Grundlagen komponentenorientierter Software

—

Untersuchung elementarer Eigenschaften moderner Komponentenarchitekturen

Diplomarbeit

von

Christian Zirpins
2zirpins@informatik.uni-hamburg.de

vorgelegt bei

Prof. Dr. Winfried Lamersdorf
Arbeitsbereich "Verteilte Systeme" (VSYS)
und
Prof. Dr. Leonie Dreschler-Fischer
Arbeitsbereich "Kognitive Systeme" (KOGS)

am Fachbereich Informatik der
UNIVERSITÄT HAMBURG

9. Juli 1999

Christian Zirpins
Im Schönenfelde 3
21109 Hamburg, Germany
Tel. +49 (40) 796 37 72

Zusammenfassung

Die vorliegende Arbeit beschäftigt sich mit grundsätzlichen Basiskonzepten einer komponentenbasierten Sichtweise, deren praktische Umsetzungen oft als *Componentware* bezeichnet werden. Der Fokus liegt dabei vornehmlich auf den Ergebnissen der Forschung. Im Zuge dessen wird zunächst der Versuch unternommen, den komplexen Themenkreis inhaltlich abzugrenzen und einführend vorzustellen. Diesbezüglich soll dann innerhalb einer breiten Übersicht bestehender Konzepte und Techniken der status quo realer Ansätze vermittelt werden. Im weiteren Verlauf werden in systematischer Weise fundamentale Aspekte allgemeiner komponentenorientierter Architekturen abgeleitet. Bezüglich dieser werden Entwurfsansätze vorgeschlagen und für ausgewählte Probleme der Bereiche Aggregation und Semantik konzeptionelle Lösungen erarbeitet. Am Ende wird durch die exemplarische Anwendung der Ergebnisse auf ein bestehendes Konzept der Praxisbezug hergestellt.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Die Komponentenidee 3

1.1.1 Evolutionsgeschichte 3

1.1.2 Wegbereiter 6

1.1.2.1 Das Softwareparadigma der Objektorientierung 7

1.1.2.2 Visuell basierte Objektmodelle auf Anwenderebene 8

1.1.2.3 Internetboom und Verteilungstrend 9

1.1.3 Fachübergreifende Gesichtspunkte 11

1.1.4 Wissenschaftliche Ansätze 12

1.2 Zielsetzungen und thematische Schwerpunkte 13

1.3 Organisatorischer Überblick 14

2 Das Softwareparadigma der Komponentenorientierung **15**

2.1 Kennzeichnung des Komponentenbegriffes 15

2.1.1 Definitionen 16

2.2 Aspekte und Eigenschaften auf technischer Ebene 19

2.2.1 Komponenten als Basiseinheiten 20

2.2.1.1 Struktur, Art und Verhalten der Basiseinheiten 20

2.2.1.2 Funktionalität und Granularität 23

2.2.1.3 Qualität 25

2.2.1.4 Technische Anpassungsfähigkeit 25

2.2.2 Integration und Interoperabilität 26

2.2.2.1 Komposition 27

2.2.2.2 Interoperation 30
2.2.2.3 Nebenläufigkeit 31
2.2.2.4 Verteilung 32
2.2.2.5 Übergeordnete Koordination und Frameworks 32
2.2.3 Formalisierung 33
2.2.3.1 Anwendung formaler Methoden 33
2.2.3.2 Typisierung 34
2.2.3.3 Spezifikation 36
2.3 Integration auf methodologischer Ebene 38
2.3.1 Domain Engineering 39
2.3.1.1 Komponentenentwicklung 39
2.3.2 Anwendungsentwicklung 40
2.3.3 Vorteile und Risiken 41

3 Übersicht bestehender Ansätze 43
3.1 Standards 43
3.1.1 Sun JavaBeans 44
3.1.2 (D)COM/OLE/ActiveX und CORBA/BOF 45
3.2 Programmiersprachen/Mechanismen 47
3.2.1 Java 47
3.2.2 STL, Component Adaptors, Gluonen, SOFA/DCUP, Sina, Aspektorientierung, Demeter 49
3.3 Koordinationssprachen 52
3.3.1 Darwin 52
3.3.2 Linda und Manifold 53
3.4 Formale Sprachen 54
3.4.1 OPUS 54
3.4.2 HOP 55
3.5 Werkzeuge/Umgebungen 56
3.5.1 Vista 56
3.5.2 Face 57
3.6 Frameworks 58
3.6.1 San Francisco 58
3.6.2 ET++ und ACE 59

3.7 Datenflußsysteme 60
3.7.1 Unix Pipes 60
3.7.2 Ptolemy 61
3.8 Skriptsprachen 61
3.8.1 Java basierte Skriptsprachen 62

4 Teilbereiche von Komponentenarchitekturen 63
4.1 Vorüberlegungen 63
4.1.1 Aspektfelder 64
4.1.2 Anforderungen 65
4.1.3 Methodologie 69
4.1.3.1 Verständnis der Komponentenmetapher . . . 69
4.1.3.2 Integration einer Refinement Sicht 71
4.2 Modellebene 71
4.2.1 Strukturmodelle 71
4.2.1.1 Eine Basisrelation für Strukturmodelle . . . 72
4.2.1.2 Strukturelle Beschränkungen 73
4.2.1.3 Schnittstellen 75
4.2.1.4 Verfeinerte Struktur 77
4.2.2 Verhaltensmodelle 80
4.2.2.1 Externes Verhalten 80
4.2.2.2 Internes Verhalten 81
4.2.3 Metamodelle 82
4.2.3.1 Anwendungsfelder 82
4.3 Beschreibungsebene 83
4.3.1 Einflußfaktoren 83
4.3.1.1 Entwurfsmethodik von Verbundkomponenten 83
4.3.1.2 Kontextbezug und Anforderungsarten 84
4.3.1.3 Integration von Refinement 85
4.3.1.4 Modellannahmen 85
4.3.2 Spezifikation 86
4.3.2.1 Spezifikationsinhalte 86
4.3.2.2 Spezifikationsstruktur 91

4.3.3 Formalisierung 95
4.3.3.1 Warum Formalismen? 95
4.3.3.2 Strukturelle Aspekte 97
4.3.3.3 Formale Semantik 97
4.3.3.4 Der zeitliche Aspekt 99
4.3.4 Typisierung im Komponentenkontext 100
4.3.4.1 Typisierungsgegenstände 102
4.3.4.2 Typrelationen 103

5 Entwurfsansätze für Komponentenarchitekturen 105
5.1 Eine exemplarische Realisierung der Modellebene 105
5.1.1 Strukturelle Modellbestandteile 106
5.1.2 Modellverhalten 110
5.2 Verhaltensbeschreibung mit Temporallogik 112
5.2.1 Temporallogik 113
5.2.1.1 Propositionale Lineare Temporale Logik . . . 115
5.2.2 PLTL basierte Formalisierungskonzepte für Komponentenmodelle 119
5.2.2.1 Grundform der Spezifikation 119
5.2.2.2 Verfeinerung der Spezifikationsstruktur . . . 123
5.2.2.3 Rolle und Problematik der Propositionen . . 127
5.3 Flexible Beschreibungsformen mit Feature-Logik 128
5.3.1 Feature Logik 128
5.3.2 Feature-Logik basierte Formalisierungskonzepte für Komponentenmodelle 131
5.3.2.1 Eine Basisspezifikation 132
5.3.2.2 Verfeinerte Beschreibungsansätze 134
5.4 Übergreifende Aspekte formaler Konzeption 136
5.4.1 Kombinationsmöglichkeiten deskriptiver und temporaler Logiken 136
5.4.2 Typisierungsansätze 140
5.4.3 Überblick und Bewertung der Ergebnisse 142

6 Realisierungsmöglichkeiten komponentenorientierter Architekturkonzepte am Beispiel JavaBeans **143**

6.1 Konzeptionelle Untersuchung 143

6.1.1 JavaBeans: Ein Überblick 144

6.1.1.1 Das Komponentenmodell 144

6.1.1.2 Die Anwendungsgsmethodologie 149

6.1.2 Analyse und Bewertung 151

6.2 Modifikationsansätze 153

6.2.1 Interne Variation 154

6.2.2 Externe Erweiterung 157

6.2.2.1 Eine dienstbasierte Metaebene 157

6.2.2.2 Generische Richtlinien 161

6.3 Fazit 162

7 Schlußbetrachtung **165**

7.1 Zusammenfassung 165

7.1.1 Ergebnisse 166

7.2 Ausblick 167

Abbildungsverzeichnis **169**

Tabellenverzeichnis **171**

Literaturverzeichnis **173**

Kapitel 1

Einleitung

Unter den vielen Schlagworten, die im Umfeld der Informatik kursieren, hat sich das der *"Componentware"* in den letzten Jahren besonders schnell und nachhaltig verbreitet.

Die Intensität dieser Verbreitung ist dabei weniger auf konkrete, inhaltliche Aspekte als auf den massiven Einsatz des Begriffes als Marketinginstrument zurückzuführen. Sowohl in den einschlägigen Medien, die jeden vermeintlich neuen Trend sofort vermarkten, als auch innerhalb der Softwarebranche, in der leicht abgeänderte Produkte gerne mit Fortschrittlichkeit suggerierenden Prädikaten aufgewertet werden, soll *"Komponentenorientierung"* den Absatz vornehmlich durch seine Sekundärattribute fördern. So wird diese spezifische Eigenschaft den entsprechenden Artefakten dann auch sehr schnell zugewiesen und in eher uneinheitlicher Weise ausgelegt, denn obwohl Componentware in aller Munde ist, herrscht über deren konkrete Inhalte noch weitgehende Uneinigkeit und sowohl in Kreisen der freien Wirtschaft als auch der klassischen Wissenschaft finden heftige, zum Teil kontroverse Diskussionen darüber statt, was die Grundlagen dieser Herangehensweise ausmacht, und in welchem Verhältnis sie zu anderen Sichten steht [BDH+98].

Überwiegendes Einvernehmen herrscht lediglich bei den durch Komponentenorientierung angestrebten Zielen, welche in ihrer Essenz dem Wunsch nach einer Lösung dringender, traditioneller Probleme der Softwaretechnik entstammen.

An erster Stelle steht dabei im allgemeinen die seit langem mit bisher eher geringem Erfolg angestrebte Aktivierung der Potenzen von *Wiederverwendung* in Bezug auf Artefakte der Softwareentwicklung. Damit eng einher gehen ebenfalls generell ungelöste Probleme der *Anpassbarkeit*, *Flexibilität* und *Wartbarkeit* von Programmsystemen sowie deren *Evolution* (besonders auch zur Laufzeit) über den gesamten Lebenszyklus. Etwas konkreter wünscht man sich Unterstützung für das umfassende *Integrationsmanagement* heterogener Anwendungs- und Systemlandschaften möglichst unter Berücksich-

tigung historisch eingebrachter, informationstechnischer Altlasten.

Wie wesentlich eine Lösung dieser Probleme ist, erkennt man bei Erweiterung des Betrachtungshorizonts über rein technische Aspekte hinaus.

Die Erstellung von Softwaresystemen ist zu einem der wichtigsten wirtschaftlichen wie gesellschaftlichen Faktoren geworden, von dem das Wohl und die Zukunft komplexer sozioökonomischer Gefüge abhängen. In der Wirtschaft ist leistungsfähige Software im harten Wettbewerb überlebenswichtig. Bei sicherheitsrelevanten Systemen — z.B. Prozeßsteuerung im Bereich von industriellen Anlagen, Kernkraftwerken oder militärischen Einrichtungen — ist der eben noch bildlich angewandte Begriff "überlebenswichtig" in Bezug auf absolute Verläßlichkeit der Informationstechnologie wörtlich zu nehmen. Im Hinblick auf gesellschaftliche Belange wären die staatliche Verwaltung, öffentliche Dienste und innere Sicherheit genauso wie das Gesundheitswesen und der Sozialbereich handlungsunfähig [FHPH95].

Seit etlichen Jahren wird nun von einer *Softwarekrise* gesprochen, welche unter Berücksichtigung der gleichermaßen weitreichenden wie folgenschweren Wechselwirkungen mit den obigen essentiellen Bereichen nicht als bloße Phrase abgetan werden kann. Die Problematik der Krise bezieht sich dabei auf die mangelnde Fähigkeit, Programme in einer dem steigenden Bedarf gerechten qualitativen und quantitativen Weise zu erzeugen. Ein wesentlicher Grund dieser Schwäche ist in der Methodik zu sehen, mit der heute Software als immer wieder grundauf neue Einzelanfertigung erstellt wird. Als vielversprechender Lösungsansatz bietet sich hier nun die Componentware Idee von einer systematischen Wiederverwendung bewährter, qualitativ hochwertiger Teilbausteine im Sinne des Vorgehens klassischer Ingenieursdisziplinen an. Softwareerstellung soll durch diese Vorgehensweise beschleunigt werden und gleichzeitig zu verläßlicheren Ergebnissen führen. Neben dem Ziel einer adäquaten Bedarfsbefriedigung verspricht diese Strategie dann auch noch eine erhebliche Steigerung der Wirtschaftlichkeit, entspricht sie doch auf betriebswirtschaftlicher Ebene dem Wechsel von klassischer Einzelanfertigung bei Werkstattfertigung zu diversifizierter Großserienproduktion bei Fließfertigung.

Bei all der Euphorie dürfen die immensen Schwierigkeiten und daraus resultierende Gefahren nicht außer Acht gelassen werden, die solche Lösungsansätze mit sich bringen. Welche Auswirkungen eine ad hoc Wiederverwendung vorhandener Codebausteine haben kann, wurde wohl am deutlichsten bei der Ariane-5 Katastrophe vor Augen geführt. Die Explosion der Rakete konnte eindeutig auf falsche Wiederverwendung von Codefragmenten innerhalb der Steuerungssoftware zurückgeführt werden [Lio96].

Die obigen Ausführungen zeigen deutlich sowohl Relevanz als auch Dringlichkeit eines Komponentenkonzeptes. Darüber hinaus zeigen sie aber auch die essentielle Bedeutung einer wissenschaftlich fundierten Lösung, die eine

systematische Anwendung ermöglicht und durch Beweismöglichkeiten Korrektheit und damit Sicherheit resultierender Systeme gewährleistet.

Fest steht, daß der Markt dem Forschungsbereich weit vorauseilt und eine geregelte wissenschaftliche Basis mit homogenem Begriffsapparat und klar definierten möglichst einheitlich standardisierten Modellen nicht existiert. Noch wesentlich weiter ist man von einer uniformen, formalen Grundlage mit fundierten Ansätzen und Methoden entfernt, welche wiederum Voraussetzung für die Unterstützung von geordneter und gesicherter, weil verifizierbarer Verwendung von Systemen mit Componentware-Charakter ist.

Führt man sich die Zielsetzungen in Bezug auf Komponentenorientierung und den daraus resultierenden umfangreichen Forderungskatalog an diesbezügliche Techniken in ganzer Breite vor Augen, werden unweigerlich Zweifel an die Realisierbarkeit geweckt, sind doch praktisch alle wesentlichen, seit langem offenen Probleme des Metiers enthalten. Es stellt sich also im folgenden die Frage nach der entsprechenden inhaltlichen Basis eines *Komponentenparadigmas*, die in der Lage ist, den hochgesteckten Erwartungen zu genügen.

1.1 Die Komponentenidee

1.1.1 Evolutionsgeschichte

Bei einer Hinterfragung der konzeptionellen Grundlagen komponentenorientierter Softwareentwicklung stößt man zunächst — angesichts der Aktualität diesbezüglicher Diskussionen vielleicht überraschend — auf deren relativ weit zurückreichende, traditionelle Wurzeln. Die Grundidee der Componentware ist nämlich keineswegs neu. Schon 1968 prognostizierte McIlreu den Wandel der Softwareerstellung von den bislang stets grundauf neuen, individuellen Einzelentwicklungen hin zu einer (Re-)Kombination vorgefertigter, bewährter Teilprogramme nach dem Vorbild traditioneller Ingenieursdisziplinen [McI68]. Er sah weiterhin die Entstehung eines globalen, umfassenden Marktes für solche "Softwarekomponenten" voraus. Von diesem Zeitpunkt an ist das Thema in die permanente wissenschaftliche Diskussion eingegangen und immer wieder Gegenstand von Untersuchungen sowie Inhalt von Forschungsprojekten mit bislang eher unbefriedigenden Resultaten gewesen — die damalige Vision ist nach wie vor Fiktion geblieben.

Seit diesen ersten Ansätzen — McIlreu dachte seinerzeit an Bausteine, welche auf einer textuellen Sourcecode-Ebene basieren sollten — hat sich die Idee der Komponenten kontinuierlich gewandelt. Vor allem das klare Wesen des Begriffes "Komponente" als sourcecodebasierter Programmbaustein ist stark aufgeweicht worden und einer Flut unterschiedlichster Auffassungen gewichen, die zu einem fließenden, schwammigen Wortcharakter geführt

haben. Der Begriff ist in seiner Auslegung heute einer großen Bandbreite möglicher Inhalte offen. Was eine Komponente *ist*, *sein kann* oder *sein sollte* reicht potentiell von sehr allgemeinen Auffassungen wie *"jegliches Artefakt der Softwareentwicklung das wiederverwendbar ist*[1]" bis zu engen Konzept-, Sprach- oder gar produktgebundenen Auslegungen wie *"Java Klasse, die den ´Beans´-Konventionen genügt*[2]". Diese beiden Beispiele geben in etwa die Extreme der bestehenden Palette möglicher Komponentendefinitionen wieder. Eine detaillierte Diskussion der wichtigsten Ansichten und Vorschläge erfolgt später in Kapitel 2.

Innerhalb der vorliegenden Arbeit wird bezüglich des Begriffsverständnisses ein Mittelweg beschritten. Im Einklang mit der Namensverwandtschaft von *Componentware* und *Software* soll der Komponentenbegriff per se nicht auf jegliche Artefakte im Umfeld von Programmentwicklungen ausgeweitet werden, welche auch Entwurfsentscheidungen und -konzepte — z.B. in Form von *Designpatterns* [GHJV94] — oder Begleitprodukte wie Dokumentation beinhalten, sondern seines Wesens nach ausführbaren Code meinen. Der Terminus "ausführbarer Code" macht dabei allerdings in Bezug auf dessen Natur zunächst keinerlei Einschränkungen und beinhaltet im Zuge dessen alle denkbaren Ausprägungen computerspezifischer Programmfragmente, angefangen von Maschinencode über Hoch- bis Metasprachen.

Die Entwicklung eines solchen Komponentenverständnisses im Sinne von ausführbaren Code ist naturgemäß eng mit der Geschichte von Programmiersprachen verbunden. An deren Anfang stand eine stark technikzentrierte Sicht, deren Ausdrucksmittel primitive Maschinensprachen waren. Diese wurden in einer zweiten Evolutionsstufe von domänenorientierten Sprachen mit ausgeprägtem Werkzeugcharakter wie etwa FORTRAN oder COBOL (für die mathematische bzw. betriebswirtschaftliche Domäne) abgelöst. Letzteren vorwiegend auf spezielle Anwendungsprobleme ausgerichteten Ansätzen folgten allgemeinere, welche sich stärker an den Programmstrukturen selber orientierten. Letztere Entwicklung brachte dabei immer mächtigere Abstraktionen auf zusehends höheren Ebenen wie Prozeduren, Module oder Pakete hervor. Zu dieser Sprachgattung der dritten Generation ist z.B. auch der objektorientierte Ansatz mit seinen Klassenkonzepten zu zählen. Mit dem Aufkommen von Koordinationssprachen wurde dann schließlich eine explizite Metaebene eingeführt, auf Basis derer die Kombination sehr mächtiger Abstraktionen möglich wurde. Der Ansatz zeigt schon deutliche Parallelen zur Komponentenidee und wird gemeinhin auch in einem Atemzug mit dieser genannt. Als vorübergehender Gipfel dieser Entwicklung vom "programming in the small" hin zu einem "programming in the large" kann das sogenannte Megaprogramming angesehen werden, dessen Inhalt die Kombination kompletter, zum Teil extrem großer Anwen-

[1] Siehe [HC91]
[2] Siehe [Sun97a]

dungssysteme (etwa die Zusammenführung von Buchungssystemen zu einem weltweiten Verbund) umfaßt.

Die Evolutionsgeschichte der Programmiersprachen führt zu einem Verständnis von Komponenten als höherwertige Abstraktionen logisch klar abgrenzbarer computerspezifisch ausführbarer Artefakte, welche explizit auf separater Ebene zu vollständigen Softwaresystemen kombiniert werden können. Komponenten sollen dabei Bausteine darstellen, welche nicht nur die Möglichkeit einer Kombination bieten, sondern diese zur sinnvollen Verwendung voraussetzen. Letzterer Vorgang wird gemeinhin durch den Begriff der *Komposition* ausgedrückt, da eine derartige Schaffung von Softwaresystemen aus Bausteinen der Zusammenstellung von Musikstücken aus Noten gleicht, bei der ja einzelne Töne für sich auch noch keine harmonische Melodie ergeben. Eine solche Sichtweise legt Einheiten mittlerer Granularität (*engl. "medium grained units"*) nahe, denn solche sind noch nicht so komplex, daß sie zu den gleichen softwaretechnischen Problemen führen würden, für deren Lösung sie erdacht wurden; andererseits erleichtert eine logisch abgrenzbare Funktionalität gewissen Umfangs aber die An- bzw. Wiederverwendung und sichert eine Existenzberechtigung neben den vorhandenen programmiertechnischen Konstrukten.

Um einen diesbezüglichen Komponentenbegriff jetzt schon etwas konkreter fassen und in argumentativer Weise verwenden zu können, soll an dieser Stelle vorab eine erste sehr allgemeine Definition im Sinne der vorliegenden Arbeit gegeben werden, welche sich an [Gri98] orientiert.

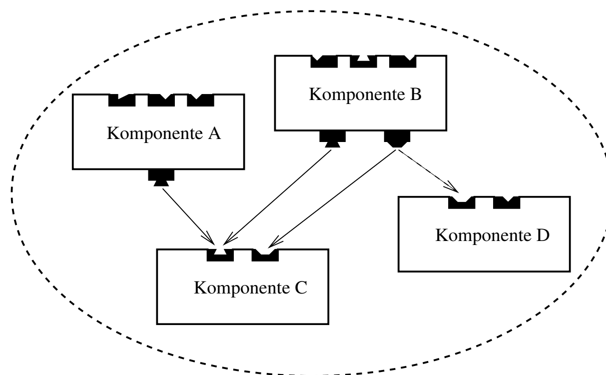


Abbildung 1.1: Softwarekomponenten

Charakterisierung 1 (Softwarekomponente) *Der Begriff* ***Softwarekomponente*** *(kurz: Komponente) steht für ein Stück Software, das klein genug ist, um es in einem Stück erzeugen und pflegen zu können, groß genug ist, um eine sinnvoll einsetzbare Funktionalität zu bieten und eine individuelle Unterstützung zu rechtfertigen sowie mit standardisierten Schnittstellen ausgestattet ist, um mit anderen Komponenten zusammenzuarbeiten.*

Abbildung 1.1 veranschaulicht die Sichtweise noch einmal in grafischer Form. Komponenten sind dort als abstrakte Bausteine dargestellt und mit Schnittstellen (*engl. "Interfaces"*) ausgestattet, welche durch sich paarweise ergänzende, opaque Polygone symbolisiert werden. Über diese Interfaces stehen die Subsysteme miteinander in einer durch Pfeile dargestellten Nutzungs- bzw. Interaktionsbeziehung. Ein umgebendes Oval deutet hier das alle Teilkomponenten verbindende Streben hin zum endgültigen Ziel des fertigen Softwaresystems an, was auch als gemeinsames Rahmenwerk (*engl. "Framework"*) interpretiert werden kann.

1.1.2 Wegbereiter

Warum lebt das Interesse an expliziten Komponenten nun gerade jetzt wieder auf, wo doch die Idee an sich schon seit dreißig Jahren im Raum steht? Eine Antwort auf diese Frage kann nicht pauschal ausfallen. Die wesentlichsten Einflußfaktoren sollen im folgenden zusammengetragen und anschließend kurz erläutert werden.

Ein erster wesentlicher Punkt ist sicherlich die Enttäuschung darüber, daß zahlreiche hochgesteckte Erwartungen, welche an das Paradigma der *Objektorientierung* bezüglich dessen softwaretechnischen Fähigkeiten — insbesondere im Bezug auf Wiederverwendung und Wartbarkeit — gestellt wurden, sich bis heute nicht erfüllt haben. Die Objektsicht ist aber auch in einem zweiten positiven Sinne Promotor der Komponententechnik, denn zwischen Objekt- und Komponentenorientierung besteht eine nahe Verwandtschaft mit intensiven, wechselseitig förderlichen Einflüssen.

Als weitere Punkte sind zwei neuere Tendenzen der Informationstechnik zu nennen, welche ideale Anwendungsfelder der Komponentensicht darstellen und der Verbreitung letzterer deshalb mittels ihres starken (Markt-) Einflusses in erheblicher Weise dienlich sind. Gemeint ist dabei als erstes der Trend zu *visualisierten Objekten* — wobei der Objektbegriff hier sehr weit gefaßt ist. Ein Beispiel dafür ist die objektorientierte Desktop-Metapher grafischer Benutzeroberflächen im Bereich der Mensch-Maschine-Kommunikation, die sich vor allem bei den Betriebssystemen zu einer Standardform der Anwenderschnittstellen etabliert haben. Als zweiter positiver Einflußfaktor ist die zunehmende Vernetzung (d.h. insbesondere auch *Verteilung*) autonomer Computerressourcen im Zuge der Integrationsbemühungen einer steigenden Anzahl leistungsfähiger Einzelplatzrechner zum einen und des ungebrochenen Internetbooms zum anderen mit Konsequenz des häufig zitierten *"Phasenwechsels der Informationstechnik"* anzusehen.

Im Zusammenspiel dieser Faktoren, das seinerseits durch wechselseitige Einflüsse geprägt ist, ergibt sich ein äußerst effizienter Nährboden für die Komponentensicht und ähnliche Ansätze.

1.1.2.1 Das Softwareparadigma der Objektorientierung

Nachdem das Softwareparadigma der Objektorientierung [Mey88], unter dem im allgemeinen die Konzepte Kapselung, Vererbung sowie Polymorphismus zusammengefaßt werden, während der letzten zehn Jahre in Bedeutung und Inhalt eingehend untersucht wurde und nunmehr weitgehend verstanden ist, wird klar, daß die Sichtweise zwar ohne Zweifel einen Meilenstein der Informatik darstellt, bekannte substantielle Probleme bei der Erstellung von Softwaresystemen aber trotzdem bestehenbleiben [Ude94]. So kann das Paradigma den gestellten hohen Erwartungen, speziell im Bezug auf Wiederverwendung und Wartbarkeit, scheinbar nicht gerecht werden, und die Produktion von Software stellt trotz voranschreitender Etablierung objektorientierter Programmbibliotheken und Frameworks immer noch eine Kreation quasi maßgeschneiderter Individuallösungen mit hohem Planungs-, Entwicklungs-, Test- sowie Wartungsaufwand dar. Dieser Umstand gewinnt unter Berücksichtigung der erheblichen Komplexitätssteigerung moderner Programmsysteme weiter an Brisanz.

Im Zuge dieser enttäuschten Erwartungen richtet sich das Interesse immer mehr auf den Themenkreis der Softwarekomponenten, welcher in der Informatik zwar wie gesehen schon seit geraumer Zeit ohne durchschlagende Erfolge diskutiert wird, dessen konkrete Umsetzung aber nun im Rahmen der fortschreitenden technischen Entwicklung in den Bereich des Möglichen rückt. Während das Komponentenparadigma Ansätze enthält, welche genau die angesprochenen aktuellen Probleme der Objektorientierung zu lösen versprechen, stellt letztere die zur Zeit fortschrittlichste Basis zu dessen Verwirklichung dar. Die Komponentensicht soll also Objektorientierung nicht etwa ablösen, sondern um neue essentielle Aspekte bereichert und so zu einem Konzeptverbund mit Synergieeffekten führen.

Konkret erweitert das Komponentenparadigma die auf klassischer Programmierung in 3GL[3] Manier basierende Objektorientierung um eine neue Form der Komposition abstrakter, gekapselter Softwarebausteine, die verglichen mit den vorhandenen Möglichkeiten der Verknüpfung von Klassen bzw. Objekten wesentlich "losere" Koppelungen — im Sinne einer Vermeidung wechselseitiger, insbesondere zusätzlich verdeckter Abhängigkeiten — ergibt. Dadurch stellen Komponenten praktisch gesehen die "dinglicheren Objekte´´ dar, welche sich durch wesentlich größere Unabhängigkeit und damit auch Vielseitigkeit ihrer (Wieder-)Verwendung auszeichnen. Andererseits sind es aber gerade auch die objektorientierten Ansätze, die durch Techniken wie etwa die späte Bindung (*engl. "late binding"*) zur Realisierung von Komponenten als am besten geeignet erscheinen. Vor allem die verteilten Objektsysteme wie z.B. *CORBA (Common Object Request Broker Architecture)* [OMG96b] sind dabei geeignete Kandidaten, da ihnen von vornherein eine

[3]Third Generation Languages s.o.

naturgemäß weitaus autonomere Objektsicht innewohnt.

Das Szenario der zwei harmonisch zusammenspielenden Paradigmen ist jedoch umstritten, denn gerade im Zusammenhang mit Objektorientierung werden Softwarekomponenten oftmals nicht als eigenständiges Konzept, sondern lediglich als Synonyme angesehen. Man kann hierbei feststellen, daß einerseits die Grenzen sicherlich — vor allem bei voller Ausschöpfung des breiten Spektrums an Objektsichten (z.B. die oben angesprochenen "verteilten Objektsysteme" oder "Businessobjects" weiter unten) — verschwommen sind, andererseits aber ein Objektbegriff, wie er in der heutigen Programmierpraxis gängiger Produktionssprachen wie *C++* üblich ist, den an Komponenten gestellten Anforderungen nicht entsprechen kann.

Objektorientierte Ansätze sind nicht nur für Programmplanung und -entwicklung eine vorteilhafte Basis, sondern eignen sich auch besonders gut für Abstraktionen auf Anwenderebene. Diese sind Inhalt des nächsten Abschnitts.

1.1.2.2 Visuell basierte Objektmodelle auf Anwenderebene

Heutige grafische Benutzeroberflächen orientieren sich an der Metapher des "virtuellen Schreibtisches" und visualisieren Objekte des täglichen (Büro-) Lebens, welche in sich geschlossene Funktionseinheiten informationstechnischer Anwendungen abstrahieren. Als wohlbekanntes Beispiel möge hier der "Mülleimer" dienen, welcher bei praktisch allen neueren Betriebssystemoberflächen für eine nichtdestruktive Löschfunktion steht.

Dieser Ansatz, welcher oft als "objektorientiert auf Anwenderebene" charakterisiert wird, ist auf der Softwareebene eine ideale und zudem natürliche Domäne für das Komponentenparadigma. Die Anwendungsobjekte sind relativ zu ihrem funktionalen Umfang von mittlerer Granularität und zudem jeweils für sich abgeschlossen. Trotzdem ergibt sich ein höherer Sinn erst aus dem Zusammenspiel solcher Komponenten — so wäre eine "Mülleimer"-Komponente ohne weitere potentiell zu löschende Einheiten wie etwa "Datei" Komponenten wenig sinnvoll.

Im allgemeinen Trend der grafischen Objektrepräsentationen hat sich mit der *visuellen Programmierung* eine neue Form der Softwareerstellung etabliert. Hierbei werden Programmbausteine als Ikonen repräsentiert, welche mittels grafischer Manipulationen — etwa dem wechselseitigen Verbinden durch Linienzüge als Abstraktion von Interaktionsbeziehungen — die Modifikation eines aus ihnen zusammengesetzten Softwaresystems zumeist in Echtzeit bewirken. Derartige Ansätze können ihr volles Potential am besten in Kombination mit Komponententechniken ausschöpfen, da solche als einzige alle notwendigen Features, wie die einfache (Re-)Kombination bzw. Komposition zur Laufzeit und die flexible Anpassbarkeit durch Customizing

bieten. Die zukünftige Bedeutung der visuellen Programmierung für große, langlebige Softwareprojekte wird in diesem Sinne stark von den Fortschritten im Bereich grundlegender Komponentenmodelle abhängen. Ein Beispiel für visuelle Programmierumgebungen ist Suns experimentelles *Java-Studio* Produkt [Sun97b], welches auf der hauseigenen Komponentenarchitektur Java-Beans basiert (siehe Abb.1.2).

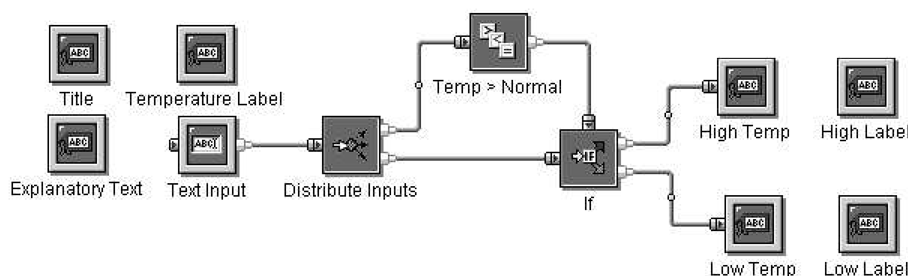


Abbildung 1.2: Visuelle Programmierung mit Java Studio-Beans

Die Tendenz geht momentan zu noch komplexeren "Objekten", zu denen auch die sogenannten *Business-* oder *Enterprise-Objects* [JGJ94] (z.B. in Form von *Java Enterprise Beans*) zu zählen sind. Business Objects sind Softwarebausteine, die in Hinblick auf Funktionalität und wechselseitiges Verhalten für komplette Kontexte von zum Teil umfangreichen Anwendungsszenarien wie etwa betriebswirtschaftliche Geschäftsprozesse stehen. Sie sollen dabei in ihrem nativen Zustand auf Anwenderebene gebrauchsfertig, d.h. erkennbar, ausführbar und kombinierbar sein. Daß *ein* Objekt im Sinne objektorientierter Programmierung für solche inhaltlich reichen Geschäftsobjekte auf Grund deren Komplexität ein wenig sinnvolles Implementationsmittel darstellt, ist offensichtlich. Der Komponentenansatz scheint hingegen auch hier ein gangbarer Weg zu sein. Als konkretes Beispiel für entsprechende Architekturen sei auf die CORBA basierte *Business Application Architecture (BAA)* [OMG96a] der OMG verwiesen.

Eine wichtige Eigenschaft von Anwendungssystemen, die auf Business Objekten basieren, ist deren inhärenter Verteilungsaspekt auf Grund der ausgeprägten Vernetzungstendenz innerbetrieblicher informationstechnischer (kurz *it*) Strukturen. Mit diesem Aspektkreis wird im nächsten Abschnitt der letzte hier betrachtete Promotor von Componentware skizziert.

1.1.2.3 Internetboom und Verteilungstrend

Die anhaltend rasant voranschreitende Entwicklung im Bereich informationstechnischer Systeme führt neben Steigerungen von Qualität und Quan-

tität einzelner Rechner bei simultanem Preisverfall zu einer verstärkten, der Integration dienenden Vernetzung autonomer Systeme, ermöglicht durch immer leistungsfähigere Kommunikationstechnik. Dieses Szenario bildet den Nährboden für einen anhaltenden Verteilungstrend, welcher sich am demonstrativsten im globalen Internetboom und der schnellen Verbreitung betrieblicher Intranets widerspiegelt. Im Rahmen dieses Kontextes entwickelt sich bei Entwicklung und Anwendung ein neues Verständnis der Datenverarbeitung, in dessen Zusammenhang oft von einem "Phasenwechsel der Informationstechnologie" gesprochen wird.

Die Verteilungstendenz schafft gleichermaßen Voraussetzung wie Bedarf für eine neue Klasse von Softwaresystemen, welche die Funktionalität verschiedener, autonomer Teile nutzen und zu komplexen, neuen Anwendungen kombinieren, um die dahinterstehenden, entfernten Ressourcen verfügbar zu machen oder optimal auszulasten. Derartige Programmgefüge sind durch eine wechselseitige Zusammenarbeit auf natürliche Weise zerlegter Teilbausteine gekennzeichnet, weswegen man auch von *Kooperationsanwendungen* spricht [MJML95].

Der Komponentengedanke paßt naturgemäß optimal in ein verteiltes Szenario, denn er enthält mit seinen Prinzipien der Interaktion lose gekoppelter Teilkomponenten zum einen die Charakteristiken von Kooperationsanwendungen und bereichert diese um weitere softwaretechnische Aspekte wie Flexibilität oder Wiederverwendbarkeit.

Die Teilsysteme von Kooperationsanwendungen können bei stärkerer Abstraktion auch als *Dienste* innerhalb eines entsprechenden, umfassenden Marktes aufgefaßt werden. Dienste werden in diesem Rahmen von Diensterbringern angeboten und durch Dienstnehmer genutzt. Das Dienstemarktszenario wird dabei erst mittels einer komplexen, durch vielfältige systemtechnische Unterstützungsmechanismen wie Typmanagement, Trading oder Interzeption erbrachten Infrastruktur ermöglicht. Ein solches System wird beispielsweise detailliert in [MJ96] beschrieben. Offensichtlich bestehen zwischen einer Komponentensicht und dieser starke Parallelen. Die dort gewonnenen Erfahrungen lassen sich daher in fruchtbarer Weise auf den Entwurf von Komponentensystemen übertragen und geben zahlreiche, wertvolle Hinweise für dortige Lösungen.

Ein aktuelles und bekanntes Beispiel für verteilte Softwaresysteme mit Komponentencharakter ist das oft mit dem "Internet" gleichgesetzte World-Wide-Web (WWW) bzw. dessen unterliegende Technik der verteilten, wechselseitig referenzierten, multimedialen Dokumente inkl. dazugehöriger Anzeige- und Navigationswerkzeuge (*Browser*). WWW-Seiten — welche Dokumente auf Basis der Sprache HTML darstellen — sind aus verschiedenen multimedialen Teilen zusammengesetzt, welche auch ausführbare, interagierende Codebausteine (meist in Form von *Java-Applets*) enthalten können. Die entsprechenden Browser können durch dynamische Integration soge-

nannter *Plugins* um die zur Darstellung der in den Seiten enthaltenden Teile benötigte Funktionalität erweitert werden. Das System beinhaltet somit einen Komponentengedanken in doppeltem Sinne.

Losgelöst von konkreten Markt- und Technologietrends läuft der inhärente Verteilungsaspekt des Komponentenparadigmas konform zu einer interessanten visionären Ideologie der autonomen nichtdeterministischen Interaktion, welche maßgeblich von Wegner in [Weg97] eingeführt wurde. Wegner betrachtet dort in kritischer Weise die Churche These und den resultierenden klassischen Berechenbarkeitsbegriff der Informatik, basierend auf dem Prinzip von Turingmaschinen. Insbesondere sagt er das Ende des streng deterministischen Algorithmus voraus und stellt eine Sicht, basierend auf Interaktion freier, unabhängig agierender Instanzen, die eine maximale Form autonomer Delegation darstellen, dagegen. Ob sich diese Vision nun bewahrheitet oder nicht, so weist sie doch klare Parallelen zur wesentlich konkreteren Komponentensicht — zumindest in ihrer idealen Form — auf, betont letztere doch auch gerade ein auf loser Interaktion beruhendes Zusammenspiel möglichst eigenständiger Softwarebausteine.

Nachdem nun einige wesentliche, überwiegend technisch orientierte Aspekte des Komponentenparadigmas mit seinen anfänglichen Grundlagen, seiner Entwicklung sowie seinen Beziehungen zu aktuellen Einflußfaktoren betrachtet wurden, sollen im folgenden noch Wechselwirkungen mit anderen z.T. fachübergreifenden Bereichen angesprochen werden.

1.1.3 Fachübergreifende Gesichtspunkte

Das Komponentenparadigma hat neben seinen technischen Aspekten vielfältige fachübergreifende Gesichtspunkte, von denen die wesentlichsten im folgenden angesprochen werden sollen.

Nicht zu vernachlässigen sind zunächst die psychologischen Hemmnisse einer solchen Vorgehensweise, die zu einer Ablehnung des Konzeptes bei den involvierten Akteuren führen kann. Diese werden praktisch in zwei Klassen geteilt, wobei die einen in vermeintlich kreativerer Weise Komponenten implementieren, während die anderen auf deren bloße Rekombination beschränkt sind. Hinzu kommt das bekannte *NIH Syndrom*[4], welches für das Mißtrauen gegenüber nicht selber erzeugter Software steht. Um das volle Potential von Komponentenansätzen nutzen zu können, scheint daher die Motivation eines neuen pragmatischeren Verständnisses der Softwareentwicklung von der Kunstform hin zu einem Produktionsprozeß Voraussetzung zu sein.

Ein weiteres Problem sind die rechtlichen Grundlagen der Komponentenverwendung. Zum einen ist die Fragestellung dabei, ob und wenn ja in welchem zeitlichen- bzw. mengenmäßigem Umfang sowie unter welchen Bedingungen

[4] “not invented here”

die Verwendung bestimmter Komponenten durch Anwender legitim ist. Zum anderen muß auch die Haftung der Anbieter in Bezug auf Nichterfüllung zugesicherter Produkteigenschaften und etwaiger Folgen geklärt sein. Hier fehlen weitgehend die zur Umsetzung notwendigen technischen Mittel.

Aus wirtschaftlicher Sicht stellt sich für den Anwender zunächst die Frage, ob eine generelle (Wieder-)Verwendung von Komponenten, seien diese nun selbsterstellt oder fremdbezogen (*off-the-shelf-components*), finanzielle Vorteile bietet. Im Hinblick auf die Kapitalintensität und den weiten Planungshorizont entsprechender Strategien ist diese Frage nicht immer leicht zu beantworten [Weg84]. Speziell bei Fremdbezug können dann noch Konflikte mit der Firmenpolitik (z.B. in Bezug auf Kernkompetenzen) entstehen.

1.1.4 Wissenschaftliche Ansätze

Wie anfangs erwähnt, sind wissenschaftlich fundierte Grundlagen zum Thema Komponentenorientierung eine unabdingbare Voraussetzung für den langfristigen Erfolg entsprechender Konzepte. Zwar bilden auch Systeme, denen ein solches Fundament abgeht, zum Teil sehr attraktive Ansätze; den Idealen der reinen Komponentenidee, mit dem Ziel einer Softwarekrise entgegenzuwirken, können sie nicht genügen und daher auch keine bedeutsamen Tendenzen — sprich: einen Paradigmenwechsel — begründen.

Als Status-Quo ist festzustellen, daß sich die Forschung noch nicht in einem gefestigten Stadium befindet. Trotzdem gibt es aber eine Reihe vielversprechender Projekte, die sich meist auf Teilbereiche des komplexen Themenkreises konzentrieren und auf Realisierbarkeit der angestrebten Konzepte hoffen lassen.

In diesem Zusammenhang stellt sich die Frage der Identifizierung und Abgrenzung relevanter Problemfelder. Nierstrasz und Dami [ND95] heben in Bezug auf komponentenorientierte Entwicklung die folgenden drei Forschungsschwerpunkte hervor:

- Verschmelzung der gegenwärtigen Abstraktionsbegriffe von Prozeßkalkülen sowie funktionalen- und objektorientierten Sprachen zu einer Konkretisierung des *Komponentenbegriffs*, welcher eine persistente, "Firstclass´´ Einheit mit Konzepten der Parametrisierung, Instanziierung und der Möglichkeit zur Skalierung darstellen sollte.
- Entwicklung von unterstützenden*Softwarewerkzeugen*, welche die Manipulation partieller Konfigurationen beherrschen und einen iterativen Zusammenbau von Komponenten in einer Vielzahl verschiedener Darstellungsschichten ermöglichen sollten.[5]

[5]Der Terminus *Darstellungsschichten* bezieht sich hier auf verschiedene Repräsenta-

- Finden ausdrucksstarker *Typsysteme*, welche auf Inferenzmechanismen oder Teilauswertung basieren und die Korrektheit von Software-Konfigurationsfragmenten in für den Programmierer transparenter Weise entscheiden können.

Fortschritte in diesen Bereichen bedürfen danach einer engen Verzahnung der theoretischen Ebene mit formaler Semantik und Typisierung sowie praktischer Gebiete mit Implementationsansätzen und Compiler- bzw Interpreterentwurf.

1.2 Zielsetzungen und thematische Schwerpunkte

Die vorliegende Arbeit strebt das Ziel an, unter Abstraktion der diffusen, weitgefächerten Begriffswelt des Komponentenparadigmas eine grundlegende Untersuchung deren wesentlicher Prinzipien und Konzepte auf fundamentalem, wissenschaftlichen Niveau durchzuführen.

Das Vorhaben teilt sich in die initiale Identifikation der relevanten Themenbereiche zum einen und die Bestimmung ihrer inhaltlichen Implikationen zum anderen. Im Zuge dessen soll durch möglichst breite Betrachtung verschiedener Sichten und konkreter Ausprägungen des Komponentenparadigmas die Bandbreite der zentralen Inhalte bestimmt werden. Die so gesammelte Materie dient als Grundlage für die Ableitung eines konzeptionellen Rahmens in Form von Aspekten und Konzepten eines allgemeinen abstrakten Architekturbegriffs für Komponentensysteme.

Aus dem Themenspektrum allgemeiner Komponentenarchitekturen sollen für einige ausgewählte Probleme konzeptionelle Lösungen erarbeitet werden. Diese betreffen vornehmlich Konzepte der Aggregation und formalen Spezifikationen innerhalb der Modell- bzw. Beschreibungsebene von Komponentenarchitekturen. Im Spezifikationsbereich soll dabei die Berücksichtigung semantischer Aspekte von Komponenteninteraktionen eine wesentliche Rolle spielen. Um solche Verhaltensaspekte in den Rahmen eines Komponentenmodells integrieren zu können, werden verschiedene Formen formaler Semantiken auf ihre Eignung hin untersucht. Insbesondere die Felder der Temporal- und Feature-Logiken erweisen sich dabei als vielversprechend und dienen dann auch als Basis eingehenderer Betrachtungen.

Um schließlich wieder den Bezug zur Praxis herzustellen, sollen die gewonnenen allgemeinen Erkenntnisse auf ein bestehendes Komponentensystem

---

tionsformen der Software zwischen voll interpretierten Sprachen und optimiertem Maschinencode. Dazwischen sind eine Vielzahl von Mischformen wie teilinterpretierter Bytecode oder Maschinencode mit gespeicherten Kompilierungs- bzw. Bindungsinformationen denkbar.

angewendet werden, wobei ein Vergleich mit den abgeleiteten komponentenorientierten Architekturkonzepten und eine Anwendung der erarbeiteten konzeptionellen Problemlösungen erfolgt.

1.3 Organisatorischer Überblick

Im weiteren Verlauf beschreibt das zweite Kapitel zunächst genauer das Paradigma der komponentenorientierten Softwareentwicklung. Zunächst erfolgt eine gegenüberstellende Untersuchung der vielfältigen Komponentendefinitionen mit abschließender Formulierung einer ersten eigenen Begriffsbestimmung. Im Anschluß erfolgt eine Betrachtung der aus den Definitionen folgenden technischen und methodologischen Eigenschaften und Aspekte. Der Schwerpunkt liegt dabei auf den wesentlichen technisch orientierten Gesichtspunkten, in die sich das Themengebiet zergliedert.

Auf Grund der Breite des Themas und der entsprechend großen Anzahl diesbezüglicher praktischer Ausprägungen ist das dritte Kapitel komplett einer ausführlichen Übersicht bestehender Ansätze gewidmet. Da eine detaillierte Beschreibung aller Konzepte den Rahmen dieser — bzw. jeder — Arbeit sprengen würde, beschränkt sich diese Übersicht auf den Versuch, durch kurze Skizzen ein möglichst vollständiges Bild des Status quo zu vermitteln.

Kapitel vier identifiziert dedizierte Konzepte und Prinzipien, die sich für die Architektur einer grundlegenden Komponentensicht ergeben. Es werden separate Ebenen eines allgemeinen abstrakten Architekturbegriffs für komponentenorientierte Systeme abgegrenzt und in ihrer inhaltlichen Bandbreite eingehend untersucht. Verschiedene Alternativen werden gegenübergestellt und diskutiert.

Inhalt des fünften Kapitels ist die konzeptionelle Ausgestaltung ausgewählter Teilbereiche der hergeleiteten abstrakten Architekturkonzepte als vorstellbare Instanziierungen der im vorherigen Kapitel abgegrenzten Lösungsmengen, wobei sowohl die Modell- als auch die Beschreibungsebene einbezogen wird. Ein Schwerpunkt dieser Betrachtungen liegt dabei neben der Skizzierung eines beispielhaften konkreten Komponentenmodells in der Untersuchung formaler Spezifikationen auf Basis logikbasierter Semantiken.

Das sechste Kapitel enthält einen Vergleich der gewonnenen Konzepte mit einem konkret existierenden Ansatz. Als Beispiel wurden JavaBeans gewählt, die eine der bekanntesten und verbreitetsten Komponentenarchitekturen darstellen. Im Anschluß an deren Überblick, Analyse und Bewertung erfolgt die Untersuchung der Möglichkeiten sowie Grenzen einer Modifikation im Hinblick auf die hergeleiteten allgemeinen Architekturkonzepte.

Die Arbeit schließt in Kapitel sieben mit der Schlußbetrachtung, welche ein Resümee der gewonnenen Ergebnisse zieht und Ausblicke auf weiterführende Ansatzmöglichkeiten mit offenen Fragen und Problemen beinhaltet.

Kapitel 2

Das Softwareparadigma der Komponentenorientierung

Nachdem in der Einleitung ein erster Eindruck von Komponenten und dem damit verbundenen Paradigma vermittelt wurde, widmet sich dieses Kapitel konkreteren Inhalten. Dabei steht eine softwareorientierte Sicht von dementsprechenden Softwarekomponenten im Vordergrund

Die volle Bandbreite des Terminus "Komponente" wird dabei nur noch im ersten Abschnitt berücksichtigt. Dieser beschäftigt sich mit der näheren Kennzeichnung des Komponentenbegriffes an sich und einem Überblick gängiger Definitionen in deren gesamter Vielfalt z.T. konträrer Ausprägungen. Am Ende werden dann aber doch Parallelen identifiziert, und es wird die erste Formulierung einer Begriffsbestimmung im konkreten Sinne dieser Arbeit vorgenommen.

Innerhalb des zweiten Abschnittes fokussiert sich die Aufmerksamkeit dann voll auf eine Ideologie der Softwarebausteine, wobei zunächst die damit verbundenen Aspekte und optionalen Attribute auf technischer Ebene betrachtet werden. Komponentenorientierung ist jedoch keine rein technische Herausforderung. Für die erfolgreiche Umsetzung des Paradigmas ist vielmehr eine spezifisch abgestimmte Methodik der ganzheitlichen Softwareentwicklung unabdingbar, mit der sich der abschließende dritte Teilabschnitt befaßt.

2.1 Kennzeichnung des Komponentenbegriffes

Im folgenden Abschnitt soll der Komponentenbegriff eingehender gekennzeichnet werden. Zwecks dessen erfolgt zunächst eine Rekapitulation der Grundideen und im Anschluß eine nähere Betrachtung der vielfältigen Komponentendefinitionen. Am Ende der Sektion wird dann der im weiteren Ver-

lauf verwendete Komponentenbegriff motiviert und anhand einer endgültigen Definition festgemacht.

Grundidee der Komponentenorientierung ist die Erschaffung neuer Artefakte durch das Zusammensetzen vorgefertigter, separater Teilbausteine bzw. Komponenten, was auch als Komposition bezeichnet wird. Sind einzelne, benötigte Bausteine nicht verfügbar, so werden sie wiederum aus anderen, feineren Sub-Teilen aufgebaut oder, falls auch solche nicht vorhanden sind, von grundauf neu erschaffen.

Abhängig von der jeweiligen Blickrichtung verschiedener Ansatzpunkte findet sich diese Komponentenidee in vielfältigen Ausprägungen wieder. Aus der Sicht verteilter Systeme steht der Interaktionscharakter separater Einheiten im Vordergrund. Unter softwaretechnischen Aspekten ist vor allem die Vorgehensweise der Anwendungskomposition innerhalb von Rahmenwerken und Mustern interessant. Forschungsbereiche, die sich mit der Wiederverwendbarkeit von Software-Artefakten beschäftigen, haben den Komponentenbegriff traditionell in sehr breiter, allgemeiner Weise geprägt und zielen dabei auf Entwicklungsstrategien bzw. Vorgangsmodelle ab. Das Umfeld von "programing in the large" bzw. Megaprogramming ist unter anderem mit Koordinationssprachen besonders an dem "Klebstoff" (engl. *glue*) interessiert, der die Komponenten verbindet und zusammenhält.

Die verschiedenen Sichten führen zu gleichfalls verschiedenen Definitionen des Komponententerminus, welche im folgenden betrachtet werden.

2.1.1 Definitionen

Auf Grund der Bandbreite möglicher Auslegungen ist der Begriff der Komponente nicht leicht zu fassen. Um eine bessere Vorstellung darüber zu gewinnen, was der Terminus beinhaltet, sollen daher im folgenden kurz die wichtigsten Definitionen zusammenfassend skizziert werden. Diese unterteilen sich grob in zwei Bereiche [Sam97]. Zunächst können bestehende Artefakte als Komponenten aufgefasst werden. Diese unterteilen sich dann weiter nach Art und Umfang ihres Wertebereiches konkret passender Einheiten. Dagegen können Komponenten aber auch als komplett neue Abstraktionsform eingeführt werden, die auf keiner konkreten Entsprechung basiert.

Der auf bestehenden Artefakten basierende Komponentenbegriff kann zunächst sehr generisch gehalten werden, ohne auf spezifischere Details einzugehen. Holibaugh definiert zum Beispiel Komponenten als *"... logical part of a system or program."* [HP88]. Ebenso allgemein spricht Kain von Komponenten als *"... a product of the development process that exhibits certain qualitys of usability and separability."* [Kai96] und nach McGregor gilt: *"A component is any unit that provides a relatively independent piece that is*

used in combination with a number of components in different configurations." [MDK96].

Hooper und Chester identifizieren den Begriff der Komponente sehr stark mit dem der Wiederverwendung, denn sie nutzen ihn *"... to mean any type of software resources that may be reused (e.g., code modules, designs, requirements specification, domain knowledge, development experience, or documentation)"* [HC91]. Ganz ähnlich umfaßt auch der Ansatz innerhalb des NATO-Standards für die Entwicklung von wiederverwendbaren Software-Komponenten (Eine Software-Komponente ist *"... a software entity intended for reuse,... "*, welche *"... may be design, code or other product of the software development process."* [Bra94]) jegliche Artefakte.

Derartige generalistischen Definitionsansätze vermögen nur eine difuse Vorstellung von Komponenten zu vermitteln, schränken aber andereseits den gedanklichen Horizont nicht auf zu enge Muster ein. Andere Definitionen lassen konkretere, konzeptionelle Aspekte einfließen und sind daher anschaulicher, zielen aber im allgemeinen mehr auf den — wenn auch weit gefaßten — Bereich ausführbarer Programmeinheiten ab.

Eine der am meisten zitierten Definitionen ist die von Nierstrasz und Dami, welche Komponenten knapp aber ungleich aussagekräftiger als *"static abstractions with plugs"* beschreiben [ND95]. Eine Komponente ist danach zunächst eine *Abstraktion*, kapselt also ihren Inhalt durch gewisse umgebende Grenzen nach außen ab. Sie besitzt zudem mit den *Plugs* definierte Zugangspunkte zur wechselseitigen Interaktion. Der problematischste Aspekt dieser Definition ist wohl die Eigenschaft der *Statik*, welcher die Persistenz und wiederholte Einsatzfähigkeit solcher Artefakte ausdrücken soll. Ob Komponenten nun aber lediglich rein statische "Formen" ohne Zustand sein sollten, welche zur Laufzeit — konzeptionell getrennte — dynamische Instanzen generieren, ist umstritten.

Von Szyperski et al. stammt folgende Ausführung: *"A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be developed independently and is subject to composition by third parties."* [Müh97]. Interessant ist hierbei die Forderung nach einer expliziten Festlegung externer Beziehungen mit Vertragscharakter, da solche die Voraussetzung und Grundlage für Verifizierbarkeit von Konfigurationen darstellen.

Als weiteres Beispiel dieser Kategorie charaktrisiert Wegner, der Komponentenorientierung als Nachfolger der Objektorientierung sieht, Komponenten plastisch als *"... a generalization of objects that extends the primitives for realizing interaction to include distributed components, graphical user interfaces, databases, robots and virtual reality."* [Weg93], wobei hier die Eigenschaft der Interaktion eine zentrale Rolle einnimmt.

Mit steigendem Maß an Bestimmtheit werden Komponenten schließlich viel-

fach als spezifische Sprachkonzepte verstanden. Man muß bei solchen Sichten beachten, daß die Potentiale des Paradigmas hierbei nur zu einem Bruchteil ausgeschöpft werden können. Kalkuliert man dies jedoch bewußt ein, ergeben sich erstmals direkt praktikabele Ansätze.

Als Beispiel möge die folgende Aussage von Booch dienen, der eine Komponente mit konkretem Bezug auf die Programmiersprache ADA als "...a logical cohesive, loosely coupled module that denotes a single abstraction..." beschreibt [Boo87]. Weitere Vertreter dieser Klasse sind die vielfältigen kommerziellen Systeme mit unmittel- oder mittelbarem Komponentenbezug, zu denen etwa Sun "*Java Beans*", Oracle "*PowerObjects*" oder Informix "*DataBlades*" zu zählen sind. Solche konkreten Beispiele werden später noch genauer untersucht.

Gleichwohl aller Unterschiede der vorgestellten Definitionen läßt sich doch eine gemeinsame Linie erkennen, die eben je nach Schwerpunkt mehr oder weniger variiert. In weitgehendem Einvernehmen werden aber zumindest die folgenden Fakten allgemein als Eigenschaften von Komponenten akzeptiert:

- Komponenten sind Abstraktionen und kapseln ihre Inhalte.
- Komponenten haben definierte Zugangspunkte bzw. Schnittstellen.
- Komponenten sind eigenständige Artefakte, welche zur Kooperation innerhalb von Rahmenwerken bestimmt sind.

Innerhalb der vorliegenden Arbeit werden Komponenten als autonome Abstraktionseinheit betrachtet. Sie sollen sich an eigenständig ausführbaren Software- bzw. Code-Einheiten orientieren, dabei aber nicht auf bestimmte Konzepte — wie objektorientierte oder funktionale Paradigmen — einschränken, weshalb die Definition auch auf einem relativ abstrakten Niveau erfolgt und zunächst keine Einzelheiten der technischen Umsetzung enthält. Auf methodologischer Ebene soll den Komponenten ebenso eine semantische Abgeschlossenheit wie die fundamentale Intention zu zielgerichteter, wechselseitiger Kooperation innewohnen. Um die Entkopplung von Komponentenersteller und -verwender sowie die Wiederverwendbarkeit von Komponenten als fundamentale Charakterzüge einbetten zu können, soll eine auf geregelter, formaler Basis beruhende ganzheitliche Spezifikation mit Vertragscharakter schon in diese grundlegende Beschreibung eingehen.

