Abbildung die Integration auf der abstrakten Ebene der Fachmodelle mit direkten Auswirkungen auf die Implementierung. Aus diesem Grund legen wir im folgenden fest, in welcher Form eine Realisierung eines Fachmodells vorgenommen wird.

> **Erläuterung 3.14 (Fachkomponente)** Die Entitäten eines Fachmodells werden im Systemmodell durch Komponenten repräsentiert. Um diese Softwarebausteine von anderen (beispielsweise technischen Komponenten) unterscheiden zu können, bezeichnen wir sie im weiteren als *Fachkomponenten* in Anlehnung an den verbreiteten Begriff der „business component“ [HS00]. Die einer Entität zugeordneten Rollen werden auf Komponententypen abgebildet.

An dieser Stelle werden Fachkomponenten noch als Blackbox betrachtet — die Möglichkeit einer Zergliederung in weitere Komponenten ist für Fachkomponenten erst im Hinblick auf ihre Realisierung interessant (siehe Abschnitt 3.6). Allerdings ist diese Eigenschaft von Komponenten ideal, um das Zusammenwirken mehrerer Fachkomponenten

im Rahmen einer Beziehung des Fachmodells zu erfassen und zu spezifizieren. Aus diesem Grund werden Beziehungen auf eine Glassbox-Sicht einer Komponente abgebildet.

Erläuterung 3.15 (Frameworkkomponente) Jede Beziehung eines Fachmodells wird im Systemmodell durch eine geeignete Komponente repräsentiert. Jede dieser Komponenten heisst *Frameworkkomponente* und ist in der Glassbox-Sicht hierarchisch zerteilt in eine Menge von Subkomponenten, den jeweiligen Fachkomponenten.

Unter einem *Komponentenframework* verstehen wir nun eine Sammlung von Fachkomponenten und Frameworkkomponenten, die entsprechend dem zugeordnetem Fachmodell organisiert und strukturiert sind. Die uniforme Abbildung von Entitäten und ihren Beziehungen auf Komponenten des Systemmodells erscheint pragmatisch und weist eine Reihe von Vorteilen auf:

- Assoziationen des Fachmodells sind nicht nur ein Hilfskonstrukt der Systemspezifikation, sondern als „first-order“-Entität auch im Laufzeitmodell vertreten. Dadurch ist es auf einfache Weise möglich, flexibel zur Laufzeit neue Assoziationen hinzuzufügen bzw. bestehende aufzulösen.
- Frameworkkomponenten bieten unter Umständen selbst eine Schnittstelle an, über die beispielsweise ihr Verhalten beeinflusst oder eine weitergehende Konfiguration vorgenommen werden kann.
- Werden auch Assoziationen Rollen zugewiesen (wie in Abschnitt 3.3.6 diskutiert), so profitiert die Konzeption von der uniformen Behandlung von Entitäten und Beziehungen.

3.5.2 Zusammenspiel

In der objektorientierten Programmierung werden Beziehungen der fachlichen Modellierung üblicherweise in geeignete Interaktionen zwischen den beteiligten Objekten übertragen. Je nach Festlegung kennen sich die Objekte untereinander und können sich gegenseitig Nachrichten schicken. Der Nachteil hierbei liegt darin, daß Objekte sich in der Verantwortung befinden, die Gruppe an benötigten Kommunikationspartnern korrekt zu verwalten. Der Designer ist bei der Umsetzung also meist gezwungen, die Verwaltung der Beziehung einem oder beiden der Kommunikationspartner zu übertragen.

Die Repräsentation eines Komponentenframeworks als eigenständige Komponente in der Realisierung bietet nun die Möglichkeit, kooperierende Fachkomponenten voneinander zu entkoppeln. Die Vorteile sind vielfältig: einerseits agiert die Frameworkkomponente

als „Zwischenhändler“ , um die jeweiligen Partner einer Interaktion voneinander zu trennen — andererseits ist dadurch der Kontrollfluß an zentraler Stelle — die Fachkomponenten reagieren hauptsächlich. Und schließlich können konkrete Interaktionsmuster einfach ausgetauscht werden; in den meisten Fällen sogar ohne Modifikationen an den beteiligten Fachkomponenten. Die Prinzipien eines solchen Zwischenhändlers finden sich im *Mediator-Pattern* aus [GHJV95] wieder und wir bezeichnen die Gluekomponente einer Frameworkkomponente im weiteren als Mediator, da sie zwischen den einzelnen Fachkomponenten vermittelt und zudem entsprechend der Festlegung aus vergangenen Abschnitten für die Verwaltung der Verbindungen zuständig ist.

Prinzipiell unterscheiden wir zwei Varianten, die es einer Fachkomponente ermöglichen, mit ihren Partnern zu interagieren:

Aktiver Verbindungsaufbau: Die Fachkomponente besitzt einen Verweis auf eine andere Fachkomponente und möchte dieser Komponente eine Nachricht schicken. Über einen geeigneten Mechanismus (z.B. eine Kommunikation mit der jeweiligen Gluekomponente) wird ein Verbindungsaufbau zu der anderen Komponente versucht. Da Komponententypen keine Aussagen über solche Verbindungswünsche enthalten, kann erst zur Laufzeit entschieden werden, ob eine solche Verbindung gestattet wird oder nicht. Die dazu benötigten Informationen sind in den beteiligten Komponentenframeworks vermerkt.

Passiver Verbindungsaufbau: Die Fachkomponente kümmert sich nicht um die richtige Verschaltung mit anderen Komponenten, sondern nimmt an, daß eine solche Verbindung bereits hergestellt wurde. Dies kann einerseits bei der Instantiierung einer Komponente geschehen oder auch dynamisch initiiert durch einen Mediator.

Frameworkkomponenten (und ihre Meditatoren) treten in beiden Fällen nicht unmittelbar in Erscheinung, zumindest nicht für die Fachkomponenten, die sie organisieren. Der Kontrollfluß einer Interaktion geht jedoch in den meisten Fällen von den einzelnen Fachkomponenten aus. Die Mediatoren einer Anwendung müssen somit für eine geeignete Verbindungsstruktur sorgen, so daß Interaktionswünsche einer Fachkomponente erkannt und entsprechend weiterverarbeitet werden. Beispielsweise kann sich ein Mediator auch direkt mit einer Fachkomponente verbinden, um ein solches Verhalten zu realisieren.

Frameworkkomponenten sind langlebige Komponenten, die insbesondere den Lebenszyklus der Fachkomponenten umfassen. Die Verknüpfung zwischen beiden wird bei der Instantiierung einer Fachkomponente hergestellt. Hierbei erhalten „betroffene“ Mediatoren die Möglichkeit, Kommunikationsverbindungen zu der Fachkomponente zu etablieren. Der genauere Mechanismus wird in Kapitel 4 diskutiert.

3.5.3 Erweiterung des Systemmodells

In Abschnitt 3.4 wurde eine allgemeine Modellierung für komponentenbasierte Systeme vorgestellt, die weitestgehend modernen Komponententechnologien wie EJB oder dem CORBA Komponentenmodell nachempfunden ist. So zeigt sich das Modell nicht ausreichend flexibel für die Abbildung von Rollen, wie sie durch Fachmodelle vorgegeben sind. In diesem Abschnitt motivieren wir eine Reihe von Ergänzungen, die dieses Manko beheben und Aufschlüsse für die Erweiterung geeigneter Komponententechnologien bieten (vgl. Kapitel 4).

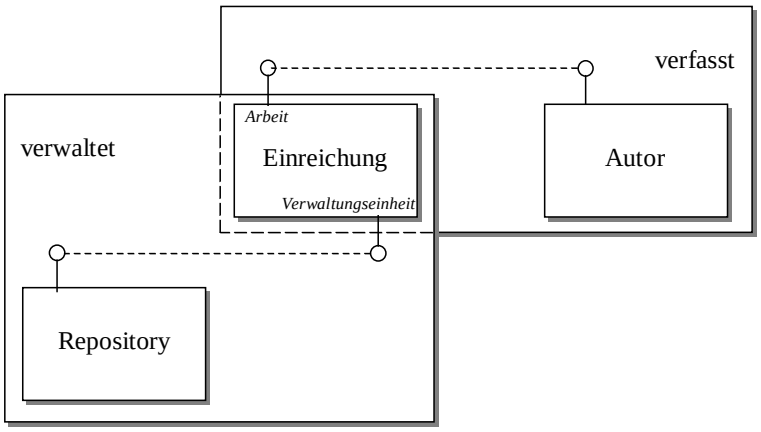


Abbildung 3.20: Mehrfache Inklusion einer Komponente

Abbildung 3.20 zeigt einen Ausschnitt der Umsetzung des Konferenzbeispiels in eine komponentenbasierte Anwendung. Die Assoziationen `verfasst` und `verwaltet` sind auf zwei Frameworks abgebildet. Die Entitäten der Fachmodelle sind auf geeignete Fachkomponenten abgebildet. Eine Sonderrolle übernimmt die Fachkomponente `Einreichung`, die in einem Framework die Rolle `Arbeit` und im anderen die Rolle `Verwaltungseinheit` spielt (zur Verdeutlichung sind die Rollennamen an die jeweiligen Schnittstellen der Komponente geschrieben). Fachkomponenten, die von einem Framework organisiert und koordiniert werden, sind im Modell Subkomponenten der Frameworkkomponente. Diese Konzeption durchbricht die strenge Kapselung von Komponenten, wie sie in Abschnitt 3.4 diskutiert wurde. Die dadurch notwendige Modifikation des Komponentenmodells betrifft einerseits die Instantiierung von Komponentenstrukturen, also die Abbildung $parent$, die nun auch degenerierte Bäume als Strukturierung der Komponenten zulassen muß.

Andererseits führt die Abkehr von der Kapselung auch dazu, daß die strukturellen Vorgaben eines Komponententyps nicht ausreichen, um die korrekte Verschaltung der jeweiligen Subkomponenten zu beurteilen. Stattdessen ist es erforderlich, die Komponententypen sämtlicher Vaterkomponenten zu betrachten, um Aussgen über die Verbindungen treffen zu können. Als einfaches Beispiel seien zwei Fachkomponenten vom Typ C_1 und C_2 gegeben, die beide zwei Fachkomponentenframeworks des Typs FW_1 und FW_2 zugeordnet seien. Hinsichtlich des Aufbaus von Kanälen zwischen C_1 und C_2 treffen beide Frameworks Aussagen, die miteinander kombiniert werden müssen. Gibt der Komponententyp FW_1 demgegenüber noch weitere Verschaltungen zu anderen Komponenten vor, so sind diese aus der Sicht von FW_2 irrelevant.

In der fachlichen Modellierung wurden Rollen lediglich als abstraktes Konstrukt zur Beschreibung bestimmter Charakteristika einer Komponente betrachtet. Nachdem nun Entitäten durch Fachkomponenten repräsentiert werden, sehen wir Rollen als Aspekte des Verhaltens einer Komponente (siehe Abbildung 3.20). Die intuitive Vorstellung besteht darin, daß die Kombination unterschiedlicher Rollen in der Definition eines Komponententyps resultiert. Komponenten von diesem Typ sind in der Lage, alle geforderten Rollen zu übernehmen. Die Definition von Rollen ist eng verwandt mit der von Komponententypen, da über einer Menge von Schnittstellen ein bestimmtes Verhalten beschrieben wird. Das Komponentenmodell wird daher dahingehend ergänzt, daß Rollen als (Blackbox-) Komponententypen spezifiziert und je nach Situation komponiert werden können. Hierbei lassen sich zwei Herangehensweisen unterscheiden:

Erweiterung des Instanzenbegriffs: Die Motivation, eine Komponente als Instanz mehrerer Komponententypen aufzufassen, betont die Unabhängigkeit der einzelnen Typen, die unterschiedlichen Sichtenmodellen entstammen. Zusätzlich erscheint diese Lösung ideal, um mehr Flexibilität zur Laufzeit zu ermöglichen: Einer Komponente könnte es erlaubt sein, ihr Verhalten zur Laufzeit entsprechend der Spezifikation neuer oder anderer Komponententypen anzupassen.

Erweiterung des Typbegriffs: Die erwünschte 1-zu-n Beziehung zwischen Komponenten und ihren Typen kann auch durch das Zusammenfassen aller n Typen zu einem neuen Komponententyp modelliert werden (analog zur Mehrfachvererbung in der Objektorientierung). Diese Variante ist insbesondere interessant, da sie einen Fokus auf die Komposition von Typen legt und entstehende Komponententypen wiederverwendbar macht.

Prinzipiell sind beide Varianten wünschenswert: eine flexible Zuordnung von Komponententypen für eine einfache Sichtenkomposition zur Laufzeit und die Definition neuer Komponententypen, falls die Komposition sich aufwendiger gestaltet und daher öfter wiederverwendet werden soll. Analog zu dem entwickelten Verständnis von Rollen bei der

fachlichen Modellierung (vgl. Erläuterung 3.2) können wir nun festhalten, was eine Rolle im Umfeld komponentenbasierter Anwendungssysteme ausmacht:

Erläuterung 3.16 (Rollen im Systemmodell) Rollen aus der fachlichen Modellierung werden auf Komponententypen des Systemmodells abgebildet. Ein Komponententyp ist in diesem Sinne keine vollständige Spezifikation des Komponentenverhaltens, sondern abstrahiert von diesem im folgenden Sinne: das Verhalten der Komponente wird nur an einer Teilmenge ihrer Schnittstellen beschrieben, wobei von den dort ablaufenden Protokollen nicht abstrahiert wird.

Die gemeinsame Erfüllung mehrerer Rollen durch dieselbe Komponente erfordert Mechanismen, um mehrere Komponententypen zusammenzuführen. Je nachdem, wie die einzelnen Komponententypen geartet sind, erfordert diese Rollenkomposition unter Umständen weitere Entscheidungen des Entwicklers, die zu einem neuen, integrierten Komponententyp führen.

Bei der Komposition von Komponententypen stellt sich insbesondere die Frage, wie das enthaltene Wissen über das Verhalten der Komponente genutzt werden kann, um den neuen Komponententyp zu spezifizieren. Manche Fälle sind sicherlich so geartet, daß die einzelnen Verhaltensannahmen sich gegenseitig nicht beeinflussen — meistens werden wir jedoch von Querbeziehungen ausgehen müssen, die im schlimmsten Fall dazu führen, daß ein Entwickler die bestehenden Komponententypen verwirft und einen neuen Komponententypen spezifizieren muß, der die unterschiedlichen Sichten auf geeignete Weise miteinander kombiniert.

3.5.4 Typkomposition

Die Möglichkeit, Komponententypen zu komponieren ist — wie bereits ausgeführt — eine essentielle Voraussetzung für die Realisierung von Rollen für Komponenten. Die Problematik besteht nun darin, daß unterschiedliche Komponententypen unterschiedliche und möglicherweise in Konflikt stehende Aussagen über Verhalten und Aufbau einer Komponente treffen.

Abbildung 3.21 zeigt schematisch einen Komponententyp, der das Ergebnis einer Komposition zweier weiterer Komponententypen darstellt (die Gluekomponente wurde hier ausgelassen). Die gestrichelten Umgrenzungen zeigen die Bestandteile dieser Typen und die Überschneidungen charakterisieren die Anteile der Komponente, an denen es durch widersprüchliche Spezifikationen zu Konflikten kommen kann. Insgesamt unterscheiden wir die folgenden Konflikte:

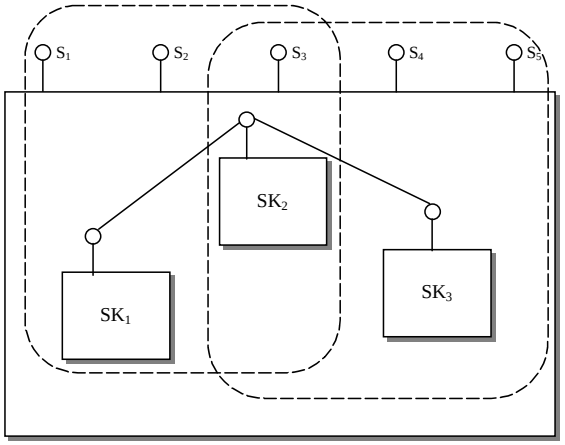


Abbildung 3.21: Komposition von Komponententypen

Blackbox-Konflikt: Die Voraussetzung für einen Blackbox-Konflikt besteht in (mindestens) einem gemeinsamen Schnittstellentyp, der in beiden Komponententypen referenziert wird. Beide Spezifikationen müssen sich auf einen Ablauf an dieser Schnittstelle einigen. Gibt es keinen solchen möglichen Ablauf, so ist eine Komposition der Komponententypen nicht möglich.

Wird beispielsweise die Schnittstelle s_3 verwendet, um eine Änderung des internen Zustands der Komponente zu kommunizieren, so entsteht bei der Zusammenführung der beiden Komponententypen ein Konflikt an dieser Stelle. Da jeder der Komponententypen als vollständige Verhaltensspezifikation einer Komponente ausgelegt ist, können Zustandsänderungen aufgrund anderer Umstände nicht vorhergesehen werden. Dieses Beispiel zeigt jedoch auch, daß ein Konflikt oft einfach zu beheben ist, indem beide Verhaltensaspekte zu einem neuen Komponententyp zusammengeführt werden, der beide Aspekte geeignet kombiniert.

Glassbox-Konflikt: Obwohl das Verständnis über eine Rolle einer Komponente vollständig durch einen geeigneten Blackbox-Komponententyp repräsentiert ist, ist es vorstellbar, auch detailliertere Informationen über die Realisierung einer Rolle abzulegen. Die beiden Glassbox-Komponententypen aus Abbildung 3.21 enthalten beispielsweise eine genaue Vorstellung darüber, wie sich das Verhalten der jeweiligen Rollen auf die einzelnen Subkomponenten auswirkt. Es ist also wünschenswert, bei der Zusammenführung dieser Komponententypen das Detailwissen ebenfalls konsistent zu erhalten.

Die Zusammenführung der Subkomponenten erfordert ein Schema, nach dem festgelegt ist, welche Komponente welchen Komponententypen entspricht. Beispielsweise könnten der Subkomponente SK_2 in Abbildung 3.21 in den zu komponierenden Komponententypen zwei unterschiedliche Typen zugeordnet sein. In diesem Fall würde sich die Typkomposition hinsichtlich der betroffenen Subkomponenten fortsetzen.

Läßt sich eine eindeutige Zuordnung der einzelnen Subkomponententypen finden, so kann ein Glassbox-Konflikt noch durch widersprüchliche Aussagen hinsichtlich der korrekten Verschaltung der Komponenten auftreten.

Logischer Konflikt: Auch wenn sich die Komponententypen (hinsichtlich gemeinsamer Schnittstellen) harmonisch zusammenführen lassen, kann ein Widerspruch in den integrierten Rollen entstehen. Ein einfaches Beispiel findet sich bei zwei Komponententypen t_1 und t_2, die eine DM- und eine Euro-Schnittstelle einer Komponente `Konto` spezifieren. Obwohl es zwei getrennte Schnittstellen sind, ist intuitiv klar, daß beide Rollen Aussagen über einen internen Zustand `Kontostand` treffen. Ein logischer Konflikt bezeichnet einen Widerspruch in diesen impliziten Annahmen. Die Verhaltensspezifikation von t_1 und t_2 ist nun in dieser Form nicht mehr korrekt.

Im Weiteren wollen wir uns nur auf die Komposition von Blackbox-Komponententypen beschränken, da dies dem primären Verständnis von Rollen und ihrer Zusammenführung entspricht. Glassbox-Komponententypen beschreiben in dieser Konzeption Komponentenframeworks und ihre Komposition ist interessant, sobald auch Frameworks Rollen zugewiesen werden. Über diese Variante wird in einem späteren Ausblick diskutiert (vgl. Abschnitt 3.7).

Auf der Ebene von Blackbox-Komponententypen wird ein klares Verständnis hinsichtlich der Komposition von Typen benötigt, um einerseits die Korrektheit der Komposition zu beurteilen und andererseits, um dem Entwickler ein Instrumentarium an die Hand zu geben, mit dessen Hilfe logische Konflikte erkannt und auf geeignete Weise behandelt werden können.

3.5.5 Subtyping

Eine inutitive Vorstellung einer Subtyp-Beziehung ergibt sich aus der Eigenschaft der Ersetzbarkeit: Eine Komponente vom Typ D kann dort eingesetzt werden, wo eine Komponente vom Typ C gefordert ist, falls D ein Subtyp von C ist. Wenn ein System mit der

Komponente vom Typ C korrekt abläuft, so ist die Korrektheit auch durch die Komponente vom Typ D gewährleistet (vgl. [LW94]). Dieser Zusammenhang definiert eine partielle Ordnung zwischen Komponententypen und wird überlicherweise mit $\sqsubseteq$ bezeichnet. Im skizzierten Beispiel gilt $C \sqsubseteq D$.

Die Aussagen dieser Relation können anhand der folgenden drei Ebenen untersucht werden:

Syntax: Ein Komponententyp basiert auf einer Reihe von Schnittstellentypen, die die Syntax der zu versendenden oder empfangenen Nachrichten bestimmen. Die Aussage $C \sqsubseteq D$ bezeichnet hierbei die Eigenschaft, daß der Komponententyp D mindestens auf denselben Schnittstellentypen wie der Typ C aufbaut.

Protokoll: Protokolle gehen einen Schritt über die rein syntaktische Festlegung von Interaktionen hinaus und definieren konkrete Abläufe, beispielsweise anhand von Notationen wie Zustandsautomaten, Temporallogik oder auch dem π-Calculus. Die Ersetzbarkeit zweier Komponententypen kann in diesem Fall anhand der jeweiligen Protokolle bestimmt werden.

Semantik: Auf der semantischen Ebene bezeichnet der Operator $\sqsubseteq$ das sogenannte *behavioral subtyping* [LW94]. Eine Relation der Form $C \sqsubseteq D$ charakterisiert ein Verhalten von D, das den von C vorgegebenen Richtlinien entspricht. Solche Richtlinien werden üblicherweise durch Pre- und Postconditions oder auch durch Invarianten repräsentiert.

Die Auswirkungen der Komposition von Rollen und damit der Zusammenführung von zwei Komponententypen B und C läßt sich nur auf der semantischen Ebene definieren. Logische Konflikte können entstehen, obwohl die beiden Komponententypen auf ganz unterschiedlichen Schnittstellentypen basieren und unterschiedliche Abläufe implizieren. Von einer erfolgreichen Zusammenführung zweier Komponententypen $B, C \in CType$ erwarten wir die folgende Eigenschaft:

$$\exists D \in CType \quad : \quad B \sqsubseteq D \wedge C \sqsubseteq D$$

Logische Konflikte sind das Ergebnis von Seiteneffekten zwischen den Verhaltenssichten der einzelnen Komponententypen. In der Konzeption der vergangenen Abschnitte beschreibt ein Komponententyp das Verhalten entsprechend der zugrundeliegenden Rolle isoliert. Die Zusammenführung zweier Rollen zwingt den Entwickler, das neue Verhalten von Grund auf neu zu bestimmen und in Form eines neuen Komponententyps festzuhalten. Dies ist nicht nur umständlich, sondern auch ineffizient, da keine Wiederverwendung des Wissens bestehender Komponententypen erfolgen kann.

Aus diesem Grund wird an dieser Stelle ein Ansatz verfolgt, der die mittelbare Modellierung eines gemeinsamen Zustands zwischen mehreren Rollen ermöglicht. Dafür wird eine Reihe zusätzlicher Schnittstellen eingeführt, über die eine Rolle diesen Zustand verändern kann und auf diese Weise Aufschlüsse über ihren eigenen Zustand zuläßt. Das folgende Beispiel verdeutlicht diesen Ansatz.

Die Rolle `Verwaltungseinheit` aus dem Beispiel aus Abschnitt 3.3.4 bezeichnet eine Komponente, die bestimmte Daten kapselt und im Rahmen des *remote repository*-Frameworks verwaltet wird. Sie wurde mit der Rolle `Arbeit` aus dem zweiten Framework kombiniert und dieser Verbund repräsentiert eine Einreichung zur OOPSLA. Ein „interner“ Zustand der Rolle `Verwaltungseinheit` ist beispielsweise die Eigenschaft „lesender und schreibender Zugriff“ bzw. „nur lesender Zugriff“. Dieser Zustand ist gleichfalls für die Rolle `Arbeit` ausschlaggebend, da ein einmal eingereichtes Dokument von den Autoren nicht mehr verändert werden darf.

Die Idee besteht nun darin, den gemeinsamen Zustand zu extrahieren und über eigene Schnittstellen wieder zu kapseln. Die den Rollen zugeordneten Komponententypen müssen diese „internen“ Schnittstellen in ihrem Verhalten berücksichtigen und nutzen diesen Mechanismus zur Synchronisation. Abbildung 3.22 zeigt diese gemeinsame Schnittstelle zu einem gemeinsamen Zustand. Das Protokoll an dieser Schnittstelle ist ähnlich zum *Observer-Pattern* aus [GHJV95], um sicherzustellen, daß beide Komponentenspezifikationen von Änderungen der jeweils anderen informiert werden.

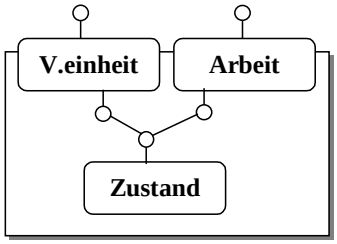


Abbildung 3.22: Kapselung eines gemeinsamen Zustands

Um also die Problematik zu vermeiden, daß Rollen (repräsentiert durch Blackbox-Komponententypen) aufgrund der starken Kapselung des jeweiligen Verhaltens nicht zusammengeführt werden können, werden *Greybox*-Komponententypen eingeführt, die dezidierte Aussagen über interne Zustände der Rolle ermöglichen. Diese Aussagen bieten die Möglichkeit der Abstimmung der einzelnen Rollen untereinander und damit die Voraussetzung für die Zusammenführung der jeweiligen Verhaltensbeschreibungen.

Das Verhalten einer Komponente ergibt sich somit aus dem Zusammenspiel einer Reihe von Greybox-Komponententypen. Ein Greybox-Komponententyp referenziert hierbei andere Typen, so daß eine partielle Ordnung existiert, die die jeweiligen Delegationsketten bestimmt. Die Komposition von Blackbox-Typen kann nun auf eine geeignete Zusammenstellung von Greybox-Komponententypen abgebildet werden.

Obwohl der Schritt von einer reinen Blackbox-Betrachtung hin zu Greybox-Komponententypen bereits als frühe Form der Realisierung aufgefaßt werden kann, ist die vorgeschlagene Zerlegung ein reines Instrument der Spezifikation. Interne Abhängigkeiten zwischen Rollen können über die internen Schnittstellen explizit gemacht werden und dem Entwickler helfen, das Zusammenspiel der Rollen zu beurteilen. Insbesondere wurden bei der fachlichen Modellierung in Abschnitt 3.3 bereits Abhängigkeiten zwischen Rollen definiert, die auf Beziehungen zwischen Greybox-Komponententypen abgebildet werden können.

An dieser Stelle lohnt sich ein Vergleich mit anderen Ansätzen zur Verhaltensspezifikation von Komponenten und Frameworks, von denen bis heute nur wenige existieren [Gra99]. Praktische Anwendung erfährt insbesondere Catalysis, das eine Methodik für die Definition von Komponenten und Frameworks vorsieht und sich hierbei der UML und der *Object Constraint Language* (kurz: OCL) bedient (vgl. [DW98]). In Catalysis beinhaltet die Spezifikation des Verhaltens einer Komponente Auswirkungen auf andere Komponenten und so referenziert jeder Komponententyp eine Reihe weiterer Komponententypen, die in Form eines *Typmodells* zusammengestellt sind. Der gekapselte Zustand einer Komponente wird durch eine Reihe von *Attributen* repräsentiert und das Komponentenverhalten ist zerteilt in unterschiedliche *Operationen*, deren Ausführung zu einer Veränderung der Werte dieser Attribute führen kann. Konkret wird der bekannte Mechanismus der Vor- und Nachbedingung genutzt, um einerseits festzulegen, wann die Ausführung einer Operation erlaubt ist und weiterhin ihre Auswirkungen auf die gegebene Menge von Attributen zu beschreiben.

Im Gegensatz zu dieser Modellierung interner Zustände über Attribute bietet der Ansatz von Greybox-Komponententypen den folgenden Vorteil: bei der Definition des Rollenverhaltens wird nicht nur die Auswirkung auf den internen Zustand beschrieben sondern zusätzlich umgekehrt der Einfluß von Veränderungen des internen Zustands auf das Rollenverhalten. Zudem wird derselbe Mechanismus für die Beschreibung von internen Zustandsänderungen verwendet wie für die Spezifikation der Auswirkungen auf andere Komponenten. Dadurch ergibt sich eine einfache Konzeption, die sehr direkt in eine spätere Implementierung überführt werden kann.

3.5.6 Komposition im Detail

Die Bereitstellung von Greybox-Komponententypen für die einzelnen Rollen eines Fachmodells bietet die Möglichkeit, die Integration auf der Ebene der Rollen ohne detailliertes Wissen über das konkrete Verhalten der jeweiligen Rollen vorzunehmen. Andererseits erfordert es eine gewisse Weitsicht, um bereits frühzeitig vorhersagen zu können, welcher Zustand einer Rolle für andere Rollen interessant sein könnte. Der Erfolg dieses Vorgehens hängt somit von der Erfahrung des jeweiligen Entwicklers eines Komponentenframeworks ab.

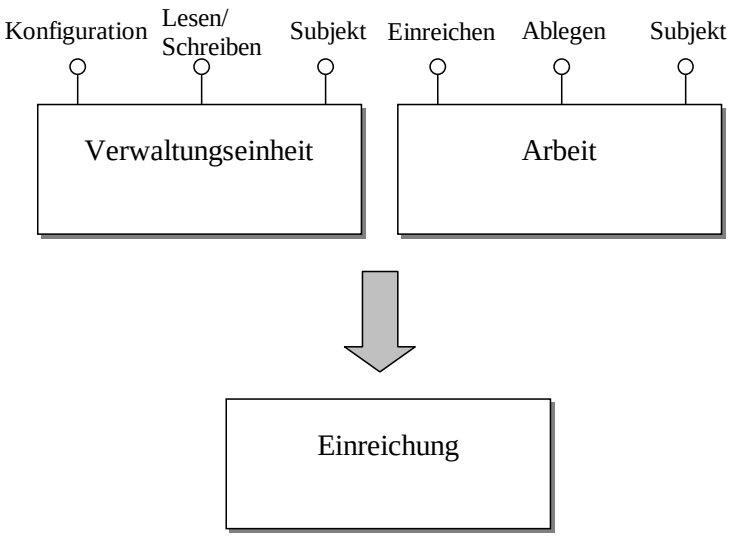


Abbildung 3.23: Komposition zweier Blackbox-Komponententypen

Abbildung 3.23 zeigt zwei Blackbox-Komponententypen, die den Rollen `Verwaltungseinheit` und `Arbeit` zugeordnet sind. Eine Fachkomponente, die die Rolle `Verwaltungseinheit` spielt, kommuniziert demnach über mehrere Schnittstellen mit ihrem Umfeld: Einerseits können die zu verwaltenden Daten über die Schnittstelle `Lesen/Schreiben` in der Komponente abgelegt bzw. aus dieser wieder ausgelesen werden. Andererseits können über die Schnittstelle `Konfiguration` bestimmte Merkmale des Datenpakets bestimmt oder modifiziert werden. Dazu gehören beispielsweise die Lese- und Schreibberechtigung sowie auch das Datum der letzten Veränderung. Schließlich bietet die Fachkomponente eine Schnittstelle `Subjekt` an, die es anderen Komponenten ermöglicht, Veränderungen der verwalteten Daten zu registrieren und geeignet zu reagieren (analog zu dem bekannten *Observer-Pattern* aus [GHJV95]).

Das durch den Komponententypen zur Rolle `Verwaltungseinheit` spezifizierte Verhalten ist nun gegeben durch die jeweiligen Abläufe an den deklarierten Schnittstellen.

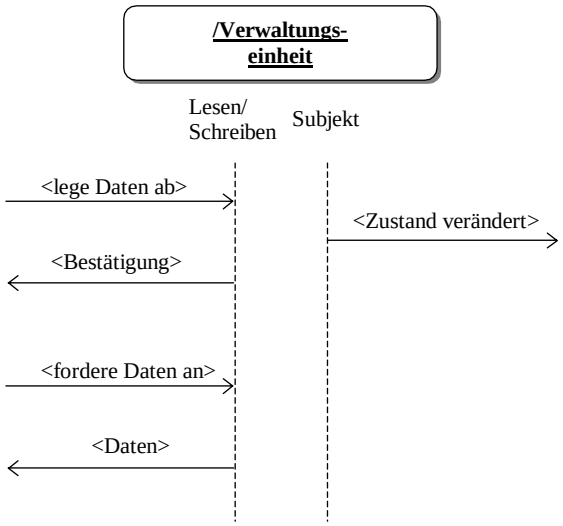


Abbildung 3.24: Blackbox-Verhalten der Rolle `Verwaltungseinheit`

Ein einfaches Verständnis vermittelt das Interaktionsdiagramm aus Abbildung 3.24. Die hier verwendete Notation stellt eine Erweiterung der bekannten Sequenzdiagramme aus der UML dar (vgl. [UML00]). In Kapitel 5 werden diese und ähnliche Beschreibungstechniken vorgestellt, wobei dort in Sequenzdiagrammen für jede einzelne Rolle eine „lifeline“ vorgesehen ist und die einzelnen Schnittstellen über Annotationen der Nachrichten unterschieden werden. In den hier gezeigten Abbildungen wurden dagegen aufgrund der Fülle an Schnittstellen jeder einzelnen Schnittstelle eine lifeline zugeordnet.

Konkret ist in Abbildung 3.24 ein Beispielablauf dargestellt, bei dem der Fachkomponente eine Nachricht geschickt wird, um bestimmte Daten aufzunehmen. Infolge dieser Aktualisierung der verwalteten Daten schickt die Fachkomponente eine Benachrichtigung über die Schnittstelle `Subjekt`, um interessierte Komponenten davon zu unterrichten. Im Anschluß werden die gespeicherten Daten wieder ausgelesen.

Gegeben sei ebenfalls ein Blackbox-Komponententyp für die Spezifikation des Verhaltens der Rolle `Arbeit` aus Abbildung 3.23. Fachkomponenten, die diese Rolle ausüben, kapseln eine wissenschaftliche Arbeit, die für eine Einreichung vorbereitet wird. Dabei können die Autoren die gespeicherte Version ihres Werkes jederzeit über die Schnittstelle `Ablegen` aktualisieren. Sobald die Arbeit fertiggestellt ist, versenden die Autoren über die Schnittstelle `Einreichen` eine Nachricht, die das Werk offiziell für die Konferenz einreicht und damit den Erstellungsprozeß abschließt.

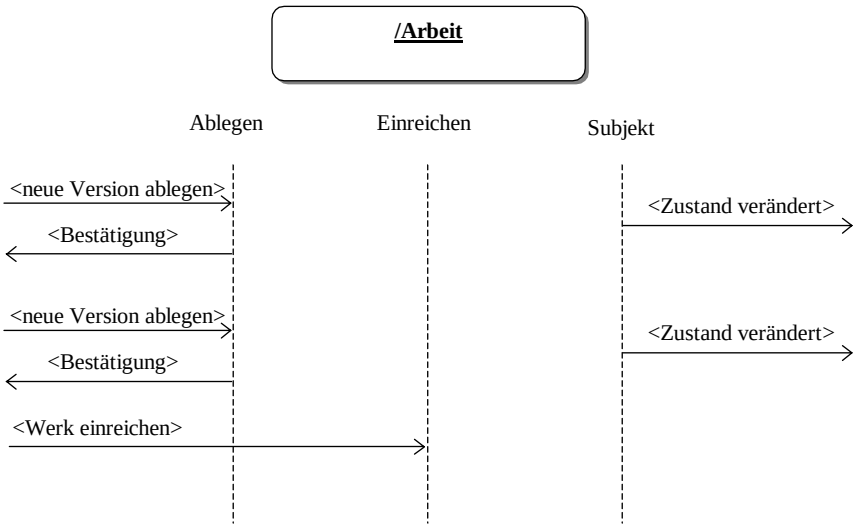


Abbildung 3.25: Blackbox-Verhalten der Rolle `Arbeit`

Das Verhalten der Rolle `Arbeit` ist durch diesen Blackbox-Komponententyp vorgegeben und in Abbildung 3.25 skizziert. Hier ist ein Beispielablauf angegeben, bei dem eine Arbeit zweimal hintereinander aktualisiert wird, bevor sie schließlich fertiggestellt ist. Eine entsprechende Nachricht an die Schnittstelle `Einreichen` vollzieht die Einreichung zur Konferenz. Anschließend ist es den Autoren nicht mehr gestattet, die eingereichte Arbeit mit neueren Versionen zu aktualisieren. Obwohl die angegebenen Interaktionsdiagramme nur ein grobes Verständnis der jeweiligen Blackbox-Komponententypen geben, wird auf eine detailliertere Verhaltensspezifikation verzichtet.

Die Zusammenführung der beiden Komponententypen beginnt mit einer Festlegung der Schnittstellen, die der integrierte Komponententyp aufweist. Die beiden Schnittstellen des Typs `Subjekt` werden im neuen Komponententyp zu einer Schnittstelle dieses Typs zusammengefaßt. Alle weiteren Schnittstellen werden in ihrer bestehenden Form zusammengeführt. Dies führt zu den folgenden Schnittstellentypen: `Konfiguration`, `Lesen/Schreiben`, `Einreichen`, `Ablegen` und `Subjekt`. An letzterer Schnittstelle zeigt sich bereits ein Blackbox-Konflikt bei der Zusammenführung. Die Abläufe an der Schnittstelle könnten nur dann in Übereinstimmung gebracht werden, wenn gleichzeitig eine Zustandsänderung über die Schnittstellen des Typs `Lesen/Schreiben` und `Ablegen` erfolgen würde. Dies entspricht sicherlich nicht dem Verständnis des integrierten Verhaltens.

In einem nächsten Schritt gilt es nun, die relevanten Zustände der beteiligten Rollen zu isolieren und die Verhaltensannahmen so zu verfeinern, daß sie Aussagen über die Veränderung dieser Zustände treffen. Für die Zusammenführung sind die folgenden Zustände interessant:

- Beide Rollen behandeln die Verwaltung eines bestimmten Datenpakets und bieten die Möglichkeit, Veränderungen desselben zu beobachten.
- Die Attributierung der Daten, ob ein lesender und schreibender Zugriff erlaubt ist oder nur ein lesender sind für beide Rollen relevant. Insbesondere muß sichergestellt werden, daß ab dem Zeitpunkt der Einreichung nur noch lesend auf die Arbeit zugegriffen werden darf.

Weitere denkbare interne Zustände sind nur für die jeweilige Rolle interessant, wie beispielsweise bestimmte Informationen der Konfiguration (Rolle `Verwaltungseinheit`) oder Details über die jeweilige Konferenz (Rolle `Arbeit`). Die identifizierten Zustände werden nun über die Einführung zusätzlicher, interner Schnittstellen „externalisiert“. Um den Zustandscharakter noch zu verdeutlichen, nennen wir die beiden Schnittstellen `Daten` für den zu speichernden Datenblock und `Zugriffsattribut` für die Festlegung, ob die Daten modifiziert werden dürfen oder nicht (siehe Abbildung 3.26).

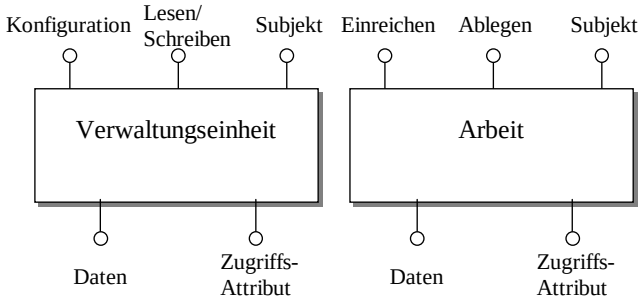


Abbildung 3.26: Erweiterte Komponententypen

Der angestrebte Mechanismus erfordert von den beiden Schnittstellentypen ein Verhalten, das nicht nur Modifikationen des jeweiligen Zustands erlaubt, sondern gleichzeitig solche Veränderungen auch kommuniziert (dies ist möglich, da alle Schnittstellen bidirektional sind). Das Verhalten der beiden Rollen muß nun einerseits so angepaßt werden, daß sich Änderungen am Zustand im Nachrichtenverkehr an den zusätzlichen Schnittstellen auswirken. Andererseits gilt es festzulegen, wie das Rollenverhalten auf Zustandsänderungen reagiert. Die jeweils zugeordneten Greybox-Komponententypen müssen beide Fälle geeignet adressieren.

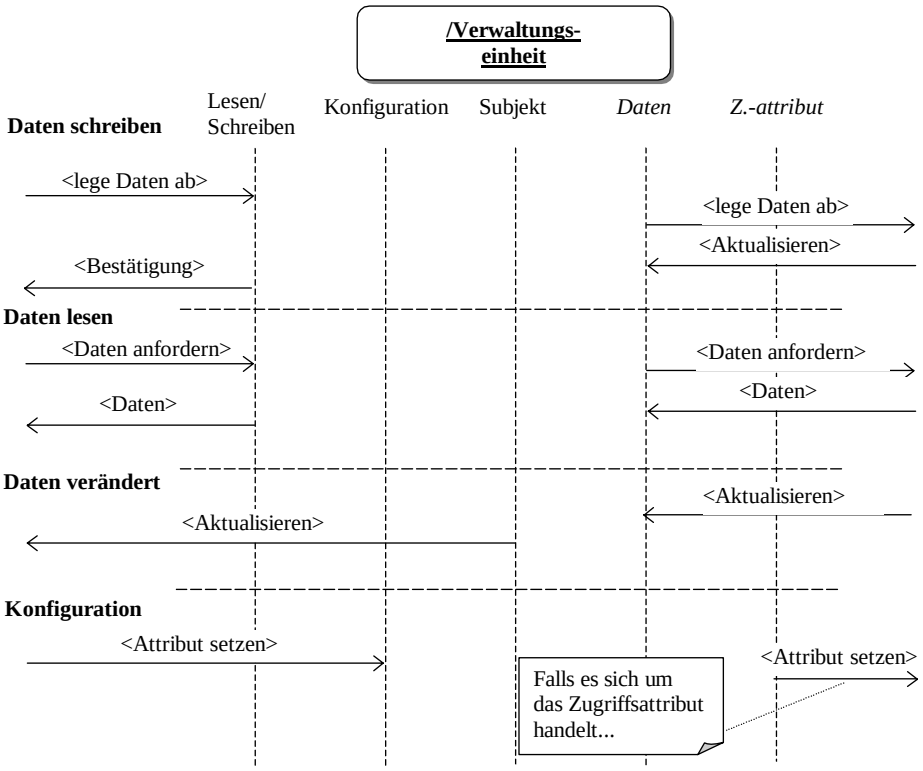


Abbildung 3.27: Glassbox-Komponententyp zur Rolle `Verwaltungseinheit`

Der bereits vorgestellte Blackbox-Komponententyp der Rolle `Verwaltungseinheit` wurde nun so modifiziert, daß das Verhalten in seinen Auswirkungen auf und Beeinflussungen durch die zusätzlichen Schnittstellen festgehalten ist. Abbildung 3.27 zeigt ein entsprechendes Interaktionsdiagramm, in dem beispielhaft typische Abläufe festgehalten sind (zur besseren Übersicht sind die einzelnen Abläufe durch horizontale, gestrichelte Linien getrennt). So wurden beispielsweise die zwei Vorgänge beim Ablegen von Daten, nämlich das eigentliche Speichern und die Notifizierung interessierter Komponenten, über die Schnittstelle `Subjekt` voneinander entkoppelt. Die entsprechende Funktionalität ist nun hinter der Schnittstelle `Daten` verborgen und nicht mehr in der Verantwortung der Rolle. In diesem Beispiel verkümmert das durch den Greybox-Komponententypen definierte Verhalten zu einer Adaption der Kommunikation auf die Schnittstelle des gemeinsamen Zustands. Das verbleibende Verhalten betrifft nur noch die Veränderungen des rollenspezifischen Zustands, beispielsweise die Modifikation von Attributen, die nicht den Schreib-/Lesezugriff betreffen.

