Ein komponentenbasiertes Software-Architekturmodell zur Entwicklung betrieblicher Anwendungssysteme

Inaugural-Dissertation
zur Erlangung des Grades
eines doctor rerum politicarum
der Fakultät Wirtschaftsinformatik und Angewandte Informatik
der Universität Bamberg

vorgelegt von

Christian Robra

aus Darmstadt

Bamberg 2003

Erstgutacher: Prof. Dr. Elmar J. Sinz

Zweitgutachter: Prof. Dr. Otto K. Ferstl

Tag der Disputation: 14. September 2004

Inhaltsübersicht

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis fachlicher Abkürzungen

2PC	Two-Phase Commit Protocol
ACID	Atomicity, Consistency, Isolation, Durability
ADK	Anwendungsfunktionen, Datenhaltung, Kommunikation
ADT	Abstrakter Datentyp
ANSI	American National Standards Institute
API	Application Programming Interface
CBD	Component-Based Development
CBS	Component-Based System
CBSE	Component-Based Software Engineering
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
COTS	Commercial off-the-shelf
CWM	Common Warehouse Model
DBVS	Datenbankverwaltungssystem
DIN	Deutsche Industrie-Norm
EBNF	Erweiterte Backus-Naur-Form
EJB	Enterprise JavaBeans
ERM	Entity-Relationship-Modell
FTP	File Transfer Protocol
GUI	Graphical User Interface
HTTP	Hypertext Transport Protocol
IAS	Interaktionsschema
IDL	Interface Definition Language
ISO	International Standardisation Organization
ISV	Independent Software Vendor
ITU	International Telecommunications Union
J2EE	Java 2 Enterprise Edition

JB	JavaBeans
KOS	Konzeptuelles Objektschema
KOT	Konzeptueller Objekttyp
LOT	Leistungsspezifischer Objekttyp
MDA	Model Driven Architecture
MOF	Meta-Object Facility
OCL	Object Constraint Language
ODMG	Object Data Management Group
OMG	Object Management Group
OOA	Object-oriented Analysis
OOD	Object-oriented Design
OOT	Objektspezifischer Objekttyp
ORB	Object Request Broker
OTS	Object Transaction Service
PC	Personal Computer
PLoP	Pattern Languages of Programs
RMI	Remote Method Invocation
RM-ODP	Reference Model for Open Distributed Processing
RPC	Remote Procedure Call
SERM	Strukturiertes Entity-Relationship-Modell
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
SOM	Semantisches Objektmodell
SQL	Structured Query Language
TOT	Transaktionsspezifischer Objekttyp
TP	Transaction-Processing
TRPC	Transactional Remote Procedure Call
UDDI	Universal Service Description, Discovery and Integration
UIMS	User-Interface-Management-System
UML	Unified Modeling Language

VE	Vorgang, Entität
VES	Vorgangs-Ereignis-Schema
VOS	Vorgangsobjektschema
VOT	Vorgangsobjekttyp
W3C	World Wide Web Consortium
WSA	Web Service Architecture
WSDL	Web Service Description Language
WWW	World Wide Web
XMI	XML Metadata Interchange
XML	Extensible Markup Language

1 Einleitung

Architekturmodelle sind ein wesentliches Hilfsmittel zur ganzheitlichen Analyse und Gestaltung sowie zur zielgerichteten Nutzung von betrieblichen Anwendungssystemen. Demzufolge kann auch mit einem geeigneten Software-Architekturmodell der softwaretechnische Entwurf eines betrieblichen Anwendungssystems unterstützt werden. Ein Software-Architekturmodell gilt als geeignet, wenn es zumindest

- ein Entwurfsvorgehen fördert, mit dem Entwickler betrieblicher Anwendungssysteme auf die verkürzten Software-Produktlebenszyklen und den gestiegenen Kostendruck reagieren können und
- den softwaretechnischen Entwurf unter der Bedingung steigender funktionaler und nicht-funktionaler Anforderungen unterstützt, die sich aus den wachsenden Automatisierungs- und Dezentralisierungsbestrebungen ergeben.

Dies erfordert Software-Architekturmodelle, die aus wiederverwendbaren und verteilbaren Entwicklungserzeugnissen bestehen. Zur Unterstützung der ganzheitlichen Analyse und Gestaltung müssen geeignete Software-Architekturmodelle außerdem in unterschiedliche Informationssystem-Architekturen integrierbar sein und insbesondere für einen lückenlosen Übergang vom Fachentwurf zum softwaretechnischen Entwurf eines Anwendungssystems sorgen.

1.1 Problemstellung

Als Elemente für Software-Architekturmodelle existieren bereits mehrere Arten von wiederverwendbaren und verteilbaren Entwicklungserzeugnissen, die jedoch die an sie gestellten diesbezüglichen Erwartungen aus unterschiedlichen Gründen nicht hinreichend erfüllen. Zu den prominentesten Vertretern gehören objektorientierte Klassen und Frameworks.

Eine alternative Erzeugnisart, über die in der letzten Zeit verstärkt diskutiert wird und von der man sich eine verbesserte Wiederverwendbarkeit und Verteilbarkeit verspricht, ist die Software-Komponente. Allerdings herrscht noch keine Einigkeit über die genaue Ausgestaltung des zugrundeliegenden Konzepts, mit dem die genannten Versprechungen erfüllt werden können. Somit entstand in den letzten Jahren ein diffuses Verständnis einer Software-Komponente, das ihren erfolgreichen Einsatz in der Anwendungssystementwicklung im Allgemeinen und in Software-Architekturmodellen im Speziellen verhinderte. Darüber hinaus führte das uneinheitliche Verständnis dazu, dass Software-Produzenten den Begriff „Komponente“ aus vermarktungspolitischen Gründen für zweckspezifisch vorgefertigte objektorientierte Klassen nutzten und damit die Bildung eines eigenständigen Grundkonzepts der Komponentenorientierung erschwerten.

Aus diesen Gründen droht die Erzeugnisart Software-Komponente, die grundsätzlich das Potenzial zur Lösung der Wiederverwendungs- und Verteilungsprobleme im softwaretechnischen Entwurf aufweist, noch vor ihrer einheitlichen Definition und ihrem praktischen Einsatz in Software-Architekturmodellen bedeutungslos zu werden.

Um dies zu verhindern, ist ein einheitliches und methodisch fundiertes Software-Komponenten-Konzept notwendig, das insbesondere die Wiederverwendbarkeit und Verteilbarkeit von Software-Komponenten sicherstellt.

Außerdem erfordert der Entwurf der auf diesem Konzept basierenden Software-Komponenten einen geeigneten methodischen Rahmen, der eine ganzheitliche und methodisch durchgängige Entwicklung von Anwendungssystemen unterstützt. Zu diesem Zweck wird ein Software-Architekturmodell benötigt, das in unterschiedliche Informationssystem-Architekturen integrierbar ist und insbesondere den lückenlosen Übergang vom Fachentwurf zum softwaretechnischen Entwurf sicherstellt.

1.2 Zielsetzung und Lösungsansatz der Arbeit

In der vorliegenden Arbeit soll ein Software-Architekturmodell entwickelt werden, das den softwaretechnischen Entwurf wiederverwendbarer und verteilbarer betrieblicher Anwendungssysteme und -teilsysteme durch den Einsatz von Software-Komponenten unterstützt.

Dafür ist zunächst die Erarbeitung eines präskriptiven Grundkonzepts der Komponentenorientierung notwendig, auf dem die einzusetzenden Software-Komponenten basieren. Die methodische Begründung bezieht das Grundkonzept aus den Anforderungen, die sich aus der zu erzielenden Wiederverwendbarkeit und Verteilbarkeit von Anwendungssystemen und -teilsystemen ergeben. Diese richten sich sowohl an das Entwicklungserzeugnis, die Software-Komponente, als auch an das Vorgehen im softwaretechnischen Entwurf und in der Implementierung.

Das präskriptive Grundkonzept dient zusammen mit einem geeigneten generischen Architekturrahmen und geeigneten Strukturmodellen von Anwendungssystemen der Erstellung eines komponentenbasierten Software-Architekturmodells. Es wird aufgezeigt, dass dieses einen lückenlosen Übergang von einer objektorientierten fachlichen Anwendungssystemspezifikation zu einem komponentenbasierten softwaretechnischen Entwurf ermöglicht und sich in verschiedenen Informationssystem-Architekturen einsetzen lässt.

Ein erster praktischer Nachweis der Wiederverwendbarkeit und Verteilbarkeit komponentenbasierter Anwendungssysteme, die mit dem Software-Architekturmodell entworfen wurden, erfolgt anhand von zwei Fallstudien, die aus unterschiedlichen Anwendungsdomänen stammen.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in fünf weitere Kapitel.

Kapitel 2 schafft die notwendigen Grundlagen für die Zielsetzung und den Lösungsansatz der Arbeit. Dazu gehören insbesondere die Einordnung des softwaretechnischen Entwurfs betrieblicher Anwendungssysteme in den Kontext der ganzheitlichen und methodisch durchgängigen Entwicklung betrieblicher Informations- und Anwendungssysteme sowie die Klärung des Architekturbegriffs, der dieser Arbeit zugrunde liegt.

Ferner wird mit dem Semantischen Objektmodell (SOM) eine Methodik zur Entwicklung betrieblicher Informations- und Anwendungssysteme vorgestellt, in deren Informationssystem-Architektur das Software-Architekturmodell exemplarisch eingeordnet ist.

Diese Grundlagen sind unmittelbare Voraussetzungen für das Verständnis der Kapitel 3, 4, 5 und 6.

In **Kapitel 3** erfolgt eine systematische Erörterung der Anforderungen, die an Entwicklungserzeugnisse und an das Vorgehen im softwaretechnischen Entwurf und in der Implementierung gerichtet sind und die aus der geforderten Wiederverwendbarkeit und Verteilbarkeit von Anwendungssystemen und -teilsystemen resultieren.

Außerdem stellt das Kapitel Prinzipien und Modelle vor, die zur Erfüllung dieser Anforderungen beitragen.

Anhand dieser Ergebnisse folgt schließlich eine Untersuchung und Bewertung der Wiederverwendungs-, Wiederverwendbarkeits- und Verteilbarkeitsunterstützung ausgewählter Erzeugnisarten.

Kapitel 4 stellt ein präskriptives Grundkonzept der Komponentenorientierung vor, das ausschließlich auf den Grundlagen des Kapitels 2 und den Ergebnissen des Kapitels 3 basiert und somit methodisch fundiert ist. Es beschreibt im Wesentlichen, wie Software-Komponenten und komponentenbasierte Anwendungssysteme im softwaretechnischen Entwurf und in der Implementierung gestaltet sein müssen, damit sie bestmöglich wiederverwendbar und verteilbar sind.

In **Kapitel 5** wird auf der Grundlage aller bisherigen Ergebnisse ein komponentenbasiertes Software-Architekturmodell konzipiert, das der Entwicklung wiederverwendbarer und verteilbarer betrieblicher Anwendungssysteme dient und zu diesem Zweck in eine umfassende Informationssystem-Architektur integriert werden kann. Es besteht aus einer Ebene für die Modellierung der Außensichten von Software-Komponenten einschließlich ihrer Beziehungen und einer Ebene für die Modellierung der zugehörigen Innensichten sowie der von den Software-Komponenten benötigten Dienste.

Ferner stellt dieses Kapitel Konzepte für Basismaschinen vor, die für die Entwicklung und Ausführung komponentenbasierter und verteilter betrieblicher Anwendungssysteme geeignet sind.

Schließlich erfolgt eine exemplarische Einordnung des Software-Architekturmodells in die Informationssystem-Architektur der SOM-Methodik.

Kapitel 6 dient einer ersten Evaluation des komponentenbasierten Software-Architekturmodells. Zu diesem Zweck werden zwei Fallstudien aus unterschiedlichen betrieblichen Domänen eingeführt und jede Fallstudie zunächst vollständig, d. h. vom Unternehmensplan über die zugehörigen Geschäftsprozesse und die objektorientierte fachliche Anwendungssystemspezifikation bis zum komponentenbasierten softwaretechnischen Entwurf modelliert.

Anhand eines exemplarischen Ausschnitts aus dem komponentenbasierten softwaretechnischen Entwurf der ersten Fallstudie wird die Wiederverwendbarkeit der darin enthaltenen Software-Komponenten aufgezeigt. Zunächst erfolgt eine Aufbereitung dieser Software-Komponenten hinsichtlich einer künftigen Wiederverwendung in unterschiedlichen betrieblichen Domänen. Anschließend werden sie ohne zusätzliche Veränderungen in den softwaretechnischen Entwurf der zweiten Fallstudie eingebracht. Schließlich dient der letzte Schritt der Anpassung von verbliebenen Software-Komponenten der zweiten Fallstudie an die wiederverwendeten Software-Komponenten aus der ersten Fallstudie.

Zum Nachweis der Verteilbarkeit der entworfenen Software-Komponenten werden diese als verteiltes Anwendungssystem implementiert.

In **Kapitel 7** folgt eine zusammenfassende Bewertung der in der Arbeit erzielten Ergebnisse.

2 Grundlagen der Entwicklung betrieblicher Informations- und Anwendungssysteme

Ein Teilgebiet der Wirtschaftsinformatik als wissenschaftliche Disziplin befasst sich mit der Konstruktion von betrieblichen Informationssystemen [FeSi01, 8; Sinz99f, 803; HeSi02, 1037 f; Hein02, 1042].

Informationssysteme sind in einer ersten Begriffsbestimmung Systeme, die Informationen verarbeiten. Dazu gehört die Erfassung, die Übertragung, die Transformation, die Speicherung und die Bereitstellung von Informationen [FeSi01, 1]. Betriebliche Informationssysteme werden in der Wirtschaft, in der Verwaltung und, bedingt durch ihre zunehmende kommunikationstechnische Vernetzung, auch in privaten Haushalten eingesetzt [FeSi01, 1 f]. Sie weisen in ihrem Aufbau und in ihrer Funktionsweise wichtige gemeinsame Eigenschaften auf, die in Abschnitt 2.1 kurz dargestellt werden. Automatisierte Teilsysteme eines betrieblichen Informationssystems sind betriebliche **Anwendungssysteme**, auf die ebenfalls in Abschnitt 2.1 kurz eingegangen wird. Die Abgrenzung zwischen einem betrieblichen Informationssystem und einem betrieblichen Anwendungssystem basiert auf dem Konzept der betrieblichen Aufgabe und dem Konzept der Automatisierung betrieblicher Aufgaben [FeSi01, 4; Ambe99a, 28 f]. Beide Konzepte werden in Abschnitt 2.1 vorgestellt.

Abschnitt 2.2 befasst sich mit der Aufgabe der Konstruktion betrieblicher Informationssysteme. Aufgrund der eingeführten Abgrenzung zwischen betrieblichem Informationssystem und betrieblichem Anwendungssystem stellt die Entwicklung betrieblicher Anwendungssysteme eine Teilaufgabe dieser Aufgabe dar. Im Verlauf des Abschnitt werden wesentliche methodische Grundlagen der Entwicklung betrieblicher Anwendungssysteme vorgestellt, die für die weiteren Ausführungen dieser Arbeit notwendig sind.

Als Hilfsmittel für die Konstruktion betrieblicher Informationssysteme setzt man vermehrt Architekturen ein, deren zugrundeliegendes Konzept in Abschnitt 2.3 erläutert wird.

Der zweite Abschnitt schließt mit einer kurzen Einführung in die Methodik des Semantischen Objektmodells (SOM) nach FERSTL und SINZ, die im Wesentlichen auf den vorgestellten Konzepten und Modellen aufbaut.

2.1 Grundlagen betrieblicher Informations- und Anwendungssysteme

Im vorangegangenen Abschnitt wurden die Begriffe „betriebliches Informationssystem“ und „betriebliches Anwendungssystem“ kurz eingeführt. Diese Definitionen sind jedoch für die folgenden Erörterungen des Problems der Konstruktion betrieblicher Anwendungssysteme zu abstrakt. Deswegen werden in diesem Abschnitt die Begriffe „betriebliches Informationssystem“ und „betriebliches Anwendungssystem“ in einem größeren Kontext betrachtet und geeignet konkretisiert.

2.1.1 Systemtheoretische Grundlagen

Wie bereits die Begriffe „Informationssystem“ und „Anwendungssystem“ andeuten, handelt es sich in beiden Fällen um Systeme. Der allgemeine Systembegriff wird in unterschiedlichen Wissenschaftsdisziplinen zur Beschreibung komplexer Sachverhalte verwendet [Schu01, 9; Hamm99, 43 f; Spie99, 722]. Als genereller theoretischer Bezugsrahmen hat sich die von VON BERTALANFFY formulierte allgemeine Systemtheorie durchgesetzt [Schu01, 9; Hamm99, 44; LeHM95, 47]. Sie stellt eine einheitliche Methodik und Terminologie für die Erfassung, Beschreibung und Untersuchung unterschiedlichster Systemklassen zur Verfügung [FeSi01, 11]. Daneben existieren mehrere unterschiedliche Entwicklungslinien und Strömungen in der Systemtheorie, die zum Teil auf der allgemeinen Systemtheorie aufbauen [Hamm99, 43 f; LeHM95, 44 ff]. Sie erweitern den sehr weit gefassten Systembegriff der allgemeinen Systemtheorie um nutzbringende Merkmale und machen ihn dadurch anwendbarer. Dieser erweiterte allgemeine Systembegriff wird im Folgenden vorgestellt und dient als methodische Grundlage für die weiteren Ausarbeitungen.