Charakterisierung 2 (Komponenten) *Komponenten sind Abstraktionen autonomer, semantisch abgeschlossener aber flexibler Softwaresysteme, welche Funktionalitäten über Mengen eindeutiger, klar abgegrenzter Schnittstellen mit spezifizierendem Vetragscharakter sowohl bereitstellen als auch beziehen, um durch wechselseitige Interaktion innerhalb eines gemeinsamen, potentiell mehrstufigen Rahmens zielgerichtet und formal verifizierbar zu kooperieren.*

Nachdem nun der Komponentenbegriff feststeht, stellen sich vielfältige Fragen bezüglich der daraus resultierenden Aspekte, dessen technischer Umsetzung zum einen und der methodologischen Vorgehensweise bei konkreter Anwendung zum anderen. Die nächsten beiden Abschnitte geben Antwort.

2.2 Aspekte und Eigenschaften auf technischer Ebene

Komponenten sind eine attraktive Vorstellung innerhalb vieler Bereiche, und Gedankengebäude entsprechender abstrakter Szenarien lassen sich leicht ersinnen — was ein Grund für die Vielzahl existierender Definitionen sein mag. Die Überführung dieser Vorstellungen in die Realität bezüglich einer technischen Umsetzung der Vorgaben und des Ableitens geregelter methodologischer Vorgehensweisen ist ungleich schwerer und bislang nicht in vollem Umfang geglückt. In dieser und der folgenden Sektion werden die wesentlichen Aspekte und verschiedenen Attribute dieser Bereiche eingehend untersucht. Die Betrachtung wird dabei entsprechend der oben vorgenommenen Definition auf eine Auslegung von Komponenten als Softwarebausteine in Form von im weitesten Sinne ausführbarem Code eingeschränkt.

Die technische Ebene dieser Sektion betrifft detailliertere konzeptionelle und formale Gesichtspunkte eines Baukastenprinzipts sowie konkrete Mechanismen zur Umsetzung ablauffähiger Komponenten in realen Systemen. Es erfolgt dabei eine Gliederung in die drei Bereiche der Komponenten selber, deren Integration bzw. Interoperabilität sowie ihrer Formalisierung.

Um zu einer Grundlage zu gelangen, müssen zunächst konzeptionelle Überlegungen bezüglich struktureller Aspekte und verschiedener Ausprägungen der Komponenten selber angestellt werden. Nützliche Softwarebausteine ergeben sich dabei aber nur unter Beachtung gewisser Richtlinien. Um in den Genuß der Vorteile zu kommen, die das Paradigma verspricht, müssen die entsprechenden Techniken in vernünftige Bahnen gelenkt werden. Nicht jeder technisch mögliche Baustein ist sinnvoll, und ohne gewisse qualitative Eigenschaften ergibt sich kein Nutzen. Ein weiterer entscheidender Komponentenaspekt betrifft die Anpassungsfähigkeit. Flexible Einheiten führen zu verbesserter Wiederverwendbarkeit und helfen, die benötigte Anzahl der Varianten zu verringern.

Komponenten sollen wechselseitig interagieren, wozu eine Untersuchung von Aspekten der Integration und Interoperabilität dient. Diese führt zunächst zur Betrachtung des eigentlichen Basismechanismus — der Komposition —, welche sich in verschiedenartige Ausprägungen aufgliedert. Eng damit verzahnt sind Fragen der Interoperabilität, Nebenläufigkeit und Verteilung innerhalb potentiell heterogener, möglichst offener Komponenten- bzw. Sy-

stemlandschaften. Die Interaktion von Komponenten soll weiterhin der Kooperation in einem übergeordneten Rahmen dienen. Der Themenbereich wird aus diesem Grunde mit Aspekten von Komponetenkoordination und -frameworks beendet.

Die sichere und effiziente Verwendung von Komponenten setzt eine eindeutig geregelte, nachvollziehbare Basis voraus. Die Untersuchung der technischen Ebene schließt in diesem Sinne mit einer eingehenden Erörterung der Grundlagen formaler Spezifikationen und darauf begründeter Verifikationsmöglichkeiten.

2.2.1 Komponenten als Basiseinheiten

Die erste Untersektion beschäftigt sich mit den Komponenten selber, welche die Basiseinheiten des Paradigmas darstellen. Zunächst werden dabei Strukturaspekte behandelt. Es folgen Überlegungen zum grundlegenden Wesen der Einheiten bezüglich Art und Verhalten. Im weiteren Verlauf erfolgt eine Betrachtung der funktionalen Anforderungen der Größenordnungen sowie des Qualitätsaspektes von Komponenten. Den Abschluß bildet eine Betrachtung von Aspekten der Anpassbarkeit und Flexibilität von Komponenten.

2.2.1.1 Struktur, Art und Verhalten der Basiseinheiten

Komponenten stellen ein Strukturiererungsmittel für Softwaresysteme dar. In diesem Zusammenhang können statische und dynamische Sichten unterschieden werden. Weiterhin ist zu klären, ob Komponenten als eher aktiv oder passiv anzusehen sind. In jedem Fall sind sie jedoch kapselnde, klar abgegrenzte Abstraktionen mit formal eindeutig spezifizierten Schnittstellen als expliziten Zugangspunkten.

Abstraktion und Kapselung Der Komponentenbegriff ist eng mit den Sichten von Abstraktion und Kapselung verbunden. Komponenten sind in diesem Sinne für sich abgeschlossene (engl. *self contained*) Gebilde, welche von einer gewissen Art der Grenze (engl. *boundary*) umgeben sind. Dieser Grenzbegriff bildet die Grundlage der Strukturierung, der kontrollierten Interaktion und der Verifikation.

Abstraktion ist meistens nicht in uneingeschränkter Form möglich. Beispielsweise sind bei der Deklaration einer Klasse in objektorientierten Systemen Vererbungsbeziehungen in der Regel nur bei explizit benannten Superklassen — also statisch — möglich. In jedem Fall sollten Komponenten *Werte erster Klasse* (engl. *first-class-values*) darstellen, um eine Parametrisierung auf möglichst hohem Abstraktionsniveau zu ermöglichen.

Schnittstellen Schnittstellen (engl. *interfaces*) haben im Komponentenparadigma eine vielfältige Natur. Sie stellen zunächst die Abstraktionsgrenze der auf diese Weise gekapselten Bausteine dar. Bezüglich dieser Grenze bilden sie dann abstrakte Zugangspunkte auf die Funktionalität der Komponente und ermöglichen in diesem Zuge gleichzeitig deren inhaltliche Strukturierung. Wechselseitige Beziehungen gründen sich dann auf Schnittstellen und werden von den dahinterstehenden konkreten Komponenten entkoppelt.

Weiterhin manifestieren sich durch Schnittstellen die spezifischen Eigenschaften der zugehörigen Einheiten in eindeutiger, formaler Weise und können so als vertragliche Grundlage einer Beziehung zwischen Komponenten und deren Nutzern verwendet werden. Auf dieser Grundlage können potentielle Konfigurationen bezüglich formaler Korrektheit verifiziert werden, und eine sichere, teilweise automatisierbare Komposition wird möglich (siehe auch 2.2.3).

Schnittstellen treten bezüglich ihrer Inhalte und Darstellungsformen in breiter Vielfalt auf. Man kann zunächst grob zwischen Benutzer- und Programmierschnittstellen unterscheiden. Benutzerschnittstellen sind dabei zwar im Hinblick auf die Integration von Altlast-Systemen durchaus interessant, sollen aber im folgenden nicht weiter betrachtet werden. Bei den Programmierschnittstellen unterscheidet das ODP-Referenzmodell (*Reference Model of Open Distributed Processing*) [Com95] dann z.B. nach dem Grad der Abstraktion drei allgemeine Formen:

- *Operationale Schnittstellen* — Operationen abstrahieren dabei eine bestimmte Funktion oder einen Dienst der zugehörigen Komponente. Operationale Schnittstellen bieten eine Menge solcher Operationen an, die wie Prozeduren in imperativen Sprachen aufgerufen werden.
- *Streamschnittstellen* — Streams sind Folgen von Daten, die durch jeweils einen Namen und einen Typ beschrieben werden. Diese eignen sich besonders für Anwendungsbereiche mit sehr großem Datenaufkommen wie etwa dem Multimediabereich im Rahmen eines Erzeuger-Verbraucher-Modells (*engl. producer/consumer model*).
- *Signalschnittstellen* — Signale ermöglichen die denkbar elementarste Form der Interaktion mittels atomarer Aktionen zwischen einem initiierenden und einem reagierenden Objekt, was vor allem in ereignisbasierten Systemen zur Anwendung kommt.

Nach ihrer Darstellungsform kann man Schnittstellen daneben folgendermaßen klassifizieren:

- *Grafische Schnittstellen* — Die Beschreibung erfolgt in grafischer Form (z.B. durch Diagramme) und kann meist nicht automatisch verarbeitet werden.

- *Programmiersprachenabhängige Schnittstellen* — Die Beschreibung der Schnittstelle erfolgt in der verwendeten Programmiersprache.

- *Programmiersprachenunabhängige Schnittstellen* — Derartige Schnittstellen werden durch unabhängige Beschreibungssprachen wie *IDLs* (*interface definition languages*) beschrieben.

- *Selbstbeschreibende Schnittstellen* — Die Schnittstelle liegt dabei zunächst lediglich als abstrakte Einheit vor. Es sind dann Möglichkeiten vorhanden, die benötigten Informationen in einem interaktiven Prozeß entweder von der Schnittstelle selber oder ausgelagerten systemtechnischen Unterstützungsmechanismen dynamisch zu erfragen.

Inhaltlich enthalten Schnittstellen zumindest eine Spezifikation der durch die Komponenten zur Verfügung gestellten Funktionalität, wobei eine Überladung durch zu großen Umfang vermieden werden sollte, um Klarheit und Übersicht zu wahren. Zu diesem Zweck finden meist die Abstraktionen *Attribut*, *Ereignis* und *Methode* bzw. Mischformen mit wechselseitiger Emulation Verwendung. Wünschenswert ist darüber hinaus zum einen die Spezifikation der von einer Komponente vorausgesetzten Funktionalität sowie die Möglichkeit, semantische Verhaltensmuster zu spezifizieren.

Bezüglich der Verhältnisses zwischen Komponenten und Schnittstellen unterscheidet man schließlich noch Einfach- und Mehrfachschnittstellen. Während Einfachschnittstellen zu klareren Konzepten führen, ergeben sich durch Mehrfachschnittstellen die besseren Strukturierungsmöglichkeiten. Die Thematik geeigneter Komponentenschnittstellen wird später noch ausführlich behandelt.

Statik oder Dynamik Komponenten können innerhalb von Softwaresystemen verschiedene Rollen einnehmen. Eine Möglichkeit ist die Sicht auf Komponenten als rein statische Strukturierungseinheit. Solche statischen Gebilde wie Funktionen, Klassen, etc. haben eine persistente Existenz unabhängig vom umgebenden Kontext, wodurch die Möglichkeit der individuellen Manipulation und Speicherung besteht. Sie erzeugen zur Laufzeit dynamische Instanzen, welche in dieser Sicht dann keine Komponenten darstellen. Eine Komposition dynamischer Instanzen ist aber in hohem Maße erstrebenswert und stellt die zweite Sicht auf Komponenten dar. Dies zu erreichen ist nicht einfach, wird aber durch verschiedene Ansätze — z.B. die Kapselung eines Objekts innerhalb einer Klasse im Kontext der Objektorientierung — angestrebt.

Statik und Dynamik sind nicht nur im Bezug auf die Einheiten selbst wichtige Eigenschaften, sondern spielen auch eine Rolle bei der Komposition, welche sich im dynamischen Fall zur Laufzeit ändern kann. Auch statische

Komponenten implizieren dabei nicht zwangsläufig eine statische Komposition. Dynamische Komposition statischer — sowie dynamischer — Gebilde ist ein wichtiger Faktor in wechselhaften Umgebungen mit evolvierenden Anforderungen, da diese Eigenschaft Evolution der Software ermöglicht, ohne ein laufendes System zu unterbrechen.

Aktiv oder passiv Begibt man sich auf den heute am weitesten verbreiteten Standpunkt einer dynamischen Komponentensicht, stellt sich im Anschluß noch die Frage, ob Komponenten eher abwartend passiver Natur sind oder eigenständig aktive Einheiten darstellen. Im allgemeinen werden separate Komponenten für sich alleine als passiv betrachtet. Bei einer ganzheitlichen, mehrstufigen Sicht, bei der eine Menge kollaborierender Teilbausteine ihrerseits auf dem nächst höheren Abstraktionsniveau als Einzelbaustein betrachtet werden können und insbesondere die fertige Applikation ebenfalls eine "High-Level" Komponente darstellt, muß diese Sicht jedoch revidiert werden.

Skalierung Ein weiterer wichtiger Aspekt ist die Skalierbarkeit von Komponenten. Eine optimale Möglichkeit der Skalierung ist dabei die mehrschichtige, verschachtelte Verwendung der gleichen Abstraktionsmittel, welche zu einer beliebigen (technischen) Granularität führt. Skalierung ist ebenfalls für den Einsatz formaler Methoden bedeutsam, denn diese sind bei mehrstufigen Modellen besonders effektiv einsetzbar. Ohne Skalierung ist zudem nur eine eingeschränkte Wiederverwendbarkeit gegeben, denn einstufige Methoden, welche Komponenten direkt zu Applikationen zusammenfügen, führen zu relativ speziellen Bausteinen mit dementsprechend engem Einsatzbereich.

2.2.1.2 Funktionalität und Granularität

In Bezug auf die An- bzw. Verwendung einer Komponente stellt ihre *Funktionalität* die wichtigste Eigenschaft dar. Komponenten erbringen — indem sie eben "etwas tun" — gewisse Dienste, auf Grund derer sie von Dienstnehmern mit entsprechenden Dienstanforderungen selektiert und eingesetzt werden. Da also Dienste die entscheidenden Faktoren für das Zustandekommen wechselseitiger Nutzungsbeziehungen sind, kommt deren Beschaffenheit und Inhalten eine grundlegende Bedeutung zu.

Ob die Funktion einer Komponente den Anforderung des Verwenders genügt, ist bei komplexen Inhalten nicht immer leicht festzustellen. Bott unterscheidet in diesem Zusammenhang drei wichtige Eigenschaften der Funktionalität [BR92]:

- *Anwendbarkeit* — Anwendbarkeit (engl. *applicability*) drückt die Wahrscheinlichkeit einer erneuten Verwendung relativ zur spezifischen

Herkunftsdomäne aus. Komponenten sollten generell eine hohe Anwendbarkeit besitzen, diese kann jedoch auf gewisse Domänen beschränkt sein.

- *Generalität* — Generalität (engl. *generality*) meint die Breite möglicher Anwendungen einer Funktionalität, also deren *Generik*. Obwohl eine erhöhte Generalität zu verbesserter Anwendbarkeit führt, sollten Komponenten nicht durch zu generische Funktionalität überladen werden. Übertriebene Verallgemeinerung führt zu erschwertem Verständnis, größerem Lernaufwand und erhöhtem Ressourcenbedarf bei verminderter Performanz.

- *Vollständigkeit* — Vollständigkeit (engl. *completeness*) bezieht sich auf die Erwartungen innerhalb einer bestimmten Anwendungsdomäne. Die Funktionalität einer Komponente ist dann vollständig, wenn sie alle Aspekte enthält, die in semantischem Sinne für ihren vorherbestimmten Anwendungskontext unabdingbar sind. Komponenten sollten also eine sinnvolle, semantisch abgeschlossene Funktionalität erbringen.

Eine Komponente sollte also möglichst weitreichend anwendbar, semantisch vollständig und mäßig generisch sein. Ein weiterer Aspekt ist ihre *Granularität*, d.h. ihr quantitativer Umfang in funktionalem aber auch physikalischem Sinne.

Der funktionale Umfang einer Komponente sollte weder zu klein noch zu groß sein. Abgesehen von der Forderung nach Vollständigkeit führen funktional zu kleine Komponenten dazu, daß zwar ihre Anwendbarkeit steigt, daß aber andererseits ihre gesamte Anzahl groß wird, was verworrene, komplizierte Komponentensysteme zur Folge hat und hohe Ansprüche an die Performanz des unterliegenden Kompositionsmechanismus stellt. Bei zu großem Funktionsumfang hingegen sinkt die Anwendbarkeit, und die Entwicklung der Komponente ist — falls sie nicht selber aus Komponenten besteht — mit denselben softwaretechnischen Problemen konfrontiert, die das Paradigma lösen soll. Bei der physikalischen Größe, ausgedrückt in Bits und Bytes, erschweren zu große Einheiten u.U. die Vermarktung durch eine resultierende Einschränkung der Transportmöglichkeiten. Zumindest bei neuen Distributionskanälen wie dem Internet ist unter diesem Aspekt mit Problemen zu rechnen.

Spricht man über den Umfang von Funktionalität, so tritt wie bei vielen qualitativen Größen das Problem der Quantifizierung auf. Geeignete allgemeine Metriken bzw. Maßstäbe werden dabei kaum zu finden sein, denn was ein (zu) "großer" oder "kleiner" Funktionalitätsumfang ist, hängt praktisch immer vom Einzelfall ab. Zumindest kann man aber eine grobe Klassifikation versuchen. In diesem Zusammenhang unterscheidet [Gri98] mit

- "*feinkörnig*, passiv genutzt",
- "*mittelkörnig*, meist (inter)aktiv" und
- "*grobkörnig*, übergreifend aktiv koordinierend"

drei praxisbezogene Granularitätsmaße.

2.2.1.3 Qualität

Bei komponentenorientierter Softwareentwicklung werden vorgefertigte Bausteine verwendet. Diese können innerhalb eines Wiederverwendungsprogramms der eigenen Institution entstanden oder von Fremdanbietern eingekauft worden sein; in jedem Fall stammen sie aber im Gros der Fälle von Dritten. *Qualitative Eigenschaften* der verwendeten Komponenten fließen dabei unmittelbar in das eigene Produkt ein und stellen somit einen wichtigen Aspekt des Paradigmas dar.

Qualität umfaßt dabei sehr unterschiedliche Aspekte wie Performanz, Effizienz und Korrektheit der Bausteine. Stilistische Aspekte der internen Komponentenimplementation, die deren Lesbarkeit, Verständnis, Wartbarkeit, Modifizierbarkeit und Erweiterbarkeit betreffen, sind dagegen im Rahmen der globalen Komponentensicht unwichtig. Ein grundsätzliches Problem bei der Behandlung von Qualität ist deren *Maß*. Sinnvolle *Metriken* sind umstritten und Objekte laufender Forschung.

Der Grad möglicher Einflußnahme bezüglich qualitativer Aspekte hängt von der Natur der verwendeten Komponenten ab. Blackbox Komponenten verschließen sich grundsätzlich jeglicher Manipulation. Whitebox Komponenten sind zwar nicht derartig restriktiv, bei ihnen verursachen Änderungen — ohne diesbezügliche Dokumentation — jedoch einen Aufwand, der den Nettonutzen klar in Frage stellt.

Im Endeffekt muß man sich somit der gegebenen Qualität unterwerfen. Um so wichtiger ist dann ein umfassendes *Qualitätsmanagement* mit wirksamer *Qualitätssicherung* durch die Komponentenhersteller. Wie wirksam diese auch sein mag, eine umfassende formale Garantie der Korrektheit von Komponenten kann sie realistisch gesehen nicht geben, und so sollten Komponentensysteme in jedem Fall möglichst *fehlertolerant* sein. Eine Möglichkeit, wenigstens bestimmte qualitative Eigenschaften verläßlich zu machen, ist durch Zertifizierung gegeben [DK93].

2.2.1.4 Technische Anpassungsfähigkeit

Anpassungsfähigkeit bzw. Flexibilität von Komponenten ist eine wesentliche Voraussetzung für deren Wiederverwendbarkeit, da speziell im Fall komplexer Komponenten die detaillierten Umstände einer erneuten Verwendung in

fremden Systemen nicht komplett übersehen werden können oder so mannigfaltig sind, daß eine Berücksichtigung aller Möglichkeiten zu einer übergroßen Menge sehr ähnlicher Komponenten führen würde.

Der Vorgang des Abänderns einer Komponente zum Zweck aktueller Wiederverwendung wird *Anpassung* genannt. Man unterscheidet *Customizing*, für das zur Entwicklungszeit entsprechende Anpassungsmechanismen in die Komponente integriert wurden und *Modifikation* als Änderung, deren Notwendigkeit sich erst zum Zeitpunkt einer Wiederverwendung ergibt.

Neben den letzteren Formen der Anpassung ist deren Zeitpunkt ein weiteres Klassifikationskriterium. Man unterscheidet hier zum einen die Änderung zur *Entwicklungszeit* und zum anderen die Änderung zur *Laufzeit*. Laufzeitänderungen erscheinen besonders attraktiv, stellen aber gleichzeitig auch eine große technische Herausforderung dar, denn sie bedürfen einer Vielzahl an Metainformationen. Solche Daten, zu denen z.B. Typinformationen zählen, werden bei den meisten klassischen Sprachansätzen zur Kompilierzeit verworfen. In diesem Zusammenhang bieten interpretierte Sprachen die besseren Möglichkeiten. Optimal wären flexible Zwischenlösungen.

Customizing wird in einigen neueren Komponentensystemen wie JavaBeans aktiv unterstützt. Die variablen Merkmale werden dabei durch sog. *Properties* repräsentiert, welche zur möglichst einfachen und effektiven Konfiguration von Komponenten durch spezielle Werkzeuge — den *"Property-Editoren"* und *"Customizern"* — beeinflußt werden können.

Der Anpassungsprozeß wäre aus Sicht des Anwendungsentwicklers optimal, wenn er ohne sein Zutun automatisch und transparent ablaufen würde. Entsprechende Mechanismen der *Selbstadaption* sind Gegenstand laufender Forschung. Voraussetzung dafür ist wieder die Bereitstellung von Metainformationen zur Laufzeit. Verschiedene Ansätze benutzen dabei in Abhängigkeit der Grundtechniken des Komponentensystems entweder die Dienste dritter Parteien (z.B. Repositories) oder erfragen die Daten durch direkte Introspektion.

2.2.2 Integration und Interoperabilität

Die zweite Teilsektion beinhaltet Aspekte rund um die Integration der zuletzt besprochenen Bausteine zu komplexeren Gebilden, welche zunächst wieder Komponenten und auf oberster Ebene letztendlich dann fertige Anwendungen darstellen. Aus technischer Sicht ergeben sich hier die wohl größten Herausforderungen. So ist nicht nur die Möglichkeit einer *Komposition* bereitzustellen, welche simultane Ausführbarkeit und Kommunikation der beteiligten Einheiten ermöglicht, sondern auch durch *Interoperabilitätsmechanismen* ein gemeinsames semantisches Verständnis zu sichern, *Verteilung* zu ermöglichen und bei *Nebenläufigkeit* konsistenzerhaltend zu synchro-

nisieren. Daneben besteht die Notwendigkeit, den geforderten inhärenten Kooperationscharakter durch einen übergeordneten Rahmen zu manifestieren, was technisch durch *Frameworks* und *Koordinationssprachen* umgesetzt wird.

2.2.2.1 Komposition

Komposition meint den Vorgang des Zusammensetzens verschiedener Komponenten, um Voraussetzungen für deren Kooperation bzw. Interaktion zu schaffen. Komposition führt dabei durch das Zusammenfügen von Teilkomponenten wiederum zu einer neuen größeren Komponente — bzw. bei einstufigen Ansätzen direkt zu einer fertigen Applikation.

Die Komposition von Teilkomponenten hat vielfältige Aspekte und kann in sehr unterschiedlichen Formen erfolgen. Es muß gewährleistet werden, daß heterogene Softwarekomponenten zur gleichen Zeit ablaufen können, und daß diesen eine Möglichkeit zum wechselseitigen Austausch von Informationen gegeben ist.

Grob kann man zunächst zwischen *interner* und *externer* Komposition differenzieren [Sam97]. Man spricht dabei von interner Komposition, wenn homogene Komponenten innerhalb eines einzelnen Softwaresystems etwa durch das Linken von Objektcode integriert werden. Bei der externen Form liegen die Komponenten als unabhängig ausführbare Einheiten vor, wobei diese in einer darunterliegenden Umgebung integriert sein müssen, welche eine zur Komposition notwendige Infrastruktur bereitstellt. Innerhalb dieser Grundkategorien finden sich wiederum zahlreiche unterschiedliche Ausprägungen. Im folgenden werden kurz einige wichtige Formen der Komposition in Anlehnung an [Sam97] und [ND95] beschrieben.

Textuelle Komposition Das textuelle Zusammenführen von Quellcode zur späteren Kompilierung kann als eine primitive Form der Komposition aufgefaßt werden. Ein gewisser grundlegender Grad der Abstraktion ist durch *Makrotechniken* zu erreichen. Weitere Beispiele dieser Klasse sind *Templates* (z.B. in C++) und *Generics* (z.B. in ADA), bei denen eine Parametrisierung möglich ist und syntaktische sowie semantische Verifikation durchgeführt wird.

Funktionale Komposition *Funktionale Komposition* ist in fast allen Programmiersprachen zu finden. Komponenten sind in diesem Fall parametrisierte, funktionale Abstraktionen, die durch Aufrufe, welche den Parametern entsprechende Argumente zuweisen, aktiviert bzw. instanziert werden. Da Funktionen selber Werte erster Klasse sind — d.h. Funktionsparameter können selber wieder Funktionen sein — wird eine Komposition höherer

Ordnung möglich, welche in gewissem Rahmen automatisiert werden kann. Der funktionale Ansatz basiert formal auf dem λ-Kalkül, und es existieren hochentwickelte Typsysteme mit mächtigen transparenten Inferenzmechanismen und weitreichenden Verifizierungsmöglichkeiten für Korrektheit von Kompositionen. Durch RPC-Mechanismen[1] wird eine systemübergreifende Komposition autonomer Teilkomponenten ermöglicht.

Objektorientierte Komposition *Objektorientierte Komposition* basiert auf den Prinzipien von Vererbung bzw. Delegation. Hierbei wird der Inhalt bestehender Komponenten übernommen und erweitert, wobei die Kompatibilität gewahrt bleibt. Durch Polimorphismus und späte Bindung wird eine dynamische Form der Komposition ermöglicht. Verteilte Objekte machen die Kompositionsform systemübergreifend nutzbar, sind jedoch wegen ihrer Kontextgebundenheit nur schwierig umzusetzen.

Komposition durch Blackboards *Blackboard Komposition*, welche in klassischen verteilten Systemen zum Einsatz kommt, ist die indirekte Verbindung von Subsystemen über adressierbare, gemeinsam zugängliche Informationsablagen innerhalb eines globalen Raumes wie etwa gemeinsamer Speicher oder Kommunikationskanäle. Blackbord-Komposition erlaubt im Gegensatz zu *lokalen* Mechanismen wie der funktionalen Kompositionen die simultane Verbindung ganzer Komponentenmengen und stellt daher eine *globale* Methode dar. Den aus diesem Ansatz resultierenden extrem losen Kopplungen stehen starke Restriktionen bezüglich möglicher Korrektheitsprüfungen gegenüber.

Kompositionsmechanismen höherer Ebenen Neben den gesehenen, fundamentaleren Kompositionsmechanismen der letzten Abschnitte existieren eine Reihe von Ansätzen auf höheren Abstraktionsniveaus. Diese sind oft sehr spezialisiert und zum Teil auch nur visionäre Modelle. Aus diesem Grund sind sie innerhalb dieser Arbeit — mit Ausnahme der Objektmodelle — nicht unmittelbar relevant, stellen aber trotzdem interessante Ansätze dar und sollen deshalb der Vollständigkeit halber erwähnt werden.

- **Objektmodelle** — Objektmodelle liefern einheitliche Basismechanismen für systemübergreifende Integration (und daher auch Komposition) bezüglich verschiedener Maschinenarchitekturen, Plattformen und Programmiersprachen. Ein Beispiel für solche Modelle ist die *CORBA* Architektur der *OMG* (siehe weiter unten).

[1] Entfernter Funktionsauruf (engl. *remote procedure call*) [CDK94]

- **Verbunddokumente** — Das Konzept der Verbunddokumente beinhaltet auf höherem Abstraktionsniveau als Objektmodelle die separierte Zusammenfassung von Teilkomponenten der Anwendungsebene innerhalb eines gemeinsamen Kontextes — dem Dokument. Als Kompositionsmechanismen finden meist *Komponentenverweis* und *-einbettung* (engl. *linking and embedding*) innerhalb von *Containern* Verwendung. Die *OpenDoc* und *OLE* Architekturen (siehe weiter unten) sind Ausprägungen dieser Klasse.

- **Verbundapplikationen** — Bei den Verbundapplikationen liegt der Schwerpunkt wie bei Verbunddokumenten auf Komposition von Teilkomponenten der Anwendungsebene, welche nun aber die Erschaffung einer einzelnen, integrierten Applikation zum Ziel hat. Ein bekanntes Beispiel dieser Technik sind *Plugin-Mechanismen*, wie sie etwa im *Communicator* Browser von *Netscape* zum Einsatz kommen.

- **Integrierte Umgebungen** — Das Konzept der integrierten Umgebung bildet die Basis für komplette Softwaredomänen. Das Intervall möglicher Lösungen variiert zwischen den Extremen monolitischer Applikationen und unabhängiger Werkzeugsammlungen [Rei95]. Konkrete Ansätze bestehen aus einer Menge von Werkzeugen zusammen mit einem unterliegenden Kommunikations- und Integrationsmechanismus.

- **offene Plattformen** — Offene Plattformen entsprechen der visionären Vorstellung von Integrations- bzw. Kompositionsmechanismen über alle Grenzen der Heterogenität hinweg. Optimalerweise sollten die systemübergreifenden Aspekte von *Objektmodellen* mit der Vereinigung von Domänen bei *integrierten Umgebungen* verbunden werden.

Unabhängig von konkreten Kompositionsmechanismen ist es in jedem Fall wichtig, eine klare Differenzierung zwischen logischen und technischen Einheiten der Komposition vorzunehmen, da sonst der technische Rahmen leicht die konzeptionelle Ebene beeinflußt, was meist eine Einschränkung bedeutet. Leroy unterscheidet in diesem Zusammenhang zwischen konzeptioneller *Modularisierung* als logischem Strukturierungsvorgang und *separater Kompilation* als technischem Übersetzungsmechanismus [Ler94].

Vor allem bei den internen Kompositionsformen besteht traditionell meist ein scharfer Schnitt zwischen separater Betrachtung von Komponenten vor der Komposition und monolitischen, optimierten Softwaresystemen danach. Dieser Umstand ist im Hinblick auf Dynamik und Flexibilität hinderlich, und so werden Kompilierung und Bindung zunehmend nicht mehr als endgültige Verschmelzung aufgefaßt. Moderne Techniken erlauben in zunehmendem Maße fließende Übergänge zwischen technisch loser Kopplung und monoliti-

scher, optimierter Vereinigung sowie zwischen High- und Lowlevel Komponenten bezüglich derer Erscheinungsform. Auf diese Weise können in Abhängigkeit der Rahmenbedingungen jeweils geeignete Kompromisse zwischen Dynamik und Performanz von Programmsystemen zur Anwendung kommen.

2.2.2.2 Interoperation

Die Komposition von Komponenten schafft eine notwendige, aber nicht hinreichende Bedingung für erfolgreiche Kooperationsbeziehungen. Um sinnvoll zu kolaborieren, bedarf es neben der Möglichkeit simultaner Ausführung und dem Vorhandensein grundlegender Kommunikationsmechanismen eines gemeinsamen Verständnisses bezüglich logischer Abläufe und semantischer Inhalte eines wechselseitigen Informationsaustausches im Zuge der Zusammenarbeit. Eine oft zitierte anschauliche Parabel basiert auf dem Elektrizitätsnetz verschiedener Länder. Zum einen passen die Stecker elektrischer Geräte oft nicht in Steckdosen fremder Länder, was durch entsprechende Adapter behoben werden kann. Zum anderen reicht diese Maßnahme, die als Komposition gesehen werden kann, aber trotzdem nicht aus, falls das fremde Elektrizitätsnetz eine andere Spannung liefert. In diesem Fall muß dann zusätzlich eine Transformation erfolgen, was entweder durch das Gerät selber oder ein separates Bauteil erfolgen kann.

Die Lösung von Interoperabilitätsproblemen ist bei Softwaresystemen auf Grund der ungleich höheren Komplexität eine wesentlich schwierigere Aufgabe. Zum einen müssen die syntaktischen Unterschiede von Schnittstellen wie Namen von Methoden und Parametern überwunden werden. Zum anderen sind unterschiedliche semantische und funktionale Verhaltensweisen der beteiligten Partner anzugleichen.

Allgemein wird die Fähigkeit von Softwarekomponenten, trotz unterschiedlicher Implementationssprachen, Schnittstellen und Plattformen bzw. Modellabstraktionen zu kommunizieren und zu kooperieren, als *Interoperabilität* bezeichnet [Kon95].

Interoperabilitätsmechanismen Interoperabilitätsmechanismen unterstützen bzw. ermöglichen — nomen et omen — die Interoperation verschiedener Softwareeinheiten. Sie unterteilen sich dabei je nach Strategie und Ansatzpunkt in verschiedene Kategorien.

- **Überbrückung von Schnittstellen** — Die Strategie der *Schnittstellenüberbrückung* (engl. *interface bridging*) gleicht *angebotene* und *vorausgesetzte Schnittstellen* kooperationswilliger Partner durch Definition einer *Schnittstellentransformationssprache* (engl. *interface transformation language — ITL*) an. Die von der Programmiersprache

abhängige ITL beschreibt detailliert, wie die Inhalte der beteiligten Schnittstellen wechselseitig ineinander überführt werden können.

- **Standardisierung von Schnittstellen** — Schnittstellenstandardisierung (engl. *interface standardization*) vereinheitlicht die allgemeinen Schnittstellen von Diensten. Zu diesem Zweck finden sog. *Schnittstellenbeschreibungssprachen* (engl. *interface definition languages — IDL* Verwendung, welche unabhängig von konkreten Programmiersprachen sind. Die abstrakten Schnittstellenbeschreibungen können mit Hilfe spezieller Compiler zu sprachspezifischen Schnittstellenrümpfen (engl. *interface stubs*) übersetzt werden, welche Implementationen der entsprechenden Schnittstellen darstellen und in konkrete Programme eingebunden werden.

Je nach Ursprung unterscheidet man bei beiden Mechanismen noch *prozedurorientierte* Interoperabilität, welche am Punkt des Prozeduraufrufes ansetzt, und *objektorientierte* Interoperabilität, welche vom Punkt des Objektes ausgeht.

2.2.2.3 Nebenläufigkeit

Komponentenorientierung steht in engem Zusammenhang mit dem Begriff der *Nebenläufigkeit.* Allgemein heißen zwei Prozesse *nebenläufig* (engl. *concurrent*), falls sie voneinander unabhängig bearbeitet werden können, wobei der Sonderfall simultaner Aktivität zu einem definierten Zeitpunkt als *Parallelität* bezeichnet wird [Bur97].

Für die Kombination von Komponentenorientierung und Nebenläufigkeit gibt es verschiedene Motivationen. Zunächst steigert die Einführung von Nebenläufigkeit, welche in der realen Welt dem Regelfall enspricht, potentiell erheblich die Modellierungsmächtigkeit von Komponentensystemen, welche im Gegenzug wiederum den Entwicklungsprozeß nebenläufiger Systeme fördern. Zudem sind Komponentensysteme bei externer Komposition Konglomerate autonomer Einheiten, weshalb ihnen naturgemäß eine inhärente Nebenläufigkeit innewohnt. Schließlich sind durch Parallelisierung von Vorgängen beträchtliche Performanzgewinne zu erzielen.

Auf Grund der Orthogonalität von Nebenläufigkeit zu Paradigmen der Komposition — besonders der Objektorientierung — und Wiederverwendung generell ist deren Integration mit dem Komponentenansatz nicht so problemlos möglich, wie es zunächst den Anschein hat. Zwischen den Teilaspekten bestehen intensive Abhängigkeiten, und eine Vielzahl von Forschungsarbeiten beschäftigt sich mit deren möglichst harmonischen Kombination [Pap95].

Im Zusammenhang mit Komponenten kann man zwischen zwei Formen der Nebenläufigkeit unterscheiden [Sam97]:

- *Interne Nebenläufigkeit* (engl. *intraconcurrency*) meint die unabhängige Ausführung von Vorgängen innerhalb einer einzelnen Komponente.
- *Externe Nebenläufigkeit* (engl. *interconcurrency*) meint die unabhängige Ausführung verschiedener Komponenten.

2.2.2.4 Verteilung

Logische und physikalische *Verteilung* ist ein bedeutsamer Aspekt komponentenorientierter Softwareentwicklung, da entsprechende Komponentensysteme oft auch gleichzeitig *verteilte Systeme* darstellen. Durch den intensiven Trend der globalen Ausbreitung von Computernetzwerken gilt das besonders für verteilte Systeme im Sinne von physikalischer Vernetzung.

In diesem Zusammenhang definiert Coulouris et al. ein entsprechendes verteiltes System als *"a collection of automous computers linked by a network, with software designed to produce an integrated computing facility"* [CDK94].

Motivation für die Konzeption eines Komponentensystems als verteiltes System ist die Erweiterung dessen allgemeiner Fähigkeiten, der Flexibilität in Bezug auf inkrementelle Expansion und der Freiheit bei einer Auswahl möglicher Hersteller. Um Komponenten für unterschiedliche verteilte Plattformen benutzen, kaufen oder verkaufen zu können, muß zwischen ihnen eine Möglichkeit zur Kommunikation und Interoperabilität bestehen [Sam97].

2.2.2.5 Übergeordnete Koordination und Frameworks

Komponenten, wie sie im Kontext dieser Arbeit verstanden werden, erfüllen für sich alleine keine vollständige Funktion im Sinne einer Anwendung, sondern sind immer zu einer späteren Kollaboration in prinzipalen Konfigurationen bestimmt, welche wechselseitige Beziehungen festlegen und die globalen Abläufe koordinieren. Dieser Aspektkreis steht naturgemäß in engem Zusammenhang zur Komposition, zielt aber im Gegensatz zu der dort verwendeten dualistisch lokalen Sichtweise auf eine übergeordnete Ebene. Letztere übergeordnete Ebene manifestiert sich in *Rahmenwerken* bzw. *Frameworks.*

Technisch orientierte Frameworks legen Anordnung und Beziehungen der Komponenten untereinander fest und definieren auf diese Weise ein abstraktes Skelett — sprich: die konzeptionelle Architektur — des angestrebten Gesamtsystems. Neben dieser eher programmiertechnischen Auslegung wird der Begriff Framework auch rein informal im Kontext domänenorientierter, konzeptioneller Wissensrepräsentationen benutzt. Letzterer Aspekt soll hier nicht betrachtet werden.

Im Falle objektorientierter Systeme werden solche Frameworks meist durch Klassenhierarchien unter Verwendung von Vererbungsmechanismen verwirklicht, was eine Quellcodebasierung zur Folge hat. Allgemein kann man in

diesem Zusammenhang je nach Kapselung der Elemente zwischen *Whitebox-* und *Blackbox-Frameworks* unterscheiden. Whitebox-Lösungen führen oft zu intensiver Verflechtung der beteiligten Komponenten, weswegen in moderneren Ansätzen bevorzugt Blackbox-Frameworks zur Anwendung kommen, welche naturgemäß losere Kopplungen ermöglichen. Nützlich und sinnvoll sind in diesem Zusammenhang auch sogenannte *Hotspots* [Pre97], welche in ansonsten relativ statischen Rahmenwerken potentiell variable Aspekte kennzeichnen und Ausgangspunkte eines späteren Customisings darstellen.

Im Falle von Komponentenorientierung erscheinen Frameworks aus einem etwas anderen Blickwinkel. Hier handelt es sich nicht mehr rein um einen Rahmen wie etwa bei abstrakten — also nichtimplementierten — Klassen, denn die Einheiten des Frameworks — die Komponenten — sind schon in ihrer endgültigen Form einsatzbereit vorhanden und stellen für sich spezifizierte, abstrakte Teilfunktionalitäten dar. Man kann dann passender von einem *Kollaborationsrahmen* [Gri98] sprechen, welcher das Zusammenwirken dieser abstrakten Funktionalitäten repräsentiert und deren kompatiblen Austausch in einer konsistenten Weise ermöglicht.

Neben der Festlegung statischer Strukturen durch Frameworks ist ebenfalls die dynamische Koordination globaler Abläufen zu regeln. Diese Aufgabe kann durch *Koordinations-* oder *Skriptsprachen* erfolgen, welche manchmal auch als *glue* — also amorpher Klebstoff — charakterisiert werden, der die beteiligten Komponenten zusammenhält [NL97].

2.2.3 Formalisierung

Formale Methoden sind für das Komponentenparadigma, wie bereits mehrfach angesprochen, von potentiell großem Nutzen. Bei der Untersuchung komponentenorientierter Vorgehensweisen ergeben sich dabei entsprechende *Anwendungsfelder* durch den Wunsch nach eindeutiger — also formaler — Beschreibung der Bausteine, um zum einen mit systematischen Methoden deren korrekte Verwendung zu sichern und zum anderen deren Auffinden in geregelter Weise zu ermöglichen. Als vielversprechender Ansatz bietet sich hier die *Typisierung* von Komponenten an. In diesem Zusammenhang kommt man dann zu der Frage nach geeigneten *Spezifikationsformen* und entsprechenden *formalen Methoden.* Dabei stellt sich vor allem die Beschreibung dynamischer Aspekte wie der Anwendungssemantik von Teilbausteinen als wünschenswert heraus, erweist sich jedoch gleichsam als äußerst problematisch.

2.2.3.1 Anwendung formaler Methoden

Komponentenorientierte Systeme basieren auf wechselseitiger Kooperation im Rahmen von Kompositionen separater Teileinheiten. Da Bausteine in

diesem Szenario aus potentiell verschiedenen Entwicklungsprozessen — etwa bei Komponenten von Drittanbietern — stammen, wurden sie untereinander nicht notwendigerweise direkt auf eine wechselseitige Interaktion abgestimmt. Es ist daher unumgänglich, die Regelmäßigkeit wechselseitiger Komponentenbeziehungen im nachhinein zu prüfen und die Abstimmung gegebenenfalls indirekt durchzuführen. Zu diesem Zweck müssen die relevanten Umstände von Kooperationen explizit manifestiert werden, was einem allgemeinen Vertragsbegriff entspricht. Als Gegenstände solcher *Verträge* finden Komponentenschnittstellen Verwendung, die neben ihrer Funktion als explizite Zugangspunkte eine *Spezifikation* der Eigenschaften erlauben, welche im Zuge wechselseitiger Beziehungen zugesichert bzw. erwartet werden. Die Nutzung solcher Interfaces impliziert dann eine Anerkennung des dadurch zum Ausdruck gebrachten Vertrages.

Eine weitere Anwendung formaler Komponentenspezifikation ist das systematische, automatisierte Retreival geeigneter Bausteine aus einem potentiell sehr großen Pool, welcher speziell bei dem angedachten Szenario entsprechender komplexer Märkte entsteht. Geeignete Bausteine können dabei entweder Alternativen oder Ergänzungen vorhandener Einheiten bzw. deren Beschreibungen sein.

Ein Weg, Verträge in direkter Weise explizit zu machen bzw. allgemein die Eigenheiten von Komponenten zu abstrahieren, um dann mittels Klassifikation und Beziehungsbildung auf höherer Ebene formal argumentieren zu können, ist durch die Einführung von Typsystemen gegeben.

2.2.3.2 Typisierung

Um den Komponentenbegriff zu formalisieren, bietet sich das vor allem aus dem Bereich der Programmiersprachen bekannte Konzept der Typisierung an. Typen ermöglichen eine konsistente, geregelte Nutzung von Komponenten, indem sie die Formulierung von Bedingungen (*engl. constraints*) für eine solche Nutzung durch andere Bausteine des komponentenorientierten Systems erlauben. Weiter ist die Formalisierung Voraussetzung bei der Schaffung wohldefinierter und automatisierbarer Methoden zum Vergleichen und Klassifizieren von Komponenten.

Im Vordergrund steht dabei vor allem die möglichst weitgehende Formalisierung der Komponentenschnittstellen (auch als *strenge Typisierung* bezeichnet), um die *Typsicherheit* von Komponenteninteraktionen zu gewährleisten. Eine sinnvolle Grundlage von *Komponententypen* sind daher *Schnittstellentypen*, die strukturelle Aussagen über die von Komponenten angebotenen Schnittstellen beinhalten. Ein Schnittstellentyp kann dann als *Prädikat* verstanden werden, das eine Klasse von Schnittstellen kennzeichnet und sich wiederum aus einer Menge von *Untertypen* zusammensetzt, welche die Inhalte der gewählten Schnittstellenart kennzeichnen.

Formalisierung durch Typisierung stellt eine Basis für die Darstellung von *Beziehungen* (*engl. relations*) zwischen Komponenten dar. Solche Beziehungen sind vor allem in dynamischen Umgebungen, in denen eine ständige Evolution und Fluktuation von Komponenten stattfindet, von entscheidender Wichtigkeit, da sie auch unter diesen erschwerten Bedingungen eine geregelte, typsichere Komponentennutzung erlauben. Beziehungen können in Abhängigkeit von den Intentionen bei ihrer Schöpfung und der Mächtigkeit ihrer Aussagen stark unterschiedlichen Charakters sein. Im wesentlichen sind dabei zwei Formen wichtig:

- *Durch Instanzen definierte Beziehungen*, die explizit bestimmt werden müssen (meist semantischer Natur).
- *Durch Definitionsregeln definierte Beziehungen*, die eine automatische Verifikation mittels formaler Regeln ermöglichen.

Eine erste relevante Beziehung ist die *Konformität* zwischen Komponententypen, da sie für das Auffinden geeigneter bzw. alternativer Komponenten in komplexen Systemumgebungen entscheidend ist. Wird etwa der Komponententyp A benötigt, kann ebenso ein möglicherweise erweiterter Baustein des Typs B genutzt werden, falls dieser konform zu Typ A ist, da sich die beiden Einheiten unter diesen Bedingungen aus Nutzersicht nicht im Verhalten unterscheiden. Der Typ A wird dann als *Supertyp* des *Subtyps* B bezeichnet. Die Subtypisierung erlaubt den Aufbau von *Typhierarchien* als Strukturen, welche die Konformitätsbeziehungen widerspiegeln. Die Konformitätsbeziehung gehört zu den durch formale Regeln definierbaren Beziehungen, was automatisierte Typvergleiche ermöglicht. Letzteres ist Voraussetzung für die generelle Praxistauglichkeit dieses Konstruktes, da in komplexen Systemen eine große Anzahl solcher Konformitätsvergleiche anfallen.

Neben Konformitätsbeziehungen ist aus komponentenorientierter Sicht die Verträglichkeit von Komponententypen entscheidend, was als *Typinteroperabilität* bezeichnet werden könnte. Eine solche Typinteroperabilität drückt dann die Fähigkeit des Zusammenwirkens von Komponenten verschiedenen Typs bezüglich Struktur und Semantik aus.

Um die Handhabung von Typen und deren Beziehungen zu erleichtern, existieren informationstechnische Mechanismen wie *Typmanager*, welche eine automatisierte Verwaltung von Typhierarchien unterstützen.

Ein Problem besteht darin, daß herkömmliche Typsysteme oft nicht ausdrucksstark genug sind, um alle wichtigen Aspekte der Komponenteninteraktion zu erfassen. In diesem Fall bieten sich verschiedene Lösungsstrategien an. Nierstrasz [NT95] schlägt die zwei folgenden vor:

1. *Schnittstellenerweiterung* — Die Schnittstellen von Komponenten können durch *Constraints* — also formale Bedingungen — erweitert

werden[2], welche zusätzliche Forderungen und Erwartungen der beteiligten Kooperationspartner ausdrücken. Solche Constraints können zum Teil statisch durch Typsysteme und zum Teil dynamisch zur Laufzeit überprüft werden.

2. *Ausdrucksstärkere Typsysteme* — Ausdrucksstarke Typsysteme existieren vor allem im Bereich funktionaler Sprachen. Wie bereits erwähnt sind die Möglichkeiten formaler Verifikation bei verteilten Mechanismen wie der Blackboard-Komposition weitaus geringer. Bei der Typisierung objektorientierter Systeme ergeben sich kaum zu überwindende Probleme [FN94]. Dies liegt vor allem an der temporalen Sensitivität von Objektzuständen, an der rekursiven Semantik und am Subtypisierungskonzept des Paradigmas. Diesen Schwierigkeiten wird meist mit expliziter Typisierung oder existentieller Quantifikation begegnet, die aber dem Anwender eine extrem umständliche Handhabung aufbürdet und großen Berechnungsaufwand bedingt.

2.2.3.3 Spezifikation

Komponentenspezifikationen dienen der Darstellung von Informationen eines Komponentenmodells in abstrakter Weise und bilden die Basis der Typisierung. Verschiedene Formen von Komponentenspezifikationen unterscheiden sich durch ihre Ausdrucksmächtigkeit sowie die daraus resultierenden Möglichkeiten automatisierter Vergleiche einzelner Beschreibungen. Grundsätzlich betrachtet man drei Gruppen möglicher Komponentenspezifikationen:

Namensbasierte Spezifikation Bei namensbasierter Spezifikation werden Komponententypen nur durch ihre Namen beschrieben. Eine Erkennung und Überprüfung von Typbeziehungen ist hier nur explizit durch manuelle, administrative Festlegung möglich. Das Sicherstellen von Interoperabilität ist unter diesen Umständen problematisch, da ein globales Verständnis der Bezeichner ohne weitreichende Standardisierungsbemühungen kaum zu erreichen ist. Ein realistisches Anwendungsgebiet ist daher nur die Komponentennutzung in kleinen, administrativen Teilbereichen. Namensbasierte Spezifikationen kommen auf Grund ihrer einfachen Realisierbarkeit in den meisten verteilten Systemplattformen wie DCE, CORBA und ANSA zum Einsatz. Ein Beispiel sind die in DCE zur Kennzeichnung von Schnittstellen verwendeten UUID (*universal unique identifier*), die am Anfang einer jeden DCE-IDL-Beschreibung zu finden sind.

[2] siehe z.B. Meyer [Mey88]: "programming by contract" in der Programmiersprache *Eifel*

Strukturelle Spezifikation Bei struktureller Spezifikation enthalten die Beschreibungen lediglich Strukturaspekte der Komponenten in Form von herkömmlichen, zumeist operationalen Schnittstellenbeschreibungen mit den darin enthaltenen Signaturen. Beziehungen zwischen Komponenten können so — zumindest auf dieser strukturellen Ebene — auch automatisiert verifiziert werden. Schwachstelle struktureller Spezifikationsformen ist das Fehlen eindeutiger semantischer Aspekte. So ist es möglich, daß strukturgleiche Komponenten doch ein voneinander abweichendes Verhalten zeigen, und eine durch die Struktur implizierte scheinbare Konformität bei Berücksichtigung der Semantik keineswegs vorhanden ist.

Erweiterte Ansätze Für die Beschreibung dynamischer Aspekte von Komponenten, insbesondere im Bereich der Semantik, existieren noch einige erweiterte Ansätze. Typsysteme in Form von algebraischen Spezifikationen drücken lediglich statische Formen der Semantik aus, welche aber bei der Untersuchung rein statischer Konfigurationen zu aussagekräftigen Ergebnissen führen. Im Falle von Komponentensystemen sind jedoch auch *dynamische Aspekte* von Bedeutung, welche in dieser Form nicht formuliert werden können. So ist bei einer Komponente, die gewisse Methoden zur Verfügung stellt, nicht nur deren Ausprägung wichtig, sondern auch deren kausale Abhängigkeiten — also die mögliche Reihenfolge der Anwendung — von Bedeutung. Solche Formen des dynamischen Verhaltens lassen sich durch *denotionale-*, *algebraische-*, *operationale-* und *axiomatische Semantiken* ausdrücken. Problematisch ist dabei die hohe Komplexität solcher Spezifikationsformen, die oft nicht einmal entscheidbar sind. Es sind in diesem Bereich daher praktikable Kompromisse in Form von Zwischenlösungen anzustreben.

Komponentenbeschreibungen auf Basis formaler Semantiken eröffnen ein weites Feld möglicher Ansätze. Im folgenden sollen noch einige Beispiele aufgeführt werden, deren genauere Betrachtung an späterer Stelle folgt.

- *Automatenmodelle* — Einfache dynamische Abläufe wie z.B. Interaktionsprotokolle lassen sich gut durch endliche Automaten ausdrücken. Es existieren ausgereifte, effiziente Methoden für deren Handhabung, die Ausdrucksmächtigkeit ist jedoch relativ gering. Eine ausführliche Beschreibung findet sich z.B. in [HU93].

- *Temporallogik* — Temporallogik [MP92] erlaubt die Beschreibung kausaler Abhängigkeiten einer Menge atomarer Aktionen durch logische Formeln mit zusätzlichen temporalen Operatoren. Insbesondere ermöglichen derartige Logiken im Gegensatz zu Automatenmodellen oder Netzen eine aktive Vorgabe semantischer Eigenschaften, ohne überflüssige Zustände einzuführen.

- *Prozeßabstraktionen* — Prozesse stellen ein geeignetes Mittel zur formalen Modellierung von verteilten Interaktionen bei Nebenläufigkeit dar. Entsprechende Ansätze sind mit Erfolg auf Objektsysteme übertragen worden. Die Betrachtung von zumindest Teilaspekten komponentenorientierter Softwaresysteme durch eine Modellierung mittels Prozeßabstraktionen scheint deshalb ein vielversprechender Ansatz zu sein.

2.3 Integration auf methodologischer Ebene

Komponentenorientierung ist keine rein technische Herausforderung. Für die erfolgreiche Umsetzung des Paradigmas ist eine spezifisch abgestimmte Methodik der ganzheitlichen Softwareentwicklung unabdingbar, denn Komponententechniken liefern zwar notwendige Grundlagen, führen aber nicht zwangsläufig zu komponentenorientierten Systemen.

Dieser Gedanke komponentenorientierter Methodologie bildet die Grundlage für attraktive Visionen zukünftiger Softwareentwicklung. So sieht z.B. Nierstrasz für die Zukunft ein Szenario basierend auf Softwareinformationssystemen (SIS) voraus [ND95]. Diese SIS enthalten neben Beschreibungen konkreter Komponentenframeworks ein gebündeltes Anwendungswissen verschiedener Domänen und stellen in diesem Sinne Expertensysteme dar. Softwareentwicklung in idealisierter Form ist dann ein Prozeß, der auf informalem Dialog mit diesem Expertensystem basiert, in dessen Verlauf eine passende Domäne abgegrenzt wird, eine Bestimmung der Anforderungen basierend auf gespeicherten abstrakten Modellen und Richtlinien erfolgt, und schließlich konkrete Artefakte — d.h. vor allem Komponenten — mitsamt der notwendigen Anpassungen zu der gewünschten Applikation führen. Da von einem Wasserfallmodell abgesehen werden soll, beinhaltet das Szenario die Möglichkeit, einzelne Schritte später in erneuten Zyklen an veränderte Anforderungen anzupassen.

Das beschriebene Szenario ist Fiktion und wirft zahlreiche offene Fragen bezüglich Umsetzbarkeit bzw. Praktikabilität auf. Komponentenorientierte Softwareentwicklung in einer weniger automatisierten Form ist heute keine Illusion mehr. Ein solcher Ansatz betrifft alle Phasen klassischer Softwaretechnik. Von der Analyse über den Entwurf bis zur Implementation ist ein angepaßtes Vorgehen notwendig. Man kann in diesem Zusammenhang theoretisch zwei Formen der Softwareentwicklung unterscheiden. Zum einen die Entwicklung wiederverwendbarer Komponenten zur späteren Komposition von Anwendungen. Zum anderen die Entwicklung von Anwendungen als Kompositionen vorgefertigter Komponenten. Um wiederverwendbare Strukturen — vor allem Komponenten — zu identifizieren, ist zudem eine Analyse im Kontext der Anwendungsdomäne notwendig. Die folgenden Abschnitte

geben einen zusammenfassenden Überblick der Thematik. Im Anschluß daran werden Vorteile und Risiken dieser Vorgehensweise betrachtet.

2.3.1 Domain Engineering

Bevor Bausteine für die spätere Komposition von Softwaresystemen entwickelt werden können, die insbesondere auch in vorteilhafter Weise wiederverwendbar sind, bietet sich zunächst eine umfassende Untersuchung des betreffenden Anwendungsfeldes — der *Anwendungsdomäne* — an, da die Wahrscheinlichkeit für das wiederholte Auftreten von Objekten und Funktionalität bei Applikationen derselben Klasse besonders hoch ist. Die Untersuchung von Anwendungsfeldern zusammen mit der darauffolgenden Entwicklung identifizierter Komponenten bilden den Inhalt des *Domain-Engineerings.* Initialer Vorgang ist dabei die *Domänenanalyse* (engl. *Domain-Analysis*), welche mit Hilfe verschiedener Informationsquellen wie existierenden Applikationen oder Expertenwissen, die zusammengefaßt ausgewertet werden, konstante bzw. variable Teile identifiziert und für eine spätere, systematische Verwendung organisiert. Domänenanalyse zielt auf eine Wiederverwendung von Untersuchungen und Entwürfen, weshalb es sich bei besagten Teilen vornehmlich um einheitliche Architekturen bzw. generische Modelle in Form von Artefakten wie Definitionen, Domänen-, Anforderungs- sowie Architekturmodelle, Standards und spezifische Sprachen handelt. Die Ergebnisse der Analyse können für generative Ansätze mit entsprechenden *Applikationsgeneratoren* oder eben zur Identifikation von allgemeingültigen bzw. domänen- oder produktspezifischen Komponenten verwendet werden. Der nächste Schritt des Domain-Engineering ist die *Domänenimplementation* (engl. *domain implementatation*), welche die Entwicklung der Komponenten beinhaltet.