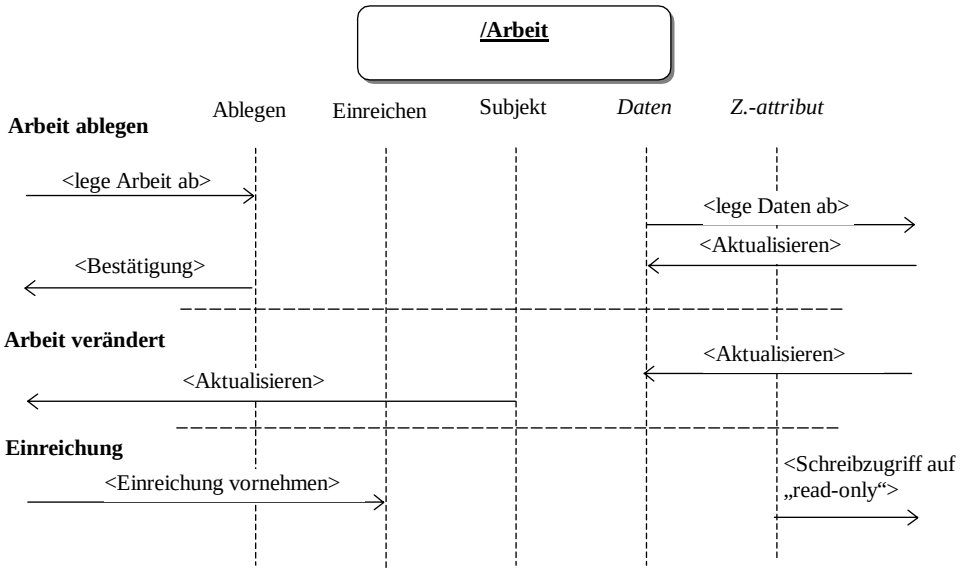


Abbildung 3.28: Glassbox-Komponententyp zur Rolle `Arbeit`

Auf ganz ähnliche Weise wie im Fall des Blackbox-Komponententyps definieren wir nun für die Rolle `Arbeit` einen Greybox-Komponententyp, der das Verhalten der Rolle ebenfalls in Bezug auf die hinzugefügten Schnittstellen charakterisiert. Hier zeigt sich auch die fachliche Logik im Rollenverhalten: Sobald sich die Autoren zu einer Einreichung ihrer Arbeit entschieden haben, wird dafür gesorgt, daß bei dieser Arbeit der Zugriffsmodus auf „read-only“ verändert wird. Die weiteren dargestellten Abläufe sind leicht nachvollziehbar: Beim Ablegen einer Arbeit z.B. wird einfach der gemeinsame Zustand über die Schnittstelle `Daten` aktualisiert.

Im letzten Schritt wurden die Verhaltensspezifikationen der beiden Rollen von je einem Blackbox-Komponententyp auf je einen Greybox-Komponententyp und einen weiteren Blackbox-Komponententyp (mit den Schnittstellen `Daten` und `Zugriffsattribut`) detailliert. Diese Delegation würde im Fall einer *Early Integration* (siehe Abschnitt 3.3.7) einen unnötigen Aufwand darstellen, da der Entwickler viel leichter einen neuen Komponententyp entwirft, der beide Verhaltensannahmen in sich vereint. Bei einer *Late Integration* ist jedoch anzunehmen, daß die Person, die die Aspekte zusammenführt nicht identisch ist mit der Person, die die einzelnen Rollen spezifiziert hat. In dieser Situation sind die Greybox-Komponententypen vorteilhaft, da sie ein klareres Verständnis des Rollenverhaltens übermitteln. Zudem bieten sie über die zusätzlichen Schnittstellen die einfache Möglichkeit zur Integration, auch wenn dies in vielen Fällen nicht so reibungslos abläuft, wie im folgenden gezeigt.

Wir nehmen nun an, zu den Rollen der zu integrierenden Fachmodelle seien die vorgestellten Greybox-Komponententypen gegeben zusammen mit einem Blackbox-Komponententyp (im weiteren als `Zustand` bezeichnet), über den das Kommunikationsverhalten an den Schnittstellen des Typs `Daten` und `Zugriffsattribut` definiert ist. Bei der Zusammenführung der beiden Verhaltensaspekte wird nun versucht, eine Zusammenstellung von Komponenten zu finden, die an den Schnittstellen ein Verhalten aufweist, das beide Verhaltensaspekte korrekt kombiniert. Das Vorgehen ist dabei wie folgt:

1. In einem ersten Schritt werden identische Schnittstellentypen in den beiden Spezifikationen herausgearbeitet und durch eine entsprechende Umbenennung vereinheitlicht.

2. Die Zusammenführung der regulären Schnittstellen erfolgt wie bereits oben beschrieben und führt zu einem neuen Komponententyp, der in Abbildung 3.26 dargestellt ist. Bei der weiteren Betrachtung der Verhaltensweise der Komponente bilden wir ein- und ausgehende Nachrichten auf die entsprechenden Schnittstellen der beiden Greybox-Komponenten ab. An einer gemeinsamen Schnittstelle (in dem Beispiel die Schnittstelle vom Typ `Subjekt`) müssen daher die Abläufe der beiden Spezifikationen identisch sein.

3. Im Gegensatz dazu werden die internen Schnittstellen mit genau einer Komponente verschaltet, die den jeweils referenzierten Komponententypen entspricht (siehe dazu die Skizze in Abbildung 3.22). Diese realisierungsnahe Verhaltensspezifikation ist auf dieser Ebene etwas umständlich, ermöglicht jedoch später eine einfache Übertragung in die Implementierung (siehe nachfolgende Abschnitte). In vorliegendem Fall beruhen beide Komponententypen auf einem Blackbox-Komponententyp `Zustand`, der den gemeinsamen Zustand kapselt. Für die weitere Betrachtung werden nun die internen Schnittstellen mit denen einer Komponente dieses Typs typgerecht verbunden.

   In den meisten Fällen wird sich dieser Schritt der Delegation zu einem gemeinsamen Zustand aufwendiger gestalten, da die referenzierten Komponententypen, die den Zustand kapseln unterschiedlich geartet sind. In diesen Fällen setzt sich die Typkomposition auf dieser unteren Ebene fort. Da auch diese Zusammenführung auf weiteren internen Zustandskomponenten basieren kann, ergibt sich ein rekursiver Kompositionsprozeß.

4. Mit dem letzten Schritt wurde eine gemeinsame Basis geschaffen, über die beide Greybox-Komponententypen gekoppelt sind. Weiterhin können nun Blackbox-Konflikte an der Menge der gemeinsamen regulären Schnittstellen untersucht werden. Dazu werden die möglichen Abläufe an diesen Schnittstellen untersucht: Ist in den Spezifikationen der beiden Greybox-Komponententypen kein gemeinsamer

Ablauf zu finden, so können die beiden Typen nicht zusammengeführt werden. Im diskutierten Fall repräsentieren beide Greybox-Komponententypen dasselbe Verhalten an der (gemeinsamen) Schnittstelle des Typs `Subjekt` in Abhängigkeit der Schnittstelle `Daten`, so daß an dieser Stelle kein Konflikt auftreten kann.

5. Ist ein Blackbox-Konflikt ausgeschlossen, so werden nun die erlaubten Abläufe für den neu entstehenden Komponententyp festgelegt. Prinzipiell können Greybox-Komponententypen wie Blackbox-Komponententypen gehandhabt werden, da sie lediglich eine Reihe zusätzlicher Schnittstellen aufweisen. Wenn nun diese internen Schnittstellen zweier Greybox-Komponententypen A und B mit eindeutigen Namen versehen werden (beispielsweise `Daten_A` und `Daten_B`), so läßt sich das Kreuzprodukt der einzelnen Abläufe zu einem neuen Greybox-Komponententyp C zusammenstellen:

$$\begin{aligned} \forall n_C \in \mathbb{N} \quad . \quad & \exists n_A, n_B \in \mathbb{N} \\ & compBeh_{bb}(C)(n_c) = compBeh_{bb}(A)(n_A) \cup compBeh_{bb}(B)(n_B) \end{aligned}$$

   Die Abläufe lassen sich deshalb auf diese Weise miteinander vereinigen, da wir im letzten Schritt bereits all jene Abläufe aus den Spezifikationen von A und B entfernt haben, die an den gemeinsamen Schnittstellen nicht dieselbe Kommunikationshistorie aufweisen. Weitere mögliche Abläufe an den Schnittstellen des Komponententyps C entfallen aufgrund der zusätzlichen Einschränkung durch die angebundene Komponente, die den gemeinsamen Zustand enthält. Insbesondere durch deren Verhaltensspezifikation ist die Kommunikation an den internen Schnittstellen weitgehend festgelegt. Die verbleibenden Abläufe charakterisieren nun das Verhalten des neuen Komponententyps als Ergebnis der beschriebenen Typkomposition.

6. An verschiedenen Stellen dieses Vorgehens können Konflikte auf unvereinbare Annahmen über das Verhalten der Komponente deuten. Gerade wenn bei der Konzeption der Greybox-Komponententypen die Abwägung zwischen rollenspezifischem und gemeinsamen Zustand nicht zielführend stattgefunden hat, werden Konflikte eine Zusammenführung unmöglich machen. In diesem Fall bleibt einseits die Möglichkeit, einen oder beide Greybox-Komponententypen zu verbessern oder andererseits einen Blackbox-Komponententyp für die neue Komponente von Grund auf zu entwerfen.

So einfach das vorgeführte Beispiel gehalten ist, so deutlich zeigt es die typische Problemstellung bei der Vorbereitung zur Typkomposition: Wie kann das Gesamtverhalten am besten aufgeteilt werden auf die jeweiligen Rollen und den gemeinsamen Zustand? Zudem ist die Vorstellung von einem Zustand hilfreich aber nicht notwendig. Allgemein

ergibt sich ein Ansatz von gemeinsamen Verhaltenselementen, die es den einzelnen Rollen gestatten, sich untereinander auszutauschen und damit zu synchronsisieren. Wie die Typkomposition zudem gezeigt hat, kann diese Zerlegung des Gesamtverhaltens durchaus mehrstufig geartet sein.

Auf dieser konkreten Ebene der Verhaltensspezifikation zeigt sich ebenso wie auf der abstrakten Ebene der fachlichen Modellierung der Weg zur einfachen Integration in der Zerlegung eines Verhaltens bzw. einer Rolle in kleinere Bestandteile, die anschließend zueinander in Beziehung gesetzt werden. Im nächsten Abschnitt werden die abstrakten Beziehungen zwischen den Rollen einer Entität mit dem gerade entwickelten Verständnis von aufeinander aufbauenden Komponententypen in Einklang gebracht.

3.5.7 Abbildung fachlicher Abhängigkeiten

Der Vorteil einer direkten Abbildung von Abhängigkeiten zwischen Rollen der fachlichen Modellierung auf eine Kooperation von Greybox-Komponententypen liegt in der Eigenschaft, daß eine Integration auf der fachlichen Ebene vorhersehbare Auswirkungen auf die Spezifikation der komponentenbasierten Realisierung bietet. Die Ausrichtung der fachlichen Modellierung auf eine spätere Integration schafft ein frühes Verständnis über den internen Aufbau einer Fachkomponente und die 1-zu-1 Abbildung auf Greybox-Komponententypen führt diese Zerlegung weiter.

Als erstes betrachten wir die Generalisierungsbeziehung zwischen zwei Rollen r_1 und r_2 eines Fachmodells. Hierüber sei eine Verfeinerung des Verhaltens der Rolle r_1 durch die Rolle r_2 repräsentiert. In der Umsetzung zu einer Greybox-Spezifikation ist es naheliegend, das Verhalten von r_2 auf dem Verhalten von r_1 basieren zu lassen.

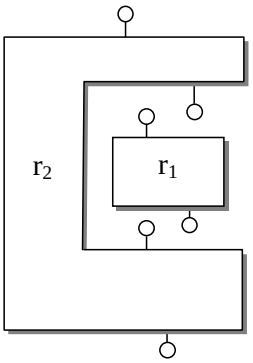


Abbildung 3.29: Erweiterung zwischen Greybox-Komponententypen

Abbildung 3.29 zeigt den allgemeinsten Ansatz, bei dem die Greybox-Spezifikation der Rolle r_1 durch die der Rolle r_2 gekapselt ist. Hierbei kann sowohl die Kommunikation an den internen Schnittstellen (repräsentiert auf der Unterseite der Kästchen) als auch an den anderen Schnittstellen beeinflusst werden. Ein solcher Aufbau scheitert jedoch an der Tatsache, daß einzelne Rollen kaum gekapselt werden können, sofern die Spezifikation offen bleiben soll für die Intergration weiterer Rollen. Aus diesem Grund erlauben wir im weiteren lediglich jene Form der Abbildung, die eine reine Hintereinanderschaltung der Rollen r_1 und r_2 beinhaltet.

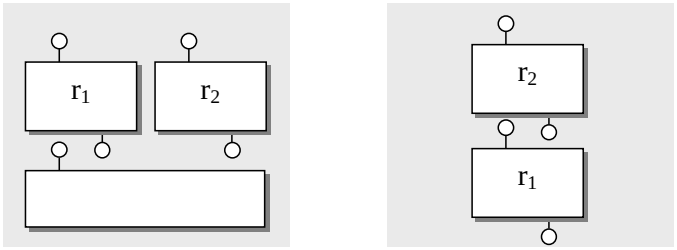


Abbildung 3.30: Abbildung fachlicher Abhängigkeiten

In Abbildung 3.30 ist dargestellt, in welcher Form eine Generalisierungsbeziehung auf die Verschaltung der zugehörigen Greybox-Komponententypen abgebildet wird. Stehen in der fachlichen Modellierung zwei Rollen in einer Generalisierungsbeziehung zueinander, so sind die zugehörigen Greybox-Komponententypen so spezifiziert, daß der Komponententyp zu der Rolle r_1 unmittelbar auf dem Verhalten des Komponententyps zu der Rolle r_2 aufbaut. Die Komponente zu r_2 stellt damit eine interne Komponente für die Spezifikation der Rolle r_1 dar.

Auch Anforderungen zwischen den einzelnen Rollen können direkt auf die Beziehung zwischen den jeweiligen Greybox-Komponententypen abgebildet werden. Eine Anforderung impliziert die Eigenschaft, daß eine Rolle r_1 in gewisser Form von der Funktionalität einer Rolle r_2 abhängig ist, mit dieser also kommunizieren muß (vgl. Abschnitt 3.3.4). Hierbei legen wir fest, daß diese Kommunikation über den Mechanismus der internen Schnittstellen zu erfolgen hat. Bei der Zusammenführung der entsprechenden Greybox-Komponententypen muß daher eine geeignete interne Komponente definiert werden, die — entsprechend den Ergebnissen des vorangegangenen Abschnitts — einen Austausch zwischen den beiden Komponententypen ermöglicht (siehe Abbildung 3.30).

Mit diesen Vorgaben entsteht ein einfaches Schema, das einem Entwickler ein Verständnis darüber vermittelt, wie die Abhängigkeiten zwischen Rollen der fachlichen Modellierung

auf die Komposition der einzelnen Greybox-Komponententypen übertragen wird. Die Beziehung zwischen Rollen unterschiedlicher Abstraktionsniveaus können auf diese Weise noch nicht auf die Ebene der Greybox-Komponententypen abgebildet werden. Zu unterschiedlich sind die Zusammenhänge, in denen die betroffenen Rollen zueinander stehen können. Mit der Einführung von Rollenkomponenten zeigen wir jedoch einen einfachen Ansatz, der diese Problematik adressiert (vgl. Abschnitt 3.6).

Beziehungen zwischen einzelnen Greybox-Komponententypen ergeben sich nicht nur aufgrund fachlicher Abhängigkeiten zwischen den jeweiligen Rollen, die sie repräsentieren. Die Abspaltung eines „gemeinsamen Zustands“ im vergangenen Abschnitt war rein technisch motiviert, um die Zusammenführung unterschiedlicher Verhaltensspezifikationen zu ermöglichen. Dieses Wissen kann jedoch auch auf abstrakter Ebene hilfreich sein, insbesondere hinsichtlich der Integration von Fachmodellen. Im vorliegenden Beispiel würde das bedeuten, daß die Rolle `Zustand` im Fachmodell eingeführt und mit den Rollen `Verwaltungseinheit` und `Arbeit` in Beziehung gesetzt wird. Zu diesem Zweck wurde bereits eine geeignete Abhängigkeit eingeführt, die `Komposition`. Im weiteren verwenden wir nun die Komposition, um bereits im Fachmodell Detailwissen über die Realisierung einer Rolle zu verankern und auf diese Weise das Modell noch weiter auf mögliche Integrationen vorzubereiten.

3.5.8 Beispiel

Die bisher eingeführten Konzepte und Ideen werden nun anhand des Beispiels der OOPSLA-Konferenz angewendet und demonstriert. Grundlage sind die zwei Fachmodelle aus Abschnitt 3.3.10, die ein abstraktes Verständnis je einer der zwei Sichten der Konferenzapplikation repräsentieren. Um die tatsächliche Integration der Sichten möglichst spät im Entwicklungsprozeß vorzunehmen, entwickeln wir in diesem Abschnitt die beiden Fachmodelle unabhängig voneinander weiter zu Komponentenframeworks, die schließlich in einem letzten Schritt zusammengeführt werden.

Der Weg vom Fachmodell zum Fachkomponentenframework bietet dem Entwickler die Möglichkeit, die beteiligten Fachkomponenten im Rahmen einer Softwarearchitektur zu organisieren, die sich optimal für das adressierte Problem eignet. Um diese Eigenschaft unterschiedlicher zugrundeliegender Architekturen deutlich herauszustellen, werden wir in beiden Sichten zwei verschiedene Architekturen einsetzen, die auf eigenen Architekturstilen basieren.

Das Remote-Repository Framework

Das Fachmodell der ersten Sicht (siehe Abbildung 3.16) beschreibt ein allgemeines Modell für die Verwaltung von Informationen (z.B. Dokumente) in einem zentralen Repository. Die analoge Umsetzung dieses Modells in ein generisches Komponentenframework erfordert zuerst die Auseinandersetzung mit einer geeigneten Softwarearchitektur, die den spezifischen Anforderungen des *Remote Repository*-Frameworks entgegenkommt. Viele Informationssysteme basieren heutzutage auf einer sogenannten *Schichtenarchitektur* (vgl. [BMR+96]). Diese strukturiert ein System in eine Reihe aufeinander aufsetzender Ebenen. Die grundlegende Idee findet sich in der Eigenschaft, daß jede Schicht Dienste einer „höherliegenden“ Schicht zur Verfügung stellt und dafür Dienste der „tieferliegenden“ Schicht in Anspruch nehmen kann. Durch die strenge Kapselung der Komponenten innerhalb einer Schicht und die Beschränkung der Abhängigkeiten auf benachbarte Schichten erweisen sich solche Systeme als skalierbar und erweiterbar.

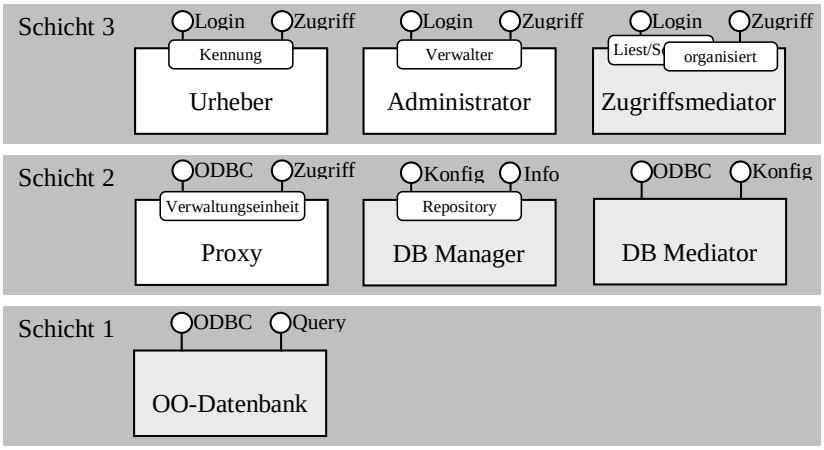


Abbildung 3.31: Schichtenarchitektur des Repository-Frameworks

Abbildung 3.31 zeigt die drei Schichten der Softwarearchitektur des *Remote Repository*-Frameworks. Durch die besondere Eigenschaft von Schichtenarchitekturen ist insbesondere eine Einschränkung in den Möglichkeiten der Verschaltung der Komponenten gegeben. Vor allem dürfen beispielsweise Komponenten der ersten Schicht nicht direkt mit Komponenten der dritten Schicht interagieren. Diese Beschränkung ist ein Teil der Spezifikation des Komponentenframeworks, und ihre Einhaltung wird zur Laufzeit durch die einzelnen Mediatoren sichergestellt.

Die einzelnen Rollen des Fachmodells sind durch geeignete Komponententypen repräsentiert. Der entsprechende Entwurf sieht konkrete Schnittstellen vor und weist den Rollen ein bestimmtes Kommunikationsverhalten zu. Beziehungen der fachlichen Modellierung

zeigen sich in den jeweiligen Mediatoren, beispielsweise dem `Zugriffsmediator` für die `Liest/Schreibt`-Beziehung, die einen Zusammenhang zwischen Verwaltungseinheiten und den jeweiligen Urhebern repräsentiert. Fachkomponenten, die die Rolle `Kennung` spielen, können eine Verbindung direkt zu Fachkomponenten mit der Rolle `Verwaltungseinheit` aufbauen, um die gespeicherten Daten zu lesen oder auch zu modifizieren. Diese Verbindungen werden zwischen Schnittstellen des Typs `Zugriff` etabliert und es ist die Aufgabe des Zugriffsmediators, über die Korrektheit solcher Verbindungen zu urteilen. Die Vorgabe besteht darin, daß der Zugriff auf eigene Dokumente stets möglich sein soll. Dazu führt das *remote-repository*-Framework einen Autorisierungsmechanismus ein, der vorsieht, daß eine Kennung sich für die Dauer einer „Sitzung“ anmeldet und innerhalb dieser Zeitspanne auf den jeweiligen Daten operieren kann. Die Rolle `Kennung` muß daher zusätzlich in der Lage sein, über eine Schnittstelle des Typs `Login` eine Sitzung aufzubauen und auch wieder zu beenden.

Das Beispiel zeigt somit beide Varianten der Kommunikation im Rahmen eines Frameworks (vgl. Abschnitt 3.5.2): Einerseits die Möglichkeit, daß sich die Partner einer Interaktion gegenseitig kennen (z.B. über Referenzen) und die Variante, bei der Kommunikationsverbindungen durch das Framework aufgebaut werden, ohne daß die jeweiligen Komponenten ihre Partner kennen. Für letzteres tritt die dem Framework zugeordnete Komponente aktiv als Zwischenhändler in Aktion. Die jeweiligen Abläufe werden nun detailliert:

- Während der Initialisierung einer neuen Urheber-Komponente werden alle Mediatoren benachrichtigt. Der Zugriffsmediator baut daraufhin eine Verbindung zur `Login`-Schnittstelle der Fachkomponente auf und fordert diese zur Autorisierung auf.
- Um den Autorisierungsmechanismus flexibel zu gestalten, kommuniziert der Zugriffsmediator mit einer weiteren Komponente, die beurteilt, ob eine Anmeldung erfolgreich war, oder nicht. In diesem Beispiel sei dies eine zusätzliche Aufgabe des `Verwalters`.
- Eine Vereinfachung des Modells besteht darin, daß jede Komponente nur maximal eine Schnittstelle eines Typs aufweisen darf. Der Zugriffsmediator kann zwar mehrere Verbindungen zu den `Login`-Schnittstellen der einzelnen `Urheber` halten — eine Unterscheidung der individuellen Kommunikationspartner ist in diesem Fall allerdings nicht mehr möglich. Daher wird die Verbindung nach abgeschlossener Autorisierung wieder abgebaut.
- Über ihre Rolle `Kennung` kann eine Fachkomponente neue Verwaltungseinheiten anlegen, mit diesen kommunizieren und sie bei Bedarf wieder entfernen.

- Voraussetzung für die Kommunikation zwischen `Kennung` und `Verwaltungseinheit` ist eine gültige Sitzung.

Das Beispiel demonstriert das Zusammenwirken zwischen den Fachkomponenten und den Mediatoren des Frameworks. Der Zugriffsmediator stellt sicher, daß die Kommunikation zwischen `Kennung` und `Verwaltungseinheit` nur dann stattfindet, wenn eine gültige Sitzung eingerichtet wurde. Hierfür bleibt dem Mediator nur das Instrument der Verschaltung. Sobald eine Fachkomponente über ihre Schnittstelle vom Typ `Zugriff` eine Verbindung aufbaut zu einer Fachkomponente mit einer Rolle `Verwaltungseinheit` wird der Zugriffsmediator informiert und überprüft den Zustand der aktuellen Sitzung. Wurde noch keine Sitzung initiiert, kommuniziert der Mediator über die `Login`-Schnittstelle mit der Fachkomponente und versucht, eine neue Sitzung zu initiieren. Bleibt dies erfolglos, so kann keine Verbindung zwischen den Fachkomponenten aufgebaut werden — eine weitere Kommunikation ist unterbunden. Eine Kombination aus aktivem und passivem Verbindungsaufbau ermöglicht somit die Entkopplung der einzelnen Fachkomponenten, ohne daß Mediatoren für die Fachkomponenten sichtbar sein müssen.

Die erste Schicht der in Abbildung 3.31 dargestellten Architektur organisiert den Zugriff auf die Verwaltungseinheiten der Anwendung, die letztlich in einer Datenbank gespeichert sind. Fachkomponenten, die eine Verwaltungseinheit repräsentieren, stellen somit nur Kapseln für eine entsprechende Entität der Datenbank dar. So bietet die zweite Schicht der Architektur die Funktionalität zur Abbildung auf die Datenbank, die in der dritten Schicht angesiedelt ist. Gerade für die Abschottung der eingesetzten Persistenzlösung haben sich Schichtenarchitekturen bewährt. Auf eine Beschreibung des konkreten Zusammenspiels zwischen Datenbank, dem Datenbankmanager und den einzelnen Proxies sei an dieser Stelle verzichtet.

Das Zusammenwirken von einzelnen Proxy-Komponenten und der Datenbank ist nicht fachlich motiviert und damit für den Anwender des Komponentenframeworks irrelevant. Insbesondere sind die Komponenten der unteren Schichten nicht mit Rollen des Fachmodells assoziiert. Dem Beispiel wurde lediglich eine Rolle `Repository` hinzugefügt, die es Komponenten der oberen Schichten ermöglicht, auf Details des Repositories zuzugreifen, wie zum Beispiel Name und Ort. Diese und weitere Fachkomponenten sind — ebenso wie beispielsweise die technische Komponente `OO-Datenbank` — integraler Bestandteil des Komponentenframeworks und ihre Repräsentation in Abbildung 3.31 wurde zur Verdeutlichung grau hinterlegt. Demgegenüber markieren weiße Kästchen die Fachkomponenten, die vom Anwender des Frameworks realisiert werden müssen.

Das Beispiel zeigt insbesondere einen Fall, in dem es sinnvoll wäre, den Persistenzmechanismus als eigenständiges Komponentenframework zu isolieren und das *remote repository*-Framework auf diesem aufzubauen. Diese Variante ist insbesondere hilfreich, wenn noch weitere Komponentenframworks einer Anwendung eine Persitenzlösung benötigen

und die Integration anhand dieses gemeinsamen Komponentenframeworks vorgenommen werden kann.

Das Review-Framework

Das zweite Fachmodell aus Abschnitt 3.3.10 beschreibt einen Review-Prozeß, der in einer Liste der für die Konferenz akzeptierten Arbeiten mündet. Grundlage ist ein generisches *Review*-Framework, das jedoch von einem sequentiellen Prozeß ausgeht, also von einem Weiterreichen der Einreichung von einem Reviewer zum nächsten. Dabei wird das Dokument stets um die Kommentare der Reviewer ergänzt und zudem soll es möglich sein, daß Reviewer bereits bestehende Kommentare ergänzen. Das entsprechende Fachmodell beschreibt die beteiligten Gutachter, die zu bewertenden Dokumente mit den jeweiligen Kommentaren und schließlich die Sortierung der Dokumente entsprechend aufgestellter Qualitätskriterien.

Aufgrund des sequentiellen Abarbeitens und Überarbeitens eines Dokuments bietet sich für diese Sicht eine *Pipes and Filters*-Architektur an (vgl. [BMR+96]). Ein solches datenflußzentriertes Design ist im Allgemeinen aus den folgenden, kooperierenden Komponenten aufgebaut: *Filter* werden mit Daten versorgt, die sie in geeigneter Weise bearbeiten und schließlich wieder ausgeben. Die Verknüpfung mehrerer Filter erfolgt mit Hilfe von *Pipes*, die es erlauben, auch mehrere Ein- oder Ausgänge von Filtern miteinander zu verknüpfen (ähnlich einem Multiplexer). Vor allem bieten Pipes die Möglichkeit, einzelne Filter voneinander zu entkoppeln, so daß ein Filter nicht zwangsläufig wissen muß, an wen bearbeitete Daten weiterzureichen sind.

Eine einfache Pipes and Filter-Architektur für das *Review*-Framework ist in Abbildung 3.32 dargestellt (der Datenfluß ist zusätzlich mit Pfeilen verdeutlicht). Hierbei „durchwandern“ Dokumente ausgehend von einer Dokumentenquelle eine Reihe von Begutachtern (die Filter), um schließlich in einer Dokumentensenke aus dem Prozeß entfernt zu werden (in der Abbildung sind die „Daten“ in einem eigenen Kästchen dargestellt). Die Gutachter schreiben Kommentare zu einem Dokument, die zudem mit einer auswertbaren Bewertung versehen werden. Diese Kommentare werden weitergereicht an den Auswertungsprozeß, der jedes Dokument mit seiner Nummer innerhalb der aktuellen Rangfolge versieht.

An dieser Stelle wird der typische Kontrollfluß vom Komponentenframework zu den einzelnen Fachkomponenten besonders deutlich: Der `Beurteilungsmediator` kontrolliert vollständig die Zuordnung von Dokumenten zu den einzelnen Reviewern, die über den passiven Verbindungsaufbau miteinander kommunizieren. Konkret gestaltet sich der Ablauf wie im folgenden skizziert:

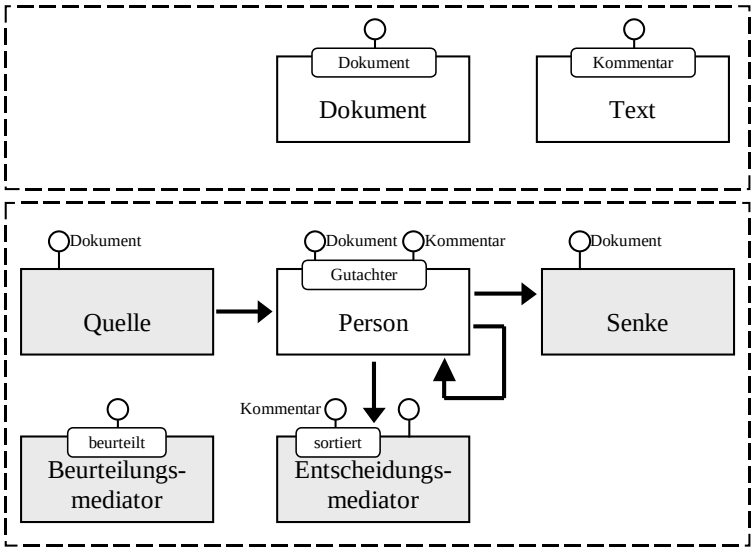


Abbildung 3.32: Pipes and Filter-Architektur des Review-Frameworks

- Bei der Initialisierung eines `Dokuments` baut der `Beurteilungsmediator` eine Verbindung zu dieser Fachkomponente auf.
- Sobald die Einreichung abgeschlossen wurde, verschickt die Fachkomponente über ihre Rolle `Dokument` eine entsprechende Nachricht, die den `Beurteilungsmediator` veranlaßt, den Reviewprozeß zu initiieren.
- Dazu bestimmt der Mediator den jeweils nächsten Reviewer und schickt diesem über dessen Schnittstelle `Dokument` einen Verweis auf das Dokument.
- Ist ein Review abgeschlossen, so kommuniziert das Dokument erneut mit dem `Beurteilungsmediator`.
- Dieser weist das Dokument nun einem weiteren Reviewer zu, falls es noch nicht von drei Reviewern beurteilt wurde.
- Steht die Beurteilung über ein Dokument fest, so kommuniziert der `Beurteilungsmediator` mit dem `Entscheidungsmediator`, der die Rangfolge der gereviewten Dokumente aktualisiert. Diese Sortierung der Dokumente ist somit zu jedem Zeitpunkt auslesbar (auch wenn noch nicht alle Dokumente einem Review unterzogen wurden).

Aufgrund des passiven Verbindungsaufbaus ist es auch in diesem Design für die Komponenten nicht notwendig, den jeweiligen Mediator zu kennen, der den Reviewablauf organisiert. Trotzdem findet sich der „Zustand“ des Ablaufs in den einzelnen Dokumenten

und diese steuern jeden weiteren Durchlauf. Weiterhin kontrolliert der `Beurteilungs-mediator` die Verbindungen zwischen den einzelnen Fachkomponenten und damit insbesondere Kanäle zwischen den Rollen `Dokument` und `Gutachter`.

Integration und Konfiguration

Komponentenframeworks sind halbfertige Anwendungen, die durch die Bereitstellung geeigneter Fachkomponenten von einem Entwickler vervollständigt und instantiiert werden können. Die Integration der beiden einzelnen Komponentenframeworks führt bereits auf der Ebene der fachlichen Modellierung zu einem Verständnis der Anforderungen an die einzelnen Fachkomponenten der Anwendung. Die Zusammenführung der Fachmodelle in Abschnitt 3.3.10 zeigt, welche Rollen der einzelnen Komponentenframeworks in den einzelnen Fachkomponenten zusammengeführt werden. Dieses Zusammenspiel soll anhand des vorgestellten Beispiels nun detailliert werden.

Die Zusammenführung der Rollen aus den beiden Fachmodellen der vorgestellten Komponentenframeworks führt zu der Auseinandersetzung mit zusätzlichen Konsistenzkriterien, die in der isolierten Betrachtung irrelevant sind. Die Behandlung von logischen Konflikten bei der Verhaltenskomposition erfolgt auf Rollenebene und betrifft in diesem Beispiel die folgenden Fachkomponenten: Autoren treten als Gutachter auf, wodurch eine entsprechende Fachkomponente implementiert werden muß, die beide Rollen realisiert. Allerdings stehen die Rollen `Gutachter` und `Kennung` in einer Generalisierungsbeziehung zueinander (siehe Abschnitt 3.3.10, wodurch sich der Aufwand auf die Implementierung einer Fachkomponente mit der Rolle `Gutachter` reduziert. Weiterhin wurden die Rollen `Dokument` und `Verwaltungseinheit` integriert, so daß sich hier die zweite Möglichkeit zur Auflösung von Querbeziehungen zwischen den Komponentenframeworks ergibt. Diese können wie folgt aufgestellt werden:

- Autoren dürfen nicht ihre eigenen Arbeiten begutachten.
- Solange ein Autor seine Arbeit noch nicht eingereicht hat, darf er oder sie nicht als Gutachter für andere Arbeiten herangezogen werden.

Die Einhaltung dieser Konsistenzbedingungen kann nur an den Integrationspunkten der einzelnen Komponentenframeworks stattfinden. Durch die Bereitstellung geeigneter Komponententypen paßt der Entwickler somit jedes Framework an die Gegebenheiten der aktuellen Anwendung an.

Die Verbindungen zwischen `Kennung` und `Verwaltungseinheit` sowie zwischen `Gutachter` und `Dokument` basieren auf denselben Schnittstellentypen und stellen daher

einen „gemeinsamen Zustand“ der beiden Komponentenframeworks dar. Dieser Zustand repräsentiert die Zusammenführung des Wissens, das die beiden Komponentenframeworks über die Fachkomponenten besitzen: Das *Remote Repository*-Framework stellt die Verknüpfung von Autoren zu ihren Werken her während das *Review*-Framework Gutachter mit den Einreichungen verbindet, die sie zu beurteilen haben. Bei der Zusammenführung der beiden Aspekte muß nun sichergestellt werden, daß Autoren nicht ihre eigenen Arbeiten zur Begutachtung erhalten. Das könnte beispielsweise dadurch sichergestellt werden, daß Verbindungen zwischen Autoren und ihrer Arbeit vom *Review*-Framework unterbunden werden, sobald die Arbeit eingereicht wurde. Da dies jedoch auch einen nur lesenden Zugriff unterbinden würde, erscheint die Lösung über das Verbindungsmanagement unzureichend.

Stattdessen können die erwähnten Konsistenzbedingungen innerhalb der Fachkomponente behandelt werden, die die Rollen `Gutachter` und `Kennung` ausübt. Die beiden Rollen sind so zu integrieren, daß jeder Autor eine im Rahmen des Reviews zugewiesene Arbeit ablehnt, sofern einseits die eigentliche Einreichung noch nicht vorgenommen wurde und andererseits es sich um eine Arbeit handelt, die vom Autor verfaßt worden ist. Dieses Beispiel zeigt deutlich den Nachteil einer rollenbasierten Integration: da Komponentenframeworks vom Anwender nicht angepaßt werden können ist es erforderlich, einen bedeutenden Teil der fachlichen Logik den Rollen zugänglich zu machen. Eine mögliche Alternative ist durch die Konfigurationsschnittstellen eines Komponentenframeworks gegeben, die während der Instantiierung eine Beeinflussung des Verhaltens ermöglichen können.

In den vorgestellten Komponentenframeworks sind bereits bestimmte Fachkomponenten realisiert, während andere vom Anwender zur Verfügung gestellt werden müssen. Das integrierte Fachmodell vermittelt dabei ein Verständnis der Zusammenführung von Rollen. Dabei kann es vorkommen, daß eine Fachkomponente (bzw. ihre spezielle Rolle) in einem Framework bereits realisiert wurde, während im anderen Framework noch keine Implementierung verfügbar ist. Für diesen Fall ist eine Zusammenführung von Implementierungsfragmenten notwendig, wie sie in Abschnitt 3.6 diskutiert wird.

3.5.9 Zusammenfassung

In den vergangenen Abschnitten wurde gezeigt, wie sich die Integration von Komponentenframeworks auf die Komposition von Rollen zurückführen läßt. In der formalen Modellierung wurde die Analogie zwischen Entitäten des Fachmodells und Komponenten des Systemmodells gezeigt, woraus sich eine direkte Abbildung von Rollen auf Komponententypen ergab. Weiterhin wurden auch Beziehungen zwischen Entitäten

auf Komponenten des Systemmodells abgebildet, wobei die in Abschnitt 3.4 eingeführte Komposition von Komponenten modifiziert wurde, um es einer Komponente (in diesem Fall einer Fachkomponente) zu erlauben, mehreren Vaterkomponenten zugeordnet zu sein. Dieser Bruch mit der traditionellen hierarchischen Verfeinerung von Komponenten (vgl. [BDD+92]) zeigt Komponenten als Integrationspunkte unterschiedlicher Sichten.

Die Komposition von Rollen in Form von Blackbox-Komponententypen kann zu unterschiedlichen Konflikten in den Aussagen über das Verhalten der Komponente führen, die teilweise auf funktionalen Abhängigkeiten zwischen den Rollen beruhen. Die Komposition von Komponententypen resuliert im günstigen Fall in einem neuen Komponententyp, der die unterschiedlichen Aspekte konsitent zusammenführt. Dies ist insofern unbefriedigend, da dieser Vorgang eine genaue Auseinandersetzung mit den einzelnen Rollen erforderlich macht und eine Wiederverwendung der existierenden Verhaltensaussagen unterbindet. In diesem Zusammenhang wurde über Greybox-Komponententypen ein Mechanismus eingeführt, der es erlaubt, über interne Schnittstellen explizit Informationen aus dem internen Rollenverhalten zu kommunizieren. Der Vorteil liegt darin, daß einerseits Rollenverhalten gekapselt und damit vor dem Anwender verborgen werden kann und andererseits die Zusammenführung von Rollen anhand dieser Integrationsschnittstellen ohne detaillierteres Wissen über das Rollenverhalten möglich wird. Der Nachteil findet sich in der Notwendigkeit, bei der Spezifikation einer Rolle sinnvolle Kombinationen mit anderen Rollen zu antizipieren und die internen Schnittstellen entsprechend zu gestalten. Geben diese Schnittstellen zuviel interne Details des Rollenverhaltens preis, so ist der Vorteil der Kapselung nicht mehr gegeben. Umgekehrt verhindert eine zu „schmale“ Schnittstelle möglicherweise zu viele sinnvolle Kombinationen mit anderen Rollen.

Die Zusammenführung von Greybox-Komponententypen wurde detailliert diskutiert und anhand eines Beispiels demonstriert. Dabei wurde insbesondere der unmittelbare Zusammenhang mit der fachlichen Modellierung herausgestellt: die Zusammenstellung von Rollen kann bereits im Fachmodell genutzt werden, um einen feingranularen Rollenaufbau zu konzipieren. Dies führt nicht nur zu einem besseren Verständnis des implizierten Verhaltens der Komponente sondern zudem zu einer vereinfachten Integration, falls sich gemeinsame Basisrollen identifizieren lassen.

Schließlich wurde anhand des OOPSLA-Beispiels demonstriert, daß sich Komponentenframeworks als Module einer mehrschichtigen Softwarearchitektur eigenen, die eine feinere Abstimmung und Optimierung der Strukturierung auf die jeweilige Sicht erlauben. In diesem Beispiel wurden ebenfalls gezeigt, daß die Anreicherung der Komponentenframeworks (d.h. der Mediatoren in der Realisierung) mit fachlicher Funktionalität die beteiligten Fachkomponenten „entlastet“ und damit vereinfacht. Damit verbunden ist jedoch der Nachteil, daß die in den Frameworks enthaltene Funktionalität vom Anwender nicht beeinflußt und vor allem nicht erweitert werden kann.

Während mit diesen Ergebnissen eine feingranulare Wiederverwendung von Designwissen möglich wird, ist die Wiederverwendung von Code nur in der Form von Komponenten oder Komponentenframeworks möglich. Gerade bei der Komposition mehrerer Komponentenframeworks erscheint dies jedoch als unpraktikabel, da mitgelieferte Komponenten eines Frameworks kaum die Anforderungen anderer Frameworks antizipieren können. Es ist somit ebenfalls eine feinere Aufteilung der Realisierung einer Komponente nötig. Das Konzept der *Rollenkomponenten* adressiert diesen Umstand und wird im nachfolgenden Abschnitt erläutert.

3.6 Erweiterung: Rollenkomponenten

Eine wichtige Motivation bei der Verwendung von Frameworks liegt in der Wiederverwendung bestehender Spezifikationen und Implementierungen. In den vergangenen Abschnitten haben wir gezeigt, wie durch den Einsatz von Greybox-Komponententypen die zu einer Rolle gehörige Verhaltensspezifikation auf die Integration mit anderen Rollen vorbereitet werden kann. Dadurch ist es möglich, die vorhandenen Spezifikationen auch weiterhin zu nutzen ohne sie (wie im Fall von Blackbox-Komponententypen) von Grund auf neu zu definieren. Bei der vorgeführten Zerlegung von Komponententypen wurde das Verhalten bewußt in einer sehr implementierungsnahen Form (über die Verschaltung von Komponenten) spezifiziert. Dies erlaubt nun im Rahmen eines Komponentenframeworks nicht nur bestehende Komponententypen unverändert einzusetzen, sondern zudem die Wiederverwendung konkreter Implementierungen dieser Komponententypen. Die Integration von Komponentenframeworks erfolgt auf der Ebene des Entwurfs durch die Abstimmung der internen Schnittstellen auf eine gemeinsame interne Komponente und setzt sich bei der Realisierung in einer entsprechenden Zusammenstellung der zugehörigen Komponenten fort. Diese Komponenten, die eine Implementierung eines Greybox-Komponententyps darstellen, bezeichnen wir im weiteren als *Rollenkomponenten* (vgl. [VN96]).

Der Vorteil dieser unmittelbaren Nähe zwischen Entwurf und Implementierung besteht darin, daß der Vorgang der Integration, ausgehend von einem abstrakten Verständnis auf der Ebene der fachlichen Modellierung (Abschnitt 3.3), mit der Zusammenführung auf der Spezifikationsebene (Abschnitt 3.5) abgeschlossen ist. Die internen Komponenten (für die Kapselung eines gemeinsamen Zustands zwischen mehreren Rollen) sind damit nicht nur ein rein konzeptuelles Hilfsmittel der Verhaltensspezifikation, sondern finden sich in dieser Form auch bei der Realisierung eines Komponentenframeworks wieder. Damit erreichen wir zudem eine feingranulare Unterteilung der Implementierungsartefakte mit der Möglichkeit, die Wiederverwendung derselben zu maximieren.

Die Integration unterschiedlicher Verhaltensaspekte über geeignet definierte Greybox-Komponententypen funktioniert nur bei Rollen, deren Entitäten auf demselben Abstraktionsniveau angesiedelt sind (siehe Abschnitt 3.3.9). Bei der Zusammenführung von Fachmodellen wurden Rollen auf unterschiedlichen Abstraktionsstufen über einer Abstraktionsbeziehung in Relation gesetzt. Diese Querbeziehungen reduzieren — wie bereits ausgeführt — deutlich die Verständlichkeit der Modelle und ihrer Abhängigkeiten untereinander. Im folgenden wird daher ein einfacher Ansatz vorgestellt, der die Problematik unterschiedlicher Abstraktionsstufen auf einfache, nachvollziehbare Weise adressiert.