Ein **System** ist in einer ersten Begriffsbestimmung ein für einen bestimmten Zweck gebildeter, zu einer abgegrenzten Einheit zusammengefasster, vollständiger Ausschnitt aus einer wirklichen oder gedachten Welt, einem Universum [Spie99, 722]. Der nicht zum System gehörende Teil des Universums wird **Umwelt** oder Umgebung des Systems genannt [Spie99, 722; FeSi01, 5; HeRo98, 513; BöFu02, 14]. Deren Bestandteile und Eigenschaften sind so weit zu klären, wie dies zum Verständnis des Systems erforderlich ist [Spie99, 722; BöFu02, 14]. Falls zur Erfüllung des Zwecks keine Abhängigkeiten zwischen dem System und seiner Umwelt berücksichtigt werden müssen, handelt es sich um ein geschlossenes System [Spie99, 723; HeRo98, 514; BöFu02, 18]. Anderenfalls liegt ein offenes System vor, das zusätzlich eine Systemschnittstelle aufweist, über die es mit seiner Umwelt interagiert [Spie99, 723; HeRo98, 514; BöFu02, 19].

Ein System kann aus der Außensicht oder aus der Innensicht betrachtet werden [Spie99, 722; FeSi01, 17 f]. Aus der Außensicht entspricht es einer Black Box, das

heißt es lässt sich nicht mehr weiter detaillieren [FeSi01, 18; Spie99, 722; BöFu02, 20 f]. Hierbei wird nur die Schnittstelle des Systems beschrieben, über die es mit seiner Umwelt in Beziehung steht [FeSi01, 18]. Die Innensicht beschreibt die **Struktur** des Systems, bestehend aus Komponenten und Verbindungen zwischen diesen [Spie99, 722; BöFu02, 21]. Demzufolge stellt die Außensicht eine Abstraktion der Innensicht dar [Spie99, 722; FeSi01, 17]. Die **Komponenten** des Systems sind ihrerseits elementar oder wiederum Teilsysteme, die aus der Außen- und aus der Innensicht betrachtet werden können [Spie99, 722; FeSi01, 18; BöFu02, 19]. Auf diese Weise entsteht eine gegebenenfalls mehrstufige Abstraktionshierarchie, die zum Verständnis komplexer Systeme beiträgt [Spie99, 722; FeSi01, 17 f; BöFu02, 18 f]. Abbildung 2.1 veranschaulicht die bisher genannten Merkmale.

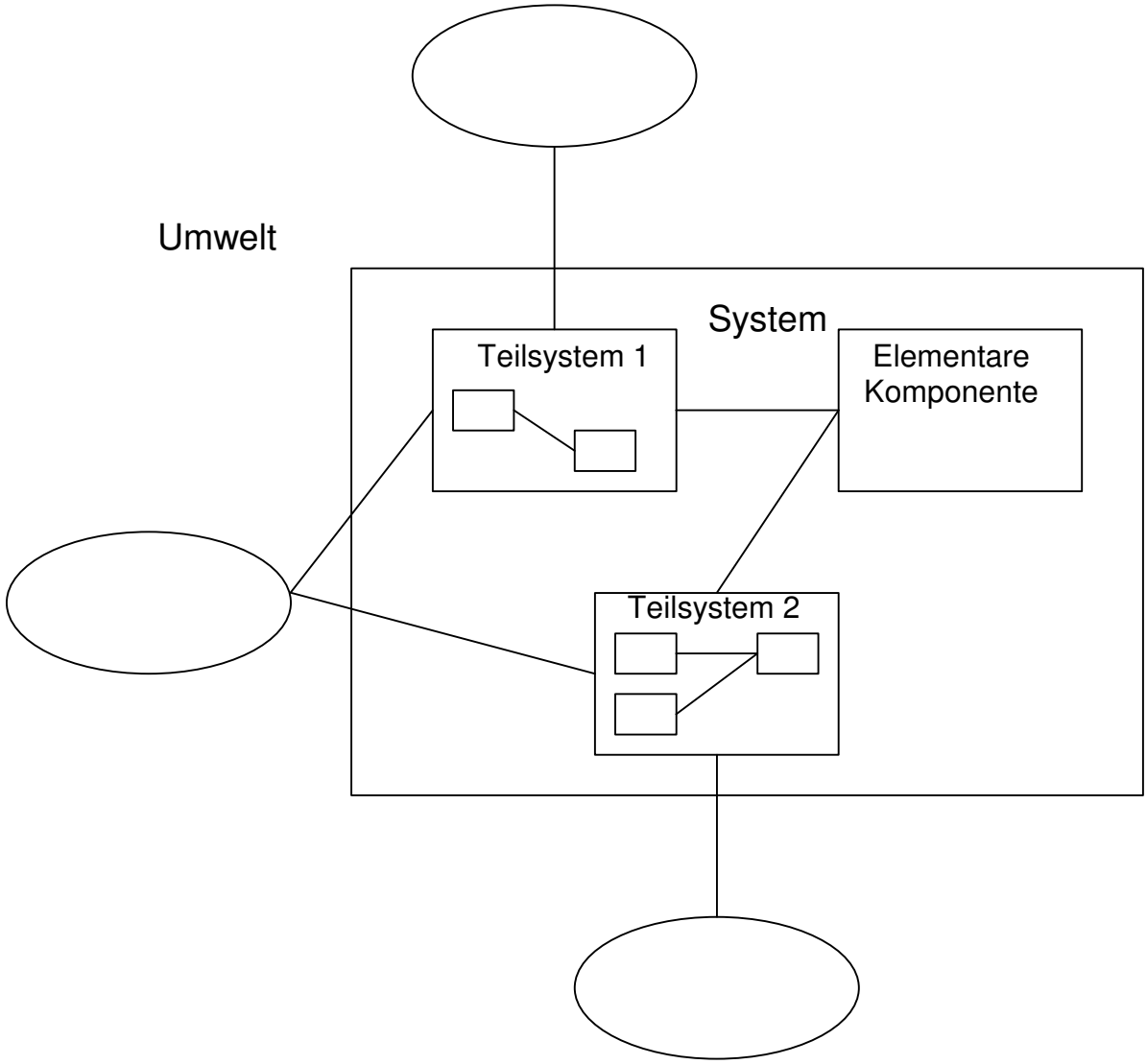


Abbildung 2.1: System, Teilsystem und Systemumwelt

Ein System besitzt zu jedem Zeitpunkt einen bestimmten **Zustand**, der sich aus den aktuellen Eigenschaftsausprägungen seiner Systemkomponenten zusammensetzt [FeSi01, 17; Spie99, 723; BöFu02, 23]. Falls sich der Zustand des Systems über den

Zeitablauf nicht ändert, liegt ein statisches System vor [Spie99, 723; HeRo98, 514]. Ansonsten spricht man von einem dynamischen System [Spie99, 723; HeRo98, 514]. Im letztgenannten Fall ändern sich die Eigenschaftsausprägungen der Systemkomponenten aufgrund eines Materie-, Energie- oder Informationsaustauschs über die Verbindungen der Systemkomponenten untereinander und, im Falle eines offenen Systems, über die Verbindungen mit der Umwelt [FeSi01, 17]. Diese Zustandsübergänge des Systems determinieren schließlich sein **Verhalten**, das die Gesamtheit aller zulässigen und korrekten Zustände beschreibt, die das System im Zeitablauf einnehmen kann [Spie99, 723; FeSi01, 17].

Neben den genannten Eigenschaften existieren weitere, anhand derer ein System näher charakterisiert werden kann. Im vorliegenden Zusammenhang sind insbesondere die Eigenschaften des Universums, aus dem ein System gebildet wurde, von Interesse. Im Falle eines wirklichen Universums liegt ein Wirklichkeitssystem bzw. reales System vor [Spie99, 723; HeRo98, 514]. Wenn das Universum gedacht ist, handelt es sich um ein Gedankensystem, auch abstraktes oder ideales System genannt [Spie99, 723; HeRo98, 514]. Ferner können Universen nach ihrer Entstehung und den zugehörigen Gesetzmäßigkeiten unterschieden werden [Spie99, 723]. Demnach ist ein aus einem naturgegebenen Universum gebildetes System ein natürliches System [Spie99, 723; HeRo98, 514]. Falls ein System aus einem von Menschen geschaffenen Universum abgegrenzt wurde, handelt es sich um ein künstliches System [Spie99, 723; HeRo98, 514]. Dazu gehören unter anderem technische Systeme, deren Realisierung anhand von technischen Mitteln erfolgte [Spie99, 723].

Anhand dieser Erläuterungen zum allgemeinen Systembegriff lassen sich bereits wesentliche Eigenschaften von Informations- und Anwendungssystemen identifizieren. Demnach handelt es sich in beiden Fällen um reale, künstliche, offene und dynamische Systeme, für die Interaktionen mit ihren Umgebungen zur Zweckbestimmung gehören [Spie99, 723].

2.1.2 Betriebliches Informationssystem

Der Begriff „Informationssystem“ wird in der Wirtschaftsinformatik sehr unterschiedlich verwendet. Gängige Definitionen sind beispielsweise in [FeSi01, 8 f] und in [BeSc98, 1 f] aufgeführt. Diese Uneinheitlichkeit ist hauptsächlich auf die mehrdeutige Verwendung des Begriffs „Information“ zurückzuführen [FeSi01, 8]. Einerseits kann damit die Tätigkeit des Informierens ausgedrückt werden, andererseits die Objektart Information im Unterschied zu anderen Objektarten, wie z. B. Materie oder Energie. In den letzten Jahren hat sich in der Wirtschaftsinformatik jedoch mehrheitlich die Verwendung des Begriffs „Informationssystem“ als informationsverarbeitendes System durchgesetzt (siehe z. B. [Hein02, 1041 f; LeHM95, 5 f; WKWI94, 80 f; BöFu02, 77; FeSi01, 9]).

Ein betriebliches Informationssystem ist demnach das informationsverarbeitende Teilsystem eines betrieblichen Systems [FeSi01, 2; Sinz99c, 349]. Der Begriff „**betriebliches System**“ wird im Folgenden als Oberbegriff für Unternehmen, Behörden, Unternehmensverbunde oder Geschäftsbereiche von Unternehmen verwendet. Die Aufgaben des betrieblichen Informationssystems resultieren aus der Zerlegung der informationsverarbeitenden Gesamtaufgabe des betrieblichen Systems und umfassen sowohl Lenkungs- als auch Leistungsaufgaben [FeSi01, 1 f]. **Lenkungsaufgaben** beziehen sich auf die Planung, Steuerung und Kontrolle der betrieblichen Prozesse [FeSi01, 1 f]. **Leistungsaufgaben** betreffen die Erstellung von Dienstleistungen in Form von Informationen [FeSi01, 1 f; Sinz99c, 349]. Diese Aufgaben werden von Menschen, Maschinen oder Mensch-Maschine-Kooperationen durchgeführt [FeSi01, 47; Hein02, 1042; LeHM95, 5 f]. Die Zuordnung von Menschen und bzw. oder Maschinen zu den genannten Aufgaben bestimmt den Automatisierungsgrad und die Automatisierungsform der jeweiligen Aufgabe [Sinz99c, 349]. In Abschnitt 2.1.4 wird die Automatisierung betrieblicher Aufgaben näher untersucht.

Aufgabenebene und Aufgabenträgerebene betrieblicher Informationssysteme

Damit Freiheitsgrade bei der Zuordnung von Menschen und bzw. oder Maschinen zu den genannten Aufgaben aufgedeckt und genutzt werden können, wird bei der Analyse und der Gestaltung betrieblicher Informationssysteme zwischen einer Aufgabenebene und einer Aufgabenträgerebene unterschieden [Sinz99c, 349]. Abbildung 2.2 illustriert die beiden Ebenen. Die **Aufgabenebene** umfasst die genannten informationsverarbeitenden Aufgaben, die über Informationsbeziehungen miteinander verbunden sind [FeSi01, 2; Sinz99c, 349]. Auf der **Aufgabenträgerebene** werden Menschen und Maschinen angegeben, die zur Durchführung der informationsverarbeitenden Aufgaben zur Verfügung stehen [FeSi01, 2 f; Sinz99c, 349]. In diesem Zusammenhang stellen Menschen **personelle Aufgabenträger** und Maschinen **maschinelle Aufgabenträger** dar. Beispiele personeller Aufgabenträger eines Informationssystems sind Personen in der Rolle von Sachbearbeitern, Managern oder Datenerfassern [FeSi01, 3]. Maschinelle Aufgabenträger eines Informationssystems sind Rechnersysteme [FeSi01, 3; Sinz99c, 349]. Der Zusammenhang zwischen einem Rechnersystem und einem Anwendungssystem wird in Abschnitt 2.1.5 erläutert. Rechnersysteme sind über ein oder mehrere Kommunikationssysteme untereinander und mit personellen Aufgabenträgern verbunden [FeSi01, 2; Sinz99c, 349]. Die **Kommunikationssysteme** stellen Kommunikationskanäle bereit, die die Informationsbeziehungen zwischen zwei Aufgaben, denen unterschiedliche Aufgabenträger zugeordnet sind, realisieren [FeSi01, 3]. Folglich sind Kommunikationssysteme spezielle Aufgabenträger für Übertragungs- und Speicherungsaufgaben [FeSi01, 3]. Aufgrund der unterschiedlichen Aufgabenträgerarten lassen sich Kommunikationssysteme für die Kommunikation zwischen Rechnern (Computer-Computer-

Kommunikation), für die Kommunikation zwischen Person und Rechner (Mensch-Computer-Kommunikation) und für die Kommunikation zwischen Personen (Mensch-Mensch-Kommunikation) unterscheiden [FeSi01, 4; Sinz99c, 349].

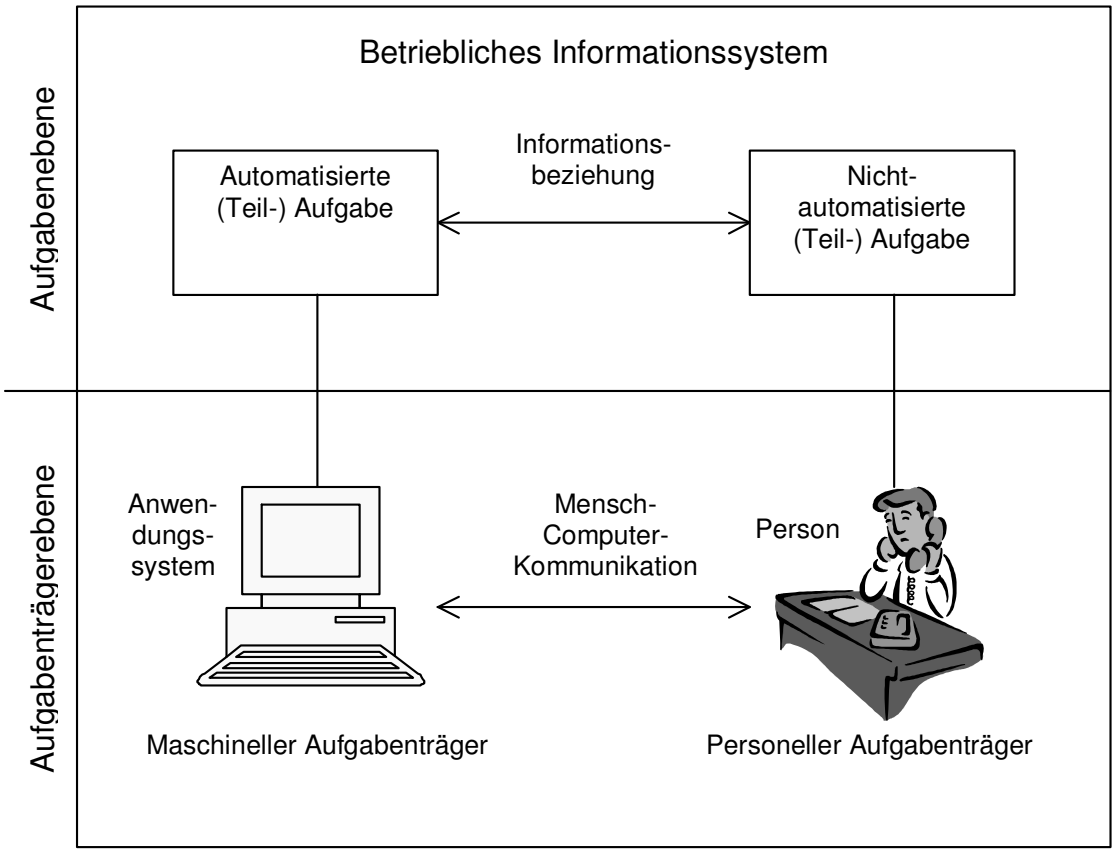


Abbildung 2.2: Betriebliches Informationssystem (in Anlehnung an [FeSi01, 3])

2.1.3 Konzept der betrieblichen Aufgabe

Sowohl die Gesamtaufgabe als auch jede Teilaufgabe eines betrieblichen Informationssystems kann anhand des Aufgabenkonzepts von FERSTL und SINZ [FeSi01, 89 ff] beschrieben werden (vgl. Abbildung 2.3). Danach besteht eine **Aufgabe** in der **Außensicht** aus einem Aufgabenobjekt, einer Menge vorgegebener Sach- und Formalziele, den Vorereignissen, die eine Aufgabendurchführung auslösen, und den Nachereignissen, die aus einer Aufgabendurchführung resultieren [FeSi01, 90]. Zur eindeutigen Abgrenzung zwischen der Spezifikation und der Durchführung einer Aufgabe wird Letzteres auch als **Vorgang** bezeichnet [FeSi01, 55 f]. Das **Aufgabenobjekt** entspricht im vorliegenden Fall den Attributen des betrieblichen Informationssystems, die von der Aufgabe betroffen sind [FeSi01, 90; Fers92, 5]. Die **Sachziele** definieren eine Menge von Zielzuständen des Aufgabenobjekts [Fers92, 6]. Falls alternative Zielzustände erreichbar sind, liegen Freiheitsgrade bei der Aufgabendurchführung vor. Hier grenzen die **Formalziele** die Menge der auszuwählenden Zielzustände ein [FeSi01, 188; Fers92, 6]. Die Außensicht einer Aufgabe abstrahiert folglich von be-

stimmten Aufgabenträgern und von einem bestimmten Verfahren für die Aufgabendurchführung [FeSi01, 90]. So kann die im letzten Abschnitt geforderte Aufdeckung und Nutzung von Freiheitsgraden bei der Zuordnung von Aufgabenträgern zu Aufgaben gewährleistet werden.