2.3.1.1 Komponentenentwicklung

Nachdem potentiell sinnvolle Komponenten im Rahmen der Domänenanalyse identifiziert wurden, können diese zu konkret verwendbaren Einheiten umgesetzt werden. Da es sich hierbei um Bausteine handelt, die dem Zweck späterer Wiederverwendung innerhalb einer Anwendungskomposition dienen sollen, spricht man bei diesem Vorgang auch von "Entwicklung *für* Wiederverwendung" (engl. *development for reuse*). Um diesen Wiederverwendungsaspekt möglichst wirkungsvoll in die Komponenten einzubringen, sind bei deren Entwurf, speziell bezüglich Funktionalität und Qualität (siehe 2.2.1.2, 2.2.1.3) spezifische Richtlinien einzuhalten. Allgemeine Entwurfsrichtlinien für Komponenten sind z.B. in [Mey95] zu finden. Bei Bausteinen, die aus einer Anwendungsentwicklung hervorgegangen sind, also nicht speziell als separate Komponenten entwickelt wurden, kann durch Generalisie-

rung in Form von Erweiterung bzw. Einengung des Anwendungsbereiches, Separierung durch Isolation und verbesserte Konfigurierbarkeit die Verwendbarkeit nachträglich verbessert werden. Allgemein sind erfolgreiche Komponenten meist das Ergebnis eines längeren Reifeprozesses über mehrfache Anwendungszyklen, ihre bloße Existenz reicht jedoch nicht aus. Der praktische Einsatz von Komponenten bedingt zunächst deren Lokalisierung in einer potentiell sehr umfangreichen Menge — etwa innerhalb eines lokalen-, domänenspezifischen- oder Referenz-Repositories für Softwarebausteine. Eine Klassifikation bzw. Typisierung von Komponenten ist für diesen Prozeß unabdingbar.

2.3.2 Anwendungsentwicklung

Die komponentenorientierte Entwicklung von Anwendungen basiert primär auf der Komposition vorgefertigter Bausteine und stellt somit ein "Entwickeln *mit* Wiederverwendung" (engl. *developing with reuse*) dar. Besonders wichtig ist, daß die Entwicklung von Anfang an mit expliziter Intention des Komponenteneinsatzes erfolgt. So darf insbesondere die Suche nach passenden Komponenten nicht erst nach der Analysephase einsetzen, sondern muß aktiv in diese eingehen, um nicht potentielle Gelegenheiten für den Einsatz von Komponenten auszulassen. Komponentenorientierung sollte ein integrierter Bestandteil des Software Lebenszyklus sein, wobei sich ein zyklisches softwaretechnisches Prozeßmodell wie das Spiralmodell von Boehm [Boe88] anbietet. Angepaßt an komponentenorientierte Anwendungsentwicklung unterscheidet [Sam97] dabei vier Phasen, die wiederholt durchlaufen werden:

1. Phase: Der erste Schritt umfaßt die Untersuchung der Anwendungsdomäne und das Retreival verschiedener möglicherweise passender Komponenten mitsamt notwendiger Rekonfigurationen der Lösungsstruktur als alternative Ansätze zur Neuimplementation.

2. Phase: Im zweiten Schritt wird eine Auswertung der alternativen Komponenten bezüglich notwendiger Änderungen und den damit verbundenen Risiken vorgenommen. Ziel ist das Treffen von Entscheidungen für die Verwendung bestimmter Bausteine.

3. Phase: Der dritte Schritt beinhaltet die Implementation einer Anwendung — oder eines Teiles — auf nächst höherer Ebene mittels der im zweiten Schritt bestimmten Komponenten.

4. Phase: Im vierten Schritt werden die Ergebnisse der vorangegangenen Schritte ausgewertet und ein erneuter Durchgang des Kreislaufes geplant. Hinzugekommene Bausteine werden bei Eignung in den Komponentenpool aufgenommen.

Ein integraler Bestandteil dieses Modells ist die *Evolution* von Softwaresystemen, welche oft durch veränderte Anforderungen im temporalen Verlauf geprägt sind. Evolution findet dabei einfach durch Austausch oder Veränderung betroffener Komponenten innerhalb eines erneuten Durchlaufes statt.

Es sei hier noch die enge Verzahnung der drei separat beschriebenen Teilaspekte Domain-Engineering, Komponenten- und Anwendungsentwicklung bemerkt. Daß Komponenten- und Anwendungsentwicklung eng einher gehen, ist anhand des Spiralmodells ersichtlich. Domain-Engineering seinerseits liefert in diesem Mosaik fundamentale Grundlagen für den Software-Lebenszyklus und profitiert im Gegenzug von dessen Ergebnissen und Erfahrungen, die dort wiederum zu neuen Erkenntnissen führen.

2.3.3 Vorteile und Risiken

Komponentenorientierte Softwareentwicklung führt idealerweise zu einer schnelleren, kostengünstigeren Herstellung verläßlicherer, besser wartbarer und darüber hinaus evolutionsfähiger Softwaresysteme. Die Anwendung des Komponentenparadigmas birgt neben diesen oft beschworenen Vorteilen aber auch einige Risiken und Probleme. Nierstrasz nennt in diesem Zusammenhang vier Punkte [ND95]:

1. Die Anwendung von Komponentenframeworks kann großen Lernaufwand beinhalten und zu einem NIH-Syndrom (Siehe 1.1.3) führen.

2. Die Neuentwicklung von Komponentenframeworks kann hohe Kosten bewirken und einen großen Zeitaufwand erfordern, was den Nettonutzen in Frage stellt

3. Die häufig angewandte Softwareentwicklung in Projekten basiert auf kurzen Planungszeiträumen und führt zu einem Interessenkonflikt mit den Langzeitzielen der Komponentenorientierung. Die Entscheidungsträger über den taktischen Einsatz konkreter Techniken sind zumeist nicht für langfristige strategische Planung zuständig und umgekehrt.

4. Komponentenframeworks bilden erst nach mehreren Anwendungszyklen eine stabile und verläßliche Entwicklungsbasis. Einerseits sollten unzuverlässige Frameworks nicht zur Anwendung kommen; andererseits kann ohne Anwendung aber keine Stabilität entstehen.

Kapitel 3

Übersicht bestehender Ansätze

Konkrete praktische Beispiele für Ansätze und Systeme im Kontext der Komponentenorientierung mit deren reichhaltigen Facetten finden sich in großer Menge. Viele davon zielen zwar nicht unmittelbar auf das Themengebiet ab, liefern aber wertvolle inhaltliche Beiträge und sind daher erwähnenswert. Da neue Komponentenarchitekturen die Erfahrungs- und Wissensbasis vorhandener Ansätze nutzen sollten, um aus deren Ergebnissen zu lernen, erscheint es sinnvoll, eine entsprechende Betrachtung in gebührlichem Umfang vorzunehmen. Aus diesem Grund widmet sich das dritte Kapitel ausschließlich diesem Inhalt.

Auf Grund der großen Anzahl der Beispiele ist eine tiefergehende Betrachtung einzelner Ausprägungen in diesem Rahmen allerdings nicht möglich, und es soll hier deshalb lediglich der Versuch unternommen werden, einen möglichst breiten Überblick des Status quo zu vermitteln. Die Beispiele sind im folgenden nach groben Kategorien geordnet, wobei der Beschreibung eines Repräsentanten jeweils die kurze Auflistung weiterer Beispiele folgt.

3.1 Standards

Im Bereich der Komponentenorientierung haben sich eine Reihe von Quasi-Standards etabliert. Es handelt sich dabei vornehmlich um Industriestandards, die mittlerweile eine weitreichende Verbreitung erreicht haben. Zu nennen sind dabei als wichtigste Vertreter Sun Microsystems *JavaBeans* Komponenten, OMG Standards rund um *CORBA* und *Geschäftsobjekte* sowie Microsofts *(D)COM* Modell mitsamt der zugehörigen Technologiefamilie.

3.1.1 Sun JavaBeans

Suns *JavaBeans* [Sun97a] Architektur ist ein Industriestandard für Softwarekomponenten, welcher auf der Sprache Java (siehe 3.2.1) basiert. JavaBeans Komponenten (kurz: *Beans*) sind abgeschlossene, wiederverwendbare Softwarebausteine mittlerer Granularität, welche vornehmlich zur visuellen Komposition mit Hilfe entsprechender Werkzeuge bestimmt sind und zumeist GUI Elemente — allerdings ohne entsprechendes Dokumentenmodell mit geeigneten Containern — darstellen. Die Basisarchitektur der JavaBeans sieht keine Verteilung vor; d.h Beans sind zur wechselseitigen Kombination innerhalb genau einer Java-Virtual-Machine vorgesehen. Trotzdem stehen den Beans Komponenten die grundsätzlichen Eigenschaften von Java mitsamt der Plattformunabhängigkeit innerhalb einer potentiell vernetzten Umgebung zur Verfügung.

JavaBeans sind zustandsbehaftete Laufzeiteinheiten, welche jeweils aus einer Menge von Java Objekten bestehen und durch entsprechende Klassen sowie eine Serialisierung der initialen Komponentenkonfiguration repräsentiert werden. Sie bedienen sich jedoch keinerlei sprachlicher Erweiterungen und können daher in beliebige Java-Umgebungen integriert werden.

Bei Beans muß zwischen Entwicklungszeit- und Laufzeitschnittstellen unterschieden werden, welche verschiedene APIs unterstützen. Während die ersten Informationen über alle Attribute, Methoden und Ereignisse enthalten, basieren die zweiten ausschließlich auf den äußersten Objekten der Komponenten, welche Zugriffsmethoden für Komponenteneigenschaften (*Properties*) und Ereignisse (*Events*) sowie allgemein zugreifbare Funktionalität bereitstellen. Beans sind daher Blackbox Einheiten, die ihre innere Implementation — bestehend aus einer beliebigen Objektmenge — kapseln.

Zur Spezifikation der äußeren (runtime) Schnittstelle sind lediglich eine Reihe von Design-Konventionen einzuhalten, welche als *Design Patterns* bezeichnet werden, jedoch im wesentlichen Namenskonventionen darstellen. Entwicklungswerkzeuge können nun die Struktur einer Bean untersuchen, anhand der Konventionen deren inhaltliches Wesen bestimmen und zur visuellen Manipulation bzw. Komposition aufarbeiten. Die besagten Eigenschaften umfassen dabei Properties, Events und Methoden.

Zur Untersuchung von Beans stellt die Architektur das Konzept der *Introspektion* zur Verfügung. Dieses basiert zum einen auf dem grundlegenden Java-Mechanismus der *Reflektion*, kombiniert mit den standardisierten Design-Patterns und zum anderen auf einer expliziten Deklaration von Metainformationen über separate Klassen (*Bean Information*).

Komponenten des JavaBean Standards können zur Entwicklungszeit in explizit vorgesehenem Rahmen individuell angepaßt werden. Dieser Rahmen wird durch *BeanProperties* (private Attribute mit entsprechenden Zugriffs-

methoden) bestimmt, welche mit Hilfe einfacher *Property-Editoren* oder individuell gesteuert durch komplexe, mitgelieferte *Customizer* manipuliert werden können.

Beans kommunizieren über einen Ereignismechanismus mit explizit über die Namenskonvention oder BeanInfo deklarierten *Events*. Grundsätzlich können daneben alle normalen Methoden einer bean-konformen Klasse in gewohnter Weise verwendet werden.

Ein weiteres Konzept der JavaBeans ist *Persistenz*. Angepaßte Komponenten können in ihrer individuellen Form langlebig gespeichert und später wieder- bzw. weiterverwendet werden. Die Persistenz beruht dabei auf der in Java enthaltenen Objektserialisierung.

Die Komposition von Beans kann entweder durch dynamische Objekt- oder Klassenkomposition erfolgen. Als entsprechende Kompositionsumgebung kommen drei Formen in Frage:

- Entwicklungsumgebungen mit expliziter Komposition durch Java Code.
- Skriptbasierte Umgebungen mit Komposition exportierter Methoden und Properties durch eine Skriptsprache.
- Prototyping Umgebungen mit Komposition durch Property Editoren, Customizer und weitere automatische Methoden.

Beans Komponenten sind — als normale Java Objekte — Werte erster Klasse und können somit als Parameter übergeben werden. Weiterhin können Kompositionen von Komponenten generell wieder zu neuen Beans führen. Aus diesen Tatsachen folgt die prinzipielle Mehrstufigkeit des Modells.

Da der JavaBeans Ansatz als besonders geeignet erscheint, die in der vorliegenden Arbeit beschriebenen Ideen anzuwenden, wird er später in Kapitel 6 noch eingehender betrachtet.

3.1.2 (D)COM/OLE/ActiveX und CORBA/BOF

Microsoft (D)COM/OLE/ActiveX Microsofts auf dem Objektmodell *COM* [Rog97] bzw. dessen verteilter Variante *DCOM* basierendes Dokumentenmodell *OLE* [Cha96] bilden zusammen mit der komponentenorientierten Erweiterung *ActiveX* [Cha96] eine über längere Zeit gewachsene Familie von Industriestandards. Die Tatsache der langen, schrittweisen Entwicklungsgeschichte resultiert dabei in einer leicht inkonsistenten Begriffsvielfalt. Innerhalb von DCOM werden mit durch *IDL* (Interface Definition Language)

und *ODL* (Object Definition Language) beschriebenen Schnittstellen aggregierbare Klassen deklariert, welche zur Laufzeit einmalig, allgemeingültig — damit auch zustandslos — initialisiert und genutzt werden können.

Weiterhin definiert die Architektur einige Basisdienste wie das *Compound Document Management* oder die *Com Services*. Neben den *Microsoft Foundation Classes (MFC)* existiert mit der *ActiveX Template Library (ATL)* ein Komponentenframework in Form von C++ Templates. Weil Komponenten in diesem Zusammenhang binärer Natur sind und daher echte Blackbox Abstraktionen darstellen, ist die Familie dieser Architekturmodelle eine gute Basis für komponentenorientierte Softwareentwicklung.

OMG CORBA/BOF Die OMG (*Object Management Group*) bildet ein Herstellerkonsortium, welches sich mit verschiedenen Standards im Kontext von Objektsystemen beschäftigt. Im Zusammenhang mit Komponententechniken ist zunächst die *CORBA* (*Common Object Request Broker Architecture*) Architektur [OMG96b, OPR96] als Bestandteil der *OMA* (*Object Management Architecture*) [OMG97] interessant, welche ein offenes verteiltes Objektsystem beschreibt. CORBA Objekte werden durch in *IDL* sprachunabhängig beschriebene Schnittstellen spezifiziert, welche auf verschiedene Zielsprachen abgebildet werden können. Wechselseitige Interaktion erfolgt über zentral vermittelnde *ORBs* (*Object Request Broker*), welche über Informationen aller beteiligter Objekte verfügen. Mit den resultierenden offenen, heterogenen Kompositionsmöglichkeiten für potentielle Komponentenarchitekturen stellt CORBA einen Basismechanismus mit Middleware Charakter dar.

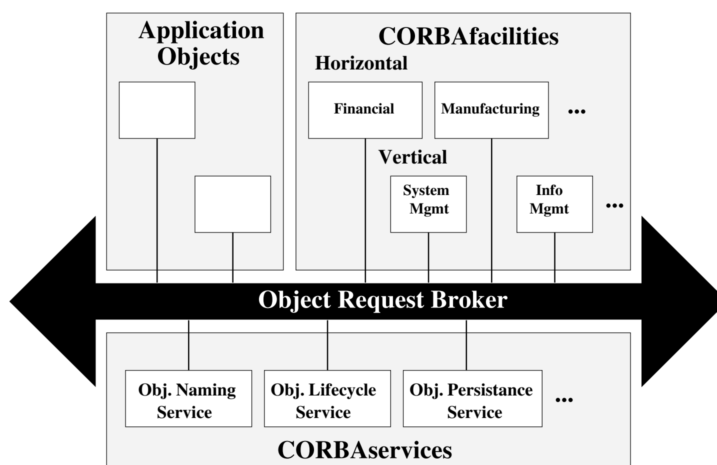


Abbildung 3.1: Object Managaement Architecture

Eine weitere Gruppe von Standards der OMG betrifft *Geschäftsobjekte*, welche sich in *allgemeine* sowie *domänenspezifische* Varianten gliedern und mit

den zugehörigen *PD* (*Presentation and Desktop*) Komponenten zu der auf die OMA aufbauenden *BAA* (*Business Application Architecture*) gehören. Es handelt sich dabei um ausführbare Softwarebausteine, die (Meta-)Daten Methoden und Business Rules spezifischer, fachlicher Komponenten kapseln und deren Architektur in der *BOF* (*Business Object Facility*) [OMG96a] beschrieben wird.

3.2 Programmiersprachen/Mechanismen

Programmiersprachen dienen der Implementation von komponentenorientierter Software. Sie werden dabei entweder nur für die Komponentenentwicklung oder zusätzlich bei deren Komposition zu endgültigen Anwendungen verwendet. Eine Sprache, die in diesem Bereich außergewöhnliches Potential aufweist, ist *Java*. Zudem existieren im Bereich der Komposition eine Reihe interessanter Kopplungsmechanismen auf Sprachniveau, die im zweiten Unterabschnitt betrachtet werden.

3.2.1 Java

Java [Sun97c] ist eine Programmiersprache mit interessanten Aspekten in Bezug auf Komponentenorientierung. Der objektorientierte, nebenläufige Sprachansatz auf Interpretationsbasis wurde von der Firma *Sun Microsystems* mit den Zielen der Portabilität, Performanz sowie Unkompliziertheit entwickelt. Die Schlüsselaspekte werden im folgenden erläutert:

- *Unkompliziertheit* — Um das Erlernen der Sprache zu erleichtern, orientiert sich Java syntaktisch und semantisch stark an C bzw. C++. Dadurch wird das vorhandene Wissen der mit diesen Sprachen vertrauten Programmierer ausgenutzt und die breite Akzeptanz von Java gefördert.

- *Objektorientierung* — Objektorientierung ist das zum heutigen Zeitpunkt wohl fortschrittlichste Paradigma der Softwareentwicklung und stellt die potentiell geeigneteste Basis zur Realisierung einzelner Komponenten dar. Java ist eine komplett objektorientierte Sprache mit Konzepten wie Modulen (sog. *Packages*), Klassen, Vererbung, Polymorphie und später Bindung. Die Vererbung ist dabei ausschließlich einfacher Natur, und Klassen sind nicht parametrisierbar. Während Objekte Werte erster Klasse darstellen, gilt dies nicht für Klassen.

- *Robustheit und Verläßlichkeit* — Bei immer komplexer werdenden Applikationen wird die Erstellung zuverlässiger Software zusehens schwieriger. Java enthält weitreichende Möglichkeiten zur Kompilier- und

Laufzeitprüfung von Programmen. Die Sprache ist stark typisiert und verfügt über einen Mechanismus zur dynamischen Freigabe nichtreferenzierter Speicherbereiche (*garbage collection*).

- *Sicherheit* — Gerade im Bezug auf die Verwendung fremder Komponenten spielt der Sicherheitsaspekt eine wichtige Rolle, da eine Gefahr nach Art des "trojanischen Pferdes" entsteht. Java verfügt über verschiedene Ansätze zur Erkennung potentiell gefährlicher Codefragmente und bietet auf diese Weise eine gewisse Schutzfunktion gegen destruktive Softwarebausteine.

- *Portierbarkeit und Architekturneutralität* — Java Programme werden in einen maschinenarchitektur-unabhängigen *Bytecode* übersetzt. Dieser ist in einer speziellen *Laufzeitumgebung* — der "virtuellen Maschine" (*virtual machine*) — durch einen speziellen Interpreter ausführbar. Laufzeitumgebung und Interpreter sind dabei für viele verschiedene Umgebungen implementiert. Überall dort ist dasselbe Bytecode-Programm unverändert einsetzbar. Zudem existiert eine umfangreiche, integrierte *Klassenbibliothek*, die z.B. GUI Unterstützung enthält. Dieser Umstand ist für Softwarekomponenten besonders attraktiv, da sich ihr potentieller Einsatzbereich dadurch im Vergleich zu maschinenspezifischem Objektcode enorm erweitert.

- *Performanz* — Obwohl es sich bei Java um eine interpretierte Sprache handelt, wurde versucht, die Performanz auf einem für die meisten Anwendungsfelder ausreichenden Niveau zu halten. Zum einen wird das durch die Tatsache erreicht, daß statt dem sonst oft üblichen textuellen Code mit dem verwendeten Bytecode eine effizientere Zwischenform gewählt wurde, zum anderen werden spezielle Beschleunigungsmechanismen wie die nachträgliche Teilkompilation zur Laufzeit (*runtime compilation*) verwendet.

- *Verteilung* — Die Sprache integriert die Möglichkeit mehrfacher *Threads* — also "leichtgewichtiger" Prozesse — sowie entsprechende Synchronistionsmechanismen. Entfernte Methodenaufrufe über Rechnergrenzen hinaus sind mit dem *RMI* (*remote method invocation*) ebenfalls in JAVA enthalten.

- *Dynamische Anpassbarkeit* — Trotz der strengen statischen Typprüfung zur Kompilierzeit besteht die Möglichkeit zur Erweiterung und Anpassung von JAVA-Programmen, indem Klassen zur Laufzeit dynamisch geladen und gebunden werden. Diese Mechanismen sind für die im Komponentenparadigma geforderten Eigenschaften des Customizings sowie der generellen Evolutionsfähigkeit von großem Nutzen.

Obwohl die Sprache objektorientierter Natur ist und somit in Bezug auf Komponentenorientierung die gleichen Unzulänglichkeiten wie andere derartige Ansätze aufweist, sind es vor allem die Plattformunabhängigkeit sowie die ausgeprägte Metaebene, die den Java Sprachansatz trotzdem zu einer interessanten Basis für komponentenorientierte Ansätze machen. Einer davon ist der im Abschnitt 3.1.1 vorgestellte JavaBeans Standard.

3.2.2 STL, Component Adaptors, Gluonen, SOFA/DCUP, Sina, Aspektorientierung, Demeter

STL Die *Standard Template Library (STL)* [MS96] ist eine Bibliothek leicht kombinierbarer C++ Container-Klassen und generischer Algorithmen, die in gewissem Sinne als Komponenten aufgefaßt werden können. Solche parametrisierbaren Templates lassen sich wegen ihrer Anpassungsfähigkeit effizient wiederverwenden. Die Hauptbestandteile von STL sind

- *Algorithm* als berechnende Prozedur, die auf verschiedenen Containern arbeiten kann,
- *Container* als Objekt, das andere Objekte beinhaltet und verwaltet,
- *Iterator* als Abstraktion eines algorithmischen Containerzugriffs, welche es Algorithmen ermöglicht, auf beliebigen Containern zu arbeiten,
- *Function-Object* als Klasse mit definiertem Funktionsaufruf Operator und
- *Adaptor*, welcher Komponenten kapselt, um eine geänderte Schnittstelle bereitzustellen.

STL wurde beim Treffen des ANSI/ISO C++ Standard Komitees am 14. Juli 1994 in den "draft standard" aufgenommen.

Component Adaptors Mit den *Component Adaptors* [YS95] existiert ein interessanter Kopplungsmechanismus für Komponenten auf dem Niveau dynamischer Aspekte der Anwendungssemantik. Ausgangspunkt ist die Verbesserung von Objektschnittstellen durch eine erweiterte Spezifikation, welche durch Protokolle beschriebene, sequentielle Abhängigkeiten der Verwendung beinhaltet. Auf diesen Schnittstellenspezifikationen basiert ein analog erweiterter Begriff der Kompatibilität mit entsprechenden Methoden zu deren Verifikation. Component Adaptors sind nun spezifische Bausteine, welche das Überbrücken von Unterschieden bei zwar funktional, jedoch nicht typkompatiblen Kooperationspartnern ermöglichen. Eine Spezifikation auf höherer Ebene (*interface mapping*) erlaubt darüber hinaus die automatische Generierung von Adaptern.

Gluonen Die in [Pin95] beschriebenen *Gluonen* stellen den Ansatz eines weiteren Kopplungs- bzw. Interoperationsmechanismus auf hohem semantischen Niveau dar. Basis sind dabei spezielle, zwischen den Teilkomponenten befindliche, wiederverwendbare Objekte — die Gluonen (siehe Abb.3.2) —, welche durch endliche Automaten beschriebene, standardisierte Kommunikationsprotokolle abstrahieren und als Mediatoren des Interaktionsprozeßes auftreten. Der Ansatz ermöglicht so eine dynamische Komposition von Komponenten zur Laufzeit.

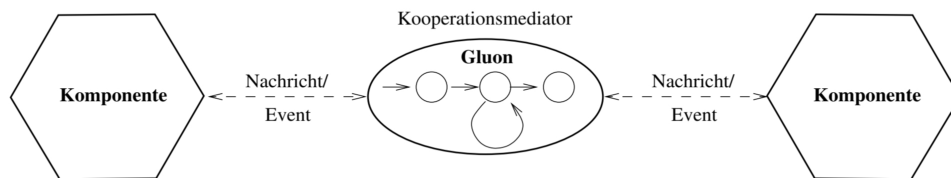


Abbildung 3.2: Gluon als Komponenetenmediator

SOFA/DCUP Die *SOFA* (*Software Appliances*) Architektur [PBJ98] beinhaltet zusammen mit dem zugehörigen SOFA Objektmodell und der Erweiterung *DCUP* (*Dynamic Component Updating*) [PBJ97] eine überschaubare Menge orthogonaler Abstraktionen, welche drei Bereiche betreffen: eine Basis für Electronic-Commerce, ein Komponentenmodell und die Unterstützung dynamischer Komponentenmodifikation in laufenden Applikationen. Letzterer Aspekt ist dabei Inhalt der DCUP Architektur, welche die transparente Adaption von Komponenten zur Laufzeit, nahtlose Zustandsübergänge der betreffenden Bausteine, die Erhaltung bestehender dynamischer Referenzen und eine integrierte Interaktion mit Komponentenprovidern ermöglicht. Das zugrundeliegende Modell betrachtet Komponenten als Frameworks objektorientierter Klassen mit permanenten und austauschbaren sowie orthogonal dazu funktionalen und kontrollierenden Teilen (siehe Abb.3.3).

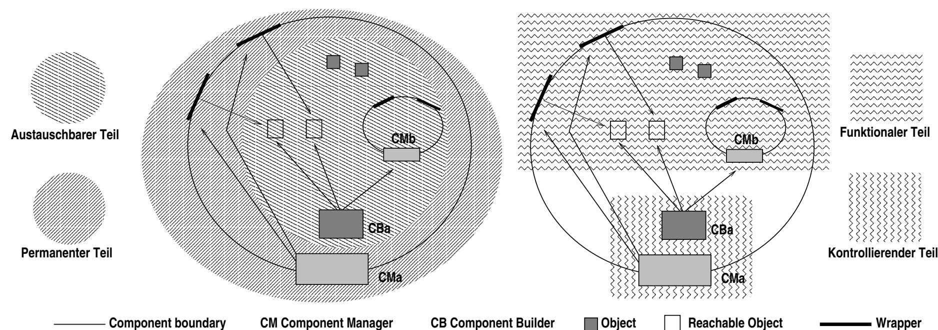


Abbildung 3.3: Struktur einer DCUP Komponente

Composition Filters / Sina *Sina* [Koo95] ist eine objektorientierte Sprache, die im *TRESE* Projekt [AT98] der Universität Twente entstanden ist und ein erweitertes Objektmodell umsetzt. Dieses *"composition filters object model"* bereichert das Prinzip der Objektorientierung um Filter, welche ein- und ausgehende Nachrichten von Objekten manipulieren. Hintergrund ist dabei die Abgrenzung und separate Komposition von *"concerns"*, welche für klar abgrenzbare Belange im Anwendungskontext stehen. Composition Filters stellen in diesem Sinne eine konkrete Umsetzung der *aspektorientierten Programmierung* (siehe unten) dar.

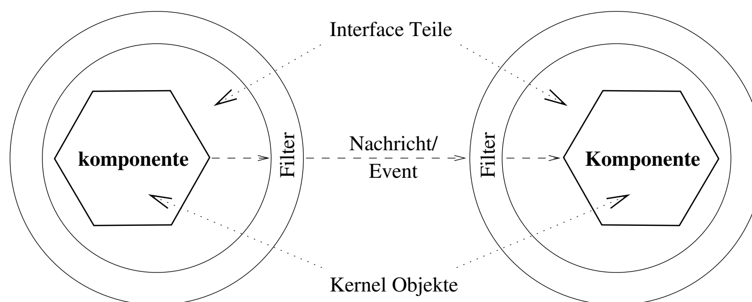


Abbildung 3.4: Composition Filters

Aspektorientierte Programmierung *Aspektorientierte Programmierung* [KIL+97, KLM+97] ist ein eigenständiges Paradigma der Softwareentwicklung, welches eine Dekomposition von Softwaresystemen in beliebige, den spezifischen Belangen angepaßte abstrakte Einheiten — den *"Aspekten"* — erlaubt. Aspekte orientieren sich dabei nicht an Codeeinheiten wie Subroutinen, Prozeduren oder Objekten, sondern stellen dazu orthogonale Einheiten dar, die durch einen speziellen Compiler — den *"Aspect Weaver"* — automatisch zu einer einzelnen separaten Einheit kombiniert werden können. Aspektorientierte Programmierung erlaubt es Ingenieuren auf diese Weise, die für ein System wesentlichen Aspekte separat zu behandeln und entsprechende Teillösungen zu erstellen, die später integriert werden können.

Demeter Der *Demeter* [Lie96] Ansatz setzt die Prinzipien der aspektorientierten Programmierung in einem konkreten Ansatz um. Die auf der abgeleiteten "*adaptiven Programmierung*" beruhende Demeter-Methode zielt dementsprechend auf eine möglichst weitgehende Separierung von Anwendungslogik und statischer Programmstruktur, was auch als *"structure shy programming"* bezeichnet wird. Zu diesem Zweck finden eigenständige modulare Einheiten (*Traversal Strategies*) zur Abstraktion der Traversion von Objekten durch Klassenhierarchien Verwendung. Solche Spezifikationen höherer Ordnung müssen auf entsprechende objektorientierte Zielsprachen wie Java oder C++ abgebildet werden.

3.3 Koordinationssprachen

Koordinationssprachen dienen dem Zweck, die Entwicklung komplexer Softwaresysteme zu erleichtern, indem sie die spezifischen Koordinationsaspekte einer Anwendung unterstützen. Der Begriff Koordination wird dabei verwendet, um einen speziellen Entwicklungsprozeß von Softwaresystemen zu umschreiben, der im Zusammenfügen unabhängiger, aktiver Teile besteht (etwa Prozesse, Tasks oder Threads). Koordinationssprachen bilden in diesem Sinne den "Klebstoff", der diese unabhängigen Teilen zusammenhält und je nach Bedarf wechselseitige Kommunikation und Synchronisation ermöglicht. Als Beispiele dieser Gattung sollen *Darwin*, *Linda* und *Manifold* dienen.

3.3.1 Darwin

Die Koordinations- bzw. Konfigurationssprache *Darwin* [MNSJ95] dient der Spezifikation von Systemen als Menge von Komponenten und deren wechselseitigen Beziehungen. Darwin wurde dabei als allgemeingültiger Ansatz in Hinblick auf die Unterstützung generischer Client/Server Architekturen entworfen. Bezüglich des Spezifikationsmediums sind sowohl grafische als auch textuelle Repräsentationen möglich.

Der Hauptnutzen von Darwin liegt darin, daß die Sprache es Systemarchitekten erlaubt, Verbundkomponenten beliebiger Granularität zu konstruieren, die zum einen aus atomaren Basiskomponenten und zum anderen wiederum aus Verbundkomponenten aufgebaut sein können. Das resultierende System ist dann als Hierarchie verschachtelter Verbundkomponenten strukturiert, wobei sich zur Laufzeit eine Menge nebenläufiger und potentiell verteilter konkreter Instanzen der Basiskomponenten ergibt.

Darwin Komponenten sind Blackbox-Einheiten, die sowohl durch deren selber bereitgestellte, als auch durch die von anderen Komponenten vorausgesetzten Dienste beschrieben werden. Verbundkomponenten ihrerseits werden durch die Schnittstellen enthaltener Teilkomponenten und deren Bindungen deklariert. Bindungen stellen dabei Beziehungen zwischen bereitgestellten und vorausgesetzten Diensten einzelner Komponenten dar. Solche Dienste werden in den Komponentenschnittstellen spezifiziert, wobei sie bezüglich der spezifischen Komponente lokalen Charakters sind. Durch diese Art der Kontextunabhängigkeit wird die Wiederverwendung gefördert und Austausch bzw. Wartung erleichtert. Es können weiterhin Parameter von Komponententypen bestimmt werden, wobei diese wiederum Komponenten sein können (`component`). Darwin Komponenten sind in diesem Sinne also Werte erster Klasse. Schnittstellen von Verbundkomponenten enthalten die Typinstanziierungen (`inst`) der enthaltenen Teilkomponenten und die Bindungen derer Dienste (`bind`). Der auf diese Weise realisierte Kompositionsmechanismus ist rein statischer Natur. Abbildung 3.5 veranschaulicht

```
Component Filter(int freq) {
  require in;
  provide out;
}
Component LoPass (int freq) {
  require in;
  provide out;
}
Component BanFilter(component(Filter) LoPass, int loFreq,...) {
  require in;
  provide out;
  ints lo: LoPass(loFreq);
  ...
}
Component SBF(int loFreq, int hiFreq) {
  require in;
  provide out;
  inst lo: LoPass(loFreq);
  inst hi: HiPass(hiFreq);
  bind lo.in -- in;
  bind hi.in -- lo.out;
  bind out -- hi.out;
}
```

Abbildung 3.5: Darwin Komponenten

noch einmal die angesprochenen Konzepte durch Beispiele von Darwin Komponenten mit entsprechenden Beziehungen.

Die Semantik von Darwin kann durch formale Methoden beschrieben werden. Zu diesem Zweck wird ein *“Higher-Order Pi-Calculus”* verwendet. Eine Beschreibung der Methode findet sich in [RE96].

3.3.2 Linda und Manifold

Linda Die Koordinationssprache *Linda* [ACG86] beruht auf einem asynchronen, assoziativen Kommunikationsmechanismus, dessen Basis ein öffentlich geteilter, globaler Raum ist. Dieser Raum wird *Tupelraum* genannt und besteht aus einer Multimenge von namentlich referenzierbaren, aktiven und passiven Tupeln, welche zur Abstraktion von Prozessen und Daten verwendet werden. Linda enthält zwar keinen direkten Komponentenbegriff, der Tupelraum kann jedoch als generische Verbindungskomponente betrachtet werden, durch die eine Integration und Koordination von Basiskomponenten fremder Programmiersprachen ermöglicht wird. Die zur Interaktion dienende Bindung von Basiskomponenten innerhalb des Tupelraumes geschieht über Mustererkennung von Tupelnamen, was zu sehr losen und flexiblen

Beziehungen führt.

Manifold *Manifold* [AHS93] ist eine Koordinationssprache, welche zur Regulierung der Kommunikation zwischen unabhängigen, kooperierenden Prozessen in massiv parallelen oder verteilten Anwendungen dient. Das fundamentale Prinzip von Manifold ist die vollständige Trennung von Anwendungslogik und Kommunikation. Das bedeutet zum einen, daß die Prozesse der Anwendungslogik in Manifold nichts von ihrer eigenen Kommunikation mit anderen Prozessen wissen, und zum anderen, daß koordinierende Prozesse zwar die Kommunikation innerhalb einer Prozeßmenge regeln, jedoch kein Wissen über die durch sie gesteuerte Anwendungslogik besitzen. Dieses Prinzip führt zu flexiblerer Software, die aus besser wiederverwendbareren Komponenten besteht und offene Systeme unterstützt.

3.4 Formale Sprachen

Formale Sprachen dienen der exakten Spezifikation komponentenorientierter Systeme und definieren dabei deren Semantik in eindeutiger Weise. Insbesondere ermöglichen sie auch systematische Methoden zur Prüfung bestimmter Eigenschaften wie z.B. eines Konsistenz- oder Korrektheitsbegriffs für entsprechende Systeme. Da zum einen keine verbreiteten formalen Sprachen für Komponentensysteme existieren und zum anderen eine enge Verwandtschaft zwischen Komponenten- und Objektorientierung besteht, werden im folgenden mit *OPUS* und *HOP* zwei Kalküle letzterer Gattung betrachtet.

3.4.1 OPUS

Das *OPUS Kalkül (object **o**riented **p**rogramming calcul**us**)* [MMS94, MMS95] wurde als elementarer Ansatz zur formalen Beschreibung objektorientierter Konzepte entworfen. Es modelliert unmittelbar die wesentlichen Eigenschaften objektorientierter Programmierung wie Objekte, Kapselung Kommunikation über parametrisierte Nachrichten und schrittweise Verfeinerung. Ungleich vieler anderer Versuche formaler objektorientierter Beschreibungen, orientiert sich der Ansatz nicht direkt am Lambda-Kalkül.

OPUS Objekte bestehen aus einem *öffentlichen* und einem *privaten* Teil, notiert als [`öffentlich` | `privat`]. Beide Teile enthalten Methoden, wobei Records von Methoden verwendet werden, um komplexere Objekte betrachten zu können. Nur auf die öffentlichen Methoden kann von außen durch Nachrichten zugegriffen werden. Eine Nachricht `N` mit Argument `A` an Ausdruck `E` wird dabei durch (`E N:A`) ausgedrückt. Methoden werden durch ein λ-Symbol, gefolgt von ihrem Namen, einem Gleichheitszeichen und schließlich einem Ausdruck geschrieben. Bei einem Methodenaufruf werden vor der

Auswertung die ungebundenen Variablen im Methodenrumpf an Nachrichtenargumente oder private Attribute gebunden. Konstante Methoden dienen der Repräsentation von Objektzuständen, welche ansonsten nicht in OPUS vorkommen. Bei ihnen folgt dem Gleichheitszeichen der Notation ein statischer Wert. Schrittweise Modifikation von Objekten durch eine Art von Vererbung wird durch den + Operator erreicht. Die Modifikation eines Ausdrucks P durch einen zweiten Ausdruck M lautet dann (P + M). Die Methoden bleiben dabei separiert, M kann nicht auf den privaten Teil von P zugreifen, und bei eingehenden Nachrichten wird die entsprechende Methode von rechts nach links gesucht (Overwriting von Methoden).

Abbildung 3.6 zeigt ein Beispiel, in dem ein zweidimensionaler Punkt durch Vererbung zu einem dreidimensionalen Punkt verfeinert wird. Die anschließende argumentfreie Nachricht ruft die `gety` Methode auf, muß aber trotzdem das leere Objekt [|] als Argument enthalten.

```
POINT := [ λgetx=x λgety=y | x=1 y=2 ]
MODIFIER := [ λgetz=z | z=3]
3DPOINT := ( POINT + MODIFIER )

3DPOINT gety:[|]
= ( POINT + MODIFIER ) gety:[|]
⇒ POINT gety:[|]
⇒ [ x=1 y=2 | ] y:[|]
⇒ 2
```

Abbildung 3.6: Objekte, Vererbung und Methodenaufruf in OPUS

3.4.2 HOP

HOP [Dam97] ist eine funktionale Sprache mit namensbasierter Interaktion zwischen Softwarekomponenten. Funktionsparameter werden dabei nicht wie in anderen funktionalen Ansätzen üblich anhand ihrer Position identifiziert, sondern ergeben sich aus ihrem Namen. Auf diese Weise wird es möglich, mit verschiedenen Formen der Modularität zu experimentieren und ein Rahmenwerk für objektorientierte Konstrukte bereitzustellen. Der Kern von HOP ist das Lambda-N Kalkül, welches eine Erweiterung des Lambda-Kalküls darstellt. Durch Verwendung von Parameternamen statt -positionen sind Ausdrücke wie `λ x y→x+y` und `λ a b→a+b`, die in traditionellen funktionalen Sprachen identisch wären, nun verschieden. Auf der anderen Seite sind Ausdrücke wie `λ(x y)→x+y` und `λ(y x)→x+y` bezüglich des Lambda-N Kalküls äquivalent. Der Anwendungsbegriff spaltet sich unter diesen Umständen in zwei Operationen auf: das dynamische Binden von Parametern (*bind*) und dessen expliziter Beendigung mit anschließender Auswertung (*close*).

3.5 Werkzeuge/Umgebungen

Mit dem neuen Medium komponentenorientierter Programmierung gehen gleichfalls neue Entwicklungsmethodologien und Werkzeuge einher, welche im Zuge des einhergehenden Trends zu visuellen Konzepten der Programmerstellung oft stark grafisch geprägt sind. Zwei Beispiele relativ visionärer, komponentenorientierter Entwicklungsumgebungen sind *Vista* und *FACE*.

3.5.1 Vista

Vista [Mey95] ist ein generisches, visuelles Kompositionswerkzeug, welches als Teil des übergeordneten *ITHACA* Projektes zur Schaffung einer kompletten objektorientierten Umgebung zur Anwendungsentwicklung entstanden ist. Vista unterscheidet sich von anderen visuellen Kompositionswerkzeugen durch seine Domänenneutralität, denn es kann an verschiedene Komponentenmengen angepaßt werden, indem deren Schnittstellen, Kompatibilitätsregeln, Kompositionsmechanismen und visuelle Abbildungen spezifiziert werden.

Eine Vista Komponente besteht aus Verhalten und Präsentation (*model* und *view*) sowie einer Menge ein-/ausgehender *Ports*, welche für benötigte-/ bereitgestellte Dienste stehen. Anwendungen werden durch die Verbindung solcher Ports beschrieben, wobei die Ports kompatibel sein müssen. Die Art der Kompatibilität kann dabei wechseln und wird mittels uninterpretierter Typen bestimmt, die jeweils im Einzelfall durch die aktuelle Komponentenmenge bestimmt werden. Aus diesem Grund ist es auch möglich, die Ports wieder mit Komponenten zu assoziieren, was jedoch für Vista transparent bleibt. Vista selber unterstützt Mehrstufigkeit durch die Möglichkeit, Kompositionen mit einer Präsentation auszustatten und spezifizierte Ports als neue Komponente zu kapseln.

Solche Kapselungen sind in Vista generell schwacher Natur. Aus Sicht des Vista Systems sind Komponenten Blackbox Einheiten, die lediglich ihre Ports spezifizieren. Aus Sicht der Komponentenmenge ist dies jedoch nicht zwangsläufig der Fall, denn es müssen nicht alle verwendeten Schnittstellenarten bekanntgegeben werden.

Bezüglich ihrer Granularität sind Vista Komponenten nicht eingeschränkt. Die unterliegende Realisierung stützt sich jedoch auf C++ Objekte, welche Verhalten Präsentation und Ports repräsentieren.

Der Bindungsmechanismus von Vista Komponenten wird durch die Komponentenmenge bestimmt. Konkret müssen dazu die abstrakten Klassen *port* und *link* implementiert werden, die den “Klebstoff” realisieren. Vista Kompositionen sind im wesentlichen statische Graphen, welche interaktiv durch

den Benutzer spezifiziert und dynamisch von Vista auf Einhaltung der jeweiligen Kompatibilitätsregeln gegengeprüft werden. Kompositionen werden jedoch in keiner Weise ausgewertet oder interpretiert.

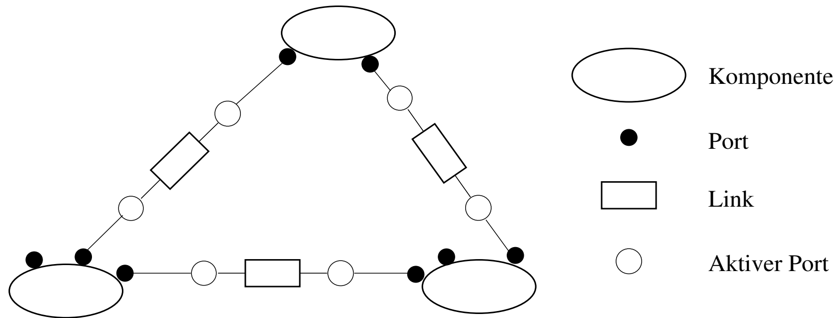


Abbildung 3.7: Komponentennetz eines Vista Frameworks

Vista wurde als Teil einer integrierten Umgebung entwickelt, die ein Software Informationssystem (*SIB*) sowie ein Werkzeug zur Spezifikation und Sammlung von Anforderungen (*RECAST*) enthält. Vista ist hierbei für den Einsatz während der Implementation und Wartung vorgesehen. RECAST, das für frühere Phasen der Softwareentwicklung eingesetzt wird, ist durch Vista implementiert. Anforderungsspezifikationen werden dabei als Komponenten visualisiert. Zugriff und Navigation wird durch SIB realisiert.

Vista ist in dem Sinne ungewöhnlich, daß die meisten sonstigen visuellen Kompositionswerkzeuge domänenabhängig sind, während Vista an beliebige Domänen anpaßbar ist. Mittels Beweis des Konzeptes durch eine prototypische Umsetzung wurden Realisierbarkeit und Nutzen eines generischen visuellen Kompositionswerkzeugs praktisch gezeigt.

3.5.2 Face

FACE [MDE97] ist ein Forschungsansatz zur Erstellung und Nutzung von Frameworks. FACE soll dabei die Nutzung von Frameworks — bzw. die Erstellung von Anwendungen — in ähnlicher Benutzerfreundlichkeit erlauben, wie dies bei heutigen Komponentensystemen wie Visual Basic der Fall ist, soll aber andererseits auch für komplexe Anwendungsdomänen wie die Erstellung von Systemsoftware und -werkzeugen anwendbar sein, die ansonsten nur mit sehr viel schwieriger zu handhabenden objektorientierten Frameworks abgedeckt werden. Die konkrete Anwendungsentwicklung in FACE umfaßt zwei Stufen:

1. Herkömmliche, vom Objektmodell abhängige Komposition verschiedener Objekte, die zumindest anfänglich in der Laufzeitanwendung kooperieren.

2. Erstellung eines abstrakten “*Schemas*” für die Anwendung, welches die Konzepte (abstrahiert durch “*class components*”) und Beziehungen einer spezifischen Anwendung beschreibt.

FACE erlaubt somit neben herkömmlicher Objektkomposition die domänenspezifische Modellierung auf Schemaniveau. Auf diese Weise kann ein Entwickler Verhalten und Kooperation initial instanzierter Objekte auf einer losgelösten, abstrakten Ebene spezifizieren, deren Semantik vorab definiert wurde. Die Möglichkeit dieser Vorab-Definitionen ist die Basis einer entscheidenden Eigenschaft: FACE ist *Domänen-* bzw *Framework anpassbar*. In diesem Sinne findet eine Aufspaltung der Entwicklerrolle in Framework- und Anwendungsentwickler statt. Aufgabe des Framework Entwicklers ist die Bereitstellung domänenspezifischer Modellbestandteile (class-components) , aus denen der Anwendungsentwickler entsprechende Schemata zusammenstellt. Eine weitere benutzerfreundliche Eigenschaft von FACE ist die leichte Integrierbarkeit mit visuellen Kompositionsumgebungen, die jedoch nicht zwingend ist.

3.6 Frameworks

Frameworks sind konkrete Umsetzungen grundlegender, globaler Strukturen von Komponentensystemen. Sie enthalten neben der übergeordneten Struktur meist allgemein anpaßbare Komponenten mit festen Rollen im Gesamtkonzept. Konkrete Frameworks sind überwiegend hochgradig domänenspezifisch und bestehen bei Realisierung durch objektorientierte Konzepte aus abstrakten Klassenhierarchien. Ein Beispiel für moderne Frameworks mit deutlicher Komponentenorientierung ist durch IBMs *San Francisco* gegeben. Klassische Beispiele aus dem objektorientierten Bereich sind *ET++* und *ACE*.

3.6.1 San Francisco

Das *IBM San Francisco Framework* [PVS97] stellt eine objektorientierte, verteilte Architektur zur Entwicklung von Business Anwendungen dar. Um die Erstellung solcher Applikationen speziell bei Umstieg auf das Paradigma verteilter Objekte mit den damit verbundenen Hürden wie Lernaufwand, Qualitätsrisiko und Kosten zu unterstützen, werden grundlegende Java-basierte Geschäftsprozeß-Komponenten mit einem zugehörigen, konsistenten Programmiermodell auf Basis einer verteilten Infrastruktur bereitgestellt.

San Francisco bildet im wesentlichen drei Schichten (Abbildung 3.8). Die unterste *Foundation* genannte, Schicht enthält Infrastruktur und Dienste für

verteilte Multi-Plattform Anwendungen mit geregelter Objektorientierung. Die zweite Schicht der *Common Business Objects* beinhaltet Definitionen allgemeiner Geschäftsobjekte, welche als Basis für anwendungsübergreifende Interoperabilität dienen können. Foundation und Common Business Objects formen zusammen die sog. "*Basis*" (*Base*), welche Anwendungen von den komplexen Aufgaben der plattformübergreifenden Netzwerktechnologien isolieren. Die oberste Schicht der *Core Business Processes* schließlich stellt konkrete Geschäftsobjekte und Standard-Geschäftslogik für vertikale Domänen bereit (z.B. verschiedene Kontenarten und Auftragsmanagement im Absatz- oder Beschaffungsbereich).

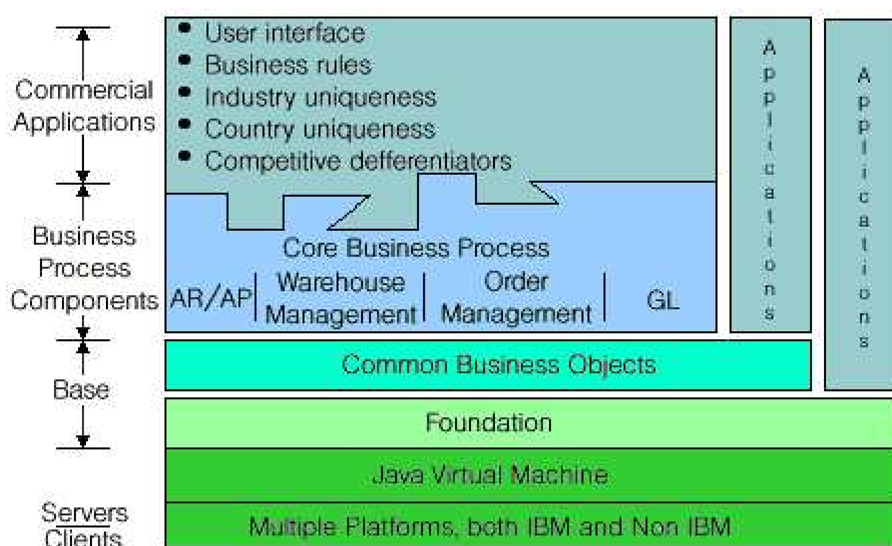


Abbildung 3.8: IBM San Francisco Architektur

Anwendungsentwicklung mit San Francisco kann auf Basis einer beliebigen Schicht erfolgen. Je nach Wunsch kann also entweder auf eine grundlegende verteilte Infrastruktur, zusätzliche Interoperabilitätsmechanismen oder fertige Komponenten mit integrierter Geschäftslogik auf hohem Abstraktionsniveau aufgebaut werden. Die Wahl von Java (siehe 3.2.1) als Implementationssprache impliziert Plattformautonomie und ermöglicht den Einsatz weiterer darauf begründeter Konzepte wie Java- oder EnterpriseBeans.

3.6.2 ET++ und ACE

ET++ *ET++* [WGM89] ist ein objektorientiertes Anwendungsframework, das die Wiederverwendung von Komponenten und Entwurfsmustern in der Domäne GUI-basierter Applikationen unterstützt. ET++ hat dabei die Form einer portablen und homogenen Klassenbibliothek, welche Komponenten grafischer Oberflächen und grundlegende Datenstrukturen enthält

sowie den Austausch von Objekten mit Komponenten des Anwendungsframeworks auf hohem Abstraktionsniveau unterstützt. Weiterhin ist eine Programmierumgebung enthalten. Das Ziel von ET++ ist eine Erleichterung der Entwicklung von stark interaktiven Applikationen mit konsistenten grafischen Benutzerschnittstellen im Sinne der Desktop-Metapher.

ACE Mit dem ***A**daptive **C**ommunication **E**nvironment* (*ACE*) [Syy98] existiert ein weiteres komplexes Rahmenwerk mit dem Schwerpunkt nebenläufiger Kommunikationssoftware. ACE bietet als objektorientiertes Framework eine große Menge wiederverwendbarer C++ Wrapper, Teil-Frameworks und in sich geschlossener Komponenten mit generellen Inhalten allgemeiner Kommunikationssysteme über eine Reihe verschiedener Plattformen. Zu den allgemeinen Inhalten zählen dabei Demultiplexing von Events sowie Event Handler Dispatching, Behandlung von Signalen, Dienst-Initialisierung, Interprozeß-Kommunikation, Verwaltung von geteiltem Speicher (shared memory), Routing von Nachrichten, dynamische (Re)Konfiguration verteilter Dienste, nebenläufige Ausführung und Synchronisation.

3.7 Datenflußsysteme

Datenflußsysteme besitzen meist sehr einfache, effiziente, in jedem Fall aber allgemeingültige datengesteuerte Kopplungsmechanismen. Derartige Systeme stellen einige der bisher erfolgreichsten Beispiele für die Wiederverwendung einfacher, universeller Bausteine dar, wofür der *Unix Pipe* Mechanismus ein gutes Beispiel ist. Daß auch weitaus komplexere Ansätze dieser Art möglich sind, zeigt der *Ptolemy* Ansatz.

3.7.1 Unix Pipes

Der Unix Pipe-Mechanismus stellt eines der erfolgreichsten Beispiele für einfache, effiziente Kompositionsmechanismen dar, wobei sein Erfolg in der Unkompliziertheit besteht, mit der die Wiederverwendung allgemeiner Teilkomponenten ermöglicht wird.

Grundlegende Anwendungen (Filter) mit untypisierten, textuell basierten Datenschnittstellen werden dabei durch einen Mechanismus gekoppelt, der den ausgehenden Datenfluß des ersten Filters in den zweiten einspeist. Auf diese Weise können Ketten aufeinanderfolgender Filter gebildet werden, die eine neue Gesamtfunktionalität ergeben. Zum Beispiel bewirkt folgende Komposition das Sortieren und die Duplikatelemination einer Adressenliste mit anschließender Ausgabe auf dem Bildschirm:

```
cat adressen.txt | sort | uniq
```

Semantische Aspekte der Komposition werden bei Unix Pipes nicht beachtet. So obliegt die Verantwortung einer sinnvollen Komposition alleine dem Benutzer.

3.7.2 Ptolemy

Die Grundlage von *Ptolemy* [BHLM94] ist eine kompakte Software Infrastruktur, auf der beliebige Entwicklungsumgebungen (sog. *Domänen*) aufbauen können. Die Software Infrastruktur, welche als *Ptolemy Kernel* bezeichnet wird, besteht dabei aus einer Familie von C++ Klassendefinitionen. Domänen werden durch Verfeinerung dieser Basisklassen definiert und arbeiten entweder auf Basis von Simulation oder Code Generierung. Im Simulationsmodus erfolgt der Aufruf von Code Segmenten durch einen Scheduler. Bei Code Generierung werden diese Segmente, welche in beliebiger Sprache vorliegen können, zu einer Menge ablauffähiger Programme verschmolzen. Es besteht nun die Möglichkeit, ohne wechselseitiges Wissen eine Interaktion verschiedener Domänen durchzuführen. Es können auf diese Weise Teilsysteme innerhalb der bezüglich Eigenschaften und Semantik optimalen Domäne entwickelt und später ohne zusätzlichen Aufwand zusammengeführt werden. Die Komposition von Subsystemen erfolgt dabei in mehrstufiger, hierarchischer Weise. Innerhalb von Domänen erfolgt die Entwicklung von Subsystemen in beliebiger — eben domänenspezifischer — Weise, vernetzte Komponentenhierarchien werden jedoch in besonderer Weise durch vorgegebene Methoden wie etwa die Möglichkeiten zur Interkonnektion von Komponenten durch Datentransfers (*Geodesic*) über definierte Zugangspunkte *(Port-Holes*) unterstützt. In Bezug auf die Domänen wurden insbesondere solche mit Datenflußsemantik umgesetzt.

3.8 Skriptsprachen

Skriptsprachen dienen der interpretierten Zusammenführung von Teilbausteinen auf Applikationsniveau. Sie stellen dadurch eine weitere Möglichkeit der Integration bzw. Komposition von Subsystemen mit dem Schwerpunkt auf unkomplizierter, schneller Entwicklung kleiner bis mittlerer Anwendungen dar. Das wohl prominenteste Beispiel dieser Familie ist neben *Pearl* die *Tool Connection Language* (*TCL*). In diesem Zusammenhang sind besonders Ansätze, welche die Integration mit Java vorsehen, erwähnenswert.

3.8.1 Java basierte Skriptsprachen

Jacl und TCLBlend *Jacl* ist eine Implementation der Skriptsprache TCL in purem Java, die sich noch in der Entwicklung befindet. Das weniger aufwendigere *TCL Blend* erlaubt die Interaktion des C basierten TCL Interpreters mit der Java virtual Machine. Beiden gemeinsam ist das sog. *Java Package*, das die Kommunikation zwischen TCL und Java ermöglicht.

Durch die so bewirkte Integration ergeben sich zahlreiche Vorteile [Joh98]. Während Java zur systematischen Entwicklung komplexer langlebiger Anwendungen geeignet ist, ergeben sich für die schnelle Erstellung einfacher Programme durch Skriptsprachen bessere Voraussetzungen. TCL ermöglicht es in diesem Sinne, existierende Javakomponenten oder Applikationen schnell und einfach zu integrieren und führt so zu einem neuen Kompositionsmechanismus. Da es sich bei TCL um eine Bibliothek handelt, steht die Skriptfunktionalität auch Java Anwendungen selber offen. Dies ist besonders bei der Komposition von JavaBeans von Vorteil, die deutlich beschleunigt wird. Der systemübergreifend integrierende Charakter von TCL ermöglicht es ferner, C basierte Altlasten in komponentenorientierte Java Systeme einzubeziehen und unterstützt darüber hinaus eine schrittweise Migration entsprechender Subsysteme.

Im Gegenzug zu diesen javaseitigen Vorteilen wird TCL durch eine erweiterte Plattformunabhängigkeit in Bezug auf Erweiterungsmöglichkeit sowie die eigene Lauffähigkeit bereichert und um die zahlreichen Möglichkeiten spezieller Java APIs erweitert.

JPython Mit*Python* existiert eine weitere plattformübergreifende Skriptsprache. *JPython* ist eine Portierung dieser Sprache nach Java. Der Ansatz dabei ist, Phyton Code unmittelbar in Java Bytecode zu kompilieren [Hug97].

Kapitel 4

Teilbereiche von Komponentenarchitekturen

Komponentenorientierte Ansätze existieren, wie der Überblick im vorangegangenen Kapitel verdeutlichte, in vielfältigen, verschiedenartigen Ausprägungen auf gänzlich unterschiedlichen Ebenen. Abstrahiert man von den ansatzabhängigen Details der konkreten Umsetzungen, konzentriert sich also auf die rein komponentenspezifischen Aspekte, kommt man zu einem konzeptionellen Kern, welcher die komponentenorientierte Basissicht darstellt.

Das folgende Kapitel beschäftigt sich mit einer detaillierten Untersuchung der Aspekte und Eigenschaften solcher abstrakten komponentenorientierten Grundlagen und faßt diese zu allgemeinen Konzepten eines abstrakten Architekturbegriffs zusammen.