3.6.1 Das „Zwiebelschalenmodell“

In Abschnitt 3.5.4 wurde gezeigt, wie ein Blackbox-Komponententyp zerlegt werden kann in einen Greybox-Komponententyp und einen weiteren Blackbox-Komponententyp. Auf diesen kann bei Bedarf eine ähnliche Zerlegung angewendet werden, so daß der ursprüngliche Blackbox-Komponententyp ersetzt wird durch eine Reihe aufeinander aufbauender Greybox-Komponententypen und einen abschließenden Blackbox-Komponententyp. Neben dieser technischen Aufteilung führt auch die Abbildung fachlicher Abhängigkeiten zwischen den Rollen einer Entität eines Fachmodells zu Greybox-Komponententypen, die zusammenwirken, um ein bestimmtes Rollenverhalten zu realisieren. In der Implementierung führt dies zu einer Gruppe von Rollenkomponenten, die in einer „Delegationskette“ angeordnet sind. Die Integration von Komponentenframeworks erfordert die Zusammenführung der jeweiligen Delegationsketten. Während bei der Komposition von Komponententypen nur jeweils eine Delegation berücksichtigt wurde, stellen wir nun ein einfaches Schema vor, daß diese Zusammenführung verständlicher gestaltet. Dazu werden die einzelnen Rollenkomponenten in verschiedene Schichten aufgeteilt, die auf einer gemeinsamen zentralen Komponente basieren. Da Rollenkomponenten in jeder dieser Schichten entweder den Kern oder eine weitere Schicht verbergen, trägt dieses Schema den Namen *Zwiebelschalenmodell*.

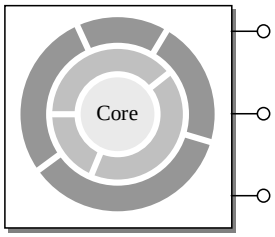


Abbildung 3.33: Schematische Darstellung: Zwiebelschalenmodell

In Abbildung 3.33 ist das Zwiebelschalenmodell schematisch dargestellt. Wie in Schichtenmodellen üblich, wird auch hier gefordert, daß jede Schicht nur mit den unmittelbar benachbarten Schichten kommunizieren darf. Die Festlegung eines Kerns ist hierbei als eher willkürlich anzusehen — je nach Bedarf können weitere Schichten vom Kern abgespalten werden. Es gilt jedoch die folgende Unterscheidung:

Kernkomponenten enthalten das essentielle Verhalten einer Fachkomponente, wie beispielsweise Zugriffsmethoden auf zentrale Attribute oder einfache Berechnungen. Der Kern stellt somit den „gemeinsamen Nenner“ sämtlichen Rollenverhaltens dar.

Rollenkomponenten realisieren rollen-spezifisches Verhalten basierend auf der Funktionalität der Kernkomponente oder der Rollenkomponenten zwischenliegender Schichten. Vor allem sind Rollenkomponenten nicht nur reine Adaptoren zwischen den Schnittstellen, sondern enthalten in den meisten Fällen einen Anteil der fachlichen Logik.

Die Aufteilung der Rollenkomponenten in einzelne Schichten hilft bei der Integration der unterschiedlichen Delegationsketten. Die Identifizierung einer zentralen Kernkomponente stellt einen einfachen Ansatz für die Zusammenführung unterschiedlicher Abstraktionen dar. Beide Mechanismen werden in den folgenden Absätzen erläutert.

3.6.2 Integration von Rollenkomponenten

An einem einfachen Beispiel wird nun gezeigt, wie die Rollenkomponenten aus unterschiedlichen Komponentenframeworks zusammengefügt werden. Dazu seien drei Rollen aus drei unterschiedlichen Komponentenframeworks gegeben, die gemeinsam von derselben Fachkomponente gespielt werden sollen. Das Verhalten jeder einzelnen Rolle ist zerlegt in eine Reihe weiterer interner Rollen, die durch entsprechende Rollenkomponenten implementiert sind. Konkret ergeben sich auf diese Weise drei Delegationsketten, und zwar in diesem Beispiel für die erste Rolle r_1, r_2, r_3, r_4, für die zweite Rolle r_1', r_3 und schließlich für die dritte Rolle r_1'', r_2''. Dabei wurden bereits gemeinsame interne Rollen für die erste und zweite Rolle identifiziert und entsprechend umbenannt.

Die richtige Zusammenstellung der Rollenkomponenten ist weitgehend durch die Vorgaben des Zwiebelschalenmodells bestimmt:

- Die erste Rolle einer jeden Delegationskette befindet sich stets in der obersten Schicht.

- Die längste Delegationskette bestimmt die Anzahl an Schalen im Modell.
- Rollenverhalten einer Schicht kann über die Einführung sogenannter *Dummykomponenten* auf höhere Schichten geführt werden.

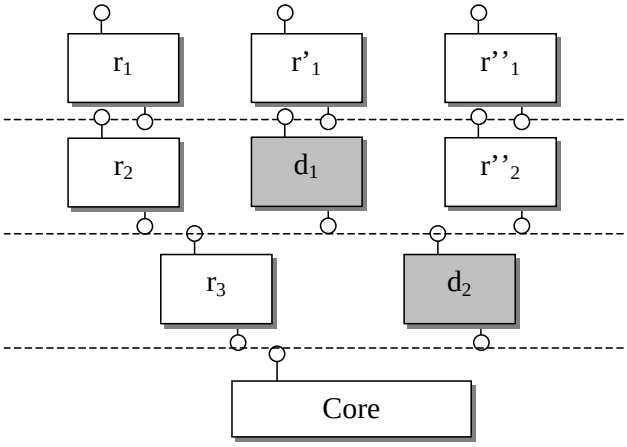


Abbildung 3.34: Zusammenführung dreier Delegationsketten

Mit Hilfe dieser Richtlinien werden die Delegationsketten zu den drei Rollen des Beispiels so zusammengeführt, wie in Abbildung 3.34 dargestellt. Durch die erste Rolle ist die Anzahl der Schichten auf vier festgelegt. Die abschliessende Rollenkomponente r_4 charakterisiert die Kernkomponente des Beispiels und wurde daher in der Abbildung mit $Core$ bezeichnet.

In der obersten Schicht befinden sich nun die „vordersten“ Rollenkomponenten, also r_1, r_1' und r_1''. Alle weiteren Rollenkomponenten der zweiten und dritten Rolle sind nun geeignet auf die unterschiedlichen Schichten aufzuteilen. Im Fall der zweiten Rolle setzt die Rollenkomponente r_1' unmittelbar auf dem Verhalten der Rollenkomponente r_3 auf. Da im Schichtenmodell nur Rollenkomponenten benachbarter Schichten miteinander kommunizieren dürfen, ist es erforderlich, über eine Dummykomponente d_1 das Verhalten von r_3 auf die nächsthöhere Schicht zu übertragen. Dies ist nur dann möglich, wenn eine entsprechende Typkomposition in dieser Schicht erfolgreich verläuft. Die Zusammenführung dieser beiden Delegationsketten ist in diesem Beispiel denkbar einfach, da durch eine gemeinsame Rollenkomponente beide Verhaltensspezifikationen auf einer gemeinsamen Basis beruhen. Ebenso erfordert die Integration der dritten Rolle eine weitere Dummykomponente, um das Verhalten der Rollenkomponente r_2'' mit dem Verhalten der Kernkomponente abzustimmen.

Mit Hilfe dieser zusätzlichen Konzepte und dem Verständnis der Zusammenführung kann nun der Begriff des Komponentenframeworks gerade im Hinblick auf die Wiederverwendung von Implementierungsbestandteilen präzisiert werden:

> **Erläuterung 3.17 (Komponentenframework)** Ein Komponentenframework ist ein unvollständiges Anwendungssystem, das auf einem klaren, durch ein Fachmodell repräsentierten Verständnis eines Anwendungsbereichs basiert. In der Spezifikation eines Komponentenframeworks sind eine Reihe von Rollen definiert mit Aussagen über das Verhalten der jeweiligen Fachkomponenten. Die Verhaltensspezifikation berücksichtigt dabei extern beobachtbare Kommunikation mit anderen Fachkomponenten ebenso, wie die Auswirkungen auf den internen Zustand der Fachkomponente, wodurch eine Integration mit anderen Komponentenframeworks ermöglicht wird. Die Implementierung der Fachkomponenten ist durch die Bereitstellung geeigneter Rollenkomponenten vorgegeben und muß durch den Anwender des Frameworks im Verlauf der Konfiguration durch passende Kernkomponenten vervollständigt werden.

Dieser Erläuterung liegt die Idealvorstellung zugrunde, daß die Integration mehrerer Komponentenframeworks anhand der jeweils gewählten internen Schnittstellen reibungslos verläuft. Dann ist es für den Anwender ausreichend, die mitgelieferten Rollenkomponenten geeignet zusammenzuführen. Die entstehende „halbfertige“ Anwendung erfordert nun lediglich die Implementierung der spezifizierten Kernkomponenten. Zeigen sich jedoch bei der Integration Konflikte in den Annahmen über das Verhalten einer Fachkomponente, so ist es erforderlich, bereits auf der Ebene der Spezifikation einzugreifen. In diesem Fall muß der Entwickler neue Greybox-Komponententypen definieren und entsprechende Rollenkomponenten implementieren.

3.6.3 Abbildung von Abstraktionsbeziehungen

Bei der Integration von Fachmodellen ist die Situation diskutiert worden, daß unterschiedliche Rollen der einzelnen Modelle unterschiedliche Abstraktionen einer Entität implizieren (siehe Abschnitt 3.3.9). Als Beispiel zeigte sich bei der Integration der beiden Komponentenframeworks für die Anwendung der OOPSLA-Einreichung die Entität des Autors: während im einen Modell Autoren als Gruppe betrachtet werden, ist im zweiten Fachmodell die genaue Unterscheidung einzelner Personen wichtig, um Autoren als Gutachter einsetzen zu können. Der Zusammenhang zwischen unterschiedlichen Entitäten unterschiedlicher Abstraktionsstufen führt im integrierten Fachmodell zu Abstraktionsbeziehungen zwischen den Rollen, die diese Entitäten repräsentieren.

Die Tatsache, daß Abstraktionsbeziehungen auf der Ebene einzelner Rollen definiert werden, kann zu Verwirrungen führen, da es sich eigentlich um eine Beziehung handelt, die

vom jeweiligen Rollenverhalten unabhängig ist. Es handelt sich vielmehr um eine Beziehung zwischen den durch die Rollen repräsentierten Entitäten und betrifft damit sämtliche Rollen einer Entität. Aus diesem Grund erscheint es naheliegend, Abstraktionsbeziehungen in der Implementierung auf die Kernkomponenten der einzelnen Fachkomponenten zu übertragen. Dafür spricht zudem eine weitere Tatsache: da es sinnvoll ist anzunehmen, daß jedes einem Komponentenframework zugeordnete Fachmodell ohne Abstraktionsbeziehungen auskommt, treten solche Beziehungen erst bei der Integration von Komponentenframeworks auf. Da die Konfiguration eines integrierten Komponentenframeworks entsprechend dem letzten Abschnitt aus der Bereitstellung geeigneter Kernkomponenten besteht, kann der Entwickler an dieser Stelle Abstraktionsbeziehungen einfach auflösen, ohne die bestehende Implementierung des Komponentenframeworks modifizieren zu müssen.

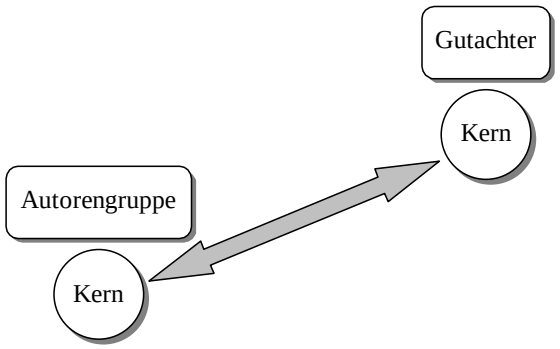


Abbildung 3.35: Auflösung von Abstraktionsbeziehungen

An dieser Stelle lohnt eine Hervorhebung der unterschiedlichen Integrationsarten. Im Fall einer *Early Integration* böte sich die Gelegenheit, das integrierte Fachmodell so anzupassen, daß die unterschiedlichen Abstraktionsebenen geeignet zusammengeführt wären. Im vorliegenden Beispiel könnte ein Entwickler beispielsweise eine gemeinsame Rolle `Person` einführen, auf die sich im weiteren die Rollen `Autorengruppe`[6] und `Gutachter` abstützen würden. Bei einer *Late Integration*, wie sie bei der Zusammenführung von Komponentenframeworks auftritt, existieren bereits Teile einer Implementierung, die in dieser Form nicht weiterverwendet werden könnten. Der Entwickler muß stattdessen die Kernkomponenten der beiden Fachkomponenten miteinander verbinden (siehe Skizze in Abbildung 3.35). Im Beispiel ist es erforderlich, daß jeder Gutachter weiß, an welchen

[6] In diesem Beispiel wurde für ein besseres Verständnis die Rolle `Autor` umbenannt in die Rolle `Autorengruppe`

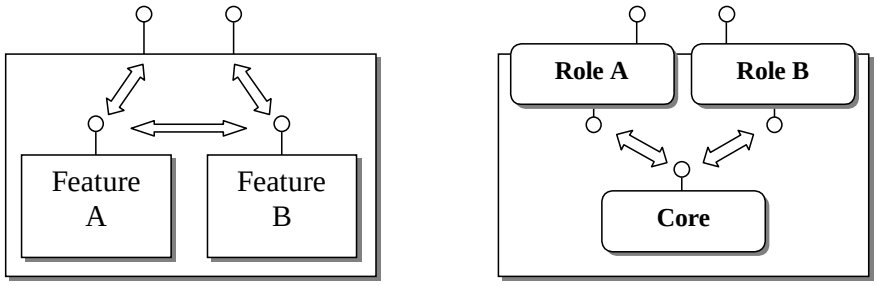
Einreichungen er oder sie selbst als Verfasser auftritt. Dieses Wissen ist in der Kernkomponente der Fachkomponente vermerkt, die die Rolle `Gutachter` spielt. Greifen Rollenkomponenten auf diese Information zu, so kommuniziert die Kernkomponente mit den entsprechenden Kernkomponenten der Rolle `Autorengruppe`.

3.6.4 Zusammenfassung

Ein wichtiges Ziel bei der Konzeption von Komponentenframeworks liegt in der Wiederverwendung bestehender Artefakte der Implementierung. In den vergangenen Abschnitten wurde gezeigt, wie das Rollenverhalten in eine Kette einzelner Greybox-Komponententypen zerlegt und mit Hilfe von mehreren Rollen- und einer Kernkomponente realisiert wird. Auf diese Weise wird einerseits die Integration begünstigt, da die Zusammenführung unterschiedlicher Rollen entweder anhand einer gemeinsamen Rollenkomponente erfolgt oder die Erstellung einer integrierenden Rollenkomponenten erst spät in den jeweiligen Delegationsketten und damit auf einem höheren Abstraktionsniveau notwendig ist (im Idealfall erst bei der gemeinsamen Kernkomponente). Andererseits vermittelt die detaillierte Zerlegung ein klares Verständnis über das Rollenverhalten.

Das vorgestellte Zwiebelschalenmodell impliziert eine Strukturierung der einzelnen Delegationsketten und zwingt bei der Integration zu der Auseinandersetzung mit Verhaltenskonflikten auf jeder der vorgesehenen Schichten. Von der äußersten Schicht bis zur innersten zeigt sich ein steigender Abstraktionsgrad, der es ermöglicht, rollenspezifisches, konkretes Verhalten (z.B. konkrete Protokollvorgaben) von allgemeinem, entitätenspezifischem Verhalten abzugrenzen. So stellt die Kernkomponente die Implementierung eines zentralen, gemeinsamen Basisverhaltens dar (betreffend des gemeinsamen Zustands aller Rollen) und ermöglicht die einfache Konsistenzerhaltung von Entitäten unterschiedlicher Abstraktionsniveaus.

Die Besonderheiten dieses Ansatzes zeigen sich im Vergleich mit den Arbeiten zu „Feature Oriented Programming“, dessen Grundlagen beispielsweise in [Pre97b] beschrieben werden. Die Vorstellung von „Features“ ist nahe verwandt mit dem Konzept der Rolle und so zeigen sich bei der Komposition von Features ganz ähnliche Problemstellungen wie sie in Abschnitt 3.5.4 im Zuge der Zusammenführung von Rollen diskutiert wurden. Prehofer beschreibt einen Delegationsmechanismus, der die isolierte Implementierung und Wiederverwendung einzelner Features ermöglicht. Die entstehenden Klassen stellen dabei einen Teil der Realisierung all jener Objekte dar, für die dieses Feature vorgesehen ist. Abhängigkeiten zwischen den einzelnen Features werden über dieselben Schnittstellen kommuniziert, über die das Featureobjekt seine reguläre Funktionalität anbietet (vgl. linke Seite in Abbildung 3.36). Auf diese Weise können Implementierungen einzelner Features prinzipiell unabhängig von einer späteren Komposition wiederverwendet werden,



Feature Oriented Programming Greybox-Komponententypen

Abbildung 3.36: Features und Rollen im Vergleich

sofern sich über die angebotene Schnittstelle alle relevanten Zustände des Featureobjekts beeinflussen lassen.

Im Gegensatz dazu bietet die Einführung interner Schnittstellen (siehe rechte Seite in Abbildung 3.36) dem Entwickler die Möglichkeit, für das Zusammenspiel zwischen den Rollenkomponenten andere, unter Umständen mächtigere Funktionalität anzubieten, als sie durch die Rollendeklaration gegeben ist. Dies erfordert eine bewußte Auseinandersetzung mit möglichen Abhängigkeiten und beeinflußt damit deutlicher die Möglichkeiten bei einer späteren Integration. Im schlimmsten Fall ist eine Überarbeitung der Schnittstellen und damit eine neue Implementierung der zugehörigen Rollenkomponente nötig.

Es zeigt sich ebenfalls ein unterschiedlich gearteter Kontrollfluß in beiden Ansätzen. Im Rahmen des Feature Oriented Programmings sorgt eine Zusammenstellung benötigter „lifter“ für die korrekte Delegation auf die einzelnen Features. Im Rollenansatz erfolgt die Abstimmung anhand einer gemeinsamen Basiskomponente, wobei lediglich die für die Integration relevanten Informationen berücksichtigt werden müssen, die von den einzelnen Rollenkomponenten über ihre internen Schnittstellen kommuniziert werden. Zudem ermöglichen interne Schnittstellen die Erkennung logischer Konflikte, wie bereits in früheren Abschnitten beschrieben.

3.7 Zusammenfassung und Ausblick

In diesem Kapitel wurden die grundlegenden Mechanismen und Konzepte vorgestellt, um bestimmte Aspekte eines Anwendungssystems in Form eines Komponentenframeworks

zu isolieren und in einem anderen Kontext wiederzuverwenden. Ein Komponentenframeworks ist dabei eng verwoben mit einem Fachmodell, das ein konzeptuelles Verständnis des Anwendungsbereichs vermittelt und Grundlage für die Integration mit weiteren Komponentenframeworks darstellt. Die Erfahrungen aus der Integration auf dieser relativ abstrakten Ebene ermöglichen eine direkte und einfache Integration der jeweiligen Implementierungen durch eine geeignete Zusammenstellung der einzelnen Rollenkomponenten.

Durch die individuelle Zuordnung einer Softwarearchitektur zu einem Komponentenframework bietet sich die Möglichkeit, die Organisation und Struktur der beteiligten Komponenten in einer für diese Sicht optimalen Weise festzulegen. Die Softwarearchitektur ist somit nicht mehr für das Gesamtsystem vorgeschrieben, sondern wird als mehrschichtige Organisation eines Anwendungssystems verstanden (ebenso wie in [Kru99]). Die Einführung langlebiger Frameworkkomponenten erlaubt die Überwachung der jeweiligen Vorgaben zur Laufzeit und entlastet zudem die beteiligten „leichtgewichtigen“ Fachkomponenten, die für den Anwender eines Frameworks mit vertretbarem Aufwand zu realisieren sind.

Die Realisierung von Fachkomponenten wurde in den vergangenen Abschnitten zudem durch ein klares Implementierungsmodell vereinfacht, das das Zusammenwirken einer Kernkomponente mit unterschiedlichen Rollenkomponenten beschreibt. Viele dieser Rollenkomponenten können bereits mit einem Komponentenframework mitgeliefert werden, wodurch sich idealerweise der Aufwand bei der Instantiierung auf die Realisierung der geforderten Kernkomponente reduziert. An dieser Stelle bietet sich dem Entwickler die Möglichkeit zur fachlichen Zusammenführung der beteiligten Komponentenframeworks und zur Abstimmung mit anderen Kernkomponenten aufgrund redundanter Modellinformationen.

Eine Erweiterung der vorgestellten Konzeption, die auch Beziehungen entsprechende „Rollen“ zuweist (vgl. Abschnitt 3.3.6) würde die Abstraktion von Beziehungen ebenso wie die Spezifikation von Beziehungen zwischen Beziehungen ermöglichen. Auf diese Weise könnte die Integration von Fachmodellen nicht nur auf Rollen- sondern gleichfalls auf Beziehungsebene stattfinden — Komponentenframeworks ließen sich auch zusammenführen, wenn sie auf den definierten Fachkomponenten dieselben oder auch ähnliche Beziehungen realisierten. Damit könnten gleichfalls Abhängigkeiten zwischen Komponentenframeworks definiert und bei der Komposition ausgewertet werden. Nicht zuletzt wäre damit auch die Möglichkeit gegeben, Komponentenframeworks selbst flexibel zu erweitern.

Kapitel 4

Technische Umsetzung

Im letzten Kapitel wurde ein formales Sytemmodell vorgestellt, das essentielle Konzepte im Umfeld von Komponenten und Komponentenframeworks erfaßt und ein klares Verständnis der Zusammenhänge ermöglicht. Dabei wurden jedoch eine Reihe von Vereinfachungen vorgenommen bzw. Abstraktionen geschaffen, um die Komplexität des Modells handhabbar zu gestalten. Um nun die Praxistauglichkeit — beispielsweise der unterschiedlichen Integrationsmechanismen — zu untersuchen, werden wir in diesem Kapitel der Frage nachgehen, wie sich die erarbeiteten Konzepte auf konkrete, komponentenbasierte Anwendungen abbilden lassen.

Dazu werden gängige Komponententechnologien kurz vorgestellt und hinsichtlich der Ergänzung um neue Bestandteile, wie z.B. Komponentenframeworks oder Rollenkomponenten untersucht. Auf diese Weise erarbeiten wir gleichermaßen Anforderungen an die technische Infrastruktur für komponentenbasierte Systeme und hinsichtlich einer adäquaten, entwicklungsbegleitenden Werkzeuglandschaft.

Ein weiterer Gesichtspunkt macht die Untersuchung der Konzepte auf der Ebene der konkreten Realisierung interessant. Die meisten Ansätze, die sich mit dem Themenfeld „separation of concern“ beschäftigen, fokussierten die Integration konkreter Quelltexte. Das hängt einerseits damit zusammen, daß sich technische Aspekte (z.B. Persistenz oder Logging) leichter als fachliche Belange isolieren und in einem anderen Umfeld wiederverwenden lassen (vgl. AOP [KLM+97]). Andererseits ist für die Mehrzahl der Entwickler die Implementierungssprache noch immer lingua franca, und neue Ideen lassen sich auf dieser Ebene am besten ausprobieren. Zusätzlich wird in diesem Kapitel daher untersucht, wie die modellbasierte Integration im Umfeld solcher quelltextnahen Ansätze zu bewerten ist.

4.1 Der Weg zur Implementierung

Getrieben von dem Wunsch nach einer effizienteren, fehlerfreien Methode der Softwareentwicklung und mit dem Aufkommen leistungsfähiger Entwicklungswerkzeuge verstärkte sich in den letzten Jahren der Trend, die konkrete Implementierung einer Anwendung unmittelbar aus einer „höheren“ Spezifikationssprache — zumindest in großen Teilen — abzuleiten. Diese Idee erscheint naheliegend, da sich viele Details der Realisierung schematisch aus einzelnen, zentralen Designentscheidungen ergeben. Letztendlich ist es das erklärte Ziel, die „Programmierung“ von Anwendungssystemen vollständig auf eine abstraktere Ebene zu verlagern und das Wissen um die korrekte Abbildung in den Quelltext, sowie den Quelltext selbst möglichst vor dem Entwickler zu verbergen. Auf diese Weise kann eine Konsistenz zwischen der Spezifikation und der Implementierung gewährleistet werden, da der Entwickler den Quelltext selbst nicht mehr modifizieren muß.

Dadurch begründet sich auch das hohe Interesse an generativen Mechanismen, die, ausgehend von einer Systemspezifikation z.B. in Form annotierter UML-Diagramme, korrekten Quelltext produzieren. Jedoch sind die Möglichkeiten, auf abstrakter Ebene Implementierungsdetails zu spezifizieren, oft ungenügend und der Generierungsprozeß für viele dynamische Anforderungen zu starr. Nicht zuletzt aus diesen Gründen werden generative Ansätze großteils im Bereich des *Rapid Prototyping* oder in der Erstellung genau umrissener Anwendungsfamilien (vgl. [CE00]) eingesetzt.

Ein wichtiges Merkmal der Softwareentwicklung mit integrierbaren Komponentenframeworks ist die Durchgängigkeit der Modellierung: Entitäten der fachlichen Modellierung und ihre Abhängigkeiten untereinander finden sich im Design eines Komponentenframeworks ebenso wie in dessen Realisierung wieder. Dabei besteht die Implementierung einer Anwendung nicht nur aus Komponenten, denen in der realen Welt eine Bedeutung zukommt. Vielmehr sind es nur diese Komponenten eines Frameworks, die für den Anwender eine Bedeutung haben und sichtbar sind. Daneben kann ein Framework eine Reihe weiterer „technischer“ Komponenten einbringen, die für die Realisierung des Zusammenspiels wichtig sind.

Für die weiteren Betrachtungen unterscheiden wir daher zwei wichtige Anwendungsfälle der Softwareentwicklung mit Komponentenframeworks:

Frameworkentwicklung: Bei der Erstellung eines Komponentenframeworks liegt der weitaus größere Teil des Aufwands in der Erstellung der Verhaltensspezifikationen für die einzelnen Rollen, der Zerlegung in passende Greybox-Komponententypen sowie in der Definition von Meditatoren, die das Zusammenspiel der einzelnen Fachkomponenten reglementieren. Anschließend werden die entsprechenden Komponenten (Rollenkomponenten, Mediatoren) implementiert und bei Bedarf durch

weitere technische Komponenten ergänzt. Die entstehenden Bestandteile der Implementierung werden nun zu Paketen zusammengefaßt, die auf einfache Weise weitergereicht werden können.

Da die Implementierung neben einer ausführlichen Dokumentation den wichtigsten Bestandteil eines Komponentenframeworks ausmacht, ist es entscheidend, die essentiellen Konzepte adäquat zu repräsentieren. So benötigt ein Frameworkentwickler vor allem Mechanismen für die präzise Definition von Schnittstellen, die Kapselung von Implementierungsdetails und die Festlegung korrekter Wege der Instantiierung.

Frameworkverwendung: Bevor ein Komponentenframework instantiiert wird, erfolgt eine Integration mit weiteren Komponentenframeworks. Obwohl der eigentliche Vorgang der Integration in weiten Teilen auf den Ebenen der fachlichen Modellierung sowie konkreter Spezifikationen des Entwurfs abläuft, werden im Ergebnis direkte Aussagen über die Zusammenführung der jeweiligen Implementierungen getroffen. Dies erfordert die Komposition von Rollenkomponenten im Rahmen einer Fachkomponente und die Erstellung geeigneter Kernkomponenten. Isbesondere kann es notwendig sein, unmittelbar in die Implementierung eines Komponentenframeworks einzugreifen, um beispielsweise zwei im Konflikt stehende Rollenkomponenten durch eine neue Rollenkomponente zu ersetzen.

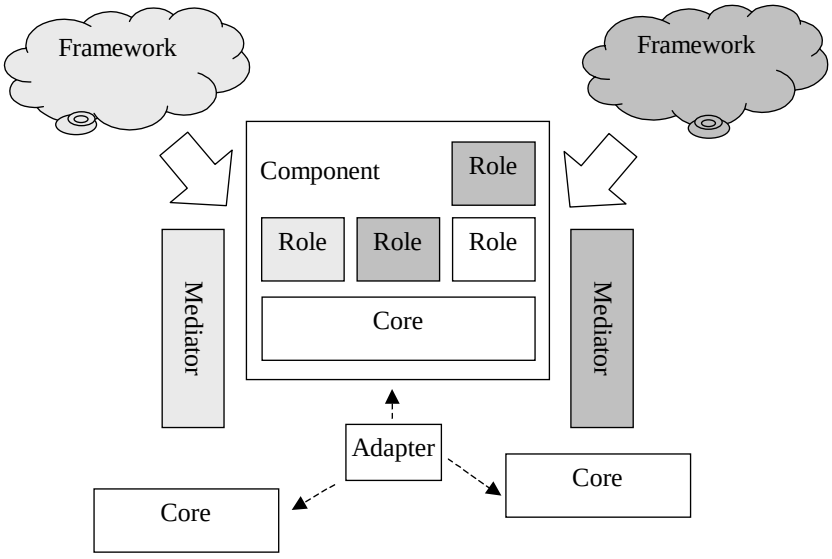


Abbildung 4.1: Zusammenbau einer Komponente

Abbildung 4.1 skizziert ein Szenario, in dem zwei Komponentenframeworks zusammengefügt und zu einem vollständigen Anwendungssystem instantiiert und konfiguriert werden. Jedes der beiden Komponentenframeworks liefert eine Reihe bereits realisierter Komponenten, die entweder die Realisierung der jeweiligen Rollen repräsentieren oder in Form von technischen Komponenten ein Teil der internen Realisierung des Komponentenframeworks darstellen. Nach einer erfolgreichen Integration der beiden Komponentenframeworks auf der Spezifikationsebene müssen nun vom Entwickler die verbleibenden Bestandteile der Implementierung fertiggestellt werden (in Abbildung 4.1 durch weiße Kästchen symbolisiert):

- Für jede Fachkomponente ist eine passende Kernkomponente zu realisieren, die den Schnittstellen der Rollenkomponenten genügt.
- Für Kernkomponenten unterschiedlicher Abstraktionsniveaus sind geeignete Adapter vorzusehen, die eine Abstimmung zwischen den Kernkomponenten sicherstellen.
- Unter Umständen ist es erforderlich, noch weitere Rollenkomponenten zu realisieren, die zwischen einer Rollenkomponente eines Frameworks und der Kernkomponente vermitteln (vgl. Abschnitt 3.6.2). Demgegenüber kann es ebenfalls erforderlich sein, eine vom Framework angebotene Rollenkomponente zu verwerfen und selbst geeignet zu realisieren.

Im Kapitel 3 werden alle „höheren“ Konzepte der Softwareentwicklung mit Komponentenframeworks auf die regulären Bestandteile einer Komponententechnologie (Komponente, Schnittstelle, Kanal) abgebildet. Da sich auch die Integration von Fachmodellen und zugehörigen Spezifikationen unmittelbar auf die Realisierung überträgt, erscheinen generative Ansätze vielversprechend. Auch erfordert die Integration von Komponentenframeworks — wie gerade gezeigt — in vielen Fälle zusätzliche Handarbeit auf der Ebene der Implementierung, weshalb wir im weiteren einen anderen Ansatz verfolgen, der Komponenten der Anwendung mit semantischen Informationen anreichert. Auf diese Weise werden abstrakte Konzepte auch in der Realisierung repräsentiert, wodurch insbesondere dem Entwickler ein Verständnis der Zusammenhänge vermittelt wird, das eine korrekte Erweiterung der integrierten Komponentenframeworks erleichtert. Dieser Ansatz, der auch in vergleichbaren Arbeiten verfolgt wird (siehe [MH01]) bietet die folgenden Vorteile:

- Die Unterstützung des Entwicklers durch geeignete Werkzeuge kann umfassender gestaltet werden: bei der Zusammenstellung von Komponentenframeworks werden bestehende Strukturen analysiert und die Korrektheit der integrierten Anwendung

sichergestellt. Insbesondere die vielfältigen Abhängigkeiten zwischen den einzelnen Rollenkomponenten können verfolgt und beispielsweise graphisch visualisiert werden.

- Durch das erweiterte Wissen über die Zusammenhänge der Bestandteile der Anwendung bietet sich die Möglichkeit, dynamische Veränderungen zuzulassen. Beispielsweise könnte die Zuordnung von Rollen zu Fachkomponenten so geartet sein, daß Rollenkomponenten erst bei Bedarf hinzugefügt werden, um ein gewünschtes Verhalten der Komponente zu ermöglichen. Die Infrastruktur muß in diesem Fall beurteilen können, nach welchen Rollen sich eine Komponente verhalten kann und wie die Komposition der entsprechenden Rollenkomponenten vorzunehmen ist.

Die Erweiterung der Implementierungsebene um High-Level-Konzepte bietet eine enge Verknüpfung zwischen Spezifikation und Realisierung, fördert somit das Verständnis des Designs und erlaubt eine Integration von Komponentenframeworks auf den unterschiedlichen Abstraktionsniveaus. Eine solche Erweiterung geht Hand in Hand mit einer umfassenden Werkzeuglandschaft, die den Entwickler und den Anwender eines Komponentenframeworks in allen Phasen der Anwendungserstellung unterstützt (siehe Abschnitt 4.6).

4.2 Komponententechnologien

Grundlage für jede komponentenbasierte Anwendung ist eine geeignete Komponententechnologie, die den Aufbau und das Zusammenspiel der einzelnen Bausteine festlegt. Die Komplexität und Flexibilität dieser Technologie beeinflussen maßgeblich die Qualität der entstehenden Anwendung sowie die Wiederverwendbarkeit ihrer Bestandteile. Entscheidet sich der Entwickler beispielsweise für die Verwendung des DCOM Komponentenstandards, so ist die Wiederverwendung der Komponenten fast ausschließlich auf Microsoft-Betriebssysteme beschränkt.

Unter dem Begriff Komponententechnologie verstehen wir im weiteren lediglich die Spezifikation der verwendeten Konzepte — konkrete Implementierungen sollen hier nicht betrachtet werden, auch wenn sie für die konkrete Entwicklung bedeutsam sind. In den folgenden Abschnitten werden drei wichtige Komponententechnologien vorgestellt und ihre dominanten Charakteristika diskutiert.

Enterprise Java Beans

Enterprise Java Beans ist eine Komponenteninfrastruktur, die die einfache, zügige Entwicklung verteilter, skalierbarer Anwendungssysteme mit Java unterstützt. Eine zentrale Idee besteht darin, die einzelnen Komponenten einer Anwendung von der Verantwortung technischer Querschnittsthemen zu befreien. Eine Anwendung läuft letztlich auf einem sogenannten *Applicationserver*, der die jeweiligen Komponenten verwaltet und übergreifende Dienste zur Verfügung stellt [Rom99]. Hierfür gibt es auf einem Applicationserver ein oder mehrere *Container*, die die Menge der Komponenten organisieren und verwalten. Jede Komponente ist genau einem Container zugeordnet.

EJBs stehen unabhängig von dem konkreten Applicationserver, auf dem sie ausgeführt werden, mindestens die folgenden Dienste zur Verfügung:

- Transaktionsmanagement
- Sicherheitsdienst
- Resourcenverwaltung
- Komponentenverwaltung
- Persistenz
- Ortstransparenz

Die meisten dieser Dienste können für eine Komponente transparent genutzt werden. Das bedeutet, daß die Dienste nicht zwangsläufig über bestimmte Schnittstellen gesteuert werden müssen. Ein Beispiel findet sich im Persistenzmechanismus, der entweder über die EJB-Infrastruktur („container-managed“) oder über die Komponente selbst („bean-managed“) realisiert werden kann.

Die EJB-Spezifikation unterscheidet zwei Arten von Enterprise Beans — vornehmlich anhand der Lebensdauer: Einerseits dienen sogenannte *Entity Beans* dazu, einen langlebigen Zustand zu kapseln und dauerhaft persistent zu halten, andererseits existieren *Session Beans*, die nur für die Dauer einer konkreten Anfrage existieren. So enthält denn auch die gängige Vorstellung einer EJB-Anwendung eine Reihe von Session-Beans, die einzelnen Geschäftsprozessen entsprechen und auf konkreten Daten (repräsentiert durch Entity-Beans) operieren.

CORBA Komponenten

Das *CORBA Component Model* (kurz: CCM) ist eine Spezifikation der *Object Management Group* (kurz: OMG) für serverseitige, sprachneutrale Komponenten, die nahtlos an die existierenden CORBA Standards anknüpft [CCM99]. Aufbauend auf diesem Middleware-Standard stehen den CORBA Komponenten somit eine Reihe nützlicher Dienste zur Verfügung, wie beispielsweise Transaktionsmanagement, Persitenzmechanismen oder die eventbasierte Kommunikation. Die Konzeption für CCM wird ein elementarer Bestandteil der CORBA 3.0 Spezifikation sein, die im Laufe des Jahres 2001 fertiggestellt wird.

Das CCM wurde in vielen Belangen durch den EJB-Standard beeinflusst und so finden sich auch in der CCM-Architektur bekannte Bestandteile[1]: Jede CORBA Komponente läuft innerhalb eines *Containers*, der die Zugriffe reglementiert und das Zusammenspiel der Komponenten organisiert. Es wird unterschieden zwischen *Entity Containers*, *Session Containers*, *Service Containers* und nicht weiter klassifizierten Containern. Die Unterscheidung erfolgt anhand bestimmter Eigenschaften der verwalteten Komponenten, die z.B. zustandsbehaftet oder zustandslos sind (und dementsprechend in analoge Klassen unterteilt werden). Zugriff auf eine CORBA Komponente erfolgt über ein assoziiertes *Home Interface*, das Komponenten über einen Namensdienst oder andere Komponenten erlangen können. Container wiederum gehören zu einem *Application Server*, der die zentrale Infrastruktur zur Verfügung stellt und so zum Beispiel die unterschiedlichen Dienste der CORBA-Middleware vermittelt.

Essentielle Unterschiede zum EJB-Standard zeigen sich bei einem deutlich mächtigeren Schnittstellenkonzept. CORBA Komponenten bieten über sogenannte *Ports* Dienste an oder nehmen von anderen Komponenten Dienste in Anspruch. Komponenten können mehrere Ports definieren und diese entweder als *facet* oder als *receptacle* deklarieren (siehe nachfolgende Abschnitte). Jede Komponente muß bestimmte Standardports anbieten, beispielsweise um dem jeweiligen Container Möglichkeiten zur Aktivierung oder Deaktivierung zu bieten. Für die eventbasierte Kommunikation sind weiterhin bestimmte Ports vorgesehen, die es einer Komponente erlauben, bestimmte Events zu versenden oder Events einer festgelegten Art zu empfangen (*event source* bzw. *event sink*). Die Schnittstelle einer Komponente definiert alle ihre Ports und weist ihnen eindeutige Bezeichner zu.

Mit diesem Ansatz schafft die OMG ein Umfeld, das die Erstellung und Verwendung komponentenbasierter Systeme deutlich vereinfacht: Kommerzielle CCM Produkte bieten Entwicklern standardisierte Möglichkeiten, um Komponenten zu definieren, zu realisieren und (evtl. dynamisch) zu konfigurieren und in eine Anwendung einzubinden (vgl.

[1] Tatsächlich kann CCM als eine Obermenge von EJB verstanden werden [WSO00].

[WSO00]). Wie jedoch bereits an der bisherigen Versionen des CORBA-Standards zu beobachten war, setzt sich diese Technologie nur allmählich im industriellen Alltag durch — zu langsam erreicht das Wissen die Entwickler und zu lange dauert es, bis die Konzepte in tragfähige Produkte umgesetzt sind.

DCOM

Das *Distributed Component Object Model* (kurz: DCOM) ist eine proprietäre Technologie, die als Erweiterung von Microsofts Komponentenlösung für Windows-Betriebssysteme das Zusammenspiel mehrerer Komponenten auf verteilten Computern ermöglicht. DCOM ist die Grundlage für eine Reihe von Produkten und Frameworks aus dem Haus Microsoft. Durch die weite Verbreitung dieser Technik zusammen mit den Betriebssystemen von Microsoft sowie aufgrund der Verfügbarkeit vieler nützlicher Komponenten (z.B. aus den Office-Paketen) findet DCOM hohen Anklang und kann zu recht als „Quasi-Standard“ bezeichnet werden.

Für die Entwicklung unternehmensweiter Anwendungen wurden um die Komponententechnologie DCOM weitere Produkte gruppiert, die unter der Bezeichnung „Back Office“ zusammengefaßt sind: Der Microsoft Transaction Server (kurz: MTS) dient einer komponentenbasierten Anwendung als Application Server und bietet eine Reihe technischer Dienste, wie Transaktionsmanagement, Resourcenverwaltung und Persistenzmechanismen.

4.3 Kernpunkte einer Umsetzung

Im folgenden beleuchten wir einzelne Aspekte des formalen Sytemmodells aus Kapitel 3 und diskutieren, auf welche Weise eine Umsetzung auf eine der vorgestellten Komponententechnologien praktikabel erscheint. Hierfür werden die vorgestellten Komponententechnologien hinsichtlich der angebotenen Möglichkeiten untersucht.

4.3.1 Komponentenparadigma

Eine Vereinfachung des formalen Modells besteht darin, alle Entitäten eines Anwendungssystems als Komponenten zu betrachten. Daraus ergibt sich nicht nur ein klares Verständnis der Konzepte sondern auch eine einheitliche Handhabung. Auf dem Weg zur technischen Realisierung wächst jedoch die Menge der Anforderungen an die Fähigkeiten

einer Komponente. Um für moderne, verteilte Anwendungssysteme tauglich zu sein, müssen Komponenten daher über Rechnergrenzen hinweg kommunizieren können. Weiterhin sind ein transparenter Persistenzmechanismus oder auch die Einbindung in übergreifende Transaktionsmechanismen wichtige technische Anforderungen, die die Komplexität einer Komponente deutlich erhöhen.

Jede Komponente in den vorgestellten Komponententechnologien benötigt aus diesem Grund eine ganze Reihe von technischen Konstrukten, die der Erfüllung dieser Basisdienste dienen und den Eindruck vermitteln, daß dieser Aufwand nur für „große“ Module gerechtfertig ist. So werden Komponenten auch oft als Module verstanden, die auf kleineren Modulen — meist Objekten — basieren und diese nach außen abkapseln (siehe [Szy00]). Ein Beispiel hierfür findet sich in der Konzeption von EJBs, die in sich aus einer Reihe von Objekten aufgebaut sein können. Eine solche Unterscheidung ist historisch gewachsen und steigert nur die Komplexität der Entwicklung.

CORBA Komponenten basieren auf dem *CORBA Object Model*, das prinzipiell keine Aussagen darüber trifft, aus welchen Bestandteilen eine Objektinstanz aufgebaut ist. Jedoch wurde in der CCM-Spezifikation das Metamodell dahingegehend erweitert, daß nun „lokale“ Schnittstellen deklariert werden können, die nicht über Rechnergrenzen hinweg aufgerufen oder als Parameter übergeben werden können. Es wird dabei angenommen, daß solche lokalen Schnittstellen ebenfalls durch reguläre Objekte realisiert werden (abhängig von der verwendeten Programmiersprache).

Die Komponentenmodelle der vorgestellten Standards sind zudem flach und unterstützen insbesondere keine hierarchische Zerlegung von Komponenten. Von einer solchen Möglichkeit wurde jedoch in der Konzeption von Komponentenframeworks (siehe Kapitel 3) Gebrauch gemacht, um einerseits die Realisierung von Frameworks als eigenständige Entität zu erleichtern (siehe Abschnitt 3.5.1) und andererseits, um Komponenten in Kern- und Rollenkomponenten aufzuteilen (siehe Abschnitt 3.6).