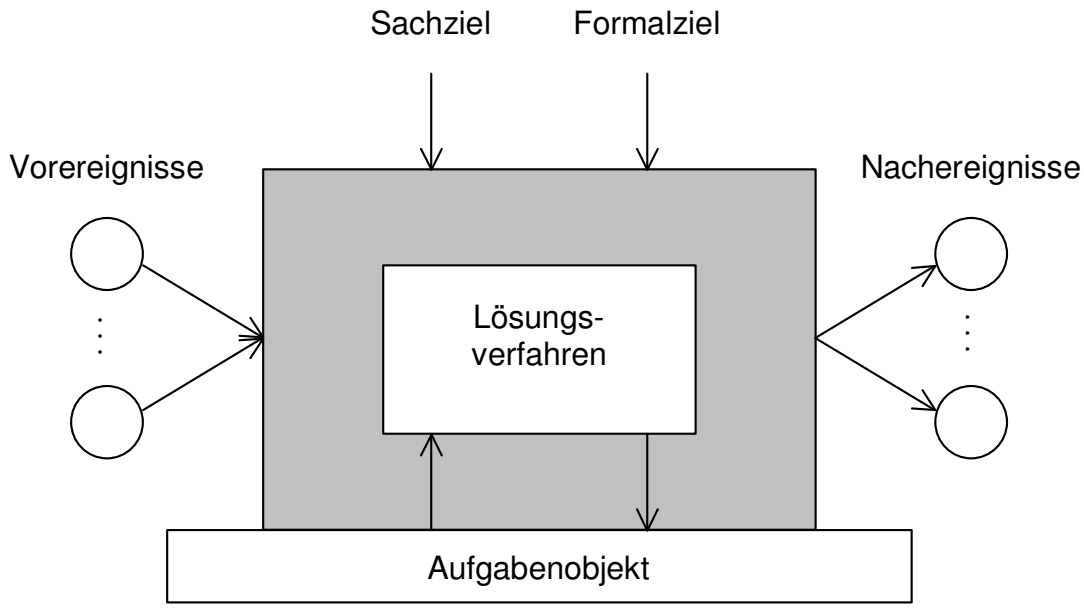


Abbildung 2.3: Struktur einer Aufgabe [FeSi01, 90]

Die **Innensicht einer Aufgabe** beschreibt das Lösungsverfahren zur Erfüllung der Sach- und Formalziele für die gewählten Aufgabenträgertypen und gibt zusätzliche aufgabenträgerspezifische Formalziele an [FeSi01, 95]. Diese **aufgabenträgerspezifischen Formalziele** beziehen sich üblicherweise auf den Ressourcenverbrauch, auf die Durchführungszeiten oder auf aufgabenträgerabhängige Zielerreichungsgrade der übrigen Sach- und Formalziele [FeSi01, 95]. Das **Lösungsverfahren**, dessen Struktur in Abbildung 2.4 dargestellt ist, bestimmt die Aufgabendurchführung, genauer gesagt den **Vorgangstyp**, unter Berücksichtigung der vorgegebenen Sach- und Formalziele. Konkrete Vorgänge zu einem Vorgangstyp werden **Vorgangsinstanzen** genannt. Elemente eines Vorgangstyps sind **Aktionen**, die entweder sequentiell oder parallel auf das Aufgabenobjekt einwirken [FeSi01, 95; Fers92, 7]. Jede Aktion wird durch einen zugehörigen **Aktionsoperator** spezifiziert [Fers92, 7]. Falls ein Aktionsoperator funktional beschreibbar ist, lässt er sich von einem maschinellen Aufgabenträger durchführen. Er ist demzufolge automatisierbar [FeSi01, 95; Fers92, 7]. Anderenfalls muss der Aktionsoperator von einem personellen Aufgabenträger durchgeführt werden. Damit die Aktionen die vorgegebenen Sach- und Formalziele erfüllen können, muss der Aufruf der entsprechenden Aktionsoperatoren in einer bestimmten Reihenfolge erfolgen. Dafür sorgt eine separate **Aktionensteuerung**, die in Verbindung mit den Aktionsoperatoren eine Steuerkette oder einen Regelkreis realisiert [FeSi01, 95; Fers92, 7]. Die Sach- und Formalziele werden der Aktionensteuerung

als Führungsgröße vorgegeben. Die Aktionensteuerung generiert aus diesen Sach- und Formalzielen Stellgrößen in Form von Aufrufen, die an die Aktionsoperatoren als gesteuerte Einheit oder Regelstrecke gerichtet werden (vgl. Abbildung 2.4). Schließlich wirken die Aktionsoperatoren auf das Aufgabenobjekt ein und melden im Falle eines Regelkreises die Reaktionen des Aufgabenobjekts bzw. Aktionenergebnisse an die Aktionensteuerung zurück [Fers92, 7].

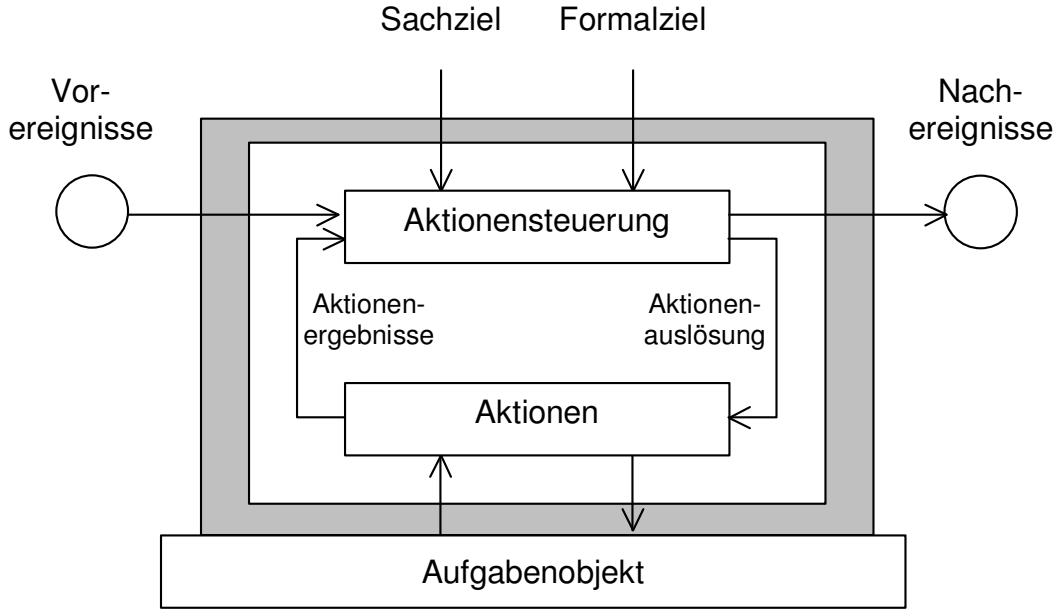


Abbildung 2.4: Struktur eines Lösungsverfahrens [FeSi01, 95]

2.1.4 Automatisierung betrieblicher Aufgaben

Die Zuordnung von Aufgabenträgern zu Aufgaben legt den **Automatisierungsgrad** der Aufgaben fest [FeSi01, 47]. Dabei lassen sich die Automatisierungsgrade vollautomatisiert, teilautomatisiert und nicht-automatisiert unterscheiden [FeSi01, 47 f; Sinz99b, 54]. Falls einer Aufgabe ausschließlich maschinelle Aufgabenträger zugeordnet wurden, ist sie **vollautomatisiert**. Analog dazu ist eine Aufgabe **nicht-automatisiert**, wenn ihr ausschließlich personelle Aufgabenträger zugeordnet wurden. Schließlich sind einer **teilautomatisierten** Aufgabe sowohl personelle als auch maschinelle Aufgabenträger zugeordnet [FeSi01, 47 f; Sinz99b, 54].

Automatisierbarkeitsanalyse

Voraussetzung für die Zuordnung von maschinellen Aufgabenträgern zu einer Aufgabe ist deren **Automatisierbarkeit**. Eine Aufgabe ist grundsätzlich **automatisierbar**, wenn ein Aufgabenmodell und ein Lösungsverfahren angegeben werden können, für die ein maschineller Aufgabenträger verfügbar ist [Sinz99b, 54]. Ob eine grundsätzlich automatisierbare Aufgabe auch tatsächlich automatisiert wird, hängt von der Erfüllung sachlicher Kriterien, wie z. B. Kosten- und Nutzeffekten ab, die in [FeSi01, 103 ff] ausführlich beschrieben sind. Mithilfe der genannten Kriterien lässt

sich jedoch nur feststellen, ob einer Aufgabe ausschließlich ein maschineller oder ein personeller Aufgabenträger zugeordnet werden kann. Über die Möglichkeit einer gemeinschaftlichen Zuordnung von maschinellen und personellen Aufgabenträgern zu einer Aufgabe sagt die Automatisierbarkeitsanalyse nichts aus. Da die Gesamtaufgabe eines betrieblichen Informationssystems üblicherweise teilautomatisiert ist, muss sie soweit zerlegt werden, bis jede resultierende Teilaufgabe entweder als automatisierbar oder als nicht-automatisierbar identifiziert worden ist [FeSi01, 48]. Daraufhin werden automatisierbaren Teilaufgaben maschinelle Aufgabenträger und nicht-automatisierbaren Teilaufgaben personelle Aufgabenträger zugeordnet [FeSi01, 48].

Aufgabenintegration

Aufgrund der Leistungssteigerungen personeller und insbesondere maschineller Aufgabenträger sind auch Aufgabenkomplexe bestimmbar, denen jeweils ein Aufgabenträger zugeordnet wird. Die Abgrenzung dieser Komplexe erfolgt im Hinblick auf eine Verbesserung und Beschleunigung der Aufgabendurchführung [FeSi01, 52]. Die Zuordnung von genau einem Aufgabenträger zu einem Aufgabenkomplex wird **Aufgabenintegration** genannt [FeSi01, 52]. Demnach ist ein Aufgabenkomplex **maschinell integriert**, wenn seine automatisierbaren Aufgaben von genau einem maschinellen Aufgabenträger durchgeführt werden, und **personell integriert**, wenn seine nicht-automatisierbaren Aufgaben von genau einem personellen Aufgabenträger durchgeführt werden [FeSi01, 53]. Darüber hinaus kann ein Aufgabenkomplex auch maschinell und personell integriert sein, wenn seine automatisierbaren Aufgaben von genau einem maschinellen Aufgabenträger und seine nicht-automatisierbaren Aufgaben von genau einem personellen Aufgabenträger durchgeführt werden [FeSi01, 53].

2.1.5 Betriebliches Anwendungssystem

Die in der Einführung des Abschnitts vorgestellte Kurzdefinition des Begriffs „betriebliches Anwendungssystem“ wird nun auf der Grundlage der Ausführungen der letzten Abschnitte weiter konkretisiert. Allerdings werden auch in diesem Abschnitt noch keine Details bezüglich der Struktur und des Verhaltens eines betrieblichen Anwendungssystems betrachtet. Dies erfolgt an den entsprechenden Stellen im weiteren Verlauf der Arbeit.

In Abschnitt 2.1.2 wurden Rechner- und Kommunikationssysteme als maschinelle Aufgabenträger für die zu automatisierenden Aufgaben eines Informationssystems vorgestellt. Vor der Übernahme von Aufgaben werden Rechner- und Kommunikationssysteme als Hardware-Systeme üblicherweise mehrstufig um **Programme** erweitert. Dabei handelt es sich hauptsächlich um Anwendungssoftware und um System-

software [FeSi01, 4; Seib01, 47]. Diese Programme bilden die zu automatisierenden Aufgaben auf die zugrundeliegenden **Rechner- und Kommunikationssysteme** ab [FeSi01, 7]. Ein solches erweitertes System wird betriebliches Anwendungssystem genannt und ist speziell für eine einzelne Aufgabe oder einen Aufgabenbereich eines Informationssystems konzipiert [FeSi01, 4; Seib01, 46]. Demzufolge stellt ein betriebliches Anwendungssystem im Vergleich zu Rechner- und Kommunikationssystemen einen „höheren“ und spezifischeren Aufgabenträger für die zu automatisierende Aufgabe dar [FeSi01, 7].

Ein Anwendungssystem kann gemäß den Ausführungen zur Systemtheorie in Abschnitt 2.1.1 aus der Außensicht und aus der Innensicht betrachtet werden. Dies wird mit dem Modell der Nutzer- und Basismaschine beschrieben, das in Abschnitt 2.2.5.1 vorgestellt wird. Dieses Modell dient wiederum als Beschreibungsrahmen für die Entwicklung von Anwendungssystemen. Weitere Ausführungen hierzu enthält der Abschnitt 2.2.5.2, der das Software-Systemmodell beschreibt.

2.2 Konstruktion betrieblicher Informations- und Anwendungssysteme

Maßgebliche Teilgebiete der Wirtschaftsinformatik sind die Konstruktion, das Management und die Nutzung betrieblicher Informationssysteme [Lehn01, 506; Sinz99f, 803]. Die vorliegende Arbeit bezieht sich auf das Teilgebiet der Konstruktion betrieblicher Informationssysteme.

2.2.1 Konstruktionsproblem

Gemäß den Ausführungen zur Systemtheorie in Abschnitt 2.1.1 entspricht ein Informationssystem einem realen, künstlichen, offenen und dynamischen System. Die Untersuchung von Systemen wird im Allgemeinen anhand einer **Untersuchungssituation** beschrieben [Fers79, 43]. Diese enthält

- eine Beschreibung des **Untersuchungsobjekts** anhand seiner bekannten Systemeigenschaften,
- eine Spezifikation des **Untersuchungsziels**, das sich auf die unbekannten Systemeigenschaften des Untersuchungsobjekts bezieht, und
- eine Anzahl von **Untersuchungsverfahren** zur Erreichung des Untersuchungsziels [Fers79, 43].

Von diesen Untersuchungsverfahren wird mindestens eines durchgeführt und das Ergebnis der Durchführung, **Problemlösung** genannt, anhand des vorgegebenen Untersuchungsziels bewertet [Fers79, 66]. Untersuchungsobjekt und Untersuchungsziel bilden zusammen das **Untersuchungsproblem** [Fers79, 43]. Im Hinblick

auf die bereits bekannten und die noch zu untersuchenden Systemeigenschaften können unterschiedliche Problemtypen identifiziert werden [Fers79, 44]. Eine ausführliche Beschreibung dieser Problemtypen enthält [Fers79, 44 ff]. Im Falle der Konstruktion eines Systems liegt ein **Konstruktionsproblem** vor. Das Untersuchungsobjekt eines Konstruktionsproblems ist ein noch nicht existierendes System, dessen Verhalten postuliert wird [Fers79, 44]. Das zugehörige Untersuchungsziel ist eine Struktur des Systems, die zu dem geforderten Verhalten führt [Fers79, 44]. Entsprechend den obigen Erläuterungen existieren üblicherweise mehrere Strukturen als Lösungen für dieses Problem. Die Menge der möglichen Strukturen kann anhand von Vorgaben sowohl in Bezug auf zu verwendende Systemkomponenten und Teilstrukturen als auch hinsichtlich qualitativer Anforderungen eingeschränkt werden [Fers79, 45; Sinz02b, 1070].

2.2.2 Konstruktionsaufgabe

Unter Berücksichtigung der Ausführungen in Abschnitt 2.1.3 ist das im vorangegangenen Abschnitt dargestellte Konstruktionsproblem auch als Außensicht einer Aufgabe interpretierbar (vgl. Abbildung 2.5). Demnach entspricht das Untersuchungsobjekt dem Aufgabenobjekt und das Untersuchungsziel dem Sachziel der Konstruktionsaufgabe. Die erwähnten Vorgaben zur Einschränkung möglicher Lösungsstrukturen stellen schließlich die Formalziele der Konstruktionsaufgabe dar. Da im vorliegenden Fall die Konstruktionsaufgabe isoliert von anderen Aufgaben betrachtet wird, ist eine Spezifikation von Vor- und Nachereignissen nicht erforderlich.

Konstruktion betrieblicher Informationssysteme als Aufgabe

Somit stellt auch die Konstruktion betrieblicher Informationssysteme eine Aufgabe im Sinne der Ausführungen in Abschnitt 2.1.3 dar. Das postulierte Verhalten wird anhand von funktionalen Anforderungen beschrieben, die durch die Gesamtaufgabe des betrieblichen Informationssystems vorgegeben sind [Sinz02b, 1070]. Die Vorgaben hinsichtlich zu verwendender Systemkomponenten und Teilstrukturen beziehen sich beispielsweise auf Architekturkonzepte, Hard- und Software-Plattformen oder Organisationsprinzipien. Weitere Vorgaben bezüglich einzuhaltender Qualitätsanforderungen können z. B. die Erweiterbarkeit, die Anpassbarkeit oder die Zuverlässigkeit des betrieblichen Informationssystems betreffen [Sinz02b, 1070]. Schließlich erfordert die Durchführung eines Untersuchungs- bzw. Konstruktionsverfahrens einen geeigneten Beschreibungsrahmen, ein Vorgehensmodell und eine entsprechende Werkzeugunterstützung. In der vorliegenden Arbeit werden insbesondere die Beschreibungsrahmen zur Konstruktion betrieblicher Informationssysteme aufgezeigt.