4.1 Vorüberlegungen

Ein Architekturmodell für komponentenorientierte Systeme benötigt Konzepte, welche verschiedene Sichten aus unterschiedlich abstrakten Blickwinkeln abdecken. Es muß daher aus mehreren Abstraktionsschichten bestehen, welche jeweils unterschiedlich konkrete Aspekte beinhalten. Es hat sich hierfür bei ähnlichen Problemstellungen eine Differenzierung von drei Ebenen bewährt.

Eine erste allgemeine *Modellebene* beschäftigt sich aus übergeordnetem Blickwinkel zunächst mit den eigentlichen Charakterzügen und Wesensmerkmalen von Komponentensystemen, ohne jedoch auf deren Umsetzung einzugehen. In dieser Sicht werden informale Begriffe, Vorstellungen und Konzepte sowie deren Beziehungen betrachtet.

Die zweite Betrachtungsebene enthält Umsetzungen abstrakter Modellbegriffe in konkretere Beschreibungsformen und wird daher als *Beschreibungs-*

ebene bezeichnet. Inhalt dieser Abstraktionsschicht sind Formalismen und Notationen in textueller oder grafischer Form. Solche Mittel erlauben die Spezifikation der Modellbestandteile in einer Weise, die bezüglich höherer Abstraktionsschichten möglichst wenig Informationsverlust mit sich bringen sollte, welcher auf Grund der Konkretisierung unvermeidbar einhergeht. Die Konstrukte dieser Ebene sind immer noch von relativ informaler Natur und erlauben daher auch noch keine direkte, automatische Verarbeitung. Diese ergibt sich erst in der folgenden Sicht.

Die unterste Betrachtungsebene, welche hier zunächst nur am Rande erwähnt wird, enthält eine Umsetzung von Notationen der Beschreibungsebene in geeignete rechnerspezifische Ansätze, d.h. konkret ausführbare Repräsentationen. Man spricht dementsprechend von der *Repräsentationsebene*.

Das Schema der Ebenen deutet schon grob auf Aspektfelder von Komponentenarchitekturen hin, die im folgenden konkretisiert werden. Deren inhaltliche Ausprägungen werden durch die Vorgaben allgemeiner Anforderungen und Wünsche bestimmt, welche auf einem übergeordneten, abstrakten Niveau an Komponentenansätze gestellt werden. Ein entsprechender Anforderungskatalog ist Inhalt der zweiten Untersektion. Konkrete Architekturen werden schließlich in maßgeblicher Weise von der dahinterstehenden Sicht des Komponentenparadigmas bestimmt. Den Abschluß der Vorüberlegungen bildet daher eine Untersuchung einiger methodologischer Aspekte, welche sich auf Architekturinhalte auswirken.

4.1.1 Aspektfelder

Die Aspekte einer Komponentenarchitektur beinhalten zunächst eine Reihe konzeptioneller Modelle. Innerhalb dieser Modelle werden die grundlegenden Begriffe auf informalem Niveau festgelegt, wozu insbesondere der zentrale Komponentenbegriff mit dessen potentiellen Ausprägungen und Bestandteilen wie Schnittstellen, Attributen oder Nachrichten zählen. Ein *Strukturmodell* beschreibt in diesem Zusammenhang die statischen Eigenschaften der wechselseitigen Beziehungen. Ein *Verhaltensmodell* geht auf dynamische Interaktion und deren Semantiken ein. Neben die beiden letzten Modelle, welche zusammen schon eine komplette informale Komponentensicht ergeben, tritt ein *Metamodell*, das den Rahmen für abstraktere übergeordnete Sichten absteckt.

Nachdem eine konzeptionelle Komponentensicht relativ grob und allgemeingültig umrissen ist, muß diese konkretisiert werden. Zu diesem Zweck werden *Spezifikationen* benötigt, welche die praktische Handhabung der Modellabstraktionen erlauben. Die entsprechenden Notationen müssen dabei keine streng formalen Sprachen darstellen und können somit in nahezu beliebiger Form grafische oder textuelle Mittel nutzen.

Auf die konkreteren Notationen der Komponentensicht können formale Modelle angewendet werden. Diese *Formalismen* ermöglichen innerhalb der Architektur exakte Umgangsformen und stellen die Basis für geregelte automatisierbare Verfahren dar. Wesentlich ist hier neben klassisch *statischen* Formalisierungen von Komponenten- und Kompositionsbegriffen die möglichst weitgehende Einbringung *dynamischer Aspekte* von Konglomeraten und der *Semantik* von Bausteinfunktionalität.

Wichtig sowohl bei der Notationsbestimmung als auch — umso mehr — bei der Bestimmung geeigneter Formalismen ist vor allem, daß die konzeptionellen Inhalte der informalen Komponentenmodelle möglichst unverfälscht und vollständig abgebildet werden. Daß dies bei der Formalisierung — zumindest nach heutigem Wissensstand — nicht umfassend möglich ist, versteht sich von selbst. Formalisierung wird daher immer nur Teilbereiche umfassen können und selbst dort bis auf wenige Ausnahmen vereinfachende Annahmen und Idealisierungen beinhalten.

Nachdem die Konzepte einer Bausteinsicht durch verschiedene Teilmodelle geklärt, durch abstrakte Spezifikationen entsprechender Systeme handhabbar gemacht und in praktikablem Rahmen durch formale Objektstrukturen und -semantiken untermauert wurden, kann deren Umsetzung in reale informationstechnische Systeme erfolgen. Auf diesen praktisch orientierten Vorgang wird weiter hinten in Kapitel 6 eingegangen.

4.1.2 Anforderungen

Die inhaltlichen Ausprägungen konkreter Komponentenarchitekturen werden durch die Vorgaben allgemeiner Anforderungen und Wünsche bestimmt, welche auf einem übergeordneten, abstrakten Niveau an Komponentenansätze gestellt werden.

Es stellt sich die Frage, welche Anforderungen in Bezug auf eine abstrakte Komponentenarchitektur allgemein wünschenswert sind. Im folgenden werden potentiell geeignete Eigenschaften diskutiert.

1. **Multidimensionale (De-)Komposition** — *“Komponenten sollen die Zerlegung von Softwaresystemen erlauben und dabei möglichst in multidimensionaler Weise orthogonale Aspekte berücksichtigen.”*

Komponenten sollen zunächst grundsätzlich dem Leitbild einer *Bausteinmetapher* genügen. Wie die Bestandteile eines Baukastensystems — z.B. die berühmten LEGO-Steine — sollen sie konstitutive abgeschlossene Einzelteile darstellen und sich im Rahmen bestimmter wohldefinierter Regeln zu einem Gesamtsystem zusammenfügen lassen.

Die Komposition von Komponenten eines Softwaresystems, welche innerhalb spezifischer Rahmenbedingungen die Aspekte einer Anwendungsdomäne abstrahieren, bzw. die Dekomposition in solche Einheiten bei deren Herleitung, sollte nicht alleine einem “divide and conquer” Prinzip folgend rein funktionalen Charakter besitzen, sondern in mehreren Dimensionen möglich sein, die sich beliebig überlappen können. Wünschenswert ist also eine Zerlegung in orthogonale, unabhängige *Aspekte* [AT98]. So eine Aufspaltung könnte etwa in technische Aspekte wie Nebenläufigkeitskontrolle und Ausnahmebehandlung oder Verhaltensmuster aus der spezifischen Sicht von Einzelkomponenten erfolgen.

2. **Geschlossenheit** — *“Die Komposition von Teilkomponenten soll wieder eine vollwertige Komponente ergeben.”*

Komponentenmodelle sollten bevorzugt mehrstufiger Natur sein. Die Unterscheidung von Komponentenstrukturen und Komponenten ist dann nur eine Frage der Sichtweise, und letztere können somit wiederum als Baustein für erneute Konstruktionen komplexerer Gesamtsysteme dienen. Insbesondere ist dann auch das fertige Softwaresystem eine Komponente. Die Forderung nach *Geschlossenheit* macht das Komponentenmodell beliebig skalierbar, was gerade im Kontext potentieller Verteilung ein wichtiger Aspekt ist. Geschlossene Modelle besitzen zudem den Vorteil, daß sie eine effiziente Basis für formale Methoden darstellen.

3. **Kontextunabhängigkeit** — *“Komponenten sollen unabhängig vom Kontext ihrer Verwendung sein.”*

Es dürfen keine versteckten Abhängigkeiten oder Seiteneffekte bestehen, um eine generelle (Wieder-)verwendbarkeit gewährleisten zu können. Dieser Punkt wird in der nächsten Forderung noch weiter konkretisiert.

4. **Kapselung** — *“Komponenten sollen für sich geschlossene Abstraktionen darstellen und sich durch eine klare Grenze von ihrer Umgebung absetzen.”*

Die essentiellen Eigenschaften sowohl der selbst erbrachten, als auch der vom Kontext vorausgesetzten Funktionalität einer Komponente, sollen von außen über eindeutige, klar definierte Schnittstellen spezifiziert sein. Interne Details sind möglichst weitgehend zu kapseln, wobei sich der Striktheitsgrad an dem Gesamtmodell orientieren muß.

Die sinnvolle Striktheit einer Komponentenkapselung ist eng mit der Aussagekraft der verwendeten Schnittstellenspezifikation verbunden. Während

allgemein schwach gekapselte Whitebox-Ansätze meist grundsätzlich verworfen werden, ist die vollständige Abgrenzung bei Blackbox-Ansätzen umstritten, da im Falle nicht spezifizierbarer interner Kontextabhängigkeiten unkalkulierbare Seiteneffekte eintreten können. Die Thematik wird später noch eingehender behandelt.

5. **Anpaßbarkeit bzw. Adaptability** — *"Komponenten sollen in gewissem Rahmen variabel und anpaßbar sein."*

Die Eigenschaft der Anpassungsfähigkeit fördert die Wiederverwendbarkeit von Komponenten in erheblichem Maße. Gerade weil Komponenten innerhalb eines domänenspezifischen Baukastensystems meist im Vorfeld der eigentlichen Anwendungserstellung entstehen, können nicht alle exakten Umstände des späteren Einsatzes vorausgesehen werden, und selbst die absehbaren Szenerien würden zu einer quantitativ unrealistisch großen Menge individueller Varianten mit lediglich marginalen Unterschieden führen.

6. **Zustand und Identität** — *"Komponenten sollen langlebig sein und sowohl Identität als auch Zustand besitzen."*

Die Forderung nach individueller veränderlicher Beschaffenheit führt dazu, daß Komponenten keinen im mathematischen Sinne funktionalen Charakter besitzen, sondern ihr Verhalten im Laufe der Zeit und abhängig von ihrem Zustand verändern können. Komponenten sind dann keine abstrakten, statischen Schablonen, die beliebig instanziert oder dupliziert werden können und in einheitlicher Weise einen immer gleichen Dienst erbringen. Sie sind vielmehr dynamische, persistente Laufzeiteinheiten, die flexibel anpaßbar sind und mit konkreten, realen Objekten der Anwendungsdomäne korrespondieren.

7. **Substituierbarkeit** — *"Komponenten eines Softwaresystems sollen sich durch kompatible Varianten ersetzen lassen."*

Jede Komponente an jeder Stelle einer Systemstruktur soll sich möglichst sowohl zur Entwicklungs- als auch zur Laufzeit durch eine nach gewissen formalen Regeln "gleichwertige" Komponente austauschen lassen. Die Anpaßbarkeit eines Komponentensystems wird dadurch erheblich erweitert, und es entsteht die Möglichkeit evolutionärer Weiterentwicklung. Wahlfreiheit zwischen verschiedenen ähnlichen Komponenten bildet die Basis für breite Komponentenmärkte mit Wettbewerb und Konkurrenz zwischen alternativen Herstellern.

8. **Offenheit** — *"Komponentenmodelle sollen die Integration heterogener Komponenten unterstützen."*

Die Möglichkeit, heterogene Komponenten in einheitlicher Weise zu verwenden, erlaubt die Auswahl der geeigneten Einheiten aus einem maximalen Pool individueller Varianten. Unrealistische globale Absprachen über notwendige Standards können so auf ein praktikables Maß reduziert werden. Weiterhin bildet die Forderung eine Voraussetzung, um informationstechnische Altlastsysteme innerhalb neuer Technologien integrieren zu können, was Umstellungsprozesse erheblich erleichtert bzw. überhaupt erst möglich macht.

9. **Formalisierung** — *"Komponenten sollen eine formale Spezifikation besitzen."*

Diese Beschreibung stellt zum einen die Basis für Austauschbarkeit von Komponenten dar und ermöglicht zum anderen die Verifizierung der Korrektheit sowie Konsistenz von zusammengesetzten Anwendungs- bzw. Komponentenstrukturen. Ferner ist Typisierung eine Grundlage zur Komponentenauswahl im Rahmen von Entwicklungsprozessen.

10. **Erweiterte Spezifikationsmöglichkeiten** — *"Komponentenmodelle sollen die Spezifikation qualitativer Aspekte und dynamischer Verhaltensmuster sowie der Semantik von Komponenten erlauben."*

Um den hohen Anforderungen gerecht werden zu können, die im allgemeinen an Komponentensysteme gestellt werden, muß der Inhalt von Bausteinspezifikationen über die in bisheriger Programmiermethodik üblichen statischen Strukturinformationen hinausgehen. Es sollten daneben in möglichst weitgehender Weise semantische Verhaltensmuster eingebracht werden. Ohne diese Informationen ist eine sichere automatisierte Verwendung von gekapselten Code-Bausteinen nicht möglich, da deren Anwendungsweise nicht aus den strukturellen Eigenschaften hervorgeht. Ferner sollte es auch möglich sein, qualitative Eigenschaften von Komponenten wie Zuverlässigkeit/Robustheit oder Performanz/Ressourcenbedarf und Vielseitigkeit/Wiederverwendungswahrscheinlichkeit einzubringen, um deren Eignung besser — und automatisierbar — abschätzen zu können.

11. **Nebenläufigkeit** — *"Komponenten sollen nebenläufig und potentiell verteilt agieren."*

Da Komponentensysteme naturgemäß Konglomerate autonomer Subsysteme darstellen, liegt es nahe, deren unabhängige Aktivität in Zeit und Raum als Grundlage zu fordern. Dies wird durch die enge Verzahnung von konkreten Ausprägungen innerhalb der Bereiche Komponenten- und Parallelarchitekturen sowie Verteilung, Vernetzung, Internet untermauert.

12. **Metaebene** — *"Komponentenmodelle sollen über explizite Metaebenen verfügen."*

Die Komplexität von Komponentenarchitekturen läßt die Verwendung einheitlicher Mechanismen für alle Aspektbereiche unrealistisch erscheinen. Speziell im Bereich der Spezifikation steht den zur automatisierten Komposition notwendigen komplexen Beschreibungen der Wunsch nach einer abstrahierenden, übergeordneten Typisierung entgegen. So eine allgemeine Klassifikation erlaubt den praktischeren Umgang mit Komponenten beispielsweise bei deren Suche zur Entwicklungszeit und behindert nicht durch Überspezifikation. Als Lösung dieser Problematik bietet sich die Verwendung einer oder mehrerer expliziter Metaebenen mit jeweils für verschiedene Zwecke optimierten Konzepten an.

4.1.3 Methodologie

Die Methodologie komponentenorientierter Softwareentwicklung beinhaltet verschiedene Aspekte des Verständnisses einer allgemeinen Komponentenmetapher. Dazu gehört die abstrakte Vorstellung eines Komponentenbegriffes und dessen prinzipielle Verwendung bzw. konkrete Anwendung in Entwicklungsprozessen. Konkrete Ausprägungen solcher Ideologien wirken sich in direkter Weise auf die Konzeption von Komponentenarchitekturen aus. Zu bedenken ist dabei, daß nicht alle Aspekte einer Komponentenideologie durch entsprechende Architekturen zu fassen sind. Vielmehr ist ein dementsprechender Versuch oft auch gar nicht sinnvoll, und Architekturen sowie deren zugrundeliegenden Modelle konzentrieren sich deshalb oftmals auf Teilaspekte, um zumindest diese in befriedigender Weise handhaben zu können.

Ein Beispiel dafür sind etwa Architekturen, die sich auf Komponentenkopplung — und somit hauptsächlich auf stark technisch orientierte Kommunikations- und Bindungsaspekte — konzentrieren. Globale Aspekte umfangreicher Komponentenstrukturen, wie etwa eine übergreifende Sicht als Framework, liegen dann meist nicht im Horizont der Architektur.

4.1.3.1 Verständnis der Komponentenmetapher

Die zentrale Frage komponentenorientierter Methodologien lautet: *"Auf welche Weise sollen Komponenten zur Konstruktion von Softwaresystemen bzw. zur Dekomposition von Problembereichen eingesetzt bzw. angewandt werden?"*

Diesbezüglich müssen zumindest zwei Varianten unterschieden werden. Zunächst könnte eine an den klassischen Methoden der Objektorientierung

angelehnte Sicht eingenommen werden, bei der Komponenten — wie die Klassen von Frameworks — als Ausgangspunkt für schrittweise Spezialisierungen mit "Vererbung" ihrer Funktionalität dienen. Diese schrittweise Spezialisierung wird auch als Verfeinerung (*engl. refinement*) bezeichnet. Die folgende Kette gibt hierzu ein Beispiel: `Fahrzeug` → `KFZ` → `PKW` → `Stufenhecklimousine`.

Eine zweite, dinglichere Sicht könnte hingegen stärker die Identität und "Einzigartigkeit" individueller Bausteine betonen, wobei Komponenten dann in konzeptionell unveränderter Form aggregiert werden und konstituierende Teile eines aus dem Konglomerat hervorgehenden Ganzen darstellen (z.B. `KFZ = ( Karosserie + Motor + Rad1 ...)`.

Nach heutigem allgemeinem Verständnis ist die letzte Sichtweise weit verbreitet. Der Verfeinerungsgedanke tritt im Zusammenhang mit Komponenten eher selten auf. Ein Grund dafür mag sein, daß der damit eng einhergehende Vererbungsbegriff negative Assoziationen mit den Schwachstellen objektorientierter Methoden weckt, welche doch — so die verbreitete Hoffnung — durch das Komponentenparadigma gerade umgangen werden sollen. Ob dies begründete Argumente sind — sprich: ob Refinement untrennbar mit Whitebox-Reuse verbunden ist — muß eingehender untersucht werden. Sollten verträgliche Methoden der Verfeinerung existieren, würde deren Integration in Komponentenmethodologien, wie sich zeigen wird, eine sinnvolle Ergänzung darstellen.

Es soll nun in einer praktischer orientierten Sichtweise der konkrete Entwicklungsprozeß von Software unter Einfluß des Komponentenparadigmas betrachtet werden. Es sind dabei zwei konträre Vorgehensweisen zu unterscheiden. Zunächst wäre es denkbar, den Komponentengedanken als rein operationales Prinzip zu verstehen und in diesem Sinne bei jedem neuen Softwareprojekt eine allgemeine Top-Down Dekomposition mit individueller Ableitung spezifischer Komponenten durchzuführen. Eine strategische Sicht würde demgegenüber komponentenorientierte Softwareentwicklung als langfristigen, übergreifenden, lediglich im Hinblick auf Domänen spezifischen Prozeß betrachten. Dieser startet dann mit einem initialen Domain-Engineering Prozeß, welcher die Komponenten bestimmt und spätere Wiederverwendung dieser Komponenten in immer neuen Projekten gemäß eines initial entwickelten Frameworks erlaubt. Der wesentliche Unterschied dieser beiden Sichten, welcher sich entscheidend auf die Konzeption einer geeigneten Komponentenarchitektur auswirkt, liegt im kausalen Zusammenhang von Komponentenerstellung und -verwendung. Die Vorstellung eines vorhandenen "Komponentenbaukastens", aus dem vorgefertigte Komponenten zur Konstruktion eines neuen Systems entnommen werden, stellt gänzlich andere Voraussetzungen — z.B. an die Komponentenspezifikation — als ein Ansatz, bei dem System- und Komponentenentwurf simultan einhergehen.

Da der heutige Componentware-Begriff von vorgefertigten Bausteinen aus-

geht, soll dies die im weiteren Verlauf bevorzugt eingenommene Sicht sein. Die bestehenden, komponentenartigen Objekte solcher Ansätze müssen hochgradig anpaßbar und variabel sein oder noch gar nicht als konkrete Komponenten im eigentlichen Sinne auftreten — etwa nur als unscharfe Beschreibung oder abstrakter Typ. Hier wäre nun die Möglichkeit des oben beschriebenen Refinements sinnvoll, welches dazu beitragen könnte, die Grenzen allgemeiner "Adaptability" auszuweiten.

4.1.3.2 Integration einer Refinement Sicht

Wenn man die methodologische Integration von Bausteinverfeinerung akzeptiert, könnte man Refinement durch Umgehung der damit im klassischen Sinne immer einhergehenden Beziehung zu Vererbungskonzepten potentiell in eine mit dem Komponentengedanken harmonierende Form bringen.

Refinement in diesem Sinne könnte dadurch erreicht werden, daß nicht eine Komponente komplett übernommen und anschließend verändert wird — wie dies im Sinne klassischer Vererbung der Fall wäre —, sondern man im Gegensatz dazu diese Komponente unter Beibehaltung ihrer Importbeziehungen komplett austauscht. Ein erhebliches Maß an Wiederverwendung ist dann durch die Subkomponenten gewährleistet, und ein Refinement Prozeß findet unmittelbar am Punkt der Implementierung statt, welche die zu verfeinernde Funktionalität (die exportierte Funktionalität der Toplevelkomponente) erbringt. Diese Vorgehensweise verletzt die Komponentensicht im Gegensatz zu einer Vererbung mit Overwriting nicht, da sie einem Reassembling der Einzelkomponenten entspricht.

Bei der Entscheidung, Refinement in ein Komponentenmodell zu integrieren, sind neben sauberer methodologischer Einbettung noch die resultierenden Konsequenzen für Komponentenschnittstellen und -spezifikationen im jeweiligen Modell zu beachten — dazu folgt später mehr.

4.2 Modellebene

Die Modellebene beherbergt den konzeptionellen Kern von Komponentenarchitekturen. Man unterscheidet auf diesem Abstraktionsniveau zunächst Modelle für Struktur- und Verhaltensaspekte. Ein optionales, ergänzendes Modell, welches jedoch nicht gezwungenermaßen in jeder Architektur enthalten sein muß, beschreibt die Konzepte einer übergeordneten Metaebene.

4.2.1 Strukturmodelle

Die folgende Sektion beschäftigt sich mit den Aspekten komponentenspezifischer Strukturmodelle. Hierzu zählen vor allem die statischen Komponenten-

bestandteile, deren Konglomerat zu einem allgemeinen Komponentenbegriff führt, sowie die wechselseitigen Beziehungen dieser Einheiten zueinander.

4.2.1.1 Eine Basisrelation für Strukturmodelle

Die Struktur des gewählten Komponentenmodells spiegelt die Beziehungen der involvierten Komponenten wieder. Diese Beziehungen der Bausteine untereinander können informell als "verwendet", "setzt sich zusammen aus" oder besser "ist komponiert aus" beschrieben werden. Mathematisch stellen solche Beziehungen Relationen dar. Bei der Abbildung struktureller Komponentenbeziehungen auf Relationen ergeben sich für letztere einige Bedingungen, welche zur adäquaten Modellierung notwendig erscheinen.

Charakterisierung 3 (Basisstrukturrelation) *Eine grundlegende, mathematische Relation für Komponentenstrukturen sollte folgende Eigenschaften besitzen:*

1. *Irreflexivität*
2. *Antisymmetrie*
3. *Transitivität*

Zunächst sollte *Irreflexivität* gefordert werden, da Komponenten sich nicht selber rekursiv nutzen können. Diese Eigenschaft ist in ihrem Charakter als eigenständige, individuelle Einheiten begründet. Die gegenseitige Nutzung gleicher Komponenten wird durch mehrere simultane Einheiten des selben Komponententyps möglich, was jedoch bei Fokussierung auf strukturelle Aspekte transparent bleibt.

Als nächste Eigenschaft ist die *Antisymmetrie* der Relation zu nennen. In Übertragung steht dahinter die Aussage, daß zwei Komponenten sich nicht wechselseitig auseinander zusammensetzen können. Dies gilt allerdings nicht für die informale Sicht der gegenseitigen Nutzung. In letzterem Sinne bedeutet es nur, daß solche Nutzbeziehungen stets von einer Einheit kontrolliert und koordiniert werden, was sich jedoch im Sinne geregelter Kooperationsverflechtungen äußerst positiv auswirkt. Die Thematik wird später noch eingehender untersucht.

Die dritte Eigenschaft der *Transitivität* bedeutet, daß eine Komponente a, welche eine weitere Komponente b als direkte Teilkomponente enthält, indirekt auch eine Komponente c als Bestandteil beinhaltet, aus welcher b komponiert ist. Die Transitivitätseigenschaft entfällt bei einer Modellsicht, welche jeweils nur Teilsichten auf einzelne Komponenten und deren direkt konstituierenden Untereinheiten erlaubt, um Abstraktionsbarrieren zwischen den Ebenen zu betonen.

Da die Basisrelation antisymmetrisch ist, sind die Graphen der entsprechenden Klasse gerichtet. Betrachtet man sich diese Klasse der *Komponentenstrukturgraphen*, so ist für den Begriff einer *Applikation* noch folgende Einschränkung zu machen: In einer Applikation bzw. Komponentenstruktur gibt es nur eine Wurzelkomponente. D.h. eine Komponente, die nicht konstituierender Teil eines anderen Bausteins ist, begründet eine Applikation (Abb.4.1.a). Die Begriffe *"Wurzelkomponente"*, *"Toplevelkomponente"* und *"Applikation"* bzw.
"Anwendung" werden im weiteren Verlauf als diesbezügliche Äquivalente benutzt.

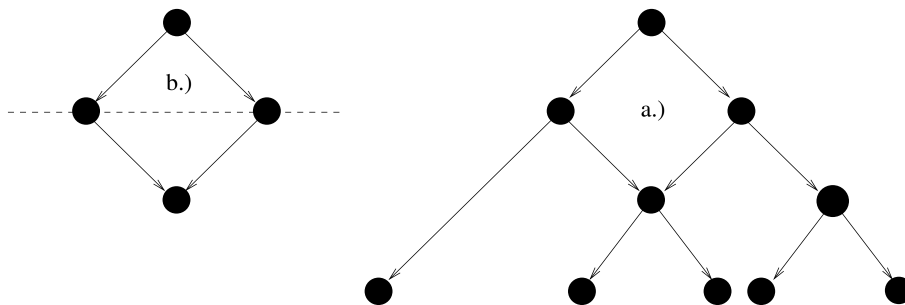


Abbildung 4.1: Grundlegende Komponentenstrukturen

4.2.1.2 Strukturelle Beschränkungen

Die bisher beschriebenen Strukturmerkmale ergeben einen sehr breiten Rahmen möglicher konkreter Modelle. Es stellt sich die Frage, ob in Bezug auf komponentengerechte Modellstrukturen möglicherweise Einschränkungen notwendig sind, welche explizit gefordert werden müssen. Insbesondere scheint die offensichtliche Problematik bezüglich mehrfacher Verwendung derselben bzw. gleichen Komponenten einer Regelung zu bedürfen. Daneben stellen sich Zyklen als kritisch heraus.

Mehrfachverwendung von Komponenten Eine wichtige Fragestellung bei der Untersuchung von Komponentenstrukturen betrifft die Handhabung von mehrfacher Verwendung einzelner Komponenten in entsprechenden Modellen.

Von der ideellen Vorstellung her legt die Benutzung individueller Komponenten als Teile von Anwendungen deren *Verbrauch* nahe. Eine verwendete und damit verbrauchte Komponente könnte bei dieser Vorstellung nicht erneut zur Komposition eingesetzt werden. Tatsächlich ist die mehrfache Verwendung aber notwendig, denn eine als Baustein realisierte, exklusive Ressource

— etwa eine Datenbasis — darf nicht durch ihre Nutzung innerhalb einer Anwendung für andere Zwecke unzugänglich werden.

So eine Mehrfachverwendung könnte dabei entweder innerhalb einer (Abb.4.1.b) oder zwischen verschiedenen (Abb.4.2) Komponentenstrukturen bzw. Applikationen stattfinden.

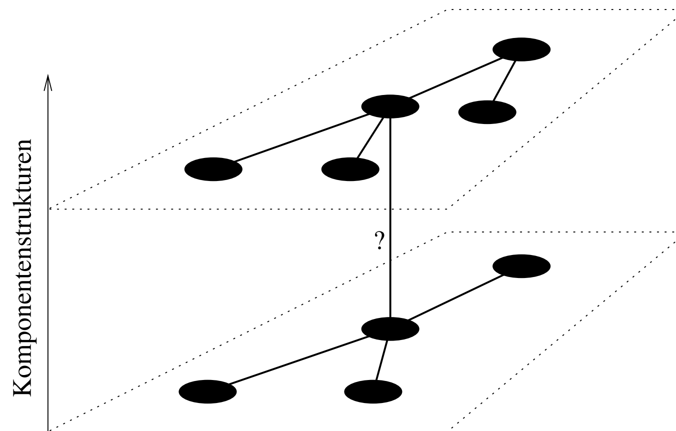


Abbildung 4.2: Applikationsübergreifende Komponentennutzung

Die Frage der Mehrfachverwendung untergliedert sich somit in einen Bereich verschiedener Probleme, deren Teilaspekte im folgenden beschrieben werden. Zunächst ist dabei zu klären, ob zum einen individuelle Komponenten und zum anderen verschiedene Instanzen eines gemeinsamen Komponententyps in sinnvoller Weise mehrfach zum Einsatz kommen können bzw. sollen. Diese letzte Frage ist jeweils im Bezug auf genau eine oder mehrere separate Komponentenstrukturen bzw. Applikationen zu beantworten.

Multiple Verwendung einer einzelnen individuellen Komponente setzt deren Fähigkeit zur Handhabung nebenläufiger Dienstzugriffe voraus. Individuelle Anpassungsmöglichkeiten bedingen in diesem Fall besondere Konzepte, ansonsten erfolgt sie pauschal.

Mehrfache Initialisierung eines Komponententyps kann im Falle einer dahinterstehenden einzelnen Ressource die Synchronisation der Instanzen notwendig machen. Individuelle Anpassung ist bei dieser Variante naturgemäß enthalten.

Bei anwendungsinternen Mehrfachverwendungen scheint eine entsprechende mehrfache Initialisierung die konzeptionell sauberere Lösung zu sein, da die Instanzen dann jeweils genau einmal einen Verbund eingehen. Daneben bestehen aber auch weitere Alternativen. So kann versucht werden, die Anwendungsfunktionalität, welche eine bestimmte Komponente nutzt, möglichst innerhalb einer einzelnen Verbundkomponente zu integrieren, statt über

mehrere zu verteilen. Die Notwendigkeit der Mehrfachverwendung wäre dann nicht mehr gegeben. Eine weitere Alternative besteht darin, die Funktionalität einer verwendeten Komponente a durch eine verwendende Komponente b “durchzuschleifen”. Eine dritte Komponente c müßte dann nicht ebenfalls a verwenden, sondern könnte auf b zugreifen.

Bei der (Ver-)Teilung einer Komponente über mehrere Anwendungen ist die verwendete Methode aus methodologischer Sicht gleichgültig, da im Falle einer Instanz von den anwendungsexternen Beziehungen leichter abstrahiert werden kann. In diesem Fall kann je nach benötigten Synchronisations- und Anpassungsmechanismen die vorteilhaftere Variante gewählt werden. Tabelle 4.1 faßt die verschiedenen Varianten abschließend noch einmal zusammen.

<table>
<tr><th></th><th>Verwendung derselben Komponente</th><th>Verwendung der gleichen Komponenten eines Typs</th></tr>
<tr><td rowspan="2">Eine Komponentenstruktur</td><td>Problematisch</td><td>Praktikabel</td></tr>
<tr><td colspan="2">Möglichst Alternativen suchen</td></tr>
<tr><td>Mehrere Komponentenstrukturen</td><td colspan="2">Wahlfrei, je nach Rahmenbedingungen</td></tr>
</table>

Tabelle 4.1: Mehrfachverwendung von Komponenten

Zyklische Abhängigkeiten Eine weitere strukturelle Beschränkung betrifft zyklische Kompositionen. Es muß verhindert werden, daß eine Komponente in einen Verbund eingeht, die selbst aus einer Komposition von Komponenten hervorgegangen ist, welche wiederum den neuen Verbund beinhalten (siehe Abb.4.3). Zyklische Abhängigkeiten dieser Art verstoßen gegen das Komponentenparadigma, da eine Komponente ein abgeschlossenes dingliches Gebilde für sich sein soll. Die rekursive Struktur führt zu unspezifizierten externen Abhängigkeiten und macht den betroffenen Baustein umgebungsabhängig. Der Strukturgraph eines komponentenorientierten Systems darf dementsprechend keine Zyklen enthalten.

4.2.1.3 Schnittstellen

Der *Schnittstellenbegriff* ist im Zusammenhang mit komponentenorientierten Strukturmodellen breiter und abstrakter zu sehen als die in herkömm-

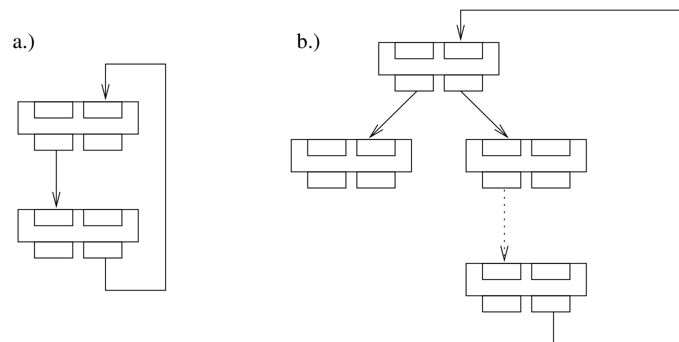


Abbildung 4.3: Zyklen

lichen Systemen üblichen operationalen Ausprägungen. Schnittstellen werden hier in diesem Sinne ganz allgemein als die äußeren Zugangspunkte von Komponenten aufgefaßt. Bei einer detaillierteren, weniger abstrakten Betrachtung bestehen die strukturellen Beziehungen im Modell dann nicht zwischen den Komponenten als Ganzes, sondern zwischen ihren Komponentenschnittstellen. Strukturelle Aspekte, die im Zusammenhang mit Schnittstellen zu bedenken sind, umfassen nun die folgende Bereiche:

- Anzahl der Schnittstellen pro Komponente
- Unterscheidung von Import- und Exportschnittstellen
- Wechselseitige Schnittstellenbeziehungen und komponentenspezifische Zuordnung
- Schnittstelleninhalte
  - Operationelle Schnittstellen
  - Streamschnittstellen
  - Signalschnittstellen

Bezüglich der *Schnittstellenanzahl* pro Komponente ist vor allem die Unterscheidung zwischen $1 : 1$ und $n : 1$ Beziehungen ein wesentlicher Faktor. Mehrere Schnittstellen pro Komponente sind dabei als pragmatischerer Ansatz zu betrachten. Es wird dadurch eine bessere Gliederung von Komponenten möglich. Multiple Sichten oder Rollen lassen sich implementieren. Durch das Hinzufügen von neuen, verbesserten Schnittstellenversionen zu den bereits bestehenden — und verwendeten — kann eine abwärtskompatible Evolution erreicht werden. Die Beschränkung auf genau eine Schnittstelle pro Komponente führt hingegen zu einem potentiell klareren, einfacheren Modell. Probleme der Spezifikation bei multiplen Schnittstellen werden umgangen, da keine internen Abhängigkeiten bestehen.

Um das Erzeuger/Konsumenten-Verhältnis von Komponenten bei wechselseitigen Beziehungen auch in Bezug auf deren Schnittstellen explizit zu machen, besteht die Möglichkeit, diese in *Import- und Exportschnittstellen zu differenzieren.* Exportschnittstellen enthalten dann die bereitgestellte Funktionalität einer Komponente, während Importschnittstellen die von ihr vorausgesetzten Leistungen untergeordneter Einheiten ausdrücken. Es stellt sich hierbei die Frage, ob Komponenten solche Importschnittstellen als integralen Bestandteil enthalten müssen. In einem Komponentenverbund — der durch die Existenz von vorausgesetzten Leistungen in jedem Fall impliziert wird — wäre nämlich als Alternative eine Identifizierung der Importschnittstelle mit den entsprechenden Exportschnittstellen ohnehin eingebundener Teilkomponenten denkbar. Diese müßten dann nur — z.B. durch eine geeignete Typisierung — referenziert werden. Eine weitere Lösung könnte schließlich in konsequenter Weise die komplette Auslagerung der Schnittstelle als eigenständige Einheit vorsehen, welche dann das absolute Bindeglied zwischen Komponenten darstellen würde.

Es bestehen somit verschiedene gangbare Optionen von *Beziehungen* zwischen Import- und Exportschnittstellen und deren *Zuordnung* zu Komponenten verschiedener Strukturebenen, wobei die Wahl von der sauberen Integration in das Gesamtmodell abhängig ist. Im weiteren Verlauf soll zunächst von multiplen, explizit zu jeder Komponente zugehörigen Im- und Exportschnittstellen ausgegangen werden.

4.2.1.4 Verfeinerte Struktur

Nachdem statische Beziehungen auf der Granularitätsebene ganzer Komponenten betrachtet wurden, wird im folgenden auf eine verfeinerte Sicht geschwenkt. Neben der eingehenderen Betrachtung multipler Schnittstellen widmet sich der Abschnitt den Schnittstelleninhalten.

Multiple Schnittstellen Die Berücksichtigung multipler Export- bzw. Importschnittstellen bietet eine Reihe methodologischer Vorteile. Aus struktureller Sicht ergeben sich dadurch zusätzliche Beziehungsformen, deren semantische Bedeutung für Komponentenmodelle zu prüfen ist. Allgemein können die in Abb.4.4 skizzierten Fälle unterschieden werden.

In der Abbildung, bei welcher die Fälle durch Buchstaben gekennzeichnet werden, sind Komponenten als flache Rechtecke mit Schnittstellen aus je nach Export- oder Importcharakter äußeren bzw. inneren Kästchen dargestellt. Beziehungen zwischen kompatiblen Interfaces werden durch Pfeile angedeutet.

Zu der schon weiter oben beschriebenen Problematik multipler Komponentenverwendung tritt offensichtlich ein simultaner Effekt auf verfeinertem

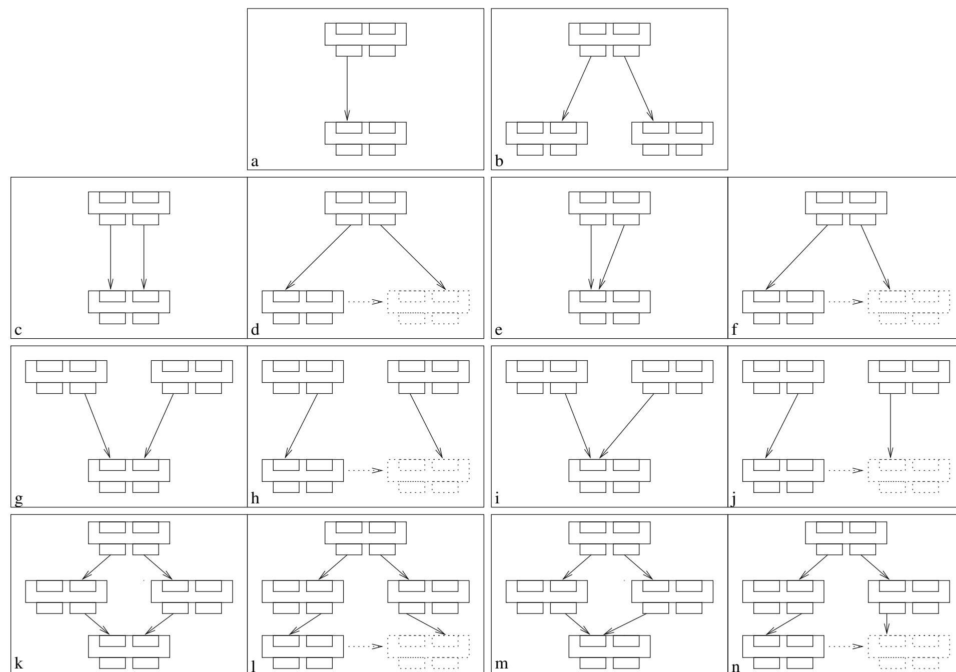


Abbildung 4.4: Mögliche Strukturen bei multiplen Interfaces

Schnittstellenniveau. Dieser kann nun schon bei einfachen Komponentenbeziehungen auftreten, nämlich immer dann, wenn es zu einer mehrfachen Bindung verschiedener Schnittstellen zweier Bausteine kommt (in den Fällen c und e). Derartige Situationen können durch mehrfache Instanzen des betroffenen Komponententyps aufgelöst werden, was in der Abbildung jeweils durch alternative Fallbehandlungen mit gestrichelten Komponenten — welche für die geklonten Einheiten stehen — angedeutet wird. Im Falle einer Simultanbeziehung mit mehr als zwei Bausteinen treten die bekannten Probleme erweitert durch zusätzliche Konfliktfälle auf.

Atomare Funktionseinheiten Verbundkomponenten setzen sich jeweils aus einer Menge von Subkomponenten zusammen. Genauer betrachtet nutzt die Toplevelkomponente des Verbundes die Funktionalität bzw. die angebotenen Dienste der involvierten Teilkomponenten. Um solche Dienste allgemein beschreiben zu können, sollten möglichst abstrakte funktionale Einheiten verwendet werden, welche ein Modell nicht in seiner Vielfalt und Aussagekraft beschränken. Auf derartige Basiseinheiten sollte sich ein Maximum der praxisüblichen Konstrukte wie Funktionen, Prozeduren oder Methoden abbilden lassen. Sehr verbreitet ist z.B. eine Kombination aus Methoden, Attributen und Ereignissen.

Als potentiell zur Modellierung all dieser Einheiten geeignete Einheiten

scheinen die sehr allgemeinen Vorstellungen der *"Nachricht"* und des *"Kanals"* vielversprechende Kandidaten zu sein.

1. **Nachricht** Der Begriff der *Nachricht* ist eine weit verbreitete Abstraktion und stellt die Grundlage klassischer Objektmodelle dar. Zwischen den einzelnen Schnittstellen kommunizieren die Komponenten in diesem Fall über ein- und ausgehende atomare Einheiten mit festgelegter Bedeutung. So eine Kommunikation findet dabei immer zwischen genau zwei kompatiblen Schnittstellen statt, die ein Import/Export Paar bilden (Abb.4.5).

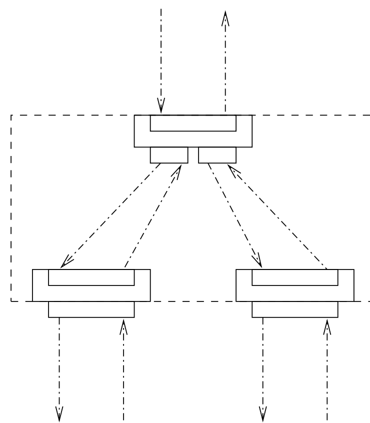
   

   Abbildung 4.5: Bidirektionaler Nachrichtenaustausch

   Bei der Abbildung von parametrisierten Konstrukten auf Nachrichten sind verschiedene Ansätze denkbar. Zunächst kann eine komplexere Sicht des Nachrichtenbegriffs diese als selber mit Parametern versehene Einheiten verstehen. Alternativ können aber auch Parameter ihrerseits wieder als Nachrichten modelliert werden. Da parametrisierte Konstrukte dann potentiell (und bei solchen mit mathematisch funktionalem Charakter — also mit Rückgabewert — in jedem Falle) aus mehreren Nachrichten bestehen, muß als weitere Erschwernis deren Reihenfolge gesichert werden. Das strukturelle Grundmodell ist dabei durch ein entsprechendes Verhaltensmodell zu ergänzen.

2. **Kanal** Eine weitere mögliche Abstraktion ist die des *Kanals.* Zwischen zwei Schnittstellen fließen dabei Daten — eventuell wiederum Nachrichten — durch eine Menge gerichteter Kanäle. Über Art, Richtung, Reihenfolge und Lebensdauer von Kanälen lassen sich alle oben angedeuteten funktionalen Konstrukte modellieren. (Abb.4.6). Als konkretes Beispiel für die Verwendung von Kanälen in einem Komponentenmodell sei auf Broy [Bro98, Bro96, BDH+98] verwiesen.

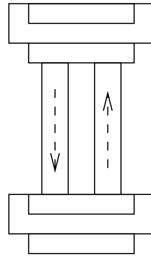


Abbildung 4.6: Kanalabstraktion

4.2.2 Verhaltensmodelle

Die Funktionalität einer Komponente wird durch atomare Einheiten — seien dies nun Methoden, Nachrichten oder Kanäle — innerhalb ihrer Schnittstellen abstrahiert. Solche Dienste, welche von den Komponenten des Systems angeboten oder vorausgesetzt werden, stellen den eigentlichen Nutzen spezifischer Bausteine dar und können beliebige Inhalte annehmen.

Gegenstand eines allgemeinen Verhaltensmodells für Komponentenarchitekturen betreffen nur den abstrakten Dienstbegriff, nicht aber deren spezielle Inhalte. Bezüglich abstrakter Dienste können nun Verhaltensaspekte auf zwei Ebenen unterschieden werden. Zum einen sind das Eigenschaften, welche Dienstnutzungen zwischen Komponenten betreffen und zum anderen solche zwischen verschiedenen Diensten einer einzelnen Komponente.

4.2.2.1 Externes Verhalten

Das externe Verhalten beinhaltet allgemeine Eigenschaften der Komponenteninteraktion. In diesem Rahmen muß festgelegt werden, nach welchen grundsätzlichen Regeln Vorgänge zwischen Komponenten ablaufen sollen.

Man kann zunächst argumentieren, daß Interaktionen zwischen Komponenten stets von oben nach unten eingeleitet werden. Eine übergeordnete Komponente beginnt in diesem Sinne das Zusammenspiel, indem sie den Dienst einer ihrer konstituierenden Teilkomponenten in Anspruch nimmt. Eine Teilkomponente wird andersherum nie von sich aus Dienstnutzungen übergeordneter Teile initiieren.

Weiterhin kann eine übergeordnete Komponente nur Dienste unmittelbarer Teilkomponenten in Anspruch nehmen, da sie nur mit deren Schnittstellen in Beziehung steht. Funktionalität tiefergelegener Schichten kann somit nur indirekt genutzt werden.

Da zwei beliebige Komponenten einer Anwendung immer in eindeutiger gerichteter Beziehung zueinander stehen und eine Dienstnutzung nur in direk-

ter Weise erfolgen kann, sind Komponenteninteraktionen immer klar durch eine ihrer beteiligten Instanzen geregelt.

Der letzte Punkt hat zur Folge, daß insbesondere eine wechselseitige Interaktion zweier Komponenten derselben Ebene immer über einen Mediator — die Toplevelkomponente — erfolgt. Diese Tatsache hat eine Reihe positiver Auswirkungen. Nach Pazzi [Paz97] liegt ein besonderer Vorteil darin, daß die Eigenschaften der Interaktion durch die Mediatorkomponente explizit gemacht und somit faßbar werden, anstatt implizit in den beteiligten Bausteinen zu verdeckten Abhängigkeiten zu führen.

4.2.2.2 Internes Verhalten

Einer der wesentlichsten Punkte bei der Modellierung von Komponentensystemen betrifft die Semantik der Teilbausteine. Diesem Aspekt wird in herkömmlichen Ansätzen bisher nur sehr wenig Beachtung geschenkt. Derartige Systeme beschreiben in der Regel nur statische Aspekte ihrer Bestandteile, wie etwa die Signaturen von Operationen innerhalb von Schnittstellen. Die Semantik dieser Operationen wird dann durch Typisierung eingebracht, die aber entweder explizit und informal — durch einfache Zuweisung bzw. Benennung — oder aber implizit und wenig aussagekräftig — durch strukturbasierte Inferenzmechanismen — ist. Durch dieses Vorgehen lassen sich weder inhaltliche Bedeutung noch sinnvolle Anwendung der funktionalen Einheiten allgemeingültig ableiten.

Der letztere Teilaspekt von Komponentensemantik — die regelgerechte Anwendung — manifestiert sich konkreter in den dynamischen Beziehungen verschiedener atomarer Dienste des Bausteins. Zu diesen Beziehungen gehören vor allem die kausalen Abhängigkeiten bzw. Ausführungsreihenfolgen der verschiedenen Funktionseinheiten. Ein Komponentenmodell sollte diese Reihenfolgebeziehungen möglichst beinhalten.

Obwohl die Berücksichtigung geregelter Anwendung ein wichtiger Aspekt ist, läßt sich Komponentensemantik nur bis zu einem gewissen Grad sinnvoll in ein entsprechendes Verhaltensmodell integrieren. Die andere wichtige Teilfrage nach Bedeutung und Wesen atomarer Diensteinheiten kann nur durch eine weitere Verfeinerung und Aufgliederung der jeweils untersten Ebene angegangen werden. An einer unteren Grenze der Modellierung müssen Dienste dementsprechend durch explizite Einteilung bzw. Zuordnung unterschieden werden. Wo genau diese untere Grenze gezogen wird, ist von den näheren Umständen des Modells abhängig. Das Extrem ist dabei die Implementierung selber. Diese beschreibt die Funktionalität in absoluter, jedoch für eine Komponentensicht ungeeignete — weil überspezifizierter — Weise.

Man kann somit festhalten, daß die Frage der Semantik in Komponentenmodellen nach heutigem Verständnis nicht allumfassend zu lösen ist. Gleichwohl

sind die Möglichkeiten in diesem Bereich aber noch nicht ausgeschöpft, und Komponentenmodelle sollten entsprechende Konzepte in verstärktem Maße integrieren.

4.2.3 Metamodelle

Komponentenmodelle müssen vielerlei Ansprüchen genügen. Zum einen sollen sie ausdrucksstark und detailliert sein, um die Vorgaben und Anforderungen auf technischem Niveau umsetzen zu können, zum anderen ist Übersichtlichkeit und die Möglichkeit allgemeiner, grober Klassifizierung zur praktischen Handhabbarkeit zu bewahren. Der gegensätzliche Standpunkt dieser Sichten läßt es sinnvoll erscheinen, die Abtrennung einer abstrakteren Metaebene in Erwägung zu ziehen.

Ansätze für entsprechende Metamodelle sollten auf einer ganzheitlichen Komponentensicht von außen beruhen. Das Hauptanliegen ist dabei die pragmatische Klassifizierung von Komponenten in grobe Themenbereiche. Ein entsprechendes Typsystem wäre gegenüber der detailierten, im technischen Sinne kompositionsorientierten Modellsicht wesentlich in seiner Komplexität reduziert. Auf Grund der informalen Natur ist diese Sicht demgegenüber aber nur durch eine explizite Form der Zuweisung von Bausteinen zu Klassen und Benennung derer wechselseitigen Beziehungen denkbar.

4.2.3.1 Anwendungsfelder

Die Anwendung von Metamodellen ist vor allem in zwei Schwerpunkten von Nutzen. Zunächst würde die abstraktere Sichtweise das Retrieval passender Bausteine aus einer umfangreichen Menge erleichtern, die aus dem potentiell sehr großen Markt zu erwarten ist. Die exakten Beschreibungsformen der Modellsicht stellen für diese Aufgabe eine Überspezifikation dar, die eher kontraproduktiv sein dürfte, denn gerade in der Entwurfsphase will man sich nicht auf exakte Details der Realisierung festlegen, sondern ist lediglich an einer bestimmten allgemeinen Funktionalität interessiert.

Als weitere Einsatzmöglichkeit ist die Nutzung von Metamodellen zur Manifestierung abstrakter Anwendungsframeworks denkbar. Mit ihrer Hilfe könnte — etwa im Sinne von *Designpatterns* [GHJV94] — eine Beschreibung allgemeiner, wiederverwendbarer Entwurfsinformationen von Komponentenstrukturen möglich werden. Ein solches Anwendungsskelett könnte dann mit Instanzen der grob umschriebenen Bausteinarten konkret ausgeprägt werden, wobei durch entsprechende Auswahl aus den relativ großen Klassen sehr weitgehende Freiheitsgrade resultierten. Die im Endeffekt betroffenen Bausteine ihrerseits wären durch die inhärente Fähigkeit des Customizings in der Lage, sich der potentiellen Varianz des Rahmenwerkes anzupassen.

4.3 Beschreibungsebene

Die Beschreibungsebene von Komponentenarchitekturen beinhaltet als wesentliche Bestandteile Spezifikationen — also Beschreibungen von Eigenschaften —, welche durch formale Systeme ausgedrückt werden und deren Sinn in der Abbildung informaler Konzepte der Modelleben besteht, um diese einer automatisierbaren Auswertung und Verarbeitung zuzuführen.

Im weiteren Verlauf werden zunächst die methodologischen Einflußfaktoren der Beschreibungsebene betrachtet. Im Anschluß folgen dann Untersuchungen der Bereiche Spezifikation, Formalisierung und Typisierung.

4.3.1 Einflußfaktoren

4.3.1.1 Entwurfsmethodik von Verbundkomponenten

Der Einfluß verschiedener Vorgehensweisen des Entwurfes von Komponentensystemen ist auf der Beschreibungsebene besonders ausgeprägt. Im Bezug auf die zwei in Abschnitt 4.1.3 gekennzeichneten Hauptrichtungen der Methodologie lassen sich folgende Einflüsse feststellen:

1. Einer **Top-Down Strategie** folgend kann eine (Verbund-) Komponente rein aus äußeren Vorgaben an die zu exportierende Funktionalität entstehen. Dies kann im Extremfall der Wunsch nach einer kompletten Anwendung sein. In der Regel werden die Vorgaben für Komponenten jedoch aus der funktionellen Dekomposition höherwertiger Komponenten entstehen.

   Bezüglich der Vorgabe einer Exportschnittstelle mittels vollständiger Spezifikation wird eine Implementation abgeleitet, aus der sich in gleicher Weise Anforderungen (d.h komplette Spezifikationen) für weitere Subkomponenten ergeben, ohne daß potentiell zur Verfügung stehende Bausteine in dieser Phase explizit berücksichtigt würden. Es entsteht hierbei die Problematik, aus den zusammengefaßten Anforderungen — entstanden aus der Implementation der Komponente als einer Einheit — eine Menge separater Subkomponenten zu ermitteln.

   Anforderungen in Form vorausgesetzter Schnittstellen, ausgedrückt mittels formaler Spezifikationen, können entweder durch Komponenten mit exakt passenden Gegenstücken erfüllt werden oder durch solche, deren Schnittstellen äquivalent zu den geforderten Spezifikationen sind. Diese Schnittstellenäquivalenzen sowie eventuell notwendige Adapterkomponenten lassen sich aus den formalen Spezifikationen ableiten.

2. Ein pragmatischerer **Bottom-Up-** bzw. **Framework-Ansatz** könnte ebenfalls von einer anzustrebenden Exportfunktionalität ausgehen, diese aber durch ausschließliche Komposition bereits vorhandener Komponenten zu erreichen suchen. Die spezifizierten Import-Funktionalitäten werden dann durch eine intern implementierte Koordinierung oder Kombination in die abstraktere Export-Funktionalität erhoben. Die formalen Spezifikationen von Exportfunktionalität, interner Implementation und einzelnen Importfunktionalitäten können dann dazu verwendet werden, die Korrektheit der so entstandenen Verbundkomponente zu verifizieren. Zu einem späteren Zeitpunkt ist es dann möglich, die Teilkomponenten im Rahmen einer speziellen Konfiguration oder Evolution durch spezifikationsäquivalente Einheiten zu ersetzen. Hier kommen wieder formale Spezifikationen zum Einsatz, welche bei Bedarf auch die Generierung entsprechender Adapter erlauben.

   Statt komplette vorhandene Komponenten bei der Definition einer neuen Verbundkomponente vorzugeben, könnte man alternativ auch von einer abstrakteren Vorgabe in Form eines Komponententyps ausgehen. Dieser Typ müßte von seiner Aussagekraft her lediglich strukturelle Vorgaben enthalten. Die Verwendung der durch Typen vorgegebenen Funktionalitäten könnte mit Hilfe semantischer Anforderungen komponentenintern zentral ausgedrückt werden. Beim Binden an eine Instanz des Typs könnten die Forderungen der Verbundkomponente gegen die entsprechenden Eigenschaften der Teilkomponente gegengeprüft werden, welche in deren Exportschnittstelle spezifiziert sind.

4.3.1.2 Kontextbezug und Anforderungsarten

Um eine korrekte — d.h. regelgerechte — Funktionsweise von Komponenten zu sichern, ist es sinnvoll, daß diese ihre Kontextbezüge explizit machen. Solche Kontextbezüge können dabei in Form von Anforderungen an externe Funktionalität in einer formalen, automatisiert verwertbaren Weise beschrieben werden. Es stellt sich die Frage, in welcher Form solche Anforderungsbeschreibungen vorliegen sollten. Dabei ergeben sich zwei Varianten, welche abermals in engem Zusammenhang mit der Entstehungsweise von Komponentensystemen stehen.

1. Als erster Gedanke bietet sich die Verwendung genau einer, bezüglich der Komponente globaler und im Wesen universellen Anforderung an. Dies entspricht dann der Offenlegung genau solcher Teile, welche in Bezug auf die Gesamtstruktur von globaler Bedeutung sind, was auch in [BW97] vorgeschlagen wird. So ein zusammengefaßtes, globales Requirement würde sich speziell bei der Top-Down Strategie ergeben, denn

aus übergeordneter Sicht ist die in der Regel erfolgende Aufteilung vorhandener Anforderungen auf eine Menge von Teilanforderungen, welche dann durch eine entsprechende Anzahl von Teilkomponenten erfüllt wird, nicht von Bedeutung. Diese Eigenschaft erweist sich aber gleichermaßen auch als Problematik des Ansatzes, denn die Art und Weise einer solchen Zerlegung — d.h. wie zerlegt werden könnte und sollte — bleibt dabei ungeklärt.

2. Als komplementäre Form ist eine Menge separater Teilanforderungen entsprechend verschiedener Importschnittstellen — also angedachter Teilkomponenten — denkbar. Es ist klar zu erkennen, daß diese Variante insbesondere dem Buttom-Up Ansatz entgegenkommt. Als offensichtliche Schwäche ergibt sich sofort das Fehlen einer Globalsicht — es stellt sich die Frage nach dem korrekten Zusammenspiel der Teilkomponenten.

Die komplementären Stärken und Schwächen der beiden Spezifikationsvarianten legen den Ansatz einer kombinierten Form nahe. Solche mehrschichtigen Spezifikationen sollten sich aber immer noch an der gewünschten Methodik orientieren und in diesem Sinne einen Schwerpunkt setzen, um eine ausreichende Pragmatik zu gewährleisten.