Beides sind Sonderfälle einer echten hierarchischen Dekomposition von Komponenten: Im ersten Fall wird zwar eine Hierarchie über Komponenten aufgebaut, allerdings wird dabei die Kapselung durchbrochen, die üblicherweise bei der Dekomposition angenommen wird. Der zweite Fall beschreibt ebenfalls eine Dekomposition, jedoch nur einen Schritt von Komponenten zu Subkomponenten — eine tiefere Zerlegung ist hier nicht notwendig. Aus diesem Grund kann das formale Modell wie folgt pragmatisch auf die Konzepte von Objekt und Komponente abgebildet werden (siehe dazu Abbildung 4.2):

- **Komponenten** und **Komponentenframeworks** des formalen Modells werden auf Komponenten der technischen Plattform abgebildet (repräsentiert durch Rechtecke

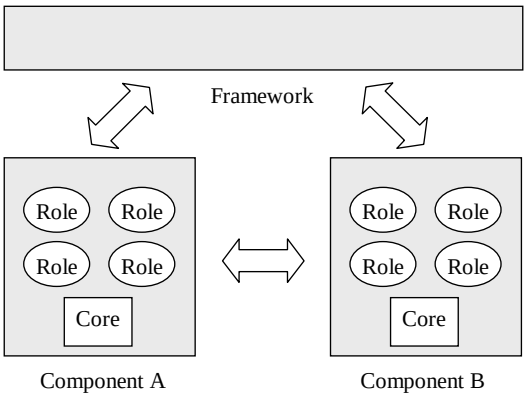


Abbildung 4.2: Aufteilung in Objekte und Komponenten

in Abbildung 4.2). Die Unterscheidung der beiden Konzepte zur Laufzeit ist wichtig, um eine korrekte Verschaltung gewährleisten zu können. Das bedeutet beispielsweise, daß Komponenten mit anderen Komponenten und Komponentenframeworks kommunizieren können — Komponentenframeworks dagegen nur mit Komponenten. Insbesondere schließt das die Möglichkeit aus Abschnitt 3.7 aus, wonach auch Komponentenframeworks Rollen zugeordnet werden, wodurch sie selbst in der Lage sind, in anderen Komponentenframeworks mitzuwirken.

- Die **Rollenkomponenten** einer fachlichen Komponente werden als Teil der Realisierung abgebildet auf Objekte der jeweiligen Programmiersprache (in Abbildung 4.2 dargestellt durch Ellipsen). Dies ergibt sich einerseits aus der engen Bindung von Subkomponenten an den Lebenszyklus ihrer Komponente, andererseits auch durch die Kapselung durch die jeweilige Komponente. Die Verwendung von Objekten erschwert jedoch die dynamische Kopplung von Rollenkomponenten. Zu diesem Zweck ist daher ein weiterer Mechanismus in jeder Komponente notwendig, der die Modifikation des internen Aufbaus ermöglicht.

- **Kernkomponenten** stellen den Teil einer Komponente dar, der durch den Anwender bereitgestellt wird. Sie repräsentieren prinzipiell einen Interaktionspunkt einer fachlichen Komponente, der dem Abgleich unterschiedlicher Abstraktionen einer Entität dient (siehe Abschnitt 3.6.3). Die Abbildung auf eine Komponente der technischen Plattform bietet somit den Vorteil einer klaren Schnittstellendefinition und die Möglichkeit der Synchronisation innerhalb der Menge aller Kernkomponenten. Die Auslagerung auf eine eigene Komponente erlaubt es zudem, die Sichtbarkeit der Schnittstelle besser zu reglementieren, da hierfür keine der vorgestellten Komponententechnologien entsprechende Mechanismen anbietet.

Aus dieser Abbildung ergeben sich unterschiedliche Klassen von Komponenten, um die das Komponentenmodell erweitert werden muß. So kann beispielsweise eine korrekte Verschaltung der unterschiedlichen Komponenten zur Laufzeit sichergestellt werden, da die Infrastruktur in der Lage ist, Fachkomponenten, Frameworkkomponenten und Kernkomponenten zu unterscheiden.

4.3.2 Namensräume

Die Komposition von Komponentenframeworks resultiert in der Koexistenz unterschiedlicher Fachmodelle im Anwendungssystem. Jedes dieser Modelle definiert eine Sicht auf die Anwendung, einen *Kontext* (vgl. Rollenbegriff in Abschnitt 2.4). Um die Integrität der Anwendung sicherzustellen, ist es erforderlich, daß diese Sichten in der technischen Umsetzung nicht verloren gehen. Das betrifft vor allem die Navigation über die Schnittstellen und Rollen einer Komponente.

Die Betrachtung der vorgestellten Komponententechnologien zeigt, daß es stets einen Mechanismus gibt, der die Erforschung einer Komponente ermöglicht. CORBA Komponenten bieten dafür bestimmte Operationen in ihrem jeweiligen *equivalent interface*. Ist die Identität einer Komponente erst einmal bekannt (z.B. über eine CORBA Objektreferenz), so können alle ihre Ports in Erfahrung gebracht werden. Auf diese Weise liegt sehr viel Verantwortung beim Entwickler, da leicht zwischen den unterschiedlichen Sichten auf eine Komponente gewechselt werden kann.

Ein pragmatischer und erprobter Weg für die Deklaration von Sichtbarkeitsbereichen sind Namensräume. Über diesen Mechanismus kann festgehalten werden, daß z.B. Rollen nur in dem Kontext bekannt sind, in dem sie definiert wurden. Die Implementierung von Namensdiensten kann — basierend auf diesem Wissen — dafür eine kontextsensitive Suche nach Bezeichnern ermöglichen.

Für die Realisierung von Namensräumen bieten die vorgestellten Komponentenplattformen einen *name service* an. Enterprise Java Beans bedienen sich hierfür eines Standards mit dem Namen *Java Naming and Directory Interface* (kurz: JNDI). Über diese Schnittstelle können den Komponenten einer Anwendung Namen zugeordnet und diese hierarchisch angeordnet werden. Insbesondere können über diesen Mechanismus verschachtelte Namensräume realisiert werden, wie sie durch ein System hierarchisch komponierter Komponentenframeworks gegeben sind. Durch die Möglichkeit, mehrere Namen an dieselbe Komponente zu binden ergibt sich zudem ein einfacher Weg zur Integration durch Identifizierung: Dieselbe Rollenkomponente kann in unterschiedlichen Kontexten durch unterschiedliche Bezeichner angesprochen werden. Ein ganz ähnlicher Mechanismus ist

in der CCM-Spezifikation vorgesehen, der auf dem CORBA *Interoperable Naming Service* basiert.

Über einen Namensdienst kann somit die Organisation der Komponenten unmittelbar aus dem Design abgebildet werden. Andererseits garantiert ein solcher Dienst nicht die korrekte Verschaltung der Komponenten, da Referenzen zwischen den unterschiedlichen Kontexten einer Anwendung ausgetauscht werden können. Um den Elementen eines Kontextes einen bestimmten Sichtbarkeitsbereich zuzuordnen, muß der Namensdienst unterstützt werden durch eine Infrastruktur, die dafür sorgt, daß Referenzen nur im jeweiligen Kontext verwendet werden können.

4.3.3 Kanäle

Im Systemmodell aus Kapitel 3 werden Kanäle als Kommunikationsmedium zwischen zwei Komponenten verwendet, wobei sie nicht zwangsläufig Beziehungen zwischen diesen Komponenten repräsentieren. Das Konzept von Kanälen ist hilfreich, um Kommunikationsabsichten zu erkennen und im Rahmen der Systemstruktur zu reglementieren, welche Komponenten auf welche Weise zusammenwirken können.

Müssen Komponenten erst einen Kanal aufbauen, bevor sie Nachrichten an eine andere Komponente versenden dürfen, so bietet sich der technischen Infrastruktur die Möglichkeit zu überprüfen, ob ein solcher Austausch entsprechend den strukturellen Vorgaben erlaubt ist oder nicht. Ein solcher Mechanismus ist für die Realisierung von Mediatoren notwendig, die Verbindungen zwischen Fachkomponenten reglementieren (vgl. Abschnitt 3.5.2).

Die Komponententechnologien aus Abschnitt 4.2 unterstützen sowohl den traditionellen Methodenaufruf als Kommunikationsform als auch das asynchrone und entkoppelte Versenden und Empfangen von Events über Kanäle. Beide Varianten sind für die Realisierung der Kanäle aus dem formalen Modell weniger geeignet: der direkte Methodenaufruf kann nicht beobachtet bzw. reglementiert werden, und das Eventmodell erlaubt keine Festlegung des jeweiligen Empfängers einer Nachricht.

Einzig die Spezifikation von CORBA Komponenten sieht „erweiterte Komponenten“ vor, die über Kanäle (engl. *connections*) miteinander vernetzt werden. Über Ports spezifiziert eine Komponente ihre Anforderungen und Möglichkeiten hinsichtlich der Verschaltung mit anderen Komponenten. Ein Kanal ist hierbei keine eigene Instanz der Anwendung, sondern lediglich ein Bestandteil der Systemkonfiguration, der einen Kommunikationswunsch zwischen zwei Komponenten repräsentiert. Dabei können grob drei Wege zur Etablierung von Kanälen unterschieden werden:

- Eine Komponente kann zur Laufzeit bei einer anderen Komponente den Aufbau eines Kanals „beantragen“ und bei Erfolg fortan mit dieser Komponente kommunizieren bis der Kanal wieder abgebaut wird.
- Zum Zeitpunkt der Konfiguration eines Anwendungssystems gibt der Entwickler vor, welche Komponenten auf welche Weise durch Kanäle verknüpft sind.
- Eine Gruppe von Komponenten, die als Paket gehandhabt und gehandelt werden besitzen bereits Wissen über ihre initiale Verschaltung untereinander.

Das Port-Konzept des CCM bietet also eine gute Ausgangslage für die Realisierung des Komponentenmodells aus Kapitel 3. Problematisch erscheint jedoch die Tatsache, daß die beteiligten Komponenten selbst die Verantwortung tragen sowohl für die Entscheidung über die Korrektheit eines Kanals als auch hinsichtlich der Verwaltung des Kanals. Ob ein Kanal aufgebaut werden darf oder nicht, ist in unserem Modell eine Entscheidung der beteiligten Komponentenframeworks. Diese sollen zudem in die Lage versetzt werden, Kanäle zwischen Komponenten zu etablieren oder zu modifizieren, ohne die betroffenen Komponenten davon in Kenntnis setzen zu müssen. Eine entsprechende Erweiterung des CCM wird in Abschnitt 4.4 vorgestellt.

In Abschnitt 3.4.1 wurden bidirektionale Kanäle motiviert, um die Charakteristik der Kollaboration zwischen Komponenten deutlich zu machen. Die traditionelle Kommunikationsform in Anwendungssystemen ist jedoch stark dienstorientiert und damit — zumindest hinsichtlich des Kontrollflusses — unidirektional. Das Konzept eines Diensterbringers und eines Dienstnutzers prägt deutlich die Kommunikation zwischen Komponenten und bestimmt in den vorgestellten Komponententechnologien das Zusammenspiel.

Kommunikationsverbindungen zwischen Komponenten repräsentieren (und implementieren) somit in den bestehenden Komponententechnologien Abhängigkeiten zwischen den Komponenten, die im formalen Modell nur auf der Ebene von Rollen modelliert werden. Hier ist ein Kanal auch lediglich ein Mittel der Verschaltung, das die Realisierung dieser Abhängigkeiten beeinflusst. Die Bidirektionalität macht dabei deutlich, daß im Zusammenspiel Informationen in beide Richtungen fließen können, wie es auch beim traditionellen Methodenaufruf der Fall ist. In der Realisierung werden bidirektionale Kanäle auf ein Paar gegengerichtete, unidirektionale Kanäle abgebildet.

Mediatoren, wie sie ebenfalls im vergangenen Kapitel diskutiert wurden, können aufgefaßt werden als (verhältnismäßig intelligente) Kanäle. Im einfachsten Fall nimmt der Mediator von einer seiner Komponenten eine Nachricht an und sendet sie an eine andere Komponente weiter. Die Schnittstelle eines Mediators ist aus diesem Grund ganz analog zu den Kanälen beispielsweise des CORBA Notification Service: während ein Kanal nur die zwei Rollen *publisher* und *subscriber* kennt, bietet ein Meditator je nach Spezifikation eine ganze Reihe unterschiedlicher Schnittstellen, an denen Komponenten eingefügt

werden können. Gleichfalls verwaltet er die Menge der Komponenten, die den jeweiligen Rollen zugeordnet sind — über einen erweiterten An- und Abmeldemechanismus, wie er ebenfalls durch den Notification-Service nahegelegt wird.

4.3.4 Schnittstellen

Im Rahmen dieser Arbeit wird eine strikte Unterscheidung der Konzepte Schnittstelle und Rolle propagiert. Der Begriff der Schnittstelle ist von entscheidender Bedeutung bei der komponentenbasierten Softwareentwicklung, so daß jede Komponententechnologie dieses elementare Konzept vorsieht. Eine Schnittstelle ist dabei stets gekoppelt an das Verständnis von Diensten, die eine Komponente anbietet und sie definiert die konkrete Syntax der entsprechenden Methoden. Schnittstellen werden z.B. in CORBA über die *Interface Definition Language* (kurz: IDL) spezifiziert, während im Umfeld von EJBs *Java Interfaces* zum Einsatz kommen.

Kommunikationsverbindungen zwischen Komponenten knüpfen an die Schnittstellen an und so bezeichnet die Aufteilung der Dienste in mehrere Schnittstellen auch die Möglichkeiten der Verschaltung einer Komponente. Im EJB-Standard ist dabei keine Möglichkeit gegeben, eine Komponente über unterschiedliche Schnittstellen anzusprechen — stets werden alle Dienste in einer Schnittstelle zusammengefaßt und über das zugeordnete *Home*-Interface angeboten. Komponenten unter DCOM können dagegen mehrere Schnittstellen zugeordnet werden — ein Ausgleich für den Mangel an Mehrfachvererbung, wie er im Bereich von EJB möglich ist. Auch die CCM-Spezifikation ist an dieser Stelle sehr mächtig: Schnittstellen können in diesem Standard in mehrere *Ports* untergliedert werden, wobei die Vernetzung der Komponenten anhand dieser Ports erfolgt.

Im Hinblick auf das Modell aus Kapitel 3 entsprechen Ports aus dem CORBA Komponentenmodell der Vorstellung von Schnittstellen: Sie stellen die kleinste Einheit der Verschaltung dar und entlasten die Komponente von der Verantwortung für den Aufbau einer Kommunikationsverbindung. Fortgeschrittene Konzepte, wie im folgenden aufgeführt, werden durch das CCM im Augenblick nicht unterstützt:

- Ports können nicht hierarchisch zu neuen Ports komponiert werden. Dadurch wäre es möglich, Schnittstellen und Rollen als einheitliches Konzept in einem komponentenbasierten Anwendungssystem zu verwenden. Insbesondere könnte auf diese Weise die Verknüpfung von Rollen anstelle der Verbindung von Ports als Grundlage der Kommunikation zwischen Komponenten herangezogen werden.

- Die Menge der Ports einer Komponente ist nicht veränderlich über ihre Lebensdauer hinweg. Im Rahmen einer dynamischen Veränderung der Rollen einer Komponente zur Laufzeit ist es somit nicht möglich, neue Rollen hinzuzufügen, sofern diese auf Schnittstellen operieren, die nicht als Bestandteil der Komponente definiert wurden.

Neben einer Reihe von fachlichen Schnittstellen werden einer Komponente meist bestimmte Standardschnittstellen zugeordnet, die häufig für bestimmte technische Lösungen bestimmt sind (z.B. für container-managed persistence). Eine dieser Standardschnittstellen dient auch zur Festlegung der initialen Konfiguration einer Komponente. Der Mechanismus im CORBA Komponentenmodell sieht hierfür eine Reihe von Attributen vor, die vom Komponentenanwender beim Zusammenfügen der Anwendung geeignet belegt werden können. Die Definition der Attribute ist hierbei Teil der Schnittstellenspezifikation.

Schließlich ist erforderlich, interne Schnittstellen speziell zu unterscheiden, über die die Kommunikation mit der Kernkomponente stattfindet. Interne Schnittstellen müssen vor anderen Komponenten verborgen werden und es erscheint pragmatisch, die Verantwortung für den Verbindungsaufbau zur Kernkomponente der jeweiligen Fachkomponente zu übertragen. Auf diese Weise besteht keine Notwendigkeit, interne Schnittstellen nach außen zu kommunizieren.

4.3.5 Rollen

Rollen repräsentieren unterschiedliche Aspekte des Verhaltens einer Komponente. Wie in Kapitel 3 ausgeführt, wird das Konzept der Rolle im Rahmen der Realisierung auf einzelne Komponententypen abgebildet. Das hierbei zugrundegelegte Verständnis eines Komponententyps ist grundsätzlich verschieden von Komponententypen der untersuchten Komponententechnologien. Hierbei handelt es sich ausschließlich um syntaktische Schnittstelleninformationen, die keine Aussagen über das Verhalten einer Komponente treffen. Mit der im vorangegangenen Abschnitt eingeführten Einschränkung, daß die Schnittstelle einer Komponente über ihre Lebensdauer hinweg konstant bleibt, können Rollen als Annotationen einer Komponente realisiert werden, die Rückschlüsse auf das Verhalten zulassen. Die Standardschnittstelle einer Komponente bietet damit — ebenso wie hinsichtlich der regulären Schnittstellen — die Funktionalität zur Abfrage der unterschiedlichen Rollen, die eine Komponente spielt. Im Fall einer dynamischen Zuordnung von Rollen wird diese Schnittstelle zusätzlich für die Veränderung des Verhaltens ergänzt. Das Verhalten der Komponente hinsichtlich dieser besonderen Schnittstelle wird in Form einer „Standardrolle“ spezifiziert.

Rollenkomponenten werden in Form von Klassen spezifiziert und durch ihre zugeordnete Fachkomponente gekapselt. Die Implementierung einer Komponente benötigt einen Mechanismus, der die folgenden Funktionalitäten umfaßt:

- Aufbau der Organisation der einzelnen Rollenkomponenten entsprechend der durch das Zwiebelschalenmodell (vgl. Abschnitt 3.6.1) vorgegebenen Implementierungssicht.
- Delegation der Kommunikationsflüsse von den Ports der Komponente zu den jeweiligen Objekten und umgekehrt von den Objekten zu den entsprechenden Ports. Dies beinhaltet ebenso die Kommunikation zu der Kernkomponente über die spezifizierten internen Schnittstellen. In der Konsequenz müssen Nachrichten der einzelnen Objekte auf geeignete Weise zusammengeführt werden.
- Dynamische Erweiterbarkeit der Implementierung einer Fachkomponente um zusätzliche Rollen bzw. Entfernung vorhandener Rollen. Hierzu muß der Mechanismus in der Lage sein, über die Korrektheit des Vorhabens zu urteilen und entsprechende Unternehmungen bei Bedarf ablehnen zu können.

An dieser Stelle zeigt sich ein klarer Unterschied zu vergleichbaren Ansätzen wie z.B. in [Bäu98]: Rollenkomponenten treten nicht selbst als Instanz der Anwendung in Erscheinung, sondern werden stattdessen durch die entsprechende Fachkomponente gekapselt und teilen sich mit anderen Rollenkomponenten dieselbe „Identität“. Das Wissen über bestimmte Rollen einer Fachkomponente wird dadurch insbesondere nicht über den Mechanismus der Objekt- oder Komponentenreferenz kommuniziert. In der Arbeit von Bäumer ist diese dezidierte Auswahl von Rollen jedoch wichtig, um darüber das Verhalten einer Komponente steuern zu können. Entsprechend der Konzeption aus Kapitel 3 verhält sich eine Fachkomponente jedoch stets analog aller ihr zugewiesenen Rollen, wodurch sich ein einfacher, pragmatischer Umgang mit Fachkomponenten ergibt. Der Nachteil gerade hinsichtlich einer dynamischen Zuordnung von Rollen zu Komponenten zeigt sich darin, daß in der Kommunikation zwischen Komponenten nicht bekannt ist, welches Rollenverhalten eine Komponente von ihrem Partner erwartet. Aus diesem Grund kann nicht beurteilt werden, ob das Entfernen einer Rolle zu einem bestimmten Zeitpunkt erlaubt ist oder nicht.

4.4 Erweiterung des CORBA Komponentenmodells

In den letzten Abschnitten wurde das CORBA Komponentenmodell in vielerlei Hinsicht als sehr ausgereift präsentiert. In diesem Modell finden sich moderne Konzepte (wie beispielsweise Ports) sowie eine durchdachte Infrastruktur, die von den eingeflossenen Erfahrungen mit Enterprise Java Beans profitiert. Obwohl bisher noch kaum Implementierungen dieses Standards existieren, wird in diesem Abschnitt das CORBA Komponentenmodell als Grundlage für die Implementierung des formalen Modells aus Kapitel 3 herangezogen. Für einige wenige Konzepte zeigt sich das CCM nicht mächtig genug, weshalb in den folgenden Abschnitten geeignete Erweiterungen motiviert und diskutiert werden.

4.4.1 Komponententypen

Komponententypen des CCM werden in Form einer erweiterten IDL-Syntax aufgeschrieben und nach einer definierten Vorschrift auf das reguläre CORBA IDL-Format abgebildet. Auf diese Weise profitiert der Entwickler von einer verständlichen Notation, ohne daß bereits vorhandene und eingesetzte Werkzeuge zur Weiterverarbeitung der Schnittstellen-Informationen angepaßt werden müßten. An dieser Stelle ist es ebenfalls möglich, Erweiterungen hinsichtlich der Spezifikation bidirektionaler Ports, Rollen und spezieller Komponentenklassen einfach unterzubringen.

Komponenten sind entsprechend der Vorgaben des CCM aufgebaut aus einer Reihe von Ports, über die sie mit anderen Komponenten kommunizieren können. Zu der Menge der bestehenden Arten von Ports fügen wir sogenannte *Biports*, um die Spezifikation bidirektionaler Kanäle zu ermöglichen. Dadurch stehen dem Entwickler die folgenden Ports für die Deklaration eines Komponententyps zur Verfügung:

Facets, Receptacles definieren die Anknüpfpunkte, über die eine Komponente bestimmte Dienste zur Verfügung stellt bzw. in Anspruch nimmt. Diese Ports charakterisieren somit die Möglichkeit der Kommunikation zwischen zwei Komponenten: nur wenn Facet und Receptacle zusammenpassen, ist ein Zusammenspiel erlaubt.

Event sources, Event sinks spezifizieren, welche Events eine Komponente verschicken bzw. empfangen kann. Analog zu der synchronen Kommunikation über Facets und Receptacles werden über diesen Mechanismus asynchrone Kommunikationsverbindungen ermöglicht.

Attributes werden über standardisierte Methoden ausgelesen bzw. modifiziert und sind vor allem für die Konfiguration einer Komponente gedacht. Attributes stellen einen

Ansatz einer einheitlichen Konfigurationsschnittstelle dar und sind für die Verschaltung der Komponenten einer Anwendung nicht interessant.

Biports ergänzen die CCM-Spezifikation um das Konzept bidirektionaler Ports. Im Gegensatz zu Facets oder Receptacles basieren sie auf zwei Schnittstellen für je eine Richtung. Die Verschaltung der Biports zweier Komponenten ist dann möglich, wenn beide Komponenten abwechselnd eine der Schnittstellen realisieren und die andere fordern.

Das folgende Beispiel stellt einen Ausschnitt aus dem *Remote-Repository*-Framework aus Kapitel 3 dar. Dieses Framework dient dazu, Informationen in Form von Dokumenten oder auch Bildern dauerhaft zu speichern und — gekapselt in einzelnen Komponenten — für die weitere Verwendung verfügbar zu machen. Die Interaktion zwischen möglichen Nutzern dieser Information und den jeweiligen Komponenten erfolgt über einen Kanal, der in der einen Richtung die Möglichkeit zum Auslesen und Schreiben der Information bietet und in der anderen Richtung Nachrichten verteilt, sobald die gespeicherten Daten verändert wurden. Der Ablauf ist ähnlich zum bekannten *Observer-Pattern* aus [GHJV95]. Beteiligte Komponenten fassen diese Schnittstellen zu Biports zusammen und ermöglichen damit den Aufbau eines Kommunikationskanals (siehe dazu Abbildung 4.3 — die graphische Notation wird in Kapitel 5 vorgestellt).

Abbildung 4.3: Bidirektionaler Kanal und Biports

Die Deklaration der beiden Biports erfolgt in der erweiterten IDL-Notation, die hinsichtlich der bereits beschriebenen Konzepte ergänzt wurde. Der entsprechende Ausschnitt aus dem *Remote Repository*-Framework stellt sich wie folgt dar[2]:

```
module Remote_Repository {
  typedef sequence <octet> ByteStream;
  interface Content_Holder {
     void demandContent();
```

[2]Sämtliche Methoden sind so gestaltet, daß sie keinen Rückgabewert liefern. Details dazu finden sich in Abschnitt 4.5.1

```
      void setContent(in ByteStream bs);
   };
   interface Content_User {
      void ContentUpdated();
      void SendContent(in ByteStream bs);
   };

   role Container_Role {
      presents multiple Content_Holder, Content_User clients;
   }
   role Client_Role {
      presents Content_User, Content_Holder container;
   }
};
```

Über die Schnittstelle `Content_Holder` ist festgelegt, über welche Methoden andere Komponenten den gespeicherten Inhalt lesen oder auch verändern können. Demgegenüber beschreibt `Content_User` den Weg der Benachrichtigung bei einem veränderten Inhalt. Im Vergleich mit dem klassischen Observer-Pattern fällt auf, daß in diesen Schnittstellen die Funktionalität zum An- und Abmelden interessierter Komponenten bei der Datenkomponente fehlt. Diese Information ist nun anhand von Kanälen repräsentiert. Die Verwaltung der Kanäle muß dabei nicht zwangsläufig im Verantwortungsbereich der Datenkomponente liegen.

Die eigentliche Ergänzung der IDL zeigt sich bei der Deklaration von Rollen im obigen Listing. Eine Rollendeklaration referenziert hierbei die Schnittstellen, über die das Verhalten einer Rolle definiert ist. Das Schlüsselwort `role` leitet eine Rollendeklaration ein und das Schlüsselwort `presents` markiert die Biports, auf die eine Rolle aufsetzt. Im gezeigten Beispiel wird eine Rolle `Client_Role` deklariert, die die beiden Schnittstellen `Content_User` und `Content_Holder` zu einem Biport `container` zusammenfaßt. Eine Komponente, die die Rolle `Client_Role` spielt, muß die jeweils erstgenannte Schnittstelle eines Biports anbieten, während sie die nachfolgende Schnittstelle erwarten kann. So wie reguläre Ports für die Verknüpfung mehrerer Kanäle ausgelegt sein können, markiert das Schlüsselwort `multiple` am Biport `clients` die 1-zu-n Verbindung zwischen Containern und Clients.

Die Erweiterung der IDL um das Konzept der Rolle wird einfach auf die IDL aus dem CORBA Komponentenmodell abgebildet. Aus jeder Rollenspezifikation wird dabei ein

Komponententyp, aus den Biports je ein Receptacle und ein Facet. Für die Rolle `Client_Role` sähe der entsprechende Komponententyp dabei wie folgt aus:

```
component Client_Role {
  provides Content_User container_out;
  uses Content_Holder container_in;
}
```

Die Umsetzung der Rolle `Container_Role` erfordert allerdings die Fähigkeit, an eine Facet mehrere Verbindungen zu knüpfen. Dies war eine Anforderung des ursprünglichen *Multiple Interfaces RFP* der OMG im Jahr 1996 (vgl. [BVD01]), wird jedoch von der aktuellen CCM-Spezifikation nicht unterstützt. Für die Abbildung muß an dieser Stelle ein geeigneter Workaround gefunden werden, worauf wir im Rahmen dieser Arbeit verzichten.

Analog zu den Überlegungen aus Abschnitt 3.5.3 können zwei unterschiedliche Herangehensweisen für die Zuordnung von Rollen zu Fachkomponenten unterschieden werden. Da Rollen auf Komponententypen abgebildet werden, bietet sich einerseits der Vererbungsmechanismus des CCM für Komponententypen an, um den Typ einer Komponente von den Typen ihrer Rollen abzuleiten. Andererseits erfordert die gewünschte Dynamik in der Zuordnung von Rollen zu Komponenten ein flexibles Typsystem, daß die Veränderung eines Komponententyps zur Laufzeit gestattet. Gemäß den Anforderungen aus Abschnitt 4.3.5 kombinieren wir beide Varianten wie im folgenden beschrieben.

Schnittstellen eines Komponententyps werden statisch vom Entwickler festgelegt und über eine Reihe von Rollen referenziert. Da entsprechend der CCM-Spezifikation jeder Komponententyp von nur maximal einem weiteren Komponententyp abgeleitet sein darf, kann über den Mechanismus der Vererbung keine Zuordnung von Rollen zu Komponenten hergestellt werden (vgl. [CCM99]). In vorausgegangenen Überlegungen wurde jedoch ausgeführt, daß Rollen in der laufenden Anwendung nicht als dezidierte Komponententypen in Erscheinung treten — insbesondere eine Komponente nicht polymorph als Typ einer ihrer Rollen erscheinen muß. Aus diesem Grund können die referenzierten Rollendeklarationen einfach in den neuen Komponententyp überführt werden. Die Notation ist wie im folgenden Beispiel dargestellt:

```
component A::Remote_Repository plays Container_Role {};
component B::Remote_Repository plays Client_Role {};
```

Über diese Deklaration in erweiterter IDL-Syntax ist festgelegt, daß zwei Komponententypen `A` und `B` die Rollen `Container_Role` und `Client_Role` spielen.

Über die Verwendung dieser Rollen ist auf diese Weise die Menge der Schnittstellen (also der Ports) des neuen Komponententyps festgelegt. Allgemein erfolgt nach dem Schlüsselwort `plays` eine Aufzählung der unterschiedlichen Rollen, die eine Komponente spielt. Ein IDL-Präcompiler wandelt die Informationen aus den entsprechenden Komponententypen um in einen „flachen“ Komponententypen, der die Portdeklarationen in geeigneter Form zusammenführt. Dabei wird vorausgesetzt, daß verwendete Ports identisch sind, wenn sie denselben Bezeichner aufweisen.

Die Anbindung der Kernkomponente einer Fachkomponente erfolgt ebenfalls über geeignete Ports, die jedoch für andere Fachkomponente nicht sichbar sind. Diese interne Schnittstelle wird ebenfalls über eine oder mehrere Rollen spezifiziert, deren Bezeichner nach dem Schlüsselwort `requires` aufgezählt werden.

```
component C plays Role1 requires Role2 {};
```

Dieses Beispiel zeigt die Deklaration einer Komponente `Comp`, deren Kernkomponente über die Schnittstellen der Rolle `Role2` angebunden wird.

Rollen charakterisieren das Verhalten einer Komponente zu einem bestimmten Zeitpunkt. Prinzipiell bietet das CCM — ebenso wie die zugrundeliegende CORBA Infrastruktur [OH98] — Mechanismen an, um die Ports einer Komponente zur Laufzeit zu erfragen und zu verwenden (z.B. eine entsprechende Verbindung zu dieser Komponente aufzubauen). Jedoch ist diese Technik für die Kommunikation mit Komponenten gedacht, deren Typ zur Übersetzungszeit noch nicht bekannt ist. Sie würde sich damit für den Fall eignen, in dem Rollenkomponenten als eigenständiger „Zugang“ zum Verhalten einer Komponente auftreten und damit direkt referenzierbar und kommunizierbar sind (vgl. Abschnitt 4.3.5). Im Rahmen dieser Arbeit ist die Schnittstelle einer Komponente statisch, und wir können Aussagen über ihr Verhalten über eine Annotation der Komponente mit ihren aktuellen Rollen vermitteln.

Entsprechend dieser Konzeption gibt es für jede Komponente einen statischen Komponententyp entsprechend den Möglichkeiten, die das CCM vorgibt. Die (initiale) Menge der Rollen definiert einerseits die unterschiedlichen Ports der Komponente, sowie ihr anfängliches Verhalten. Zur Laufzeit kann eine Komponente hinsichtlich der gerade ausgeführten Rollen befragt werden bzw. eine Veränderung dieser Menge an Rollen vorgenommen werden. Diese Funktionalität ist Bestandteil der Standardrolle und wird im nachfolgenden Abschnitt diskutiert.

4.4.2 Standardrolle und Verbindungsaufbau

Entsprechend der Spezifikation des CORBA Komponentenmodells bietet jede Komponente eine Funktionalität an, die es anderen Komponenten ermöglicht, Informationen über

die Menge der deklarierten Ports zu erhalten und über diese zu navigieren (vgl. `Navigation`-Schnittstelle in [CCM99]). Zu diesem Zweck wurde eine besondere Schnittstelle eingeführt, das *equivalent-interface*. Ein Aufruf der Methode `get_component` liefert eine Implementierung dieser besonderen Schnittstelle zurück. Im Rahmen einer rollenbasierten Verhaltensspezifikation für Komponenten kann das zugehörige Verhalten in Form einer Standardrolle definiert werden. Um rollenspezifische Funktionalität mit aufzunehmen, wird die Schnittstelle `Navigation` wie folgt ergänzt:

```
module Components {
  interface Navigation {
    // ... bestehende Deklarationen
    valuetype Role {
      public RepositoryID roleID;
      public RoleName Name;
    }
    typedef sequence<Role> Roles;

    boolean playsRole(in RoleName Name);
    Roles getRoles();
    void addRole(in RoleDescription Role)
      raises (ErrorException);
    void removeRole(in RoleName Name)
      raises (ErrorException);
  }
}
```

Die neu hinzugefügten Methoden bieten die folgenden Funktionalitäten:

playsRole: stellt fest, ob sich die Komponente gerade entsprechend einer bestimmten Rolle verhält.

getRoles: liefert analog zu der Methode `describe_facets` für Facets eine Liste von Beschreibungen der aktuell von der Komponente übernommenen Rollen. Jede Beschreibung beinhaltet die Repository-Id des Komponententyps einer Rolle (also der jeweiligen Schnittstelle) und den Namen der Rolle. Namen werden hierbei relativ zum Namensraum des Typs der Komponente gesehen.

addRole: fügt eine neue Rolle dem Verhalten der Komponente hinzu. Die Rolle wird dabei über ihren Namen referenziert und muß der Komponente bereits bekannt sein.

Andernfalls — ebenso wie in dem Fall, daß die Rolle bereits von der Komponente gespielt wird — erzeugt die Komponente eine entsprechende Fehlermeldung.

removeRole: entfernt eine Rolle und paßt das Verhalten der Komponente entsprechend an. Falls der übergebene Name keine Rolle der Komponente bezeichnet, wird eine `ErrorException` ausgelöst.

Die dynamische Veränderung der Rollen einer Komponente birgt deutliche Risiken, da prinzipiell unbekannt ist, welches Rollenverhalten die jeweiligen Kommunikationspartner voraussetzen. Um nun zu verhindern, daß das Verhalten der Beteiligten einer Interaktion unerwünscht verändert wird, muß für die Dauer des Zusammenspiels das angenommene Rollenverhalten zugesichert werden. Die Lebensdauer eines Kanals umfaßt sicherlich die Dauer der darüber abgewickelten Interaktionen, und somit können wir fordern, daß nur dann Rollen entfernt werden dürfen, wenn an allen ihren spezifizierten Schnittstellen keine Kanäle mehr existieren. Auf diese Weise muß nur noch bei der Erzeugung eines Kanals die Menge der Rollen der Partnerkomponente überprüft werden. Umgekehrt ist diese Bedingung sicherlich oft zu restriktiv, da sie unter Umständen die Löschung von Rollen verhindert, die tatsächlich nicht mehr benötigt werden.

Der Aufbau bidirektionaler Verbindungen zwischen den Ports zweier Komponenten ist von den erwarteten Rollen unabhängig. Die Funktionalität, die für jedes Receptacle im equivalent interface einer Komponente generiert wird, umfaßt den Aufbau und Abbau von Kanälen. Auf der Implementierungsebene können die entsprechenden Methoden auf einfache Weise angepaßt werden, so daß der Aufbau eines Kanals einhergeht mit dem Aufbau eines Rückkanals, wodurch die bidirektionale Verbindung geschaffen wird. Ähnliches gilt für den Abbau von bidirektionalen Kanälen.

4.4.3 Abhängigkeiten

Im vorigen Abschnitt wurde die Spezifikation der Blackbox-Sicht von CORBA-Komponenten und ihren Rollen dargestellt. Abhängigkeiten zwischen Rollen, wie sie in Abschnitt 3.3.4 diskutiert wurden, beeinflussen ebenfalls die Menge der Ports einer Komponente. Für die diesbezüglich relevanten Arten von Abhängigkeiten führen wir nun entsprechende Notationen zur Deklaration einer Rolle ein:

Generalisierung: Für die Darstellung einer Generalisierungsbeziehung zwischen zwei Rollen wird die übliche Notation der Vererbung verwendet. Die somit spezifizierte Rolle übernimmt sämtliche Ports von der generelleren Rolle und kann diese um

weitere Ports ergänzen. Hierbei dürfen keine Namenskonflikte auftreten. Das folgende Beispiel zeigt die Deklaration einer Rolle `Author`, die mit einer Rolle `Person` in einer Generalisierungsbeziehung steht.

```
role Author : Person {}
```

Aggregation: Die Zusammenführung von Rollen in Form einer Aggregation kann auf der Implemetierungsebene ebenfalls durch eine Generalisierungsbeziehung ersetzt werden. Das folgende Beispiel zeigt die Abbildung der Aggregation aus den Rollen `Verwaltungsobjekt` und `Arbeit` zu einer Rolle `Einreichung`:

```
role Einreichung : Verwaltungsobjekt, Arbeit {}
```

Anforderung: Eine Anforderung zwischen zwei Rollen einer Komponente repräsentiert das Zusammenwirken der jeweiligen Funktionalitäten zur Laufzeit. Die Notation ist hierbei wie folgt durch das Schlüsselwort `relies_on` bestimmt:

```
role Einreichung relies\_on Container {}
```

Abhängigkeiten zwischen den Rollen einer Komponente werden hauptsächlich in den früheren Phasen der Entwicklung verwendet, um das Zusammenspiel der einzelnen Rollenkomponenten (in Form des Zwiebelschalenmodells) präzisieren zu können. Die Überführung in die Rollendeklaration ermöglicht einerseits Plausiblitätsprüfungen bei der Generierung eines Komponententyps, andererseits kann das Wissen bei der Veränderung der Menge aller Rollen einer Komponente verwendet werden, um Konsistenzüberprüfungen durchzuführen. Soll beispielsweise eine neue Rolle hinzugefügt werden, so muß überprüft werden, ob bereits die Rollen vorhanden sind, zu denen eine Abhängigkeitsbeziehung besteht.

In Abschnitt 3.5.7 wurde gezeigt, wie Abhängigkeiten zwischen Rollen auf das Zusammenspiel zwischen Rollenkomponenten abgebildet werden. Aus den oben genannten Abhängigkeiten resultieren auf diese Weise Kompositionsbeziehungen zwischen den einzelnen Rollenkomponenten. Solche Beziehungen können bereits in der fachlichen Modellierung auftreten und werden auch in der Spezifikation von Rollen in Form einer Kompositionsbeziehung festgehalten:

Komposition: Die Deklaration einer Rolle kann sich auf weitere Rollen einer Fachkomponente beziehen, die — im Gegensatz zur Anforderung — lediglich das interne Verhalten charakterisieren und für andere Fachkomponenten nicht sichtbar sind. Eine Kompositionsbeziehung ist repräsentiert durch das Schlüsselwort `requires`, wie in folgendem Beispiel gezeigt:

```
role Einreichung requires Time {}
```

4.4.4 Rollenkomponenten

Die Detaillierung des Rollenverhaltens einer Komponente erfolgt in Form von Greybox-Komponententypen. Um zu einem „leichteren“ Implementierungsmodell zu gelangen, werden Rollenkomponenten nicht als Komponenten des CCM realisiert, sondern als einfache Objekte, falls die Implementierungssprache objektorientiert ist. Diese Objekte sind durch ihre jeweilige Fachkomponente gekapselt und können nicht von anderen Fachkomponenten aufgerufen werden. Für die Deklaration des Typs einer Rollenkomponente (also des Greybox-Komponententyps — siehe Abschnitt 3.5.5) wird aus pragmatischen Gründen ebenfalls die IDL des CORBA-Standards verwendet.

Ein Greybox-Komponententyp spezifiziert die Kommunikation einer Rollenkomponente mit anderen Rollenkomponenten. Dabei wird die Interaktion über die regulären Schnittstellen von der Interaktion über die internen Schnittstellen unterschieden. Zusammen mit der Möglichkeit zum bidirektionalen Zusammenspiel kann somit der Typ einer Rollenkomponente basierend auf insgesamt vier IDL-Schnittstellen beschrieben werden. Für die Spezifikation der regulären Schnittstellen bietet es sich an, das Wissen aus der (Blackbox-)Rollendeklaration aus den vergangenen Abschnitten weitgehend wiederzuverwenden. Hierfür können auf einfache Weise aus der bestehenden Rollendeklaration (die in Form eines CCM-Komponententyps vorliegt) geeignete IDL-Deklarationen generiert werden.

Vereinfachend sei angenommen, daß eine Rollenkomponente keine identischen internen und regulären Schnittstellen besitzt. Dies ist akzeptabel, da Rollenkomponenten meist als Adaptoren konzipiert sind und damit im Regelfall eine Schnittstelle einer zugrundeliegenden Rollenkomponente modifizieren. Ausgehend von der Rollendeklaration können nun spezielle Schnittstellendeklarationen generiert werden, wobei jede Methode um den Namen des Biports ergänzt werden muß, in dessen Rahmen sie referenziert wird.

Für die Festlegung der Schnittstelle einer Rollenkomponente ist es erforderlich, die strukturierten Schnittstellen, wie sie aufgrund der Rollendeklaration vorgegeben sind, auf eine oder mehrere flache Schnittstellen abzubilden. Dazu werden die Bezeichner der jeweiligen Biports verwendet, um eindeutige Schnittstellen- und Methodennamen zu generieren. Im Fall der Rolle `Container_Role` führt dieses Schema zu zwei Schnittstellen, die der Kommunikation am Biport `clients` entsprechen.

```
module Remote_Repository {
  typedef sequence <octet> ByteStream;

  interface Clients_Content_Holder {
     void Clients_demandContent();
```

```
      void Clients_setContent(in ByteStream bs);
   }

   interface Clients_Content_User {
      void Clients_contentUpdated();
      void Clients_sendContent(in ByteStream bs);
   }
};
```

Zusätzlich zu diesen Schnittstellen, die die Blackbox-Sicht der Rolle repräsentieren, sind die internen Schnittstellen relevant, die die Kommunikation mit nachgeordneten Rollenkomponenten ermöglichen. Dazu sei angenommen, daß es in der Fachkomponente des Containers einen internen Zustand gibt, der aussagt, ob der Container noch leer ist oder bereits mit Inhalt belegt wurde. Die entsprechende Rolle trägt den Bezeichner `IsEmpty` und aus der zugehörigen Deklaration werden analog zu dem eben beschriebenen Beispiel die folgenden Schnittstellen generiert (der zugehörige Port sei ebenfalls mit `clients` bezeichnet):

```
module Remote_Repository {
   interface Clients_Flag_Holder {
      boolean Clients_isEmpty();
      void Clients_contentSet();
   }
   interface Clients_Flag_User {
      void Clients_Flag_Updated();
   }
}
```

Dieses Ergebnis ist ganz ähnlich zu den aus der Rollendeklaration `Container_Role` generierten Schnittstellen: der darüber realisierte Zustand kann abgefragt und verändert werden. Bei einer Veränderung wird eine geeignete Benachrichtigung versendet. Aufbauend auf diesen Schnittstellen wird nun ebenfalls das Gerüst für die Rollenkomponente der Rolle `Container_Role` erzeugt. Dazu wird standardmäßig ein Bezeichner aus dem Rollennamen und dem Wort `Component` generiert.

```
module Remote_Repository {
   interface Container_Role_Component :
           Clients_Content_Holder, Clients_Flag_User
   {
```

```
      void setRole_Clients_Content_User(Clients_Content_User rc);
      void setInternalRole_Clients_Flag_Holder(Clients_Flag_Holder rc);
  }
}
```

Prinzipiell wurde aus der Rollendeklaration der Rolle `Container_Role relies_on IsEmpty` die korrekte Zusammenstellung der Schnittstellen abgeleitet. Da jede generierte Schnittstelle mit einem Port korrespondiert, finden sich in der Deklaration einer Rollenkomponente die Schnittstellen aller Ports, die sie anbietet sowie die Schnittstellen aller Ports zugrundeliegender Rollen, die sie referenziert. Die Deklaration der Schnittstelle umfaßt weiterhin Methoden, die es später der Infrastruktur einer Fachkomponente ermöglichen, die einzelnen Rollenkomponenten miteinander zu verknüpfen. Um einer Rollenkomponente ihren Kommunikationspartner der höheren bzw. tieferen Schicht zuzuweisen, sind entsprechende Methoden definiert, deren Name aus `setRole` bzw. `setInternalRole` und dem jeweiligen Schnittstellennamen zusammengesetzt ist.