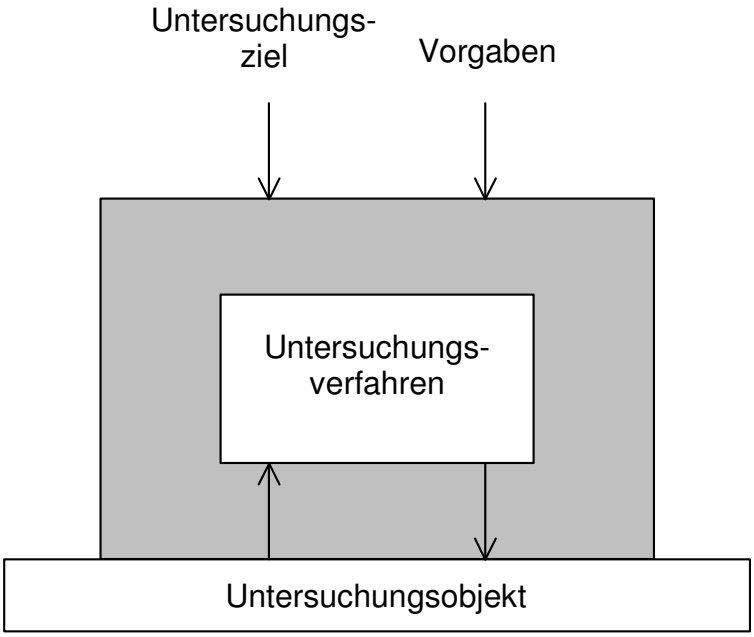


Abbildung 2.5: Untersuchungsaufgabe

Analog zu den Erläuterungen in Abschnitt 2.1.2, unterscheidet man auch bei der Konstruktion betrieblicher Informationssysteme zwischen einer Aufgabenebene und einer Aufgabenträgerebene eines Informationssystems. Somit wird zunächst die Aufgabenstruktur des betrieblichen Informationssystems festgelegt [Sinz02b, 1071]. Dies erfolgt mittels der in Abschnitt 2.1.4 beschriebenen sukzessiven Zerlegung der Gesamtaufgabe des Informationssystems in automatisierbare und nicht-automatisierbare Teilaufgaben. Daran schließt sich die Definition der Struktur der personellen Aufgabenträger und der Struktur der maschinellen Aufgabenträger des betrieblichen Informationssystems an [Sinz02b, 1071 f]. Die Festlegung der Struktur der personellen Aufgabenträger ist für die vorliegende Arbeit nur soweit von Interesse, wie sie für die Gestaltung von Kommunikationsbeziehungen zu maschinellen Aufgabenträgern erforderlich ist. Zur Festlegung der Struktur der maschinellen Aufgabenträger gehören unter anderem die Spezifikation, der Entwurf und die Implementierung der Anwendungssysteme des Informationssystems und ihrer Kommunikationsbeziehungen [Sinz02b, 1072]. Schließlich werden die personellen und maschinellen Aufgabenträgerstrukturen der Aufgabenstruktur des Informationssystems zugeordnet [Sinz02b, 1072]. Dies geschieht anhand der in Abschnitt 2.1.4 erläuterten Automatisierung betrieblicher Aufgaben. Dabei werden auch notwendige Kommunikationsbeziehungen zwischen Personen und Anwendungssystemen aufgedeckt, für die geeignete Mensch-Computer-Kommunikationsschnittstellen zu spezifizieren sind [Sinz02b, 1072].

Die dargestellte Reihenfolge der durchzuführenden Aufgaben bei der Konstruktion betrieblicher Informationssysteme bestimmt den weiteren Aufbau des Abschnitts 2.2.

Nach einer kurzen Einführung in die Grundlagen der Modellierung betrieblicher Informationssysteme wird die Festlegung der Aufgabenstruktur betrieblicher Informationssysteme anhand von Modellen kurz beleuchtet. Schließlich wird in Abschnitt 2.2.5 die Entwicklung betrieblicher Anwendungssysteme zur Festlegung der Struktur der maschinellen Aufgabenträger näher untersucht.

2.2.3 Modelltheoretische Grundlagen

Die Konstruktion eines Informationssystems erfolgt üblicherweise auf Basis eines Modells des Informationssystems, da auf diesem Weg die hohe Komplexität, die ein Informationssystem im Allgemeinen besitzt, beherrschbar gemacht werden kann [FeSi01, 119; LeHM95, 79; Hein02, 1046; Hamm99, 6]. Folglich gehören Modelle zu den wichtigsten Hilfsmitteln in der Wirtschaftsinformatik [FeSi01, 119; Hamm99, 6]. Trotzdem existiert in dieser Wissenschaftsdisziplin kein einheitlicher Modellbegriff [Hamm99, 6; LeHM95, 73; Schu01, 8]. Das liegt unter anderem daran, dass die Wirtschaftsinformatik zwar ein eigenständiges, aber auch ein interdiziplinäres Fachgebiet ist und sich die Definitionen des Modellbegriffs in den beteiligten Wissenschaftsdisziplinen unterscheiden [LeHM95, 27 f].

Traditionellerweise versteht man unter einem **Modell** das vereinfachte Abbild der Realität oder eines Realitätsausschnitts (vgl. z. B. [LeHM95, 27; Hein02, 1046; BöFu02, 176; Hamm99, 10 f]). Die Vereinfachung besteht darin, dass nur diejenigen Sachverhalte der Realität bzw. des Realitätsausschnitts im Modell Berücksichtigung finden, die hinsichtlich der Ziele des Modellnutzers relevant erscheinen [Hamm99, 7]. Auf diese Weise wird die Komplexität der Realität bzw. des Realitätsausschnitts in Bezug auf bestimmte Ziele des Modellnutzers beherrschbar gemacht. Häufig werden die Begriffe „Modell“ und „Konzept“ synonym verwendet. Zur besseren Unterscheidung der beiden Begriffe wird üblicherweise der Begriff „**Konzept**“ als vereinfachtes Vorbild und der Begriff „Modell“ als vereinfachtes Nachbild der Realität oder eines Realitätsausschnitts verstanden [LeHM95, 83; Hein02, 1046]. Abbildung 2.6 veranschaulicht diesen Zusammenhang. Die oben aufgeführte Definition eines Modells lässt die Projektionsrichtung jedoch offen, so dass ein Modell im vorliegenden Fall sowohl ein Vorbild als auch ein Nachbild sein kann. Der Begriff „Konzept“ wird in Abschnitt 3.4 noch einmal aufgegriffen und für die Arbeit weiter konkretisiert.

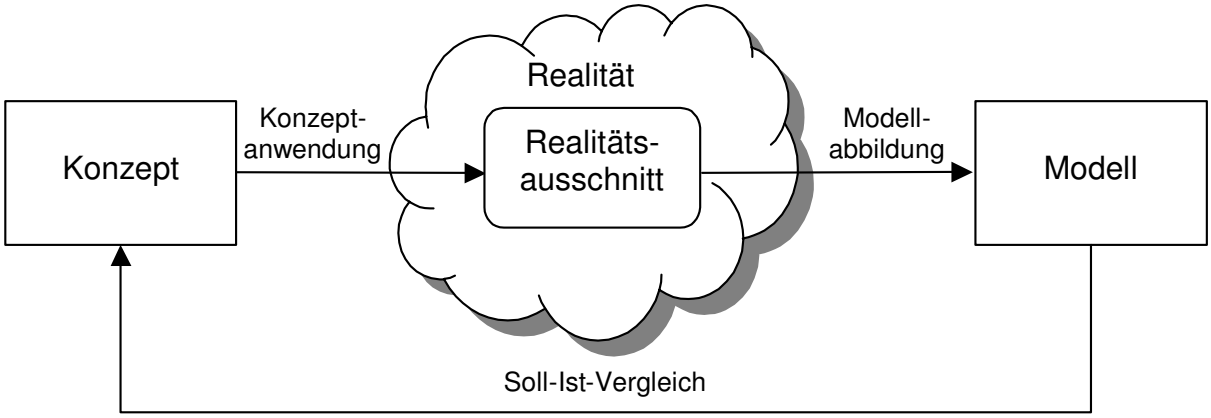


Abbildung 2.6: Modell, Konzept und Realität (in Anlehnung an [Hein02, 1046])

Das vereinfachte Abbild der Realität oder eines Realitätsausschnitts entsteht aufgrund der Anwendung einer Abbildungsvorschrift auf die Realität oder den Realitätsausschnitt. Damit die Ziele des Modellnutzers erfüllt werden können, wird zumindest eine Ähnlichkeit zwischen der Realität oder dem Realitätsausschnitt und dem Abbild gefordert [LeHM95, 27; Hein02, 1046; Hamm99, 10]. In diesem Zusammenhang wird zwischen der strukturellen Ähnlichkeit, der funktionellen Ähnlichkeit und der Verhaltensähnlichkeit unterschieden [LeHM95, 27; Hamm99, 10]. Diese Ähnlichkeit wird durch die Verwendung homomorpher Abbildungen gewährleistet [FeSi01, 18 f]. Eine Gleichheit der Realität oder eines Realitätsausschnitts und des Modells erreicht man durch die Verwendung einer isomorphen Abbildung [LeHM95, 79; HeRo98, 285; Hamm99, 12]. Dies widerspricht jedoch dem Zweck der Modellbildung, der gerade in der Vereinfachung der Komplexität der Realität bzw. des Realitätsausschnitts in Bezug auf bestimmte Untersuchungs- oder Gestaltungsziele besteht.

Abbildtheoretischer Modellbegriff

Der vorgestellte traditionelle Modellbegriff entspricht in der Modelltheorie dem **abbildtheoretischen Modellbegriff** [Hamm99, 9 f; LeHM95, 29]. „Kerngedanke des abbildtheoretischen Modellbegriffs ist die Repräsentationsfunktion des Modells in bezug auf das in der Realität vorliegende Erkenntnisobjekt“ [Hamm99, 10]. Ein häufig genannter Kritikpunkt des abbildtheoretischen Modellbegriffs ist, dass für die Modellbildung eine objektiv wahrnehmbare Wirklichkeit vorausgesetzt wird [LeHM95, 29; Hein02, 1046 f; Hamm99, 16]. Das würde bedeuten, dass ein Original von jedem denkbaren Subjekt in ein identisches Modell abgebildet werden kann [Hamm99, 16]. Dies ist jedoch nachgewiesenermaßen nicht möglich.

Konstruktivistischer Modellbegriff

Deswegen entwickelte HAMMEL [Hamm99] einen **konstruktivistischen Modellbegriff**, der den traditionellen abbildtheoretischen Modellbegriff um den Schritt der subjektbezogenen Realitätswahrnehmung erweitert [Hamm99, 27 f]. Demnach steht das

Subjekt in einer Kontextbeziehung zur Realität (vgl. Abbildung 2.7). Im Rahmen dieser Kontextbeziehung perzipiert das Subjekt die Realität und interpretiert sie zu einem so genannten Modellobjekt. Von diesem Modellobjekt konstruiert das Subjekt schließlich zielbezogen ein Abbild [Hamm99, 27 f].

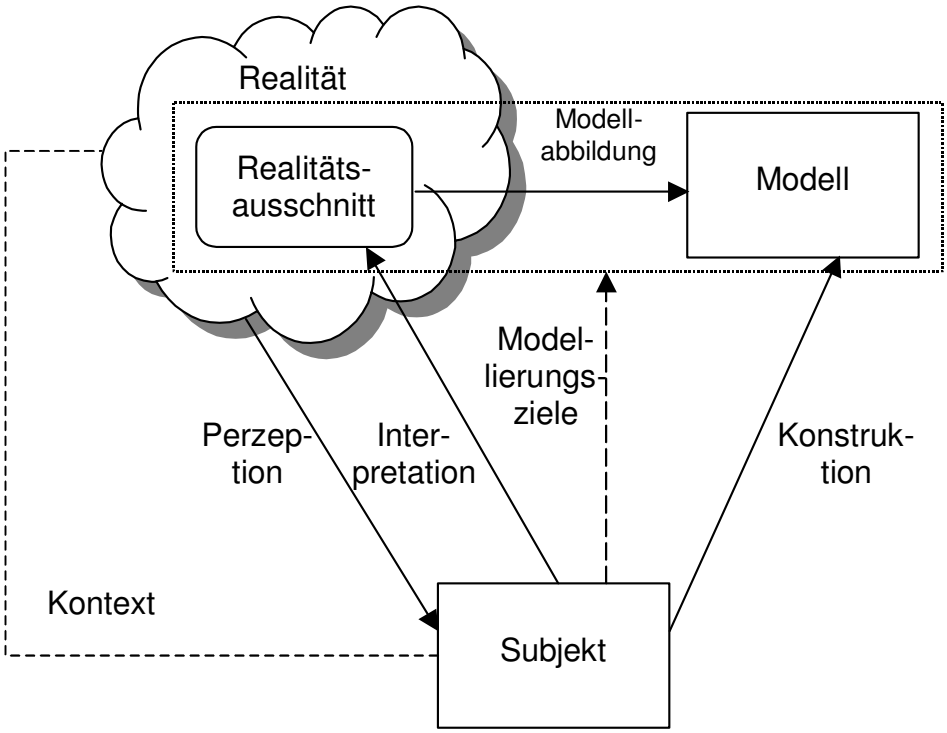


Abbildung 2.7: Konstruktivistischer Modellbegriff (in Anlehnung an [Hamm99, 27])

Das dieser Arbeit zugrunde gelegte Modellverständnis präzisiert das bisher beschriebene allgemeine Modellverständnis. Danach umfasst ein Modell (vgl. Abbildung 2.8) [Sinz01a, 311; Sinz96, 125; FeSi01, 120 f]:

- Ein **Objektsystem** als Ursystem,
- ein **Modellsystem** als Bildsystem und
- eine **Abbildungsvorschrift**, die die Komponenten des Objektsystems auf die Komponenten des Modellsystems zielorientiert abbildet.

Beim Objektsystem handelt es sich in diesem Zusammenhang meistens um einen Ausschnitt eines betrieblichen Systems, dem Informationssystem, und seiner Umwelt. Das Modellsystem ist dagegen üblicherweise ein formales oder semi-formales System. Die Modellbildung zielt auf die Unterstützung der Analyse und der Gestaltung eines betrieblichen Informationssystems sowie der Entwicklung seiner automatisierten Teilsysteme, der betrieblichen Anwendungssysteme [Sinz01a, 311; Sinz96, 125; FeSi01, 120 f; Sinz99d; 458]. Mit der Forderung nach **Struktur- und Verhaltenstreue** des Modellsystems in Bezug auf das Objektsystem wird der geforderten Ähnlichkeit von Objektsystem und Modellsystem Rechnung getragen [Sinz01a, 312;

FeSi01, 18 f; Sinz99d, 458 f]. Da das Objektsystem einem realen System entspricht, stellt bereits die Erfassung des Objektsystems eine Modellbildung dar [Sinz01a, 311; Hamm99, 51; Sinz99d, 459]. Somit können die Erfassung des Objektsystems und die eigentliche Modellbildung nicht eindeutig voneinander getrennt werden. Das bedeutet, dass die Modellbildung wesentlich von den Fähigkeiten des modellbildenden Subjekts bezüglich der Wahrnehmung und der Interpretation des Objektsystems sowie bezüglich der Kreativität bei der Erstellung des Modellsystems abhängt [FeSi01, 121; Sinz99d, 459; Sinz01a, 312].

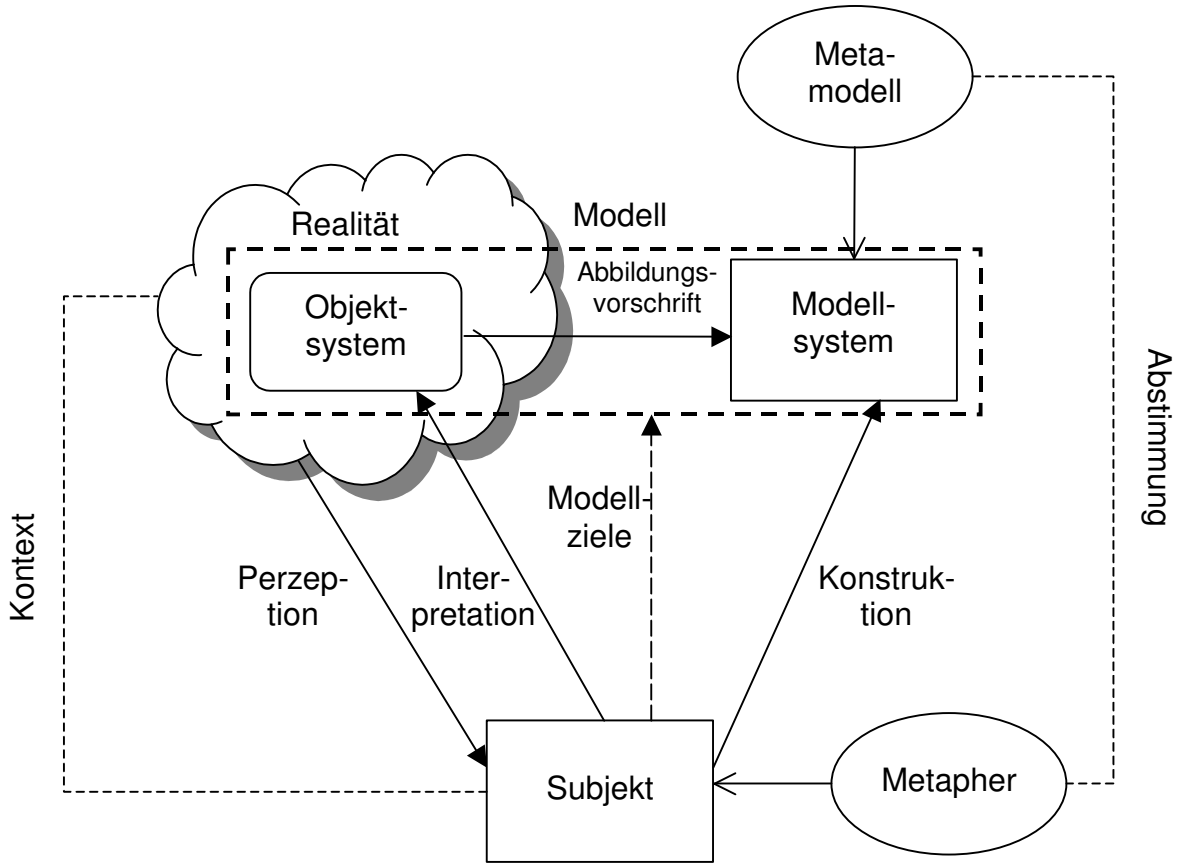


Abbildung 2.8: Zugrunde gelegter Modellbegriff [Sinz01a; Sinz96; FeSi01; Hamm99]

Modellierungsansatz

Damit mit der Modellbildung, auch Modellierung genannt, trotzdem die Ziele des Modellnutzers erfüllt werden können, ist ein geeigneter Beschreibungsrahmen erforderlich, der einerseits die Erfüllung der geforderten Struktur- und Verhaltenstreue des Modellsystems in Bezug auf das Objektsystem unterstützt und andererseits ein Begriffssystem für die Spezifikation des Modellsystems zur Verfügung stellt [FeSi01, 121]. Ein solcher Beschreibungsrahmen wird **Modellierungsansatz** genannt und besteht aus einer Metapher und einem mit der Metapher abgestimmten Metamodell [FeSi01, 121].