4.3.1.3 Integration von Refinement

In Sektion 4.1.3.2 wurde eine komponentenorientierte Form der Verfeinerung motiviert. Es stellt sich heraus, daß dadurch in Bezug auf die Spezifikation keinerlei Konsequenzen entstehen. Der spezielle Refinement-Ansatz könnte einfach mittels direkter Übernahme von Anforderungsspezifikationen (Importschnittstellen) als Funktionalitätszusicherungen (Exportschnittstellen) — eventuell erweitert und/oder mit veränderter Implementation versehen — erreicht werden. Es wäre somit theoretisch eine Integration in beliebige Ansätze der Beschreibungsebene möglich. Zu bedenken bleibt allerdings, daß trotz der unproblematischen Konsequenzen für Schnittstellen und Spezifikationen die schon weiter oben angesprochene Frage nach einer sauberen methodologischen Integration des Refinementgedankens in das Komponentenparadigma bedacht werden muß.

4.3.1.4 Modellannahmen

Um zu geeigneten Spezifikationsansätzen zu gelangen, sind zunächst einige wesentliche Punkte zu klären, die als Voraussetzungen für entsprechende Überlegungen einen konzeptionellen Rahmen abstecken. Zum einen muß

festgelegt werden, auf welche Weise die externen Spezifikationen von Komponenten bei deren Entwurf entstehen und zum anderen, wie die Anforderungen einer Komponente an ihre Subkomponenten und deren Spezifikationen zustande kommen. Antworten auf diese Fragen bestimmen die Struktur von Komponenten- bzw. Frameworkspezifikationen. Aus den Überlegungen der letzten Sektion lassen sich nun zusammenfassend die folgenden Aussagen treffen:

- Die — gewünschte — externe Funktionalität einer Komponente ist initial bekannt. Diese entspricht den bereitgestellten Diensten des Bausteins.
- Die funktionalen Anforderungen einer Komponente ergeben sich zum Entwicklungszeitpunkt, da sie von internen Implementationsentscheidungen beeinflußt werden.
- Zumindest die funktionalen Einheiten — etwa Nachrichtentypen — sind zum Entwurfszeitpunkt bekannt. Dies ist notwendig, um Spezifikationen eindeutig ausdrücken zu können.
- Die Partitionierung der geforderten Funktionalität in disjunkte Mengen, welche den Subkomponenten entsprechen, ist nicht notwendigerweise zum Entwurfszeitpunkt bekannt.

4.3.2 Spezifikation

Eine *Spezifikation* drückt in expliziter Form die Eigenschaften des Spezifikationsgegenstandes aus. Der Nutzen von Spezifikationen liegt darin, daß sie eine unmißverständliche Grundlage für die kommunikative Argumentation über Rahmenbedingungen darstellen. Dies gilt zum einen für das Propagieren eines Angebots und zum anderen für das Retrieval einer Nachfrage.

Die folgende Sektion untersucht in diesem Zusammenhang Aspekte der *Komponentenspezifikation.* Die zentrale Frage dabei ist:

> Welche Modellaspekte sollten spezifiziert werden, welche Ausdrucksmittel sind dafür geeignet, und wie sollten diese organisiert bzw. strukturiert werden?

4.3.2.1 Spezifikationsinhalte

Inhaltlich sollte eine Beschreibung von Komponenten zumindest zwei Teilaspekte enthalten. Als Mindestanforderung kann zunächst die in allen herkömmlichen Ansätzen enthaltene interne Strukturinformation, — d.h.

Schnittstellen und Signaturen von z.B. Funktionen, Prozeduren oder Methoden — angesehen werden. Darüber hinaus ist ein gewisses Maß an semantischer Information wünschenswert, um Bedeutung und Verhalten der Strukturelemente zu umschreiben.

Im folgenden wird zunächst der *Abstraktionsgrad* von Komponentenspezifikationen hinterfragt, d.h. es wird untersucht, wie detailliert eine entsprechende Beschreibung sein sollte. Im Anschluß folgt dann eine genauere Betrachtung konkreter Inhalte, welche sich in die Schwerpunkte *Struktur-* und *Verhaltensspezifikation* gliedert und am Ende kurz auf qualitative Aspekte eingeht.

Abstraktionsgrad Bei der Spezifikation von Komponenten können verschieden abstrakte Sichten eingenommen werden. Die Wahl der passenden Perspektive wird dabei durch die Spezifikationsanforderungen bestimmt: Zum einen soll die formale Beschreibung hinreichend sein, um die Anwendung einer Komponente zu ermöglichen, zum anderen soll aber von allen unnötigen Details abstrahiert werden, um keine zusätzlichen Abhängigkeiten zu schaffen. Man kann hier die entsprechenden Extreme der *Black-Box* und *White-Box* Ansätze unterscheiden. Black-Box Ansätze abstrahieren dabei vollkommen von allen internen Details, stellen aber u.U. eine zu sehr vereinfachte Sicht dar. White-Box Ansätze legen jegliche Details unmißverständlich offen, schaffen aber auf diese Weise starke Abhängigkeiten. Büchi schlägt in [BW97] alternativ eine "*Grey-Box*" Variante vor, welche neben der abstrakten, externen Sicht eine festzulegende Untermenge der Detailinformationen nach außen sichtbar macht und somit die Möglich bietet, genau solche Interna zu spezifizieren, deren Kenntnis für externe Anwendungen des Bausteins essentiell sind. Als grundsätzliche Möglichkeiten der Spezifikation von Komponenten sieht Büchi die Varianten

- Vor-/Nachbedingungen (engl.: *pre-/post-conditions*),
- algebraische bzw. denotionale Spezifikationen,
- abstrakte Anweisungen (engl.: *abstract statements*) sowie
- Invarianten und temporale Properties.

Er schlägt dabei abstrakte Anweisungen — z.B. in Form von Spracherweiterungen — als besten Ansatz für Grey-Box Komponenten vor.

Interne Strukturspezifikation Die komponenteninterne Struktur wird innerhalb eines Strukturmodells festgelegt. Sie umfaßt Gliederungseinheiten von Bausteinen, welche zur Organisation der extern wirksamen Aspekte

dienen. Das sind z.B. externe Zugangspunkte, angebotene und vorausgesetzte Dienste, Einheiten der Interaktion von möglicherweise verschiedener Granularität oder Bausteinattribute. Die konkreten Ausprägungen solcher strukturellen Elemente sind mannigfaltig. Für Interaktionseinheiten sind etwa Begriffe wie Funktion, Prozedur, Methode, Subroutine aber auch Ereignis, Exception, Nachricht und Kanal sowie viele weitere gebräuchlich. Für die Komponentenspezifikation eignen sich besonders abstrakte Begriffe, die auf möglichst viele konkretere Ausprägungen abgebildet werden können. Als grundlegende Bestandteile einer Spezifikationssprache bieten sich folgende Ausdrucksmittel als vielversprechende Möglichkeiten an:

- ***Komponenten*** und/oder
- ***Schnittstellen***, als strukturelle Einheiten für
- ***Nachrichten*** zur Modellierung der Kommunikation zwischen Komponenten mit
- ***Signaturen*** und/oder
- ***Typen(-Information)*** zur Modellierung von Parametern/Daten und
- ***Attribute*** als Hilfsmittel zur Einführung unterstützender Zustände

Die Wahl von *Nachrichten* zur Modellierung der Kommunikation zwischen Komponenten erlaubt durch ihre Abstraktheit eine Abbildung auf viele konkrete Konstrukte wie Operationen (Funktionen, Prozeduren ...), Ereignisse, Ausnahmen etc. Insbesondere können sowohl synchrone als auch asynchrone Kommunikationsprozesse beschrieben werden. Es stellt sich noch die Frage, ob Nachrichten Parameter beinhalten sollen, oder ob Parameter als eigenständige Nachrichten modelliert werden. Die Berücksichtigung von *Daten(-Typen)* ist nicht zwingend notwendig, ermöglicht aber eine reichere und ausdrucksstärkere Beschreibung.

Externe Strukturspezifikation Neben der internen Komponentenbeschreibung stellt sich die Frage, wie und ob die externe Struktur, d.h. Beziehungen von Komponenten in einer Komponentenstruktur, durch die Spezifikationen beschrieben werden sollen. Beziehungen zwischen Komponenten können den gegenseitigen Bezug auf Strukturelemente — zumeist Interfaces — sowie eine Vorgangsbeschreibung der Interaktion beinhalten. In der Literatur finden sich zu dieser Thematik verschiedene Beiträge mit unterschiedlich abstrakten Vorgehensweisen:

Olaffson beschreibt in [OB97] die Beziehungen zwischen Komponenten in einem integrierten Ansatz durch *vorausgesetzte Schnittstellen (engl. Required Interfaces)* [OB97]. Murer schlägt dagegen zur Spezifikation der einzelnen Aspektbereiche *Schichten verschiedener Interoperabilitätsinformation*

[MSW97] vor. Als dritten Ansatz verwenden Mens und Hondt eine separate Beschreibung der von einer Komponente genutzten Funktionalität durch *Reuse Contracts*, welche von konkreten Schnittstellen abstrahieren und so auch deren potentielle Evolution einbeziehen [MSL97, HLS97]. Als gangbarer Ansatz soll im folgenden insbesondere das Prinzip der vorausgesetzen Importschnittstellen hervorgehoben werden.

Verhaltensspezifikation Strukturbeschreibungen bieten für sich genommen noch keine ausreichende Spezifikation von Komponenten, da sie keine Hinweise auf Bedeutung und Funktion der Koponentenbestandteile enthalten. Semantik und Verhalten stellen aber wichtige Aspekte von Komponenten dar und sollten in deren Spezifikation einfließen, um die Voraussetzungen für eine automatische Auswertung zu schaffen.

Bei der Beschreibung von externer Komponentenfunktionalität lassen sich nun verschiedene Verhaltensbereiche identifizieren, in die sich das Gesamtverhalten aufteilt. Zunächst kann grob zwischen bereitgestellter Export- und vorausgesetzter Importfunktionalität unterschieden werden. Diese beiden Oberbereiche untergliedern sich auf gleiche Weise in die folgenden drei Unterkategorien verschiedener Verhaltensaspekte:

- Allgemeine, zentrale Komponenteneigenschaften
- Einzelne Organisationseinheiten (z.B. *export bzw. required Interfaces*)
- Wechselseitige Abhängigkeiten der Organisationseinheiten (z.B. *Synchronisation*)

Die Thematik der Organisation von Teilgebieten der Komponentenbeschreibung in Spezifikationsstrukturen wird weiter unten noch genauer behandelt. Bezüglich der einzelnen Bereiche sind nun die Eigenschaften der Handlungsweisen zu beschreiben. Solche Eigenschaften können sich etwa aus Synchronisationsbedingungen (z.B. "Critical Sections" bzw. exklusive Ressourcen) oder Anwendungslogik (z.B. `not read_file until open_file`) ergeben.

Da solche Eigenschaften formal schwer zu fassen sind, muß man sich hierbei auf Teilaspekte beschränken. Als sinnvolle dynamische bzw. semantische Spezifikationsbestandteile bieten sich vor allem temporale bzw. kausale Beziehungen an. Solche Beziehungen könnten in Bezug auf strukturelle Bestandteile formuliert werden, wobei sich hier besonders die grundlegenden funktionalen Einheiten anbieten. Auf diese Weise lassen sich dann beispielsweise die sinnvollen Abfolgen von Methodenaufrufen beschreiben. Um komplexere Verhaltensaspekte zu behandeln, kann ein Zustandsbegriff eingeführt werden, welcher Inhalt des nächsten Paragraphen ist.

Die Rolle von Zuständen Der Begriff des Zustandes spielt eine wichtige Rolle bei der Beschreibung semantischer Komponentencharakteristik, da er eine nützliche Abstraktion zur Spezifikation von Verhaltensweisen darstellt. Zustände sind in diesem Sinne Schnappschüsse der dynamisch veränderlichen Aspekte eines Bausteins. Eine Verhaltensweise kann dann durch ihre Wirkung auf diese veränderbaren Eigenschaftswerte dargestellt werden, indem man den Zustand vor und den Zustand nach ihrer Aktivität angibt.

Bei der Rolle von Zuständen innerhalb der Komponentenspezifikation kann man zwei Fälle unterscheiden. Zum einen können Zustände implizit integriert sein, falls ein Formalismen genutzt wird, welcher grundsätzlich auf Zuständen beruht (z.B. Automatenmodelle). Hierbei müssen oft zwanghaft Zustände bestimmt werden, die keine direkte Beziehung zum Inhalt aufweisen. Zum anderen kann auch eine explizite Einführung von Zuständen mit direkter inhaltlicher Beziehung stattfinden. Diese sind dann der verständlicheren, effektiveren Spezifikation hilfreich, ohne daß dazu ein Zwang durch den zugrundeliegenden Formalismus bestünde.

Im Zusammenhang mit Zuständen kommt man zu einer Reihe teilweise offener Fragestellungen. Zunächst sollte man den Zusammenhang zwischen den idealen, modellhaften Zuständen einer konzeptionellen Komponente und den vereinfachten Spezifikationszuständen beachten. Es stellt sich hier die Frage, ob solche Zustände verschiedener Ebenen identifiziert werden können oder ob diese eher unabhängig voneinander sind. Des Weiteren kann man auch Spezifikationen ohne Zustände, welche temporal allgemeingültig auf rein externer Perspektive beruhen, in Erwägung ziehen. Es ist zu prüfen, inwiefern solche Ansätze zu adäquaten Abbildungen führen und wo deren konkrete Vorteile liegen.

Spezifikation nichtfunktionaler Eigenschafen Nicht alle bedeutsamen Eigenschaften von Komponenten sind quantitativer Natur. Wichtige Faktoren wie Performanz, Wiederverwendbarkeit oder Verläßlichkeit sind qualitative Begriffe, welche nicht ohne weiteres in diskreten Skalen zu erfassen sind. Bei der Konzeption eines geeigneten Spezifikationsansatzes muß in Erwägung gezogen werden, solche Größen einfließen zu lassen und somit zum Gegenstand automatisierbarer Methoden zu machen. Das entsprechende Forschungsgebiet ist allerdings noch in einer relativ frühen Phase, und pragmatische Lösungen, welche sich auch für reale Systeme eignen würden, existieren nach Wissen des Autors z.Z. nicht. Eine in diesem Zusammenhang erwähnenswerte Arbeit ist der *NoFun-Ansatz*, welcher in [Fra97] beschrieben wird.

4.3.2.2 Spezifikationsstruktur

Nachdem nun umrissen ist, welche Inhalte eine Komponentenspezifikation haben sollte, ist im nächsten Schritt deren Struktur zu untersuchen. Zunächst sei dabei vorausgesetzt, daß eine Spezifikation in separater Form gemeinsam mit dem durch sie beschrieben Baustein vorliegt, und nicht etwa eine globale Beschreibung der Gesamtstruktur bzw. Anwendung alle komponentenbezogenen Teilaspekte zentral integriert. Dies ist damit begründet, daß Komponenten prinzipiell für sich abgeschlossene Einheiten bilden und sich in verschiedene Strukturen einbinden lassen sollen. Die hier zu betrachtenden Spezifikationen stehen also immer in direktem Zusammenhang zu einem einzelnen korrespondierenden Baustein bzw. Bausteintyp. Im folgenden wird nun die strukturelle Organisation solcher Beschreibungen betrachtet, wobei als erstes der enge Zusammenhang mit dem Schnittstellenbegriff beleuchtet wird. Im Anschluß an Überlegungen zu integrierten Zentralspezifikationen endet der Abschnitt mit dem Gedanken einer externen Spezifikationsverteilung.

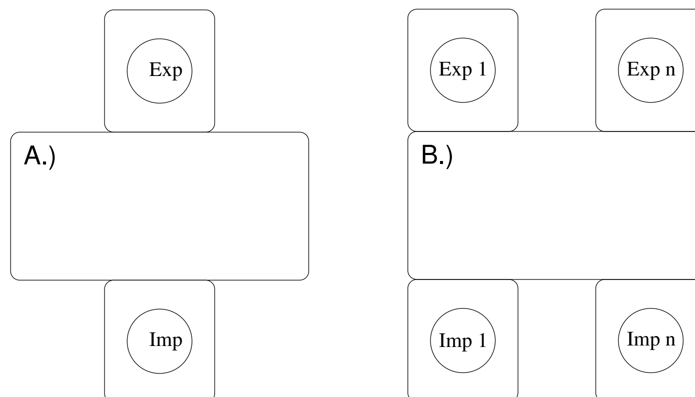


Abbildung 4.7: Schnittstellenspezifikation

Schnittstellenspezifikation Die Struktur von Komponentenspezifikationen steht in engem Zusammenhang mit dem Begriff der Schnittstelle. Auf Modellebene manifestieren sich Komponenten nach außen grundsätzlich über solche Schnittstellen als einzige definierte Bezugspunkte. Dies hat den Hintergrund, daß alle komponentenexternen Interaktionsbeziehungen dann in expliziter Weise über einheitliche Wege ablaufen und dadurch einer geregelten Form unterliegen. Die Schnittstellen müssen daher in diesem Bild gleichermaßen Träger — oder zumindest Zugangspunkt — und eigentlicher Gegenstand der Spezifikationen sein. In diesem Sinne stellt sich Komponentenspezifikation als Schnittstellenspezifikation dar. Abbildung 4.7 illustriert den Fall von Schnittstellen als jeweilige Träger ihrer eigenen separaten Spe-

zifikation.

In Abschnitt 4.2.1.3 wurden nun zwei mögliche Varianten der Schnittstellenmenge pro Komponente diskutiert. Diese Varianten wirken sich auch auf die Struktur einer Schnittstellenspezifikation aus.

Der Ansatz mehrerer Export- und/oder Importschnittstellen pro Komponente stellt wie gesehen einen äußerst pragmatischen Ansatz mit guten Gliederungsmöglichkeiten und der Unterstützung multipler Sichten/Rollen sowie abwärtskompatibler Evolution dar.

Aus Spezifikationssicht kommt nun ein wichtiger Aspekt hinzu: verschiedene Schnittstellen sollten nicht grundsätzlich unabhängig voneinander sein, da sonst außer organisatorischen Aspekten kein Vorteil gegenüber mehreren Komponenten mit jeweils einer Schnittstelle besteht. Bei mehreren Schnittstellen ist somit die wechselseitige Synchronisation der Funktionalitäten zu beachten.

Als Beispiel möge folgende Situation dienen: Eine technische Komponente verfüge über verschiedene Schnittstellen für Betrieb (`B`) und Wartung (`W`). Diese können genau dann nicht unabhängig voneinander agieren, wenn Betrieb und Wartung sich wechselseitig ausschließen. In diesem Fall müssen die entsprechenden Dienste etwa in folgender Weise schnittstellenübergreifend synchronisiert werden:

```
B:startWorking → B:stopWorking → W:startService →
W:stopService → B:startWorking ...
```

Für die Konzeption einer diesbezüglichen Spezifikation ist die Frage von Bedeutung, ob sich die Synchronisation der multiplen Exportinterfaces auf das externe Verhalten von Komponenten auswirken kann. Wenn ja, müssen die Synchronisationsbedingungen in die Modellspezifikationen integriert werden. Dies könnte durch einen zentralen Synchronisationsteil erfolgen, wie bei der Komponente D.) in Abbildung 4.8 veranschaulicht wird.

Zu bedenken ist weiterhin der Fall multipler Komponentennutzung, bei dem verschiedene Exportschnittstellen von verschiedenen Verbundkomponenten genutzt werden. Es ist dann zu beachten, daß die theoretische Lebendigkeit der Komponente bei Inter-Export-Beziehungen (Komponentennutzung durch verschiedenen Strukturen/Anwendungen) gewährleistet werden muß. Eventuell müßten Inter-Export-Beziehungen dieser Art in einer restriktiveren Form vorliegen.

Als zweite Möglichkeit wurde genau eine Export- und Importschnittstelle pro Komponente in Erwägung gezogen. Bei diesem Ansatz treten die Probleme der Spezifikation bei multiplen Schnittstellen naturgemäß nicht auf. Er führt dementsprechend zu unkomplizierteren Architekturen. Trotzdem sollte auch in diesem Fall nicht auf einen Synchronisationsteil verzichtet

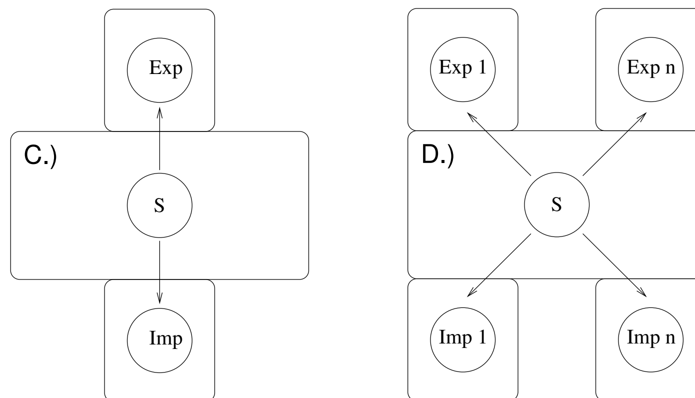


Abbildung 4.8: Synchronisierte Einzel- und Multischnittstellenspezifikation

werden. Dessen Aufgabe ist es, hier die Beziehungen zwischen dem Import- und dem Exportteil explizit zu machen (Abb. 4.8 Komponente C.)).

Zentrale Spezifikationen Komponentenspezifikationen enthalten, wie gesehen, Beschreibungen von Schnittstellen. Es liegt daher nahe, für die einzelnen Schnittstellen eines Bausteins Spezifikationen aufzustellen, welche dann zusammen eine disjunkte Menge separater Teilbeschreibungen ergeben. Wie oben ausgeführt, fehlen bei diesem Ansatz die wechselseitigen Abhängigkeiten einzelner Schnittstellen, was besonders bei multiplen Schnittstellen zu unangemessenen Resultaten führt. Es sind also zumindest zusätzliche Beschreibungen für die Synchronisation abhängiger Teilschnittstellen zu ergänzen. An diesem Punkt stellt sich die Frage, ob Komponentenspezifikationen überhaupt einen Bezug zum Schnittstellenbegriff benötigen, oder ob eine einzelne zentrale Gesamtspezifikation nicht alle wesentlichen Merkmale auszudrücken vermag.

Bestärkt wird obige Frage durch die Tatsache, daß ein abstraktes, formales Modell prinzipiell vollkommen auf Schnittstellen verzichten kann und sich auf die ausschließliche Betrachtung einer Komponente als Ganzes und ihrer kommunikativen Basiseinheiten (z.B. Nachrichten) beschränken läßt. Eine Deklaration von Schnittstellen als Teilmengen der gesamten Nachrichtenmenge ist aus dieser Sicht rein organisatorischer Natur. Die Spezifikation von semantischen Abhängigkeiten zwischen verschiedenen Schnittstellen — konkreter deren Kommunikationseinheiten — ist nicht auf diese Organisationsstruktur angewiesen. So eine Zentralspezifikation (siehe Abbildung 4.9) kann die Komponentenbeschreibung ferner in verschiedenen Gebieten bereichern. Sie ist etwa in der Lage, Aspekte zu berücksichtigen, welche sich aus der internen Implementation des Bausteins ergeben und nicht in direktem Zusammenhang zu einzelnen Schnittstellen stehen, oder bei denen diese Beziehung nicht direkt explizit ausgedrückt werden soll. Weiterhin bietet sie

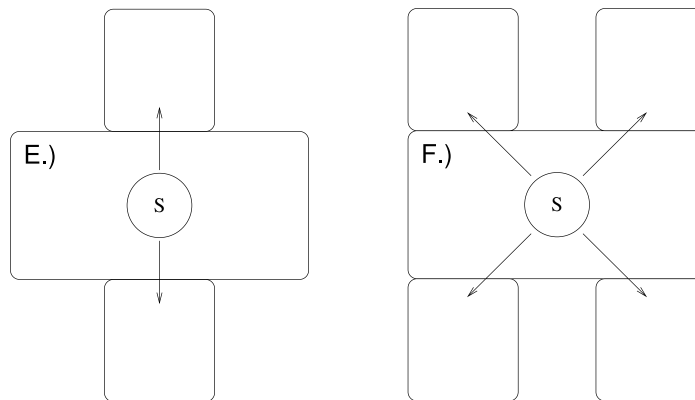


Abbildung 4.9: Zentralspezifikation

eine umfassendere Sicht, mit der die schnittstellenübergreifenden Zusammenhänge übersichtlicher und eindeutiger formuliert werden können.

Ein problematischer Aspekt der Zentralspezifikation liegt hingegen darin, daß bei Komponentensystemen generell Teilspezifikationen in Bezug auf einzelne Schnittstellen notwendig sind. Da Komponentenanwendung jeweils bestimmte einzelne Schnittstellen nutzt, ist ein Abgleich mit genau deren Eigenschaften notwendig. Teilspezifikationen müssen also entweder abgeleitet werden, oder ein Spezifikationsabgleich muß immer generell über die Zentralspezifikation als Ganzes geschehen. Beide Varianten bedeuten zusätzlichen Aufwand.

Auslagerung von Anforderungspezifikationen Eine Möglichkeit, externe Beziehungen von Komponenten auszudrücken, ist — wie gesehen — die Spezifikation von geforderten Eigenschaften etwa mittels Import- oder vorausgesetzten Schnittstellen. Bezüglich der Zuordnung dieser Spezifikationen zu Komponenten sind nun verschiedene Möglichkeiten zu unterscheiden.

Zunächst kann eine Verbundkomponente mit Anforderungen an ihre konstituierenden Teilkomponenten deren Spezifikationen exklusiv beinhalten. Es existieren dann zwei separate Spezifikationen von zum einen geforderter Import- und zum anderen erbrachter Export-Funktionalität. Die Interaktions- bzw. Kompositionsfähigkeit der Partner kann dann mittels Abgleich der Spezifikationen geprüft werden.

Als zweite Alternative ist ein referentieller, verteilter Ansatz denkbar, bei dem die Verbundkomponente ihre Forderungen mittels Verweis auf die Spezifikation einer konstituierenden Teilkomponente ausdrückt. Eine solche Referenz könnte entweder direkt erfolgen, wobei der entsprechende Teilbaustein — in diesem Fall natürlich im Sinne einer Bottom-Up Strategie (4.3.1.1) — initial existieren muß, oder auf indirekte Weise etwa eine abstraktere externe

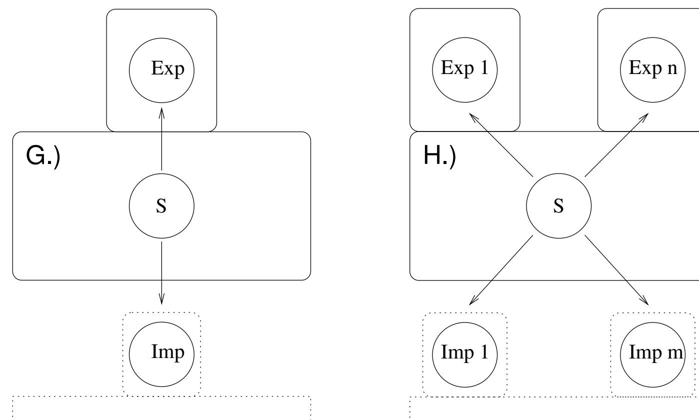


Abbildung 4.10: Externe Importspezifikation

Typbeschreibung nutzen. Abbildung 4.10 veranschaulicht diese Externalisierung von Anforderungsspezifikationen für den Fall einer direkten Referenz

4.3.3 Formalisierung

Für die Formalisierung von Komponentenkonzepten bieten sich verschiedene Ansätze an. Neben klassischen Automatenmodellen scheinen vor allem logikbasierte Ansätze erfolgversprechende Wege zu eröffnen.

Im folgenden wird zunächst die Verwendung formaler Theorien im Zusammenhang mit Komponentenarchitekturen kritisch hinterfragt und schließlich gerechtfertigt. Im Anschluß folgen dann Überlegungen zu möglichen Formalisierungsansätzen, die sich grob in strukturelle und semantische Bereiche gliedern. Innerhalb dieser Bereiche wird jeweils ein spezieller Ansatz hervorgehoben.

4.3.3.1 Warum Formalismen?

Bevor im Anschluß konkrete Möglichkeiten der Formalisierung betrachtet werden, sollen zunächst Fragen nach zugedachter Rolle, gewünschter Wirkung und potentiellen negativen Begleiterscheinungen formaler Methoden in Komponentenarchitekturen zur Sprache kommen.

Für die Anwendung von Formalismen spricht die Tatsache, daß diese zur beweisbaren Korrektheit von Systemen bezüglich verschiedener, alternativer Aspekte führen und damit die Voraussetzungen für komplette und gleichzeitig gesicherte Abstraktionen schaffen. Dadurch kann ein Retrieval von beweisbar korrekten Komponenten unter weniger strengen Recherchebedingungen für informalere Modelle stattfinden, wobei die Möglichkeiten von Teilspezifikationen bzw -verifikationen und deren Kombination besteht. Im

Anschluß kann eine gesicherte, automatisierbare Komposition erfolgen, welche in direkter Weise zu neuen, komplett spezifizierten Bausteinen führt und so auch Wiederverwendbarkeit fördert. Als positiver Nebeneffekt führt der Vorgang einer formalen Beschreibung zu einem besseres Systemverständnis. Des weiteren stellen Formalismen eine Basis für Standardisierungen als Voraussetzung für offene Märkte dar.

Den genannten offensichtlichen Vorteilen stehen eine Reihe möglicher Nachteile formaler Methoden gegenüber. Allen voran ist dabei das Akzeptanzproblem zu nennen, welches den Einsatz entsprechender Ansätze häufig verhindert. Der Widerstand entsteht dabei aus dem unvorteilhaften, oft als unangenehm und lästig (*"look and feel"*) empfundenen formalen Entwicklungsprozeß von Software, welcher mit dem Einsatz entsprechender Methoden einhergeht. Unter ungünstigen Umständen kann dabei der einhergehende informale Entwicklungsprozeß behindert werden. Nicht zu unterschätzen ist daneben der prinzipielle rechnerische Aufwand von deduktivem Retrieval und Beweisen.

Formalismen dienen im allgemeinen dazu, exakte Aussagen über Eigenschaften von Systemen machen zu können. Der wohl wichtigste Aspekt dabei ist, daß mit Formalismen die Korrektheit eines durch sie spezifizierten Systems bewiesen werden kann. Im Gegensatz dazu steht die Alternative der Verifikation von Systemen mit Hilfe von Testreihen und Simulationen[1]. Beide Ansätze haben ihre spezifischen Vor- und Nachteile, weshalb keiner ganzheitlich zu überzeugen vermag. Wo auf der einen Seite die vollständige rationale Formalisierung zu untragbarem Aufwand führt, können auf der anderen Seite empirische Tests in sensitiven Bereichen nicht zur notwendigen, vollkommenen Sicherheit führen (siehe Ariane Unfall [Lio96]). Als möglicher Ausweg könnte ein Kompromiß angestrebt werden, welcher die Ansätze in ergänzender Weise vereint.

Trotz der ernstzunehmenden Schattenseiten steht die letztendliche Vorteilhaftigkeit einer Integration formaler Methoden in Komponentenarchitekturen außer Frage. Es muß jedoch als wesentliche Voraussetzung deren praxisgerechte Anwendung bedacht werden. Zunächst ist daher zu fordern, daß die zur Anwendung kommenden Methoden in einem Rahmen *menschlich* überschaubarer Komplexität verbleiben, der zu einem möglichst einfachen Verständnis beiträgt. In jedem Fall sollte dafür eine automatisierte Unterstützung durch *Systemwerkzeuge* integriert werden. Diese sollte unter anderem auch eine Wiederverwendung von Spezifikationen bzw. Spezifikationsmustern gestatten und begünstigen, um den schwierigen, komplexen Spezifikationsprozeß effektiver zu gestalten.

[1] Es sei bemerkt, daß diese unterschiedlichen Ansätze in direkter Analogie zu den konträren philosophischen Strömungen von Rationalismus und Empirik [Stö98] stehen. Sie wurden im 18. Jahrhundert von Immanuel Kant zu einer einheitlichen Sicht vereinigt, was als Denkanstoß für analoge Überlegungen innerhalb der Informatik dienen könnte.

4.3.3.2 Strukturelle Aspekte

Die Struktur einer Komponente drückt sich in der Beschreibung ihrer Schnittstelle(n) aus, welche lediglich Strukturaspekte der Komponenten in Form von herkömmlichen überwiegend operationalen Schnittstellenbeschreibungen mit den darin enthaltenen Signaturen umfassen. Dies wird meist durch syntaktische Elemente einer entsprechenden *Beschreibungssprache* erreicht. Solche Schnittstellenbeschreibungssprachen (*Interface Definition Languages IDL*) führen dabei zu einer Auslagerung der Strukturinformationen aus den eigentlichen Artefakten — den Komponenten —, welche dann allesamt als separate Teile vorliegen. Allgemein läßt sich sagen, daß IDL gleichermaßen weit verbreitet wie ausgereift sind und im allgemeinen auch für Komponentenansätze befriedigende Lösungen darstellen.

Statische Strukturbeziehungen zwischen Komponenten können durch solche Beschreibungsformen automatisiert verifiziert werden. Bekannte Schwachstelle struktureller Spezifikationsformen ist, wie schon früher erwähnt, das Fehlen eindeutiger semantischer Verhaltensaspekte. So ist es möglich, daß strukturgleiche Komponenten ein voneinander abweichendes Verhalten zeigen, und eine durch die Struktur implizierte scheinbare Konformität bei Berücksichtigung der Verhaltenssemantik keineswegs vorhanden ist. Für Komponentenarchitekturen folgt hieraus die Tatsache, daß strukturelle Beschreibungen in jedem Fall durch Verhaltensbeschreibungen ergänzt werden sollten.

Ein spezieller Nachteil der angesprochenen IDL ist die relative Starrheit der resultierenden Spezifikationen. Strukturelle Eigenschaften werden direkt und absolut kodiert, weshalb spätere Änderungen zu Problemen führen können. Als flexibler Ansatz für dynamischere Spezifikationen struktureller (und weiterer) Aspekte könnte ein auf *Feature-Logik* basierendes Konzept dienen.

Feature-Logik Feature-Logik ist eine typisierte Logik, die bei der Formalisierung von flexiblen (Struktur-)Merkmalen einer Komponente hilfreich sein könnte. Die Terme der Feature-Logik beschreiben Objektmengen mit bestimmten Eigenschaften (engl. *Features*) sowie deren Beziehungen. Diese Terme können dazu verwendet werden, Objekte — also auch Komponenten sowie deren konstituierende strukturelle Einheiten — mit bestimmten Eigenschaften dynamisch zu selektieren und einer weiteren Verarbeitung zuzuführen. Der Ansatz wird im nächsten Kapitel eingehend untersucht.

4.3.3.3 Formale Semantik

Bei der Auswahl geeigneter Formalismen liegt ein wesentliches Kriterium — wie schon mehrfach angesprochen — in der Fähigkeit, semantische Ei-

genschaften adäquat darstellen zu können. Formale Systeme werden unter diesem Gesichtspunkt allgemein in drei Gruppen eingeteilt:

- **Algebraische (denotionale) Semantik** — Denotionale Semantiken bedienen sich streng mathematischer Modelle. Das Verhalten von Operationen einer Komponente wird durch algebraische Formeln ausgedrückt. Ein Beispiel für algebraische Semantik im Kontext von Komponentenmodellen ist der Ansatz von Broy [Bro96, Bro98]. Dort werden Schnittstellen durch typisierte Kanäle beschrieben, in denen zeitlich getaktete Ströme (*timed streams*) in Komponenten hinein- bzw. hinausfließen. Semantisch werden die Komponenten durch eine Menge von Funktionen zwischen den Input- und Output-Kanälen spezifiziert. Mit diesem Modell lassen sich dann auch Komposition und Verfeinerung von Komponenten sowie beliebige Mischformen daraus darstellen.

- **Axiomatische Semantik** — Bei dieser Gruppe von Formalismen erfolgt eine Deklaration semantischer Aspekte durch Axiomensysteme, welchen das betrachtete System (z.B eine Komponente) genügen muß. Prominentes Beispiel hierfür sind Pre- und Postconditions. Die semantischen Constraints temporaler Logik fallen ebenso in dieses Gebiet. Axiomatische Semantik erlaubt die Validierung von Systemen bezüglich ihrer Spezifikationskonformität.

- **Operationale Semantik** — Bei operationalen Ansätzen erfolgt die Beschreibung der Semantik eines Systems durch das korrespondierende Verhalten einer grundlegenden Basismaschine. Genauer werden die Zustandsübergänge eines solchen Referenzsystems herangezogen. Diese drücken per Definition eindeutige und von ihrer Bedeutung her klar verständliche Verhaltensaspekte aus. Etwas pragmatischer kann eine einfache, klare und wohlverstandene Programmier-, Pseudo- oder Meta- Sprache verwendet werden, um die Semantik komplizierter, fremdartiger oder zu vergleichender Systeme in eine bekannte, begreifbare Form zu überführen.

Für Komponentenarchitekturen eignen sich besonders axiomatische und operationale Semantiken, da sie pragmatische Lösungen erlauben. Die Verwendung von Zusicherungen ersterer Kategorie hat sich im Bereich der Programmiersprachen — etwa in Eiffel — bewährt [Mey88]. Schnittstellen von Komponenten werden dabei durch zusätzliche Anforderungen ergänzt, welche die Erwartungen und Versprechen der beteiligten Interaktionspartner ausdrücken.

Büchi schlägt hingegen speziell für Komponentenspezifikationen eher abstrakte Anweisungen aus dem Bereich der operationalen Systeme vor [BW97].

4.3.3.4 Der zeitliche Aspekt

Um Anwendungsvorgänge von Komponenten darzustellen, sind besonders die kausalen Beziehungen bzw. zeitlichen Abläufe zwischen deren grundlegenden funktionalen Struktureinheiten von Interesse, welche im folgenden auch als *zeitlicher* oder *temporaler Aspekt* bezeichnet werden sollen.

Endliche Automaten Um solche temporalen Aspekte auszudrücken, bedienen sich Ansätze aus verschiedenen Bereichen oft dem mathematischen Modell des *endlichen Automaten (EA)* [HU93]. Die relative Häufigkeit dieser Lösung hat zwei wesentliche Gründe: Zum einen sind EA leicht verständlich und zum anderen einfach durch Zustandsdiagramme darzustellen.

Die Verwendung von Automatenmodellen erfolgt im allgemeinen nach folgendem Schema [Ara95]: Ein EA A_K wird verwendet, um den temporalen Aspekt einer Ausprägung (etwa eines Objektes oder einer Komponente) des Typs T zu modellieren. Transitionen des EA A_K werden nun mit Operationen beschriftet, welche von Instanzen des Typs T ausgeführt werden sollen. Die Zustände des EA korrespondieren dann mit den Zuständen der Instanz. Eine Transition von A_K, welche mit p beschriftet ist und zwischen Zustand z_1 und Zustand z_2 liegt, modelliert den Umstand, daß eine Operation p bei einer Instanz o von T genutzt werden kann, wenn sich o im Zustand z_1 befindet. Nachdem die Operation aufgerufen wurde, wechselt o in den Zustand z_2. Der temporale Aspekt des Instanzverhaltens wird somit durch Paare *(Zustand, Operation)* beschrieben.

Eine wesentliche Eigenschaft dieses Ansatzes ist die beschränkte Rolle der temporalen Aspekte als Beschreibung einer Folge von Operationen und Zustandsübergängen von Ausprägungen bestimmten Typs. Eine aktive Spezifikation der Art und Weise, in der eine Instanz Operationen ausführt, liegt außerhalb des konzeptionellen Rahmens der Methode. Des weiteren sollte die Beschreibung temporaler Aspekte möglichst orthogonal zu den übrigen Prinzipien von Ansätzen verlaufen, um ein harmonisches Zusammenspiel zu gewährleisten.

Die Nachteile von Ansätzen, welche auf EA basieren, lassen den Wunsch nach Alternativen aufkommen. Ein zur Beschreibung zeitlicher Aspekte potentiell besser geeigneter Formalismus ist durch die *Temporale Logik* gegeben.

Temporale Logik *Temporale Logik* ist eine Erweiterung der *modalen Logik*, die potentielle Aussagen der Form "möglicherweise", "unter Umständen", etc. beinhaltet. Letztere werden um qualitative zeitliche Aspekte, welche informal etwa Bedeutungen wie "ab jetzt", "später einmal", "als nächstes" oder "solange bis" besitzen ergänzt. Temporallogik beschreibt so die sich wandelnde Wahrheit logischer Ausdrücke im Laufe der Zeit.

Die Einordnung temporaler Logik in das Schema formaler Semantiken ist nicht ganz unproblematisch. Der Formalismus basiert auf logischen Aussagen, welche sicherlich zur Gruppe der axiomatischen Semantik hinzuzuzählen sind. Andererseits kann die temporale Logik aber auch als formale Sprache mit eindeutiger Semantik aufgefaßt werden, was wiederum eine Zuordnung zur operationalen Semantik rechtfertigen würde.

Temporale Aussagenlogik hat in Bezug auf die Beschreibung zeitlicher Aspekte von Komponenten eine Reihe vorteilhafter Eigenschaften [Ara95]. Im Gegensatz zu Ansätzen mit endlichen Automaten ist es möglich, die temporalen Eigenschaften eines Objektes unmittelbar zu beschreiben, wobei Sequenzen funktionaler Struktureinheiten beschrieben werden, welche gesendet und empfangen werden. Es besteht bei diesem Ansatz nicht der Zwang, Zustände anzugeben, deren einziger Zweck darin besteht, dem Formalismus zu genügen. Trotzdem besteht die Möglichkeit, nach freier Entscheidung sinnvolle Zustände zu definieren, die dem besseren Verständnis zuträglich sind und die Verhaltensbeschreibung erleichtern.

Mit dem umschriebenen Ansatz gibt man in aktiver Weise die gewünschten semantischen Eigenschaften eines Bausteins vor, anstatt, wie in Beschreibungen mit Automaten üblich, in passiver Weise die inherenten Eigenschaften einer Implementation abzuleiten. Semantische Aussagen in Gestalt temporallogischer Formeln lassen sich dabei leicht formulieren und lesen, da sie verständlich und überschaubar sind.

Bezüglich ihrer grundsätzlichen Eigenschaften läßt sich abschließend feststellen, daß die im Komponentenkontext relevante Form der temporalen Aussagenlogik grundsätzlich entscheidbar ist. D.h. semantische Aussagen in Form von logischen Formeln lassen sich immer verifizieren. Zu diesem Zweck existieren effiziente Algorithmen zur Verifizierung der Erfüllbarkeit von Formeln in temporaler Aussagenlogik.

4.3.4 Typisierung im Komponentenkontext

Typisierung ist im Bereich der Programmiersprachen eine weit verbreitete Technik. Es stellt sich nun die Frage, ob diese auch im Rahmen von Komponentenarchitekturen eine nutzbringende Ergänzung darstellen würde, und welche Rolle ihr hierbei zugedacht werden soll.

Ein erster wesentlicher Faktor ist die nützliche Tatsache, daß Typen *Invarianten* ihrer Instanzen darstellen. Daraus resultiert eine einfache Verifizierbarkeit korrekter Beziehungen zwischen Bausteinen anhand ihrer Typen, was allgemein als Typsicherheit bezeichnet wird. Gerade im Zusammenhang mit Komponenten, die mit ihrer Bestimmung zur Kompositionen ein ständiger Gegenstand wechselseitiger und potentiell wechselhafter Beziehungen sind, erscheint die Möglichkeit eines entsprechenden automatischen *Typecheckings*

äußerst nützlich.

Durch Typisierung fließen zusätzliche (Typ-)Informationen in ein System ein. Diese Informationen erlauben es, bei der Auswertung — i.A. Kompilierung — *Optimierungen* vorzunehmen, welche ohne sie nicht möglich wären. In diesem Sinne führt Typisierung zu sowohl *effizienteren* als auch *performateren* Systemen. Diese Eigenschaft ist für Komponentenansätze sehr wichtig, um trotz ihrer hohen Komplexität eine praktisch sinnvolle Verwendbarkeit zu ermöglichen.

Für die beiden genannten Faktoren ist es im Zusammenhang zur Komponentenorientierung besonders wünschenswert, daß Typinformationen beim Übergang des Entwicklungsstadiums in den *laufenden Betriebszustand* erhalten bleiben, da ein großer Teil der Funktionalität einer Komponentenarchitektur zur Laufzeit benötigt wird. Dementsprechende generelle Möglichkeiten sind etwa durch entsprechende Fähigkeiten des Compilers oder generell bei Skriptsprachen gegeben.

Die Komposition wird noch durch andere Eigenschaften von Typen unterstützt. Da Typen ihrerseits in verschiedenartigen Beziehungen stehen, und solche einen direkten Bezug zu denen der entsprechenden Instanzen aufweisen, lassen sich nützliche Rückschlüsse ziehen. Die Einführung von *Subtypbeziehungen* und *Typhierarchien* eröffnet hier die Möglichkeit einer *Typsubstitution*, welche zu erheblich größerer Flexibilität bei der Komposition führt. Die Unterscheidung von *Typkonformität* und *Typinteroperabilität* erlaubt hierbei Rückschlüsse auf beide Kompositionspartner.

Ein weiterer relevanter Punkt ist die Eigenschaft von Typen, ihre Instanzen zu *klassifizieren*. Auf diese Weise wird ein schnelles Auffinden spezifischer Einheiten anhand derer Typen möglich. In Übertragung auf Komponentenarchitekturen heißt das, daß nicht zwangsläufig Komponenten, sondern allgemeinere Typen als Gegenstand der Argumentationen dienen können. Eine Anwendungsmöglichkeit ist dabei etwa die weiter oben behandelte Spezifikation mit "unscharfen" bzw. externen Anforderungsbeschreibungen, welche in Form von Typen umgesetzt werden könnte.

Im Rahmen der konzeptionell besonders reichen Komponentenarchitekturen stellt sich nun einmal mehr auch in diesem Zusammenhang die Frage, ob Universallösungen zu den gewünschten Resultaten führen können. Konkret formuliert lautet diese: Verlangt das Komponentenparadigma nach *multiplen Typsystemen*? Ansatzpunkt für eine Differenzierung ist hier zunächst die Unterscheidung verschiedener Sichten: Zum einen stehen dabei komponenteninterne- komponentenexternen Aspekten gegenüber, zum anderen Kompositions- und Metamodelle. Des weiteren könnte in bestimmten Fällen eine Unterscheidung zwischen *Verhaltenstypen* und *Strukturtypen* relevant sein.

4.3.4.1 Typisierungsgegenstände

Unter Typisierung versteht man, wie erwähnt, im allgemeinen die Zusicherung einer invarianten Eigenschaft für ein bestimmtes Objekt, welches eben dieses Types ist. Es stellt sich nun die Frage, welche Eigenschaften man einem Typsystem zugrunde legt, nach welchen Eigenschaften man also die Objekte des zu typisierenden Systems einteilen möchte.

- *Struktureigenschaften* — Statische Strukturbestandteile von Komponenten wie Interfaces, Operationen, Nachrichten,
- *Verhaltenseigenschaften* — Dynamische Verhaltensaspekte von Komponenten. Z.B. zeitliche bzw. kausale Bedingungen über funktionalen Struktureinheiten.

Bezüglich der im Rahmen einer Komponentenarchitektur bedeutsamen Informationen sind beide Varianten relevant, denn bei einer Komposition von Bausteinen müssen diese sowohl in ihrer strukturellen als auch dynamisch/funktional zusammenpassen. Möchte man dann also beide Eigenschaften durch Typisierung berücksichtigen, bestehen zwei Möglichkeiten der Kombination:

- *Strukturelle und dynamische* Eigenschaften in *zwei getrennten Typsystemen.*
- *Strukturelle und dynamische* Eigenschaften, vereint in *einem einzelnen Typsystem.*

Welche Möglichkeit zum Tragen kommt, ist stark von den formalen Mitteln der Spezifikation abhängig. Kommen verschiedene separate Spezifikationen unterschiedlicher (semantischer) Aspekte zum Einsatz, welche dann noch auf verschiedenen Formalismen basieren, ist eine Integration nicht immer durchführbar.

Als zunächst letzter Aspekt sei noch auf die nicht grundsätzlich eindeutige Beziehung von Spezifikation und Typisierung hingewiesen. Man kann hier nämlich zwei komplementäre Varianten unterscheiden:

- *Typen als Gegenstände der Spezifikation* oder
- *Spezifikation als Grundlage der Typisierung.*

Es stellt sich hier die Frage, inwiefern diese Varianten auf die Anwendbarkeit von Typisierung im Komponentenbezug wirken. Im nachfolgenden Kapitel über konkrete Ansätze wird die Thematik noch einmal aufgegriffen.

4.3.4.2 Typrelationen

Typsysteme beinhalten verschiedene Relationen zwischen Typen, d.h. bestimmte Typen gehören homogenen Mengen einer gleichen spezifischen Eigenschaft an. In Bezug auf solche *Typrelationen* kann man dabei zwei, für Komponentenarchitekturen grundsätzlich wichtige Varianten unterscheiden:

Zunächst sei hier die klassische *Typkonformität* genannt, welche in Bezug auf verschiedene Aspekte wie Syntax, Syntax mit Semantik und purer Semantik ausgelegt werden kann. Komponenten müssen sich hierbei in der formalen Verhaltensspezifikation nicht exakt — also syntaktisch — entsprechen. Wesentlich ist vielmehr eine konforme Semantik.

Die strukturellen Aspekte sind weniger flexibel. Wenn hierbei keine exakte syntaktische Identität gefordert wird, erfolgt zwangsweise ein Kompromiß in Bezug auf die Treffsicherheit des wechselseitigen Matchings logisch gleichbedeutender Strukturen. Der Grund dafür ist, daß eine externe Sichtweise das "Wesen" der Strukturen nur implizit als Abstraktion erfaßt. Kompromittiert man aber diese Abstraktion, ist jede angewandte Methode im strengen Sinn nicht mehr als eine heuristische Richtlinie und, da nicht eindeutig, nur von begrenztem Nutzen.

Zur klassischen Typkonformität mit dem Hintergrund der Substitution von Typen kommt im Komponentenmodell eine Typrelation, welche man als *Typinteroperabilität* oder *Typkooperation* bezeichnen könnte. Letztere drückt die Fähigkeit des Zusammenwirkens von Komponenten verschiedenen Typs bezüglich Struktur und Semantik aus. Es kommt dabei nicht darauf an, gleiche, sondern passende Typen zu kennzeichnen. Strukturelle und darunter besonders funktionale Einheiten — etwa Nachrichten — müssen bei Typinteroperabilität zwischen den Kandidaten paarweise Gegenstücke besitzen.

Für den Fall zeitlicher bzw. kausaler Bedingungen muß auch bei simultaner Proklamation ein gemeinsam gangbarer Weg existieren. Etwas genauer heißt das, daß auch bei Einhaltung beider semantischen Regelwerke noch eine zulässige Möglichkeit kontinuierlicher Anwendung der Komponenten- bzw. Interface Funktionen bestehen muß. So eine Relation ist für das Typsystem eines Komponentenmodells von grundlegender Wichtigkeit. Sie erlaubt z.B. in bestehenden Frameworks ein Retrieval fehlender Komponenten. Eine weitere Anwendungsmöglichkeit ist das frühzeitige Typechecking gewünschter Kompositionen, was einen wichtigen Beitrag zur präventiven Fehlervermeidung leistet.

Kapitel 5

Entwurfsansätze für Komponentenarchitekturen

Das vorliegende Kapitel beschäftigt sich mit einigen speziellen Konzepten und Ideen für komponentenorientierte Architekturen. Obwohl die Konzeptionierung einer kompletten Architektur über den Rahmen dieser Arbeit hinausgeht, sollen sich die behandelten Aspekte dabei durch Beachtung wechselseitiger Bezüge zu einer möglichst harmonischen Gesamtsicht ergänzen.

Der erste Abschnitt enthält den groben Entwurf eines *Komponentenmodells*. Der Ansatz ist dabei sehr allgemein gehalten, da er vor allem als Ausgangspunkt für die in den nachfolgenden Abschnitten behandelten Beschreibungsmethoden dienen soll. Diese Beschreibungsmethoden teilen sich dann in drei Abschnitte. Zunächst wird die Beschreibung von *Verhaltensaspekten* durch zeitlich/kausale Bedingungen mittels *temporaler Logik* behandelt. Im Anschluß wird die *dynamische Komponentenspezifikation* mit *Feature-Logik* beleuchtet, bevor am Ende des Kapitels eine abrundende Betrachtung übergreifender Aspekte der *formalen Integration* den Abschluß bildet.

5.1 Eine exemplarische Realisierung der Modellebene

Wie schon in vorangegangenen Kapiteln geschehen, wird für die Darstellung des Beispielmodells eine Gliederung in Struktur und Verhalten vorgenommen. Die Betrachtung einer expliziten Metaebene entfällt in diesem Ansatz.

5.1.1 Strukturelle Modellbestandteile

Der erste zentrale Begriff des Komponentenmodells ist die Komponente selber. Die folgende Definition orientiert sich an der bereits in Kapitel 2.1.1 vorgeschlagenen Charakterisierung.

Begriff 1 (Komponente) *Eine* Komponente *(Abb. 5.1) ist die* ***Abstraktion eines autonomen, semantisch geschlossenen Softwaresystems*** *beliebiger Granularität. Sie ist innerhalb eines globalen Rahmens zur* Komposition *mit anderen Ausprägungen fähig, um zusammen das gemeinsame höhere Ziel einer zusammengesetzten Anwendung zu erreichen, welche aus Modellsicht ihrerseits wieder eine Komponente darstellt. Eine Komponente ist entweder* atomar, *d.h.*
im Rahmen des Modells nicht weiter teilbar, oder eine Komposition *weiterer Komponenten, die dann Subsysteme darstellen.*

Zentraler Bestandteil einer Komponente ist der Komponentenkern, *welcher deren konkrete Realisierung und dynamische Zustände kapselt. In ihm manifestieren sich ferner die anpaßbaren* Komponenteneigenschaften *sowie die internen Abhängigkeiten der verschiedenen* Komponentenschnittstellen.

Komponenten stellen ihre Funktionalität nach außen über Mengen eindeutiger, klar abgegrenzter Exportschnittstellen *bereit, welche über deren Spezifizierung den Rahmen von Vertragsverhältnissen als Subsystem potentieller Nutzerkomponenten charakterisieren. Die Anforderungen einer nicht atomaren Komponente an ihre potentiellen Subsystemen wird durch vorausgesetzte* Importschnittstellen *ausgedrückt.*

Der oben definierte Komponentenbegriff enthält als weitere Bestandteile *Komponentenkern* und *-schnittstellen.* Diese Konzepte werden nachfolgend exakter beschrieben.

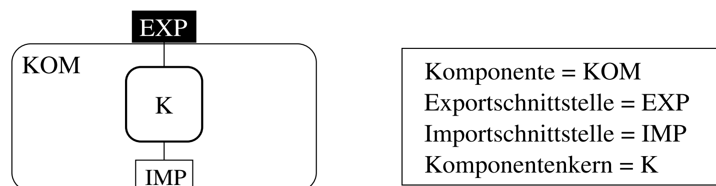


Abbildung 5.1: Einfache Komponente

Begriff 2 (Komponentenkern) *Ein* Komponentenkern *(Abb. 5.2) ist der zentrale Bestandteil einer Komponente und dient als gemeinsamer* ***interner Bezugspunkt****. Komponentenintern ist er Bindeglied zu den Aspekten*

von Implementation und Zustand. Komponentenextern ist er Träger der Abstraktionen von Komponenteneigenschaften sowie einer zentral übergreifenden Spezifikation.

Letztere liegt in Form formaler Beschreibungen der bereitgestellten und vorausgesetzten Funktionalität der Komponente vor, welche sowohl statische als auch dynamische bzw. semantische Anteile umfassen, wobei letztere in Form des temporalen Aspektes berücksichtigt werden. Die Komponentenspezifikation besteht im einzelnen aus folgenden Teilen:

- *Eine* ***Menge von Spezifikationen*** *der Komponentenschnittstellen, aufgeteilt in Teilmengen bezüglich der Import- und Exportteile,*
- *Ein* ***zentraler temporaler Aspekt*** *der wechselseitigen Abhängigkeiten aller durch die Schnittstellenmenge gegebenen importierten und exportierten Funktionalitäten sowie der internen Aspekte.*

Um eine Erweiterbarkeit der Komponente zu ermöglichen, läßt sich der Kern um Spezifikationsbestandteile ergänzen.

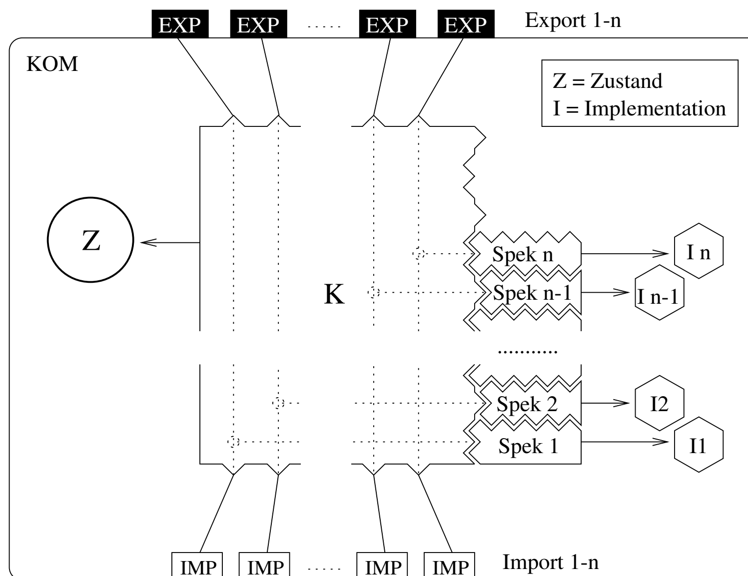


Abbildung 5.2: Der Komponentenkern

Begriff 3 (Komponentenschnittstelle) Schnittstellen *fungieren als explizite* ***externe Bezugspunkte*** *für Komponentenbeziehungen. Sie dienen dabei der eindeutigen Spezifikation von Funktionalität in sowohl statischem als auch dynamischem bzw. semantischem Sinne und stellen einen Vertrag*

zwischen nutzender und erbringender Komponente dar. Schnittstellen teilen sich in Importschnittstellen und Exportschnittstellen auf, welche die Bereiche vorausgesetzter und bereitgestellter Funktionalität abdecken. Aus beiden Bereichen können mehrere Schnittstellen pro Komponente vorliegen. Atomare Komponenten besitzen ausschließlich Exportschnittstellen. Verbundkomponenten zeichnen sich durch zusätzliche Importschnittstellen aus.