4.4.5 Mediatoren

Mediatoren eines Komponentenframeworks werden entsprechend den Erkenntnissen aus Kapitel 3 ebenfalls auf reguläre Komponenten der Implementierung abgebildet. Allerdings sind Mediatoren für den Nutzer eines Frameworks nicht sichtbar und können daher unter weitgehenden Freiheiten realisiert werden. Dabei ist zu berücksichtigen, daß eine Mediator-Komponente alle konkreten Ausprägungen einer Beziehung der fachlichen Modellierung repräsentiert. Aus diesem Grund sollten sämtliche Fachkomponenten, die relevante Rollen der Beziehung spielen können, dem Mediator bekannt sein. Für den Aufbau der Verbindung zwischen Mediator und Fachkomponente unterscheiden wir zwei Ansätze:

- Die CCM-Spezifikation sieht einen Standardmechanismus für die Konfiguration und Verbindung von CORBA Komponenten vor. Ohne Interaktion mit der jeweiligen Komponente kann damit bei der Instantiierung einer Fachkomponente die Verbindung zu ihren jeweiligen Mediatoren hergestellt werden. Durch dynamische Veränderungen einer Fachkomponente — beispielsweise durch das Hinzufügen einer neuen Rolle zur Laufzeit — ist diese Variante allein nicht ausreichend.
- Das benötigte Verhalten zur Anmeldung einer Fachkomponente bei relevanten Mediatoren kann auch als Bestandteil der Standardrolle spezifiziert und in einer entsprechenden Rollenkomponente für ein Framework realisiert werden.

4.4.6 Component Implementation Framework

Die Entwicklung von Komponenten erfordert stets die Realisierung ähnlich gearteter Funktionalitäten, beispielsweise im Hinblick auf das Persistenzmanagement oder die Handhabung des Lebenszyklus einer Komponente. Um Entwickler bei solchen Standardaufgaben zu entlasten, ist im Umfeld des CORBA Komponentenmodells das *Component Implementation Framework* (kurz: CIF) definiert, um die Realisierung dieser technischen Funktionalitäten zu automatisieren. Abbildung 4.4 zeigt, wie aus Komponenten- und Schnittstellenspezifikationen, die in CIDL bzw. regulärer IDL-Syntax vorliegen, vom CIF das Grundgerüst einer passenden Implementierung generiert wird.

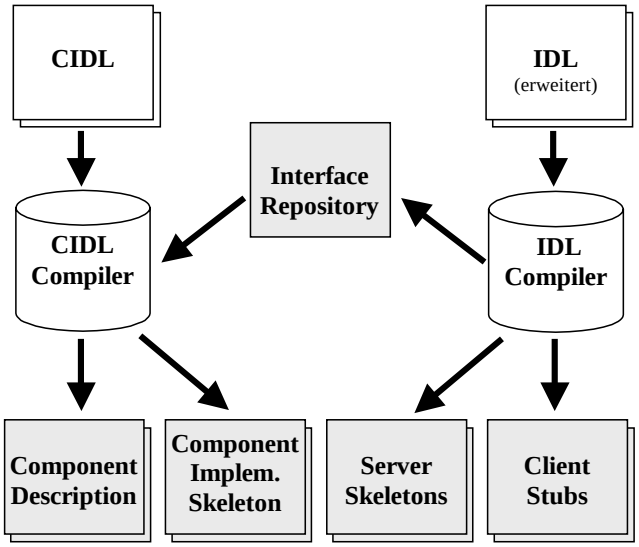


Abbildung 4.4: Artefakte einer Komponentenspezifikation (nach [WSO00])

Die bislang diskutierten Erweiterungen des CORBA Komponentenmodells sind als Ergänzung der CIDL-Syntax konzipiert und können von einem geeigneten Präcompiler in die reguläre CIDL-Syntax übertragen werden. Ihre Auswirkungen auf die Implementierung sind ebenfalls schematisch und können aus diesem Grund einfach vom CIF in das Gerüst der entstehenden Komponenten eingefügt werden. So zum Beispiel die Verwaltung der Rollen einer Komponente aber ebenso die Infrastruktur für das Zusammenspiel der Rollenkomponenten, die in den nachfolgenden Abschnitten vorgestellt wird.

4.5 Implementierung

Die vergangenen Abschnitte skizzieren die Eckpunkte einer Umsetzung der in Kapitel 3 eingeführten Konzepte in eine geeignete technische Infrastruktur basierend auf dem CORBA Komponentenmodell. Die Implementierung zweier besonders relevanter Aspekte wird im folgenden detailliert vorgestellt und diskutiert. Dies betrifft einerseits die Realisierung einer geeigneten Umgebung für die diversen Rollenkomponenten, die innerhalb einer Fachkomponente angesiedelt sind und andererseits die entsprechenden Kernkomponenten, die vom Anwender bereitgestellt werden.

4.5.1 Infrastruktur für Rollenkomponenten

Die einzelnen Rollenkomponenten fügen sich entsprechend dem Schema des Zwiebelschalenmodells zu einer Implementierung einer Fachkomponente zusammen. Um die Anordnung der Rollenkomponenten in den einzelnen Schichten zu organisieren und den Informationsfluß zu reglementieren ist eine passende Infrastruktur erforderlich. Konkret stellen wir die folgenden Anforderungen:

1. Die Infrastruktur muß in der Lage sein, zu einer Rolle (also ihrem Bezeichner, siehe Abschnitt 4.4.2) eine passende Rollenkomponente zu finden und zu instantiieren. Dabei ist zu beachten, daß zwischen Rollendeklaration und Rollenkomponente prinzipiell eine 1-zu-n Beziehung besteht. So kann während der Integration von Fachmodellen (vgl. Kapitel 3) festgelegt worden sein, daß zwei Rollen von einer Rollenkomponente adressiert werden. In diesem Fall würde das Hinzufügen von einer der beiden Rollen dazu führen, daß die bestehende Rollenkomponente durch eine neue Rollenkomponente ersetzt wird, die nun beide Rollen übernimmt.

2. Die Zuordnung der einzelnen Rollenkomponenten zu den einzelnen Schichten des Zwiebelschalenmodells erfolgt entsprechend den Richtlinien aus Abschnitt 3.6.2. Dabei ist die Zusammenführung der jeweiligen Delegationsketten nicht immer ohne zusätzliches Entwurfswissen möglich. Ein denkbarer Lösungsansatz könnte nun vorsehen, daß ein Entwickler für unterschiedliche Szenarien der Rollenkomposition die korrekte Abbildung auf die diversen Schichten hinterlegt. Kann die Zusammenführung von Rollen auch mit Hilfe dieses erweiterten Wissens nicht bewerkstelligt werden, so lehnt die Infrastruktur z.B. das Hinzufügen einer neuen Rolle ab.

3. Die Infrastruktur ist für die korrekte Verbindung der einzelnen Rollenkomponenten verantwortlich. In Abschnitt 3.6.1 wurden die Schnittstellen einer Rollenkomponente so konzipiert, als würden sie unmittelbar mit den Rollenkomponenten der benachbarten Schichten kommunizieren. Tatsächlich besteht an den einzelnen Schnittstellen zwischen den Schichten eine m-zu-n-Beziehung zwischen Rollenkomponenten. Dadurch ist es erforderlich, durch eine Zwischenschicht die korrekte Verbreitung der Nachrichten zu gewährleisten.

Die ersten beiden Anforderungen erfordern einen aufwendigen Mechanismus, der die Verwaltung der Rollenkomponenten und des Wissens über ihre korrekte Instantiierung ermöglicht. Denkbar ist hier beispielsweise ein werkzeuggestützter Ansatz, der es einem Entwickler erlaubt, Detailwissen über die Realisierung einzelnen Rollen zu spezifizieren und in eine geeigneten Datenbank zu überführen, auf die dann die Infrastruktur der Fachkomponente zur Laufzeit zurückgreift. An dieser Stelle sei ein solche Lösung vorausgesetzt, die insbesondere der Infrastruktur die Kenntnis der internen und externen Schnittstellen einer Rollenkomponente ermöglicht. Damit konzentrieren wir uns im folgenden auf die dritte Anforderung, die Etablierung einer Kommunikationsschicht zwischen den einzelnen Schichten des Zwiebelschalenmodells.

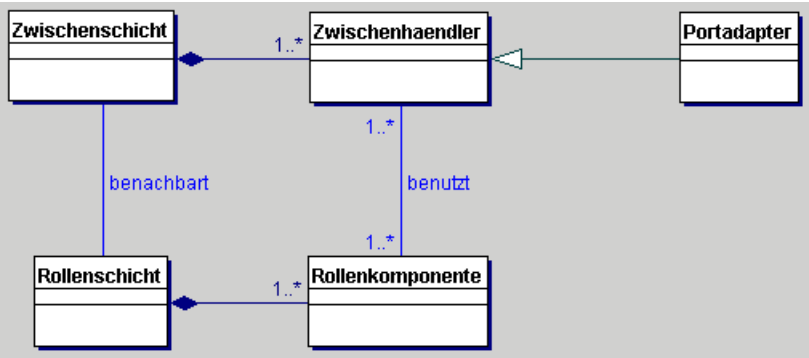


Abbildung 4.5: Klassendiagramm zur Infrastruktur

Die einzelnen Implementierungsklassen der Infrastruktur sind im Diagramm von Abbildung 4.5 dargestellt. Alle Rollenkomponenten derselben Schicht werden durch die Klasse `Rollenschicht` zusammengefaßt. Diese Klasse bietet beispielsweise Funktionalität, um die Schnittstellen der Schicht zu benachbarten Schichten zu erfragen oder um herauszufinden, welche Komponente mit welcher Schnittstelle verbunden ist. Rollenkomponenten kommunizieren nicht direkt miteinander, sondern über Objekte des Typs `Zwischenhändler`, die analog zum *Proxy-Pattern* aus [GHJV95] beide Kommunikationspartner voneinander entkoppeln. In einem Objekt der Klasse `Zwischenschicht` sind

alle Zwischenhändler zusammengefaßt, die zwei bestimmte Rollenschichten voneinander trennen. Auf diese Weise ergibt sich eine alternierende Schichtung von Rollen- und Zwischenschicht.

Verbindung von Rollenkomponenten

Die Verbindung der einzelnen Rollenkomponenten aus zwei benachbarten Rollenschichten erfolgt in zwei Schritten. Für jede der Schnittstellen wird erst ein eigener Zwischenhändler instantiiert, der diese Schnittstelle implementiert. Anschließend wird den entsprechenden Rollenkomponenten der unteren Schicht ein Verweis auf diesen Zwischenhändler und dem Zwischenhändler ein Verweis auf die Rollenkomponenten der oberen Schicht übergeben. Im Fall der Schnittstelle `Clients_Content_User`, die weiter oben vorgestellt wurde, weist der zugehörige Zwischenhändler die folgende Schnittstelle auf:

```
module Remote_Repository {
  interface Clients_Content_User_Proxy : Clients_Content_User {
      void setRole_Clients_Content_User(Clients_Content_User rc);
  }
}
```

Obwohl auch diese Schnittstelle einfach aus den vorhandenen Rollendeklarationen generiert werden kann, erscheint es aufgrund der einfachen Funktionalität unhandlich, für jeden Zwischenhändler eine entsprechende Implementierung bereitzustellen. Auch wenn pro Zwischenschicht nur ein Zwischenhändler existieren würde, der sämtliche Schnittstellen realisiert, würde sich diese Problematik verschärfen, da die Menge der zu realisierenden Zwischenhändler aufgrund der Kombinationsmöglichkeiten deutlich ansteigen würde. Sinnvoll wäre dagegen eine generische Implementierung eines Zwischenhändlers, die sich flexibel an die jeweils geforderten Schnittstellen anpassen könnte.

Die m-zu-n-Beziehung zwischen Rollenkomponenten benachbarter Schichten wird wie folgt durch den Zwischenhändler adressiert: m Rollenkomponenten der einen Schicht besitzen eine Referenz auf dieselbe Zwischenhändlerinstanz, und diese leitet die Kommunikation weiter an n Rollenkomponenten der anderen Schicht. Auf diese Weise vereinfacht sich die Implementierung einer Rollenkomponente, da die m-zu-n Beziehung für diese transparent ist.

Portadapter

Der Aufbau der schichtenorientierten Realisierung einer Fachkomponente durch die Infrastruktur führt zu einem Instanzengeflecht, wie es beispielhaft in Abbildung 4.6 skizziert ist. Für die Anbindung der Rollenkomponenten an die regulären und internen Schnittstellen, die in Form von CCM-Ports definiert sind, bietet sich die Einführung weiterer Zwischenschichten an. Auf diese Weise erfolgt die Anbindung transparent für beteiligte Rollenkomponenten und entspricht zudem den Vorgaben des CORBA Komponentenmodells.

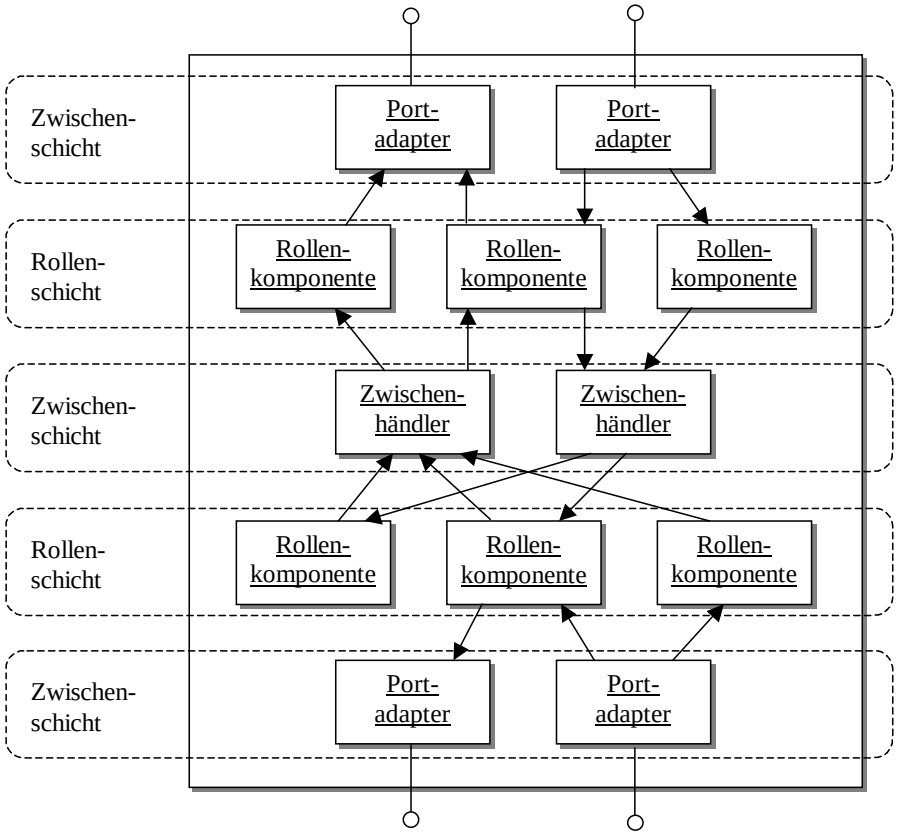


Abbildung 4.6: Zusammenspiel der unterschiedlichen Schichten

Die Zwischenhändler dieser speziellen Zwischenschichten bezeichnen wir als *Portadapter* (siehe Abbildung 4.5). Diese Instanzen agieren gegenüber ihren Rollenkomponenten — wie jeder Zwischenhändler — als weitere Rollenkomponente und hinsichtlich des

CORBA Implementierungsmodells als Implementierung eines Ports (vgl [CCM99]). Damit sind Portadapter die Zugangspunkte für andere Komponenten einer Anwendung und ihre (CORBA-)Referenzen können entsprechend ausgetauscht und verwendet werden.

Kommunikationsverhalten

Ein Ergebnis aus Kapitel 3 betrifft die Komposition einzelner Verhaltensspezifikationen, wie sie durch Rollen repräsentiert sind. Hierbei wurde vorausgesetzt, daß die resultierende Verhaltensspezifikation keine Konflikte an Schnittstellen enthält, die eine gemeinsame Basis mehrerer Rollen darstellen. Übertragen auf die Zusammenstellung von Rollenkomponenten bedeutet dies, daß sich Rollenkomponenten an gemeinsamen Schnittstellen identisch verhalten, also dieselben Nachrichten verschicken. Um solche identischen Abläufe oder auch Konflikte aufgrund fehlerhafter Rollenkomponenten zu erkennen, benötigen wir ein geeignetes Ablaufmodell für die Kommunikation innerhalb des Zwiebelschalenmodells.

Hierfür wurde in der formalen Modellierung ein getaktetes Kommunikationsverhalten zugrundegelegt, das es erlaubt, die Kommunikationsflüsse an den Schnittstellen der Rollenkomponenten zu vergleichen. Eine solche Taktung wird wie folgt auf das Ablaufmodell der Implementierung übertragen: Alle Rollenkomponenten einer Rollenschicht werden in einem Takt mit Nachrichten versorgt und können selbst Nachrichten versenden. Die gesendeten Nachrichten werden zusammengeführt, um im nächsten Takt wieder weitergeleitet zu werden. Daraus folgt, daß keine der gesendeten Nachrichten einer Rollenkomponente im selben Takt verarbeitet wird.

Auch das asynchrone Versenden von Nachrichten aus dem formalen Modell kann in der Implementierung einfach nachgebildet werden. Mit dieser wichtigen Eigenschaft des Ablaufmodells ist sichergestellt, daß Rollenkomponenten während eines Taktes nicht auf Antworten anderer Rollenkomponenten warten müssen. In der Umsetzung wird dementsprechend gefordert, daß in der Schnittstelle von Rollenkomponenten keine Methode Rückgabewerte liefert. Die Beispiele aus Abschnitt 4.4 sind entsprechend dieser Vorgabe gestaltet. Hier wurde der Abruf eines gespeicherten Inhalts in zwei Methoden aufgeteilt: eine für die Anforderung und eine für das Übermitteln der Daten.

Die letzte Eigenschaft des formalen Modells, die auch das Ablaufmodell der Implementierung bestimmt und vom Entwickler einer Rollenkomponente berücksichtigt werden muß, zeigt sich darin, daß den Methodenaufrufen innerhalb eines Taktes keine Reihenfolge zugesichert wird. Der Entwickler einer Rollenkomponente kann sich somit nicht darauf verlassen, daß die Nachrichten in derselben Reihenfolge bearbeitet werden, in der sie versendet wurden.

Mit diesen Festlegungen ergibt sich nun ein Ablaufmodell, das sich ausschließlich im Verhalten der Zwischenschicht und ihrer Bestandteile niederschlägt:

1. Jeder Zwischenhändler hat zwei Datenstrukturen, um Nachrichten zu speichern. Die erste enthält alle Nachrichten, die im aktuellen Takt zu versenden sind, während die zweite die Nachrichten für den darauffolgenden Takt enthält.

2. Eingehende Nachrichten werden von einem Zwischenhändler gespeichert. Da dieser in der vorliegenden Konzeption den Urheber der Nachricht (also die jeweilige Rollenkomponente) nicht kennt, kann gleiches Verhalten unterschiedlicher Rollenkomponenten prinzipiell nicht erkannt werden. Als einfache Lösung bietet es sich an dieser Stelle an, eine Nachricht nicht zu speichern, falls diese (in gleicher Form) bereits in der Liste vermerkt ist. Dieser Ansatz hat allerdings den Nachteil, daß eine Rollenkomponente nun eine Nachricht in einem Zeitintervall nicht mehrfach versenden darf.

3. Über eine spezielle Methode werden Zwischenhändler in einem ersten Durchlauf über den Beginn eines neuen Taktes informiert. Sie übernehmen nun die Liste der empfangenen Nachrichten für das Versenden und stellen eine leere Liste für neue Nachrichten bereit.

4. In einem zweiten Durchlauf werden alle Zwischenhändler dazu aufgefordert, gespeicherte Nachrichten an die jeweiligen Empfänger auszuliefern.

5. Die Reihenfolge der Benachrichtigung aller Zwischenhändler innerhalb der Durchläufe ist nicht bedeutsam. Es bietet sich jedoch an, über die Instanzen der Klasse `Zwischenschicht` stets alle Zwischenhändler einer Schicht aufzurufen und dann zur nächsten Schicht weiterzugehen. Dies ist über eine zyklische Verlinkung aller Zwischenschichten realisierbar.

Dieses Ablaufmodell erfordert einen eigenen Kontrollfluß (z.B. einen Thread) für jede Fachkomponente. Ist diese Anforderung durch die gewählte Komponentenlaufzeitumgebung nicht erfüllt, so sind weitere Mechanismen erforderlich, um den Kontrollfluß von den vorgelagerten Portadaptern geeignet umzulenken. Ein solcher Ansatz ist verhältnismäßig einfach zu realisieren und wird daher nicht weiter diskutiert.

4.5.2 Kernkomponenten

Kernkomponenten werden realisiert als reguläre Komponenten des CORBA Komponentenmodells. Die Schnittstellen und das Verhalten einer Kernkomponente ergeben sich aus

den diversen Rollen, die eine Kernkomponente spielen soll. Damit ist es möglich, eine Komponentendeklaration im Rahmen eines Komponentenframeworks mitzuliefern, die vom Anwender des Frameworks nur noch entsprechend ergänzt und realisiert werden muß. Ergänzungen betreffen hierbei Schnittstellen, die die Verbindung mit anderen Kernkomponenten ermöglichen. Über diese Kommunikationskanäle wird die Handhabung redundanter Repräsentationen ermöglicht, wie sie bei der Integration unterschiedlicher Abstraktionsniveaus entstehen.

Im Beispiel aus Kapitel 3 wurde dieser Sachverhalt anhand der Zusammenführung der Rollen `Gutachter` und `Autorengruppe` demonstriert. Da eine Autorengruppe unter Umständen mehrere Gutachter enthält, können beide Rollen nicht auf dieselbe Entität abgebildet werden. Dennoch ist ein enges Zusammenspiel zwischen den Rollen wichtig, um sicherzustellen, daß ein Gutachter nicht Arbeiten für ein Review zugeteilt bekommt, die er selbst (mit-)verfaßt hat. Sind diese Rollen derselben Fachkomponente zugeordnet, so findet der benötigte Abgleich über die internen Schnittstellen statt, wie in Abschnitt 3.5.8 dargestellt. Werden dagegen die Rollen durch unterschiedliche Fachkomponenten ausgeübt, so findet die benötigte Abstimmung über eine Kommunikation der jeweiligen Kernkomponenten statt. Damit also die Fachkomponente mit der Rolle `Gutachter` Arbeiten eines Autors hinsichtlich einer Begutachtung ablehnen kann, kommuniziert ihre Kernkomponente mit den assoziierten Kernkomponenten der Rolle `Autorengruppe`, um herauszufinden, bei welchen Arbeiten ein Autor mitgewirkt hat.

Über die Abstimmung von Kernkomponenten ist damit dem Anwender eines Frameworks die Möglichkeit gegeben, unterschiedliche Repräsentationen aus den einzelnen Teilmodellen konsistent zu halten. Ob in diesem Zusammenhang Informationen redundant gespeichert oder stets aus dem vorhandenen Wissen abgeleitet werden, bleibt im Ermessen des Entwicklers.

4.6 Werkzeugunterstützung

Die Entwicklung von Komponentenframeworks ist deutlich komplexer als die Entwicklung einzelner Komponenten oder ganzer komponentenbasierter Anwendungen. Besonders aufwendig gestaltet sich die Abstimmung der Querbeziehungen zu anderen Bestandteilen einer Anwendung und die Dekomposition des Verhaltens von Fachkomponenten in einzelne Rollenkomponenten. Ebenso kompliziert ist die Instantiierung und Konfiguration eines Komponentenframeworks. Angefangen vom einfachen Bereitstellen geforderter Kernkomponenten bis hin zum Austausch einzelner Rollenkomponenten bieten sich dem Anwender vielfältige Möglichkeiten, ein Komponentenframework auf seine Bedürfnisse hin anzupassen. Damit auch bei großen Anwendungssystemen diese Anpassung —

die im Zuge der Integration mehrerer Komponentenframeworks erfolgt — erfolgreich ist, ist eine umfassende, durchgängige Werkzeugunterstützung eine zwingende Voraussetzung für die erfolgreiche Softwareentwicklung mit Komponentenframeworks. Die folgende Auflistung skizziert die Anforderungen an eine geeignete Werkzeuglandschaft:

Fachmodellverwaltung: Das konzeptionelle Verständnis über die Entitäten und Beziehungen eines Komponentenframeworks findet sich im zugeordneten Fachmodell wieder. Für die Spezifikation, Organisation und Verwaltung von Fachmodellen bieten sich moderne CASE-Tools, wie beispielsweise Together [Tog] oder Rational Rose [Rat] an. Diese Werkzeuge bieten die Möglichkeit, Fachmodelle mit Hilfe graphischer Beschreibungstechniken der UML zu definieren, wobei auch entsprechende Erweiterungen (vgl. Kapitel 5) in akzeptabler Form berücksichtigt werden können. Zudem können die gespeicherten Modellinformationen exportiert werden und stehen für weitere Bearbeitungsschritte zur Verfügung.

Frameworkentwicklung: Fachmodelle werden im Verlauf der Entwicklung kontinuierlich weiter detailliert. Dabei entsteht Schritt für Schritt ein Implementierungsmodell mit Aussagen über beteiligte Komponenten und ihr Zusammenspiel. Auch hierbei können modellbasierte Werkzeuge, wie die gerade erwähnten, dem Entwickler Unterstützung anbieten, beispielsweise bei der Festlegung und Auswertung von Abläufen in Form von Interaktionsdiagrammen. Weiterhin bietet sich das CIF für die pragmatische Generierung von Schnittstellenbeschreibungen und für die Zusammenstellung von Softwarepaketen an. Letzteres stellt das Austauschformat für Komponentenframeworks dar und beinhaltet umfassende Informationen über die enthaltenen Komponenten sowie Vorgaben hinsichtlich einer korrekten Instantiierung. Und schließlich ist eine prototypische Ablaufumgebung sinnvoll, um Testabläufe mit dem Komponentenframework durchzuführen und unterschiedliche Konfigurationsszenarien zu erproben.

Modellintegration: Die Integration von Komponentenframeworks ist sicherlich der aufwendigste Vorgang bei der Erstellung einer Anwendung. Unterschiedliche Fachmodelle müssen hierbei konsistent zusammengeführt werden, was letztlich zu der Komposition von Verhaltensannahmen führt, wie sie durch Rollenspezifikationen gegeben sind. Ein geeignetes Werkzeug unterstützt den Entwickler bei der Zusammenführung, indem beispielsweise Abhängigkeitsgeflechte visualisiert und Konflikte erkannt werden. Dazu gehört insbesondere die Darstellung und Organisation der jeweiligen Rollenkomponenten entsprechend dem Zwiebelschalenmodell, wie es in Abschnitt 3.6.1 diskutiert wurde.

Komposition und Konfiguration: Aus der Modellintegration entstehen unterschiedliche Szenarien, wie bestimmte Rollen einer Fachkomponente zusammenwirken können. Die zugehörigen Rollenkomponenten müssen nun zusammengefügt oder zumindest in der Infrastruktur der Fachkomponente vorgemerkt werden, um sie bei Bedarf in den Verbund mit aufnehmen zu können. Dabei entstammen einige Rollenkomponenten den beteiligten Komponentenframeworks während andere im Zuge der Integration vom Anwender selbst realisiert wurden. Um eine korrekte Instantiierung aller Rollen- und Kernkomponenten zu erleichtern, ist ein Werkzeug nötig, das die Implementierungsstrukturen aufzeigt und die Vollständigkeit überprüft. Gerade auch hinsichtlich des Debuggings der Anwendung ist ein solches Werkzeug hilfreich, um zu jedem Zeitpunkt der Ausführung die gerade eingenommenen Rollen der beteiligten Fachkomponenten und das Verbindungsgeflecht aufzeigen zu können.

Projektunterstützung: Für die Entwicklung moderner Softwaresysteme sind derzeit eine Vielzahl nützlicher Werkzeuge verfügbar, die unterschiedliche Aspekte im Rahmen des Software-Engineerings adressieren: Konfigurationswerkzeuge übernehmen beispielsweise die Versionsverwaltung der Quelltexte, CASE-Tools erlauben Roundtrip-Engineering zwischen Quelltext und Anwendungsmodell und Kommunikationstools ermöglichen den einfachen Austausch zwischen den Entwicklern. Solche und ähnliche Werkzeuge sind auch in der Softwareentwicklung mit Komponentenframeworks sinnvoll. Dazu gehören auch bisher eher weniger genutzte Werkzeuge zur Prozeßunterstützung: mit ihrer Hilfe wird die Anpassung und Durchführung des gewählten Softwareentwicklungsprozesses vereinfacht (siehe Kapitel 6).

4.7 Zusammenfassung

In diesem Kapitel wurde gezeigt, wie die Konzeption des formalen Modells aus Kapitel 3 auf eine bestehende Komponententechnologie, namentlich das CORBA Komponentenmodell, abgebildet werden kann. Das CCM hat sich dabei durch eine Reihe innovativer und doch pragmatischer Konzepte als geeignete Implementierungsplattform erwiesen, obwohl bisher erst vereinzelt Implementierungen dieser OMG-Spezifikation zu finden sind (z.B. OpenCCM [OPE]).

Grundlage der Realisierung ist die Infrastruktur einer jeden Fachkomponente, die das Zusammenspiel der einzelnen Rollenkomponenten organisiert und ihre Aufteilung in die unterschiedlichen Schichten des Zwiebelschalenmodells entsprechend den Vorgaben des Entwicklers vornimmt. Hierbei zeigen sich isolierte Rollendeklarationen als nicht ausreichend präzise: prinzipiell muß für jede zu erwartende Kombination von Rollen eine

entsprechende Zusammenführung der zugehörigen Delegationsketten angegeben werden (vgl. Abschnitt 3.6.2). Dann ist auch eine Veränderung der Rollen einer Komponente zur Laufzeit einfach möglich.

Weiterhin wurde die Handhabung von Rollenverhalten an identischen Schnittstellen vereinfacht. Nach den strengen Vorgaben aus der formalen Modellierung müssen Rollenkomponenten an gemeinsamen Schnittstellen ein identisches Kommunikationsverhalten aufweisen. Der Zwischenhändler müsste sich nur für eine der Rollenkomponenten als Ursprung der Nachrichten entscheiden. Die hier gewählte Lösung ist deutlich pragmatischer und schreibt identisches Verhalten nicht zwingend vor. Voraussetzung ist allerdings ein korrektes Verhalten der Rollenkomponenten, denn Blackbox-Konflikte können auf diese Weise nicht zur Laufzeit erkannt werden.

Einige Detailfragen wurden bei der Konzeption der technischen Umsetzung außer acht gelassen. So wurde insbesondere die Vereinfachung des formalen Modells übernommen, daß sich strukturelle Veränderungen (z.B. das Hinzufügen oder Entfernen von Fachkomponenten, Rollen und Kanälen) als Kommunikationsverhalten niederschlagen und als solches nicht gesondert behandelt werden müssen. Der in Kapitel 3 vorgestellte Mechanismus über die Benachrichtigung der beteiligten Mediatoren erscheint jedoch einfach mit Hilfe der Konzepte des CCM realisierbar.

Letztlich ist es gelungen, mit wenigen syntaktischen Erweiterungen an der Spezifikationssprache und standardisierten Realisierungen für Fachkomponenten eine einfache Umsetzung der Konzepte des formalen Modells zu erzielen, die sich zudem nahtlos in das CORBA Komponentenmodell einfügt. Dabei ist jedoch zu erwarten, daß die vorgestellte Infrastruktur gerade hinsichtlich der anvisierten modernen und komplexen Anwendungssysteme nur eine geringe Leistung zeigt. Durch das umfassende Objektgeflecht, das für die Realisierung einer Fachkomponente eingesetzt wird, steigt der Kommunikationsbedarf und sinkt der Durchsatz. Es ist jedoch zu erwarten, daß an vielen Stellen die konzpierte Flexibilität nicht unbedingt benötigt wird und die Infrastruktur an diesen Stellen zu Gunsten einer verbesserten Leistungsfähigkeit vereinfacht werden kann.

Kapitel 5

Frameworkdokumentation mit der UML

Obwohl in den letzten Jahren eine ganze Reihe teils kommerzieller, objektorientierter Frameworks entwickelt und angeboten wurden, war die erfolgreiche Wiederverwendung stets von einer unverhältnismäßig langen Lernkurve begleitet [FSJ99]. Die Idee, über ein Framework anwendungsspezifisches Know-How zu kapseln und für „jedermann“ anwendbar zu gestalten, scheitert überwiegend an mangelhafter Dokumentation [Mat00, FHLS97]. Dafür lassen sich hauptsächlich zwei Ursachen ausmachen und zwar bezüglich der Fragen *was* dokumentiert wird und *wie* Informationen festgehalten werden.

Einerseits erscheint es absolut notwendig, daß die Dokumentation eines Frameworks speziell auf die unterschiedlichen Typen von Entwicklern zugeschnitten wird, die mit dem Framework über seine Lebensdauer in Kontakt kommen. Die Fülle der enthaltenen Information muß daher auf ein Minimum beschränkt werden und gleichzeitig hinreichend umfassend sein, um dem Entwickler das jeweils nötigte Verständnis zu vermitteln [RWL96]. Beispielsweise benötigt ein Entwickler, der mehrere Komponentenframeworks zu einem konsistenten Ganzen zusammenfügen möchte, ein besseres Verständnis der inneren Zusammenhänge als jemand, der ein Framework lediglich isoliert einsetzen möchte. Die einzelnen Rollen des Framework-Entwicklungsprozesses werden ausführlich in Kapitel 6 vorgestellt und diskutiert. An dieser Stelle wird festgehalten, daß es gerade für die Dokumentation von Komponentenframeworks besonders wichtig erscheint, die Informationsmenge anhand des jeweiligen Bedarfs zu organisieren und somit das *information-hiding*

von der Realisierung in die Dokumentation zu übertragen. Angelehnt an die Ausführungen von Johnson [Joh92] umfaßt die Dokumentation eines Komponentenframeworks die folgenden Informationen:

Zweck: Welche Ziele werden mit dem Framework verfolgt? Welche Probleme adressiert? Zudem eine Beschreibung der Anwendungsdomäne, die bei Komponentenframeworks durch das entsprechende Fachmodell repräsentiert ist.

Verwendung: Eine Beschreibung der Anpassung und Konfiguration des Frameworks. Damit insbesondere die Vorstellung des Frameworkentwicklers, auf welche Weise das Framework verwendet werden soll.

Design: Dies umfaßt die Spezifikation von Verhalten und Struktur der einzelnen Bestandteile des Frameworks.

Die andere Ursache einer mangelhaften Dokumentation von Frameworks findet sich in den teils sehr unterschiedlichen Dokumentationstechniken, die bisher in diesem Bereich eingesetzt wurden und noch werden. An dieser Stelle fehlt es insbesondere an Richtlinien, wie diese Varianten erfolgreich zu einem Gesamtbild eines Frameworks und seiner Einsatzmöglichkeiten kombiniert werden können. Bewährt haben sich vor allem die folgenden Techniken (siehe [FSJ99]):

Beispiele: Einer der einfachsten Wege, die Handhabung eines Frameworks zu vermitteln, besteht in der Angabe und Diskussion geeigneter Beispielanwendungen. Wie in Handbüchern gängiger Programmiersprachen finden sich auch bei Frameworks wie z.B. Java Swing eine Fülle an Beispielen und unterschiedlichen Benutzungsszenarien. Obwohl sich hiermit ein schneller Lernerfolg einstellt, besteht ein häufiges Problem für Entwickler in der adäquaten Umsetzung des erworbenen Wissens vom einfachen Umfeld der Beispielanwendung auf die tatsächliche Problemstellung, welche meist deutlich komplexer ist.

Cookbooks: Unter einem *Cookbook* wird eine Sammlung von *Rezepten* verstanden, die — ähnlich zu der Idee hinter Patterns — für eine ganz bestimmte Aufgabe einen konkreten Lösungsvorschlag bereithalten und dessen Durchführung beschreiben. Das Cookbook bietet darüber hinaus eine sinnvolle Organisation einer Menge von Rezepten mit passenden Querbezügen. Das Kochbuch zu dem Framework *HotDraw* beispielsweise hat sich über die Jahre sowohl für den Einstieg als auch als praktischer Leitfaden bewährt [Joh92].

Contracts: Im Allgemeinen versteht man *Contracts* als gegenseitige, verbindliche Zusagen hinsichtlich der Übernahme bestimmter Verantwortlichkeiten während einer

Kommunikation über eine Schnittstelle [Mey92]. Im Umfeld mehrerer kollaborierender Objekte oder Komponenten stellen Contracts bestimmte Anforderungen an die einzelnen Teilnehmer der Kommunikation [HHG90]. In dieser Hinsicht sind Contracts eng verwandt mit dem Konzept der Rolle, wie es in dieser Arbeit diskutiert wird: Eine Kooperation mehrerer Komponenten kann nur stattfinden, wenn jede dieser Komponenten eine bestimmte Verantwortung übernimmt. Im Gegensatz zu Rollen können jedoch mit der Hilfe von Contracts bestimmte Invarianten eines Zusammenspiels übergreifend spezifiziert werden und müssen nicht auf isolierte Eigenschaften einzelner Komponenten aufgeteilt werden. Aufgrund dieser Charakteristik erscheinen Contracts für die Dokumentation von Frameworks vorteilhaft.

Design Patterns: Ein *Pattern* ist ein isolierter, bewährter Lösungsansatz für eine konkrete Problemstellung. In Form von *Design Patterns* hat sich dieses Konzept in den letzten Jahren vor allem für Fragestellungen des Feindesigns von Anwendungen durchgesetzt (siehe [GHJV95]). Doch auch für Fragestellungen geeigneter Architekturformen wurde eine Reihe von *Architectural Patterns* entwickelt [BMR+96], die auch bei der Entwicklung der Beispielanwendung in Abschnitt 3.5.8 eingesetzt wurden. Da die Architektur eines Frameworks dokumentiert und kommuniziert werden muß, sind Patterns ein geeignetes Hilfsmittel.

Hooks: In [FHLS97] werden sogenannte „Hooks“ als Beschreibungsform für konkrete Zielsetzungen bei der Anpassung eines Frameworks vorgestellt. Die verwendete, strukturierte Darstellung von Anforderung, Typ, beteiligten Komponenten und unterstützenden Kommentaren ist mit dem musterbasierten Ansatz von Johnson [Joh92] vergleichbar.

Die Fülle an Dokumentationsvarianten, die im Umfeld von Frameworks eingesetzt werden, deutet auf die Bemühungen hin, den Lernprozeß beim Anwender zu beschleunigen und die Anwendung zu vereinfachen. Nicht zuletzt lassen sich hierbei auch methodische Mängel ausmachen: Gerade bei der Entwicklung von Frameworks wird nicht frühzeitig genug mit der Dokumentation begonnen, so daß das Wissen über implizite Designentscheidungen während der Entwicklung oft verloren geht. Im Umfeld dieser Arbeit ist eine qualitativ hochwertige Dokumentation eines Komponentenframeworks besonders entscheidend, da ein Komponentenframework (im Gegensatz zu einer Komponente) weitreichende Annahmen über die Struktur und die Elemente eines Anwendungssystems trifft. Nur wenn diese Annahmen über die Umgebung eines Frameworks richtig verstanden werden, kann eine korrekte Instantiierung des Frameworks vorgenommen werden.

In diesem Kapitel wird eine graphische Beschreibungstechnik entworfen, die eine solide Grundlage für die Dokumentation von Komponentenframeworks darstellt. Die unterschiedlichen Beschreibungstechniken stellen eine Erweiterung der *Unified Modeling*

Language dar, die sich im Verlauf der letzten Jahre zum De-Facto-Standard für die objektorientierte Modellierung herauskristallisiert hat. Die optimale Auswahl und Kombination dieser Techniken für die präzise und zielgerichtete Dokumentation eines Frameworks liegt jedoch letztendlich beim Entwickler.

5.1 Grundzüge der UML

Die Unified Modeling Language wurde von Grady Booch, Ivar Jacobson und James Rumbaugh als Standardnotation für objektorientierte Analyse und Design vorgeschlagen (vgl. [RJB98]). Die UML führt eine Reihe bekannter und bewährter Techniken anderer Methoden zusammen und erweitert sie geeignet. So finden sich Konzepte aus OOA/OOD, OMT und OOSE. Die UML selbst ist jedoch keine Methode, sondern eine Ansammlung von Beschreibungstechniken für statische und dynamische Charakteristika eines Anwendungssystems. Die folgende Übersicht beschreibt knapp eine Klassifizierung der unterschiedlichen Diagrammarten (siehe [BRS97]):

Strukturdiagramme modellieren die Bestandteile eines Systems und können Aussagen über deren Funktionalität enthalten. Strukturdiagramme existieren in zwei Varianten: *Klassendiagramme* zeigen die Klassen eines Programms, ihre Attribute und Operationen und die Beziehungen und Abhängigkeiten untereinander. *Objektdiagramme* zeigen das Instanzengeflecht der unterschiedlichen Objekte einer Anwendung.

Use Case-Diagramme modellieren die Benutzer eines Systems und ihre Interaktionen mit dem System auf einer sehr hohen Abstraktionsebene. Sie dienen zur groben Strukturierung detaillierterer Beschreibungen der Systemfunktionalität, wie sie beispielsweise durch Sequenzdiagramme repräsentiert werden.

Sequenzdiagramme sind auch als *Message Sequence Charts* bekannt und zeigen beispielhaft die Kommunikation zwischen den Beteiligten eines Systems. Die UML-Variante dieser Beschreibungsform ist angereichert mit Konstrukten sowohl zur Erzeugung und Verwerfung von Objekten als auch für die Unterscheidung zwischen synchroner und asynchroner Kommunikation.

Kollaborationsdiagramme sind eine spezielle Form von Objektdiagrammen mit zusätzlicher Information über das Kommuniaktionsverhalten zwischen den Objekten. Kollaborationsdiagramme legen einen Schwerpunkt auf die Beziehungen zwischen den Objekten, während Sequenzdiagrammme die zeitliche Abfolge betonen.

Zustandsdiagramme modellieren den Datenzustand einer Anwendung und die Veränderungen im Lebenszyklus der Objekte einer bestimmten Klasse. Der Zustand eines Objekts besteht aus den aktuellen Werten seiner Attribute sowie Referenzen zu anderen Objekten und unter Umständen deren Zustände.

Aktivitätsdiagramme sind eine besondere Form von Zustandsdiagrammen für die Modellierung des Kontroll- und Datenflusses. Auf unterschiedlichen Abstraktionsebenen können sie insbesondere verwendet werden, um Geschäftsprozesse zu modellieren.

Implementierungsdiagramme existieren in zwei Ausprägungen: *Komponentendiagramme* zeigen die Struktur des Quelltextes und seine Unterteilung in einzelne Komponenten. *Deploymentdiagramme* zeigen die Verteilung von Objekten zur Laufzeit auf physikalische Ausführungsorte zur Laufzeit.

5.2 Erweiterungsmechanismus

Wohl mit ein ausschlaggebender Grund für die Durchsetzung der UML als „Standard“ in der großen Menge unterschiedlicher Notationen besteht in der Vorgabe konkreter Möglichkeiten zur Erweiterung. So existieren heute eine Vielzahl von UML-Erweiterungen für alle möglichen Anwendungsgebiete. Als Beispiel sei eine Ergänzung der UML genannt für die Spezifkation von Systemen basierend auf mobilen Agenten [KRSW01]. Die folgenden Mechanismen sind für die Erweiterung der UML vorgesehen:

Zusicherungen (engl. *constraints*) repräsentieren bestimmte Bedingungen, die ein Modell weiter einschränken. Sie können beispielsweise über die *Object Constraint Language* spezifiziert werden und sind dann integraler Bestandteil der Modellierung.

Merkmale (engl. *tags*) sind Annotationen der Form *Schlüsselwort = Wert* und detaillieren bestimmte Charakteristika einzelner Modellelemente.

Stereotypen bieten die Möglichkeit, das Metamodell der UML geeignet zu verfeinern. Bezeichner für Stereotypen werden in spitze Doppelklammern gesetzt. Alternativ dürfen Stereotypen auch durch eigene graphische Repräsentationen dargestellt werden (z.B. ein Icon).

Seit der UML-Spezifikation in der Version 1.3 können Erweiterungen in Form sogenannter *Profile* strukturiert und organisiert werden. Dadurch wird es unter anderem möglich, Erweiterungen des Standards zu modularisieren und zueinander in Beziehung zu setzen

[DSB99]. Konkret werden in diesem Kapitel die folgenden drei Profile entwickelt (siehe Abbildung 5.1):

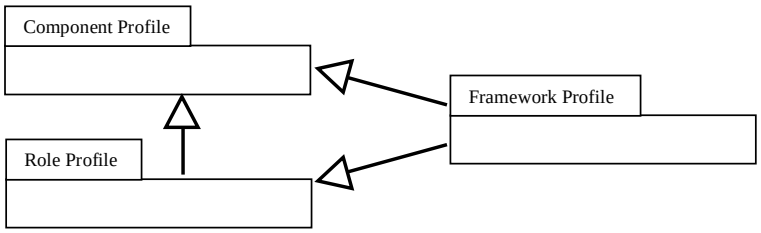


Abbildung 5.1: Erweiterungen der UML

Component Profile: Obwohl grundlegende Konzepte wie Schnittstelle und Komponente bereits einen bedeutenden Bestandteil der UML darstellen, fehlen wichtige Notationen für die Beschreibung komponentenbasierter Anwendungen. Mit diesem Profil präzisieren wir die Möglichkeit der UML, hierarchische Kompositionenen von Komponenten darzustellen.