Metapher

Unter einer **Metapher** wird im Allgemeinen ein sprachlicher Ausdruck verstanden, der „eine aus einem bestimmten Bedeutungszusammenhang stammende, bildhafte Vorstellung auf einen anderen Bedeutungszusammenhang anwendet“ [FeSi01, 121]. Im vorliegenden Zusammenhang wird eine Metapher primär als kognitives Instrument zur Erfassung und Interpretation der Realität eingesetzt [Hamm99, 40 ff; HeRo98, 352]. Falls eine Metapher, die von einem Modellierer zur Modellbildung gewählt wurde, offengelegt wird, trägt dies zum besseren Verständnis des Modells bei [Hamm99, 42]. Ferner kann mit einer Metapher auch die Struktur- und Verhaltenstreue des Modellsystems in Bezug auf das Objektsystem überprüft werden. Hierfür überträgt man die Metapher, die vom Modellierer zur Erfassung des Realität und deren Interpretation zu einem Objektsystem verwendet wurde, auf die Spezifikation des Modellsystems [FeSi01, 121 f].

Metamodell

Ein **Metamodell** bestimmt das Begriffssystem für die Spezifikation des Modellsystems und muss der gewählten Metapher entsprechen [FeSi01, 122]. Mit einem Metamodell können im Allgemeinen mehrere Modellsysteme konstruiert werden, wobei jedes Modellsystem eine Extension des zugehörigen Metamodells darstellt [FeSi01, 124]. Somit ist es den Modellsystemen hierarchisch übergeordnet und definiert das Spektrum an Darstellungsmöglichkeiten für die Modellsysteme [LeHM95, 82]. Ein Metamodell bestimmt [FeSi01, 122]

- die verfügbaren Arten von Modellbausteinen,
- die Arten von Beziehungen zwischen den Modellbausteinen,
- die Regeln für die Kombination von Modellbausteinen und Beziehungen sowie
- die Bedeutung der Modellbausteine und Beziehungen.

Falls ein Metamodell zumindest semiformal, z. B. grafisch beschrieben ist, kann die **Konsistenz** und die **Vollständigkeit** des Modellsystems in Bezug auf das Metamodell überprüft werden [FeSi01, 123]. Leider werden für viele Modellierungsansätze keine oder nur informal beschriebene Metamodelle angegeben, so dass eine Überprüfung auf Konsistenz und Vollständigkeit der Modellsysteme, die mit diesen Modellierungsansätzen erzeugt wurden, nicht möglich ist [Sinz96, 128f].

Abbildung 2.8 veranschaulicht die bisher erläuterten Zusammenhänge zwischen Metamodell und Metapher sowie zwischen Metamodell, Metapher und Modell.

Meta-Metamodell

Analog zur Spezifikation eines Modellsystems wird auch für die Spezifikation eines Metamodells ein Begriffssystem benötigt, von dem das Metamodell eine Extension

darstellt und das dem Metamodell folglich hierarchisch übergeordnet ist. Ein solches Begriffssystem für Metamodelle wird **Meta-Metamodell** genannt. Es definiert, analog zu einem Metamodell,

- die verfügbaren Arten von Meta-Modellbausteinen,
- die Arten von Meta-Beziehungen zwischen den Meta-Modellbausteinen,
- die Regeln für die Kombination von Meta-Modellbausteinen und Meta-Beziehungen und
- die Bedeutung der Meta-Modellbausteine und Meta-Beziehungen.

Ein mögliches Meta-Metamodell ist in Abbildung 2.9 dargestellt. Es beschreibt so genannte Meta-Objekttypen, die durch Meta-Beziehungen miteinander verknüpft werden können. Als Meta-Beziehungen stehen Generalisierungsbeziehungen (*ist_ein*-Beziehung), Assoziationsbeziehungen (*verbindet*-Beziehung) und Attribut-Zuordnungsbeziehungen (*besitzt*-Beziehung) zur Verfügung. Außerdem können jeder Meta-Beziehung zwei Kardinalitäten zugeordnet werden, die sowohl die maximale als auch die minimale Anzahl der durch die Meta-Beziehung verknüpften Meta-Objekttypen festlegen [FeSi01, 123 f; Sinz96, 129]. Dieses Meta-Metamodell ist selbstbeschreibend, das heisst, es definiert sein eigenes Begriffssystem.

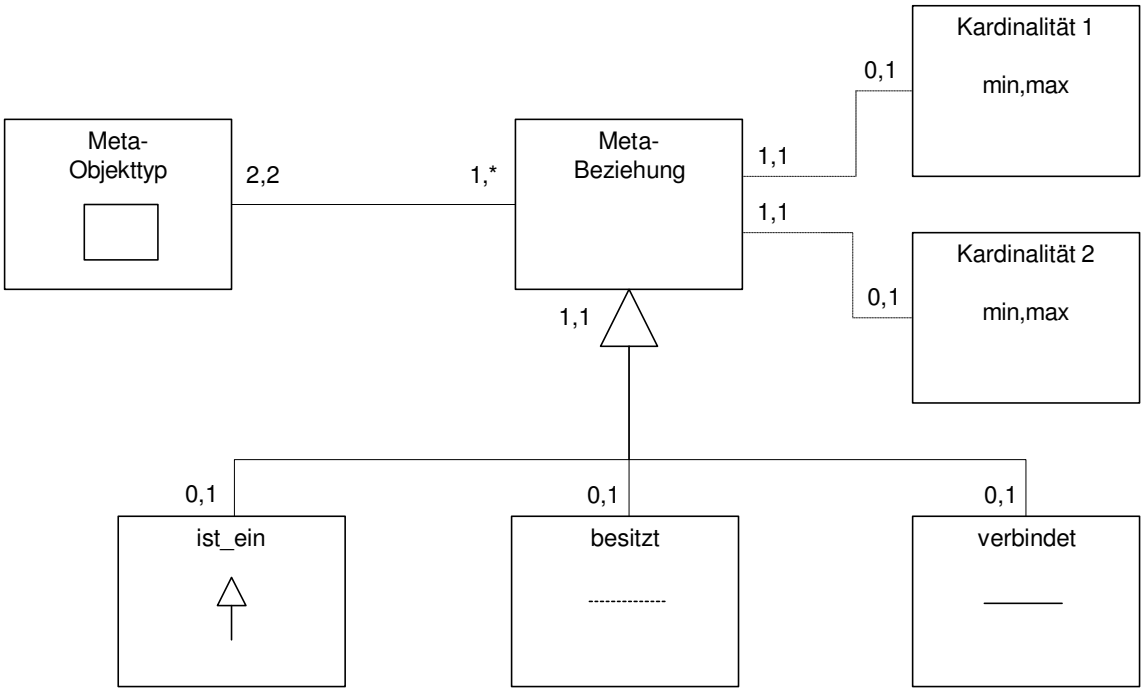


Abbildung 2.9: Meta-Metamodell (in Anlehnung an [FeSi01, 124])

Meta-Ebenen von Modellsystemen

Wie erwähnt, stehen das Modellsystem und das Metamodell sowie das Metamodell und das Meta-Metamodell in hierarchischen Beziehungen zueinander. Das Modell-

system ist eine gültige Extension des Metamodells und das Metamodell ist eine gültige Extension des Meta-Metamodells. Ferner kann das Modellsystem weiter unterteilt werden in ein Schema des Modellsystems und mehrere hierarchisch untergeordnete Ausprägungen des Modellsystems. Diese Unterteilung wird beispielsweise in der Datenmodellierung verwendet, indem auf der Schemaebene Datenobjekttypen einschließlich zugehöriger Beziehungen und auf der Ausprägungsebene konkrete Datenobjekte und zugehörige Beziehungen spezifiziert werden. Zusammenfassend gesehen bilden Meta-Metamodell, Metamodell, Schema und Ausprägungen aufeinanderfolgende Ebenen derselben Hierarchie (vgl. Abbildung 2.10). Diese Ebenen werden auch als **Meta-Ebenen** von Modellsystemen oder **Abstraktionsebenen** der Modellierung bezeichnet [FeSi01, 123].

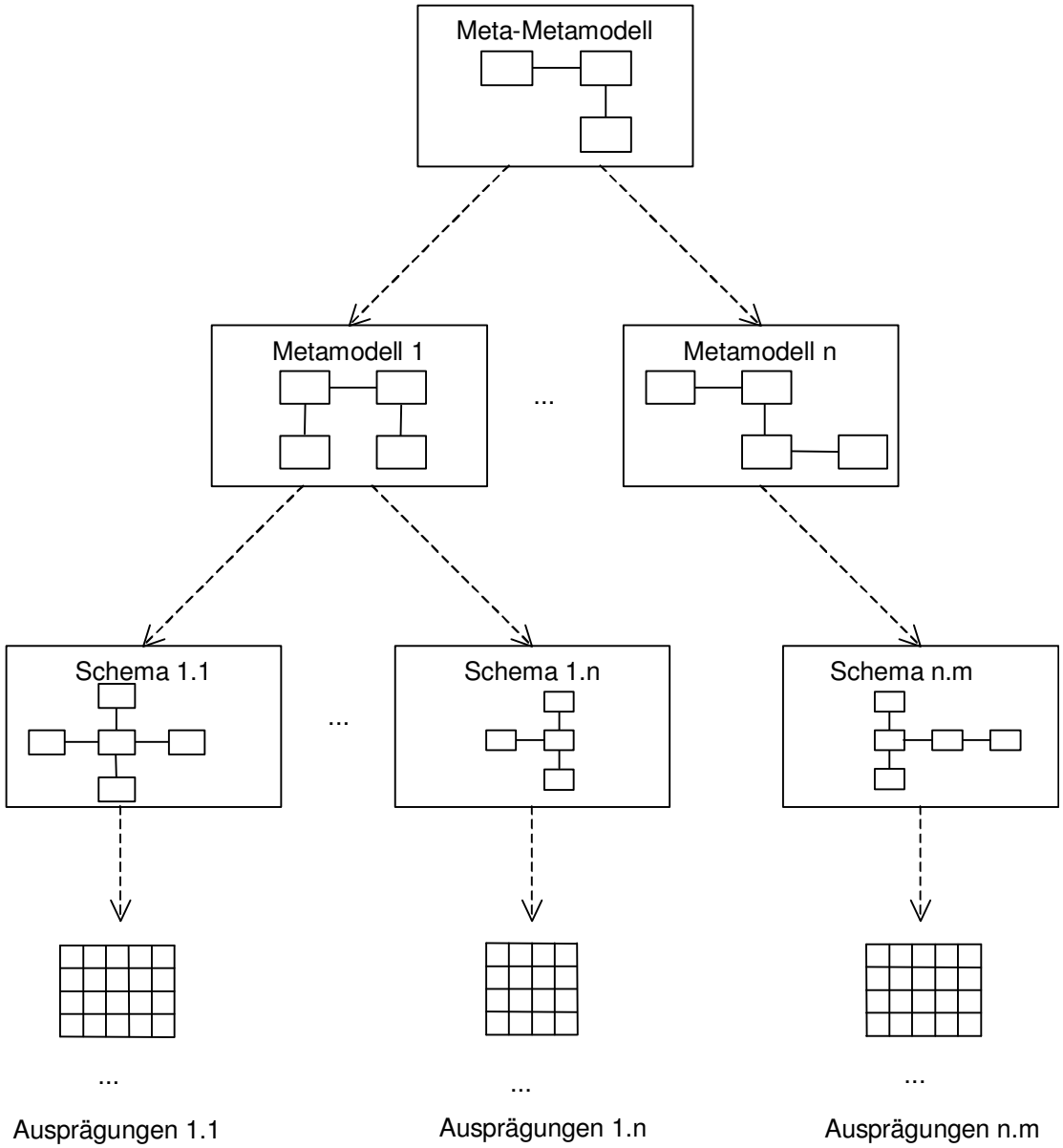


Abbildung 2.10: Meta-Metamodell, Metamodell, Schema und Ausprägungen

Zur Modellierung betrieblicher Informationssysteme existieren mehrere Modellierungsansätze, deren Eignung für jede Einsatzsituation festgestellt werden muss. Grundsätzlich können Modellierungsansätze anhand der Merkmale Modellierungsreichweite, Modellierungszweck und Modellumfang charakterisiert werden [Sinz01b, 313].

Modellierungsreichweite

Die **Modellierungsreichweite** beschreibt die gewählte Abgrenzung des Objektsystems [Sinz01b, 313]. In Bezug auf die Modellierung betrieblicher Informationssysteme entspricht dies entweder dem betrieblichen Informationssystem oder dem betrieblichen Anwendungssystem.

Modellierungszweck

Zur Bestimmung des Modellierungszwecks kann im Wesentlichen unterschieden werden zwischen Beschreibungs-, Erklärungs- und Gestaltungsmodellen [Sinz01b, 313; LeHM95, 37 ff; Hamm99, 8 f]. Ein **Beschreibungsmodell** gibt einen Sachverhalt der Realität in systematisierter und möglichst leicht verständlicher Form wieder [Hamm99, 8; LeHM95, 37]. Es kann entweder ein Vorbild oder ein Nachbild sein [Hamm99, 8]. Ein Beschreibungsmodell gibt allerdings nicht an, weshalb und auf welche Weise ein abzubildender Sachverhalt entstanden ist [Hamm99, 8]. Beispiele für Beschreibungsmodelle sind Organigramme oder Landkarten [LeHM95, 37]. Ein **Erklärungsmodell** basiert auf einem Beschreibungsmodell und erweitert es um Aussagen über die Zusammenhänge und Gesetzmäßigkeiten des abzubildenen Sachverhalts [LeHM95, 38; Hamm99, 8]. Dadurch kann der abzubildende Sachverhalt grundlegend erläutert werden [Hamm99, 8; LeHM95, 37]. Zu den Erklärungsmodellen gehören auch kausale Modelle, die die Ursache-Wirkungsbeziehungen des abzubildenden Sachverhalts darstellen, und Prognosemodelle, die der Vorhersage der zukünftigen Entwicklung des abzubildenden Sachverhalts dienen [LeHM95, 38]. Ein Beispiel für ein Erklärungsmodell ist das Ertragsgesetz in der betriebswirtschaftlichen Produktionstheorie [Hamm99, 8]. Ein **Gestaltungsmodell** basiert wiederum auf einem Erklärungsmodell und erweitert es um erforderliche Einflussgrössen zur Erfüllung bestimmter Aufgaben oder Anforderungen [LeHM95, 38]. Zu den Einflussgrössen gehören beispielsweise Ansatzpunkte für ein steuerndes Eingreifen in den abzubildenden Sachverhalt oder für die Gestaltung des abzubildenden Sachverhalts [LeHM95, 38 f]. Eine häufige Form von Gestaltungsmodellen sind normative bzw. präskriptive Modelle [LeHM95, 39]. Sie erweitern ein Erklärungsmodell um eine Sinn-, Zweck-, oder Zielkomponente [LeHM95, 39]. Somit kann ein gewünschter Zielzustand des abzubildenden Sachverhalts bzw. der Weg dorthin festgelegt werden [LeHM95, 39]. Weitere Formen von Gestaltungsmodellen sind Entscheidungs- und Planungsmodelle [LeHM95, 39]. Auch sie erweitern ein Erklärungsmodell um eine

Zielkomponente [LeHM95, 39; Hamm99, 8]. Dadurch können Modellnutzer, die die Zielkomponente vorgegeben haben, bei der Auswahl eines Optimums aus vorliegenden Alternativen oder bei der Erreichung der vorgegebenen Ziele unterstützt werden [LeHM95, 39]. Beispielsweise sind Optimierungsmodelle der linearen Programmierung häufig verwendete Entscheidungsmodelle [Hamm99, 9].

Modellumfang

Mit dem **Modellumfang** werden die im Modellsystem erfassten Aspekte des Objektsystems beschrieben [Sinz01b, 313]. Mögliche Aspekte betrieblicher Informationssysteme sind beispielsweise Geschäftsprozesse, Daten- oder Funktionsstrukturen.

2.2.4 Modellierung der Aufgabenstruktur betrieblicher Informationssysteme

Die Aufgabenstruktur betrieblicher Informationssysteme wird anhand eines Modells der Aufgabenstruktur festgelegt. Zur Erstellung des Modells liegen eine Vielzahl von Modellierungsansätzen vor, deren Entwicklung einem evolutionären Prozess unterworfen ist [Sinz96, 125]. Die Modellierungsansätze stellen Beschreibungsrahmen zur Modellierung der Aufgabenstruktur dar. In Abschnitt 2.2.4.2 werden Klassen von Modellierungsansätzen vorgestellt, deren praktische Relevanz nachgewiesen ist. Als Voraussetzung für das Verständnis der gewählten Klassifizierung werden im Folgenden zunächst unterschiedliche Sichten auf die Aufgabenebene und die Aufgabenträgerebene eine betrieblichen Informationssystems vorgestellt.