Eine ***Exportschnittstelle*** *besteht aus folgenden Teilen:*

- Statische Strukturinformationen *der exportierten Funktionalität, d.h.* ***Namen*** *von Schnittstelle, Nachrichten als verwendeten syntaktischen Einheiten und Attributen zur Modellierung von Zuständen.*
- Semantische Voraussetzungen *über die Anwendung der exportierten Funktionalität. Diese liegen in Form des* **temporalen Aspektes** *über den Nachrichten und Attributen einer Schnittstelle vor, welcher deren zeitliche Abhängigkeiten ausdrückt. Der temporale Aspekt einer Schnittstelle leitet sich aus dem zentralen Komponentenkern ab.*

Eine ***Importschnittstelle*** *besteht aus folgenden Teilen:*

- Statische Strukturinformationen *über importierte Funktionalität in Form von vorausgesetzten Komponenten Schnittstellen oder Nachrichten und Attributen.*
- Semantische Anforderungen *bezüglich der Verwendung der importierten Funktionalität. Diese liegen in Form des temporalen Aspektes über der letztendlichen Menge von Nachrichten bzw. Attributen vor. Der temporale Aspekt leitet sich wiederum aus dem zentralen Komponentenkern ab.*

Engt man die Betrachtung einer Komponentenstruktur auf deren Schnittstellen ein, so ergibt sich eine korrespondierende Schnittstellenstruktur mit wechselseitigen Anforderungen und Bedingungen bezüglich statischer und dynamischer Eigenschaften der Gegenstücke. Diese Struktur entspricht Vertragsverhältnissen zwischen den Kooperationspartnern. Abbildung 5.3 veranschaulicht diese Sicht für eine beispielhafte Konfiguration mit vier Import- und Exportschnittstellen.

Wie aus dem Schnittstellenbegriff hervorgeht, sollen als atomare Interaktionseinheiten im Rahmen dieses Beispielmodells *Nachrichten* dienen, welche nun definiert werden:

Begriff 4 (Nachricht) *Eine* Nachricht *ist die Abstraktion einer* ***atomaren Interaktionseinheit*** *zwischen Komponenten mit eindeutiger, expliziter Semantik. Diese Semantik wird innerhalb des Modells als gegeben betrachtet und nicht weiter aufgeschlüsselt. Nachrichten werden zwischen den*

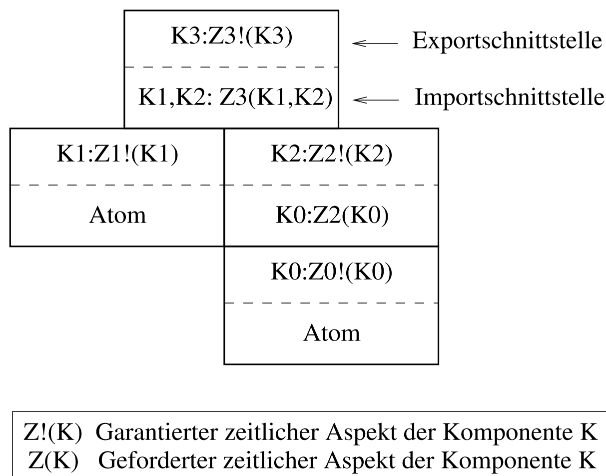


Abbildung 5.3: Beispiel einer Schnittstellenstruktur

Schnittstellen genau einer sendenden und einer empfangenen Komponente ausgetauscht. Nachrichten werden in diesem Sinne bezüglich der betrachteten Schnittstelle immer als entweder eingehend oder ausgehend gekennzeichnet. Die öffentliche Berücksichtigung ausgehender Nachrichten erlaubt dabei eine Modellierung asynchroner Kommunikation zwischen Komponenten.

Um die Möglichkeit zu schaffen, relevante Zustände modellieren zu können, werden als letzte strukturelle Modellbestandteile *Komponentenattribute* eingeführt:

Begriff 5 (Attribut) *Ein* Attribut *ist die Abstraktion einer* ***Komponenteneigenschaft*** *mit eindeutiger, expliziter Semantik, welche als gegeben betrachtet wird. Die Attributwerte entstammen immer einer endlichen diskreten Domäne. Das Lesen bzw. Setzen eines Attributes erfolgt durch spezielle Nachrichten. Da Nachrichten keine Parameter besitzen, sei für jedes Element der Attributdomäne eine korrespondierende Nachricht gegeben.*

Der Komponentenbegriff beinhaltet deren Intention zur zielgerichteten wechselseitigen Kooperation. Einzelne Zusammenschlüsse dieser Art werden als *Kompositionen* bezeichnet und wie folgt festgelegt:

Begriff 6 (Komponentenkomposition) *Eine* Komponentenkomposition *ist die* ***Kombination von Komponenten zu einem Konglomerat****, welches selbst wieder eine Komponente ist (Abb. 5.4). Die einzelnen Komponenten werden dabei als Teilkomponenten bezeichnet, deren Konglomerat als Verbund- bzw. Toplevelkomponente.*

Bei der Komposition wird die Teilfunktionalitäten einzelner Komponenten aggregiert und in einen gemeinsamen neuen Kontext angehoben. Ferner können diese Teilfunktionalitäten auch synergetisch zu "Mehrwertfunktionalitäten" kombiniert werden.

Als Bedingung *für die Komposition müssen die* vertraglichen Bedingungen *aller Kooperationspartner, welche in deren* Komponentenschnittstellen *bzw.* Komponentenkernen *spezifiziert sind, eingehalten werden. Dies kann dabei auch indirekt unter Zuhilfenahme von Interoperabilitätmechanismen zur Integration heterogener Komponenten erfüllt werden.*

Als Beschränkung *sei festgelegt, daß Kompositionen nicht zu unmittel- oder mittelbaren* Zyklen *innerhalb einer Komponentenstruktur führen dürfen. Ferner sei* multiple Komposition — *also die Kombination einer Komponente als Bestandteile mehrerer verschiedener Toplevelkomponenten — ausgeschlossen.*

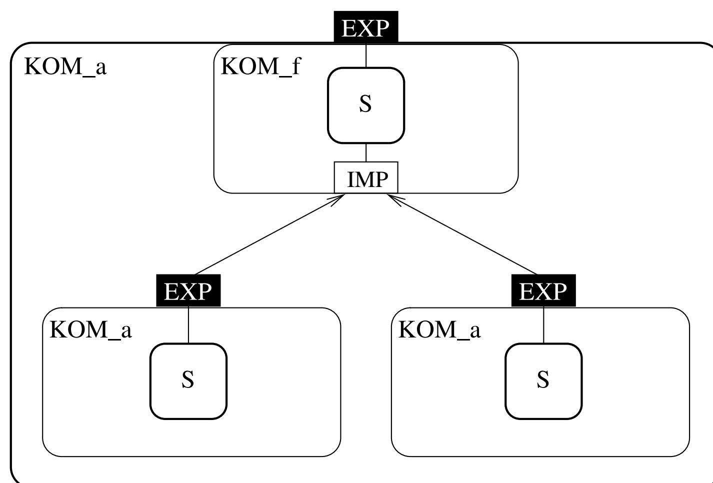


Abbildung 5.4: Komponentenkomposition aus Modellsicht

5.1.2 Modellverhalten

Das *Modellverhalten* besteht in Aktivitäten der Interaktion zwischen Komponenten. Da langlebige Systeme im Mittelpunkt des Interesses stehen, sollen diese Aktivitäten *nicht terminieren*, sondern kontinuierlich und somit potentiell unendlich voranschreiten.

Die grundlegende Aktivität innerhalb des vorliegenden Komponentenmodells besteht im wechselseitigen *Nachrichtenaustausch*. Ein solcher Nach-

richtenaustausch findet zwischen einer sendenden und einer empfangenen Komponente statt. Der Zeitpunkt des Austausches ergibt sich aus der Bereitschaft beider Kommunikationspartner, d.h. es existieren *keine Warteschlangen* zur Pufferung von Nachrichten. Die Menge möglicher Nachrichten ist auf eine endliche Anzahl *beschränkt*, da das Modell hauptsächlich der Untersuchung organisatorischer Aspekte von Komponentensystemen dienen soll.

Die Bausteine des Systems sollen grundsätzlich unabhängig agieren, d.h. das System stellt eine *nebenläufige* Umgebung dar. Um unter diesen Umständen eine gleichmäßige, kontinuierliche Funktion des Gesamtsystems zu gewährleisten, sei ein *faires Verhalten* von Bausteinen unterstellt, d.h. aus einer Menge potentieller Möglichkeiten des Narichtenaustausches wird die jeweils realisierte fair ausgewählt. In theoretischem Sinne wird ein bestimmter Nachrichtenaustausch genau dann garantiert, wenn eine Möglichkeit zu dessen Realisierung unendlichen oft wiederkehrt.

Aus den atomaren Aktivitäten des Nachrichtenaustausches setzen sich wiederum komplexere *Interaktionen* zwischen Komponenten zusammen. Interaktionen finden immer zwischen genau zwei Modellkomponenten statt, wobei eine der beiden immer die Toplevelkomponente einer Komposition ist. Diese Toplevelkomponente ist stets *Initiator* der Aktivität. Zwei Teilkomponenten einer Struktur können in diesem Sinne immer nur indirekt über die Toplevelkomponente interagieren, welche in diesem Fall die Funktion eines *Mediators* übernimmt. Abbildung 5.5 visualisiert diese Zusammenhänge.

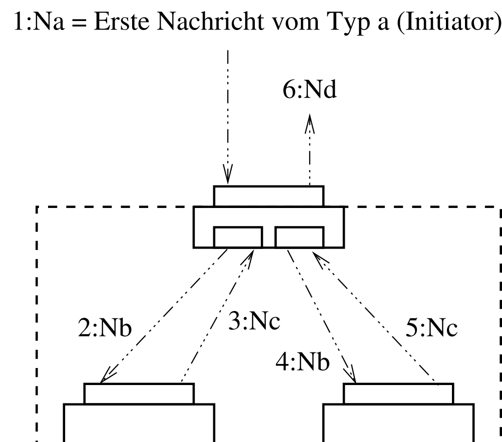


Abbildung 5.5: Komponenteninteraktion als Nachrichtensequenz

Die erlaubten globalen Nachrichtensequenzen ergeben sich aus den, bezüglich der beteiligten Komponenten, lokalen Gegebenheiten. Diese Einzelkomponenten garantieren auf der einen Seite ihr eigenes Verhalten in Form von Sequenzen über einer Menge eigner Nachrichten und fordern auf

der anderen Seite ein ebenso bestimmtes Verhalten ihrer Partner. Um solche lokalen Bedingungen explizit zu machen, dienen formale Beschreibungen der temporalen Aspekte, welche über Schnittstellen offengelegt werden. Abbildung 5.6 veranschaulicht das sich ergebende Netzwerk wechselseitiger Beziehungen.

Abbildung 5.6: Globale Interaktion als Wechselwirkung lokaler Verhaltensaspekte

5.2 Verhaltensbeschreibung mit Temporallogik

Komponenten sollen ihr garantiertes und gefordertes Verhalten in Form des temporalen Aspektes innerhalb ihrer Schnittstellen offenlegen, um den korrekten Ablauf potentieller, wechselseitiger Interaktionsbeziehungen garantieren zu können. Besagte temporale Aspekte sollen in dem hier vorgeschlagenen Ansatz durch eine spezielle *Logik* ausgedrückt werden. Diese *temporale Logik* und deren Anwendung machen den Inhalt der folgenden Untersektion aus.

5.2.1 Temporallogik

Die temporale Logik (siehe z.B. [Gab92, GHR92]) ist, wie erwähnt, eine Erweiterung der modalen Logik um qualitative zeitliche Aspekte. Sie beschreibt somit die sich wandelnde Wahrheit logischer Ausdrücke im Laufe der Zeit.

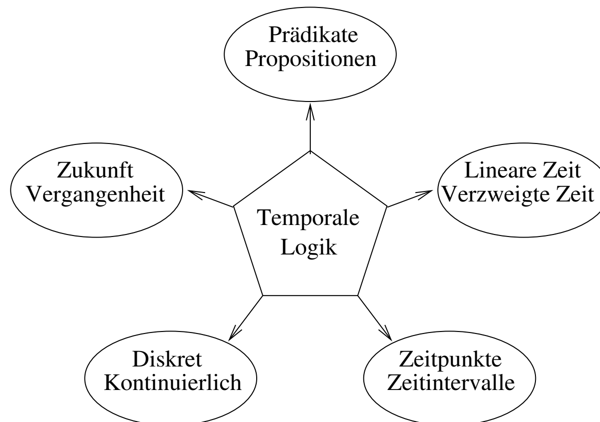


Abbildung 5.7: Formen temporaler Logik

Temporale Logiken stellen ein weites Feld mit vielen unterschiedlichen Ausprägungen dar (Abb. 5.7). Man unterscheidet hier zunächst nach Art der zugrundeliegenden klassischen Logik zwischen *propositionaler-* bzw. *Aussagenlogik* sowie *Prädikatenlogik*. Als nächstes existieren bezüglich der Modellierung von Zeit *diskrete* und *kontinuierliche* Varianten. Bezüglich des so modellierten Zeitbegriffs, findet eine weitere Differenzierung nach dessen Eigenschaften statt. Hier existieren zum einen Varianten mit *linearem Zeitverlauf* und zum anderen solche mit alternativen, *verzweigenden Zeitlinien*. Beide Varianten starten entweder an einem festgelegten Anfang — der Gegenwart — oder beziehen die Vergangenheit mit ein. Bezüglich des Gegenstandes der logischen Betrachtungen kommen alternativ *Zeitpunkte* oder *Zeitintervalle* in Betracht.

Aus den vielfältigen Ausprägungen temporaler Logik muß nun die Menge solcher Varianten herausgefiltert werden, welche zur Beschreibung des temporalen Aspektes von Komponenten geeignet sind. Aus diesen ist dann der vielversprechendste Kandidat zu wählen, welcher im Anschluß als Ausgangspunkt weiterer Betrachtungen dienen soll. Im folgenden wird bezüglich der relevant erscheinenden Kriterien eine Abgrenzung vorgenommen.

Zunächst muß eine geeignete *Grundform der Logik* bestimmt werden. Formal betrachtet ergeben sich dabei zwei wesentliche Fakten. Zum einen ist Prädikatenlogik in Bezug auf die Aussagekraft mächtiger als propositionale Logik, zum anderen ist aber Prädikatenlogik im Gegensatz zu propositionaler Lo-

gik nicht generell entscheidbar [Ara91]. Während für die propositionale Variante effiziente Algorithmen zum Test von Erfüllbarkeit temporallogischer Formeln bestehen, die eine automatisierte Verifikation von Spezifikationen möglich machen, ist dies für eine entsprechende Prädikatenlogik nicht der Fall. Legt man auf praktische Anwendbarkeit Wert, kann somit von letzterer abgesehen werden. Darüber hinaus hat sich propositionale Logik in vielen Studien als geeignet erwiesen[1].

Als nächstes muß eine sinnvolle Form des *Zeitbegriffs* abgegrenzt werden. Temporale Logiken existieren für verschiedene *Zeitlinien*. Diese können ihren Anfang in der unendlichen *Vergangenheit* oder einem konkreten Punkt — namentlich der Gegenwart — haben und von dort in die *Zukunft* verlaufen. Ferner kann man zwischen *eindeutigen* und *verzweigten* Zeitlinien unterscheiden. Es stellt sich die Frage, welches Zeitmodell zur Abbildung des Komponentenparadigma am sinnvollsten ist.

Betrachtet man alleine die *initiale Komposition* eines Komponentenframeworks bzw. einer komponentenorientierten Anwendung, so sind eindeutige Zeitlinien, welche vom Punkt der Gegenwart in die Zukunft verlaufen, geeignete Ausdrucksmittel, denn die entsprechenden Systeme sind vorher noch nicht existent. Ihr Dasein beginnt im Kompositionszeitpunkt und verläuft von dort an eindeutig in die Zukunft. Ein Vergangenheitsbezug auf die Zeit vor der Initialisierung ist somit nicht relevant.

Betrachtet man hingegen die *Rekonfiguration* oder *Evolution* bestehender Systeme mit langlebigen, wiederverwendbaren Komponenten, kann die Antwort nicht so pauschal ausfallen. Hier wird die *Historie* relevant. Zur Behandlung dieses Szenarios sind zwei Ansätze denkbar.

Zunächst wäre es in dieser Situation denkbar, den Startpunkt einer Zeitlinie — also die Gegenwart des temporalen Systems — in die reale Vergangenheit zu legen. Dies ist möglich, da die relevante Historie in jedem Fall nur endlich weit in die Vergangenheit reicht, nämlich maximal bis zum Kompositionszeitpunkt des betrachteten Systems. Man kann somit die gesamte relevante Zeitspanne allein mit Hilfe der eingeschränkten, zukunftsorientierten Sicht abdecken. Nachteil so einer Methode ist die Verfälschung der realen Zeitverhältnisse. Will man dies vermeiden, d.h. soll die Modellgegenwart mit der formalen Gegenwart übereinstimmen, muß auf eine Zeitlinie mit Vergangenheit und Zukunft zurückgegriffen werden, die zu komplexeren Methoden führt. Aus rein formaler Sicht ist dies jedoch nicht notwendig, denn die theoretische Ausdruckskraft erhöht sich dadurch nicht [Gab92]. Letztere Tatsache soll dann auch als Hauptargument für die Wahl rein *zukunftsorientierter Zeitlinien* im vorliegenden Modell dienen.

Die wesentlichen Ereignisse des Komponentenmodells bestehen aus dem Sen-

[1] Die Variante wurde insbesondere mehrfach erfolgreich im Zusammenhang mit Prozeßbeschreibungen angewandt (z.B in [MW84])

den und Empfangen von Nachrichten und sind somit aus idealisierter Sicht zeitverzugslos. Entsprechend ist die Wahl von *diskreten Zeitpunkten* als Betrachtungsgegenstände der Logik sinnvoll.

Die Verwendung verzweigender statt linearer Zeitlinien kann hier nicht generell ausgeschlossen werden, soll aber im Rahmen dieser Arbeit nicht untersucht werden. Als Fazit kommt man also zu einem Zeitbegriff mit einer unendlichen Anzahl diskreter, zukunftsgerichteter Ereigniszeitpunkte. Die entsprechende Zeitlinie ist in Abbildung 5.8 dargestellt.

Abbildung 5.8: Lineare Zeitlinie mit diskreten Zeitpunkten

Aus klassischer Grundlogik und Zeitcharakteristik ergibt sich der im weiteren verwendete Formalismus, nämlich *propositionale lineare temporale Logik* (*PLTL*). Diese Logik soll nun in der folgenden Sektion formal eingeführt werden.

5.2.1.1 Propositionale Lineare Temporale Logik

Temporale propositionale Logik basiert auf der Aussagenlogik, also auf logischen Formeln über atomaren Aussagen bzw. Propositionen. Dieser Grundansatz wird nun zunächst durch eine zeitabhängige Zuweisung von Wahrheitswerten erweitert. Es werden dabei unendlich viele, diskrete Zeitpunkte auf einer linearen zukunftsorientierten Zeitlinie mit festem Startpunkt betrachtet. Für jeden dieser Zeitpunkte findet eine eigene Interpretation im Sinne klassischer Aussagenlogik statt.

Als zweite wesentliche Erweiterung werden die bekannten Operatoren der Aussagenlogik um zusätzliche temporale Operatoren ergänzt. Diese sind in der folgenden Tabelle 5.1 aufgelistet, wobei p bzw. q atomare Propositionen sein mögen:

Operator	Semantik
$\Box\, p$	"von nun an" p
$\Diamond\, p$	"manchmal" p
$\Delta\, p$	"nächstes mal" p
$p\mathcal{U}q$	p "bis" q

Tabelle 5.1: Temporale Operatoren

Die temporalen Operatoren hängen von dem gewünschten logischen System ab und können dementsprechend variieren. Bei der hier getroffenen Auswahl handelt es sich zunächst um eine praktische Grundmenge, die später noch eingehender untersucht wird.

Der von nun an Operator $\Box\ p$ meint, daß p in diesem und allen folgenden Zeitpunkten den Wert "wahr" besitzt. Der *manchmal* Operator $\Diamond\ p$ sagt aus, daß p möglicherweise in diesem und einigen späteren Zeitpunkten, mit Sicherheit aber irgendwann, den Wert "wahr" annehmen wird. Bei dem *nächstes mal* Operator $\triangle\ p$, ist p genau im folgenden Zeitpunkt "wahr". Der letzte *bis* Operator $p\mathcal{U}q$ meint, daß entweder p ab jetzt für alle Zeit "wahr" ist, oder genau solange, bis q den Wert "wahr" annimmt.

Die Bedeutung der Operatoren kann man sich am einfachsten mit Hilfe einer Zeitlinie veranschaulichen. Zeitpunkte der Linie sind genau dann mit bestimmten Propositionen beschriftet, wenn diese in dem Punkt gelten — also "wahr" sind. Im unteren Bereich wird nun mit dem jeweiligen Operator ein zusammengesetzter Ausdruck über atomaren Propositionen gebildet. Im oberen Bereich wird dann eine mögliche korrespondierende Folge "wahrer" Zeitpunkte dieser atomaren Propositionen angegeben. Abbildung 5.9 veranschaulicht die Operatoren auf diese Weise.

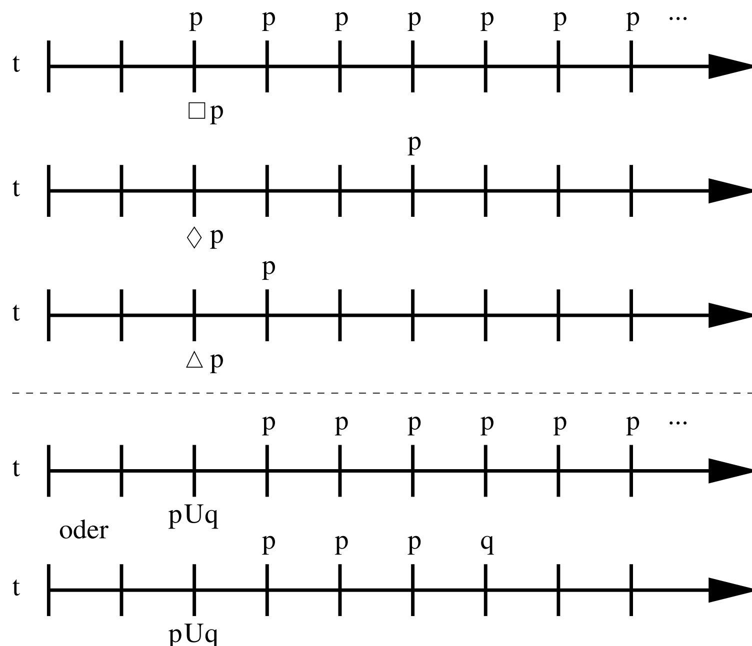


Abbildung 5.9: Bedeutung von PLTL Operatoren über der Zeitlinie

Im folgenden werden die soeben informal motivierten Begriffe propositionaler temporaler Logik formal eingeführt.

Lineare Zeitlinien Bei der hier betrachteten linearen temporalen Logik ist die Struktur der Zeit eine total geordnete Menge $(S, <)$. Die formale Definition einer solchen Zeitstruktur lautet wie folgt:

Definition 1 (Lineare Zeitstrukturen) *Sei AP eine Menge grundlegender Propositionen. Eine Zeitstruktur ist definiert als*

$$\begin{array}{rl} M = (S, x, L) & \textit{mit} \\ S & \textit{als Menge von Zuständen,} \\ x : \mathcal{N} \rightarrow S & \textit{als unendliche Zustandssequenz und} \\ L : S \rightarrow 2^{AP} & \textit{als Mengen wahrer Propositionen pro Zustand.} \end{array}$$

PLTL Syntax Die PLTL Syntax besteht aus einem Grundgerüst klassischer Aussagenlogik, ergänzt um die oben eingeführten temporalen Operatoren. Eine entsprechende wohlgeformte PLTL-Formel muß der folgenden Definition genügen:

Definition 2 (PLTL Formeln) *Die Menge gültiger Formeln in propositionaler linearer temporaler Logik (PLTL) ist die kleinste Menge von Formeln, welche durch die folgenden Regeln generiert werden:*

(1) Jede atomare Proposition P ist eine Formel.

(2) Sind p und q Formeln, dann sind $(p \wedge q)$ und $\neg p$ Formeln.

(3) Sind p und q Formeln, dann sind $(p \mathcal{U} q)$ und $\triangle\, p$ Formeln.

Als im mathematischen Sinne gebräuchliche Abkürzungen werden des weiteren die folgenden Operatoren eingeführt:

$$\begin{array}{rcl} p \vee q & = & \neg(\neg p \wedge \neg q) \\ p \Rightarrow q & = & \neg p \vee q \\ p \equiv q & = & (p \Rightarrow q) \wedge (q \Rightarrow p) \\ true & = & p \vee \neg p \\ false & = & \neg true \\ \Diamond\, p & = & (true \mathcal{U} p) \\ \Box\, p & = & \neg \Diamond \neg p \\ \Diamond^{\infty} p & = & \Box \Diamond\, p \\ \Box^{\infty} p & = & \Diamond \Box\, p \\ (p\, \mathcal{B} q) & = & \neg((\neg p)\, \mathcal{U} q) \end{array}$$

PLTL Semantik Die Semantik von PLTL Formeln wird in Bezug auf lineare Zeitlinien $M = (S, x, L)$ definiert. Der Ausdruck $M, x \models p$ meint "In Struktur M ist die Formel p in Zeitlinie x wahr". $\models$ wird nun induktiv über die Formelstruktur definiert:

(1)	$x \models P$	iff $P \in L(s_0)$ für atomare Propositionen P
(2)	$x \models p \wedge q$	iff $x \models p$ und $x \models q$
	$x \models \neg p$	iff Es gilt nicht das $x \models p$
(3)	$x \models (p \mathcal{U} q)$	iff $\exists j (x^j \models q$ und $\forall k < j (x^k \models p))$
	$x \models \triangle\, p$	iff $x^1 \models p$

Für die anderen — abgeleiteten — temporalen Operatoren ist eine Definition ihrer Semantik nicht notwendig, kann aber zur besseren Anschauung ebenfalls durchgeführt werden. Für sie gilt bei autonomer Definition entsprechend:

$x \models \Diamond\, p$	iff $\exists j (x^j \models p)$
$x \models \Box\, p$	iff $\forall j (x^j \models p)$
$x \models (p \mathcal{B} q)$	iff $\forall j (x^j \models q$ impliziert $\exists k < j (x^k \models p))$

Grundbegriffe des logischen Systems Um nun das formal eingeführte System anwenden zu können, fehlen noch einige grundsätzliche Definitionen. Im folgenden wird nach Einführung des *Modellbegriffs* beschrieben, wann PLTL-Formeln *erfüllbar* bzw. *gültig* sein sollen.

Definition 3 (Modell) *Sei p eine PLTL Formel. Jede Zeitstruktur $M = (S, x, L)$ mit $M, x \models p$ definiert ein **Modell** von p.*

Definition 4 (Erfüllbarkeit) *Eine PLTL Formel p heißt **erfüllbar**, falls eine lineare Zeitstruktur $M = (S, x, L)$ existiert, so daß $M, x \models p$*

Definition 5 (Gültigkeit) *Eine PLTL Formel p heißt **gültig**, falls jede Zeitstruktur $M = (S, x, L)$ ein Modell von p definiert. Man schreibt dann $\models p$. Als Implikation ergibt sich, daß p gültig ist, wenn $\neg p$ nicht erfüllbar ist.*

Verifikation Um PLTL bei der Komponentenspezifikation einsetzen zu können, muß es eine Möglichkeit automatisierter Verifikation entsprechender Beschreibungen geben. Zu diesem Zweck kann die Erfüllbarkeit von PLTL-Formeln durch effiziente Tableau-basierte Algorithmen getestet werden [MW84, Ara92]. Die Algorithmen erhalten eine Formel F als Eingabe und berechnen einen Graphen (*Erfüllbarkeitgraph*), welcher alle Modelle repräsentiert, die F erfüllen. Falls F nicht erfüllbar ist, signalisiert der Algorithmus dies durch einen leere Ausgabe.

5.2.2 PLTL basierte Formalisierungskonzepte für Komponentenmodelle

Für die Beschreibung zeitlicher Aspekte von Komponenten mit temporaler Logik gibt es verschiedene Möglichkeiten. Zunächst wird eine Methode betrachtet, die auf zentralen logischen Ausdrücken beruht. Als mögliche Erweiterung wird dann im Anschluß eine untergliederte Spezifikationsform als verfeinerte Variante vorgeschlagen. Im letzten Teil wird die Rolle von Propositionen untersucht und deren Tauglichkeit kritisch hinterfragt.

5.2.2.1 Grundform der Spezifikation

Der erste mögliche Ansatz besteht in einer Beschreibung des Komponentenverhaltens durch zentrale, temporallogische Ausdrücke mit schnittstellenübergreifender Bedeutung. Die Methode lehnt sich an den entsprechenden Ansatz von Arapis an; in [Ara95] ist eine detaillierte Beschreibung zu finden.

Strukturelle Bestandteile Als strukturelle Bestandteile der Spezifikation werden ausschließlich Nachrichten betrachtet. Schnittstellen hingegen werden durch Nachrichtenmengen abgebildet. Man unterscheidet dabei eine Export- sowie mehrere Importschnittstellen und entsprechend eine Menge *öffentlicher* und mehrere Mengen *subkomponentenspezifischer* Nachrichten. *Attribute* werden bei diesem Ansatz durch deren korrespondierende Nachrichten berücksichtigt.

Temporale Aussagen Jede Nachricht einer Komponente wird mit einer entsprechenden atomaren Proposition in PLTL assoziiert. Die Tatsache, daß einer Proposition p in der zu einem Zeitpunkt (t) gehörenden Interpretation der Wert “wahr” zugeordnet wird, bedeutet, daß die mit p assoziierte eingehende (ausgehende) Nachricht in diesem Zeitpunkt von der betrachteten Komponente empfangen (gesendet) wird. Man setzt dabei voraus, daß immer genau eine Nachricht zur Zeit pro Komponente gesendet oder empfangen wird[2]. Mit den Propositionen kann die zeitliche Abhängigkeit zwischen Nachrichten durch PLTL-Formeln ausgedrückt werden. Diese Formeln stellen grundlegende temporale Bedingungen einer Komponente dar. Sie werden in zwei Mengen eingeteilt, wobei die erste Menge öffentlicher Bedingungen dem Exportanteil entspricht und nur die entsprechenden öffentlichen Nachrichten berücksichtigt. Die zweite Menge teilkomponentenspezifischer Bedingungen kann alle Nachrichten verwenden und entspricht einer zentralen Importspezifikation.

[2] D.h. in jeder Interpretation wird genau einer Proposition der Wert “wahr” und allen anderen der Wert “falsch” zugeordnet

Beispiel Zur Veranschaulichung des Verfahrens soll das schon im vorherigen Kapitel verwendeten Beispiels einer Maschine mit Produktions- und Wartungszyklus beschrieben werden. Die entsprechende atomare Komponente besitzt ein Attribut, mit dem ihr Zustand modelliert wird, der entweder "Wartung" oder "Produktion" sein kann. Sie kann zudem zwei Nachrichten empfangen, welche den Start oder Stop der Produktionsphase einleiten. Eine exemplarische Notation dieser Beschreibung ist in Abbildung 5.10 zu sehen.

Komponente: $Maschine$;
Attribute:
$Status\colon \{Produktion, Wartung\}$;
Öffentliche Nachrichten:
$\downarrow Start$, $\downarrow Stop$;
Öffentliche Bedingungen:
$Status := (Produktion \vee Wartung)$;
$\Box\,(Start \Rightarrow ((Status == Produktion) \wedge \vartriangle Stop))$;
$\Box\,(Stop \Rightarrow (\blacktriangle Start \wedge \vartriangle((Status := Wartung) \vee Start)))$;
$\Box\,((Status == Wartung) \Rightarrow (\vartriangle(Status := Produktion)))$;

Abbildung 5.10: Beispielhafte Komponentenspezifikation "Maschine"

Die Spezifikation enthält ebenfalls eine Menge temporale Bedingungen der Komponente. Attributspezifische Zuweisungsnachrichten werden in den Formeln durch die übliche Notation mit := abgekürzt. Die Abfrage eines Attributs entspricht einer Teilformel, welche die frühere Zuweisung des entsprechenden Wertes garantiert und wird durch == abgekürzt. Die erste Bedingung besagt in diesem Sinne, daß der Zustand der Maschine am Anfang entweder "Produktion" oder "Wartung" ist. Die zweite Bedingung sagt aus, daß in allen zukünftigen Zeitpunkten eine "Start" Nachricht stets impliziert, daß der Zustand gleichzeitig "Produktion" ist und im nächsten Ereigniszeitpunkt eine "Stop" Nachricht empfangen wird — also weder ein Zustandswechsel noch eine weitere "Start" Nachricht dazwischenliegen. Die dritte Bedingung bezieht sich auf das Produktionsende. Sie enthält einen bisher nicht betrachteten temporalen Operator $\blacktriangle p$, der sich auf die Vergangenheit bezieht und besagt, daß p genau im vorherigen Zeitpunkt den Wert "wahr" hatte. Bedingung drei meint dann also, daß in allen zukünftigen Zeitpunkten eine "Stop" Nachricht stets impliziert, daß die unmittelbar vorhergehende Nachricht "Start" war und daß im direkt folgenden Zeitpunkt die Maschine entweder durch entsprechende Zuweisung den Status "Wartung" annimmt oder durch eine Nachricht "Start" die Produktion ohne Zwischenstop wieder aufnimmt. Um die Wartungsphase zeitlich zu begrenzen, drückt die letzte Bedingung aus, daß in allen zukünftigen Zeitpunkten der Status "Wartung" impliziert, daß im unmittelbar folgenden Ereigniszeitpunkt wieder der Zustand "Produktion" zugewiesen wird und der Betrieb somit fortgesetzt wer-

den kann. Zu dieser Spezifikation gehört eine entsprechende Menge konformer Nachrichtensequenzen, welche den temporalen Bedingungen genügt. Eine davon ist mit zugehöriger Zeitlinie in Abb.5.11 dargestellt.

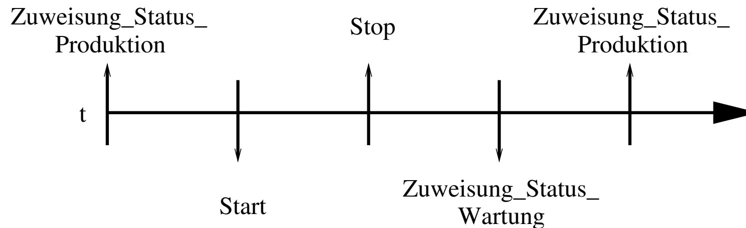


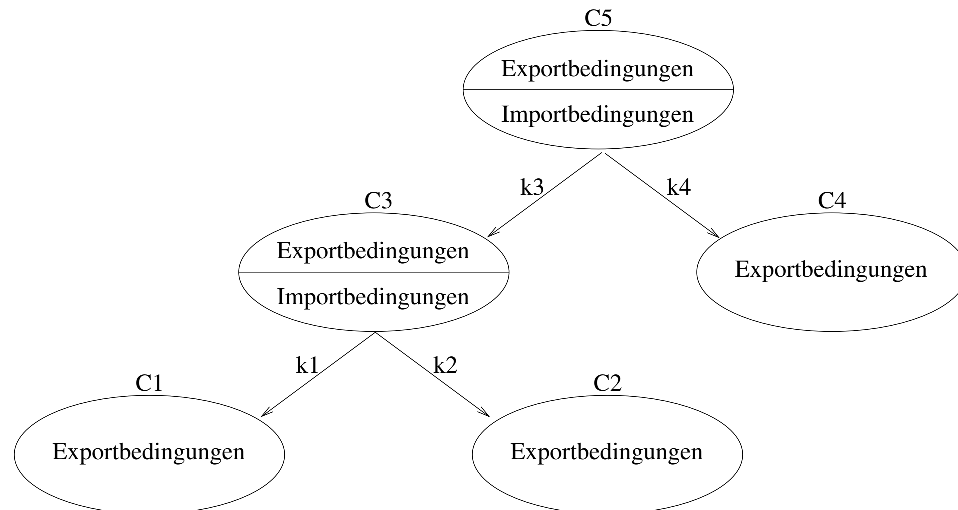
Abbildung 5.11: Mögliche korrekte Nachrichtensequenz der Maschine

Lokale und globale Zeit Ein wichtiger Aspekt temporaler Verhaltensbeschreibungen ist die Unterscheidung von *globaler* und *lokaler Zeit*. Eine PLTL-Formel bezieht sich in ihrer Grundform nur auf die komponenteninterne, lokale Zeitrechnung, d.h Abfolge von Ereigniszeitpunkten. Bei der Beschreibung von Vorgängen zwischen Komponenten muß hingegen eine globale Zeitrechnung zugrundegelegt werden, die für alle beteiligten Komponenten gleichermaßen gilt. Auf Grund der einfacheren Spezifikation und größeren Übersichtlichkeit ist es jedoch wünschenswert, einzelne temporale Aussagen trotzdem lokal auszudrücken und erst später in Bezug auf das Gesamtsystem zu bringen. Die entsprechende Transformation der lokalen in eine globale Zeit unter Beibehaltung temporaler Formelaussagen wird als *Universalisierung* (eng.*universalization*) bezeichnet.

Verbundkomponenten Ein *Koponentenverbund* wird über die Toplevelkomponente spezifiziert. Die teilkomponentenspezifischen Nachrichten und Bedingungen definieren dabei die extern geforderten strukturellen Einheiten und deren zeitliches Verhalten. Jede dieser Nachrichten ist dabei mit einem Bezeichner der zugehörigen Teilkomponente versehen und hat dort ein entsprechendes Gegenstück. Die Verwendung öffentlicher Nachrichten der Toplevelkomponente erlaubt die Berücksichtigung derer Exportbedingungen zur übergreifenden Synchronisation. Da bei jedem Nachrichtenaustausch die Toplevelkomponente entweder als Sender oder Empfänger auftritt, wird deren Modellsicht als Mediator hier sehr gut abgebildet.

Validierung Der wichtigste Grund einer Komponentenspezifikation ist die sich daraus ergebende Möglichkeit einer Validierung korrekter Kompositionen. Das beschriebene Modell erlaubt in diesem Sinne eine zweistufige Vorgehensweise.

Zunächst muß die Konsistenz der Spezifikationen einzelner, atomarer Komponenten geprüft werden. Dies geschieht durch Konjunktion der öffentlichen temporalen (Export-) Bedingungen, welche dann auf Erfüllbarkeit — also Existenz eines Modells — geprüft werden. Nach diesem Konsistenztest ist sichergestellt, daß die Komponenten für sich genommen ohne interne Konflikte innerhalb von Kompositionen angewandt werden können.



$$
\begin{aligned}
Komposition\,C3 &= Importbedingungen - C3 \; \wedge \\
& Exportbedingungen - C1 \; \wedge \\
& Exportbedingungen - C2 \\
Komposition\,C5 &= Importbedingungen - C5 \; \wedge \\
& Exportbedingungen - C3 \; \wedge \\
& Exportbedingungen - C4
\end{aligned}
$$

Abbildung 5.12: Verifikation einer Komponentenstruktur

Als zweiter — und wesentlicher — Schritt wird die Konsistenz von Komponentenstrukturen überprüft. Dies geschieht durch sukzessive Validierung der einzelnen beteiligten Verbunde. Abbildung 5.12 zeigt eine solche Struktur, die aus zwei Verbunden besteht.

Zum Test eines Verbundes ist es dabei immer nur notwendig, die unmittelbar importierten Teilkomponenten zu berücksichtigen. Von deren innerer Struktur — also von einem weiteren potentiellen Verbund — kann abstrahiert werden. In diesem Sinne erfolgt zum Konsistenztest eine Konjunktion von Importbedingungen der Toplevelkomponente und allen Exportbedin-

gungen der verwendeten Teilkomponenten. Die entsprechende, vollständige Spezifikation der Beispielstruktur ist in Abb. 5.12 zu sehen.

Als zusätzliche Bedingung ist die interne Kompatibilität der Import- und Exportbedingungen von Toplevelkomponenten sicherzustellen, d.h. jede Nachrichtensequenz, welche die Importbedingungen erfüllt, muß bei Beschränkung auf die öffentlichen Nachrichten auch den Exportbedingungen genügen. Dies drückt sich in der Formel

$$Importbedingungen \Rightarrow universalisierteExportbedingungen$$

aus. Die Universalisierung der Exportbedingungen ist notwendig, da zwischen den öffentlichen prinzipiell beliebige verbundinterne Nachrichten liegen können. Universalisierung ist in diesem Sinne ein notwendiger Wechsel von lokaler auf globale Zeit.

Fazit Das vorgestellte Konzept von Arapis [Ara95] ist zunächst ein tragbarer Grundansatz für die Spezifikation des Komponentenmodells. Es ist komplett, durchgehend schlüssig und mathematisch bewiesen [Ara91, Ara92]. Zu bedenken ist allerdings, daß die Grundgedanken der ursprünglichen Arbeit erstens auf einer objektorientierten Sicht beruhen und zweitens auf Entwurfsmethodologie abzielen. Es ist daher angebracht, den Ansatz in Hinblick auf das in der vorliegenden Arbeit zugrundeliegende Szenario komponentenorientierter Systemarchitekturen neu zu überdenken und entsprechend zu optimieren.

5.2.2.2 Verfeinerung der Spezifikationsstruktur

Ein Defizit der vorherigen Methode ist deren stark zentralisierte und komprimierte Form. Einzelne Aspekte der Spezifikation werden dabei nicht separiert betrachtet. Dies kann zum einen dazu führen, daß verschiedene Aspekte in einzelnen Aussagen vermascht werden, zum anderen aber auch dazu, daß sich einzelne Aspekte zergliedern und über eine Vielzahl von Einzelaussagen verstreuen. Als Folgen dieser Tatsache werden die im Komponentenmodell hervorgehobenen multiplen Exportschnittstellen nicht getrennt betrachtet, und generell ist sowohl die Übersichtlichkeit vorhandener als auch die einfache Erstellung neuer Komponentenbeschreibungen fraglich. Im folgenden werden unter Rückgriff auf das dem vorherigen Ansatz zugrundeliegende Modell einige Ansätze vorgeschlagen, die einer feineren, übersichtlicheren Gliederung der Spezifikation zuträglich sein könnten.

Adaption von PLTL-Prozeßsynchronisation Die in der vorherigen Sektion betrachtete Möglichkeit formaler Spezifikation begründet sich auf einem Modell zur Synchronisation nebenläufiger Prozesse nach Manna und

Wolper [MW84]. Ausgehend von diesem Grundansatz sollen nun einige Anmerkungen zur PLTL-gestützten Spezifikation des Komponentenmodells folgen.

In dem ursprünglichen Modell von Manna/Wolper werden Mengen kommunizierender, sequentieller *Prozesse* betrachtet. Diese Prozesse, welche in Form von *CSP* (*Communicating Sequential Processes*) [Hoa78] Programmen vorliegen, untergliedern sich jeweils in einen *funktionalen* und einen *synchronisierenden* Teil. Letzterer besteht dabei aus wechselseitigen, atomaren Operationsaufrufen und wird durch PLTL-Formeln beschrieben. Die Beschreibungen synchronisierender Teile aller Prozesse des betrachteten Systems können kombiniert und auf Erfüllbarkeit, d.h. Existenz eines Modells, geprüft werden. Aus den gefundenen Modellen wird ein zentraler *Synchronisator* genannter Prozeß abgeleitet, welcher den synchronisierenden Teil des Gesamtsystems darstellt und bei allen Interaktionen beteiligt ist. Der Ansatz wird in Abbildung 5.13-A skizziert.

Die Anpassung an den Kontext zusammengesetzter Systeme besteht nun darin, daß Prozesse als interagierende Objekte- bzw. Komponenten eines separaten, einstufigen Verbundes *interpretiert* werden. Die zu synchronisierenden Prozesse bzw. die dadurch modellierten Komponenten werden zum einen in verwendete Teilkomponenten und zum anderen in verwendende Toplevelkomponenten, also im- und exportierte Funktionalität, untergliedert. Die *duale* Interpretation von Prozessen eröffnet die Möglichkeit einer universellen, integrativen Behandlung von Spezifikationen verschiedener Modellaspekte sowie deren gemeinsame Verifikation. In diesem Sinne kann man dann den Synchronisator als *lokale* Toplevelkomponente innerhalb eines *fokussierten* Komponentenverbundes betrachten, welcher wiederum Teil einer globalen Struktur sein kann. Der modifizierte Ansatz ist in Abbildung 5.13-B zu sehen.

Das so skizzierte Beschreibungsschema erlaubt auch *Teilspezifikationen*. Auf diese Weise kann der vorausgesetzten Annahme begegnet werden, daß konkrete Funktionalitäten von später verwendeten Teilkomponenten zum Entwurfs- bzw. Spezifikationszeitpunkt der Toplevelkomponente noch nicht bekannt sind. Zum Entwurfszeitpunt erlaubt der Ansatz eine Spezifikation der Komponentenfunktionalität aus externer Sicht. Dadurch wird in diesem Moment eine Verifikation von Konsistenz in Bezug auf interne Synchronisationsbedingungen sowie Relationen zu externen Funktionalitäten möglich. Soll später eine konkrete Komposition der Toplevelkomponente mit Teilkomponenten erfolgen, erlaubt das Modell den Abgleich derer Spezifikationen zum Kompositionszeitpunkt.

Beim Ansatz von Arapis wird die Spezifikation selber in zwei Teile bezüglich Export und Import gegliedert. Diese beiden Teilspezifikationen können als *Schnittstellen* gedeutet werden, und es folgt somit, daß der Ansatz von genau einer Im- und Exportschnittstelle ausgeht. Diese Tatsache führt in zweierlei

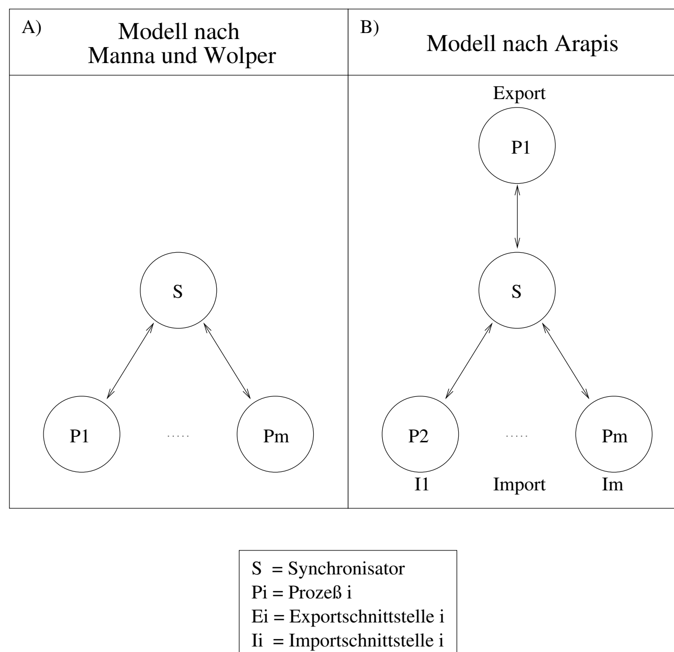


Abbildung 5.13: Modifikation des Spezifikationsmodells

Hinsicht zu einer unvorteilhaften Spezifikationsstruktur: Zunächst werden Aussagen über verschiedene integrierte Teilkomponenten miteinander vermascht. Des weiteren ist keine eindeutige Unterteilung exportierter Funktionalität in separate Aspekte gegeben, die den multiplen Exportschnittstellen des Komponentenmodells entspricht.

Wie die Ausführungen verdeutlichen, ist der Grundansatz modifizierter Prozeßsynchronisation mit Unterteilung in genau einen Nutzer und n konstituierende Teile auch im Komponentenkontext ein angemessenes Konzept. Potential für Verbesserungen besteht allerdings durch eine Restrukturierung der Spezifikation, um wesentliche Strukturaspekte der Modellebene adäquater abbilden zu können.

Restrukturierung der Spezifikationsstruktur Die Spezifikation einzelner Verhaltensaspekte geschieht nach obigem Ansatz durch das Aufstellen von PLTL-Formeln. Die verwendeten Propositionen entsprechen dabei den gesendeten bzw. empfangenen Nachrichten bezüglich eindeutig spezifizierter Interaktionspartner, die sich in genau einen potentiellen Nutzer und n Konstituenten teilen. Da solche Formeln durch Konjunktion verknüpft werden können, ist es möglich, sie bezüglich verschiedener Eigenschaften in Teilmengen zu untergliedern. Die Aufgabe besteht nun darin, eine im Komponentenkontext sinnvollen Strukturierung der Einzelaspekte vorzunehmen.

Für die Strukturierung der Einzelaspekte von Komponentenspezifikationen können drei Kriterien angegeben werden:

1. **Im-/Export** — Aspekte sollten nach *im-* und *exportierter Funktionalität* getrennt werden. Das bedeutet, Formeln, die entweder nur Nachrichten bezüglich des Nutzers oder der Konstituenten beinhalten, zu separieren.

2. **Schnittstellen** — Der modellhafte *Schnittstellenbegriff* sollte innerhalb der Spezifikation ein Pendant besitzen. Dabei sind zum einen *Importschnittstellen* zu beachten, welche sich in natürlicher Weise durch die verschiedenen Teilkomponenten abspalten, zum anderen sollten aber auch verschiedene *Exportschnittstellen* mit naturgemäß nur organisatorischem Charakter berücksichtigt werden.

3. **Interne Synchronisation** — Wechselwirkungen der separierten Teilaspekte sollten durch explizite *Synchronisationsaspekte* zusammengefaßt werden.

Ein exemplarischer Versuch für die konkrete formale Spezifikation einer Komponente nach obigem Ansatz ist die in Tabelle 5.2 gezeigte Idee einer exportorientierten Strukturierung.

Als Grundprinzip erfolgt hier eine Zerlegung nach Exportschnittstellen. Die Spezifikationen der Exportschnittstellen wird dann in die folgenden drei Bereiche unterteilt:

- **Lokal:** Beschreibung der Exportschnittstelle ausschließlich in Bezug auf die eigenen Bestandteile.

- **Export:** Beschreibung der Exportschnittstelle bezüglich ihrer Abhängigkeiten zu den n anderen Exportschnittstellen.

- **Import:** Beschreibung der Exportschnittstelle bezüglich ihrer Verwendung von importierter Funktionalität der m Teilkomponenten.

Ein auffälliges Merkmal dieses Ansatzes ist die Unterordnung von Aspekten genutzter Teilkomponenten bezüglich der Exportaspekte. Dies ist aber im Komponentenkontext plausibel, denn unter methodologischen Gesichtspunkten der Anwendung komponentenorientierter Systeme ergibt sich, wie in Kapitel 4 gesehen, die Ableitung benötigter Teilfunktionalität aus den Vorgaben der angestrebten Exportfunktionalität.

Ein direkter Nachteil dieser Vorgehensweise ist hingegen die erneute Zergliederung von Importschnittstellen. Da gerade diese Teilspezifikationen im laufenden Betrieb zum Abgleich dynamischer Kompositionen benötigt werden,

Gesamtspez. =	Export 1 $\parallel$	$\wedge$ ... $\wedge$	Export n $\parallel$
Lokal	E_1^l	$\wedge$ ... $\wedge$	E_n^l
$\wedge$			
Export	$E_1^{e_1}$ $\vdots$ $\wedge$ $\vdots$ $E_1^{e_n}$	$\wedge$ ... $\wedge$	$E_n^{e_1}$ $\vdots$ $\wedge$ $\vdots$ $E_n^{e_n}$
$\wedge$			
Import	$E_1^{i_1}$ $\vdots$ $\wedge$ $\vdots$ $E_1^{i_m}$	$\wedge$ ... $\wedge$	$E_n^{i_1}$ $\vdots$ $\wedge$ $\vdots$ $E_n^{i_m}$

Tabelle 5.2: Exportbasierte Spezifikation

wäre deren Separierung nicht zuletzt aus Gründen der Performanz äußerst wünschenswert. Als vorläufige Lösung kann man hier auf eine redundante Lösung ausweichen, bei der die Aspekte einzelner Importschnittstellen einfach einmalig extrahiert und separat hinzugefügt werden.

5.2.2.3 Rolle und Problematik der Propositionen

Propositionen bzw. Aussagen sind die grundlegenden Einheiten der propositionalen temporalen Logik. Die angestrebte Logik soll es erlauben, Aussagen über die Anwendungssemantik der funktionalen Einheiten von Komponenten zu machen. Die Aussagen der Logik müssen daher mit eben diesen funktionalen Einheiten in eineindeutiger Relation stehen.

Die wichtigste Anwendung des formalen Apparats besteht in der Verifikation von Verträglichkeit logischer Formeln — bzw. der Anwendungssemantik — verschiedener Komponenten. Diese Verifikation setzt eine übereinstimmende Interpretation der verwendeten Propositionen als Modellbestandteile — genauer Nachrichten der Modellebene — voraus. Um letztendlich mit Formeln arbeiten zu können, bedeutet das, daß die Syntax von Propositionen, welche in unterschiedlichen formalen Ausdrücken für die — aus Modellsicht — identischen funktionalen Einheiten stehen, ebenfalls identisch sein muß.

Eine einfache Lösung dieses Problems sind syntaktische Namenskonventio-

nen, wie sie auch bei dem oben gesehenen Ansatz von Arapis zum Einsatz kommen. Zusammengesetzte Komponenten beziehen sich auf Teilkomponenten bzw. deren Nachrichten durch Benutzung von syntaktischen Bezeichnern, die in den jeweiligen Bausteinen direkte Entsprechungen besitzen. Diese Vorgehensweise, die im Rahmen einer Entwurfsmethodologie durchaus angemessen sein mag, ist im Szenario eines globalen Komponentenmarktes kaum konsequent durchsetzbar und scheidet daher — zumindest in dieser reinen Form — aus.

Um diesen Problemen zu begegnen, sollen in den folgenden Sektionen Ansätze aus den Bereichen flexibler, logikbasierter Strukturbeschreibungen und Typisierung diskutiert werden.

5.3 Flexible Beschreibungsformen mit Feature-Logik

Ein genereller Nachteil traditioneller Strukturbeschreibungen ist ihre Starrheit. Strukturelle Eigenschaften werden meist fest in den Systemen kodiert, weshalb spätere Änderungen zu Problemen führen. Zudem führen absolute Strukturangaben, im Zusammenhang mit logikbasierten Verhaltensbeschreibungen, wie im letzten Abschnitt gesehen, auf globaler Interkomponentenebene zu Identifikations- bzw "Matching"-Problemen. Als flexibler Ansatz für dynamischere und generische Spezifikationen struktureller Aspekte könnte ein auf *Feature-Logik* basierendes Konzept dienen.

Im folgenden wird zunächst "Feature-Logik" als formale Basis eingeführt. Es schließt sich eine Diskussion über die darauf basierenden möglichen Spezifikationskonzepte an.

5.3.1 Feature Logik

Feature-Logik[3] ist eine typisierte Form der Logik, welche Ausdrücke enthält, die Objektmengen mit bestimmten Eigenschaften (engl. *Features*) sowie deren Beziehungen beschreiben. Die Logik kann dazu verwendet werden, Objekte mit bestimmten Eigenschaften dynamisch zu selektieren.

Features und Feature-Terme Grundbestandteil der Logik sind *Features*, welche die Abstraktion einer bestimmten Eigenschaft darstellen. Solche Features werden durch Paare von Namen und zugehörigen Werten charakterisiert, wobei der Bezeichner für die Art einer Eigenschaft steht und der Wert für dessen konkrete Ausprägung. Die konkreten Eigenschaften stehen

[3] siehe etwa [Smo92] oder im Zusammenhang mit Versionskontrolle [ZS96, Zel96]

in Beziehung zu einer Menge von Objekten, denen eben diese Eigenschaften potentiell innewohnen. Mengen dieser Objekte werden durch *Feature-Terme* beschrieben. Diese Feature-Terme sind logische Ausdrücke über Features. Die einfachste Form bestünde aus einem einzelnen Feature und würde für alle Objekte stehen, welche die entsprechende Eigenschaft besitzen. Features und Feature-Terme werden in Tabelle 5.3 charakterisiert.

Begriff	Bedeutung
Feature	Paar (Feature-Name:Feature-Wert) funktionaler Natur
Feature Term	Konstrukt aus Features mit Mengenoperatoren

Tabelle 5.3: Grundbegriffe der Feature-Logik

Für die Konstruktion von Feature-Termen aus Features können verschiedene *Mengenoperatoren* verwendet werden. Die entsprechenden syntaktischen Konstrukte für Feature-Terme und deren Semantik sind in Tabelle 5.4 aufgelistet.

Notation	**Name**	**Interpretation**
a	Literal	Menge die a beinhaltet
V	Variable	
$\top$ oder []	Decke	Universalmenge; Konsistenz
$\bot$ oder $\{\}$	Boden	Leere Menge; Inkonsistenz
$f : S$	Auswahl	Der Wert von f ist S
$f \uparrow$	Existenz	f ist definiert
$f \downarrow g$	Übereinstimmung	f und g haben den gleichen Wert
$f \uparrow g$	Differenz	f und g haben verschiedene Werte
$S \sqcap T$oder$[S,T]$	Schnitt	S und T sind wahr
$S \sqcup T$oder$\{S,T\}$	Vereinigung	S oder T sind wahr
$\sim S$	Komplement	S ist unwahr
$S \rightarrow T$	Implikation	wenn S wahr ist, dann auch T

Tabelle 5.4: Syntax und Semantik von Feature-Termen

Durch Anwendung der Operatoren können eine Vielzahl semantischer Aussagen modelliert werden. Bestimmte Objektmengen werden etwa typischerweise durch Aufzählung ihrer Eigenschaften bzw. Konjunktion der Features ($\sqcap$ oder $[\ldots]$) *selektiert.* Negationen ($\sim$) und Disjunktionen ($\sqcup$ oder $\{\ldots\}$) erlauben die Auswahl von *Alternativen.* Weitere Bedeutungen ergeben sich aus der jeweiligen Anwendungsdomäne.

Finden die vorgegebenen Eigenschaften eines Feature-Terms innerhalb der Objektmenge eine Entsprechung, nennt man diesen *konsistent*, andernfalls *inkonsistent.*

Die durch Feature-Terme beschriebenen Objektmengen erben die Eigen-

schaften ihrer Bestandteile, welche jedoch *angeglichen* bzw. "*unifiziert*" werden müssen, da Features grundsätzlich funktionaler Natur sind , d.h. daß sie bezüglich eines Objekte genau einen Wert besitzen.

Rollen Die mathematische Funktionalitätseigenschaft von Features erlaubt einerseits deren Einsatz zur Selektion von Objektmengen und ist somit Grundlage des Konsistenzbegriffs. Andererseits sind aber nicht alle interessierenden Eigenschaften funktionaler Natur. Das Konzept der Feature-Logik wurde daher von Smolka [Smo92] durch Elemente *deskriptiver Logik* generalisiert. Deskriptive Logik ergänzt die grundlegende Feature-Logik dabei um mengenbewertete Eigenschaften, welche als *Rollen* (*Roles*) bezeichnet werden. Jede Rolle eines Objektes enthält eine Menge von *Konzeptbeschreibungen* (*Concept Descriptions*), welche Syntax und Semantik der grundlegenden Feature-Logik (Features, Feature-Terme) erben. Bedingungen über diesen Rollen werden mittels Quantoren ausgedrückt (Abb. 5.5).