Role Profile: Der Begriff der Rolle tritt bereits in unterschiedlicher Form in der Spezifikation der UML auf, beispielsweise im Zusammenhang von Assoziationen zwischen Klassen [UML00][1]. Ein klares Verständnis des Rollenkonzepts für Komponenten wird in dieser Erweiterung die Möglichkeit bieten, isolierte Verhaltensbeschreibungen zu spezifizieren, zueinander in Beziehung zu setzen und schließlich zu kombinieren. Dafür ist es einerseits erforderlich, bestehende Diagramme (insbesondere Interaktionsdiagramme) für die „Intra-Objekt“-Kommunikation zu erweitern. Andererseits führen wir im Hinblick auf eine spezifikationsnahe Implementierung Notationen für Rollen- und Kernkomponenten ein.

Framework Profile: Die Dokumentation von Komponentenframeworks erfordert Möglichkeiten um festzulegen, wie das Framework verwendet werden soll und erweitert werden kann. Dazu gehört unter anderem ein starker Bezug zur Implementierung: welche Komponenten sind vorgegeben, welche müssen vom Anwender bereitgestellt werden?

Letztlich hängt die prägnante Dokumentation von Komponentenframeworks in vielen Bereichen von der Existenz aussagekräftiger Beschreibungstechniken ab. Dies betrifft beispielsweise die Repräsentation von Fachmodellen, die Vermittlung des Verständnisses über die zugrundeliegende Softwarearchitektur und Aussagen über dynamische Veränderungen an Struktur und Verhalten des Komponentengeflechts. In vielen dieser Bereiche

[1] Die hier verwendete Spezifikation der UML ist zum Zeitpunkt der Fertigstellung dieser Arbeit noch immer im Beta-Stadium. Es ist zu erwarten, daß einzelne Details noch verändert werden.

sind aktuelle Ansätze mit der Verbesserung und Erweiterung der durch die UML gegebenen, graphischen Beschreibungstechniken beschäftigt. An dieser Stelle konzentrieren wir uns daher auf essentielle Erweiterungen, die hauptsächlich die in dieser Arbeit diskutierten Konzepte wiederspiegeln. Dabei verzichten wir auf eine detaillierte Abbildung der Erweiterungen in die Datenstrukturen, die sich im Semantik-Kapitel der UML-Spezifikation finden [UML00]. Eine solche Fundierung schafft zwar ein klares Verhältnis der Erweiterungen zu bestehenden Konzepten der UML, ist jedoch oft sehr umständlich und kann zudem verhältnismäßig einfach aus den Beschreibungen der folgenden Abschnitte hergestellt werden.

5.3 Component Profile

Grundlegende Konzepte für die Spezifikation komponentenbasierter Anwendungen sind bereits in der UML enthalten: Komponenten können, basierend auf einer Reihe von Schnittstellen, definiert und das Zusammenspiel zwischen Komponenten mit Hilfe einfacher Abhängigkeiten festgehalten werden. Obwohl bereits dieser Ansatz bei einfachen Systemen ausreichend erscheint, zeigen sich gerade bei der Spezifikation moderner, großer Anwendungen diese Möglichkeiten als zu beschränkt:

- Komponentendiagramme stellen nur Komponententypen und ihre generellen Abhängigkeiten untereinander dar. Sie beinhalten insbesondere keine Aussagen über die Art der korrekten Instantiierung und über die Verbindungen zwischen den Komponenten zur Laufzeit. Es fehlt ebenfalls das Konzept von Kanälen, um Kommunikationsverbindungen zwischen Komponenten repräsentieren zu können.
- Im Umfeld von Kanälen bekommen Schnittstellen eine zusätzliche Bedeutung als Zugangspunkt zu bestimmter Funktionalität einer Komponente. Dabei macht es Sinn, Zugangspunkte und die angebotene Funktionalität und damit Schnittstelle und Schnittstellentyp konzeptionell zu unterscheiden (vgl. *Ports* im CORBA Komponentenmodell [CCM99]). Diese Unterscheidung ist in den Notationen der UML nicht vorgesehen. Obwohl in Abschnitt 3.5.8 ebenfalls davon ausgegangen wurde, daß eine Komponente maximal eine Schnittstelle eines Typs aufweist, erscheint die genannte Unterscheidung wichtig genug, um eine entsprechende Erweiterung der Komponentendiagramme zu diskutieren[2].
- Entsprechend dem Verständnis der UML sind Strukturen statischer Natur. Gerade in der Entwicklung komponentenbasierter Anwendungen sind jedoch dynamische

[2] Um die Unterscheidung deutlich zu machen, sprechen wir im weiteren auch von Ports als Instanz eines Schnittstellentyps

> Veränderungen im Komponentengeflecht ein wichtiger Bestandteil, um Systeme flexibel und anpaßbar zu gestalten.

Prinzipiell zeigt sich an diesen Problemstellungen die Tatsache, daß die UML nicht vorgesehen war als Beschreibungssprache für Softwarearchitekturen im allgemeinen und komponentenbasierte Anwendungssysteme im speziellen. Demgegenüber bieten herkömmliche *Architekturbeschreibungssprachen* [SG96], wie beispielsweise *Rapide* [LKA+95] oder *UniCon* [SDK+95], Möglichkeiten zur deutlich präziseren Formulierung — in vielen Fällen sogar mit entsprechender Werkzeugunterstützung. Letztlich erfahren diese Sprachen — wohl auch aufgrund der einhergehenden Formalismen — nur wenig Verbreitung im industriellen Umfeld. Hier liegt jedoch inzwischen die Stärke der UML, die als Standard akzeptiert und in vielen Bereichen erfolgreich eingesetzt wird. Aus diesem Grund existieren eine Reihe von Bemühungen, die UML hinsichtlich der Spezifikation von Softwarearchitekturen zu erweitern (vgl. [HNS99]). Die fehlende Präzision wird oft durch halbformale Zusicherungen (beispielsweise in Form von OCL-Constraints — siehe z.B. Catalysis [DW98]) ausgeglichen.

Die Grundelemente der Spezifikation komponentenbasierter Systeme wurden in Kapitel 3 eingeführt und werden nun in die UML übertragen: die Struktur einer Anwendung ergibt sich aus einer Reihe von Komponenten, die über Schnittstellen bzw. Ports die Kommunikation mit anderen Komponenten ermöglichen. Im Gegensatz zu vielen vergleichbaren Ansätzen (nicht zuletzt das CORBA Komponentenmodell) unterscheiden wir nicht in Ports für „erbrachte“ und „genutzte“ Dienste, sondern behandeln alle Ports einer Komponente als gleichgestellte Anknüpfungspunkte. Die eigentliche Verbindung von Komponenten erfolgt schließlich über Kanäle zwischen ihren jeweiligen Ports. Kanäle sind dabei weniger mächtig als das bekannte Konzept des *Konnektors* aus den Architekturbeschreibungstechniken [DMT99] — sie sind nur bilateral und verknüpfen intuitiv stets dieselben Schnittstellentypen miteinander.

Eine einfache Möglichkeit, Aussagen über Struktur und Verhalten eines komponentenbasierten Systems zu treffen, besteht in der Darstellung der beteiligten Komponenten zu einem festgelegten Zeitpunkt. Mehrere solcher *snapshot*s des Systems sind insbesondere hilfreich, um dynamische Veränderungen zu verdeutlichen, wie zum Beispiel das Hinzufügen von Komponenten oder den Aufbau eines neuen Kanals. Da im Rahmen dieser Arbeit der Begriff „Komponente“ im Sinne von „Instanz eines oder mehrerer Komponententypen“ verwendet wird, nennen wir diese Diagramme *Komponentendiagramme*[3].

In Abbildung 5.2 ist eine Komponente c (zu einem bestimmten Zeitpunkt) dargestellt, die aus zwei weiteren Komponenten $cc1$ und $cc2$ aufgebaut ist. Komponenten werden —

[3] Im Gegensatz zum UML-Standard, der unter diesem Begriff die Darstellung von Typen versteht.

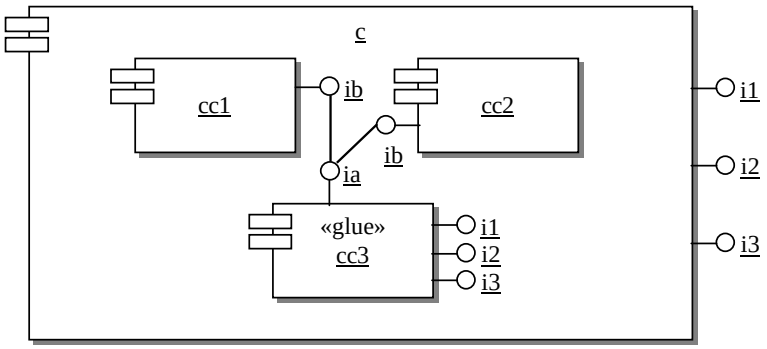


Abbildung 5.2: Ein Komponentendiagramm

analog zur UML-Notation — mit Rechtecken dargestellt, die zwei kleinere Rechtecke am linken Rand aufweisen (alternativ kann der Stereotyp «`Component`» verwendet werden). Schnittstellen sind in diesem Diagramm in der üblichen „Lollipop“-Variante repräsentiert (vgl. [UML00]). Neu im Vergleich zum Standard sind die folgenden Details des Diagramms (vgl. Abschnitt 3.4.3):

- Der innere Aufbau einer Komponente zeigt stets eine Gluekomponente, die mit dem Stereotyp «$glue$» markiert ist. Diese Komponente weist zumindest die gleichen Schnittstellen wie die Vaterkomponente auf. In Abbildung 5.2 hat die Gluekomponente ebenso wie ihre Vaterkomponente Schnittstellen vom Typ $I1$, $I2$ und $I3$.

- Schnittstellen treten als Instanzen eines bestimmten Schnittstellentyps auf und können bei Bedarf mit eigenen Bezeichnern versehen werden.

- Zwischen den Schnittstellen der Komponenten können nach bestimmten Regeln Kanäle geschaltet sein. Diese sind stets ungerichtet, durch eine einfache Linie repräsentiert und stellen eine Voraussetzung für die Kommunikation zwischen den jeweiligen Komponenten dar. Verbindungen dürfen nur zwischen Komponenten desselben Vaters vorgenommen werden. So wäre ein Kanal zwischen der Komponente $cc1$ und einer Komponente außerhalb nicht erlaubt.

Analog dazu werden Komponententypen in *Komponententypdiagrammen* spezifiziert. Ein solches Diagramm definiert den möglichen Aufbau einer oder mehrerer Komponenten aus Subkomponenten sowie deren Verschaltung. Die notationelle Unterscheidung von Instanz und Typ erfolgt — wie in der UML vorgesehen — durch ein Unterstreichen des Instanzenbezeichners. Ein möglicher Komponententyp für die Komponente c ist in Abbildung 5.3 abgebildet.

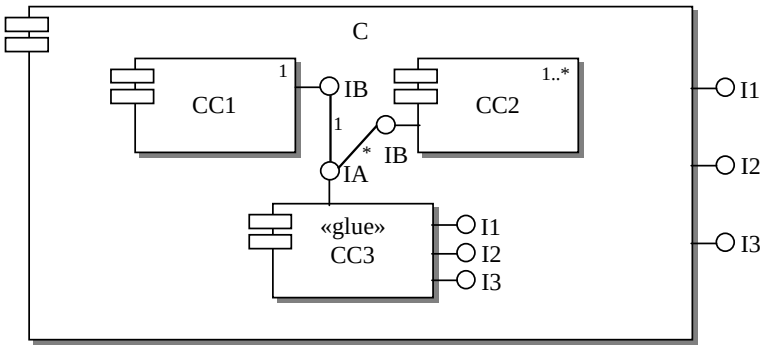


Abbildung 5.3: Ein Komponententypdiagramm

Über die Darstellung des Komponententypen ist sowohl die Menge der erlaubten Schnittstellen festgelegt als auch die Möglichkeiten des strukturellen Aufbaus. Hierfür ist jeder Subkomponente eine Kardinalität zugeordnet, die die Menge der Komponenten dieses Typs einschränkt. Während von jeder Gluekomponente stets genau eine Instanz existieren muß, können in jeder Komponente des Typs C eine oder mehrere Komponenten des Typs $CC2$ vorkommen. Auf ganz ähnliche Weise ist die Menge der Kanäle spezifiziert. Jeder eingezeichnete Kanal zwischen den Schnittstellen zweier Komponente trägt eine Kardinalität, die angibt, ob der Kanal existieren muß („`1`“) oder lediglich optional ist („`*`“). Für die Kombination von Komponententypen stellen wir fest, daß das Fehlen eines Kanals zwischen zwei Schnittstellen die Bedeutung trägt, daß der Aufbau solcher Kanäle nicht erlaubt ist. Und schließlich können auch Kardinalitäten für Schnittstellentypen angegeben werden, wovon in Abbildung 5.3 kein Gebrauch gemacht wurde.

Letztlich sind die Möglichkeiten des Komponententypdiagramms, den Aufbau einer Komponente zu spezifizieren, limitiert und müssen bei Bedarf um detailliertere Zusicherungen erweitert werden. So sind inbesondere die dynamischen Veränderungen am strukturellen Aufbau der Komponente nicht leicht darstellbar. Für diesen Zweck bietet sich die Verwendung mehrerer Instanzdiagramme an, die in einer Reihe von Snapshots die strukturellen Veränderungen darstellen (siehe auch [BRS+00]).

Für die Spezifikation des Verhaltens einer Komponente können weitere Diagrammtypen der UML eingesetzt werden. Wir benötigen hierbei nur kleinere Ergänzungen: Einerseits muß es für eine Komponente möglich sein, neue Kanäle aufzubauen, bestehende zu entfernen und eine Veränderung in der Menge ihrer Subkomponenten herbeizuführen. Andererseits muß die Kommunikation der Komponente so gekennzeichnet werden, daß die jeweils adressierte Schnittstelle erkenntlich ist.

Der Mechanismus für Auf- und Abbau von Kanälen richtet sich sehr stark nach der jeweiligen Implementierung, also der zugrundeliegenden Komponentenplattform. Im CORBA-Komponentenmodell (vgl. Kapitel 4) sind beispielsweise für jeden einzelnen Port einer Komponente Methoden vorgesehen, die es anderen Komponenten erlauben, eine Verbindung auf- bzw. abzubauen. Um eine implementierungsunabhängige Beschreibung dieser Vorgänge zu erzielen, sei die Funktionalität für das Verbindungsmanagement in der Standardschnittstelle einer jeden Komponente deklariert (vgl. Abschnitt 4.4.2). Über diese Schnittstelle signalisiert eine Komponente einen Wunsch zur Veränderung des Verbindungsgeflechts und — falls es sich um eine Gluekomponente handelt, die repräsentativ für die Vaterkomponente agiert — zum Hinzufügen oder Entfernen einer Subkomponente. Diese Schnittstelle wird auch verwendet, um die Komponente von der Existenz neuer Kanälen zu informieren, die auf ihre eigene oder die Initiative einer anderen Komponente zurückgehen (das betrifft ebenfalls weitere strukturelle Veränderungen).

Für die Unterscheidung der einzelnen Schnittstellen im Kommunikationsverhalten einer Komponente führen wir eine Annotation der Nachrichten mit dem jeweiligen Schnittstellenbezeichner ein. Die Schreibweise `s1!foo()` bezeichnet eine Nachricht `foo()`, die über die Schnittstelle `s1` versendet wird. Für die Annotation einer eingehenden Nachricht wird die folgende Schreibweise eingeführt: `s1?bar()`. Die Notation entstammt der FOCUS-Methodik [BDD+92] und bewährt sich insbesondere bei Zustandsdiagrammen, bei denen sich — im Gegensatz zu Interaktionsdiagrammen — die Richtung einer Nachricht nicht aus der graphischen Darstellung ergibt.

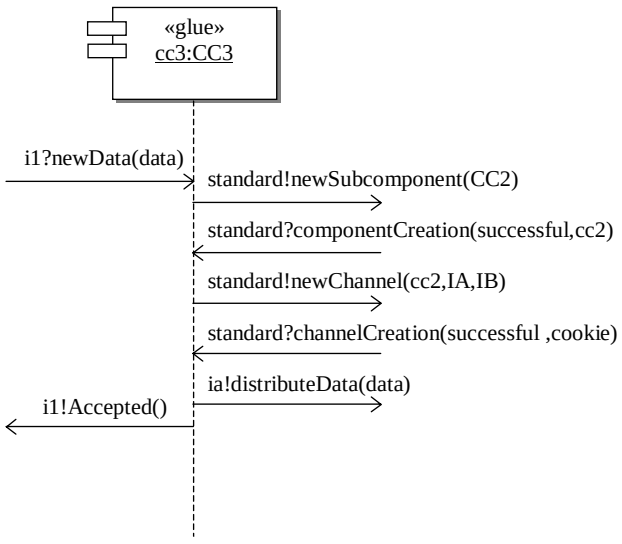


Abbildung 5.4: Sequenzdiagramm für die Erzeugung einer neuen Komponente

Ein Sequenzdiagramm mit dieser erweiterten Notation findet sich in Abbildung 5.4. Da Komponenten über den Mechanismus der Kanäle voneinander entkoppelt sind, ist aus einem solchen Sequenzdiagramm nicht immer der eigentliche Kommunikationspartner ersichtlich. In diesem Beispiel ist ein willkürlicher Ablauf der Gluekomponente aus Abbildung 5.2 dargestellt, in dessen Verlauf sie über ihre Standardschnittstelle die Erzeugung einer neuen Subkomponente sowie den Aufbau eines neuen Kanals zu dieser Komponente veranlaßt. Obwohl sie im Zuge der Instantiierung der Subkomponente einen direkten Verweis auf diese Komponente erhält und diesen auch für den neuen Kanal benötigt, ist keine ausschließliche Kommunikation mit der Subkomponente gegeben: entsprechend der Typdeklaration (vgl. Abbildung 5.3) sind weitere Komponenten über Kanäle an der Schnittstelle des Typs `IA` verbunden.

Das auf diese Weise entstehende Component Profile befähigt einen Entwickler, Struktur und Verhalten einer komponentenbasierten Anwendung unter Verwendung der in Kapitel 3 eingeführten Basiskonzepte zu spezifizieren. Die Verwendung von Gluekomponenten, die den Repräsentanten einer Vaterkomponente darstellen, führt dabei zu einem klaren Verständnis der internen Zusammenhänge.

5.4 Role Profile

Das Konzept der Rolle ist bereits seit den ersten Versionen der UML bekannt im Kontext von Assoziationen und in der Beschreibung von Designmustern [FS97]. Die Handhabung von Rollen ist in diesem Zusammenhang jedoch uneinheitlich: einerseits kann jede Repräsentation einer Entität im Modell als Rolle aufgefaßt werden und andererseits ergeben sich aufgrund der Assoziationen zu anderen Entitäten weitere Rollen, die nur im Kontext der Assoziation Gültigkeit besitzen und deren Zusammenhang mit anderen Rollen der Entität nicht immer nachvollziehbar festgelegt ist. In den unterschiedlichen Wirkungsbereichen der OMG wurde daher in den letzten Jahren viel über das Konzept der Rolle diskutiert (vgl. [Cum00]) und viele der Ergebnisse sind in die aktuelle Version der UML eingeflossen, die nun ein präziseres und umfangreicheres Verständnis von Rollen und ihrer Verwendung in der Modellierung aufweist.

Rollen zeigen sich in der UML hauptsächlich bei der Darstellung von *Kollaborationen*. Eine Kollaboration abstrahiert von den tatsächlichen Gegebenheiten einer Anwendung, indem das Zusammenspiel zwischen Entitäten basierend auf *ClassifierRole*s festgelegt wird. Die Interaktion erfolgt entsprechend festgelegter Beziehungen, die ebenfalls über Rollen, sogenannte *AssociationRole*s, definiert sind. Bei der „Instantiierung“ einer Kollaboration werden konkrete Entitäten und Assoziationen benannt, die sich entsprechend der festgelegten Rollen verhalten und deren Zusammenspiel den Vorgaben der Kollaboration

entspricht. Rollen in der UML sind damit prinzipiell ein Typkonzept, auch wenn sich die Spezifikation in dieser Hinsicht nicht festlegt (vgl. [UML00]):

> *„Roles (in collaborations) are somewhat between types and instances. Like instances, they identify distinct occurences of a single classifier. Like types, they describe a reusable element that can have many distinct instances.“*

So werden Rollen auch nicht als Alternative zu dem Konzept „Typ“ verwendet, sondern in enger Verzahnung mit diesem. Jeder Rolle in der UML ist ein bestimmter Typ zugeordnet und jedes Objekt ist nach wie vor von einem Typ, kann sich aber entsprechend der Spezifikation mehrerer Rollen verhalten. Notationell werden die jeweiligen Rollennamen — mit einem Schrägstrich getrennt — hinter den Instanzbezeichner geschrieben. So weist `p1/lead:Point` auf ein Objekt `p1` der Klasse `Point` hin, das sich entsprechend der Rolle `lead` verhält. Der Rolle `lead` ist hierbei ebenfalls der Typ `Point` zugeordnet.

Der Schwerpunkt des Konzepts Rolle in der UML liegt auf der Spezifikation von Kollaborationen zwischen mehreren Instanzen des Systems. Hierbei darf eine Instanz mehrere Rollen aus derselben oder auch aus unterschiedlichen Kollaborationen spielen, wodurch bereits eine gute Grundlage für Rollen im Rahmen komponentenbasierter Anwendungen existiert. Hinsichtlich der Repräsentation von Rollen in Fachmodellen und in der Ebene der Realisierung durch Komponentenframeworks zeigen sich jedoch die folgenden Mängel:

- Rollen können im Rahmen einer Assoziation in Beziehung zueinander gesetzt werden. Dies betrifft jedoch lediglich Rollen unterschiedlicher Entitäten und eignet sich nicht zur Spezifikation des Zusammenspiels unterschiedlicher Rollen derselben Instanz. Für die Darstellung von Fachmodellen ist daher eine notationelle Erweiterung nötig, die insbesondere die Zusammenführung von Rollen ermöglicht. Auch für die dynamische Veränderung der Zuordnung von Rollen und Entitäten sind geeignete Darstellungsformen erforderlich.

- Für die Darstellung interner Abläufe einer Komponente bieten sich Zustandsdiagramme an. Durch die Kapselung einzelner Verhaltensaspekte einer Komponente in Rollen können jedoch auch Interaktionsdiagramme (und speziell Sequenzdiagramme) verwendet werden, um das Zusammenwirken der einzelnen Rollen einfacher zu visualisieren. Dies erfordert eine Erweiterung dieser Diagrammart, um neben der „Inter-Objekt-“ auch die „Intra-Objekt-Interaktion“ repräsentieren zu können.

- Schließlich gilt es, die Konzepte der Realisierung einer Fachkomponente durch eine Reihe von Rollenkomponenten zu repräsentieren und deren Zusammenspiel zu charakterisieren.

Für die Darstellung von Rollen und ihren Abhängigkeiten und Beziehungen untereinander führen wir *Rollendiagramme* ein, die insbesondere für die Repräsentation von Fachmodellen genutzt werden. Die graphische Darstellung von Rollen erfolgt wahlweise über die Annotation mit dem Stereotyp «`Role`» oder mit Hilfe von Rechtecken mit abgerundeten Ecken, wie sie von Riehle vorgeschlagen werden [Rie00]. Diese Rechtecke sind ähnlich zu der Darstellung von Komponententypen mit Schnittstellen versehen und können einander überlagern, um die gemeinsame Nutzung eines Schnittstellentyps zu markieren.

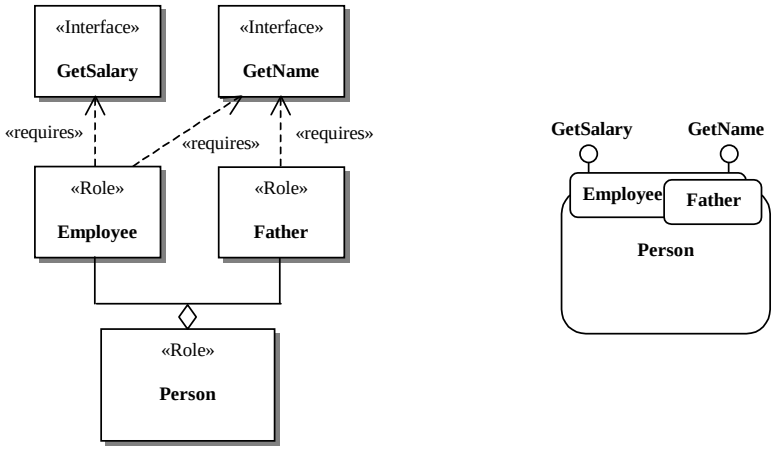


Abbildung 5.5: Zwei Varianten der Darstellung von Rollen

Abbildung 5.5 zeigt zwei äquivalente Darstellungen der Rolle `Person`, die aus den beiden Rollen `Employee` und `Father` aggregiert ist und auf den Schnittstellentypen `GetSalary` und `GetName` aufbaut. Die Abhängigkeit einer Rolle von ihren Schnittstellen ist auf der linken Seite der Abbildung über eine *Dependency*-Beziehung mit dem Stereotyp «`requires`» repräsentiert, die durch einen gestrichelten Pfeil dargestellt wird. Die Darstellung auf der rechten Seite ist als Überlagerung von Rollen gestaltet und drückt die Zusammenhänge einfacher und intuitiver aus. In vielen Fällen sind die Zusammenhänge jedoch zu komplex, um auf diese Weise repräsentiert zu werden. Insbesondere im Fall von Generalisierungsbeziehungen zwischen Rollen erscheint die ausführlichere Notation empfehlenswerter. In diesem Fall können auch abstrakte Rollen repräsentiert und durch den Stereotyp «`abstract`» markiert werden. Während Beziehungen zwischen Entitäten als reguläre Assoziationen zwischen ihren Rollen repräsentiert werden (wofür das Konzept der Assoziation aus der UML ausreichend ist), werden die (internen) Abhängigkeiten durch die Notationen aus Abbildung 5.6 dargestellt. Entsprechende Beispiele für Rollendiagramme finden sich in der Diskussion über Fachmodelle in Abschnitt 3.3.5 sowie in weiteren Abschnitten aus Kapitel 3.

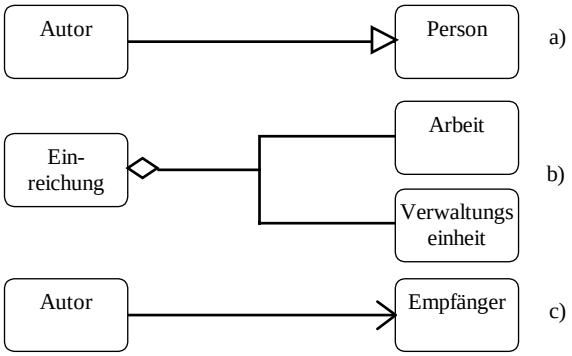


Abbildung 5.6: Abhängigkeiten zwischen Rollen

Einzelne oder mehrere Rollen aus einem Rollendiagramm werden durch ein Komponententypdiagramm detailliert. Hierdurch wird vor allem der Schritt von einer reinen Blackbox-Betrachtungsweise zu einer Greybox-Sicht vollzogen. Im Komponententypdiagramm treten daher weitere Schnittstellen auf, die durch den Stereotyp «`internal`» als komponentenintern markiert sind. Mit den Erweiterungen des Component Profile kann zudem nun ein konkretes Rollenverhalten mit dem Komponententyp verknüpft werden. Dazu wird — wie bereits zuvor beschrieben — die Veränderung struktureller Merkmale durch eine zeitliche Reihung von Komponentendiagrammen dargestellt. Im Zusammenhang mit Rollen betrifft das einerseits die Verschaltung von Komponenten durch Kanäle, aber zudem auch die Veränderung der Menge an Rollen, die eine bestimmte Komponente spielt. Für die Darstellung der Rollen einer Komponente im Komponentendiagramm bietet sich einerseits die UML-konforme Notation an, wenn zugelassen wird, daß auch mehrere Rollen aufgelistet werden dürfen. Andererseits können die Rechtecke der Rollen ebenso wie im Rollendiagramm auch an die Schnittstellen der Komponente angelagert werden, um die Zugehörigkeit auszudrücken. Die Abbildung von Rollen auf Komponententypen ist hierbei unterschiedlich zu der Sichtweise, wie sie sich aus [UML00] ergibt: in der UML stellen Rollen einen Klassifizierungsmechanismus neben der Typsicht auf Objekte dar.

Weiterhin werden nun die Interaktionsdiagramme der UML so erweitert, daß es einem Entwickler möglich ist, das interne Zusammenspiel der einzelnen Rollen einer Komponente darzustellen. Hierbei werden lediglich Sequenzdiagramme betrachtet, da diese für den gewünschten Zweck besonders geeignet sind. Im Vergleich zu Zustandsdiagrammen, die üblicherweise für die Spezifikation des internen Verhaltens eines Objekts (oder auch einer Komponente) herangezogen werden, lassen sich im Sequenzdiagramm die einzelnen — durch Rollen repräsentierten — Verhaltensaspekte leichter trennen und darstellen.

Hierfür erlauben wir die Aufteilung der *lifeline* einer Komponente in weitere Linien entsprechend der Menge der jeweiligen Rollen der Komponente.

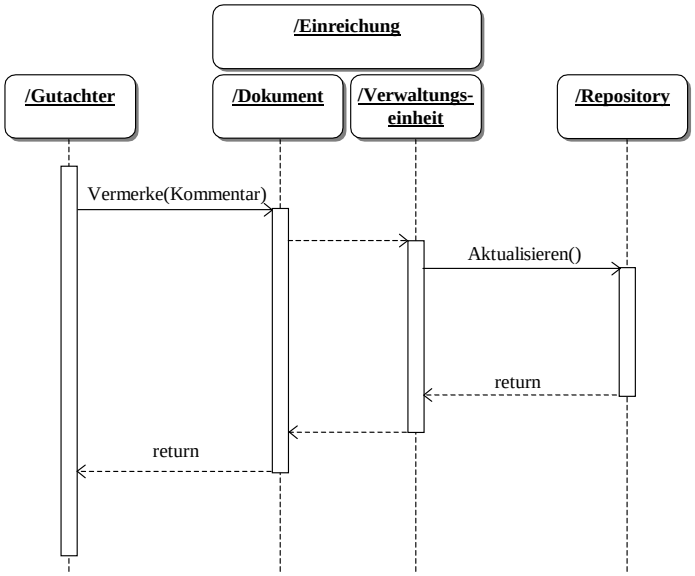


Abbildung 5.7: Interaktionsdiagramm mit Rollen

Aus dem in Kapitel 3 diskutierten Beispiel ergab sich für die Rolle einer `Einreichung` eine Aggregation aus den Rollen `Dokument` und `Verwaltungseinheit`. Im Sequenzdiagramm aus Abbildung 5.7 ist nun ein einfacher Ablauf dargestellt, der sowohl die „externe“ Kommunikation einer Entität der Rolle `Einreichung` als auch das interne Zusammenspiel der einzelnen Rollen enthält (vereinfachend wurde an dieser Stelle auf die weiter oben eingeführte Notation für die Unterscheidung von Schnittstellen verzichtet). In der Darstellung wird das Rechteck der aggregierenden Rolle oberhalb der Rechtecke der aggregierten Rollen gezeichnet, um auszudrücken, daß beide Rollen von derselben Instanz gespielt werden.

Eingehende Nachrichten an einer Schnittstelle, die von mehreren Rollen referenziert wird, werden durch einen Pfeil dargestellt, der an jeder der entsprechenden Lifelines einen Kopf trägt. Alternativ ist es auch erlaubt, den Pfeil nur an der Rolle enden zu lassen, die auf die entsprechende Nachricht reagiert. Die Verwendung von Aktivitätsbalken (wie in Abbildung 5.7) ist sinnvoll, da sich die Aktivität der Instanz aus dieser Information ableiten läßt.

Eine interessante Möglichkeit, die sich aus dieser Erweiterung ergibt, betrifft das Zusammenspiel der Rollen einer Komponente: bereits auf einer verhältnismäßig abstrakten Ebene können die einzelnen Verhaltensaspekte zueinander in Beziehung gesetzt werden.

In Abbildung 5.7 charakterisiert ein gestrichelter Pfeil einen Rollenwechsel, der eine Abhängigkeit der Rollen untereinander ausdrückt. In dieser Blackbox-Sicht sind solche Informationen noch wenig aussagekräftig und müssen gegebenenfalls durch entsprechende Kommentare präzisiert werden. Im Verlauf der Verfeinerung stehen dem Entwickler jedoch Greybox-Pendants für die einzelnen Rollen zur Verfügung, und nun kann über die internen Schnittstellen mit dem gewohnten Mechanismus die Interaktion zwischen den einzelnen Rollen im Detail spezifiziert werden. Hierbei ist darauf zu achten, daß — entsprechend den Vorgaben des Zwiebelschalenmodells — eine bestimmte Verschaltung zwischen den internen Schnittstellen besteht. Im Beispiel aus obiger Abbildung dürfen die Rollen `Dokument` und `Verwaltungseinheit` nicht direkt miteinander kommunizieren. Statt dessen ist es erforderlich, eine gemeinsam referenzierte Rolle einzuführen, über die der Austausch abgewickelt wird. Beispiele für diese erweiterten Sequenzdiagramme finden sich in diversen Abschnitten aus Kapitel 3.

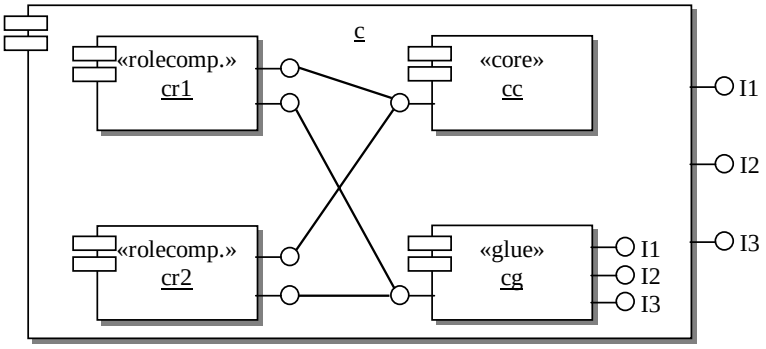


Abbildung 5.8: Aufbau einer Komponente mit Rollenkomponenten

Mit der Einführung von Greybox-Komponententypen besteht bereits ein detailliertes Verständnis über die Realisierung einzelner Rollen, das das Verhalten der zugehörigen Rollenkomponente bestimmt. Für die Komposition einzelner Rollenkomponenten bieten sich reguläre Komponentendiagramme aus Abschnitt 5.3 an, wie in Abbildung 5.8 dargestellt. Für die Unterscheidung von Rollenkomponenten und Kernkomponenten werden hierbei die Stereotypen «`rolecomponent`» und «`core`» eingeführt. In obiger Abbildung ist ein zweischichtiger Aufbau enthalten (`cr1` und `cr2` in der ersten, die Kernkomponente in der zweiten Schicht), der nicht unmittelbar ersichtlich ist. Es bietet sich daher an, Rollenkomponenten entsprechend ihren Schichten anzuordnen und somit die Struktur des Zwiebelschalenmodells zu übernehmen.

5.5 Framework Profile

Mit Hilfe der in den letzten beiden Abschnitten eingeführten Erweiterungen können nun Komponentenframeworks, wie sie in dieser Arbeit vorgestellt und diskutiert werden, auf einfache Weise dokumentiert werden. Für eine vollständige Frameworkspezifikation, die dem Anwender das benötigte Wissen über Konzepte und Zusammenhänge vermittelt, sind mindestens die folgenden Bestandteile nötig:

- Ein Fachmodell basierend auf Rollen, sowie ihren externen und internen Abhängigkeiten untereinander. Für die Darstellung von Fachmodellen wird das Role Profile aus dem letzten Abschnitt verwendet.
- Eine Konkretisierung des Verhaltens der einzelnen Rollen — basierend auf Schnittstellen und zugeordneten Kommunikationabläufen. Hierfür werden verschiedene Komponententypen deklariert und definiert, sowohl in der Blackbox-Sicht als auch in einer geeigneten Greybox-Sicht (außer bei Kernkomponenten), um das Zusammenspiel zwischen den Rollen darzulegen und damit eine Integration unterschiedlicher Komponentenframeworks zu ermöglichen. Zudem eine Anordnung der vorgesehenen Rollenkomponenten entsprechend dem Zwiebelschalenmodell (vgl. Abschnitt 3.6.1).
- Ein Verständnis darüber, welche Bestandteile des Frameworks vom Anwender zu realisieren sind und welche Teile mitgeliefert werden. Im Normalfall sind alle benötigten Rollenkomponenten bereits realisiert, und die jeweiligen Fachkomponenten müssen lediglich durch die Realisierung der entsprechenden Kernkomponenten vervollständigt werden. In Sonderfällen, wie sie in Abschnitt 3.6 diskutiert werden, erfordert die Integration von Komponentenframeworks die Ersetzung bestehender Rollenkomponenten.
- Weitere Komponenten des Frameworks, insbesondere Mediatoren und technische Komponenten, sind vor dem Anwender verborgen. Bei der Instantiierung eines Komponentenframeworks ist es jedoch hilfreich, einen Konfigurationsmechanismus vorzusehen, um bestimmte Charakteristika optimal auf die aktuelle Anwendung anzupassen. Hierfür ist eine Komponente als Repräsentant des Frameworks nötig, über die solche Einstellungen vorgenommen werden. Als Mechanismus ist beispielsweise eine attribut-basierte Schnittstelle denkbar, wie sie im CORBA Komponentenmodell vorgesehen ist. Dadurch ist es dem Anwender möglich, die Instantiierung und Konfiguration weitgehend werkzeugunterstützt vorzunehmen.

Für die Dokumentation von Komponentenframeworks können somit die bisher eingeführten Erweiterungen der UML verwendet werden. Insbesondere bietet sich die Darstellung von Komponententypen als hierarchisch zerlegter Komponentenverbund an, um die einzelnen „plugs“ des Frameworks zu spezifizieren und die erlaubten Verbindungen untereinander darzulegen. Mit Hilfe einzelner Komponentendiagramme können dynamische Veränderungen an der Menge der Subkomponenten, deren Rollen und den einzelnen Verbindungen verdeutlicht werden. Letztlich bleibt für das Framework Profile nur die Einführung eines Stereotyps «`realized`», um in der Spezifikation auf die bereits implementierten Bestandteile der Anwendung hinzuweisen.

5.6 Zusammenfassung

In diesem Kapitel wurde eine Erweiterung der UML für die Spezifikation von Komponentenframeworks vorgestellt und diskutiert. Die Grundlage ist hierbei die Einführung graphischer Beschreibungstechniken für die Spezifikation komponentenbasierter Anwendungen. Entsprechend den Ergebnissen aus Abschnitt 3.4 werden Schnittstellen und interner Aufbau einer Komponente in Form von Komponententypdiagrammen und konkrete Ausprägungen mit Hilfe von Komponentendiagrammen dargestellt. Während diese Art der Beschreibung für statische Aussagen über eine Anwendung ausreichend ist, sind strukturelle Veränderungen nur umständlich darstellbar. In den vergangenen Abschnitten wurde eine zeitliche Aneinanderreihung von Komponentendiagrammen vorgeschlagen, um strukturelle Veränderungen zur Laufzeit (Auf- und Abbau von Kanälen, Erzeugen und Entfernen von Subkomponenten) darzustellen. Diese Repräsentation ist jedoch in vielen Fällen umständlich und zu unpräzise. Wie in der Einleitung dieses Kapitels dargestellt, verfolgen eine Reihe aktueller Arbeiten das Ziel, solche Details in der Spezifikation einer komponentenbasierten Anwendung (beispielsweise unter Verwendung von Zusicherungen) genauer repräsentieren zu können.

Weiterhin wurde eine Notation für Rollen und ihre Beziehungen eingeführt, die in den vergangenen Kapiteln für die Darstellung von Fachmodellen eingesetzt wurde. Durch die direkte Abbildung von Rollen auf Komponententypen ist zudem eine nahtlose Eingliederung des Rollenbegriffs in die komponentenrelevanten Diagrammarten gegeben. Von besonderer Bedeutung ist die Erweiterung von Sequenzdiagrammen hinsichtlich der Unterstützung von Rollen. Damit kann einerseits das Zusammenspiel zwischen den Komponenten hinsichtlich der jeweils angesprochenen Rolle verfeinert werden. Andererseits ist es auf diese Weise möglich, Abhängigkeiten zwischen den Rollen derselben Komponente zu detaillieren und in Form von Kommunikationsverhalten zu spezifizieren. Damit ergibt

sich ein direkter Übergang zu den Rollenkomponenten und ihrer korrekten Zusammenstellung im Rahmen der Realisierung einer Fachkomponente.

Komponententypdiagramme eignen sich zudem für die Spezifikation von Komponentenframeworks, wie in Abschnitt 5.5 gezeigt. Die hierarchische Zerlegung einer Komponente wird hierbei so interpretiert, daß eine „Frameworkkomponente“ eine Reihe von „plugs“ (über entsprechende Rollen) definiert, in die bei der Instantiierung des Frameworks die jeweiligen Fachkomponenten gesetzt werden. Hierbei ergeben sich zwei wichtige Unterschiede zu regulären Komponententypdiagrammen: Einerseits werden lediglich erlaubte Verbindungen zwischen den Subkomponenten spezifiziert, da der Zusammenhang zu den Schnittstellen der Frameworkkomponente für den Anwender nicht relevant ist. Damit entfällt prinzipiell auch die Verwendung der Gluekomponente (vgl. Abschnitt 5.3). Andererseits ist die implizite Annahme über eine strenge Kapselung von Komponenten aufgehoben, da Fachkomponenten mehrere plugs aus mehreren Komponentenframeworks gleichzeitig ausfüllen können.

Kapitel 6

Prozeßmusterorientiertes Vorgehensmodell

Softwareentwicklung basierend auf der Zusammenstellung und Integration geeigneter Komponentenframeworks verspricht eine deutliche Steigerung von Effektivität und Effizienz und verbessert damit die Balance zwischen Kosten, Zeit und Qualität. In Kapitel 3 wurden Komponentenframeworks als sehr flexible Basistechnik konzipiert, die — ebenso wie objektorientierte Frameworks — die Wiederverwendung von Designwissen und zugehörigen Implementierungen erlauben, im Gegensatz zu diesen jedoch deutlich flexibler und handhabbarer gestaltet sind. Aufgrund der Verwendung eines mächtigen Rollenkonzepts zeigen sich Komponentenframeworks „integrationsoffen“ und ermöglichen damit die Zerlegung und spätere Komposition einzelner Anwendungssichten. Der tatsächliche Nutzen von Komponentenframeworks erfordert jedoch die pragmatische Anwendbarkeit und die Beherrschung der einhergehenden Prozesse. Hierfür wurden in Kapitel 3 und Kapitel 4 die Grundlagen geschaffen in Form einer technischen Realisierung und der Einführung graphischer Beschreibungstechniken für die Spezifikation von Komponentenframeworks.

Die umfassenden Erfahrungen aus dem Bereich objektorientierter Frameworks im speziellen [FSJ99] und aus den Bemühungen zur Wiederverwendung von Software im allgemeinen [JGJ97] zeigen jedoch, daß das Vorhandensein moderner Konzepte allein noch nicht zu ihrer erfolgreichen Verwendung und damit zu dem gewünschten Nutzen für die Softwareentwicklung führt. Im Fall von Komponentenframeworks ist zudem davon auszugehen, daß die gesteigerte Flexibilität mit einer gesteigerten Komplexität einhergeht

und auf diese Weise die Situation noch verschärft wird. Die Untersuchung der vorgestellten Beispiele aus den vorangegangenen Kapiteln zeigt eine Reihe von Vereinfachungen, die in Alltagssituationen nicht vorausgesetzt werden können. Insbesondere bei der Integration unterschiedlicher Fachmodelle und des zugrundeliegenden Verständisses über die Konzepte und Zusammenhänge des Anwendungsbereichs ist davon auszugehen, daß in vielen Fällen umfangreiche Anpassungen erforderlich sind. Daraus resultiert ein langfristiger Erfahrungsprozeß für die Entwicklung und Anwendung eines Komponentenframeworks, wobei die Ausgestaltung anhand der gewonnenen Erkenntnisse kontinuierlich verbessert wird.