2.2.4.1 Modellierungsrelevante Sichten

Wie in Abschnitt 2.2.3 erläutert, können Modellierungsansätze allgemein anhand der Merkmale Modellierungsreichweite, Modellierungszweck und Modellumfang charakterisiert werden. Im vorliegenden Fall sind die Ausprägungen der Merkmale Modellierungsreichweite und Modellierungszweck für alle Modellierungsansätze gleich. Die Modellierungsreichweite umfasst das gesamte betriebliche Informationssystem und der Modellierungszweck bezieht sich auf Beschreibungs- und Gestaltungsmodelle. Folglich unterscheiden sich die Modellierungsansätze nur im Modellumfang. Da die in Betracht gezogenen Modellierungsansätze für die Aufgabenstruktur von Informationssystemen keine vollständige Modellierung der in Abschnitt 2.1.3 erläuterten Aufgabenmerkmale unterstützen, wird der Modellumfang anhand von unterschiedlichen Sichten auf die Aufgabenmerkmale beschrieben [Sinz02b, 1073; FeSi01, 127]. Ein Modellierungsansatz bezieht sich somit auf eine oder mehrere dieser Sichten [FeSi01, 127].

Es können drei statische Sichten und eine dynamische Sicht unterschieden werden. Die drei statischen Sichten beziehen sich auf die Aufgabenspezifikation, die dynami-

sche Sicht auf die Aufgabendurchführung. Zu den statischen Sichten gehören demnach [Sinz02b, 1074]

- die **Datensicht** zur Beschreibung der Aufgabenobjekte eines Informationssystems,
- die **Funktionssicht** zur Spezifikation der Aktionsoperatoren eines Informationssystems, die auf die Aufgabenobjekte des Informationssystems einwirken, und
- die **Interaktionssicht** zur Bestimmung der Kommunikationskanäle zwischen den Aktionsoperatoren eines Informationssystems.

Als einzige dynamische Sicht ist die **Vorgangssicht** zu nennen, mit der die zielorientierten Durchführungen der Aktionsoperatoren auf den Aufgabenobjekten eines Informationssystems beschrieben werden können [Sinz02b, 1074]. Für die Spezifikation der Aufgabenziele eines Informationssystems ist keine eigene Sicht vorgesehen. Diese werden aus Gründen der Komplexitätsbewältigung üblicherweise auf einer separaten Modellierungsebene beschrieben (siehe dazu die Ausführungen in Abschnitt 2.3 und Abschnitt 2.4.1).

Die Modellierungsansätze für die Aufgabenstruktur von Informationssystemen können nun danach klassifiziert werden, welche der genannten Sichten sie unterstützen. Falls zu einem Modellierungsansatz die entsprechende Metapher und das entsprechende Metamodell angegeben sind, bestimmen diese die vom Modellierungsansatz unterstützten Sichten. Viele Modellierungsansätze lassen nur die Modellierung einzelner Sichten zu. Zur Modellierung aller genannten Sichten werden die Modellierungsansätze üblicherweise miteinander kombiniert [Sinz02b, 1074]. Dies führt in der Regel zu Problemen, da einerseits die Metamodelle der verschiedenen Modellierungsansätze nicht immer integrierbar und andererseits die Metaphern der verschiedenen Modellierungsansätze häufig inkompatibel sind [Sinz02b, 1074 ff; FeSi01, 127 f]. Dagegen unterstützen umfassende Modellierungsansätze die abgestimmte Modellierung mehrerer Sichten. Dabei werden die einzelnen Sichten formal als Projektionen auf das Metamodell des Modellierungsansatzes spezifiziert [Sinz99d, 459; Sinz96, 128]. Dies wird in Abbildung 2.11 veranschaulicht.

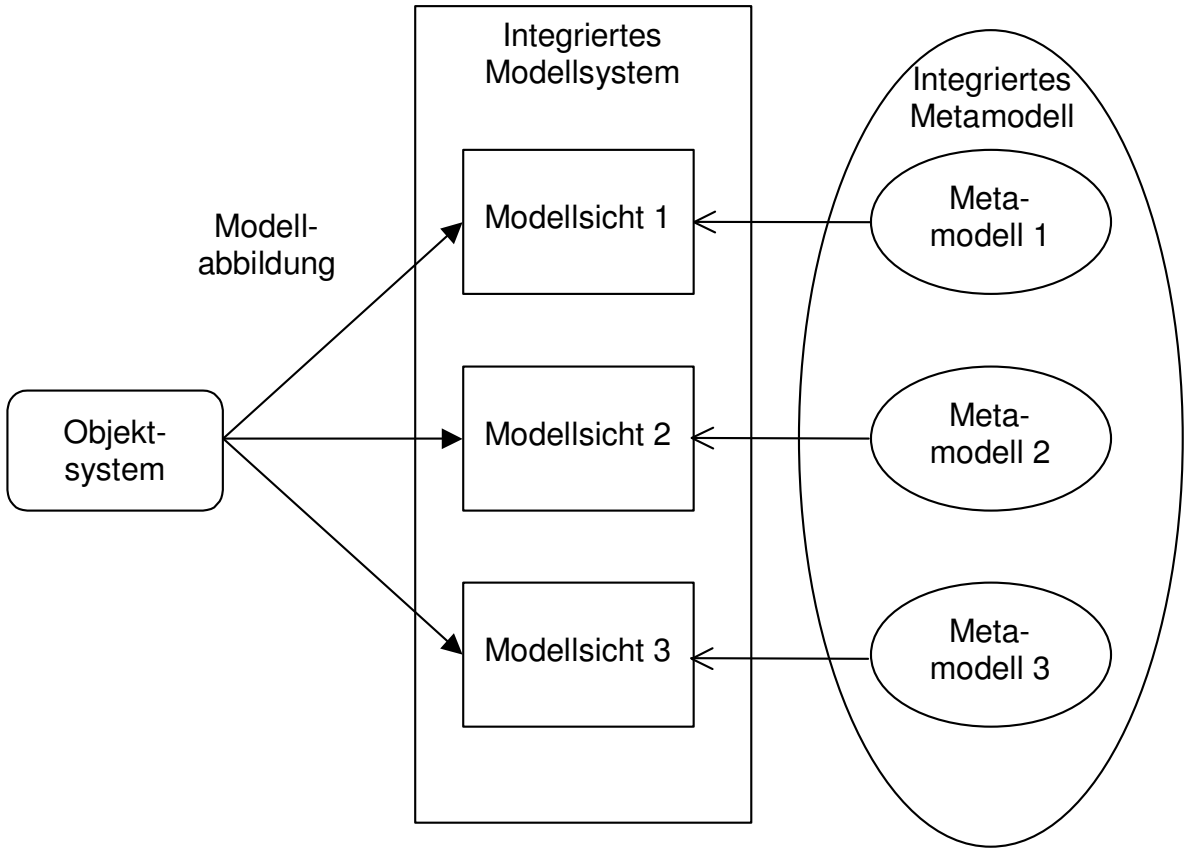


Abbildung 2.11: Abgestimmte Modellierung mehrerer Sichten [Sinz02c, 1-18]

Im folgenden Abschnitt werden fünf Klassen von Modellierungsansätzen für die Aufgabenstruktur von Informationssystemen kurz vorgestellt und dabei die jeweils unterstützten Sichten bestimmt.

2.2.4.2 Modellierungsansätze

Die erste Klasse von Modellierungsansätzen umfasst die **Ansätze der Funktionszerlegung**. Dabei werden Funktionen eines Informationssystems über mehrere Hierarchiestufen in Teilfunktionen zerlegt und Schnittstellen der (Teil-) Funktionen zu anderen (Teil-) Funktionen und zur Umgebung des Informationssystems festgelegt. Die von einer Funktion zu bearbeitenden Daten werden separat als Eingangsdaten, Ergebnisdaten oder Speicherdaten der entsprechenden Funktion spezifiziert [Sinz02b, 1074; Sinz99d, 459]. Demnach unterstützen die Modellierungsansätze dieser Klasse nur die Funktionssicht.

In einer weiteren Klasse werden die **Datenflussansätze** zusammengefasst. Anhand dieser Ansätze wird die Aufgabenstruktur eines Informationssystems in Form von Datenflüssen beschrieben, die durch so genannte Aktivitäten transformiert werden. Die Aktivitäten lassen sich, analog zu den Funktionen in der Funktionsmodellierung, hierarchisch zerlegen. Ferner können die relevante Umgebung des Informationssystems und die temporäre Pufferung von Datenflüssen spezifiziert werden [Sinz02b,

1075; Sinz99d, 459; FeSi01, 125]. Somit beziehen sich die Modellierungsansätze dieser Klasse primär auf die Interaktionssicht.

Die Klasse der **Datenmodellierungsansätze** gehört zu den bekannteren Klassen. Mit diesen Ansätzen wird ausschließlich die Datenstruktur eines Informationssystems modelliert. Eine solche Datenstruktur besteht im Allgemeinen aus Datenobjekttypen einschließlich zugeordneter Attribute und aus verschiedenartigen Beziehungen zwischen den Datenobjekttypen [Sinz02b, 1074; Sinz99d, 459; FeSi01, 125]. Daher unterstützen die Modellierungsansätzen dieser Klasse nur die Datensicht.

Die bisher genannten Modellierungsansätze unterstützen jeweils nur eine Sicht auf die Aufgabenebene eines Informationssystems, wobei die Vorgangssicht von keiner der genannten Klassen berücksichtigt wird. Diese Ansätze waren für die Modellierung der Aufgabenstruktur eines Informationssystems bis in die 1990er Jahre weit verbreitet. Zur Modellierung mehrerer Sichten wurden sie üblicherweise kombiniert, was zu den im vorangegangenen Abschnitt erwähnten Problemen führte. Seit den 1990er Jahren werden sie jedoch zunehmend von zwei weiteren Klassen von Modellierungsansätzen verdrängt, die gemeinsam eine integrierte Sicht auf die Aufgabenstruktur eines Informationssystems ermöglichen.

Die bedeutendere der beiden Klassen ist die der **objektorientierten Ansätze**. Aus objektorientierter Sicht besteht die Aufgabenstruktur eines Informationssystems aus einer Menge von Objekttypen, die über verschiedenartige Beziehungen miteinander verbunden sind. Jeder Objekttyp besitzt Attribute, Operatoren und Nachrichtendefinitionen. Aus den Objekttypen werden während der Durchführung Instanzen erzeugt, die anhand des Austauschs von Nachrichten über die spezifizierten Beziehungen miteinander kommunizieren. Dabei löst der Empfang einer Nachricht an einem Objekt die Durchführung eines entsprechenden Operators aus [Sinz02b, 1075 f; Sinz99d, 459; FeSi01, 125 f]. Eine ausführliche Erläuterung und Untersuchung des Grundkonzepts der Objektorientierung folgt in Abschnitt 3.5.1. Die Besonderheit objektorientierter Ansätze ist die Integration mehrerer Sichten in einem einzigen Konzept. Die Daten- und die Funktionssicht werden durch die Objekttypen und die zugehörigen Beziehungen abgedeckt. Letztere werden von den Objekten während der Durchführung als Kommunikationskanäle verwendet und stellen somit die Interaktionssicht dar. Außerdem werden auch Teile der Vorgangssicht durch die Spezifikation von Zustandsübergängen der Objekte und Ereignisflüssen zwischen den Objekten unterstützt [Sinz02b, 1076; FeSi01, 127].

Die Klasse der **geschäftsprozessorientierten Ansätze** beschreibt schließlich den Übergang von einer primär statischen und strukturorientierten Sicht auf ein Informationssystem zu einer dynamischen und verhaltensorientierten Sicht [FeSi01, 126]. Im einfachsten Fall wird ein Informationssystem anhand eines ereignisgesteuerten Ab-

laufs von Aufgabendurchführungen beschrieben. Umfassendere Ansätze beziehen darüber hinaus die unternehmenszielorientierte Leistungserstellung, deren Lenkung und die zur Leistungserstellung und Lenkung verwendeten Ressourcen mit ein [FeSi01, 126; Sinz99d, 459]. Daher unterstützen geschäftsprozessorientierte Ansätze zwar primär die Vorgangssicht, können jedoch auch Teile anderer Sichten, wie z. B. die Funktions- und die Datensicht abdecken [Sinz02b, 1076]. Beispielweise können geschäftsprozessorientierte Ansätze auch mit objektorientierten Ansätzen kombiniert werden. Dies entspricht dann der Minimalkombination zur Abdeckung aller Sichten auf die Aufgabenebene eines Informationssystems [Sinz02b, 1074]. Beispiele für Ansätze, die einen geschäftsprozessorientierten Ansatz und einen objektorientierten Ansatz integrieren, sind der Ansatz der objektorientierten Ereignisgesteuerten Prozesskette (oEPK) von Scheer [ScNZ97] und der Ansatz des Semantischen Objektmodells von Ferstl und Sinz, der in Abschnitt 2.4 kurz dargestellt wird.

2.2.5 Entwicklung betrieblicher Anwendungssysteme

Die Festlegung der Struktur der maschinellen Aufgabenträger eines betrieblichen Informationssystems ist Gegenstand der Entwicklung betrieblicher Anwendungssysteme. Dabei resultiert die Struktur der betrieblichen Anwendungssysteme eines betrieblichen Informationssystems aus einer Folge von Abgrenzungsschritten [Sinz99e, 667]:

- Zunächst wird das betriebliche System gegenüber seiner Umwelt abgegrenzt.
- Daraufhin wird das Informationssystem des betrieblichen Systems bestimmt und für die Aufgaben des Informationssystems, die automatisiert werden sollen, ein Gesamt-Anwendungssystem spezifiziert.
- Aus diesem Gesamt-Anwendungssystem werden schließlich einzelne Anwendungssysteme abgegrenzt.

Bei jedem dieser Abgrenzungsschritte werden auch die Schnittstellen zu den jeweils umgebenden Teilsystemen identifiziert [Sinz99e, 667].

Die Entwicklung von Anwendungssystemen ist auch unter der Bezeichnung Software-Prozess, Software-Entwicklung, Software-Engineering oder Software-Technik bekannt [Somm01, 55; FlZü02, 764; BrSi00, 3; Press87, 19; Dene91, 13; Sinz99e, 667]. Die Teilaufgaben dieser Aufgabe können aus Sicht der Struktur eines Anwendungssystems, der zu erbringenden Teilleistungen oder des Vorgehens bei der Aufgabendurchführung beschrieben werden [Sinz99e, 667]. Im Folgenden werden alle drei Perspektiven betrachtet. Da die beiden letztgenannten Perspektiven mittels der erstgenannten Perspektive methodisch begründbar sind, wird zunächst ein Modell zur Beschreibung der Struktur eines Anwendungssystems vorgestellt. Dieses Modell dient wiederum als Basis für ein umfassenderes Software-Systemmodell, mit dem

schließlich die Teilaufgaben während der Entwicklung von Anwendungssystemen definiert werden.

2.2.5.1 Modell der Nutzer- und Basismaschine

Die Struktur eines Anwendungssystems lässt sich anhand des **Modells der Nutzer- und Basismaschine** beschreiben [FeSi84, 74 ff; FeSi01, 286 ff]. Es stellt ein Anwendungssystem aus der Außensicht und aus der Innensicht dar (vgl. Abbildung 2.12).

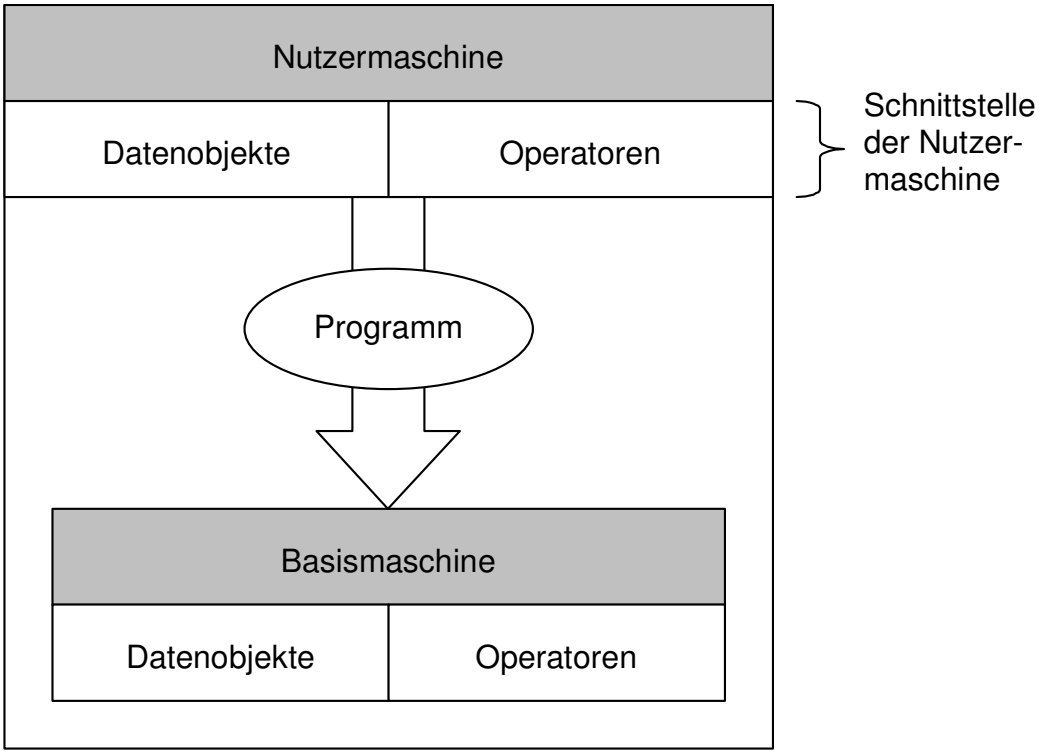


Abbildung 2.12: Nutzer- und Basismaschine [FeSi01, 287]

Die Außensicht eines Anwendungssystems, **Nutzermaschine** genannt, beschreibt seine **Schnittstelle** in Form von externen Datenobjekten und zugehörigen Operatoren. Das Lösungsverfahren der automatisierten Aufgabe wird aus der Außensicht nur anhand dieser Datenobjekte und ihrer Operatoren beschrieben. Somit abstrahiert die Außensicht von der konkreten Realisierung des Lösungsverfahrens der Aufgabe [FeSi01, 286 f].Aus der Innensicht besteht ein Anwendungssystem aus internen Datenobjekten und Operatoren, **Basismaschine** genannt, und einem Programm, das die externen Datenobjekte und Operatoren gemäß dem Lösungsverfahren der automatisierten Aufgabe auf die internen Datenobjekte und Operatoren abbildet. Mit anderen Worten wird die Schnittstelle unter Verwendung der verfügbaren internen Datenobjekte und Operatoren entsprechend des Lösungsverfahrens der automatisierten Aufgabe von einem Programm realisiert [FeSi01, 286 f]. Sowohl die Erstellung des

Programms als auch das Programm selbst wird **Implementierung** genannt.Die Komplexität eines Anwendungssystems lässt sich somit differenzieren in die Komplexität der Nutzermaschine und die der Basismaschine. Der Komplexitätsabstand zwischen der Nutzermaschine und der Basismaschine determiniert ferner die Komplexität des zugehörigen Programms. Ein weiter Komplexitätsabstand zwischen der Nutzer- und der Basismaschine und damit eine hohe Komplexität des Programms kann durch eine mehrstufige Zerlegung des Rechnersystems in weitere Nutzer- und Basismaschinen mit zugehörigen Programmen reduziert werden (vgl. Abbildung 2.13). Dadurch können die Komplexitätsabstände je Stufe überschaubar gehalten werden [FeSi01, 289].