Notation	Name	Interpretation
$\forall r(S)$	Universale Quantifikation	Alle Werte von r sind in S
$\exists r(S)$	Existentielle Quantifikation	Mindestens ein Wert von r ist in S

Tabelle 5.5: Rollen-Quantoren in Konzeptbeschreibungen

Der *Existenzquantor* $\exists r(S)$ hat die Bedeutung "S enthält mindestens einen Wert von r". Der Ausdruck $\exists Nachricht(\uparrow Start)$ würde somit alle Objekte umschreiben, die eine Nachricht "$\uparrow Start$" besitzen. Der Ausdruck

$$[\exists Nachricht(\uparrow Start), \exists Nachricht(\uparrow Stop)]$$

würde sich dann auf alle Objekte beziehen, deren Nachrichtenmenge "$\uparrow Start$" *und* "$\uparrow Stop$" beinhaltet. Der *Allquantor* $\forall r(S)$ mit der Bedeutung "S enthält alle Werte von r" kann hingegen als *Filter* verwendet werden: Der folgende Term

$$\forall Nachricht(\uparrow Start \sqcup \uparrow Stop) = \sim \exists Nachricht(\sim\uparrow Start \sqcap \sim\uparrow Stop)$$

selektiert in diesem Sinne grundsätzlich alle Objekte außer denen, die weder "$\uparrow Start$" noch "$\uparrow Stop$" enthalten.

Funktionale Features werden in dieser Sicht als Spezialfall von Rollen mit jeweils genau einem Wert betrachtet und können so in das Schema integriert werden.

Beziehungen *Beziehungen* zwischen Objektmengen lassen sich durch Merkmale von Objektmengen in Form von Features bzw. Rollen darstellen.

Eine Beziehung zwischen zwei Objektmengen A und B wird im wesentlichen dadurch modelliert, daß eine Rolle von A die Objektmenge B als Wert enthält. Es lassen sich so beliebige n-zu-m Beziehungen beschreiben. Der folgende Ausdruck zeigt ein Beispiel:

$$A = [\exists Komponente(Maschine), nutzt : \exists Komponente(Motor)]$$

Der Term A repräsentiert die eins-zu-eins Beziehung "*nutzt*" eines Verbundes zwischen Toplevelkomponenten $Maschine$ und Teilkomponenten $Motor$. Die entsprechenden Komponentenmengen können grafisch als *Venn Diagramme* dargestellt werden. Beziehungen sind dann gerichtete Graphen zwischen den Mengen (Abb. 5.14).

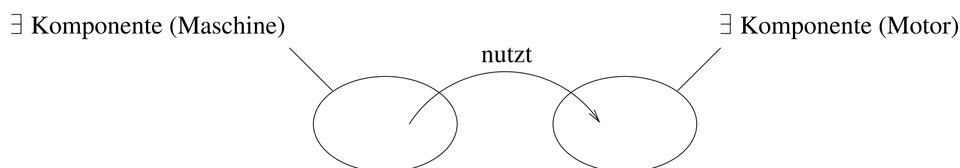


Abbildung 5.14: Beziehungsdiagramm

Beziehungen dieser Art sind existentiell mit den beteiligten Objektmengen verbunden, d.h. ist eine der Rollen inkonsistent, so gilt das auch für die Beziehung. Dies hat zur Folge, daß zum einen nur konsistente Beziehungen — also solche mit nichtleeren Ausgangs- und Endmengen — selektiert werden können, und zum anderen die Auswahl von Objekten auch immer deren Beziehungen mit einbezieht.

Im Zusammenhang mit solchen Beziehungen ist der Begriff der "*vollständigen Konfigurationen*" von wesentlicher Bedeutung. Eine Konfiguration (= Objektmenge, die sich aus einem Ausdruck der Feature-Logik ergibt) ist bezüglich einer Beziehung genau dann *vollständig* bzw. *konsistent*, wenn alle über diese Beziehung referenzierten Objekte auch in der Konfiguration enthalten sind. Für Konfigurationen, die bezüglich einer bestimmten Beziehung *inkonsistent* sind, existieren Methoden *automatischer Erweiterung* zur nachträglichen Vervollständigung.

5.3.2 Feature-Logik basierte Formalisierungskonzepte für Komponentenmodelle

Grundsätzlich erlaubt Feature-Logik die Auswahl von Objekten anhand einer Beschreibung ihrer Eigenschaften. Bezüglich des Komponentenmodells kann diese formale Semantik in zwei Bereichen angewandt werden. Zum einen auf die Selektion von *Struktureinheiten* und zum anderen auf die von *Verhaltensaspekten*. Als Features und somit Selektionskriterien müssen

in expliziter Weise geeignete Modelleigenschaften bestimmt werden, welche bezüglich beider Objektpools — Strukturen bzw. Formeln — einheitlich sein können (Tabelle 5.6).

	Formale Semantik	**Modell-semantik**	
Gegen-stände	Beschreibung abstrakter Objektmengen	Beschreibung von Mengen bestehend aus strukturellen Modellbestandteilen wie Nachrichten oder Schnittstellen	Beschreibung von Mengen bestehend aus Verhaltensaspekten in temporaler Logik
Aus-drucks-mittel	Features	Explizite Modelleigenschaften	

Tabelle 5.6: Formale und modellbezogene Sicht der Feature-Logik

Die Möglichkeiten, ein Komponentensystem mit diesen Mitteln abzubilden, sind äußerst vielfältig. Als exemplarisches Beispiel wird im folgenden ein kompaktes Basiskonzept informal skizziert. Im Anschluß folgen dann einige weiterführende Ideen bezüglich verfeinerter Sichten.

5.3.2.1 Eine Basisspezifikation

Komponentenorientierte Strukturspezifikationen mit Feature-Logik könnten auf Basis des folgenden Schemas verwirklicht werden. Der Ansatz beinhaltet eine globale Objektmenge von *Struktureinheiten* und drei wesentliche Features:

- **Objekt**: Art der Struktureinheit
- **Informale_Semantik**: Eindeutige, abstrakte Bedeutung
- **Formale_Semantik**: Verhaltensspezifikation

"Objekt" bezeichnet die Struktureinheiten, wobei in diesem vereinfachten Konzept der Wertebereich als { Komponente, Attribut, Nachricht, Verhaltensaspekt } definiert ist. Das zweite Feature "Semantik" teilt sich in eine informale Variante, die für eine explizite Typisierung der internen — über die

Sicht der Architektur hinausgehenden — Bedeutung von Strukturelementen steht, und eine formale Variante mit temporallogischer Beschreibung von zeitlich/kausalen Verhaltensaspekten. Der Wertebereich umfaßt hier beliebige Bezeichner bzw. PLTL-Formeln.

Die Objekte sind ferner durch eine Reihe von *bidirektionalen Beziehungen* verknüpft, wobei stets Komponenten als zentraler Bezugspunkt involviert sind. Der duale Ansatz erleichtert später den Umgang mit den Spezifikationen, da er die Selektion auf Basis beliebiger Seiten ermöglicht. Folgende Beziehungen sind bezüglich des zentralen Komponentenkerns enthalten:

- **Export**: Menge exportierter Interaktionseinheiten (=Nachrichten)
- **Zustand**: Attributmenge, die den Komponentenzustand abstrahiert
- **Verhalten**: Menge der Verhaltensaspekte (=PLTL-Formeln)
- **Teil**: Toplevelkomponente bzw. Teilkomponentenmenge

Zur Veranschaulichung zeigt Abbildung 5.15 das Spezifikationsschema am Beispiel der bekannten Maschinenkomponente.

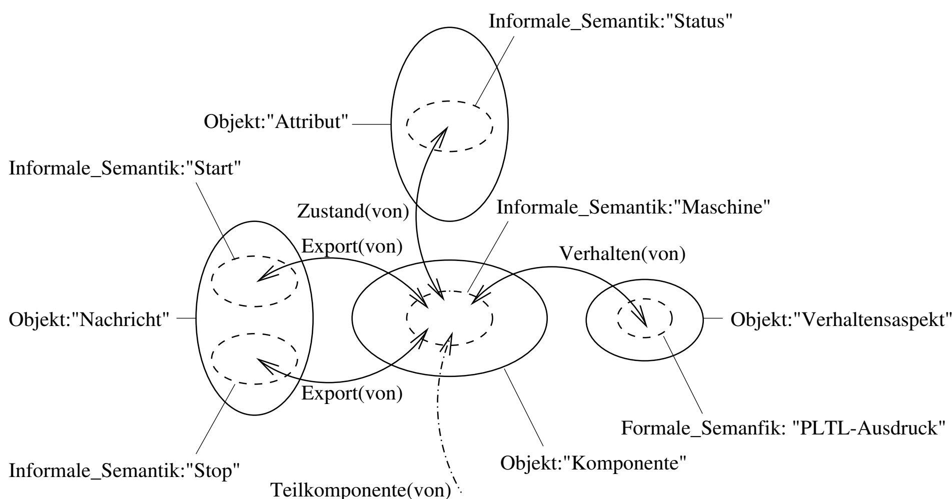


Abbildung 5.15: Anwendung der Feature Logik zur Komponentenstrukturbeschreibung

Es ist nun möglich, mit Hilfe von Feature-Logik Teile einer Komponente *dynamisch zu selektieren.* Die Nachrichtenmenge einer gegebenen Komponente kann demnach wie folgt umschrieben werden:

$$A = \left[\begin{array}{l} \exists\, Objekt(Nachricht), \\ Export_von : \exists\, Informale_Semantik(Maschine) \end{array} \right]$$

Es ist so ebenfalls möglich, zu einem gewünschten — also gegebenen — Verhalten die Menge aller Komponenten zu finden, welche diesem genügen. Der entspreche Ausdruck lautet dann:

$$A = \left[\begin{array}{l} \exists\, Objekt(Komponente), \\ Verhalten : \exists\, Formale_Semantik(Wunschverhalten) \end{array} \right]$$

Wie diese Beispiele zeigen, ist eine Spezifikation nach dem vorgestellten Konzept durch seine dynamischen Beziehungen weitaus weniger starr als herkömmliche Systeme. Des weiteren beinhaltet der Ansatz eine erste, wenn auch separierte Integration von Strukturspezifikation und Verhaltensbeschreibung.

5.3.2.2 Verfeinerte Beschreibungsansätze

Als weitere exemplarische Ansätze sollen nun noch einige alternative, verfeinerte Konzepte zur Sprache kommen, welche die Vielfalt der Möglichkeiten verdeutlichen.

Potentielle Verfeinerungen des oben gezeigten Ansatzes bieten sich vor allem in zwei Bereichen an. Zum einen kann die *interne Struktur* von Komponenten detaillierter modelliert werden, zum anderen sind die *Beziehungen zwischen Komponenten* genauer abbildbar.

Bezüglich des ersten Aspektes bieten sich sofort *Schnittstellen* als zusätzliche Struktureinheiten an. Des weiteren könnten aber auch beliebige andere Einheiten wie etwa *Kanäle* berücksichtigt werden. Da *Verhaltensaspekte* einer Komponente meist nur teilweise — etwa bei der Verifikation einer Komposition — benötigt werden, scheint es an dieser Stelle sinnvoll zu sein, eine Untergliederung bezüglich der gewählten Strukturebenen vorzunehmen. Ein entsprechendes Spezifikationsschema ist in Abbildung 5.16 zu sehen.

Die Strukturierung der Komponente bezieht in diesem Fall zwei weitere Ebenen, nämlich *Schnittstellen* und — exemplarisch für eine beliebige Abstraktion stehend — *Kanäle* mit ein. Letztere Ebenen stellen allerdings formal gesehen keine Erweiterung der Ausdrucksmächtigkeit dar, denn die Verhaltensspezifikation mittels PLTL berücksichtigt weiterhin die atomaren Nachrichten. Aus diesem Grund nehmen Nachrichten in dem Ansatz eine Sonderstellung ein, indem sie in unmittelbarer Beziehung zur Komponenteneinheit stehen. *Kompositionsbeziehungen* zwischen Komponenten werden in diesem Ansatz *indirekt* berücksichtigt. Komponenten referenzieren die Strukturebenen jeweils als geforderte oder bereitgestellte Einheiten und liefern zu diesen eine spezifische Form der Verhaltensbeschreibung. Mittels Feature-Logik kann nun zu einer Komponente nicht nur deren Strukturierung flexibel selektiert werden, sondern es ist auch ein Retrieval aller potentiellen Gegenstücke möglich. Zu diesen liegen dann auch unmittelbar die relevanten Teile ihrer

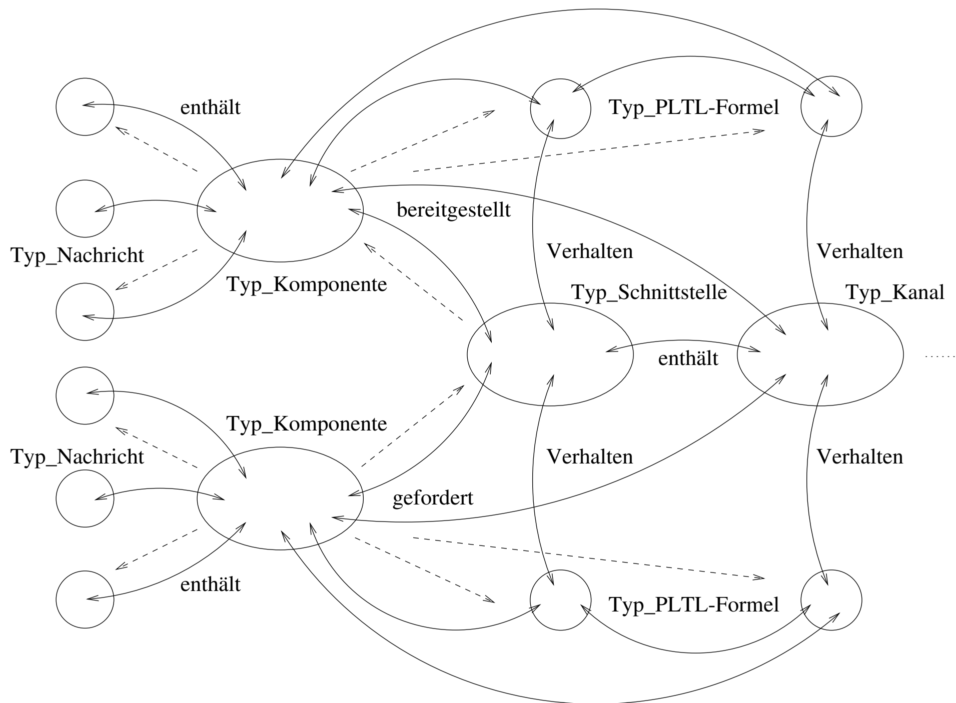


Abbildung 5.16: Multi-Level Modell mit beliebigen Strukturen

Verhaltensbeschreibungen vor, welche sofort mit den eigenen Gegenstücken abgeglichen werden können.

Um die gewünschten Informationen durch Feature-Logik zu ermitteln, kann eine sukzessive Vorgehensweise angewandt werden. Für eine Objektmenge mit den Features "Objektart" und "Semantik" sind die geforderten Importschnittstellen einer bekannten Komponente wie folgt bestimmbar:

$$I = \exists\, gefordert_von\, (Semantik : \text{“}Komponentenname''\text{)}$$

Mögliche Gegenstücke werden dann mit dem nachfolgenden Ausdruck gefunden:

$$B = \exists\, bereitgestellt\, (Semantik : \text{“}Schnittstellenname''\text{)}$$

Sollen nun die zeitlichen Verhaltensaspekte abgeglichen werden, müssen die PLTL-Ausdrücke der Schnittstellenebene sowohl von Bereitsteller- als auch Nutzerseite bestimmt werden. Die entsprechende Formel der Feature-Logik lautet:

$$F = \exists\, Verhalten_von \left(\begin{bmatrix} Semantik : \text{“}Schnittstelle1'', \\ \left\{ \begin{array}{l} \exists\, gefordert(Semantik : \text{“}Komponente1''), \\ \exists\, bereitgestellt(Semantik : \text{“}Komponente2'') \end{array} \right\} \end{bmatrix} \right)$$

Bei Interesse an weiteren Strukturstufen kann schließlich mit dem nächsten Ausdruck die Menge von in der aktuellen Stufe befindlichen Untereinheiten

selektiert werden:

$$L = \left\{ \begin{array}{l} \forall enthalten_in \left(\left[\begin{array}{l} Semantik: \text{“}Schnittstelle1\text{''}, \\ \exists gefordert(Semantik: \text{“}Komponente1\text{''}) \end{array} \right] \right), \\ \\ \forall enthalten_in \left(\left[\begin{array}{l} Semantik: \text{“}Schnittstelle1\text{''}, \\ \exists bereitgestellt(Semantik: \text{“}Komponente2\text{''}) \end{array} \right] \right) \end{array} \right\}$$

Für weitere Ebenen sowie die jeweiligen spezifischen Verhaltensaspekte gelten die gleichen Formeln wie oben.

Der nächste konsequente Schritt einer weiterführenden Konzeptionierung ist die *Multilevel-Spezifikation*, welche interne Strukturierungsbegriffe von Komponenten gänzlich offen läßt. Die Grundidee ist dabei eine Abstraktion von konkreten Strukturelementen zwischen den Grenzen von Komponente und funktionaler bzw. kommunikativer Basiseinheit (i.A. Nachrichten). Verschiedene Level wie Kanäle, Prozeduren, Methoden oder Schnittstellen ergeben sich dann aus den konkreten Beziehungen, welche durch Feature Logik formuliert werden. Jeder Level kann dabei wieder seine eigenen semantischen Constraints enthalten, welche zu Vergleichszwecken durch Retrieval-Mechanismen der Feature-Logik konzentriert und vereint werden. Die Constraints in temporaler Logik sind dabei durch ihren Bezug auf eindeutige Basiseinheiten der Kommunikation nicht von den Ebenen abhängig.

5.4 Übergreifende Aspekte formaler Konzeption

Bei den bisherigen Betrachtungen des Themengebietes Spezifikation wurden Ansätze zu verschiedenen Aspekten wie der Beschreibung kausalen/zeitlichen Verhaltens oder der flexiblen Strukturbeschreibung behandelt. Die Ausführungen konzentrierten sich dabei jeweils auf den separierten Aspekt, ohne weiter auf *Wechselwirkungen* einzugehen. Eben diese Wechselwirkungen, seien sie nun positiver oder negativer Natur, sollten aber ebenfalls beachtet werden. Letztere sind insbesondere dann von Interesse, wenn Einzelbetrachtungen Defizite aufweisen, bei denen die Synergie eines integralen Konzeptes auf Lösungsansätze hoffen läßt. Im folgenden wird zunächst die Kombination der logikbasierten Spezifikationsansätze betrachtet. Als klassische Form übergreifender, integrierender Formalisierung schließen sich Fragen bezüglich möglicher *Typisierungen* an.

5.4.1 Kombinationsmöglichkeiten deskriptiver und temporaler Logiken

Wie bereits gesehen, scheint durch temporale Logik alleine keine zufriedenstellende Komponentenbeschreibung möglich zu sein, denn es kann zwar ei-

ne aussagekräftige, kausale Verhaltensbeschreibung formuliert werden, das Festmachen derer Gegenstände — also der agierenden Strukturen — ist aber nicht in ausreichend flexibler Weise möglich. Genau an dieser Stelle könnte nun deskriptive Feature-Logik möglicherweise einen positiven Beitrag leisten. Abbildung 5.17 veranschaulicht nochmal die Rollen der Logiken bei Komponentenspezifikationen.

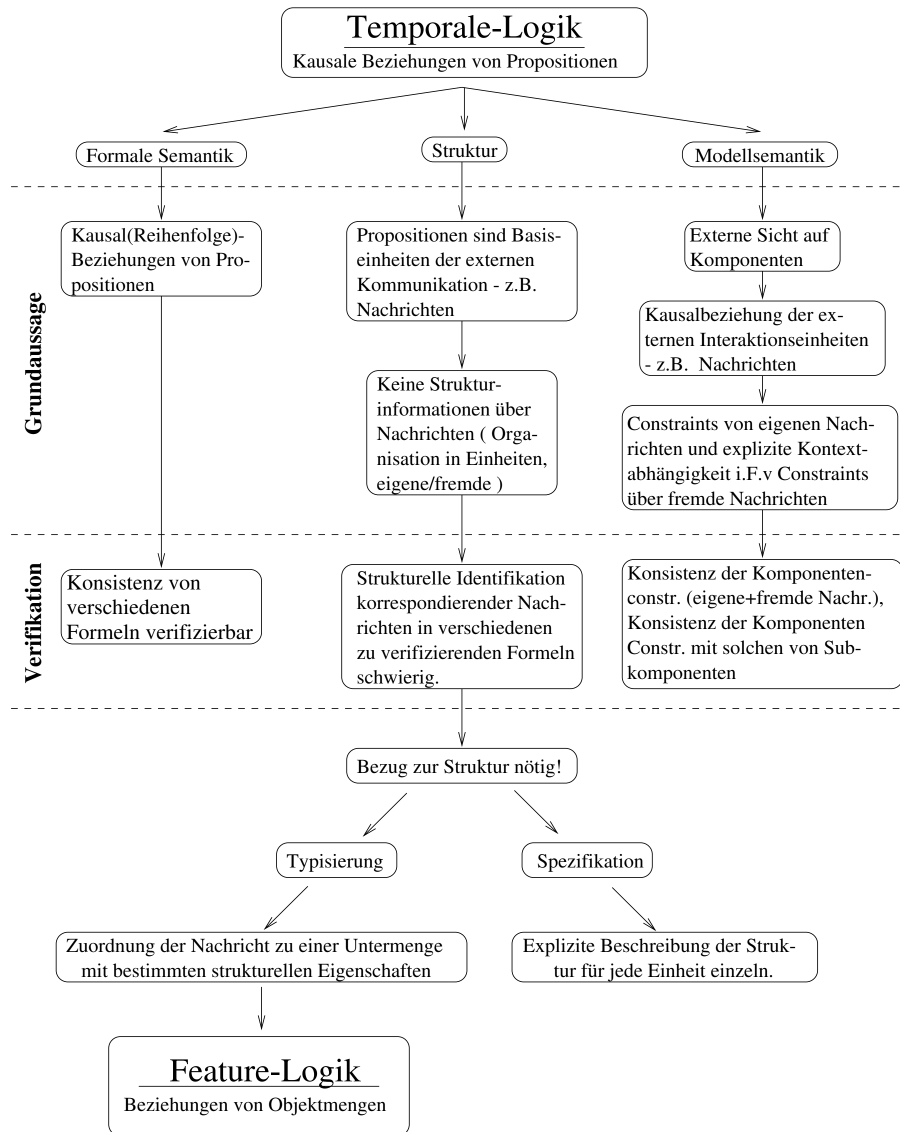


Abbildung 5.17: Rollenverteilung der Logiken

Ähnlich wie im letzten Kapitel für Feature-Logik werden in der Abbildung formale und modellbezogene Semantik für temporale Logik verglichen. Die kausalen Folgen abstrakter Propositionen des Formalismus bekommen im

Modell die Bedeutung von *Kommunikationsprotokollen* bezüglich atomarer, externer Interaktionseinheiten. Gleichzeitig wird der fehlende Bezug zur strukturellen Semantik deutlich, denn die verschiedenartigen Modelleinheiten werden durch Propositionen abstrahiert, welche ihrer Natur nach gleichartig sind und keine Differenzierung erlauben. Diese Problematik entfaltet ihre negativen Einflüsse bei der Anwendung, welche hauptsächlich im Abgleich von Kompositionen durch Verifikation temporaler Constraints auf ein gemeinsames Modell besteht. Man kommt in die Situation, daß syntaktisch potentiell divergente Propositionen Abbilder identischer Modelleinheiten darstellen. Für ein strukturelles Matching besteht dann keine Informationsgrundlage. Genau in diesem Strukturbezug temporaler Constraints liegt der Berührungspunkt zur deskriptiven Feature-Logik. Eine Kombination der beiden Logiken birgt die Hoffnung verbesserter Bausteinbeschreibungen, denn sie könnte es möglich machen, die Semantik von Verhalten und Struktur zu verbinden.

Es stellt sich nun die Frage nach den *Integrationsmöglichkeiten* der Formalismen. Grundsätzlich gäbe es dabei zwei Varianten zur Kombination von Feature- und temporaler Logik:

- **Beschreibung temporaler Constraints mit Propositionen in Form von deskriptiven Feature-Termen.**

  Der erste Ansatz besteht darin, die bisherigen temporalen Ausdrücke um aussagekräftigere Propositionen zu erweitern. Nimmt man etwa den Ausdruck

  $$\Box\,(Start \Rightarrow \triangle\, Stop),$$

  käme man durch Ersetzung der Propositionen durch Feature-Terme nach dem Basisschema im letzten Kapitel etwa zu folgendem ersten Ergebnis:

  $$\Box\,([Semantik : \text{“Start”}] \Rightarrow \triangle\,[Semantik : \text{“Stop”}])$$

  Die Propositionen haben nun statt der syntaktischen eine formale Identität erhalten. Hierdurch ist jedoch nichts gewonnen, denn die “formale Identität” der Feature-Terme beruht letztlich wieder auf den syntaktischen Bezeichnern, welche den Wertebereich der Features ausmachen!

  Ein gewisser positiver Effekt läßt sich dennoch erwirken: Propositionen

können etwas ausführlicher etwa wie folgt beschrieben werden:

$$\Box \left(\begin{array}{c} \left[\begin{array}{l} Objekt : \text{“Nachricht”}, \\ Semantik : \text{“Start”}, \\ \exists\, Import_von(Semantik : \text{“Maschine”}) \end{array} \right] \\ \Rightarrow \ \Delta \\ \left[\begin{array}{l} Objekt : \text{“Nachricht”}, \\ Semantik : \text{“Stop”}, \\ \exists\, Import_von(Semantik : \text{“Maschine”}) \end{array} \right] \end{array} \right)$$

Durch den Bezug auf eine gegebene Komponente schränkt sich der *Namensraum* für Nachrichtenbezeichner auf diese ein. Dadurch wird ein anschließendes Matching zumindest erleichtert.

- **Beschreibung von Mengen struktureller Modellbestandteile durch Feature-Logik anhand temporaler Constraints als Features.**

  Der zweite Integrationsansatz sieht vor, Features zu definieren, deren Wertebereich Ausdrücke temporaler Logik sind. Das Ergebnis ist aber noch unbefriedigender als im vorherigen Fall, denn ein zur Selektion nötiger Vergleich von Features entspricht genau dem gleichen Vorgang wie bei der herkömmlichen Spezifikationsform. Die Problematik anonymer Propositionen bleibt gänzlich unberührt und wird nur in eine andere Ebene verlagert.

Nach längerer intensiver Untersuchung der “Matching”-Problematik äquivalenter funktionaler Einheiten verschiedener Komponenten zeichnet sich als Ergebnis ab, daß es zu einer expliziten und einheitlichen Benennung — was im übertragenen Sinne einer expliziten Typisierung entspricht — auf globaler Ebene z.Z. keine Alternative gibt. Durch Versuche, funktionale Einheiten mit Hilfe von Feature-Logik abstrakt zu umschreiben, können die erhofften Resultate wie gesehen nicht in vollständig befriedigender Form erreicht werden.

Die Ursachen liegen darin, daß man das Wesen des Unterschieds verschiedener funktionaler Einheiten aus einer Außensicht nicht formal fassen kann. Dieses Wesen liegt in der den Einheiten *innewohnenden* Semantik. Letztere wird in vollständiger Weise nur durch die Implementation beschrieben, welche aber durch eine Außensicht gekapselt werden soll. Kapselung impliziert stets eine, wenn auch frei wählbare Grenze externer Einblicke. Ab dieser Grenze — der *Abstraktionsbarriere* — läßt sich die Semantik nur durch explizite Benennung unterscheiden.

Wenn man berechtigterweise globale *Namenskonventionen* aus pragmatischen Gründen ausschließt, so muß man auf zumindest *semiautomatische* oder gar komplett *benutzergetragene* Matching-Mechanismen zurückgreifen. Es kommen dann z.B. explizit vom Benutzer definierte Beziehungen in Frage, welche in Repositories gehalten und bei Bedarf abgefragt werden können. Ein konkretes Beispiel für so ein System wäre ein *Typmanager*.

5.4.2 Typisierungsansätze

Um sich den Problembereichen der bisher behandelten Formalisierungsansätze anzunähern, bietet sich als denkbarer Ansatzpunkt traditionell die ergänzende Verwendung von Typisierung an (siehe z.B. in [Nie95]). Nicht zuletzt deuteten auch die Ergebnisse des letzten Abschnitts auf (explizite) Typisierung als ein potentiell weiterführendes bzw. unumgänglich notwendiges Konzept hin. Angewendet auf die Problematik strukturellen Informationsmangels temporallogischer Ausdrücke, bietet sich zunächst eine Anwendung des Prinzips auf Propositionen an.

Propositionen könnten demnach neben den komponenteninternen Bezeichnern noch deren *Typinformationen* beinhalten. Das hieße, in expliziter Weise global eindeutige Klassen mit semantisch als gleich erachteten Propositionen bzw. Nachrichten einzuführen. Der Semantikbegriff bezöge sich dabei auf die interne Bedeutung von Einheiten unterhalb der Abstraktionsbarriere, nicht jedoch auf die Verhaltenssemantik, welche im Kontext der Architektur beschrieben werden soll. Der Nutzen von Typisierung ergibt sich nun durch Einführung von *Typbeziehungen*, welche den Rückschluß auf Eigenschaften der entsprechenden Instanzen erlauben, wobei die Grundlage der Beziehungen unerheblich ist. Die wesentlichen Beziehungsformen sind dabei *Konformität* und *Kompatibilität*.

Durch die Verwendung solcher Typen funktionaler Einheiten alleine wäre aber nach wie vor keine eineindeutige Zuordnung verschiedener Bezeichner/Typ Paare möglich, da Typen immer für eine ganze Menge von Instanzen stehen. Käme es etwa zu der Situation, daß kompatible Typen in einer Formel mehrmals, in einer anderen hingegen nur einmal aufträten, würde für das Matching der Propositionen keine weitere Informationsgrundlage existieren. Diese Tatsache gilt dabei nicht nur für das mehrfache Auftreten eines Typs, sonder ergibt sich schon für konforme Arten.

Es stellt sich in diesem Zusammenhang die Frage, ob wie hier geschehen die Logik typisiert werden soll, oder ob nicht umgekehrt Typen Logik enthalten könnten. In diesem Fall würde die Typisierung auf Semantik der Komponentenebene beruhen. Es muß also geklärt werden, wie die beiden Formalismen *(statische) Typisierung* und *(semantische) Spezifikation* in Zusammenhang gebracht werden sollten.

Als zweite Alternative für Typisierungsansätze sind in diesem Sinne logikbasierte Typen zu betrachten. Zunächst scheint sich dabei die Integration temporaler Constraints in ein Typsystem als natürliche Lösung anzubieten. Die Idee dabei ist eine Typisierung auf Grundlage von PLTL-Spezifikationen, wie sie auch in [Ara95] und [Nie95] angedacht wurde. Komponenten(-Schnittstellen) werden dabei auf Grund der kausalen Beziehungen ihrer ein- und ausgehenden Nachrichten typisiert. Die Spezifikation dieser Verhaltensaspekte erfolgt mit Hilfe von PLTL-Formeln. Der strukturellen Klassifizierung von Typen werden in diesem Sinne einfach temporale Constraints eben dieser Strukturtypen zugegeben, und jede Instanz ist damit verpflichtet, ihnen zu genügen. Ein Typmanagement kann dann automatisch Typkonformität validieren. Matchingprobleme treten nicht auf, da alle Propositionen eindeutige Typbezeichner einer globalen Typmenge sind. So verwendet ein Komponententyp z.B. die Funktionalität einer Subkomponente, indem er deren global eindeutige Typinformation referenziert und der Nomenklatur seiner Spezifikation zugrunde legt. Leider gilt auch für diesen Ansatz die Tatsache, daß er zwar für Instanzen des gleichen Typs ein kompatibles Verhalten im Rahmen der Spezifikation garantiert, jedoch keine eineindeutige Identität sicherstellt, da Typen eben immer für ganze Klassen von Instanzen stehen, die sich auf detaillierterer Ebene durchaus unterscheiden können.

Am Ende dieses Abschnittes soll noch kurz — als Einschub — die Thematik von Typbeziehungen im speziellen Fall PLTL-basierter Typen angesprochen werden. Bezüglich sinnvoller Typbeziehungen können die Ergebnisse von [Nie95] als Ansatzpunkt genommen werden, die in ähnlicher Form schon in früheren Kapiteln als wesentlich erkannt wurden. Nierstrasz unterscheidet folgende Relationen:

- Typkonformität oder -substituierbarkeit
- Typbefriedigung oder -kompatibilität

Es stellt sich dann aber die Frage, wie die Verifizierung von PLTL-Spezifikationen auf Konformität (siehe oben oder [Ara95]) in Beziehung zur Verifikation von Typbeziehungen steht. So ist die Existenz mindestens eines gemeinsamen mathematisch logischen Modells (einer Ausführungsfolge, welche beiden temporalen Constraints genügt) sicher kein geeignetes Kriterium für eine Typkonformität/-substituirbarkeit. Vielmehr müßte hier das Kriterium von [Nie95] angewandt werden, welches besagt, daß ein Typ A durch einen zweiten Typ B austauschbar ist, wenn folgende Bedingungen erfüllt sind:

1. Alle endlichen, gültigen Ausführungsfolgen (Modelle) des Typs A sind auch Modelle des Typs B.

2. Alle Propositionen, welche im Anschluß an ein beliebiges Modell des Typs A für diesen gültig sind, sind auch für den Typ B im Anschluß an dasselbe Modell gültig.

Der kurze Blick auf Typisierung in Bezug zu den exemplarischen Architekturaspekten des Kapitels soll mit diesem letzten Ergebnis beendet sein.

5.4.3 Überblick und Bewertung der Ergebnisse

Nachdem nun einige formale Konzepte vorgestellt wurden, soll im folgenden eine kurze Bewertung der Ergebnisse vorgenommen werden.

Die behandelten Konzepte umfaßten Formalismen zur Beschreibung semantischer Konzepte von Komponenten. Zunächst wurde ein auf temporaler Logik basierender Ansatz zur Spezifikation von kausalen Verhaltensmustern betrachtet. Dieser erlaubte eine natürliche Formulierung bereitgestellter und erwünschter Eigenschaften von Bausteinen. Auf diese Weise dargestellte Verhaltensmuster konnten auf gegenseitige Verträglichkeit bzw. Konformität geprüft werden, wobei zu diesem Zweck effektive Algorithmen bestehen. Der zweite Ansatz, welcher sich auf Feature-Logik gründete, hatte dynamische Spezifikationsmöglichkeiten von internen und externen Komponentenstrukturen zum Inhalt. Die typisierte Feature-Logik erlaubte die Auswahl beliebiger Struktureinheiten anhand derer Eigenschaften. Beschreibungen von Bausteinen waren auf diese Weise relativ zu deren in- und externen Merkmalen möglich und dadurch dynamisch und flexibel anstatt wie bei herkömmlichen Beschreibungen — etwa in IDL — fest kodiert und entsprechend starr.

Neben den attraktiven Potentialen der Ansätze ergaben sich leider auch explizite Grenzen der Anwendbarkeit. Die erkannten Eigenschaften der Konzepte machen deren Einsatz in bestimmten Szenarios äußerst fragwürdig. Die Problematik ließ sich dabei an einem Zielkonflikt zwischen Abstraktion und Spezifikation festmachen. Während Komponenten einerseits durch Spezifikation in generischer Weise eindeutig beschrieben werden sollen, werden andererseits durch Abstraktionsbarieren Grenzen dieser Beschreibung vorgegeben, welche dann ein — bezüglich des Komponentensystems — global einheitliches Verständnis der Abstraktionen voraussetzen. Eben dieses global einheitliche Verständnis ist in Szenarios großer, d.h. weltweiter Komponentenmärkte problematisch.

Trotz der Einschränkungen bleiben die Konzepte in anderem Kontext durchaus vielversprechend. Bei Fokussierung auf eingeschränktere Szenarien wie lokale Entwicklungsumgebungen oder ausgewählte Domänen ist die Problematik eines global eindeutigen Verständnisses von Abstraktionen wesentlich einfacher zu handhaben. In diesem Sinne betrachtet das folgende Kapitel u.a. einige Ansätze zur Integration der formalen Semantiken in konkrete Techniken und Systeme.

Kapitel 6

Realisierungsmöglichkeiten komponentenorientierter Architekturkonzepte am Beispiel JavaBeans

In den letzten beiden Kapiteln wurden grundlegende Eigenschaften von Komponentenarchitekturen und verschiedene Ansätze zu deren Umsetzung betrachtet. Dieses Kapitel untersucht nun die gewonnenen Ergebnisse anhand einer konkreten Technik. Als exemplarisches Beispiel wird dabei die Sun JavaBeans Architektur verwendet, welche einen der z.Z. bedeutendsten Komponentenstandards darstellt. Im folgenden werden JavaBeans zunächst in Sektion 6.1 mit den in dieser Arbeit hergeleiteten allgemeinen Eigenschaften abstrakter Komponentenarchitekturen verglichen. Es folgen in Sektion 6.2 Ansätze zur Erweiterung von JavaBeans um fehlende bzw. wünschenswerte Konzepte. Das Kapitel schließt mit einem zusammenfassenden Fazit in der Sektion 6.3.

6.1 Konzeptionelle Untersuchung

Die schon in Kapitel 3.1.1 erwähnte *JavaBeans* Architektur ist ein von Sun Microsystems eingeführter Industriestandard für Softwarekomponenten, welcher auf der plattformunabhängigen Sprache *Java* (siehe Kapitel 3.2.1) basiert. Dieser Standard wird in der *JavaBeans API Spezifikation* [Sun97a] definiert, welche die Kernspezifikation der JavaBeans Komponentenarchitektur beschreibt. Eng damit verbunden sind eine Reihe weiterer Spezifikationen verwandter Themen. Zu diesen gehören die *Java Core Reflection API*, *Java Object Serialization*, *JDK 1.1 — AWT Enhancements*, die *JAR*

file specification, *Remote Method Invocation* und *Java IDL*.

JavaBeans Komponenten (kurz: *Beans*) sind abgeschlossene, wiederverwendbare Softwarebausteine mittlerer Granularität, welche vornehmlich zur visuellen Komposition mit Hilfe entsprechender Werkzeuge bestimmt sind und zumeist GUI Elemente darstellen; allerdings ohne entsprechendes Dokumentenmodell mit geeigneten Containern. Die Basisarchitektur der JavaBeans sieht keine Verteilung vor, d.h Beans sind zur wechselseitigen Kombination innerhalb genau einer Java-Virtual-Machine vorgesehen. Trotzdem stehen den Beans Komponenten die grundsätzlichen Eigenschaften von Java mitsamt der Plattformunabhängigkeit innerhalb einer potentiell vernetzten Umgebung zur Verfügung.

Die grundlegenden Ziele von JavaBeans sind dann auch in Plattformunabhängigkeit sowie Mobilität in vernetzten Umgebungen zu sehen und lehnen sich somit eng an die wesentlichen Vorteile der Basissprache Java an. Während konkurrierende Ansätze von Microsoft (COM) oder Macintosh (OpenDoc) die jeweiligen Plattformgrenzen nicht überschreiten, ist JavaBeans in diesem Sinne eher ein integratives als ein weiteres unabhängiges Komponentenmodell, da ein deutlicher Schwerpunkt auf der Überbrückung von Systemgrenzen liegt.

Die Sektion unterteilt sich im folgenden in eine erweiterte Übersicht der JavaBeans Architektur sowie deren anschließende Analyse und Bewertung in Bezug auf die Ergebnisse der letzten Kapitel.

6.1.1 JavaBeans: Ein Überblick

Die vorliegende Untersektion liefert einen konzeptionellen Überblick der Komponentenarchitektur JavaBeans. Es soll dabei zunächst in einem ersten Teil das zugrundeliegende Komponentenmodell untersucht werden. Der zweite Teil beinhaltet dann eine Betrachtung methodologischer Anwendungsaspekte.

6.1.1.1 Das Komponentenmodell

Komponentencharakterisierung JavaBeans Komponenten sind *zustandsbehaftete Laufzeiteinheiten*, die jeweils aus einer Menge potentiell verknüpfter Java Objekte bestehen. Insbesondere handelt es sich bei Beans nicht um Klassen, welche zur Generierung von Laufzeitobjekten instanziiert werden müssen. Es handelt sich jedoch auch nicht um herkömmliche Objekte, sondern um Mengen von Objekten mitsamt deren jeweils eigenen Initialzuständen. Beans werden entsprechend durch die den Objekten zugehörigen Klassen sowie eine Serialisierung der initialen Objektkonfigurationen repräsentiert. Da sich Beans keinerlei Spracherweiterungen bedienen

— d.h. weder von einer speziellen Basisklasse noch einem Interface erben müssen — können sie in beliebige Java-Umgebungen integriert werden.

Kapselung und Abstraktionsgrenze Bean Komponenten sind Sammlungen von Objekten. Aus externer Sicht bestehen sie jedoch immer aus genau einem Objekt. Ist intern mehr als ein Objekt vorhanden, fungiert eines davon explizit als externer Zugangspunkt.

Als Distributionsform der Architektur dienen *JAR* (*Java Archive Format*) Dateien, welche JavaBeans enthalten. Obwohl Beans keine einfachen Klassen sind, liegen sie bei Migrationen stets in "eingefrorener" Form vor, die in eine Laufzeit-Repräsentation erhoben werden muß. Analog dem `new` Operator in Java, der zur Erzeugung von Instanzen einer Klasse dient, existiert in Java eine öffentliche, statische Methode `java.beans.Beans.instantiate`, welche eingefrorene Beans in Laufzeitkomponenten überführt. Innerhalb eines Java Programms würde die "Reinkarnation" einer Bean dann etwa wie folgt aussehen:

```
SomeBean sb = ( SomeBean )
        java.beans.Beans.instantiate ( 'MyBean' );
```

Die Schnittstelle der Komponente ist danach durch die öffentlichen Methoden und Attribute der Bean gegeben. Intern besteht der Baustein aus allen Objekten, die zur Instanziierungszeit durch den speziellen Initialisierungscode der Komponente erzeugt wurden.

JavaBeans sind in diesem Sinne Blackbox-Einheiten, da ihre innere Struktur vor den potentiellen Nutzern verborgen wird, falls dies nicht explizit durch den Komponentenentwickler umgangen wird.

Schnittstellen Bei JavaBeans Komponenten werden mit *Entwicklungszeit-* und *Laufzeitschnittstellen* zwei grundsätzlich verschiedene Arten extern zugreifbarer Funktionalität unterschieden.

Die beiden Arten unterscheiden sich dabei sowohl konzeptionell als auch vom inhaltlichen Umfang des Codes. Letztere Eigenschaft ergibt sich aus dem Umstand, daß eine JavaBean Komponente wesentlich mehr Code und Funktionalität beinhaltet, solange sie sich im Prozeß der Komposition befindet. Zum Zeitpunkt des "Einfrierens" einer Bean bzw. Anwendung in die endgültige Komponentenform wird nicht mehr benötigter entwicklungszeitspezifischer Code verworfen.

Beide Schnittstellenarten werden durch Mengen sog. *APIs* (*Application Programming Interfaces*) definiert, welche eine Bean unterstützt. APIs sind dabei nach Java Terminologie einfache Java Interfaces, also im wesentlichen Sammlungen von Methodensignaturen.

- Die ***Entwicklungszeitschnittstelle*** besteht aus APIs, welche die grundlegende Inspektion einer gesamten Komponente inklusive aller interner Details sowie deren Anpassung erlauben. Die Inspektion einer Komponente gibt Aufschluß über deren zugreifbare Attribute, aufrufbare Methoden und bereitgestellte Events sowie unterstützte Event-Handler.

- Die ***Laufzeitschnittstelle*** basiert hingegen ausschließlich auf der API eines einzelnen Objekts der Komponente, welches explizit als Zugangspunkt bestimmt wurde und somit die gesamte Bean repräsentiert. Dieses Objekt definiert über seine Methoden die extern sichtbaren Komponenteneigenschaften (*Properties*) und Ereignisse (*Events*) sowie jegliche sonstige allgemein zugreifbare Funktionalität.

Zur Spezifikation der äußeren (Laufzeit-)Schnittstelle sind eine Reihe konzeptioneller *Design-Konventionen* einzuhalten, welche als *Entwurfsmuster* bzw. *Design Patterns* bezeichnet werden, jedoch im wesentlichen Namenskonventionen darstellen. Durch diese Konventionen wird die Semantik von Methoden sichtbar gemacht, welche Properties oder Events repräsentieren können bzw. einfach öffentliche Methoden im herkömmlichen Sinne darstellen. Komponentennutzer wie etwa automatische Entwicklungswerkzeuge können nun die Struktur einer Bean untersuchen, anhand der Konventionen deren inhaltliches Wesen in Form von Properties bzw. Events bestimmen und zur visuellen Manipulation bzw. Komposition aufarbeiten.

Zur Untersuchung von Beans stellt die Architektur das Konzept der *Introspektion* zur Verfügung. Dieses basiert zum einen auf dem grundlegenden Java-Mechanismus der *Reflektion*, kombiniert mit den standardisierten Design-Patterns und zum anderen auf einer expliziten Deklaration von Metainformationen über separate Klassen (*Bean Information*).

Allgemeingültigkeit Beans Komponenten sind — als normale Java Objekte — Werte erster Klasse und können somit als Parameter übergeben werden. Weiterhin können Kompositionen von Komponenten generell wieder zu neuen Beans führen. Aus diesen Tatsachen folgt die prinzipielle Mehrstufigkeit des Modells.

Bindungstechnik Beans kommunizieren über einen Ereignismechanismus mit explizit über die Namenskonvention oder BeanInfo spezifizierten *Events*. Grundsätzlich können daneben alle normalen Methoden einer beankonformen Klasse in gewohnter Weise verwendet werden, und auch Vererbung ist wie gewohnt möglich. Man kommt nun zu zwei Kompositionsformen:

- **Objektkomposition** Eine Komponente $K1$ wird mit einer Komponenten $K2$ kombiniert, indem $K1$ eine Referenz auf $K2$ gegeben wird. Dies erfolgt entweder durch das Setzen eines Property Attributes oder durch den Aufruf der Registrierungsmethode für einen Event. Komposition setzt hierbei nicht zwangsweise die Bereitstellung diesbezüglichen Codes — bekannt als "Klebstoff" oder "glue" — voraus. Allgemein sind drei *Anwendungsszenarios* vorgesehen:
  - Verwendung innerhalb von *Java Entwicklungsumgebungen*, in denen Beans durch individuell geschriebenen Code instanziiert, konfiguriert und komponiert werden.
  - Verwendung innerhalb von *skriptbasierten Umgebungen*, in denen Beans durch die Nutzung innerhalb einer Skriptsprache exportierter Methoden und Properties instanziiert und konfiguriert werden.
  - Verwendung innerhalb *grafischer Entwicklungstools*, in denen Beans durch generische oder speziell bereitgestellte Property-Editoren und Kompositions-Assistenten konfiguriert bzw. zusammengesetzt werden.
- **Klassenkomposition** Da die Abstraktionsgrenzen von JavaBeans durch Objekte repräsentiert werden, und diese wiederum Instanzen von Java Klassen sind, kann ein JavaBeans Baustein von der Grenzklasse einer beliebigen Basiskomponente durch Vererbung abgeleitet werden.

Der Kompositionsvorgang des JavaBeans Modells ist dynamischer Natur und kann fest einkompiliert werden. Derartige fest kompilierte Kompositionen bestehen im wesentlichen aus den Properties der inneren Komponenten sowie dem internen Komponentengraphen und werden in speziellen Dateien persistent abgelegt.

Interoperabilität und Verteilung Komponenten nach Beanstandard sind sprachunabhängig, solange ein Compiler existiert, welcher die Programme der spezifischen Basissprache in Java Bytecode übersetzt. In diesem Sinne gibt es einen *Binärstandard* für die Repräsentation von Beans. Die Beans an sich liegen dabei als Java *Bytecode Repräsentationen* vor und werden in Java *Class-Dateien* gespeichert, welche Unicode Namen von Methoden und Properties der Bean enthalten. Der Zustand einer Bean wird durch spezielle Dateien persistent gehalten, die ein Binärformat für serialisierte Objekte beinhalten.

Die Verteilung von Komponenten ist in der JavaBeans Architektur zwar explizit vorgesehen, wird dort aber nicht direkt festgelegt. Um dem Entwickler exklusiv die Entscheidungsfreiheit zwischen lokaler und entfernter

Verarbeitung zu überlassen, arbeiten Beans primär in lokalen Umgebungen einzelner Java Virtual Machines. Es stehen jedoch durch die Java-Basierung verschiedene alternative Mechanismen zur Interaktion mit externen Servern zur Verfügung. Die drei wichtigsten davon sind:

- **RMI** Mittels der *Java Remote Method Invocation (RMI) Facility* können auf einfache Weise verteilte Java Anwendungen entwickelt werden. Die verteilten Schnittstellen werden in Java entwickelt und dienen dann im Anschluß als Basis zur Implementation von Clients sowie Servern. Java RMI Aufrufe werden automatisch und transparent von Clients zu Servern weitergeleitet.

- **IDL** Durch *Java Interface Definition Language (IDL)* wird die *OMG CORBA* Architektur implementiert, welche einen wesentlichen Industriestandard für verteilte Objekte darstellt. Alle Schnittstellen werden in *CORBA IDL* definiert, auf deren Basis Kommunikationsrümpfe zur Interaktion entsprechender Clients und Server automatisch generiert werden können. Die Unterstützung von CORBA eröffnet JavaBeans Komponenten die Partizipation in einer offenen, multilingualen und herstellerübergreifenden, verteilten Umgebung. Umgekehrt beinhaltet die OMG CORBA Architektur in der Version 3.0 offiziell die Unterstützung von JavaBeans.

- **JDBC** Durch das *Java Database Connection (JDBC) API* ist der Zugriff auf lokale oder entfernte SQL-basierte Datenbanken möglich.

Eine weitere Möglichkeit verteilter Verarbeitung besteht durch die *Migration* der zustandsbehafteten Bean Komponenten zwischen verschiedenen Systemumgebungen — z.B. Rechnerknoten eines Netzwerkes. Diese Migrationsfunktionalität ist eine wesentliche Erweiterung zu Interoperabilitätsmechanismen wie RMI oder CORBA, welche lediglich den Zugriff auf entfernte Ressourcen erlauben. Durch die Technologie ergeben sich interessante Anwendungsmöglichkeiten z.B. im Bereich der Geschäftsprozeßmodellierung.

Nebenläufigkeit In ihrer Eigenschaft als Java Objekte können Beans innerhalb einer *Multi-Threading* Umgebung *nebenläufig* agieren. Das bedeutet, daß Methodenaufrufe bzw. Events potentiell mehrfach simultan auftreten können. In diesem Zusammenhang auftretende Konflikte liegen dabei alleine in der Verantwortlichkeit des Entwicklers.

Spezifikation/Beschreibung/Semantik Innerhalb der JavaBeans Architektur ist keine formale Spezifikation von Komponenten enthalten. Auch

die relativ verbreitete Formulierung von Vor- und Nachbedingungen mit formalen Notationen wie etwa einem Predikatenkalkül oder der Z-Notation ist nicht vorgesehen.

Die einzige Spezifikation von Beans ist durch deren Schnittstellen gegeben, welche die Properties, Events und Methoden einer Komponente offenlegt, die von außen zugreifbar sind. Properties werden dabei durch ihren Typ spezifiziert, Methoden durch Ihre Signaturen.

Verifikation formaler Korrektheit innerhalb der JavaBeans Architektur besteht im wesentlichen aus Typechecking, also dem Prüfen von Typkonformität.

In Bezug auf die inhaltliche Bedeutung der Architekturbestandteile existiert für JavaBeans keine wohldefinierte semantische Grundlage. Ebensowenig ist ein formales Verhaltensmodell verfügbar. Es besteht jedoch die Möglichkeit einer Argumentation über den Properties, denn:

- Durch den integrierten Mechanismus der Introspektion ist es möglich, die öffentlichen Properties und Methoden einer Bean eindeutig zu numerieren.
- Durch die vorgegebenen Entwurfsmuster sind Namen von Methoden mit Semantiken assoziiert. Eine Methode zur Registrierung eines *unicast event listeners* folgt zum Beispiel stets folgender Konvention:

```
aSubclassOfEventListener
   setaSubclassOfEventListener(aSubclassOfEventListener t);
```

- Durch die Signaturen sind Methoden streng typisiert, und das in der JavaBeans Architektur enthaltene Konzept der *Plug-Kompatibilität* bezüglich bestimmter Methoden, welches die Kompositionsfähigkeit ausdrückt, deckt sich mit Typkonformität.

Die JavaBeans Architektur läßt daneben viele wesentliche Aspekte unberücksichtigt. So lassen sich etwa keinerlei Aussagen über die Kompatibilität von *Kommunikationsprotokollen* zur Komposition bestimmter Komponenten treffen. Dementsprechend kann man auch die Freiheit von *Deadlocks* bei im Rahmen der Architektur erlaubten Strukturen nicht ausschließen. Des weiteren sind somit ebenfalls keine Äquivalenztransformationen und ganz generell keine Korrektheitnachweise im Rahmen des Bean Modells enthalten.

6.1.1.2 Die Anwendungsgsmethodologie

Anwendungsdomäne Die JavaBeans Architektur ist auf keine spezielle Anwendungsdomäne beschränkt. Konzeptionsseitig ist jedoch durch die starke Bindung an Java eine gewisse Richtung vorgegeben. Da sich Java zum

Inbegriff und quasi Standard für mobile, plattformübergreifende Software im Internet avanciert hat, liegt ein Schwerpunkt für Anwendungsdomänen von JavaBeans in verteilten Systemen auf Basis des World Wide Web als populärster Internetstruktur.

Softwareentwicklungsprozeß Der Entwicklungsprozeß von Software mit JavaBeans teilt sich in zwei Bereiche: zum einen Entwurf sowie Implementation von Beans und zum anderen deren Anwendung — also Komposition. Da bei der Erstellung einer Bean Teilkomponenten genutzt werden können, sind die beiden Rollen jedoch nicht strikt getrennt. Allgemein stützt sich der Prozeß auf keine spezielle Entwicklungsumgebung, sondern kann in gleichwertiger Weise durch manuelle (Java) Programmierung, Nutzung von Skriptsprachen oder visuelle Kompositionsumgebungen geschehen. In all diesen Fällen stehen die grundlegenden Unterstützungsmechanismen der Architektur zur Verfügung, welche aus Introspektion zur Analyse nutzbarer Methoden und Properties sowie den Entwurfsmustern zum Rückschluß der Semantiken bestehen.

Granularität Beans sind als Komponenten mittlerer Granularität vorgesehen. Da sie intern aus einer beliebigen Menge von Objekten und weiteren Beans bestehen können, übersteigt ihr Umfang potentiell den von herkömmlichen Java Objekten. Der Umstand, daß Beans extern durch genau ein Objekt repräsentiert werden, setzt ihrer Größe jedoch deutliche Schranken. Die JavaBean API Spezifikation charakterisiert die Granularität der Komponenten in diesem Sinne durch einen Vergleich, indem Beans näher bei COM/ActiveX Controls als bei OpenDoc Komponenten angesiedelt werden, da sie prinzipiell kein "High-End Dokumenten API" bereitstellen. Dieser Umstand ist nicht als Schwäche zu deuten, da im Java Kontext durch die *Enterprise JavaBeans* [MH99] eine explizite Lösung für komplexere Komponenten vorgesehen ist.

Anpassungsfähigkeit Komponenten des JavaBean Standards können zur Entwicklungszeit und in geringerem Umfang auch zur Laufzeit, immer jedoch in explizit vorgesehenem Rahmen, individuell angepaßt werden. Dieser Rahmen wird durch *BeanProperties* (private Attribute mit entsprechenden Zugriffsmethoden) bestimmt, welche mit Hilfe einfacher *Property-Editoren* oder individuell gesteuert durch komplexe, mitgelieferte *Customizer* manipuliert werden können.

Distribution und Retrieval Die JavaBeans Architektur selber enthält kein eigenes Repository-Format, und auch Methoden zum standardisierten — wünschenswerter Weise verteilten — Retrieval benötigter Bausteine von

Komponentenherstellern fehlen. Der einzige diesbezügliche Mechanismus ist durch die Verwendung eines speziellen Archiv Formats für Dateien gegeben, der als *Java Archive Format (JAR)* bezeichnet wird. Vorkonfigurierte Komponenten können so in ihrer individuellen Form langlebig gespeichert und später wieder- bzw. weiterverwendet werden, wobei die Persistenz auf der in Java enthaltenen *Objektserialisierung* beruht. Im einzelnen bestehen die Archivdateien aus dem Verzeichnis aller in der Datei enthaltenen Klassen, den serialisierten Objekten und sonstiger Ressourcen. Daneben können — optional — noch folgende Bestandteile hinzukommen:

- Ein angepaßter Katalog *lokalisierter Nachrichten*.
- *Dokumentation* der Inhalte in HTML-Format.
- Zusätzliche *entwurfsspezifische (Meta-)Komponenten* wie Property-Editoren oder Kompositions-Assistenten sowie die zugehörigen Laufzeitinformationen.
- Zusatzinformationen — z.B. Versionsnummern — die durch Blöcke von *Named-Value* Paaren in RFC822 Notation repräsentiert werden.

Fazit Die JavaBeans Architektur stellt einen interessanten Ansatz der Kategorie plattformübergreifender Komponentensysteme dar. Die Basis dafür liegt in vielversprechenden Eigenschaften der zugrundeliegenden Java Technologie. Wesentlich ist dabei das Konzept der abstrakten Code-Repräsentationen, welche in beliebigen Implementationen der standardisierten *Java Virtual Machine* ablauffähig sind und dabei noch einem pragmatischen Sicherheitsmechanismus unterliegen. Durch die verschiedenen heterogenen Komponentensysteme der Marktführer im Internetbereich, wie Microsoft mit *"ActiveX"*oder Netscape mit *"LiveConnect"*, entsteht gerade in diesem Sektor das Problem eines fehlenden Komponentenmodells, welches alle Plattformen gleichermaßen unterstützt. In diesem Bereich der Internetanwendungen liegt das wohl aussichtsreichste Anwendungsfeld für die Etablierung von JavaBeans.

6.1.2 Analyse und Bewertung

Die vorangegangenen Untersektionen gaben eine erweiterte Übersicht der JavaBeans Architektur. Hier soll nun eine kurze Analyse und Bewertung ihrer wesentlichen Eigenschaften in Bezug auf die Ergebnisse der vorangegangenen Kapitel vorgenommen werden, welche sich mit allgemeingültigen Konzepten von Komponentenarchitekturen in abstrakter Form beschäftigten.