Das Ziel dieses Kapitels liegt nun darin, spezielle Anforderungen und Gegebenheiten der Softwareentwicklung von und mit Komponentenframeworks in eine entsprechende Methodik einzubetten und auf diese Weise dem Entwickler und dem Anwender Richtlinien und Anleitungen für eine zielgerichtete Unterstützung während der unterschiedlichen Phasen des Entwicklungsprozesses zur Verfügung zu stellen. Eine solche Methodik basiert auf Erfahrungen aus der allgemeinen Softwarepraxis („Was zeichnet eine verständliche Dokumentation aus?“), aus der Entwicklung von Komponenten („Wie werden Schnittstellen präzise spezifiziert?“ oder „Wie läßt sich die Wiederverwendung einer Komponente steigern?“) und schließlich aus den Besonderheiten von Komponentenframeworks („Wie findet ein Entwickler die optimale Abstraktion?“), auf die im weiteren gezielt eingegangen wird.

Das hier vorgestellte Vorgehensmodell basiert auf den Beschreibungstechniken aus Kapitel 5 sowie einer Reihe unterschiedlicher *Prozeßmuster* [BRSV98c]. Prozeßmuster stellen einen sehr flexiblen Ansatz zur Gestaltung von Entwicklungsprozessen dar, der zudem leicht an bestimmte Gegebenheiten (in diesem Fall die Entwicklung oder Verwendung von Komponentenframeworks) angepaßt werden kann.

6.1 Vorgehensmodelle

Der Nutzen eines klar definierten und strukturierten Vorgehens für die Entwicklung komplexer Software, wie sie durch moderne, große Anwendungssysteme gefordert wird, ist allgemein anerkannt. Demgegenüber divergieren die Ansichten darüber, welches Vorgehensmodell für welche Projektsituation am vorteilhaftesten ist — zu unterschiedlich sind die Projekte, die Erfahrungen der Beteiligten und die verfügbaren, teils kommerziellen Vorgehensmodelle mit ihren individuellen Stärken und Schwächen. Bei genauerer Betrachtung verbreiteter Ansätze, angefangen vom klassischen Wasserfallmodell [Roy70]

über das Spiralmodell [Boe86] bis hin zum modernen V-Modell [DW99], zeigen sich unterschiedliche Defizite in der Anwendung zur Erstellung komponentenbasierter Anwendungen. Nach Szyperski eignen sich diese Vorgehensmodelle lediglich für die Erstellung von Komponenten, nicht aber zur Entwicklung kompletter Anwendungssysteme — ein geeigneter Ansatz für komponentenbasierte Anwendungssysteme sei noch nicht verfügbar [Szy97].

Seit dieser Aussage sind neue Vorgehensmodelle entstanden oder bestehende im Hinblick auf Komponenten verfeinert worden, beispielsweise das V-Modell [ABD+99], Catalysis [DW98] oder der Rational Unified Process [Kru99]. Die grundlegenden Probleme der Softwareentwicklung mit Komponenten können als verstanden angesehen werden, und in den folgenden Abschnitten werden wir daher nur kurz auf elementare Anforderungen und Randbedingungen eingehen, um dann den weiteren Schwerpunkt auf die Besonderheiten bei der Erstellung und Verwendung von Komponentenframeworks zu legen.

> **Erläuterung 6.1 (Vorgehensmodell)** Ein Vorgehensmodell basiert auf einer Reihe von Dokumententypen und definiert eine übergreifende Strategie in Form einer Sammlung konkreter Entwicklungsschritte und Richtlinien. Ein Entwicklungsschritt bedeutet dabei stets eine Veränderung eines oder mehrerer der definierten Dokumententypen. Ein Entwicklungsprozeß ist eine „Instanz“ eines Vorgehensmodells und repräsentiert eine konkrete Abfolge von Entwicklungsschritten.

Vorgehensmodelle unterteilen einen Entwicklungsprozeß üblicherweise in eine Reihe von *Phasen*, beispielsweise „Analyse“, „Definition“, „Entwurf“, „Implementierung“ und „Systemtest, Abnahme, Wartung“ [Fri95]. Die Aufstellung der eingesetzten Dokumententypen orientiert sich an dieser Aufteilung und unterscheidet daher z.B. Analysedokumente von Entwurfsdokumenten. Einzelne Arbeitsschritte beschreiben nun die Transformation eines Dokuments (etwa in Form einer Verfeinerung) oder die Erstellung neuer Dokumente aus bestehenden Informationen. Um einen konkreten Entwicklungsprozeß an die Gegebenheiten der jeweiligen Projektsituation anzupassen, bieten Vorgehensmodelle oft eine Möglichkeit zum *Tailoring*, also der Feinabstimmung der eingesetzten Dokumente und Arbeitsschritte.

6.1.1 Komponentenbasierte Softwareentwicklung

Traditionelle Vorgehensmodelle sind hauptsächlich ausgerichtet auf die Erstellung und nicht auf die Wiederverwendung von Software. Diese Beobachtung gilt speziell im Umfeld eines Paradigmas wie der Objektorientierung, das vor allem für die adäquate Repräsentation eines Anwendungsbereichs und weniger für die pragmatische Wiederverwendung bestehender Softwarefragmente ausgelegt ist. So ist die wichtigste Forderung an ein

komponentenbasiertes Vorgehensmodell die umfassende und ganzheitliche Einbindung der Wiederverwendung von Softwarekomponenten. Das Zusammenspiel aus Analyse und Synthese muß sorgsam aufeinander abgestimmt werden, um den Nutzen einer verstärkten Wiederverwendung nicht durch einen unnötig hohen Aufwand zunichte zu machen. Neben diesem Aspekt stellen wir eine Reihe weiterer Anforderungen an ein geeignetes Vorgehensmodell, die wir im folgenden kurz diskutieren:

Iterativ, inkrementell: Moderne Anwendungssysteme erreichen oft einen Umfang und eine Komplexität, die zu Entwicklungszeiten von bis zu mehreren Jahren führen können. Zusammen mit der Erkenntnis, daß ein Großteil der bei der Entwicklung anfallenden Kosten im Bereich der Wartung eines Systems liegen, ist gerade auch bei komponentenbasierten Anwendungen ein „langlebiger“ Entwicklungsprozeß notwendig, der kontinuierliche Veränderungen auch nach der Auslieferung des Produkts umfaßt. In diesem Zusammenhang ist eine einmalige Anforderungsanalyse, wie sie durch viele Vorgehensmodelle nahegelegt ist, nicht praktikabel: je länger die Entwicklung dauert, desto wahrscheinlicher werden Veränderungen in der ursprünglichen Menge der Anforderungen.

Eine iterative, inkrementelle Herangehensweise (vgl. z.B. [Kru99]) impliziert eine kontinuierliche Auseinandersetzung mit den aktuellen Anforderungen und sichert somit auch langfristig die Adäquanz der jeweils erstellten Lösung. Ein iteratives Vorgehen bezeichnet hierbei die sukzessive Erweiterung und Verfeinerung einer Anwendung in einem mehrfachen Durchlaufen der oben aufgeführten Phasen von Analyse bis Systemtest. Demgegenüber wird in der inkrementellen Entwicklung eine Reihe aufeinander aufbauender Prototypen definiert, von denen jeder einzelne beispielsweise iterativ realisiert wird.

Architekturzentriert: Die Architektur eines Anwendungssystems vermittelt das prinzipielle und übergreifende Verständnis der eingesetzten Konzepte und ihrer Zusammenhänge. Softwarearchitekturen — wie sie auch in dieser Arbeit vorgestellt wurden — repräsentieren beispielsweise die zugrundeliegende Sicht auf den Anwendungsbereich, die Einbettung in die technische Infrastruktur, aber auch das Wissen über Anwendung und Erweiterung eines Softwaresystems (vgl. Abschnitt 2.2).

Eine Softwarearchitektur ist eine mehrschichtige Spezifikation, die je nach Sichtweise und Zielsetzung unterschiedlich repräsentiert sein kann. Damit agiert die Architektur eines Softwaresystems als gemeinsame Basis für die Zusammenführung unterschiedlicher fachlicher und technischer Aspekte und für das Verständnis der beteiligten Personen. Ein architekturzentriertes Vorgehensmodell stellt diesen kritischen Bereich der Entwicklung in den Vordergrund und erreicht damit sowohl

eine handhabbare Komplexität als auch ein verbessertes Zusammenspiel der unterschiedlichen, beteiligten Rollen [Kru99]. Gerade im Bereich der komponentenbasierten Entwicklung gelingt auf diese Weise eine bessere Organisation und Strukturierung des Vorgehens anhand der beteiligten Komponenten verbunden mit einer präziseren Trennung der Zuständigkeiten.

Anwendungsfallgetrieben: Die Aufteilung von Verantwortlichkeiten auf die einzelnen Komponenten einer Anwendung führt häufig zu einer isolierten und introvertierten Betrachtung und Realisierung einer Komponente unter Vernachlässigung einer ganzheitlichen Sicht auf das Anwendungssystem. Ein geeignetes Vorgehensmodell wirkt dieser Tendenz entgegen, indem die Aufgabe jeder einzelnen Komponente im Zusammenhang definierter Kollaborationen beschrieben wird, die sich aus den einzelnen Use-Cases des Anwendungssystems ableiten lassen [ML98].

Top-Down und Bottom-Up: Das „klassische“ top-down Vorgehen, bei dem, ausgehend von einer Analysephase, Schritt für Schritt ein Design zu einer lauffähigen Anwendung verfeinert wird, ist ungeeignet für Entwicklungsprozesse, in denen die Wiederverwendung bestehender Komponenten im Zentrum steht. Konkrete Ausgestaltungen der beteiligten Komponenten stehen hier erst zu einem sehr späten Zeitpunkt in der Entwicklung zur Verfügung. Dadurch reduziert sich beispielsweise die Wahrscheinlichkeit, auf den diversen Komponentenmärkten Komponenten zu finden, die den spezifizierten Anforderungen entsprechen oder sich zumindest mit erträglichem Aufwand anpassen lassen. Demgegenüber läßt sich bei einem reinen Bottom-Up Vorgehen die Wiederverwendung maximieren, da das System ausgehend von einer Menge geeigneter, vorhandener Komponenten konzipiert wird. Hierbei stellt sich jedoch häufig (zu spät) heraus, daß die eigentlichen Anforderungen durch die Anwendung nicht genügend berücksichtigt wurden und in der Folge umfangreiche Verbesserungen nötig sind.

Ein Vorgehensmodell, das den top-down Ansatz mit dem bottom-up Ansatz in geeigneter Form kombiniert, erzielt eine durch die Anforderungen getriebene Entwicklung bei einem gesteigerten Grad an Wiederverwendung bestehender Lösungen. In einem so gearteten Entwicklungsprozeß werden die analysierten Anforderungen kontinuierlich mit den Funktionalitäten passender, wiederzuverwendener Komponenten abgebildet und durch entsprechende Rückschlüsse modifiziert und verfeinert. Damit wird der Aspekt der Wiederverwendung bereits in frühe Phasen der Entwicklung einbezogen.

Evolutionär: Die Langlebigkeit moderner Entwicklungsprozesse fordert eine intensive Auseinandersetzung mit den zu erwartenden Veränderungen an einem Anwendungssystem aufgrund sich wandelnder Anforderungen („Design for Change“). Ein

evolutionäres Vorgehensmodell propagiert die frühzeitige Berücksichtigung kritischer Einflüsse und die Ausrichtung entsprechender Designentscheidungen im Hinblick auf zu erwartende Veränderungen. Idealerweise kann in einem solchen Fall veränderten Anforderungen durch einfache, lokal begrenzte Modifikationen der Anwendung begegnet werden (beispielsweise durch das Austauschen einzelner Komponenten).

Anpassbar: Damit ein Entwickler einen hohen Nutzen durch das verwendete Vorgehensmodell erfährt, ist es erforderlich, den Entwicklungsprozeß sehr detailliert zu beschreiben und konkrete Vorgaben und Richtlinien zu verfassen. Diese Präzisierung der Aussagen erfordert eine genaue Abstimmung auf die konkrete Projektsituation. Unter dem Begriff *Tailoring* wird die flexible Anpassung eines Vorgehensmodells bezeichnet, die hauptsächlich am Projektanfang durchgeführt wird, ebenso jedoch auch im Verlauf der Entwicklung eine Berücksichtigung einer veränderten Projektsituation ermöglicht. Gerade hinsichtlich langlebiger Entwicklungsprozesse ist diese Anpassbarkeit eine wichtige Voraussetzung für den Projekterfolg.

Erweiterbar: Gerade die Softwareentwicklung ist eine Disziplin, die einer kontinuierlichen Wandlung unterzogen ist. Neue Erkenntnisse aber auch neue Bedingungen aus dem Projektumfeld (spezielle Hardware, neue Technologien) erfordern ein Vorgehensmodell, das nicht starr ausgelegt ist, sondern die Möglichkeit zur Erweiterung bietet. Nur auf diese Weise ist gewährleistet, daß ein Entwicklungsprozeß „state of the art“ ist und den Anforderungen der jeweiligen Projektsituation genügt.

Moderne Vorgehensmodelle, wie beispielsweise der Rational Unified Process [Kru99] oder Catalysis [DW98] erfüllen diese Anforderungen weitestgehend und sind somit prinzipiell für die komponentenbasierte Softwareentwicklung geeignet. Dennoch sind die Unterschiede mannigfaltig und so ist die Entscheidung für ein Vorgehensmodell stets kritisch, da ein späterer Wechsel nur unter hohem Aufwand möglich ist.

6.1.2 Entwicklung und Anwendung von Komponentenframeworks

Prinzipiell gelten die Erkenntnisse aus dem vergangenen Abschnitt ebenfalls für die Softwareentwicklung mit Komponentenframeworks. Die zugrundeliegenden Mechanismen — Analyse und Synthese — sind lediglich deutlich komplexer in ihrer Anwendung. Aus diesem Grund trägt die Wahl eines geeigneten Vorgehensmodell noch entscheidender zum Erfolg bzw. Mißerfolg eines Softwareprojekts bei. Die folgenden Besonderheiten müssen hierbei besonders berücksichtigt werden:

- Komponentenframeworks als „unit of reuse“ weisen eine strukturierte Schnittstelle auf, die entsprechend den vorgesehenen „plugs“ organisiert ist. Damit ist ebenso wie bei Komponenten eine Kapselung gegeben, wobei sich die Instantiierung deutlich aufwendiger gestaltet. Um das Verständnis der Konzepte und Zusammenhänge zu vermitteln, wird jedem Komponentenframework ein Fachmodell zugeordnet, das insbesondere über entsprechende Rollenspezifikationen die „Contracts“ der einzelnen Schnittstellen definiert. Die Wahl der richtigen Abstraktionsebene für ein Fachmodell entscheidet hierbei über die Wiederverwendbarkeit: ein zu abstrakt gestaltetes Modell kann zwar leicht auf unterschiedliche Anwendungssituationen angepaßt werden, ist jedoch aufgrund der schwachen Aussagen über den Anwendungsbereich von geringem Nutzen. Demgegenüber repräsentiert ein zu konkretes Fachmodell zwar fest umrissene Anwendungsfälle, das zugehörige Komponentenframework ist jedoch nur in einer geringen Zahl von Fällen einsetzbar.

- Die Anforderungen der Integration von Komponentenframeworks laufen prinzipiell der gewünschten Kapselung zuwider. Damit das Zusammenspiel zwischen den jeweiligen Implementierungen geeignet abgestimmt werden kann, muß ein Komponentenframework „interne Details“ preisgeben. Entsprechend den Ergebnissen aus Kapitel 3 steht Komponentenframeworks über Kern- und Rollenkomponenten ein zweistufiger Mechanismus zur Verfügung, der die Zusammenführung ermöglicht, auch wenn dies unter Umständen ein Austausch bestehender Implementierungsfragmente eines Komponentenframeworks bedeutet. Ein solches Eingreifen in die Realisierung eines Komponentenframeworks erfordert ein wohlüberlegtes Vorgehen bei der Konzeption und Implementierung eines Komponentenframeworks.

- „Late Integration“, also die späte Zusammenführung der einzelnen Aspekte einer Anwendung, beschränkt die Freiheiten des Anwenders bei der Beeinflussung von Design und Implementierung. So können die durch die Komponentenframeworks vorgegebenen Strukturen nicht beeinflußt werden, weswegen auch die Anpassung unterschiedlich implizierter Abstraktionsstufen mit Hilfe eines Hilfskonstrukts (vgl. Abschnitt 3.6.3) erfolgt.

Insgesamt kann den speziellen Anforderungen hinsichtlich der Entwicklung und Anwendung von Komponentenframeworks mit einem Vorgehensmodell, das für die komponentenbasierte Softwareentwicklung ausgelegt ist (und damit den oben aufgeführten Punkten genügt), gut entsprochen werden. Für ein besseres Verständnis führen wir im weiteren die einzelnen Phasen auf, die bei der Anwendung eines Komponentenframeworks auftreten:

Auswahl Komponentenframeworks: In einer Kombination aus top-down und bottom-up Vorgehen gleicht der Entwickler die Analyse des Anwendungsbereichs mit den Rollenmodellen kommerzieller Komponentenframeworks ab. Das Ergebnis dieser

Aktivität besteht in einer Aufstellung diverser Komponentenframeworks, die entweder selbst entwickelt oder von Drittanbietern eingekauft werden können. Hierbei spielen zusätzliche Faktoren wie z.B. Preismodelle oder Qualität eine wichtige Rolle (vgl. [BJR+01]).

Integration Rollenmodelle: Durch die Integration von Rollenmodellen (entsprechend den Erkenntnissen aus Abschnitt 3.3.7) entsteht aus unterschiedlichen Abstraktionen des Anwendungsbereichs eine neue Sicht auf das System. Bei der Zusammenführung entdeckt der Entwickler — mit der Hilfe geeigneter Werkzeuge — bei der Komposition von Rollenspezifikationen auftretende Konflikte. Aus diesem Grund verläuft der Integrationsvorgang in mehreren Iterationen, bis eine geeignete Konstellation von Komponentenframeworks gefunden und integriert ist.

Komposition Rollenkomponenten: Eines der Ergebnisse der Integration ist ein genaues Verständnis darüber, wie die einzelnen Rollenkomponenten zusammenwirken, um die Funktionalität einer Fachkomponente zu erbringen. Diese Realisierung erfordert unter Umständen vom Anwender, bestimmte Rollenkomponenten neu zu erstellen.

Realisierung Kernkomponenten: Der letzte Schritt besteht in der Realisierung der einzelnen Kernkomponenten, die — zusammen mit den Rollenkomponenten — die „plugs“ der eingesetzten Komponentenframeworks ausfüllen. Hierbei wird auch die Anpassung unterschiedlicher Abstraktionsebenen durchgeführt, wie in Abschnitt 3.6.3 erläutert.

Der Aufwand für die Integration der Komponentenframeworks nimmt einen beträchtlichen Anteil an der gesamten Entwicklung ein. Hierbei müssen nicht nur die jeweiligen Fachmodelle zusammengeführt werden sondern im weiteren die den Rollen zugeordneten Verhaltensspezifikationen auf geeignete Weise kombiniert und dabei entstehenden Konflikten begegnet werden. Der Erfolg der Integration hängt nicht zuletzt von den verwendeten Komponentenframeworks ab: je besser ein Komponentenframework entworfen wurde, je genauer also die tatsächlichen Anwendungsfälle antizipiert wurden, desto einfacher gestaltet sich die Zusammenführung und umso weniger Aufwand erfordert die Anpassung. Die erfolgreiche Softwareentwicklung mit Komponentenframeworks hängt also von Weitsicht und Geschick bei der Erstellung eines Komponentenframeworks ab. Qualitätsbestimmende Merkmale werden nun anhand der folgenden Eigenschaften diskutiert:

Größe: Der Umfang eines Frameworks beeinflußt entscheidend die Handhabbarkeit und Komplexität der entstehenden Anwendung. Je monolithischer ein Komponentenframework geartet ist, desto geringer werden Anwendbarkeit und damit Wiederverwendung. Umgekehrt ist eine Anwendung aus vielen kleinen Komponentenframeworks flexibler und präziser auf die Anforderungen zugeschnitten — der Aufwand

für die Integration erhöht sich jedoch deutlich. Auch im Hinblick auf sich ändernde Anforderungen erscheint die „richtige“ Größe eines Komponentenframeworks als essentiell, da beispielsweise neue Anforderungen einen Austausch eines verwendeten Komponentenframeworks notwendig machen.

Abstraktionsgrad: Erfahrungen mit objektorientierten Frameworks zeigen, daß die richtige Wahl der Abstraktionsebene essentiell für die Wiederverwendbarkeit eines Frameworks ist. Sind die Annahmen über den Anwendungsbereich zu vage, so ist die Aussagekraft und somit der Nutzen des Frameworks beschränkt — wird ein ganz spezieller Anwendungsbereich vorausgesetzt, so ist dagegen die Wiederverwendbarkeit des Frameworks nicht besonders hoch.

Information Hiding: Komponentenframeworks wurden in dieser Arbeit so konzipiert, daß alle Realisierungsdetails außerhalb der einzelnen Fachkomponenten (und damit vor allem technische Komponenten und Mediatoren) vor dem Anwender des Frameworks verborgen sind[1]. Doch auch hinsichtlich der eingesetzten Fachkomponenten besteht die Möglichkeit für den Frameworkentwickler, bestimmte Bereiche des Fachmodells auszublenden oder bestimmte Fachkomponenten zwar einzuführen, sie jedoch nicht in Form eines „plugs“ in die Schnittstelle des Frameworks aufzunehmen. In diesen Fällen wird die benötigte Kernkomponente als Bestandteil des Frameworks mitgeliefert. Auf diese Weise läßt sich eine Unterscheidung der Fachkomponenten einführen in solche, die unmittelbar für die Instantiierung des Frameworks benötigt werden und solche, die nur für erweiterte Formen der Integration interessant sind.

Die aufgeführten Punkte stellen nur einen kleinen Teil der zu berücksichtigen Qualitätskriterien dar. Weitere Merkmale, wie beispielsweise eine umfangreiche Dokumentation oder die Einbeziehung bestehender Standards, ergeben sich aus den allgemeinen Erfahrungen des Software-Engineerings (siehe z.B. [Sam97]). Letztlich hängt die Qualität und damit die Wiederverwendbarkeit eines Komponentenframeworks von Geschick und Erfahrung der Entwickler ab, und es ist zu erwarten, daß erst die kontinuierliche Verwendung und Verbesserung eines Frameworks im Rahmen eines langlebigen Entwicklungsprozesses zur gewünschten Qualität führt.

[1] Die Möglichkeit, auch technische Komponenten in den Vorgang der Integration einzubeziehen, wird in Abschnitt 3.7 diskutiert.

6.1.3 Allgemeine Richtlinien

Die von einem Vorgehensmodell vorgeschlagenen Strategien enthalten bewährtes Wissen hinsichtlich unterschiedlicher Aspekte der Entwicklung. Es bietet insbesondere konkrete Richtlinien und Vorgaben, um beteiligte Personen bei der effizienten Projektabwicklung zu unterstützen. In diesem Abschnitt werden eine Reihe von Überlegungen hinsichtlich der Entwicklung von Komponentenframeworks als „best practices“ verfaßt, die Qualität und Wiederverwendbarkeit eines Frameworks steigern.

Minimierung von Abhängigkeiten: Frameworks entstammen dem Wunsch, bewährte Designelemente aus bestehenden Anwendungen zu isolieren und für vergleichbare Situationen wiederverwendbar zu gestalten. Um die Anwendbarkeit des entstehenden Komponentenframeworks zu steigern, gilt es, die Abhängigkeiten innerhalb des bestehenden Systems zu analysieren. Das Framework soll nun jene Komponenten umfassen, deren Abhängigkeiten untereinander hoch sind, während die Abhängigkeiten zu anderen Bestandteilen der Anwedung minimal sind. Diese Eigenschaft der „lokalen Auswirkungen“ sorgt für einen geringen Integrationsaufwand und damit für eine Austauschbarkeit der beteiligten Frameworks einer Anwendung im Rahmen sich ändernder Anforderungen. Diese Flexiblität ist eine Voraussetzung für die langelebige Anwendungsentwicklung mit Komponentenframeworks (vgl. auch [Mat00]).

Standardisierte Modelle: Ein gemeinsames Verständnis des Anwendungsbereichs ist eine ideale Voraussetzung für die Integration von Komponentenframeworks. Für viele Anwendungsdomänen existieren heute bereits Modellierungen, die sich in der Praxis bewährt haben und auf einfache Weise erweitert werden können. Je mehr sich die Verwendung solcher „vorgefertigten“ Modelle durchsetzt, desto einheitlicher wird die Terminologie, mit der das Domänenwissen zwischen Entwicklern und Anwendern ausgetauscht wird. In IBM’s *San Francisco*-Framework [SAN] stehen eine Reihe von „common business objects“ zur Verfügung, die gängige Bereiche aus der Geschäftswelt modellieren (z.B. Mitarbeiter, Geschäftspartner, etc.).

Integrationsoffenheit: Im Gegensatz zu objektorientieren Frameworks sind Komponentenframeworks nicht monolithisch konzipiert und besonders auf das Zusammenspiel mit anderen Komponentenframeworks ausgelegt. Gerade da der in dieser Arbeit eingeführte Mechanismus auf Basis von Abhängigkeiten einzelner Rollen unter Umständen sehr komplex ist, entscheidet eine umfassende Dokumentation der Querbeziehungen zwischen den Frameworks über den Erfolg späterer Integrationsbemühungen. Aber auch ohne konkrete Vorstellungen über weitere Aspekte der Anwendung ist die „Öffnung“ eines Komponentenframeworks bei der Entwicklung wichtig: gelingt es dem Entwickler beispielsweise, gemeinsame Zustände im

Rollenverbund richtig vorherzusagen, so vereinfachen sich Integration und Implementierung deutlich. Dennoch gilt die Richtlinie, möglichst viele Aspekte des Verhaltens in der Rolle zu kapseln, um die Anwendung (und damit die Realisierung der Kernkomponenten) nicht unnötig zu erschweren.

Dekomposition: Durch die Dekomposition des Rollenverhaltens einer Fachkomponente in eine Delegationskette unterschiedlicher Rollenkomponenten entsteht ein präzises Verständnis der durch das Komponentenframework vorgegebenen Implementierung. Prinzipiell gilt die Regel, daß eine feingranulare Zerlegung sowohl hinsichtlich der Rollen einer Fachkomponente als auch der Rollenkomponenten bei ihrer Realisierung die Integration vereinfacht. Dies ergibt sich aus der höheren Wahrscheinlichkeit, gemeinsame Rollen bzw. Rollenkomponenten identifizieren zu können. Eine feingranulare Zerlegung geht jedoch ebenfalls mit erhöhter Komplexität einher, so daß auch hier ein geeigneter Mittelweg gefunden werden muß.

In dieser Form sind „best practices“ für einen Entwickler unhandlich, nicht direkt anwendbar und in ihren teils gegenläufigen Auswirkungen nicht abschätzbar. Aus diesem Grund stellen wir im weiteren Prozeßmuster vor, die eine verständliche, anwendbare Repräsentation dieser Richtlinien ermöglichen.

6.2 Einführung in Prozeßmuster

Viele der — teils kommerziellen — Vorgehensmodelle, die in der Einleitung zu diesem Kapitel aufgeführt und mehr oder minder häufig in Alltagsprojekten eingesetzt werden, erfüllen großteils die Anforderungen aus Abschnitt 6.1.1. Dennoch weisen sie konzeptbedingte Vor- und Nachteile auf, die den Entwicklungsprozeß nachhaltig beeinflussen und wodurch die Entscheidung für ein konkretes Vorgehensmodell als kritischer Faktor des Projekterfolgs anzusehen ist. Ein Ansatz, der noch stärker Aspekte wie Flexibilität und dynamische Anpassung an die Projektsituation bietet und damit Probleme zu starrer Vorgaben umgeht, ist durch sogenannte *Prozeßmuster* gegeben, wie wir sie in [BRSV98a, BRSV98c] für die Entwicklung komponentenbasierter Anwendungen vorgeschlagen haben. Ein Vorgehensmodell basierend auf Prozeßmuster verwendet einen Mechanismus, der Entwicklern häufig bereits aus Konzepten wie „Designpattern“ [GHJV95] oder „Architectural Pattern“ [BMR+96] geläufig ist. Ein solches Vorgehensmodell basiert auf den folgenden drei Bestandteilen:

- Eine hierarchische *Ergebnisstruktur* deklariert und organisiert die Artefakte, die im Verlauf eines Entwicklungsprozesses erstellt werden. So umfaßt die Ergebnisstruktur zum Beispiel Analysedokumente gleichfalls wie einzelne Testabläufe. Obwohl

die vorgesehenen Dokumententypen entlang der klassischen Entwicklungsphasen organisiert sind, ist durch die Ergebnisstruktur keine Reihenfolge impliziert. Vielmehr ist davon auszugehen, daß im Verlauf der Entwicklung ganz unterschiedliche Dokumente erstellt werden bis die Ergebnisstruktur komplett „gefüllt“ und damit das Projekt abgeschlossen ist.

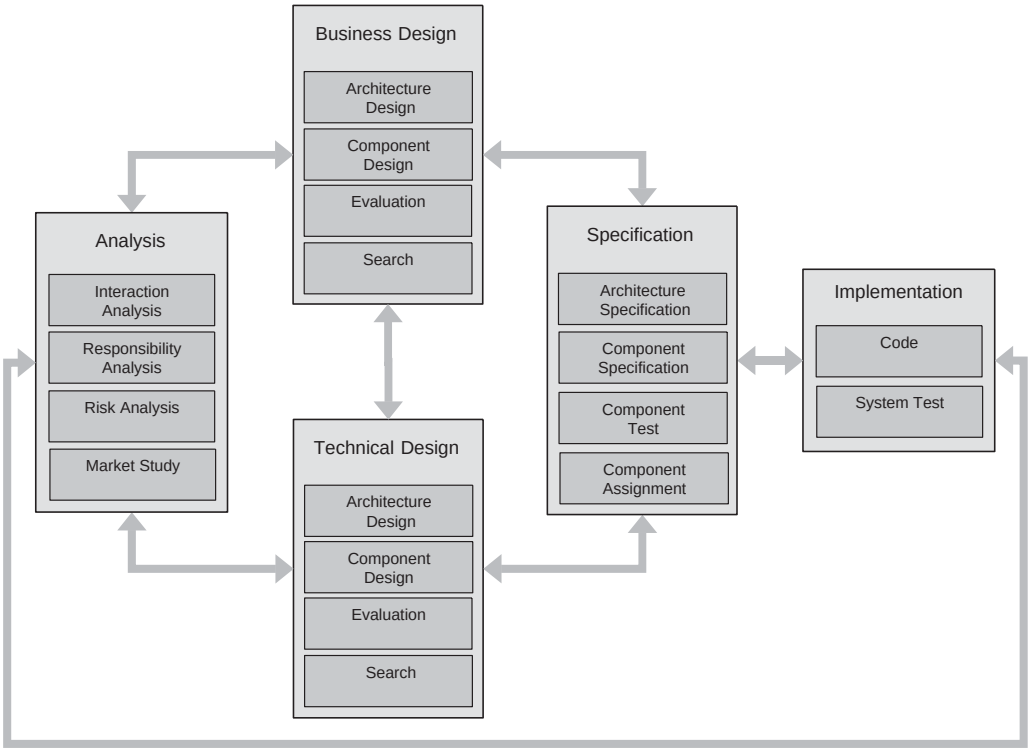


Abbildung 6.1: Eine Ergebnisstruktur (aus [BRSV98c])

Eine hierarchische Ergebnisstruktur ist in Abbildung 6.1 dargestellt. Zu Beginn eines Projekts kann eine solche Vorgabe entsprechend der Projektsituation angepaßt werden. Hierbei fallen unter Umständen bestimmte Dokumente weg oder es werden neue Dokumentklassen eingeführt.

- Zwischen den einzelnen Dokumententypen bestehen *Beziehungen*, aus denen konkrete Konsistenzkriterien abgeleitet werden. Im Verlauf der Entwicklung helfen diese Aussagen bei der Beurteilung der Korrektheit der entstehenden Artefakte und sind daher kontinuierlich zu überprüfen. Ein Beispiel für eine solche Beziehung ist die Verfeinerung zwischen Dokumententypen, nach der ein verfeinertes Dokument detailliertere Informationen als das andere Dokument enthält. Beziehungen sind in Abbildung 6.1 als einfache Doppelpfeile repräsentiert.

- Eine Reihe von *Prozeßmustern* repräsentieren die übergreifende Strategie bei der Entwicklung und werden in Form sogenannter Prozeßmusterkataloge zusammengefaßt und dokumentiert. Ein einzelnes Prozeßmuster zeigt dabei eine Reihe von Arbeitsschritten auf, die in einer bestimmten Projektsituation zu einem bestimmten Ziel führen. Die Dokumentation eines Prozeßmusters erfolgt — ähnlich zu den Designmustern aus [GHJV95] — anhand einer Schablone, die zumindest aus den folgenden Punkten besteht:

  Zusammenfassung: Ein kurzer Überblick über das Prozeßmuster mit den Kernaussagen der nachfolgenden Punkte.

  Kontext: Eine Beschreibung der Projektsituation, in der das Prozeßmuster angewendet werden soll (auch als *Kontext* bezeichnet). Die Projektsituation umfaßt dabei den internen Zustand der Entwicklung sowie externe Einflußgrößen, die beispielsweise durch den Kunden und durch Zulieferer geprägt sind. Die (angepaßte) Ergebnisstruktur mit teilweise bereits fertiggestellten Dokumenten charakterisiert hierbei den internen Zustand, also insbesondere den Fortschritt im Entwicklungsprozeß.

  Problem: Aus der augenblicklichen Projektsituation allein ist nicht immer eine konkrete Problemstellung und damit ein Handlungsbedarf ablesbar. Aus diesem Grund werden typische Probleme und Zielsetzungen beschrieben, die durch dieses Prozeßmuster adressiert werden. Grundlage der Problemstellung sind häufig Lücken in der Ergebnisstruktur aber ebenso externe Einflußfaktoren, wie beispielsweise ein erhöhter Kosten- und Termindruck oder veränderte Anforderungen von Seiten des Kunden.

  Lösung: Eine detaillierte Beschreibung eines bewährten Lösungsansatzes beinhaltet konkrete Arbeitsschritte und erfaßt das Resultat als veränderten internen Zustand und Auswirkungen auf das Projektumfeld. Das Ergebnis ist somit insbesondere eine Veränderung der durch die Ergebnisstruktur vorgegebenen Dokumente. Handlungsanweisungen werden anhand unterschiedlicher Rollen der an der Entwicklung beteiligten Personen festgehalten und bei Bedarf priorisiert.

  Beispiel: Um die pragmatische Anwendung des Prozeßmusters zu erleichtern, wird durch ein Beispiel der Zusammenhang zwischen Kontext, Problem und vorgeschlagener Strategie erläutert.

  Dieser Aufbau erfaßt die wichtigsten Informationen, die für die Anwendung eines Prozeßmusters erforderlich sind. In [BRSV98a] findet sich eine detailliertere Auflistung mit zusätzlichen Informationen, unter anderem mit einer graphischen Darstellung der für das Muster relevanten Bereiche der Ergebnisstruktur.

- Eine Reihe von *Rollen* beschreibt die Aufgabenbereiche der an der Entwicklung beteiligten Personen. Auf diese Weise können die durchzuführenden Aktivitäten genauer aufgeschlüsselt werden. So bezeichnet die Rolle „Systemarchitekt“ eine Person, die sich bereits frühzeitig mit Organisation und Struktur einer Anwendung auseinandersetzt und anhand vorgegebener Qualitätskriterien die Softwarearchitektur konzipiert und verfeinert. Die Aufgabe des „Projektleiters“ besteht unter anderem in der Konzeption des Entwicklungsprozesses sowie in der Kontrolle der Abläufe hinsichtlich Kosten, Zeit und Qualität. Gerade im Hinblick auf die besonderen Anforderungen im Umfeld komponentenbasierter Softwareentwicklung sind zusätzliche Rollen notwendig: So ist ein „Komponentenentwickler“ für die Bereitsstellung von Komponenten verantwortlich, sei es über den Einkauf von dritter Seite oder über eine Eigenentwicklung. Details über dieses Rollenverständnis finden sich in unseren Arbeiten über Componentware [BRS98, BRSV98a, BRSV98b, BRSV99].

Ein Entwicklungsprozeß basierend auf Prozeßmustern läuft nach folgendem Schema ab: ausgehend von der momentanen Projektsituation werden unterschiedliche Problemstellungen identifiziert und priorisiert. Zu jedem Problem existiert ein, im Idealfall sogar mehrere Prozeßmuster, die diese Problematik adressieren und ein geeignetes Vorgehen vorschlagen. Die Auswahl der besten Strategie und der damit verbundenen Arbeitsschritte erfolgt anhand der gegebenen Informationen über interne und externe Auswirkungen. Auf diese Weise beinhaltet der Entwicklungsprozeß eine kontinuierliche Auseinandersetzung mit der Soll- und Ist-Situation im Projekt. Dabei ist es durchaus möglich, die Anwendung von Prozeßmustern in sich hierarchisch zu gestalten, wobei längerfristige Strategien die Zielsetzung für kurzfristige Strategien vorgeben oder zumindest beeinflussen.

Die hohe Flexibilität eines Vorgehensmodells basierend auf Prozeßmustern zeigt sich einerseits in der dynamischen, kontextbezogenen Auswahl der nächsten Schritte, aber auch in der Tatsache, daß ein gegebener Prozeßmusterkatalog jederzeit um neue Prozeßmuster erweitert werden kann. So findet sich in [BRSV98a] bereits eine erste Auswahl an praktikablen Prozeßmustern für die komponentenorientierte Softwareentwicklung, die nun im folgenden Abschnitt um ausgesuchte Prozeßmuster für die Entwicklung und Anwendung von Komponentenframeworks erweitert wird.

6.3 Auswahl einiger Prozeßmuster

Im folgenden werden einige ausgesuchte Prozeßmuster vorgestellt, um die Konzepte aus dem vergangenen Abschnitt zu verdeutlichen. Die aufgeführten Beispiele betreffen Strategien zur Umsetzung der Richtlinien aus Abschnitt 6.1.3 und repräsentieren auf diese Weise „best-practices“ für die Erstellung und Anwendung von Komponentenframeworks.

6.3.1 Entkopplung von Rollen

Zusammenfassung: *Kollaborationen zwischen den Rollen eines Frameworks werden so umgestaltet, daß die einzelnen Rollen ein Minimum an Wissen über das Zusammenspiel tragen. Das Wissen über die konkreten Abläufe wird stattdessen in das Komponentenframework verlagert.*

Kategorie: Entwicklung Komponentenframework

Kontext: Basierend auf der fachlichen Modellierung eines Anwendungsbereichs besteht ein Verständnis darüber, welche Entitäten für eine bestimmte Funktionalität zusammenwirken. Diese Kollaborationen sind spezifiziert durch den Austausch von Nachrichten zwischen den Entitäten und unter Umständen repäsentiert durch ein oder mehrere Sequenzdiagramme.

Problem: Ganz in der Tradition der Objektorientierung werden Kollaborationen häufig so modelliert, daß der Kontrollfluß über alle beteiligten Entitäten verteilt ist. Das Problem hierbei ist, daß die entstehenden Rollen der einzelnen Entitäten unter Umständen spezifisch für die jeweilige Kollaboration sind. So ist eine Rolle `Patient` im Umfeld eines Krankenhauses möglicherweise eine andere als eine Rolle `Patient` in der Arztpraxis aufgrund unterschiedlicher Abläufe.

Je mehr Annahmen innerhalb einer Rollendefinition getroffen werden über das Umfeld der Komponente und das Zusammenspiel mit anderen Komponenten, desto geringer ist die Wahrscheinlichkeit einer Wiederverwendung dieser Rolle in einem anderen Kontext.

Lösung: Durch das Einbringen des Mediators eines Frameworks in die bestehende Kollaboration werden in einem ersten Schritt die einzelnen Rollen voneinander entkoppelt. Entsprechend dem *Mediator*-Pattern aus [GHJV95] agieren Rollen nur noch mittelbar miteinander. Nachrichten werden nur noch an den „Vermittler“ versendet, der für die richtige Weiterleitung sorgt.

Zudem wird das Wissen um die korrekten Abläufe im Rahmen einer Kollaboration dem Komponentenframework übertragen. Auf diese Weise können Rollen von der jeweiligen Kollaboration weitgehend entkoppelt werden. In der Folge ergeben sich „schlankere“ Rollenspezifikationen, da eine Rolle nur noch entitätenspezifisches Verhalten umfaßt (vgl. „Hollywood-Prinzip“ [Szy97]).

Beispiel: Auf der linken Seite von Abbildung 6.2 ist ein einfacher Ablauf im Umfeld einer Arztpraxis aufgezeichnet. Fällt in der Praxis ein Gerät aus, so wird der `Arzt` davon in Kenntnis gesetzt und informiert im Weiteren alle `Patienten`, deren Behandlung von diesem Gerät abhängig ist. Die Rolle des Arztes ist somit zugeschnitten auf die Abläufe in der Praxis und die direkte Kommunikation mit den Patienten. Obwohl das durchaus wünschenswert ist, kann die Rolle des Arztes nicht im Krankenhaus verwendet werden, wo grundlegend andere Konstellationen gegeben sind.

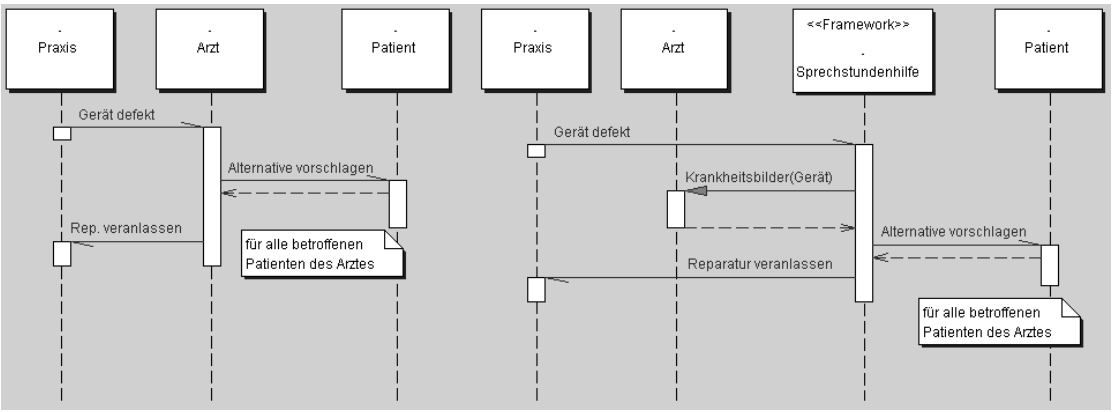


Abbildung 6.2: Entkopplung von Rollen

Eine geeignete Lösung ist auf der rechten Seite von Abbildung 6.2 dargestellt. Die Verantwortung für die korrekte Interaktion der Beteiligten wird in eine (gedachte) Rolle des Frameworks ausgelagert, die hier zur Veranschaulichung `Sprechstundenhilfe` benannt wurde. Der gesamte Ablauf wird von dieser Rolle koordiniert und der Arzt muß lediglich bestimmen, welche Krankheitsbilder durch den Ausfall des Geräts nicht mehr behandelt werden können. Die Auswahl der jeweiligen Patienten anhand dieser Informationen wird von der Sprechstundenhilfe übernommen. Auf diese Weise ergibt sich beispielsweise eine klare, „entlastete“ Rolle `Arzt`, die auch in anderen Zusammenhängen (z.B. für ein Krankenhaus) sinnvoll ist.

6.3.2 Zerlegung von Kollaborationen

Zusammenfassung: *Durch die hierarchische Zerlegung einer Kollaboration werden die entstehenden Abläufe kleiner, handlicher und wiederverwendbar.*

Kategorie: Entwicklung Komponentenframework

Kontext: Die Grundlagen für die Erstellung eines Komponentenframeworks sind festgelegt. So besteht Einigkeit über die Menge der beteiligten Entitäten, ihre jeweiligen Rollen und das konkrete Zusammenspiel.

Problem: Die vorgesehene Kollaboration ist zu umfangreich oder zu komplex. Dies behindert gleichermaßen Verständnis und Wiederverwendbarkeit. Unter Umständen enthält die vorgesehene Kollaboration zudem Abläufe, die nicht direkt den Kernkompetenzen der Entwickler entsprechen.

Lösung: Interaktionen können auf unterschiedliche Art und Weise aus kleineren Modulen aufgebaut werden (vgl. [Krü99]). Obwohl die in dieser Arbeit vorgestellte Konzeption von Komponentenframeworks keine direkte hierarchische Zergliederung erlaubt, kann eine Kollaboration auch hier in mehrere Komponentenframeworks aufgeteilt werden, die über ihre Rollen an gemeinsamen Fachkomponenten zusammenwirken. Damit verbunden ist die Herauslösung eines eigenständigen Fachmodells aus der vorhandenen Anwendungssicht. Dies erscheint umso lohnenswerter, je höher die Wiederverwendbarkeit in anderen Umgebungen einzuschätzen ist.