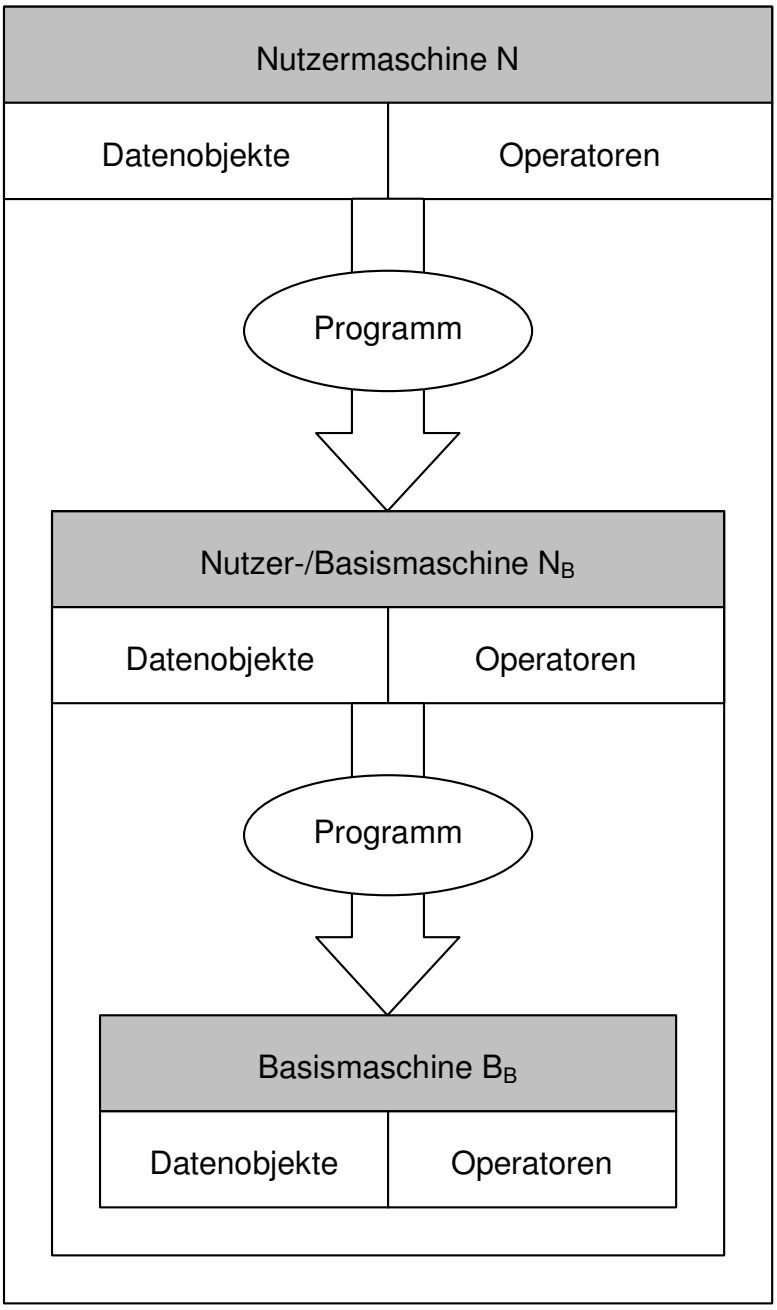


Abbildung 2.13: Mehrstufige Anordnung von Nutzer- und Basismaschinen [FeSi01, 288]

Neben dem beschriebenen Merkmal der Komplexitätsreduktion weist dieses Strukturierungsprinzip außerdem das Merkmal der Flexibilität auf. Dies bezeichnet die Eigenschaft, dass eine Basismaschine mithilfe von entsprechenden Programmen beliebig viele Nutzermaschinen realisieren kann und dass umgekehrt eine Nutzermaschine durch entsprechende Programme von unterschiedlichen Basismaschinen realisiert werden kann. Diese Zuordnungsfreiheit besteht bei einer mehrstufigen Anordnung von Nutzer- und Basismaschinen auf jeder Stufe [FeSi01, 289]. Wenn im Falle einer mehrstufigen Anordnung die Schnittstellen der Basismaschinen standardisiert sind, ist es außerdem möglich, unterschiedliche Basismaschinen mit gleicher Schnittstelle einzusetzen. Damit wird auch die für die Wiederverwendbarkeit erforderliche Portierbarkeit von Anwendungssystemen erfüllt.

2.2.5.2 Software-Systemmodell

Zur Definition der Teilaufgaben, die bei der Entwicklung von Anwendungssystemen durchgeführt werden müssen, reicht das Modell der Nutzer- und Basismaschine nicht aus, da die Umgebung, in die ein Anwendungssystem integriert werden soll, mit diesem Modell nicht spezifizierbar ist. Deswegen wird im Folgenden das Software-Systemmodell eingeführt, das diesen Aspekt zusätzlich berücksichtigt.

Das **Software-Systemmodell**, das in Abbildung 2.14 dargestellt ist, baut auf dem Modell der Nutzer- und Basismaschine auf und erweitert es um eine System- und eine Verfahrensumgebung [Sinz02b, 1080 f; Sinz99e, 668].

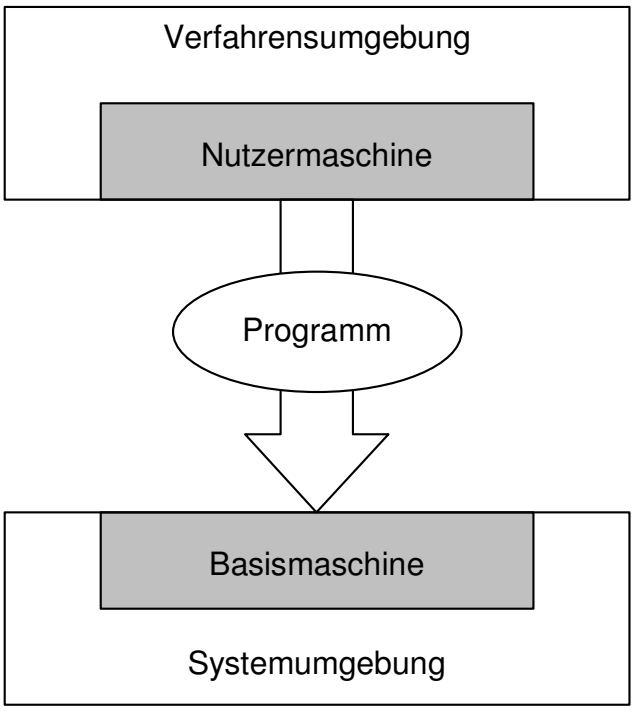


Abbildung 2.14: Software-Systemmodell [Sinz99e, 667]

Demnach ist die Nutzermaschine eines Anwendungssystems in eine **Verfahrensumgebung** eingebettet, die alle Nutzermaschinen maschineller Aufgabenträger und alle personellen Aufgabenträger, mit denen das Anwendungssystem in Beziehung steht, umfasst [Sinz02b, 1081; Sinz99e, 668]. Folglich werden die Schnittstellen für die Mensch-Computer-Kommunikation im Hinblick auf diese Verfahrensumgebung spezifiziert [Sinz02b, 1081]. Analog dazu sind die Basismaschinen eines Anwendungssystems in eine **Systemumgebung** eingebettet, die alle Basismaschinen maschineller Aufgabenträger, mit denen das Anwendungssystem in Beziehung steht, enthält [Sinz02b, 1081; Sinz99e, 668]. Somit werden die Schnittstellen für die Computer-Computer-Kommunikation in Bezug auf diese Systemumgebung definiert [Sinz02b, 1081].

2.2.5.3 Teilaufgaben der Anwendungssystementwicklung

Das vorgestellte Software-Systemmodell stellt einen Beschreibungsrahmen für die Entwicklung von Anwendungssystemen dar. Anhand des Software-Systemmodells werden nun die Teilaufgaben, die bei der Entwicklung von Anwendungssystemen durchgeführt werden müssen, definiert.

Demnach umfasst die Entwicklung von Anwendungssystemen

- die Spezifikation der Nutzermaschine und der Verfahrensumgebung,
- die Spezifikation der Basismaschinen und der Systemumgebung sowie
- die Erstellung des Programms.

Zur Spezifikation der Nutzermaschine gehört die Beschreibung der Schnittstellen des Anwendungssystems. Die Spezifikation der Verfahrensumgebung enthält im Wesentlichen die Beschreibung der Personen und der Schnittstellen anderer Anwendungssysteme, die mit dem Anwendungssystem in Beziehung stehen. Analog zur Spezifikation der Nutzermaschine umfasst die Spezifikation der Basismaschinen die Beschreibung der Schnittstellen der zugrundeliegenden Basismaschinen. Im Rahmen der Spezifikation der Systemumgebung werden die Schnittstellen anderer Basismaschinen, mit denen die zugrundegelegten Basismaschinen in Beziehung stehen, beschrieben. Schließlich besteht die Erstellung des Programms aus der Festlegung der Software-Architektur des Anwendungssystems, auch **Programmieren-im-Großen** genannt, und der programmtechnischen Realisierung der einzelnen Komponenten der Software-Architektur, dem so genannten **Programmieren-im-Kleinen** [Somm01, 68 f; PoBl93, 19 f; FlZü02, 775 f; Meie00, 13 ff; Dene91, 43 f; Pres87, 21]. Die Festlegung der **Software-Architektur** besteht im Allgemeinen aus folgenden Schritten [Balz82, 186; PoBl93, 48; Hamm99, 63; Somm01, 69]:

- Das Programm, genauer gesagt das Programmsystem, wird in mehrere Teilsysteme aufgeteilt. Die Aufteilung erfolgt auf Basis eines gewählten Architekturstils [ShGa96, 19; Bass01, 394; BaCK98, 25 f; Somm01, 228].
- Jedes Teilsystem wird in Komponenten zerlegt und deren Beziehungen untereinander definiert. Dabei wird jede Komponente aus der Außensicht anhand ihrer Schnittstellen beschrieben.
- Schließlich werden für jedes Teilsystem die Innensichten der enthaltenen Komponenten festgelegt. Diese bestehen jeweils aus einem Lösungsverfahren, welches das in den Schnittstellen spezifizierte Verhalten realisiert.

Darüber hinaus müssen die realisierten Komponenten getestet, zu einem Gesamtsystem zusammengefügt und dieses ebenfalls getestet werden [Somm01, 68 f; FlZü02, 776; Dene91, 43 f; PoBl93, 19 f; Meie00, 15 ff].

Die genannten Teilaufgaben bei der Entwicklung von Anwendungssystemen werden nun für die methodische Begründung der resultierenden Teilleistungen und des zugrunde liegenden Vorgehens herangezogen.

Resultierende Teilleistungen der Anwendungssystementwicklung

Die zu erzielende Gesamtleistung, die aus der Durchführung der Aufgabe Entwicklung eines Anwendungssystems entstehen soll, besteht aus einem Anwendungssystem, welches das gewünschte Verhalten und die dafür notwendige Struktur unter Berücksichtigung eventuell vorgegebener Formalziele aufweist. Zur sach- und formalzielbezogenen Lenkung der Aufgabe ist diese Gesamtleistung in der Regel jedoch zu komplex [Sinz99e, 668]. Deswegen wird die Gesamtleistung in Teilleistungen zerlegt, deren Zielerreichungsgrade einzeln überprüfbar sind. Da die Teilleistungen außerdem aufeinander aufbauen, ist mit der Überprüfung der letzten Teilleistung auch die Überprüfung der Gesamtleistung abgeschlossen. Die Abgrenzung der Teilleistungen erfolgt anhand des Software-Systemmodells, und zwar „von den Rändern zur Mitte“ [Sinz99e, 668]. Somit umfasst die erste Teilleistung die Ergebnisse der Spezifikation der Nutzermaschine und ihrer Verfahrensumgebung sowie die Ergebnisse der Spezifikation der Basismaschinen und ihrer Systemumgebung. Diese Teilleistung wird häufig **Fachkonzept** genannt [Sinz99e, 668; Hamm99, 60; Meie00, 11]. Die darauf aufbauende Teilleistung enthält das Ergebnis der Definition der Software-Architektur und wird als **Software-Konzept** bezeichnet [Sinz99e, 668]. Dieses geht wiederum ein in die Erstellung der letzten Teilleistung, das so genannte **Programmsystem** [Sinz99e, 668]. Abbildung 2.15 veranschaulicht die Zuordnung der genannten Teilleistungen zum Software-Systemmodell. Die Teilleistungen sind in kursiver Schrift dargestellt.

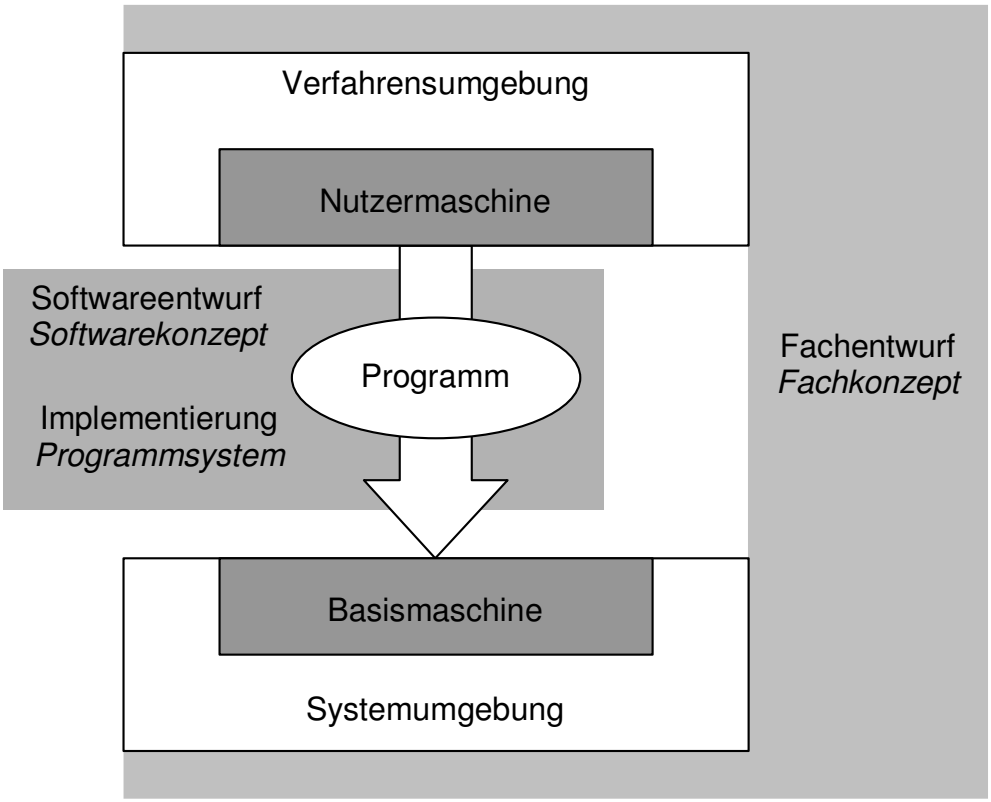


Abbildung 2.15: Phasen und Teilleistungen bei der Entwicklung von Anwendungssystemen (in Anlehnung an [Hamm99, 60])

Zusätzlich zu diesen aus dem Software-Systemmodell abgeleiteten Teilleistungen wird vor der Erbringung der erstgenannten Teilleistung ein **Durchführungsplan** und nach der Erstellung der letztgenannten Teilleistung eine **Abnahme** und Inbetriebnahme des Programmsystems benötigt [Sinz99e, 668; Balz82, 74 f; Meie00, 5 ff ; FlZü02, 776].