Zunächst stellt sich die Frage, inwiefern JavaBeans dem in Kapitel 4.1.2 formulierten Anforderungskatalog genügen. Hierzu kann man zunächst feststellen, daß die Beans Architektur einen Großteil dieser Anforderungen erfüllt. Einige wesentliche Aspekte sind jedoch nicht gegeben.

Die wesentliche Eingangsforderung nach *multidimensionaler (De-) Komposition* wird zwar zunächst im Hinblick auf das Fehlen einer mehrdimensionalen Aspektsicht nur halb, aber dennoch zum wesentlichen Teil erfüllt. Des weiteren genügt die JavaBeans Architektur den Punkten *Geschlossenheit*, *Anpaßbarkeit bzw. Adaptability*, *Zustand und Identität*, *Offenheit* sowie *Nebenläufigkeit* in vollem Umfang.

Die weniger gut erfüllten Forderungen lassen sich grob in zwei Gruppen einteilen. Die eine betrifft *Kapselung und Abstraktion* die andere *Formalisierung und Spezifikation*. In die erste Kategorie fallen Forderungen nach *Kontextunabhängigkeit* und *Kapselung*, welche zusammengefaßt folgendes beinhalten:

> Komponenten sollen für sich geschlossene Abstraktionen darstellen und eine klare Grenze besitzen. Zugangs- bzw. Berührungspunkte mit der Umgebung sollen über explizit spezifizierte Schnittstellen eindeutig festgelegt und publiziert werden. Neben diesen veröffentlichten Bezugspunkten sollen Komponenten gänzlich unabhängig vom Kontext ihrer Verwendung sein.

Beans besitzen mit dem sie nach außen repräsentierenden einzelnen Objekt zwar eine klare Grenze, diese ist jedoch schwammig und genügt den Anforderungen nicht. Sie verfügen dabei über genau eine operationale Exportschnittstelle. Diese liegt nicht in expliziter Form vor, sondern ergibt sich indirekt aus der Klasse des Objektes bzw. aus den Unicode-Namen der Methoden im Bytecode. Sie ist zwar durch Introspektion dynamisch extrahierbar — was eine durchaus positive Eigenschaft ist —, das Ergebnis wird jedoch durch Entwurfsmuster und andere Filter verfälscht. Noch weitaus schwerwiegender kommt hinzu, daß keine vorausgesetzte Funktionalität in Form einer Importschnittstelle existiert. Es ist daher nicht eindeutig ersichtlich, auf welche externen Ressourcen der Umgebung sich eine Komponente stützt. Aus diesem Grunde sind Beans nicht kontextunabhängig.

Die zweite Kategorie umfaßt die Forderungen nach *Formalisierung*, *erweiterten Spezifikationsmöglichkeiten* und indirekt *Substituierbarkeit*. Die Kategorie läßt sich inhaltlich wie folgt zusammenfassen:

> Komponenten sollen eine formale Spezifikation besitzen. Diese soll neben strukturellen Eigenschaften die Beschreibung qualitativer Aspekte und dynamischer Verhaltensmuster sowie der

> Semantik von Komponenten erlauben. Diese formale Grundlage soll u.a. Korrektheits- bzw. Konsistenzbeweise erlauben und als Basis für Beziehungen wie Kompatibilität und Äquivalenz dienen, wodurch etwa Substitution gleichwertiger Komponenten eines Softwaresystems möglich wird.

Wie gesehen, sind die Spezifikationsmöglichkeiten von Beans äußerst dürftig, entbehren jeglicher formaler Grundlage, und definierte Semantik ist somit ebenfalls abwesend. In diesem Bereich ist lediglich die durch den Java-Ursprung übernommene strenge Typisierung zu nennen. Durch die Typsicherheit ist zumindest ein gewisses Maß struktureller Kontrolle von Kompositionsbeziehungen gewährleistet.

Bezüglich der Modellebene der JavaBeans Architektur fällt die nahezu uneingeschränkte Möglichkeit wechselseitiger Verknüpfungen und Interaktionsmuster auf, welche durch den relativ direkten, objektorientierten Ursprung ohne inhaltliche Modifikationen bedingt sind. Dieses Maß an Allgemeingültigkeit läßt für ein Komponentenparadigma ungewöhnlich viele Freiheiten. Zunächst ist der Kompositionsbegriff gar nicht ausdrücklich definiert. Von Aggregation durch Objektreferenzen bis Refinement durch Vererbung ist der Charakter wechselseitiger Beziehungen von Komponenten offen. Demzufolge bestehen keine eindeutig gerichteten Kompositionsbeziehungen. Letztere sind zudem für jede Bean in beliebiger Anzahl zulässig. Genausowenig wie die Struktur von Komponentensystemen ist deren Verhalten regulativen Beschränkungen unterworfen. Dies gilt für internes und externes Verhalten: Sowohl die Frage nach Interaktionsmustern zwischen Beans — d.h. welche Bean einer Struktur die Initiative ergreift, also Methoden einer anderen Bean aufruft — wie auch die nach temporalen Aspekten bzw. kausalen Zusammenhängen der internen Einheiten einer Komponente — also die Reihenfolge von Methodenaufrufen — bleibt offen.

Eine von Komponentensystemen erwartete Dogmatik mit ordentlicher, geregelter Struktur und dezidiertem Verhalten ergibt sich durch die dargelegten Sachverhalte nicht auf implizite Weise. Es liegt vielmehr der typische Tatbestand objektorientierter Ansätze vor, die "im Prinzip" auch zu strukturierterer und auf vielfache Weise verbesserter Software führen können.

6.2 Modifikationsansätze

Nachdem die JavaBeans Architektur vorgestellt und analysiert wurde, sind bezüglich der im Verlauf dieser Arbeit als günstig erkannten Qualitäten allgemeiner komponentenorientierter Systeme, verschiedene Schwachstellen der Bean-Technologie zutage getreten. Nachfolgend werden nun einige exemplarische Ansätze für optimierende Modifikationen des JavaBeans-Modells besprochen.

Die Sektion ist nach Ansätzen aufgeteilt, die zum einen aus konzeptionellen und inhaltlichen Änderungen der JavaBeans Architektur selber und zum anderen aus separaten Erweiterungen in Form systemtechnischer Unterstützungsmechanismen bestehen.

6.2.1 Interne Variation

In Abschnitt 6.1.2 wurden mehrere für komponentenorientierte Systeme problematische Aspekte aufgetan. Zunächst wurden dabei Schwachstellen im Bereich *Kapselung*und *Abstraktion* identifiziert. Diese sind, wie dargestellt, größtenteils auf den Schnittstellenbegriff des Bean-Modells zurückzuführen.

Als erstes fällt auf, daß bei Beans, gegeben durch repräsentierende Klassen, Schnittstellen als solche gar nicht vorliegen. Ein Beispiel soll dies verdeutlichen: In Rückgriff auf das mittlerweile bekannte Beispiel der Produktionsmaschine wäre deren Schnittstelle aus der Java-Klasse des Bean-Objektes abzuleiten, was etwa wie folgt aussehen könnte:

```
import java.io.Serializable;
public class MaschinenBean implements
   Serializable, StartListener, StopListener {
       private boolean wartung;
       public MaschinenBean() {
              this.wartung = false;
       }
       public boolean getWartung() {
              return this.wartung;
       }
       public void setWartung( boolean w ) { }
       public void startAngeordnet( StartEvent start ) {...}
       public void stopAngeordnet( StopEvent stop ) {...}
       ...
              ...
              MaschinenleitstandBean leitstand;
              leitstand = ( MaschinenleitstandBean )
                          java.beans.Beans.instantiate(
                                   'MeinMaschinenleitstand' );
              ...
              leitstand.addStartListener(this);
              leitstand.addStopListener(this);
              ...
       ...
}
```

Abbildung 6.1: Schematische Klassendefinition einer JavaBean Komponente

Das Beispiel in Abb. 6.1 verdeutlicht zunächst die wenig anschauliche Dar-

stellungsform der Schnittstelle. Diese ergibt sich indirekt aus den über die gesamte Klasse verteilten Deklarationen und Signaturen. Eine Verbesserung dieses Umstandes läßt sich durch die Verwendung des *Java Interface Konzepts* erreichen [Gri98]. Diese Interfaces abstrahieren vom Implementationscode und enthalten daher lediglich Signaturen und Deklarationen, was zu einer weitaus anschaulicheren Form führt. Das entsprechende Java Interface der `MaschienenBean` ist in Abb. 6.2 abgebildet.

```
public interface MaschinenBean_If extends
   Serializable, StartListener, StopListener {
       private boolean wartung;
       public boolean getWartung();
       public void setWartung( boolean w );
       public void startAngeordnet( StartEvent start );
       public void stopAngeordnet( StopEvent stop );
}
```

Abbildung 6.2: Das Java Interface der `MaschinenBean`

Wie schon die Standardform der indirekten Bean-Schnittstelle ist auch das Java Interface Konzept ein sprachabhängiger Ansatz, welcher naturgemäß in einem gewissen Zielkonflikt mit der Forderung nach Offenheit von Komponentenarchitekturen steht. Eine interessante Alternative zu Interfaces ist daher durch Nutzung von *Java-IDL* gegeben, welche ab JDK-Version 1.2 fester Bestandteil der Java Technologie ist. Auf diese Weise ist die Schnittstellenspezifikation nicht nur in klarer, expliziter Form, sondern darüber hinaus auch sprachunabhängig möglich. Als weitere positive Eigenschaft stellt IDL einen sanften Übergang zwischen Modellierung und Implementation dar [Gri98], denn die abstrakte Schnittstellenbeschreibung ist konkret genug, um aus ihr — mit Hilfe eines sog. *IDL-Compilers* — automatisch Teile des Implementationscodes zu generieren. Abbildung 6.3 zeigt die entsprechende IDL-Spezifikation für die Schnittstelle der `Maschinenbean`:

Eine weitere Problematik der Bean Architektur ergibt sich durch den Umstand, daß externe Abhängigkeiten nicht aus dem als Schnittstelle identifizierten Rahmen ersichtlich sind. Zur Veranschaulichung soll wiederum das Maschinenbeispiel herangezogen werden. Die Maschinenkomponente ist zur Integration in eine Komponentenumgebung vorgesehen, in der sie durch Komposition mit einem Partnerbaustein zusammengesetzt wird, der Start- bzw. Stop-Nachrichten an sie sendet. Letzterer Part soll im Beispielszenario von einer als `MaschinenleitstandBean` bezeichneten Komponente übernommen werden. Die besagten Nachrichten werden dabei in der JavaBeans Implementation als speziell definierte Ereignisse `StartEvent` und `StopEvent` modelliert. Eine Komposition

```
#include ‘‘StartListener.idl’’
#include ‘‘StopListener.idl’’
module Produktionssystem_Idl_Mod
{
   interface MaschinenBean_Idl_If : StartListener, StopListener
   {
      attribute boolean wartung;
      boolean getWartung();
      void setWartung(in boolean w );
      void startAngeordnet(in StartEvent start );
      void stopAngeordnet(in StopEvent stop );
   };
};
```

Abbildung 6.3: Die `MaschinenBean` Schnittstelle in CORBA IDL

der Beans wird dann durch Registrierung der `MaschinenBean` als Ereignisempfänger der Ereignisquelle `MaschinenleitstandBean` mittels derer `addStartEvent` bzw. `addStopEvent` Methoden erreicht. Dieser Kompositionsvorgang, sowie vor allem die dabei verwendete Referenzierung eines konkreten `MaschinenleitstandBean` Objektes sind an beliebiger Stelle innerhalb des Implementationcodes verborgen (Abb. 6.1). In der abgeleiteten Schnittstelle entfallen sie dann restlos (Abb. 6.2).

Es ist also das Ziel anzustreben, externe Abhängigkeiten in Form verwendeter Objektreferenzen klar ersichtlich zu machen. Ein simpler Ansatz, der diesbezüglich Besserung verspricht, ist durch Definition entsprechender Properties möglich, welche eben die gewünschten Referenzen beinhalten (Abb. 6.4).

```
public interface MaschinenBeanIf extends
   Serializable, StartListener, StopListener {
       private boolean wartung;
       private MaschinenleitstandBean leitstand;
       public MaschinenLeitstandBean getLeitstand();
       public void setLeitstand(
              MaschinenleitstandBean leitstand );
       public boolean getWartung();
       public void setWartung( boolean w );
       public void startAngeordnet( StartEvent start );
       public void stopAngeordnet( StopEvent stop );
}
```

Abbildung 6.4: Das verbesserte `MaschinenBean` Interface

Die verbleibende Problematik fehlender formaler Spezifikation ist fundamentaler und weitreichender als die vorherigen Aspekte und läßt sich demzufol-

ge nicht in befriedigender Form durch einfache konzeptionelle Richtlinien lösen. Es soll trotzdem ein Ansatz dieser Kategorie vorgestellt werden. Dessen Basis ist durch die temporale Verhaltensspezifikation mittels PLT-Logik gegeben, welche den zeitlichen Aspekt von Komponenten beschreibt.

Prinzipiell bietet Java hier die passenden Abstraktionen: Der Eventmechanismus liefert die atomaren Interaktionseinheiten, über denen die logischen Spezifikationen gebildet werden können. Properties dienen dabei der optionalen Kennzeichnung von Komponentenzuständen. Es stellt sich nur die Frage, wie derartige formale Spezifikationen in das Komponentensystem integriert werden können.

Eine Möglichkeit dies zu erreichen, ist durch die Erweiterung der bestehenden Komponententechnologie um geeignete Konstrukte, welche die Konzepte des formalen Apparats direkt unterstützen, gegeben. Zu diesem Punkt schlägt Nierstrasz in [Nie95] die Verwendung eines *Typsystems* vor. Dort wird ein Typsystem für objektorientierte Sprachen vorgeschlagen, das es den Benutzern erlaubt, zeitliche Aspekte des Objektverhaltens zu beschreiben. Ferner werden Regeln hergeleitet, um die Typsicherheit solcher Beschreibungen zu prüfen. Obwohl der Ansatz von einem anderen Formalismus ausgeht, scheinen sich die meisten Ideen und Ergebnisse auch für den besagten PLTL-basierten Ansatz zu eignen [Ara95]. Das vorgeschlagene Typsystem könnte in diesem Sinne als Ansatzpunkt dienen, die wesentlichen Konzepte der PLTL-basierten Verhaltensbeschreibungen in die JavaBeans Architektur zu integrieren.

6.2.2 Externe Erweiterung

Zur Modifikation eines Komponentensystems wie JavaBeans muß dieses nicht zwangsläufig intern abgeändert werden. Dem bestehenden System können auch Erweiterungen durch externe *Unterstützungsmechanismen* aufgesetzt werden. In diesem Sinne sollen im folgenden zwei Ansätze externer Ausweitung vorgestellt werden. Diese Ansätze sind dabei nicht streng auf JavaBeans beschränkt, sondern durch die Offenheit und Generik der beteiligten Mechanismen — welche dem Kontext verteilter Systeme entstammen — auch für andere konkrete Komponentenarchitekturen denkbar.

6.2.2.1 Eine dienstbasierte Metaebene

Ein wesentliches Manko der JavaBeans Architektur ist das Fehlen formaler Spezifikationsmöglichkeiten, insbesondere im Bereich der Semantik. Als möglicher Verbesserungsansatz bietet sich die Verwendung eines erweiterten Typsystems an, welches die Konzepte einer PLTL-basierten Verhaltensbeschreibung in das Komponentensystem integriert. Ein solches Typsystem

kann auch als übergeordnete, abstrakte Metaebene auf bestehende Systeme aufgesetzt werden. In diesem Sinne ist es denkbar, das im Bereich der verteilten Systeme bewährte Modell der *Dienstabstraktion*[1] im Kontext komponentenorientierter Architekturen wie JavaBeans anzuwenden.

Dienstmodellierung [MJ96] stellt ein erfolgreiches Konzept verteilter Systeme dar, das auf der Abstraktion von Ressourcen einer globalen Umgebung bzw. der sie verwaltenden Manager [CDK94] als *Dienste* [Jon94] mit entsprechender Kategorisierung durch *Diensttypen* [CM95] beruht. Diensttypen, welche im Kern auf abstrakten Schnittstellenbeschreibungen beruhen, können zur besseren Differenzierung verschiedener Instanzen sog. *Dienstattribute* definieren, die der einfachen semantischen Spezifikation dienen.

Der Ansatz besteht nun in einer Abstraktion von Komponenten durch Dienste, kombiniert mit einer Erweiterung des Dienstmodells, bei der komplexere PLTL-Spezifikationen an die Stelle der einfachen Dienstattribute treten. Das auf diese Weise um eine Metaebene bereicherte Komponentenmodell könnte nun auf verschiedene Techniken zurückgreifen, die zur konkreten Anwendung der Dienstsicht entwickelt wurden.

Komponentenorientierte Dienstnutzung Dienstnutzung unterteilt sich im allgemeinen Fall grob in die drei Bereiche *Diensttypfindung*, *Dienstauswahl* und *Dienstzugriff*. Im Sinne der ursprünglichen JavaBeans Spezifikation sollen aber die im Dienstkonzept enthaltenen dynamischen Aspekte einer späten variablen Dienstselektion zur Laufzeit nicht auf das Komponentenmodell übertragen werden. Ein neutralerer Ansatzpunkt ist durch den statischen Gebrauch von Diensttypfindung sowie Dienstauswahl bei Entwurf und Entwicklung komponentenorientierter Systeme gegeben.

Erster Schritt der komponentenorientierten Dienstnutzung ist die Auswahl geeigneter Diensttypen, also abstrakter Umschreibungen der gewünschten Komponenten. Die Festlegung von Diensttypen geschieht auf manueller Basis durch Programmierer, die in der Entwicklungsphase komponentenorientierter Anwendungen geeignete Bausteine mit spezifischen, durch den Typ beschriebenen Eigenschaften suchen, wobei auch die Berücksichtigung von Verhaltensmustern möglich sein soll. Eine Unterstützung in dieser Phase kann durch einen *Browser* zur Visualisierung von Typhierarchien erfolgen, der seine Informationen von einem *Typmanager* bezieht, dessen Aufgabe die Verwaltung von Typen und Beziehungen ist.

Inhalt des zweiten Schritts ist die Auswahl konkreter Bausteine, die den Spezifikationen des gewünschten oder eines dazu konformen Diensttyps

[1] Eine konkrete Dienstarchitektur wurde am Fachbereich Informatik der Universität Hamburg in den Projekten *TRADE*, *BERTANGRAM* und *DYNAMICS* des Arbeitsbereichs "Verteilte Systeme" (VSYS) entworfen und implementiert. Nähere Informationen finden sich unter `http://vsys-www.informatik.uni-hamburg.de/`

genügen, was auch als *(diensttypbasiertes) Trading* bezeichnet wird. Eine Beschränkung des möglicherweise komplexen Angebotes kann durch Angabe gewünschter Ausprägungen von Dienstattributen geschehen. Solche Dienstattribute können auch zur Spezifikation semantischer Verhaltensbeschreibung dienen. Bei der Suche nach Komponenten sind dabei zwei Arten von Verhaltensattributen zu unterscheiden. Entweder wird das konkret gewünschte Verhalten einer Komponente vorgegeben, oder es werden solche Verhaltensmuster spezifiziert, mit denen die gesuchten Komponenten kooperieren sollen und daher verträglich sein müssen.

Erfolgte in Schritt zwei eine sukzessive Auswahl der jeweils als nächstes benötigten Komponenten unter Angabe der bekannten temporalen Bedingungen vorgesehener Kompositionspartner, so wurden diese Bedingungen bei der Suche passender Bausteine berücksichtigt, und die Konsistenz der entstandenen Struktur ist gewährleistet. Ansonsten müssen die Kompositionsbeziehungen des Komponentensystems in einem anschließenden Schritt der Typverifikation durch den Compiler, Generator oder ein spezielles Werkzeug nachträglich überprüft werden. Bei dieser Aufgabe kann wiederum ein Typmanager helfen, der die Diensttypen beteiligter Bausteine auf Kompatibilität prüft, indem er deren um temporallogische Ausdrücke erweiterten Attribute abgleicht.

Systemtechnische Unterstützungsmechanismen Konkrete komponentenorientierte Dienstnutzung bedingt nun eine Umsetzung der oben angesprochenen modellhaften Abstraktion des Dienstbegriffs und der damit verbundenen Konzepte auf die Gegebenheiten realer Systeme. Hierzu werden eine Reihe unterstützender, systemtechnischer Mechanismen verwendet, deren grundlegende Bestandteile *Trader* und *Typmanager* sind.

Trader Durch *Trader* existieren Mechanismen, die einer Strukturierung des Komponentenangebotes dienen können und es Entwicklern ermöglichen, anhand der von ihnen spezifizierten Diensttypen Bausteine gleichen bzw. konformen Typs aufzufinden und auszuwählen. Neben dem rein strukturellen Diensttyp können zum Auffinden von Komponenten auch Dienstattribute herangezogen werden. Dienstattribute ermöglichen eine Charakterisierung von Komponenten über strukturelle Aspekte hinaus auf semantischer Ebene. Mögliche Dienstattributtypen sind Bestandteil der Diensttypen. Konkrete Bausteine eines Diensttyps können sich nun in den Ausprägungen der Dienstattribute unterscheiden und ermöglichen so eine weitere Eingrenzung der Auswahl. Durch Erweiterung der normalerweise simplen Attribute zu komplexen temporallogischen Formeln könnten auch kausale Anwendungsaspekte berücksichtigt werden.

Da Komponenten eines Diensttyps, der zu dem eigentlich gewünschten kon-

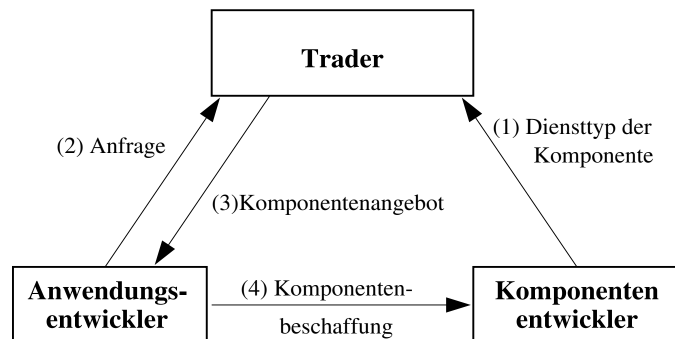


Abbildung 6.5: Component-Trading

form ist, in gleicher Weise wie die des eigentlichen Typs genutzt werden können, muß der Trader auch solche kompatiblen Bausteine bei seiner Suche berücksichtigen. Im komponentenorientierten Fall ist darüber hinaus insbesondere auch die Kooperationsfähigkeit bzw. Interoperabilität von Typen wichtig. In beiden Fällen benötigt der Trader Informationen über Typbeziehungen, die von einem *Typmanager* bereitgestellt werden.

Typmanager Die Aufgabe eines Typmanagers ist die Verwaltung von Typen und Beziehungen zwischen Typen. Das Typmanagement ist hier als eine anwendungsunabhängige Systemfunktion zu sehen, die von den einzelnen Bestandteilen eines komponentenorientierten Systems zur spezifischen Problemlösung eingesetzt werden kann.

Die Nutzung des Typmanagements beginnt bei der Diensttypfindung. Hier gewähren interaktive Browser dem Entwickler direkten Zugriff auf die vom Typmanager verwaltete Typhierarchie, um eine Auswahl geeigneter Diensttypen treffen zu können. Bei der Komponentenauswahl spielt das Typmanagement erneut eine wichtige Rolle, da das hier verwendete diensttypbasierte Trading auf der Konformität und Interoperabilität von Diensttypen beruht. Typmanager geben hier dem Trader Auskunft über die Typen der von ihm verwalteten Komponenten sowie deren Beziehungen. Im letzten Schritt der Typverifikation, in dem Kompositionsbeziehungen auf Konsistenz geprüft werden, liefert der Typmanager Informationen über die Interoperabilität von Diensttypen beteiligter Bausteine an die entsprechenden Tools einer Entwicklungsumgebung.

Eine wesentliche Aufgabe von Typmanagern ist die Verwaltung von Beziehungen zwischen Typen, und zwar in Bezug auf dessen komponentenorientierte Anwendung vor allem von Konformitäts- und Interoperabilitätsbeziehungen zwischen den angepaßten Diensttypen. Da diese verschiedenen kontextabhängigen Faktoren unterliegen, muß eine Typverwaltung prinzipiell beliebige Beziehungen zwischen Typen verwalten können. Zudem sollte

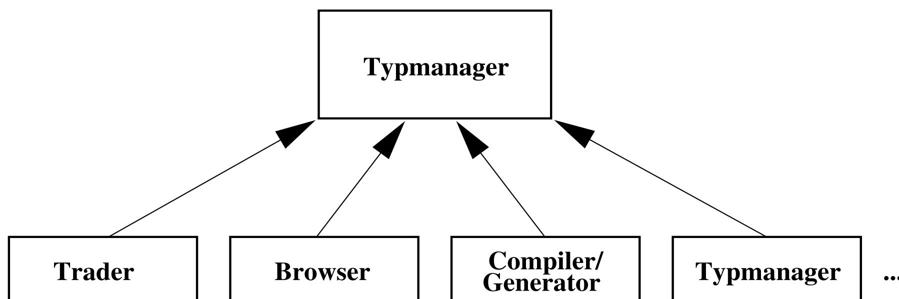


Abbildung 6.6: Klienten eines Typmanagers

die Evaluierung so weit wie möglich auf automatischer Basis erfolgen, da eine manuelle Vorgehensweise auf Grund der quantitativen Verhältnisse nur in begrenztem Rahmen als praktikabel erscheint. Gerade in dieser Hinsicht deuteten die Ergebnisse des letzten Kapitels aber erhebliche Einschränkungen an. Trotzdem erscheint der Ansatz in einem definierten Rahmen — etwa einer Anwendungsdomäne oder Entwicklungsumgebung — als sinnvoll. Unter diesen Bedingungen sind die dargelegten positiven Effekte zu erwarten.

6.2.2.2 Generische Richtlinien

Mechanismen zur Verwaltung generische Richtlinien [TGML97] stellen eine relativ neue Form der Unterstützung interaktionsbasierter und dabei vor allem verteilter Systeme dar. Ausgangspunkt sind ganz allgemeine Anforderungen, die von den Teilnehmern an die Art der Kooperation gestellt werden. Bei einer Anwendung dieses Konzeptes auf komponentenorientierte Systeme sind im Kontext dieser Arbeit vor allem erweiterte Anforderungen in Form von semantische Verhaltensmuster interessant.

Als Instrument zur Formulierung von Anforderungen kommen generische Richtlinien (engl. *Policies*) zum Einsatz. Policies sind dabei Spezifikationen, die Verhaltensweisen und Eigenschaften von allgemeinen Objekten — also auch Komponenten — beschreiben. Policies werden durch logischen Formeln verwandte Konstrukte beschrieben, deren wesentliche Bestandteile *Properties* sind. Bei diesen Properties handelt es sich um konkrete Objekteigenschaften, die separat innerhalb eines *Propertymanagers* [Sch97] verwaltet werden. Es bietet sich nun eine Erweiterung von Policies um temporallogische Konzepte an. Dadurch könnten Policies kausale Aussagen über Properties berücksichtigen.

Ein als *Policymanager* [Gö97] bezeichneter Mechanismus verwaltet nun die Richtlinien und ist in der Lage, sie automatisch zu evaluieren. Ferner können durch das *Policymanagement* die Aussagen verschiedener Richtlinien in Einklang gebracht und in einer neu generierten Policy zusammengefaßt werden.

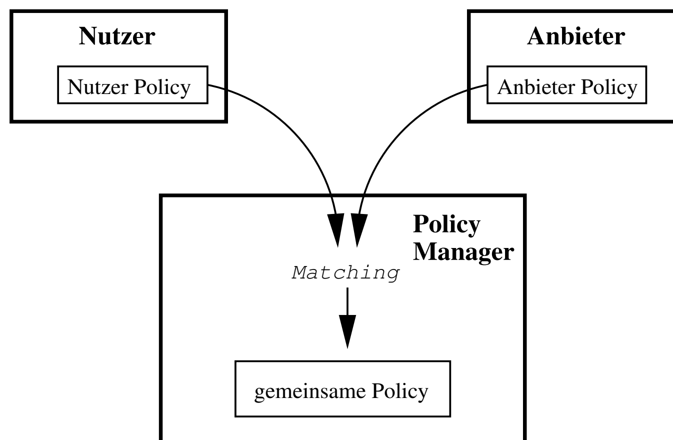


Abbildung 6.7: Prinzip des Policymanagements

Eine umfangreiche Untersuchung der Integration von Policymanagement in Komponentensysteme findet sich in [Tet99].

6.3 Fazit

Inhalt des vorliegenden Kapitels war eine exemplarische Untersuchung der Umsetzungsmöglichkeiten allgemeiner komponentenorientierter Konzepte dieser Arbeit innerhalb konkret existierender Systeme. Als Beispiel wurde die weit verbreitete JavaBeans Technologie gewählt, da diese durch ihren einfachen Aufbau relativ flexibel ist und — trotz ihres noch recht rudimentären Characters — gerade in verteilten Umgebungen wie dem Internet einen quasi Standard darstellt.

Bei der konzeptionellen Analyse wurden verschiedene Schwächen der Architektur, insbesondere in den Bereichen *Kapselung* und *formale Spezifikation*, aufgedeckt. Diese ließen sich im wesentlichen auf die Allgemeinheit des Modells zurückführen, welches einen konzeptionellen Zusatz der Java Technik darstellt. Infolge dieser Tatsache resultieren die Nachteile zum einen aus dem hohen Freiheitsgrad des Komponentenmodells sowie zum anderen aus fehlenden Erweiterungen.

Zur Kompensation dieser Nachteile wurden exemplarisch einerseits striktere interne Variationen des Komponentenmodells sowie andereseits externe Erweiterungen durch systemtechnische Unterstützungsmechanismen vorgeschlagen. Es zeigte sich, daß schon durch einfache Richtlinien, wie der grundsätzlichen Verwendung von Interfaces oder der expliziten Handhabung externer Referenzen durch Properties, Verbesserungen zu erreichen sind. Weiterhin wurde deutlich, daß externe Erweiterungen der Architektur —

etwa in Form von Konzepten einer Dienstsicht oder generischer Richtlinienverarbeitung — potentiell zu deren Verbesserung in verschiedenen Bereichen beitragen könnten. Neben einer Ausweitung der Spezifikationsmöglichkeiten kann der gesamte Entwicklungsprozeß komponentenorientierter Anwendungen in direkter Weise profitieren.

Insbesondere zeigen die Ergebnisse, daß in konkreten Architekturen durchaus Ansatzpunkte für die Integration der in den vorherigen Kapiteln beschriebenen formalen Spezifikationsformen existieren. Es müssen jedoch die in Bezug auf breite Anwendbarkeit der Konzepte geäußerten Bedenken berücksichtigt werden. Die dort erkannte und bemängelte Notwendigkeit expliziter Standardisierung identifizierender Bezeichner für grundlegende semantische Einheiten führt aber nicht in jedem Kontext zum Scheitern. Im Rahmen einer begrenzten Umgebung — wie etwa spezifischer Anwendungsdomänen — sind die Konzepte potentiell tragbar und können dort zu den beschriebenen Vorteilen effektiverer Entwicklung von zuverlässigeren Systemen führen.

Kapitel 7

Schlußbetrachtung

Das letzte Kapitel enthält zunächst einen zusammenfassenden Überblick der wesentlichen Inhalte dieser Arbeit. Ferner wird ein Resumee der Resultate mit den daraus gewonnenen Erkenntnissen gezogen. Den Abschluß bildet eine Darstellung von Ansatzpunkten weiterführender Untersuchungen.

7.1 Zusammenfassung

Das Konzept der *Komponentenorientierung* stellt bezüglich Anwendung und Entwicklung moderner Softwaresysteme ein dringliches Thema der Informatik dar. Dahinter steht, besonders in Bezug auf das Ausbleiben der erhofften Wirkungen objektorientierter Softwaretechniken, die akute Gefahr einer *Softwarekrise*. Neben dieser ernstzunehmenden Problematik fungieren neue Tendenzen wie der *Internetboom*, der anhaltende *Verteilungstrend* und die Verbreitung komplexer, visuell basierter Objektmodelle hohen Abstraktionsniveaus wie *Business-* oder *Enterprise-Objects* als Wegbereiter des aktuellen Trends der *Componentware*. Die Erschließung der Materie befindet sich allerdings insbesondere im Hinblick auf wissenschaftliche Aspekte noch in einem frühen, rudimentären Stadium. Die vorliegende Arbeit gibt in diesem Sinne einen breiten Überblick der Thematik komponentenorientierter Systeme und extrahiert die wesentlichen Prinzipien und Konzepte auf fundamentaler Ebene. Die Intention lag dabei vornehmlich in einer Bestandsaufnahme der Komponententechnik im Forschungsbereich.

Im Anschluß an die Einführung und inhaltliche Motivation der Thematik befaßte sich die Arbeit im zweiten Kapitel zunächst mit der Zielsetzung, die konzeptionelle Bandbreite des Komponentenparadigmas auszuloten. Zu diesem Zweck wurden wesentliche aktuelle Sichten des Genres anhand ihrer informalen Komponentendefinitionen gegenübergestellt, analysiert und in ihrer Essenz zusammengefaßt. Als Resultat ergab sich die Charakterisie-

rung eines im weiteren Verlauf zugrundegelegten allgemeinen Komponentenbegriffs. In der Folge wurden dessen inhaltliche Aspekte in Bezug auf Technik bzw. Methodologie identifiziert und ausführlich dargestellt.

Um das Komponentenparadigma anschaulicher zu machen und Hinweise auf dessen allgemeine Eigenschaften zu sammeln, beinhaltete das dritte Kapitel den Versuch, die große Menge diesbezüglicher praktischer Ansätze und Systeme möglichst umfassend wiederzugeben. Es fand dabei eine Aufgliederung nach Kategorien statt, zu deren charakteristischen Darstellung eine Vielzahl typischer Beispiele herangezogen wurden.

Nach der ausführlichen Betrachtung verschiedenartiger komponentenorientierter Sichten sowie deren vielfältiger Aspekte, Ansätze und praktischer Ausprägungen, lag die Zielsetzung des vierte Kapitel in der Ableitung allgemeiner, grundlegender Merkmale des Komponentenparadigmas. Zu deren systematischen Erfassung wurde die Nutzung eines Architekturbegriffs motiviert, der mit jeweils einer Modell-, Beschreibungs- und Repräsentationsebene verschiedene abgegrenzte Abstraktionsgrade beinhaltet. In diesem Sinne erfolgte die Untersuchung allgemeiner Anforderungen, Einflußgrößen und Inhalte einer abstrakten Komponentenarchitektur für dedizierte Bereiche. Die Schwerpunkte lagen dabei vor allem auf strukturellen Aspekten der Modellebene und Gesichtspunkten formaler Spezifikation unter besonderer Berücksichtigung von Semantik.

Im Hinblick auf die identifizierten Merkmale abstrakter Komponentenarchitekturen wurden im fünften Kapitel für einige ausgewählte Probleme der Bereiche Aggregation und Semantik konzeptionelle Lösungen erarbeitet. Hierbei wurde als erstes der exemplarische Entwurf eines konkreten, aber sehr allgemeinen Komponentenmodells vorgenommen. Es folgte eine Darstellung zweier logikbasierter Lösungsansätze zur formalen Spezifikation semantischer Komponenteneigenschaften. Diese bezogen sich zum einen auf die Beschreibung kausaler Verhaltensaspekte durch Temporallogik und zum anderen auf flexible, dynamische Strukturbeschreibungen mittels Feature-Logik.

Um abschließend einen Praxisbezug herzustellen, wurden die gefundenen Prinzipien auf eine konkrete Technik angewendet, wobei die Wahl auf *JavaBeans* als eine der bekanntesten Komponentenarchitekturen fiel. Es erfolgte dabei eine Analyse in Bezug auf die als wesentlich erkannten Kriterien. Ferner wurden Ansätze aufgezeigt, die gefundenen konzeptionellen Lösungen umzusetzen.

7.1.1 Ergebnisse

Die Arbeit machte zunächst deutlich, daß trotz verwirrender Komplexität und inhaltlicher Vielfalt des Komponentenparadigmas grundlegende Merk-

male identifiziert werden können, welche die Voraussetzung zur Ausbildung einer geordneten Basis bilden.

Im Rahmen dieser Arbeit konnten für ausgewählte Teilbereiche einer Komponentensicht derartige fundamentale Merkmale identifiziert werden und machten dort den Nachweis möglich, daß sich auf ihnen prinzipiell geregelte wissenschaftliche Methoden für komponentenorientierte Systeme gründen lassen.

Die Untersuchung konkreter konzeptioneller Ansätze formaler Spezifikation demonstrierte die Potentiale logikbasierter Methoden im Kontext komponentenorientierter Systeme. Es zeigte sich dabei die generelle Eignung derartiger formaler Semantiken zur Beschreibung und Auswertung inhaltlicher Verhaltens- und Strukturmerkmale von Softwarekomponenten.

Neben den vielversprechenden Möglichkeiten der Ansätze ergaben sich auch Hinweise auf die Grenzen allgemeiner Verwendbarkeit. Die grundsätzliche Möglichkeit einer sinnvollen Anwendung konnte aber durch das Aufzeigen von Integrationsmöglichkeiten in ein konkretes System untermauert werden.

7.2 Ausblick

Das nach wie vor junge Themengebiet der Componentware bietet vielfältige Ansatzpunkte für weitere Untersuchungen. Deren Umfang wurde unter anderen anhand dieser Arbeit deutlich gemacht. Vor allem im Bereich fundamentaler wissenschaftlicher Grundlagen und darauf begründeter Vorgehensweisen ist weiterhin Pionierarbeit zu leisten.

Die hier vorgestellten fundamentalen Merkmale und Prinzipien bilden einen möglichen Schritt auf dem Weg zu einer kompletten konzeptionellen Basis komponentenorientierter Systeme. Sie stellen einen Anfang dar, der weiter ausgebaut und verfeinert werden kann, um zu praktisch anwendbaren Ergebnissen zu gelangen.

Im Hinblick auf eine faktische Einbettung formaler Semantik als integraler Bestandteil neuer oder Zusatz bestehender Komponentensysteme können die erarbeiteten konzeptionellen Lösungen formaler Spezifikationen weiter präzisiert werden.

Für die konkreten Modifikations- und Erweiterungsvorschläge der JavaBeans Architektur müssen endgültige Beweise durch Implementation erbracht werden. Die Möglichkeit der Erweiterung durch systemtechnische Unterstützungsmechanismen in Form von Policymanagement ist dabei bereits Gegenstand laufender Untersuchungen [Tet99].

Abbildungsverzeichnis

1.1 Softwarekomponenten 5
1.2 Visuelle Programmierung mit Java Studio-Beans 9

3.1 Object Managaement Architecture 46
3.2 Gluon als Komponenetenmediator 50
3.3 Struktur einer DCUP Komponente 50
3.4 Composition Filters 51
3.5 Darwin Komponenten 53
3.6 Objekte, Vererbung und Methodenaufruf in OPUS 55
3.7 Komponentennetz eines Vista Frameworks 57
3.8 IBM San Francisco Architektur 59

4.1 Grundlegende Komponentenstrukturen 73
4.2 Applikationsübergreifende Komponentennutzung 74
4.3 Zyklen 76
4.4 Mögliche Strukturen bei multiplen Interfaces 78
4.5 Bidirektionaler Nachrichtenaustausch 79
4.6 Kanalabstraktion 80
4.7 Schnittstellenspezifikation 91
4.8 Synchronisierte Einzel- und Multischnittstellenspezifikation . 93
4.9 Zentralspezifikation 94
4.10 Externe Importspezifikation 95

5.1 Einfache Komponente 106
5.2 Der Komponentenkern 107
5.3 Beispiel einer Schnittstellenstruktur 109

5.4 Komponentenkomposition aus Modellsicht 110
5.5 Komponenteninteraktion als Nachrichtensequenz 111
5.6 Globale Interaktion als Wechselwirkung lokaler Verhaltensaspekte . 112
5.7 Formen temporaler Logik 113
5.8 Lineare Zeitlinie mit diskreten Zeitpunkten 115
5.9 Bedeutung von PLTL Operatoren über der Zeitlinie 116
5.10 Beispielhafte Komponentenspezifikation "Maschine" 120
5.11 Mögliche korrekte Nachrichtensequenz der Maschine 121
5.12 Verifikation einer Komponentenstruktur 122
5.13 Modifikation des Spezifikationsmodells 125
5.14 Beziehungsdiagramm 131
5.15 Anwendung der Feature Logik zur Komponentenstrukturbeschreibung 133
5.16 Multi-Level Modell mit beliebigen Strukturen 135
5.17 Rollenverteilung der Logiken 137

6.1 Schematische Klassendefinition einer JavaBean Komponente . 154
6.2 Das Java Interface der `MaschinenBean` 155
6.3 Die `MaschinenBean` Schnittstelle in CORBA IDL 156
6.4 Das verbesserte `MaschinenBean` Interface 156
6.5 Component-Trading 160
6.6 Klienten eines Typmanagers 161
6.7 Prinzip des Policymanagements 162

Tabellenverzeichnis

4.1 Mehrfachverwendung von Komponenten 75

5.1 Temporale Operatoren 115
5.2 Exportbasierte Spezifikation 127
5.3 Grundbegriffe der Feature-Logik 129
5.4 Syntax und Semantik von Feature-Termen 129
5.5 Rollen-Quantoren in Konzeptbeschreibungen 130
5.6 Formale und modellbezogene Sicht der Feature-Logik 132

Literaturverzeichnis

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, Aug 1986.

[AHS93] F. Arbab, I. Herman, and P. Spilling. An overview of manifold and its implementation. *Concurrency: Pratice and Experience*, 5(1):23–70, Feb 1993.

[Ara91] C. Arapis. Temporal specifications of object interactions. In *Proceedings Third International Workshop on Foundations of Models and Languages for Data and Objects*, pages 15–35, Aigen, Austria, Sep 1991.

[Ara92] C. Arapis. *Dynamic Evolution of Object Behaviour and Object Cooperation.* PhD thesis, Centre Universitaire d'Informatique, University of Geneva, 1992.

[Ara95] C. Arapis. *A Temporal Perspective of Composite Objects*, chapter 5, pages 123–152. In Nierstrasz and Tsichritzis [NT95], 1995.

[AT98] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters - aop'98 workshop position paper. Technical Report CS-93-41, TRESE project, University of Twente, Centre for Telematics and Information Technology, P.O. Box 217, 7500 AE, Enschede, The Netherlands, 1998.

[BDH+98] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plasil, G. Pomberger, W. Pree, M. Stahl, and C. Szyperski. What characterizes a (software) component? *Software Concepts & and tools*, 19(1):49–56, 1998.

[BHLM94] J. Buck, S. Ha, E.A. Lee, and D.G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, special issue on "Simulation Software Development'', Jan 1994.

[Boe88] B.W. Boehm. A spiral model of software development and enhancement. *IEEE Software*, 25(5):61–72, May 1988.

[Boo87] G. Booch. *Software Components with ada: Structures, Tools and Subsystems.* Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

[BR92] F. Bott and M. Ratcliffe. *Reuse and Design*, pages 35–51. In Hall [Hal92], 1992.

[Bra94] C. L. Braun. Nato standard for the development of reusaable software components. Technical Report Teil 1 von 3, NATO, 1994.

[Bro96] Manfred Broy. Towards a mathematical model of a component and its use. In *Componentware Users Conference 1996, Munich, Proceedings.* SIGS Publications, to appear, 1996.

[Bro98] Manfred Broy. Compositional refinement of interactive systems modelled by relations. In *International Symposium Compositionality 1997.* Lecture Notes in Computer Science, Springer, to appear, 1998.

[Bur97] H. Burkhart. *Parallele Programmierung*, pages 479–503. In Rechenberg and Pomberger [RP97], 1997.

[BW97] M. Büchi and W. Weck. A plea for grey-box components. Technical Report TUCS Technical Report No 122, Turku Centre for Computer Science (TUCS), 1997.

[CDK94] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design.* Addisson-Wesley, second edition, 1994.

[Cha96] D. Chappel. *ActiveX und OLE Verstehen.* Microsoft Press, Deutschland, 1996.

[CM95] B. Christiansen and M. Münke. Diensttypmanagement in offenen verteilten Systemen unter besonderer Berücksichtigung von DCE und CORBA. Studienarbeit, Universität Hamburg, Fachbereich Informatik, 1995.

[Com95] International Electronical Commission. *Basic Reference Model of Open Distributed Processing — Part3: Architecture. International Standard 10746-3.* International Organisation for Standardization, 1995.

[Dam97] L. Dami. A lambda-calculus for dynamic binding. Technical report, Centre Universitaire d´Informatique, Feb 1997.

[DK93] M.F. Dunn and J.C. Knight. Certification of reusable software parts. Technical Report CS-93-41, University of Virginia, 1993.

[FHPH95] J. Friedrich, Th. Herrmann, M. Peschek, and A. Rolf (Hrsg.). *Informatik und Gesellschaft.* Spektrum Akademischer Verlag, Heidelberg; Berlin; Oxford, 1995.

[FN94] K. Fisher and M. Nielsen. Notes on typed object-oriented programming. In *Lecture Notes in Computer Science: Proceedings TACS'94*, pages 844–885. Springer-Verlag, 1994.

[Fra97] X. Franch. The convenience for a notation to express non-functional characteristics of software components. pages 101–110, Sep 1997.

[Gab92] D.M. Gabbay. Temporal logic: Mathematical foundations. Technical Report MPI-I-92-213, Max-Planck-Institut für Informatik, Im Stadtwald, W 6600 Saarbrücken, Germany, Mar 1992.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series, 1994.

[GHR92] D.M. Gabbay, I.M. Hodkinson, and M.A. Reynolds. Temporal logic: Mathematical foundations, part 2. Technical Report MPI-I-92-242, Max-Planck-Institut für Informatik, Im Stadtwald, W 6600 Saarbrücken, Germany, Sep 1992.

[Gö97] M. Göllnitz. Generische Verarbeitung von Richtlinien für offene verteilte Systeme. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1997.

[Gri98] F. Griffel. *Componentware.* dpunkt. Verlag, 1998.

[Hal92] P. A. V. Hall, editor. *Software Reuse and Reverse Engineering in Practice.* Chapman & Hall, 1992.

[HC91] J. W. Hooper and R. O. Chester. *Software Reuse: Guidelines and Methods.* Plenum Press, New York, 1991.

[HLS97] K. De Hondt, C. Lucas, and P. Steyaert. Reuse contracts as component interface descriptions. pages 43–49. Turku Centre for Computer Science General Publication No 5, Sep 1997.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug 1978.

[HP88] R. Holibough and J. Perry. Phase i testbed description: Requirements and selection guidelines. Technical Report CMU/SEI-88-TR-13, Software Engineering Institute, Carnegie Mellon University, Sep 1988.

[HU93] J.E. Hopcroft and J.D. Ullman. *Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie.* Adison-Wesley, 1993.

[Hug97] J. Hugunin. Python and java: The best of both worlds. URL:http://www.python.org/jpython/ipc6paper.html, 1997.

[JGJ94] L. Jacobsen, M. Griss, and P. Johnson. *Software Reuse — Architecture, Process and Organisation for Business Success.* ACM Press/Addison Wesley, 1994.

[Joh98] R. Johnson. Tcl and java integration. Technical report, Sun Microsystems Laboratories, 901 San Antonio, MS UMTV-29-232, Palo Alto, CA 94303-4900, Feb 1998.

[Jon94] K. Jones. Vermittlung und Verwaltung von Diensten in offenen verteilten Systemen: Ein Objekt- und Architekturmodell. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1994.

[Kai96] J. B. Kain. Components: The basics: Enabling an application or system to be the sum of its parts. *Object Magazine*, 6(2):64–69, Apr 1996.

[KIL+97] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar. *Aspect Oriented Programming*, chapter II: Proceedings of the ECOOP Workshop on Composibility Issues in Object-Orientation (CIOO'96), pages 63–74. In Mühlhäuser [Müh97], 1997.

[KLM+97] G. Kiczales, J. Lamping, A Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-oriented programming. volume 1241, pages 220–242. Springer-Verlag, June 1997.

[Kon95] D. Konstantas. *Interoperability of Object-Oriented Applications*, chapter 3, pages 69–95. In Nierstrasz and Tsichritzis [NT95], 1995.

[Koo95] P. Koopmans. *Sina user´s guide and reference manual.* Dept. of Computer Science, University of Twente, Enschede, Netherlands, 1995.

[Ler94] X. Leroy. Manifest types, modules, and separate compilation. In *Proceedings POPL'94*, pages 109–122. ACM Press, 1994.

[Lie96] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996.

[Lio96] J.L. Lions. Ariane 5 — flight 501 failure — report by the inquiry board. Technical report, European Space Agency (ESA), Paris, France, Jul 1996.

[McI68] M.D. McIlreu. Mass produced software components. In P. Nauer and Randell, editors, *Software Engineering: Report on a Conference by the NATO Science Committee*, pages 138–155. NATO Scientific Affairs Division, 1968.

[MDE97] T.D. Meijler, S. Demeyer, and R. Engel. *Class Composition in FACE, a Framework Adaptive Composition Environment*, chapter II: Proceedings of the ECOOP Workshop on Composibility Issues in Object-Orientation (CIOO'96), pages 117–124. In Mühlhäuser [Müh97], 1997.

[MDK96] J. D. McGregor, J. Doble, and A. Keddy. A pattern for reuse: Let architectural reuse guide component reuse. *Object Magazine*, 6(2):38–47, Apr 1996.

[Mey88] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[Mey95] V.d. Mey. *Visual Composition of Software Applications*, chapter 10, pages 275–303. In Nierstrasz and Tsichritzis [NT95], 1995.

[MH99] V. Matena and M. Hapner. *Enterprise Java Beans 1.1 Specification Draft.* Sun Microsystems, May 1999.

[MJ96] K. Müller-Jones. Koordinierte Nutzung von Diensten in offenen verteilten Dienstemärkten. Dissertation, Universität Hamburg, Fachbereich Informatik, 1996.

[MJML95] K. Müller-Jones, M. Merz, and W. Lamersdorf. Kooperationsanwendungen: Integrierte Vorgangskontrolle und Dienstvermittlung in offenen verteilten Systemen. In F. Huber-Wäschle amd H. Schauer and P. Widmayer, editors, *GISI 95 — Herausforderungen eines globalen Informationsverbundes für die Informatik*, pages 518–525, Zürich, 1995. Springer Verlag.

[MMS94] T. Mens, K. Mens, and P. Steyaert. OPUS: a Formal Approach to Object-Orientation. In Maurice Naftalin, Tim Denvir, and Miquel Bertran, editors, *FME´94: Industrial Benefit of Formal*

Methods, number 873 in Lecture Notes in Computer Science, pages 326–345. Springer-Verlag, 1994. Proceedings of the Second International Symposium of Formal Methods Europe. Barcelona, Spain, October 1994.

[MMS95] T. Mens, K. Mens, and P. Steyaert. OPUS: a Calculus for Modelling Object-Oriented Concepts. In D. Patel, Y. Sun, and S. Patel, editors, *Proceedings of the 1994 International Conference on Object Oriented Information Systems*, pages 152–165. Springer-Verlag, 1995.

[MNSJ95] J. Magee, N.Dulay, S.Eisenbach, and J.Kramer. Specifying distributed software architectures. In *In Proceedings of the Fifth European Software Engineering Conference, Barcelona*, Sep 1995.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume Specification. Springer-Verlag, New York Berlin Heidelberg, 1992.

[MS96] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, Inc., One Jacob Way, Reading, Massachusetts 01867, 1996.

[MSL97] K. Mens, P. Steyaert, and C. Lucas. *Reuse Contracts: Managing Evolution in Adaptable Systems*, chapter I: Proceedings of the ECOOP Workshop on Adaptability in Object-Oriented Software Development, pages 37–42. In Mühlhäuser [Müh97], 1997.

[MSW97] T. Murer, D. Scherer, and A. Würtz. *Improving Component Interoperability*, chapter III: Proceedings of the ECOOP Workshop on Component-Oriented Programming (WCOP'96), pages 150–158. In Mühlhäuser [Müh97], 1997.

[Müh97] Max Mühlhäuser, editor. *Spezial Issues in Object-Oriented Programming, Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP'96.* dpunkt.Verlag, 1997.

[MW84] Z. Manna and P. Wolper. Synthesis of communicatin process. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, Jun 1984.

[ND95] O. Nierstrasz and L. Dami. *Component-Oriented Software Technologie*, chapter 1, pages 3–28. In Nierstrasz and Tsichritzis [NT95], 1995.

[Nie95] O. Nierstrasz. *Regular Types*, chapter 4, pages –122. In Nierstrasz and Tsichritzis [NT95], 1995.

[NL97] O. Nierstrasz and M. Lumpe. Komponenten, Komponentenframeworks und Gluing. *HMD — Theorie und Praxis der Wirtschaftsinformatik*, 197:8–23, 9 1997.

[NT95] O. Nierstrasz and D. Tsichritzis, editors. *Object-Oriented Software Composition.* Prentice Hall International (UK) Ltd, 1995.

[OB97] A. Olafsson and D. Bryan. *On the Need for "Required Interfaces" of Components*, chapter III: Proceedings of the ECOOP Workshop on Component-Oriented Programming (WCOP'96), pages 159–165. In Mühlhäuser [Müh97], 1997.

[OMG96a] OMG. *Common Facilities RFP-4, Common Business Objects and Business Object Facility.* Object Management Group (OMG), Framingham, MA., USA, Jan 1996.

[OMG96b] OMG. *The Common Object Request Broker: Architecture and Specification.* Object Management Group (OMG), Framingham, MA., USA, Jul 1996.

[OMG97] OMG. *A Discussion of the Object Management Architecture.* Object Management Group (OMG), Framingham, MA., USA, Jan 1997.

[OPR96] R. Otte, P. Patrick, and M. Roy. *Understanding CORBA: The Common Object Request Broker Architecture.* Prentice Hall, One Jacob Way, Reading, Massachusetts 01867, 1996.

[Pap95] M. Papathomas. *Concurrency in Object-Oriented Programming Languages*, chapter 2, pages 31–68. In Nierstrasz and Tsichritzis [NT95], 1995.

[Paz97] L. Pazzi. *An Explizit Modelling Perspective for Compound and Aggregate Entities in the Object Paradigm*, chapter III: Proceedings of the ECOOP Workshop on Component-Oriented Programming (WCOP'96), pages 166–171. In Mühlhäuser [Müh97], 1997.

[PBJ97] F. Plasil, D. Balek, and R. Janecek. Dcup: Dynamic component updating in java/corba environment. Technical report, Charles University, Faculty of Mathematics and Physiks, Department of Software Engineering, Malostranske namesti 25, 11800 Prague 1, Czech Republic, 1997.

[PBJ98] F. Plasil, D. Balek, and R. Janecek. Sofa/dcup: Architecture for component trading and dynamic updating. Technical report, Charles University, Faculty of Mathematics and Physiks, Department of Software Engineering, Malostranske namesti 25, 11800 Prague 1, Czech Republic, 1998.

[Pin95] X. Pintado. *Gluons and the Cooperation between Software Components*, chapter 12, pages 321–349. In Nierstrasz and Tsichritzis [NT95], 1995.

[Pre97] W. Pree. *Komponentenbasierte Softwareentwicklung mit Frameworks.* dpunkt.verlag, 1997.

[PVS97] J. Peter, M. Vollmer, and W. Stripf. IBM San Francisco — Anwendungsentwicklung mit Java-Geschäftsprozeß-Komponenten. *HMD — Theorie und Praxis der Wirtschaftsinformatik*, 197:76–90, September 1997.

[RE96] M. Radestock and S. Eisenbach. Semantics of a higher-order coordination language. In *Proceedings of the First International Conference in Coordination Languages and Models, Cesena, Italy*, Apr 1996.

[Rei95] S. P. Reiss. *The FIELD Programming Environment: A Friendly Integrated Environment for Learning and Development.* Kluwer Academic Publishers, 1995.

[Rog97] D. Rogerson. *Inside COM.* Microsoft Press, Redmond, WA., USA, 1997.

[RP97] P. Rechenberg and G. Pomberger, editors. *Informatik-Handbuch.* Carl Hanser Verlag, München Wien, 1997.

[Sam97] J. Sametinger. *Software Engineering with Reusable Components.* Springer-Verlag Berlin Heidelberg New York, 1997.

[Sch97] M. Schmidt. Generische, orthogonale un persistente Verarbeitung von Eigenschaften: Eine Implementierung des CORBA Property Service in Java. Studienarbeit, Universität Hamburg, Fachbereich Informatik, 1997.

[Smo92] G. Smolka. Feature-constrained logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

[Stö98] H. J. Störig. *Kleine Weltgeschichte der Philosophie.* Fischer, 1998.

[Sun97a] Sun. *Java Beans 1.01 API Specification.* Sun Microsystems, 1997.

[Sun97b] Sun. *Java Beans for Java Studio: Architecture and API - A Technical White Paper.* Sun Microsystems, Palo Alto, CA., USA, 1997.

[Sun97c] Sun. *JDK 1.1 API Specification.* Sun Microsystems, 1997.

[Syy98] U. Syyid. *The Adaptive Communication Environment: "Ace´´.* Hughes Network Systems, 11717, Exploration Lane, Germantown MD, 20876, 1998.

[Tet99] D. Tetau. Regelgestützte Komponentenkomposition. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1999.

[TGML97] M. T. Tu, F. Griffel, M. Merz, and W. Lamersdorf. Policy management for open service markets. In H.König und K. Geihs, editor, *Proceedings of the International Working Conference on Distributed Applications and Interoperable Systems (DAIS'97).* Chapman& Hall, 1997.

[Ude94] J. Udell. Componentware. *Byte*, 19(5):46–56, May 1994.

[Weg84] P. Wegner. Capital-intensive software technology. *IEEE Software*, 1(3), Jul 1984.

[Weg93] P. Wegner. Towards component-based software technology. Technical Report CS-93-11, Brown University, 1993.

[Weg97] P. Wegner. Why interaction is more powerful than algorithms. *Communications of the ACM*, 40(5):80–91, May 1997.

[WGM89] A. Weinand, E. Gamma, and R. Marty. Design and implementation of et++, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, Jul 1989.

[YS95] D. M. Yellin and R. E. Strom. Collaboration specification and component adaptors. Technical report, IBM Research Division T.J. Watson Research Center, P.O. Box 218 Yorktown Heights, NY 10598, 1995.

[Zel96] A. Zeller. *Configuration Management with Version Sets.* PhD thesis, Technical University of Braunschweig, Germany, Nov. 1996.

[ZS96] A. Zeller and G. Snelting. Unified versioning through feature logic. Technical Report 96-01, Technical University of Braunschweig, Germany, Mar 1996.

Erklärung:

Hiermit versichere ich, die vorstehende Arbeit selbständig und ohne fremde Hilfe unter ausschließlicher Nutzung der angegebenen Quellen und Hilfsmittel angefertigt zu haben.

Hamburg, 1. Juni 1999

Christian Zirpins