Entsprechend den Richtlinien aus Abschnitt 6.1.3 ist bei der Zerlegung darauf zu achten, daß die Kommunikation und hiermit die Abhängigkeiten zwischen den einzelnen Frameworks minimiert werden. Weiteren Einfluß auf die Unterteilung nehmen eventuell bereits vorliegende Komponentenframeworks, die geeignet eingebunden werden sollen.

Ein entscheidender Vorteil der Zerlegung eines vorhandenen Komponentenframeworks besteht in der Möglichkeit, daß sich mit der Aufteilung der Abläufe auch die einzelnen Verantwortlichkeiten der involvierten Fachkomponenten aufteilen. Dadurch vereinfachen sich die einzelnen Kollaborationen und die jeweiligen Anforderungen an die Komponenten.

Beispiel: Eine rein schematische Vorstellung von der Aufteilung eines Frameworks in weitere, feinere Bestandteile vermittelt Abbildung 6.3. Auf der linken Seite ist eine Kollaboration zwischen drei Entitäten dargestellt, die sich entsprechend den Rollen A_1, A_2 und A_3 verhalten. Direkt daneben ist derselbe Interaktion gezeigt, wobei ein Teil des Ablaufs in einer eigenständigen Kollaboration gekapselt ist. Hierbei wurden die diversen Rollen auf das im Modul relevante Verhalten konzentriert (B_1, B_2

und B_3). Ähnlich entfallen bestimmte Funktionalitäten, die in der äußeren Kollaboration nötig waren, wodurch die A-Rollen ebenfalls vereinfacht werden können. Mehr noch, die Rolle A_3 spielt überhaupt nur in der referenzierten Kollaboration eine Rolle und muß deswegen im umgebenden Zusammenspiel nicht berücksichtigt werden (wohl aber im Gesamtframework).

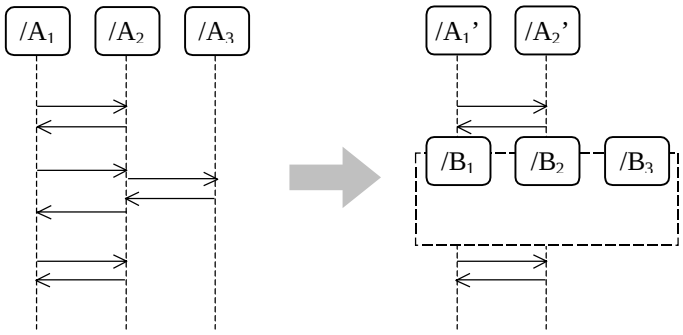


Abbildung 6.3: Aufteilung einer Kollaboration

Die Kopplung zwischen den beiden Komponentenframeworks zeigt sich an einer internen Abhängigkeit beispielsweise zwischen der Rolle A'_1 und B_1. Jede Fachkomponente, die sich entsprechend den Vorgaben von A'_1 verhält, muß ebenfalls die Rolle B_1 ausüben können.

6.3.3 Entwurf von Rollenkomponenten

Zusammenfassung: *Die geeignete Aufteilung des Rollenverhaltens auf Rollenkomponenten und Kernkomponente schafft eine gemeinsame Basis, die im Idealfall in unveränderter Form auch für zusätzliches Rollenverhalten geeignet ist und auf diese Weise die Integration vereinfacht.*

Kategorie: Entwicklung Komponentenframework

Kontext: Das Design eines Komponentenframeworks ist soweit vorangeschritten, daß für die beteiligten Entitäten festgelegt ist, welche Rollen sie spielen sollen und auf welche Schnittstellendefinitionen sie sich dabei berufen. In einem weiteren Schritt gilt es nun, für die einzelnen Rollen geeignete Rollenkomponenten zu spezifizieren und zu realisieren.

Problem: Die Aufteilung des Gesamtverhaltens einer Komponente auf Rollenkomponenten und Kernkomponente beeinflusst entscheidend den Aufwand bei einer späteren Instantiierung und bei einer Integration mit anderen Komponentenframeworks.

Rollenkomponenten sind nicht nur Adapter für das Verhalten der Kernkomponente, sondern können zudem rollenspezifisches Verhalten kapseln und die Kernkomponente somit entlasten (vgl. Abschnitt 3.6).

Eine Kernkomponente ist idealerweise der „größte, gemeinsame Teiler“ aller Rollenkomponenten. Da die Menge der Rollen einer Komponente jedoch a priori nicht bekannt ist, könnte ein Entwickler bestrebt sein, eine sehr umfassende Kernkomponente zu konzipieren. Dadurch wären zwar zukünfigte Integrationsvorhaben erleichtert, jedoch die Anwendung des Frameworks aufgrund der komplexen Kernkomponente erschwert. Da dieser Bestandteil vom Anwender eines Frameworks geliefert wird, sinkt der eigentliche Nutzen der mitgelieferten Rollenkomponenten. Umgekehrt führen zu schwergewichtige Rollenkomponenten unter Umständen dazu, daß ähnliche Verhaltensabläufe mehrfach realisiert werden, da sie in der Menge der Rollenkomponenten nicht wiederverwendet werden können.

Lösung: Eine grobe Richtlinie besteht darin, jeglichen Zustand der Komponente im Kern zu kapseln, damit alle Rollenkomponenten darauf zugreifen und sich darüber synchronisieren können. Bei einem solchen Aufbau sind Rollenkomponenten selbst zustandslos und werden zu einfachen Adaptern reduziert. Eine verfeinerte Aufteilung erfordert die Auseinandersetzung damit, ob ein Teil des zu spezifizierenden Verhaltens

- rollenspezifisch ist. Hierbei sind hauptsächlich interaktionsbezogene Informationen betroffen, wie beispielsweise der jeweilige Kommunikationspartner oder Protokollspezifika. Jedoch sind auch eher technische Belange in vielen Fällen geeignet für die Kapselung in Rollenkomponenten. So zum Beispiel im Fall einer Rolle, die für jede erhaltene Anfrage sofort eine Quittung zurücksenden muß.
- entitätenspezifisch ist. Hierunter fällt jener Teil des Verhaltens, der zwar für die Erfüllung einer Rolle notwendig ist, sich jedoch dieser nicht eindeutig zuordnen läßt. Ein eindeutiges Kriterium sind Abläufe, deren Ausführung resp. Ergebnisse für andere Rollen gleichfalls interessant sind.

Entsprechend dem Zwiebelschalenmodell kann eine Rolle auch durch mehrere, hintereinandergeschaltete Rollenkomponenten realisiert werden. Eine solche Aufteilung erhöht die Wahrscheinlichkeit, für unterschiedliche Rollen eine gemeinsame Basis zu finden und trägt somit zur Wiederverwendbarkeit des Komponentenframeworks bei.

Beispiel: Eine weltweit eingesetzte Brokerage-Anwendung erlaubt ihren Anwendern das zeitnahe Kaufen und Verkaufen von Wertpapieren auf den Aktienmärkten der

Welt. Das System basiert unter anderem auf einer Datum-Komponente, die unter anderem vermerkt, ab wann ein bestimmter Kundenauftrag ausgeführt werden soll. Hierfür ist eine Rolle `Datum` über zwei Schnittstellen `get/set_location` und `get/set_date` definiert (siehe Darstellung in Abbildung 6.4). Über die erste Schnittstelle wird die Komponente mit Informationen darüber versorgt, in welcher Zeitzone sich der Anwender befindet, während die zweite Schnittstelle dazu dient, konkrete Werte hinsichtlich Datum und Uhrzeit abzuspeichern bzw. auszulesen.

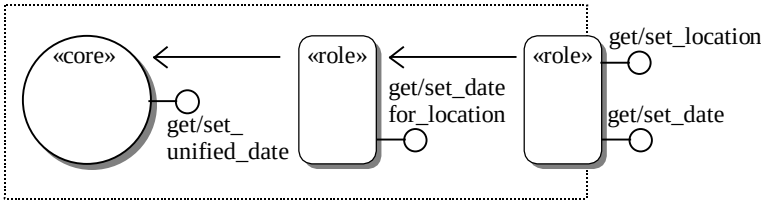


Abbildung 6.4: Realisierung der Rolle `Datum`

Neben der Rolle `Datum`, die relativ anwenderbezogen ist, sind noch weitere Rollen denkbar — nicht zuletzt für den direkten Zugriff des Systems für eine korrekte Auftragsausführung. Aus diesem Grund wurde die Realisierung der Rolle `Datum` auf eine Kernkomponente und zwei Rollenkomponenten aufgeteilt[2]: In der Kernkomponente wird der jeweilige Zeitpunkt in einem systemspezifischen, normalisierten Format gespeichert. Eine nachfolgende Rollenkomponente kapselt das Wissen um eine korrekte Umrechnung des Zeitformats für andere Zeitzonen. Die darauf aufbauende Rollenkomponente unterscheidet sich nur insofern, als daß die gewünschte Zeitzone einmal angegeben wird und für alle weiteren Aufrufe das Verhalten der Rollenkomponente bestimmt.

Das spezifizierte Verhalten von `Datum` wurde also soweit zerlegt, daß rollenspezifisches Verhalten in der entsprechenden Rollenkomponente und gemeinsames Verhalten in der Kernkomponente gekapselt wird. Die unterschiedlichen „Schalen“ des Aufbaus bieten eine Flexibilität, die die spätere Integration zusätzlicher Rollen begünstigt.

[2] Abbildung 6.4 zeigt eine vereinfachte Darstellung, da sowohl die Gegenspieler der Schnittstellen als auch die entsprechende Verschaltung ausgelassen wurden.

6.4 Zusammenfassung

In diesem Kapitel wurde die Bedeutung eines wohldurchdachten, flexiblen Vorgehensmodells für die Entwicklung und Anwendung von Komponentenframeworks hervorgehoben. Ausgehend von einer Identifikation konkreter Anforderungen an einen Entwicklungsprozeß mit Komponenten im allgemeinen und Komponentenframeworks im speziellen wurde ein Ansatz vorgestellt, der eine klar definierte Ergebnisstruktur als Leitfaden und einen Katalog von Prozeßmustern als konkrete Repräsentation von „best practices“ vorsieht. Prozeßmuster als Bausteine eines flexiblen Vorgehensmodells sind im Bereich des Software-Engineering ein vielbeachterer Ansatz und finden sich in ähnlicher Form in anderen Arbeiten wieder (z.B. in Catalysis [DW98]).

In diesem Kapitel wurde nur eine knappe Einführung in die einhergehenden Konzepte geboten, die viele weitere Aspekte eines Entwicklungsprozesses vernachlässigt. Dazu gehören beispielsweise Aussagen über zu verwendende Werkzeuge, Metriken und Techniken. Eine umfassendere Beschreibung von Prozeßmustern, die detailliert auf Struktur, Organisation und Abläufe eingeht und inbesondere auch eine geeignete Sprache für die Organisation und das Zusammenspiel von Prozeßmustern und ihren implizierten Abläufen bietet, findet sich beispielsweise in [BRSV98a, BRSV99].

Wichtig ist jedoch die hohe Flexibilität des vorgestellten Ansatzes, die es auf der einen Seite erlaubt, bewährte Praktiken aus anderen Vorgehensmodellen (Wasserfallmodell, Spiralmodell, etc.) abzubilden und auf der anderen Seite die nahtlose Erweiterung hinsichtlich der Entwicklung und Anwendung von Komponentenframeworks ermöglicht. Hierfür wurden einige ausgewählte Prozeßmuster vorgestellt, die insbesondere eine Ausgestaltung eines Komponentenframeworks mit dem Ziel einer verbesserten Wiederverwendbarkeit zum Ziel haben. Auf ganz ähnliche Weise können weitere Erkenntnisse der vergangenen Kapitel, beispielsweise hinsichtlich der unterschiedlichen Vorgehen bei der Integration in Form von Prozeßmustern repräsentiert werden.

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit wurde das Konzept der Komponentenframeworks als vielversprechender Ansatz für die Bewältigung der Komplexität bei der Entwicklung moderner Anwendungssysteme vorgestellt. Komponentenframeworks kombinieren die Wiederverwendung von Entwurfswissen (wie z.B. Designmuster [GHJV95]) mit der Wiederverwendung konkreter Bestandteile der Implementierung (wie z.B. Komponenten [Gri98]). In dieser Eigenschaft sind Komponentenframeworks vergleichbar mit objektorientierten Frameworks, weisen im Vergleich jedoch eine Reihe von Vorteilen auf. So vermeidet die Erweiterung durch Delegation die Probleme der Codevererbung (siehe [Szy97]), die in objektorientierten Frameworks gebräuchlich ist. Zudem ergibt sich auf diese Weise eine Flexibilität, die auch die dynamische Veränderung eines Komponentenframeworks ermöglicht. Vor allem sind Komponentenframeworks „integrationsoffen“ und damit auf ein Zusammenwirken untereinander vorbereitet. Hierzu wurden Mechanismen vorgestellt, die es erlauben, Abhängigkeiten zwischen Komponentenframeworks auf geeignete Weise zu spezifizieren. Die zugrundeliegenden Konzepte ermöglichen eine Softwareentwicklung in Form einer Integration ausgewählter Komponentenframeworks und einer Instantiierung durch die Bereitstellung erforderlicher Fachkomponenten. Die damit verbundenen Ideen und Konzepte werden wie folgt zusammengefaßt:

- Fachmodelle repräsentierten das konzeptionelle Verständnis des Anwendungsbereichs in Form unterschiedlicher Rollen und ihrer Beziehungen untereinander. Dabei werden insbesondere Beziehungen zwischen den unterschiedlichen Rollen einer Entität erfaßt, wodurch einerseits eine feingranulare Aufteilung und andererseits die Spezifikation von Abhängigkeiten zwischen unterschiedlichen Fachmodellen ermöglicht wird.

- Die „Schnittstelle“ eines Komponentenframeworks orientiert sich an den Rollen ihres Fachmodells, wodurch sich das Verständnis der Zusammenhänge vereinfacht und damit der Lernaufwand reduziert wird. Der Anwender eines Komponentenframeworks stellt für den Einsatz eines Komponentenframeworks geeignete Fachkomponenten bereit, die die Ausübung der vorgesehenen Rollen übernehmen. Hierfür wird jeder Rolle eine Verhaltenspezifikation zugeordnet, die in Form eines Komponententyps definiert wird. Neben diesen Fachkomponenten kann ein Komponentenframework auf technische Komponenten zurückgreifen, die für den Anwender jedoch nicht sichtbar sind. Für die Implementierung einer Fachkomponente liefert ein Komponentenframework geeignete Rollenkomponenten, die rollenspezifisches Verhalten realisieren.

- Die Integration von Komponentenframeworks erfordert eine Abbildung der jeweiligen Konzepte und Beziehungen, wie sie durch die assoziierten Fachmodelle repräsentiert sind. Die Zusammenführung auf dieser abstrakten Ebene setzt sich fort in einer Verhaltenskomposition einzelner Rollen und schließlich in der Zusammenstellung der entsprechenden Kernkomponenten. Hierbei können unterschiedliche Konflikte auftreten, die entweder eine Integration oder zumindest die Verwendung der vorgesehenen Rollenkomponenten verhindern. In letzterem Fall kann die Entwicklung einer neuen Rollenkomponente die einzelnen Verhaltensannahmen auf korrekte Weise zusammenführen.

- Da die Implementierung eines Komponentenframeworks unmittelbar an seinem Fachmodell ausgerichtet ist, können Modelle bei der Integration nicht geeignet verändert werden, um unterschiedliche Abstraktionsniveaus miteinander zu vereinen. Für diesen Sonderfall wurde ein Mechanismus vorgestellt, der die redundante Repräsentation der jeweiligen Informationen ermöglicht. Die Konsistenz der beteiligten Fachkomponenten wird hierbei über einen Austausch der zugehörigen Kernkomponenten sichergestellt.

Die unterschiedlichen Konzepte und Mechanismen sorgen zwar für eine gesteigerte Wiederverwendbarkeit und einen reduzierten Integrationsaufwand, bedeuten andererseits jedoch auch eine erhöhte Komplexität in der Entwicklung einer Anwendung. Aus diesem Grund wurde in dieser Arbeit eine Methodik für die Entwicklung und Anwendung von Komponentenframeworks erarbeitet, die aus den folgenden Teilen aufgebaut ist:

- Das *formale Systemmodell* aus Kapitel 3 präzisiert die verwendeten Konzepte und ihre Beziehungen untereinander. Es ist mehrschichtig aufgebaut und zeigt auf diese Weise insbesondere die nahtlose Einbettung von Komponentenframeworks in allgemeine, komponentenbasierte Systeme. Obwohl an einigen Stellen Vereinfachungen vorgenommen wurden, konnten anhand des Systemmodells typische Probleme und

Konflikte bei der Integration von Fachmodellen und entsprechenden Realisierungen aufgezeigt und geeignete Lösungsansätze verdeutlicht werden.

- In Kapitel 4 wurde die *technische Umsetzung* der erarbeiteten Ergebnisse auf eine geeignete Komponentenplattform diskutiert und am Beispiel des CORBA Component Models vorgeführt. Diese Spezifikation zeigt sich in vielerlei Hinsicht als sehr ausgereift und einfach erweiterbar im Bezug auf Komponentenframeworks. Der Nachteil liegt in einem Mangel geeigneter Implementierungen dieser Komponentenplattform.

- Eine Reihe *graphischer Beschreibungstechniken* wurden in Kapitel 5 als Erweiterung des UML-Standards speziell hinsichtlich der Einführung von Rollen und darauf aufbauender Komponentenframeworks konzipiert und diskutiert. Für die Darstellung des strukturellen Aufbaus sowie der Signatur von Komponenten wurden Komponententyp- und Komponentendiagramme eingeführt, wobei die letzteren ebenfalls für die Repräsentation struktureller Veränderungen herangezogen werden. Sequenzdiagramme wurden hinsichtlich der unterschiedlichen Rollen, die eine Fachkomponente einnehmen kann, erweitert und bieten nun die Möglichkeit, auch interne Abläufe einer Komponente in geeigneter Form darzustellen.

- Schließlich bieten die Ergebnisse aus Kapitel 6 die Grundzüge eines flexiblen Vorgehensmodells, das für die Entwicklung und die Verwendung von Komponentenframeworks besonders geeignet ist. Der verwendete Ansatz basiert auf Prozeßmustern als Bausteine eines flexiblen Entwicklungsprozesses. Eine Auswahl an Prozeßmustern für die Entwicklung eines Komponentenframeworks repräsentiert „best practices“ und demonstriert die Anwendbarkeit.

Aufbauend auf den unterschiedlichen Diskussionen bieten sich vor allem die folgenden Wege für weiterführende Arbeiten an. Die Koordination zusammengeführter Komponentenframeworks über Rollen gemeinsamer Entitäten erfordert stets die Kenntnis der Rollen der jeweils anderen Komponentenframeworks. In einigen Fällen wäre es jedoch wünschenswert, Abhängigkeiten zu einem weiteren Komponentenframework zu definieren, ohne die dort benötigten Rollen kennen zu müssen. Auf der Ebene der fachlichen Modellierung entspräche dies der Definition von Abhängigkeiten zwischen Beziehungen, die in der Implementierung einen direkten Austausch zwischen den Komponentenframeworks (bzw. den Schnittstellen ihrer Frameworkkomponenten) erforderlich machen würde. In diesem Zusammenhang erscheint es auch sinnvoll, „Higher-Order-Frameworks“ als Organisationsstrukturen zu evaluieren, die ihrerseits Anknüpfpunkte für Komponentenframeworks bieten.

Weitere Arbeiten könnten sich mit der Verfeinerung des Zwiebelschalenmodells beschäftigen. Hierbei wäre es insbesondere interessant, die Abhängigkeiten zwischen Rollen-

und Kernkomponenten so zu spezifizieren, daß die Zusammenstellung durch die Infrastruktur zur Laufzeit automatisch vorgenommen werden kann. Weiterhin erscheint diese Realisierung einer Fachkomponente mit den unterschiedlichen Zwischenschichten und Portadaptern nicht zuletzt aufgrund der vielfältigen Kommunikationsflüsse nur für die prototypische Realisierung eines Anwendungssystems geeignet. Für die praxistaugliche Realisierung von Komponentenframeworks erscheint aus diesem Grund eine pragmatischere Umsetzung notwendig.

Auch die Anwendbarkeit des skizzierten Vorgehensmodells für die Entwicklung und Anwendung von Komponentenframeworks ist als weiterführende Arbeit denkbar. Dies erfordert eine genauere Untersuchung der eingehergehenden Konzepte anhand praxisorientierter Beispielentwicklungen.

Abbildungsverzeichnis

1.1 Integration von Komponentenframeworks 5
1.2 Bausteine der komponentenbasierten Softwareentwicklung 11

3.1 Modellintegration auf unterschiedlichen Entwicklungsstufen 24
3.2 Aufbau der Modellierung 25
3.3 Entitäten und Beziehungen 29
3.4 Zwei Rollen 31
3.5 Die „verfasst“-Assoziation 33
3.6 Abhängigkeiten zwischen Rollen 34
3.7 Zwei Arten der Zerlegung einer Rolle 36
3.8 Einreichung zur Konferenz: Ein Fachmodell 38
3.9 Generalisierung zwischen Rollen von Assoziationen 39
3.10 Fachmodelle und Integrationsmodelle 43
3.11 Abstraktionsstufen der fachlichen Modellierung 45
3.12 Fachmodellintegration auf der Entitätenebene 46
3.13 Vermeidung von Assoziationen zwischen Fachmodellen 49
3.14 Abstraktionsbeziehung zwischen Rollen 52
3.15 Fachmodelle in Schichten 53
3.16 Fachmodell für ein Repository 54

3.17 Fachmodell des Reviewprozesses 54
3.18 Integrierte Fachmodelle 55
3.19 Ein Komponentendiagramm 62
3.20 Mehrfache Inklusion einer Komponente 72
3.21 Komposition von Komponententypen 75
3.22 Kapselung eines gemeinsamen Zustands 78
3.23 Komposition zweier Blackbox-Komponententypen 80
3.24 Blackbox-Verhalten der Rolle `Verwaltungseinheit` 81
3.25 Blackbox-Verhalten der Rolle `Arbeit` 82
3.26 Erweiterte Komponententypen 83
3.27 Glassbox-Komponententyp zur Rolle `Verwaltungseinheit` 84
3.28 Glassbox-Komponententyp zur Rolle `Arbeit` 85
3.29 Erweiterung zwischen Greybox-Komponententypen 88
3.30 Abbildung fachlicher Abhängigkeiten 89
3.31 Schichtenarchitektur des Repository-Frameworks 91
3.32 Pipes and Filter-Architektur des Review-Frameworks 95
3.33 Schematische Darstellung: Zwiebelschalenmodell 100
3.34 Zusammenführung dreier Delegationsketten 102
3.35 Auflösung von Abstraktionsbeziehungen 104
3.36 Features und Rollen im Vergleich 106

4.1 Zusammenbau einer Komponente 111
4.2 Aufteilung in Objekte und Komponenten 118
4.3 Bidirektionaler Kanal und Biports 126
4.4 Artefakte einer Komponentenspezifikation (nach [WSO00]) 136
4.5 Klassendiagramm zur Infrastruktur 138
4.6 Zusammenspiel der unterschiedlichen Schichten 140

5.1 Erweiterungen der UML 152
5.2 Ein Komponentendiagramm 155
5.3 Ein Komponententypdiagramm 156
5.4 Sequenzdiagramm für die Erzeugung einer neuen Komponente 157
5.5 Zwei Varianten der Darstellung von Rollen 160
5.6 Abhängigkeiten zwischen Rollen 161
5.7 Interaktionsdiagramm mit Rollen 162
5.8 Aufbau einer Komponente mit Rollenkomponenten 163

6.1 Eine Ergebnisstruktur (aus [BRSV98c]) 178
6.2 Entkopplung von Rollen 182
6.3 Aufteilung einer Kollaboration 184
6.4 Realisierung der Rolle `Datum` 186

Literaturverzeichnis

[ABD+99] ANSORGE, Dirk ; BERGNER, Klaus ; DEIFEL, Bernd ; HAWLITZKY, Nicholas ; MAIER, Christoph ; PAECH, Barbara ; RAUSCH, Andreas ; SIHLING, Marc ; THURNER, Veronika ; VOGEL, Sascha: Managing Componentware Development - Software Reuse and the V-Modell Process. In: JARKE, Matthias (Hrsg.) ; OBERWEIS, Andreas (Hrsg.): *LNCS 1626, Advanced Information Systems Engineering*, Springer Verlag, June 1999, S. 134–148

[Adr93] ADRION, W. R.: Research Methodology in Software Engineering. In: WALTER TICHY AND NICO HABERMANN AND LUTZ PRECHELT (Hrsg.): *Summary of the Dagstuhl Workshop on Future Directions in Software Engineering* Bd. 18, ACM Software Engineering Notes, SIGSOFT, 1993, S. 36–37

[AF98] ALLEN, Paul ; FROST, Stuart: *Component-Based Development for Enterprise Systems*. Cambridge University Press, 1998 (Managing Object Technology Series)

[Arc97] BROY, Manfred (Hrsg.) ; DENERT, Ernst (Hrsg.) ; RENZEL, Klaus (Hrsg.) ; SCHMIDT, Monika (Hrsg.): Software Architectures and Design Patterns in Business Applications / Technische Universität München, Institut für Informatik. 1997 (TUM-I9746). – Forschungsbericht

[ART] *ARTWORK : Architectures and Techniques for Computer Aided Engineering Workbenches - Förderprojekt der Bayerischen Softwareoffensive*. `http://artwork.in.tum.de/`

[BCK98] BASS, Len ; CLEMENTS, Paul ; KAZMAN, Rick: *Software Architecture in Practice*. Addison Wesley Publishing Company, January 1998

[BDD+92] BROY, Manfred ; DEDERICH, Frank ; DENDORFER, Claus ; FUCHS, Max ; GRITZNER, Thomas ; WEBER, Rainer: The Design of Distributed Systems — an introduction to FOCUS / Technische Universität München. 1992 (TUM-I9202). – Forschungsbericht

[BJR+01] BERGNER, Klaus ; JACOBI, Carsten ; RAUSCH, Andreas ; SIHLING, Marc ; VILBIG, Alexander: Make-or-Buy von Softwarekomponenten. In: *OBJEKTSpektrum* 1 (2001), Januar/Februar

[BMR+96] BUSCHMANN, Frank ; MEUNIER, Régine ; ROHNERT, Hans ; SOMMERLAD, Peter ; STAL, Michael: *Pattern-Oriented Software Architecture : A System of Patterns*. John Wiley & Sons, New York, 1996

[Boe86] BOEHM, Barry W.: A Spiral Model of Software Development and Enhancement. In: *ACM Sigsoft Software Engineering Notes* 11 (1986), Nr. 4, S. 22–42

[Bos00] BOSCH, Jan: *Design and Use of Software Architectures*. Addison Wesley Publishing Company, 2000

[Bro98] BROY, Manfred: Compositional Refinement of Interactive Systems Modelled by Relations. In: DE ROEVER, William-Paul (Hrsg.) ; LANGMAACK, Hans (Hrsg.) ; PNUELI, Amir (Hrsg.): *Compositionality: The Significant Difference, LNCS 1536*, Springer Verlag, 1998, S. 130–149

[BRS97] BERGNER, Klaus ; RAUSCH, Andreas ; SIHLING, Marc: Using UML for Modeling a Distributed Java Application / Technische Universität München, Institut für Informatik. 1997 (TUM-I9735). – Forschungsbericht

[BRS98] BERGNER, Klaus ; RAUSCH, Andreas ; SIHLING, Marc: Componentware - The Big Picture. In: *Proceedings of the International Workshop on Component-Based Software Engineering (CBSE 98)*, 1998

[BRS+00] BERGNER, Klaus ; RAUSCH, Andreas ; SIHLING, Marc ; VILBIG, Alexander ; BROY, Manfred: A Formal Model for Componentware. In: SITARAMAN, Murali (Hrsg.) ; LEAVENS, Gary (Hrsg.): *Foundations of Component-Based Systems*, Cambridge University Press, 2000, S. 189–210

[BRSV98a] BERGNER, Klaus ; RAUSCH, Andreas ; SIHLING, Marc ; VILBIG, Alexander: A Componentware Methodology based on Process Patterns / Technische Universität München. 1998 (TUM-I9823). – Forschungsbericht

[BRSV98b] BERGNER, Klaus ; RAUSCH, Andreas ; SIHLING, Marc ; VILBIG, Alexander: An Integrated View On Componentware - Concepts, Description Techniques, and Development Process. In: LEE, Roger (Hrsg.): *Proceedings of the IASTED Conference on Software Engineering*, 1998, S. 77–82

[BRSV98c] BERGNER, Klaus ; RAUSCH, Andreas ; SIHLING, Marc ; VILBIG, Alexander: A Componentware Development Methodology based on Process Patterns. In: *PLOP'98 Proceedings of the 5th Annual Conference on the Pattern Languages of Programs*, 1998

[BRSV99] BERGNER, Klaus ; RAUSCH, Andreas ; SIHLING, Marc ; VILBIG, Alexander: Componentware - Methodology and Process. In: *Proceedings of the International Workshop on Component-Based Software Engineering (CBSE 99)*, 1999

[BRSV00] BERGNER, Klaus ; RAUSCH, Andreas ; SIHLING, Marc ; VILBIG, Alexander: Putting the Parts Together - Concepts, Description Techniques, and Development Process for Componentware. In: *HICSS 33, Proceedings of the Thirty-Third Annual Hawaii International Conference on System Sciences*, IEEE Computer Society, January 2000

[BS01] BROY, Manfred ; STØLEN, Ketil: *Specification and Development of Interactive Systems - FOCUS on Streams, Interfaces and Refinement, to appear*. 2001

[Bäu98] BÄUMER, Dirk: *Softwarearchitekturen für die rahmenwerkbasierte Konstruktion großer Anwendungssysteme*, Fachbereich Informatik, Universität Hamburg, Diss., 1998

[BVD01] BROSE, Gerald ; VOGEL, Andreas ; DUDDY, Keith: *Java Programming with CORBA — Advanced Techniques for Building Distributed Applications*. 3rd Edition. OMG Press, Wiley Computer Publishing, 2001

[CCM99] *CORBA Components — Joint Revised Submission*. `http://cgi.omg.org/cgi-bin/doc?orbos/99-02-05`. March 1999

[CE00] CZARNECKI, Krzysztof ; EISENECKER, Ulrich W.: *Generative Programming — Methods, Tools, and Applications*. Addison Wesley Publishing Company, May 2000

[CHOT99] CLARKE, Siobhán ; HARRISON, William ; OSSHER, Harold ; TARR, Peri: Separating Concerns throughout the Development Lifecycle. In: *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99*, 1999

[Com] *The ComponentSource Homepage — The Definitive Source of Software Components*. `http://www.componentsource.com/`

[Cum00] CUMMINS, Fred A. *A Definition and Justification of Role Objects*. `http://www.omg.org/cgi-bin/doc?bom/2000-05-01`. May 2000

[CXS] *DataAccessTechnologies: The Component-X Studio Homepage*. `http://www.enterprise-component.com/products/studio.htm`

[DMNS97] DEMEYER, Serge ; MEIJLER, Theo D. ; NIERSTRASZ, Oscar ; STEYAERT, Patrick: Design Guidelines for 'Tailorable' Frameworks. In: *Communications of the ACM* 40 (1997), October, Nr. 10, S. 60–64

[DMT99] DASHOFY, Eric ; MEDVIDOVIC, Nenad ; TAYLOR, Richard: Using Off-The-Shelf Middleware to Implement Connectors in Distributed Software Architectures. In: *Proceedings of the ICSE'99*, ACM Press, 1999

[DSB99] D'SOUZA, Desmond ; SANE, Aamod ; BIRCHENOUGH, Alan: First Class Extensibility for UML — Packaging of Profiles, Stereotypes, Patterns. In: FRANCE, Robert (Hrsg.) ; RUMPE, Bernhard (Hrsg.): *«UML»'99 — The Unified Modeling Language, Beyond the Standard, LNCS 1723*, Springer Verlag, 1999, S. 265–277

[DW98] D'SOUZA, Desmond F. ; WILLS, Alan C.: *Objects, Components, and Frameworks with UML - The Catalysis Approach*. Addison Wesley Publishing Company, 1998

[DW99] DRÖSCHEL, Wolfgang ; WIEMERS, Manuela: *Das V-Modell 97*. Oldenburg Verlag, 1999

[EJB] *SUN Microsystems: Enterprise JavaBeans Homepage*. `http://java.sun.com/products/ejb/`

[FHLS97] FROEHLICH, Garry ; HOOVER, James ; LIU, Ling ; SORENSON, Paul: Hooking into Object-Oriented Application Frameworks. In: *Proceedings of the 19th Int. Conf. on Software Engineering (ICSE '97)*, ACM Press, 1997, S. 491–501

[FPR00] FONTOURA, Marcus ; PREE, Wolfgang ; RUMPE, Bernhard: UML-F: A Modeling Language for Object-Oriented Frameworks. In: BERTINO, Elisa (Hrsg.): *Proceedings of ECOOP 2000 Object-Oriented Programming Conference, LNCS 1850*, Springer Verlag, 2000, S. 63–83

[Fri95] FRICK, A.: *Der Software-Entwicklungsprozeß — Ganzheitliche Sicht*. Carl Hanser Verlag München Wien, 1995

[FS97] FOWLER, Martin ; SCOTT, Kendall: *UML distilled, Applying the Standard Object Modeling Language*. Addison Wesley Publishing Company, August 1997

[FSJ99] FAYAD, Mohamed E. ; SCHMID, Douglas C. ; JOHNSON, Ralph E.: *Building Application Frameworks — Object-Oriented Foundations of Framework Design*. Wiley Computer Publishing, 1999

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Publishing Company, 1995

[Gra99] GRAW, Günter: Specification of behaviour in component frameworks. In: BOSCH, Jan (Hrsg.) ; SZYPERSKI, Clemens (Hrsg.) ; WECK, Wolfgang (Hrsg.): *Proceedings of the Fourth International Workshop on Component-Oriented Programming (WCOP '99)*, 1999, S. 38–42

[Gri98] GRIFFEL, Frank: *Componentware - Konzepte und Techniken eines Softwareparadigmas*. dpunkt Verlag, 1998

[GS94] GARLAN, David ; SHAW, Mary: An Introduction to Software Architecture / Carnegie Mellon University, Pittsburgh. 1994 (CMU-CS-94-166). – Forschungsbericht

[GW00] GENILLOUD, Guy ; WEGMANN, Alain: Role is an <X>: a Foundation for the Concept of Role / EPFL-DSC Lausanne. 2000 (DSC/2000/023). – Forschungsbericht

[HHG90] HELM, Richard ; HOLLAND, Ian M. ; GANGOPADHYAY, Dipayan: Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In: *Proceedings of the ECOOP/OOPSLA'90*, ACM Press, October 1990, S. 169–180

[HNS99] HOFMEISTER, Christine ; NORD, Robert ; SONI, Dilip: Describing software architecture with UML. In: DONOHOE, Patrick (Hrsg.): *Proceedings of Working IFIP Conference on Software Architecture*, Kluwer Academic Publishers, February 1999, S. 145–160

[HO93] HARRISON, William ; OSSHER, Harold: Subject-oriented programming (a critique of pure objects). In: *Proceedings of the OOPSLA '93*, ACM Press, September 1993, S. 411–428

[HS93] HØYDALSVIK, Geir M. ; SINDRE, Guttorm: On the Purpose of Object-Oriented Analysis. In: *Proceedings of the OOPSLA'93*, 1993, S. 240–255

[HS00] HERZUM, Peter ; SIMS, Oliver: *Business Component Factory - A Comprehensive Overview of Component-Based Development for the Enterprise*. OMG Press - Wiley Computer Publishing, 2000

[JGJ97] JACOBSON, Ivar ; GRISS, Martin ; JONSSON, Patrik: *Software Reuse: Architecture Process and Organization for Business Success*. Addison Wesley Publishing Company, June 1997

[Joh92] JOHNSON, Ralph E.: Documenting Frameworks Using Patterns. In: *Proceedings of the OOPSLA'92*, ACM Press, 1992, S. 63–76

[KLM+97] KICZALES, Gregor ; LAMPING, John ; MENDHEKAR, Anurag ; MAEDA, Chris ; LOPES, Christina V. ; LOINGTIER, Jean-Marc ; IRWIN, John: Aspect-Oriented Programming. In: *Proceedings of the ECOOP '97*, Springer Verlag, 1997

[Krü99] KRÜGER, Ingolf: Towards the Methodical Usage of Message Sequence Charts. In: SPIES, Katharina (Hrsg.) ; SCHÄTZ, Bernhard (Hrsg.): *Formale Beschreibungstechniken für verteilte Systeme - 9. GI/ITG Fachtagung*, Herbert Utz Verlag, Juni 1999

[KRB96] KLEIN, Cornel ; RUMPE, Bernhard ; BROY, Manfred: A stream-based mathematical model for distributed information processing systems: The SysLab system model. In: NAIJM, Elie (Hrsg.) ; STEFANI, Jean-Bernard (Hrsg.): *Formal Methods for Open Object-based Distributed Systems (FMOODS'96)*, ENST France Telecom, 1996, S. 323–338

[KRSW01] KLEIN, Cornel ; RAUSCH, Andreas ; SIHLING, Marc ; WEN, Zhaojun: Extension of the Unified Modeling Language for Mobile Agents. In: SIAU, Keng (Hrsg.) ; HALPIN, Terry (Hrsg.): *Unified Modeling Language: Systems Analysis, Design and Development Issues*, Idea Group Publishing, 2001

[Kru99] KRUCHTEN, Philippe: *The Rational Unified Process - an Introduction*. Addison Wesley Publishing Company, 1999

[Lew98] LEWANDOWSKI, Scott M.: Frameworks for Component-Based Client / Server Computing. In: *ACM Computing Surveys* 30 (1998), Nr. 1

[LKA+95] LUCKHAM, David C. ; KENNEY, John J. ; AUGUSTIN, Larry M. ; VERA, James ; BRYAN, Doug ; MANN, Walter: Specification and Analysis of System Architecture Using Rapide. In: *IEEE Transactions on Software Engineering* 21 (1995), Nr. 4, S. 336–355

[LW94] LISKOV, Barbara H. ; WING, Jeannette M.: A Behavioral Notion of Subtyping. In: *ACM Transactions on Programming Languages and Systems* 16 (1994), Nr. 6, S. 1811–1841

[Mat00] MATTSSON, Michael: *Evolution and Composition of Object-Oriented Frameworks*, University of Karlskrona/Ronneby, Department of Software Engineering and Computer Science, Diss., 2000

[MBF00] MATTSSON, Michael ; BOSCH, Jan ; FAYAD, Mohamed E.: Framework Integration - Problems, Causes, Solutions. In: *Communications of the ACM* 42 (2000), Nr. 10

[McI68] MCILROY, Doug: Mass Produced Software Components. In: NAUR, Peter (Hrsg.) ; RANDELL, Brian (Hrsg.): *Proceedings of the NATO Software Engineering Conference*, 1968, S. 138–155

[Mey92] MEYER, Bertrand: Applying design by contract. In: *Computer* 25 (1992), October, Nr. 10, S. 40–51

[Mey97] MEYER, Bertrand: *Object-Oriented Software Construction*. 2nd Edition. Prentice Hall, 1997

[MH01] MEZINI, Mira ; HAUPT, Michael: Neue Programmierparadigmen: Integrationsorientierte Programmierung. In: *OBJEKTSpektrum* 2 (2001), März/April, S. 48–54

[ML98] MEZINI, Mira ; LIEBERHERR, Karl: Adaptive plug-and-play components for evolutionary software development. In: *Proceedings of the OOPSLA '98*, ACM Press, October 1998, S. 97–116

[MN96] MOSER, Simon ; NIERSTRASZ, Oscar: The Effect of Object-Oriented Frameworks on Developer Productivity. In: *Computer* 29 (1996), September, Nr. 9, S. 45–51

[OH98] ORFALI, Robert ; HARKEY, Dan: *Client/Server Programming with Java and CORBA*. 2nd Edition. John Wiley & Sons, 1998

[OPE] *The OPENCCM Platform*. `http://corbaweb.lifl.fr/OpenCCM/`

[Pre97a] PREE, Wolfgang: *Komponentenbasierte Softwareentwicklung mit Frameworks*. dpunkt Verlag, 1997

[Pre97b] PREHOFER, Christian: Feature-Oriented Programming: A Fresh Look at Objects. In: AKSIT, Mehmet (Hrsg.) ; MATSUOKA, Satoshi (Hrsg.): *Proceedings of the ECOOP'97: Object-Oriented Programming, LNCS 1241*, Springer Verlag, 1997

[Rat] *Rational Software Corporation: The Rational Rose Homepage*. `http://www.rational.com/products/rose/index.jsp`

[Rie00] RIEHLE, Dirk: *Framework Design — A Role Modeling Approach*, Swiss Federal Institute of Technology Zürich, Diss., 2000

[RJB98] RUMBAUGH, James ; JACOBSON, Ivar ; BOOCH, Grady: *The Unified Modeling Language Reference Manual*. Addison Wesley Publishing Company, December 1998

[Rom99] ROMAN, Ed: *Mastering Enterprise Java Beans and the Java2 Platform, Enterprise Edition*. John Wiley & Sons, Inc., 1999

[Roy70] ROYCE, Winston W.: Managing the Development of Large Software Systems: Concepts and Techniques. In: *WESCON Technical Papers, Western Electronic Show and Convention*, 1970

[Rum96] RUMPE, Bernhard: *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*, Technische Universität München, Diss., 1996

[RWL96] REENSKAUG, Trygve ; WOLD, Per ; LEHNE, Odd A.: *Working with Objects — The OOram Software Engineering Method*. Manning Publications Co., Greenwich, 1996

[Sam97] SAMETINGER, Johannes: *Software Engineering with Reusable Software Components*. Springer Verlag, 1997

[SAN] *IBM Software: The San Francisco Homepage*. `http://www.ibm.com/java/sanfrancisco/`

[SBF96] SPARKS, Steve ; BENNER, Kevin ; FARIS, Chris: Managing Object-Oriented Framework Reuse. In: *Computer* 29 (1996), September, Nr. 9, S. 52–61

[SDK+95] SHAW, Mary ; DELINE, Robert ; KLEIN, Daniel ; ROSS, Theodore ; YOUNG, David ; ZELESNIK, Gregory: Abstractions for Software Architectures and Tools to Support Them. In: *IEEE Transactions on Software Engineering* 21 (1995), Nr. 4, S. 356–372

[SG95] SHAW, Mary ; GARLAN, David: Formulations and Formalisms in Software Architecture. In: VAN LEEUWEN, Jan (Hrsg.): *Computer Science Today: Recent Trends and Developments, LNCS 1000*, Springer Verlag, 1995, S. 307–323

[SG96] SHAW, Mary ; GARLAN, David: *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, April 1996

[Sha94] SHAW, Mary: Procedure Calls Are The Assembly Language of Software Interconnection / Carnegie Mellon University. 1994 (CMU-CS-94-107). – Forschungsbericht

[Sih95] SIHLING, Marc. *Entwurf und Implementierung eines Modulkonzepts für die Sprache GOS, Diplomarbeit*. `http://www4.in.tum.de/~sihling`. 1995

[Sih00] SIHLING, Marc: Integrating Component Frameworks. In: MURTHY, Jayaram (Hrsg.) ; CHEN, Shu-Ching (Hrsg.): *Proceedings of the ISCA 2nd International Conference on Information Reuse and Integration*, International Society for Computers and Their Applications, November 2000

[Swi] *SUN Microsystems: Java Foundation Classes Homepage*. `http://java.sun.com/products/jfc/`

[Szy97] SZYPERSKI, Clemens: *Component Software - Beyond Object-Oriented Programming*. Addison Wesley Publishing Company, 1997

[Szy00] SZYPERSKI, Clemens: Component Software and the Way Ahead. In: SITARAMAN, Murali (Hrsg.) ; LEAVENS, Gary (Hrsg.): *Foundations of Component-Based Systems*, Cambridge University Press, 2000, S. 1–20

[Tha00] THALHEIM, Bernhard: *Entity-Relationship Modeling, Foundations of Database Technology*. Springer Verlag, 2000

[Tog] *TogetherSoft Corporation: Together Control Center — a Model-Build-Deploy Platform*. `http://www.togethersoft.com/us/products/index.html`

[UML00] Objekt Management Group: *OMG Unified Modeling Language Specification — Version 1.4 beta R1*. November 2000

[Vet91] VETTER, Max: *Aufbau betrieblicher Informationssysteme mittels objektorientierter, konzeptioneller Datenmodellierung*. 7. Auflage. B.G. Teubner Stuttgart: Leitfäden der Informatik, 1991

[VN96] VANHILST, Michael ; NOTKIN, David: Using Role Components to Implement Collaboration-Based Designs. In: *Proceedings of the OOPSLA'96*, ACM Press, October 1996, S. 359–369

[WSO00] WANG, Nanbor ; SCHMIDT, Douglas C. ; O'RYAN, Carlos. *Overview of the CORBA Component Model*. `http://www.cs.wustl.edu/~schmidt/PDF/CBSE.pdf`. 2000