Resultierendes Vorgehen bei der Anwendungssystementwicklung

Auf Basis der Teilaufgaben und der aufeinander aufbauenden Teilleistungen lässt sich schließlich auch das Vorgehen bei der Durchführung der Gesamtaufgabe Entwicklung eines Anwendungssystems bestimmen. Das Vorgehen wird üblicherweise in Form eines **Vorgehensmodells** spezifiziert [Somm01, 24; FlZü02, 776; PoBl93, 17; Pres87, 20]. Es unterstützt die sach- und formalzielbezogene Lenkung der Leistungserstellung, indem es die Durchführung der Gesamtaufgabe in **Phasen** und, falls nötig, in Teilphasen strukturiert und die Reihenfolge- und Koordinationsbeziehungen zwischen den Phasen und Teilphasen festlegt [Sinz99e, 669; Somm01, 24; FlZü02, 776; PoBl93, 17; Pres87, 20; Balz82, 15 ff]. Außerdem definiert es die Schnittstellen zwischen den Phasen in Form von so genannten **Phasenübergängen** und bestimmt die Granularität der in einem Phasenübergang zu übergebenden Teilleistung [Sinz99e, 669, Somm01, 24; FlZü02, 776; PoBl93, 17]. Für weitergehende Ausführungen zum Thema Vorgehensmodelle wird auf die einschlägige Literatur verwiesen

(siehe z. B. [Somm01; PoBl93; Balz82]). Im Folgenden werden nur die grundsätzlichen Phasen vorgestellt, die anhand der dargestellten Teilaufgaben und Teilleistungen methodisch begründbar sind.

Danach erfolgt in der ersten Phase mit der Bezeichnung **Planung** die Definition des erwähnten Durchführungsplans [Sinz99e, 669; Meie00, 5 f; Balz82, 74 f]. Daraufhin wird in der zweiten Phase, **fachliche Anwendungssystemspezifikation** oder auch kurz **Fachentwurf** genannt, das oben erläuterte Fachkonzept spezifiziert [Somm01, 57; PoBl93, 19; Pres87, 21; Balz82, 95 ff; Meie00, 5 ff; Dene91, 38 f]. In der dritten Phase mit der Bezeichnung **softwaretechnischer Entwurf** oder **Software-Entwurf** schließt sich die Erstellung des ebenfalls beschriebenen **Software-Konzepts** an [Somm01, 57; PoBl93, 19; Pres87, 21; Balz82, 186 ff; Meie00, 13 ff; Dene91, 43]. Die vorletzte Phase, Realisierung oder **Implementierung** genannt, umfasst die Codierung der im Software-Konzept abgegrenzten Komponenten [Somm01, 57; PoBl93, 19 f; Pres87, 21; Balz82, 369 ff; Meie00, 15 f; Dene91, 43 f]. Das Testen der einzelnen Komponenten, deren Integration zum erwähnten Programmsystem sowie das Testen des Programmsystems selbst werden entweder der Realisierungsphase zugeordnet oder stellen eigenständige Phasen dar [Somm01, 57; PoBl93, 19 f; Pres87, 21; Balz82, 412 ff; Meie00, 16 f; Dene91, 43 f]. Schließlich gehört zur letzten Phase, der so genannten **Einführung**, die Durchführung der oben angeführten Abnahme und Inbetriebnahme des Programmsystems [Balz82, 437 ff; Somm01, 57; PoBl93, 19 f; Meie00, 16 f]. In Abbildung 2.15 sind die Phasen Fachentwurf, softwaretechnischer Entwurf und Implementierung, zusammen mit den entsprechenden Teilleistungen, dem Software-Architekturmodell zugeordnet.

2.3 Architekturen betrieblicher Informations- und Anwendungssysteme

Zur Unterstützung der Entwicklung betrieblicher Informationssysteme werden vermehrt Architekturen verwendet [FoBa01, 290; ShGa96, 1; BaHH00, 1 f]. Gemäß der Architekturlehre im Bauwesen umfasst eine **Architektur** den Bauplan eines Systems, bestehend aus den Komponenten und ihren Beziehungen aus unterschiedlichen Blickwinkeln, und die Konstruktionsregeln zur Erstellung des Bauplans [Sinz02a, 1055; Sinz99a, 32; FoBa01, 291]. Dieses Begriffsverständnis lässt sich unmittelbar auf Architekturen betrieblicher Informationssysteme und betrieblicher Anwendungssysteme übertragen [Sinz02a, 1055; Sinz99a, 32]. Der Bauplan eines Informations- oder eines Anwendungssystems ist ein auf bestimmte Untersuchungsziele ausgerichtetes vereinfachtes Abbild der Struktur- und der Verhaltensmerkmale eines realen Informations- oder Anwendungssystems [Somm01, 225; StWo01, 373]. Das bedeutet, er entspricht einem Modellsystem eines realen Systems [Sinz02a, 1055; StWo01, 373]. Somit sind als Konstruktionsregeln zumindest ein Metamodell und eine Meta-

pher erforderlich [Sinz02a, 1055]. Zur Betonung des Modellcharakters wird anstelle des Begriffs „Architektur“ auch der Begriff „Architekturmodell“ verwendet.

Die Architektur eines Informations- oder eines Anwendungssystems unterstützt dessen Analyse, Gestaltung und Wiederverwendung sowie die Kommunikation zwischen seinen Entwicklern und Nutzern untereinander und miteinander [Somm01, 225 f; StWo01, 372 ff; ShGa96, 16; Bass01, 391; FoBa01, 291 f; Sinz02a, 1056].

In Bezug auf die Analyse von Informations- oder Anwendungssystemen trägt eine Architektur zur Erkennung und Nutzung grundsätzlicher Systemstrukturen bei [ShGa96, 16; FoBa01, 291 f; Bass01, 391]. Außerdem ist es anhand einer Architektur möglich, zu überprüfen, ob ein Anwendungssystem die geforderten funktionalen und nicht-funktionalen Anforderungen erfüllen kann [Somm01, 226; FoBa01, 298; StWo01, 375]. Somit lassen sich auch Fehler in der Struktur oder im Verhalten eines Anwendungssystems schneller und einfacher erkennen und beseitigen [StWo01, 375 f].

Hinsichtlich der Gestaltung eines Anwendungssystems kann die Verwendung einer Architektur sicherstellen, dass das zu erstellende Anwendungssystem die geforderten funktionalen und nicht-funktionalen Anforderungen erfüllt [StWo01, 373 ff; Somm01, 226; ShGa96, 16; FoBa01, 292].

Außerdem beschreibt eine Architektur die notwendigen Zusammenhänge und Annahmen für die Wiederverwendung einzelner Komponenten oder Teilstrukturen während der Gestaltung eines Informations- oder Anwendungssystems [Somm01, 226; StWo01, 377]. Neben Komponenten oder Teilstrukturen können auch vollständige Architekturen wiederverwendet werden [Somm01, 226]. Dabei werden die Teilstrukturen oder Architekturen häufig auch als wiederverwendbares Entwurfswissen in Form von Mustern bereitgestellt [FoBa01, 301; ShGa96, 19 f]. Weitere Ausführungen zur Wiederverwendung mithilfe von Mustern folgen in Abschnitt 3.1.3.6.

Schließlich stellt eine Architektur denjenigen, die an der Analyse, Gestaltung und Nutzung eines Informations- oder Anwendungssystems beteiligt sind, eine gemeinsame Sprache zur Verfügung und unterstützt auf diese Weise die Kommunikation zwischen ihnen [StWo01, 373 ff; FoBa01, 292; Somm01, 225; ShGa96, 16].

Damit eine Architektur die genannten Zwecke erfüllen kann, ist ein umfassendes und hinreichend detailliertes Modellsystem erforderlich [Sinz02a, 1056]. Dieses weist jedoch im Allgemeinen eine hohe Komplexität auf, die wiederum hinderlich für die Erfüllung der genannten Zwecke ist. Darüber hinaus erfordert jeder der genannten Zwecke auch eine andere Perspektive auf das Informations- bzw. Anwendungssystem. Beispielsweise ist für das Erkennen der Grobstruktur eines Anwendungssystems eine abstraktere Sichtweise notwendig als für die Gestaltung der funktionalen

und nicht-funktionalen Anforderungen. Ferner werden zur Erfüllung der genannten Zwecke üblicherweise verschiedene Ausschnitte des Modellsystems benötigt und nicht das Modellsystem im Gesamten [Bass01, 400].

Zur Bereitstellung der geforderten unterschiedlichen Perspektiven lässt sich ein Modellsystem in verschiedene Ebenen unterteilen, von denen jede das gesamte Modellsystem aus einem anderen Blickwinkel beschreibt. Die geforderten Ausschnitte eines Modellsystems werden anhand von Sichten als Projektionen darauf verwirklicht. Darüber hinaus trägt die Ebenen- und Sichtenbildung zur Bewältigung der Komplexität des Modellsystems bei. Mit der Sichtenbildung wird die **typmäßige Komplexität**, die die Artenvielfalt der Modellbausteine ausdrückt, beherrschbar gemacht. Dagegen sorgt die Ebenenbildung für die Bewältigung der **extensionalen Komplexität**, die die Menge der Ausprägungen zu den verfügbaren Modellbausteinen beschreibt [Sinz96, 127].

Anhand der genannten Merkmale einer Architektur lassen sich einzelne Architekturen, aber auch Klassen von Architekturen beschreiben und gestalten. Damit verschiedene Architekturen bzw. Architekturklassen kombiniert oder verglichen werden können, sollten ihre Beschreibungen einheitlich sein. Zu diesem Zweck wird ein generischer Architekturrahmen benötigt, der alle geforderten Beschreibungsmerkmale aufweist.

2.3.1 Generischer Architekturrahmen

Mit dem **generischen Architekturrahmen**, der in Abbildung 2.16 dargestellt ist, können Informations- und Anwendungssystem-Architekturen gestaltet werden, die mehrere Modellebenen aufweisen und auf jeder Modellebene mehrere Sichten als Projektionen auf das Modellsystem besitzen.

Eine **Modellebene** beschreibt ein Informations- oder Anwendungssystem vollständig unter einem bestimmten Blickwinkel [Sinz02a, 1057; Sinz99a, 32]. Wie im vorangegangenen Abschnitt erwähnt, unterstützt ein Blickwinkel eine oder mehrere Zielsetzungen, die mit der Modellbildung verfolgt werden. Häufig benötigte Blickwinkel sind die Außen- und die Innenbetrachtung, der Aufgaben- und der Aufgabenträgerblickwinkel sowie der fachliche und der softwaretechnische Blickwinkel [Sinz02a, 1057; Sinz99a, 32]. Eine **Sicht** erfasst einen bestimmten Ausschnitt des Modellsystems einer Modellebene [Sinz02a, 1057]. Demzufolge stellt eine Sicht in der Regel eine unvollständige Beschreibung einer Modellebene dar. Erst die Gesamtheit aller auf einer Modellebene definierten Sichten geben das vollständige Modellsystem einer Modellebene wieder [Sinz02a, 1057].

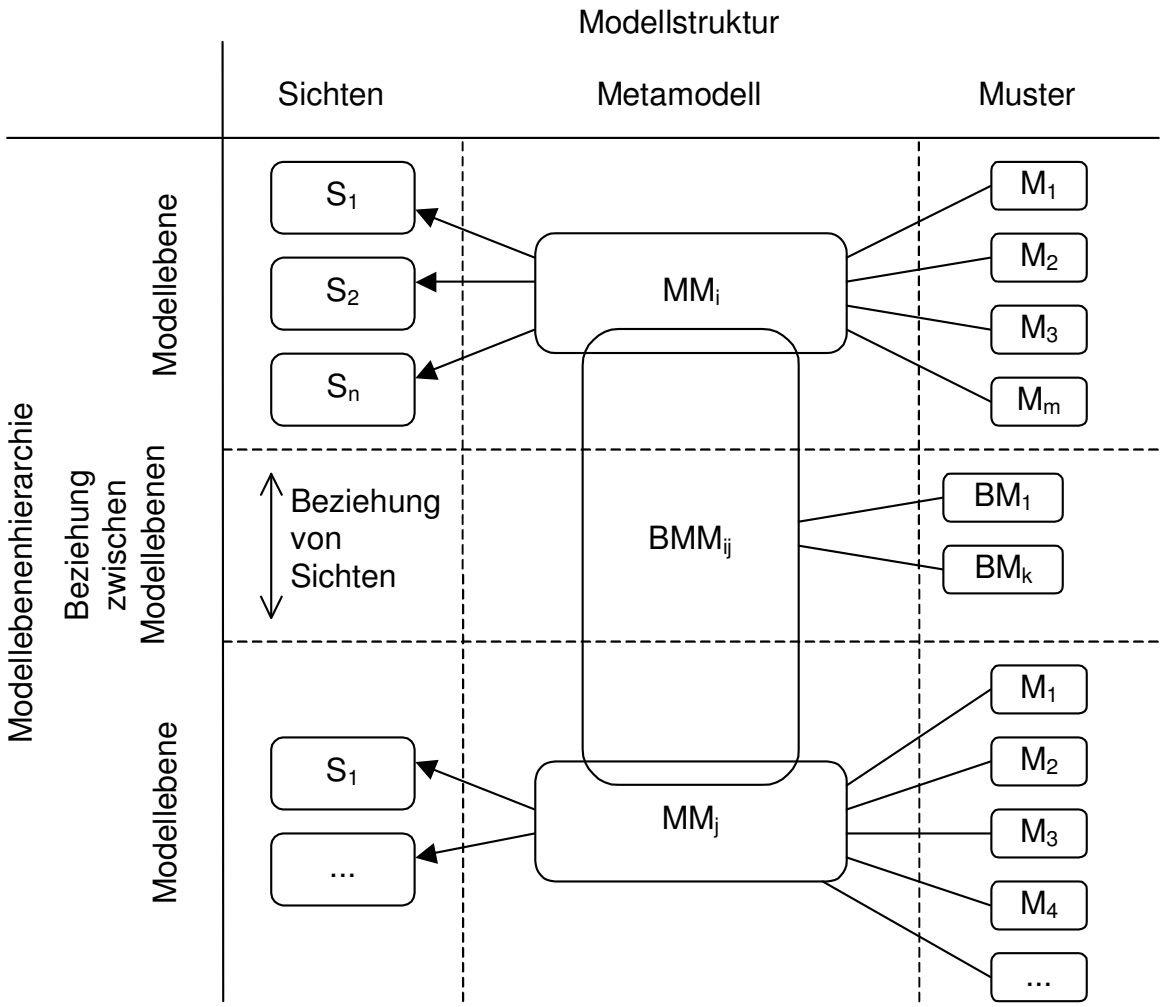


Abbildung 2.16: Generischer Architekturrahmen für Informationssysteme [Sinz02a, 1056]

Für jede Modellebene stehen als Konstruktionsregeln eine Metapher, ein mit der Metapher abgestimmtes Metamodell, gegebenenfalls mehrere Sichten als Projektionen auf das Metamodell und ebenenspezifische Muster zur Verfügung [Sinz02a, 1057 f; Sinz99a, 32]. Die Metapher beschreibt den Blickwinkel, unter dem ein Informations- oder Anwendungssystem auf einer Modellebene betrachtet wird. Das Metamodell stellt ein mit der Metapher korrespondierendes Begriffssystem bereit. Die Bestandteile eines Metamodells wurden bereits in Abschnitt 2.2.3 vorgestellt. Eine Sicht im Sinne einer Konstruktionsregel entspricht formal einer Projektion auf das Metamodell der Modellebene [Sinz02a, 1057 f; Sinz99a, 32]. So kann gewährleistet werden, dass die Metamodelle der verschiedenen Sichten einer Modellebene aufeinander abgestimmt sind. Schließlich stellen **Muster** heuristisches Modellierungswissen zur Verfügung, das die Freiheitsgrade bei der Gestaltung eines Modellsystems einer Modellebene im Hinblick auf die dort geltenden Modellierungsziele einschränkt [Sinz02a, 1057 f]. Die Wiederverwendung anhand von Mustern wird in Abschnitt 3.1.3.6 noch einmal aufgegriffen.

Damit die Modellsysteme verschiedener Modellebenen einer Architektur nicht isoliert voneinander existieren, sondern zu einem umfassenden Modellsystem integriert werden können, sieht der generische Architekturrahmen die Beschreibung von Beziehungs-Metamodellen und zugehörigen Beziehungsmustern zwischen benachbarten Modellebenen vor [Sinz02a, 1058; Sinz99a, 32]. Ein **Beziehungs-Metamodell** legt die Beziehung zweier angrenzender Modellebenen fest, indem es die Verknüpfung zwischen den **Metaobjekten** der einen Modellebene und den Metaobjekten der anderen Modellebene beschreibt [Sinz02a, 1058; Sinz99a, 32]. Anhand von Beziehungs-Metamodellen können die Metamodelle der einzelnen Modellebenen einer Architektur zu einem integrierten Metamodell eines umfassenden Modellsystems verbunden werden [Sinz02a, 1058]. Allerdings bestehen trotz eines Beziehungs-Metamodells weiterhin Freiheitsgrade bei der Zuordnung von Metaobjekten der einen Ebene zu denen der anderen. Zur Einschränkung dieser Freiheitsgrade können wiederum Muster verwendet werden, die heuristisches Modellierungswissen in Bezug auf die Zuordnungs- oder Transformationsbeziehungen benachbarter Modellebenen bereitstellen [Sinz02a, 1058]. Diese spezielle Art von Mustern wird auch **Beziehungsmuster** genannt [Sinz02a, 1058; Sinz99a, 32].

Anhand der vorgestellten Merkmale des generischen Architekturrahmens können nun verschiedene Architekturen bzw. Architekturklassen gestaltet werden. Dafür ist eine Festlegung bestimmter Ausprägungen zu den Merkmalen erforderlich [Sinz02a, 1056]. Im vorliegenden Zusammenhang werden entweder Informationssystem-Architekturen bzw. -Architekturklassen oder Anwendungssystem-Architekturen bzw. -Architekturklassen gestaltet. Da gemäß den Ausführungen in Abschnitt 2.1.5 Anwendungssysteme automatisierte Teilsysteme eines Informationssystems sind, stellen Anwendungssystem-Architekturen Teilarchitekturen einer Informationssystem-Architektur dar (vgl. Abbildung 2.17). Informationssystem-Architekturen werden im folgenden Abschnitt vorgestellt, Anwendungssystem-Architekturen in Abschnitt 2.3.3.

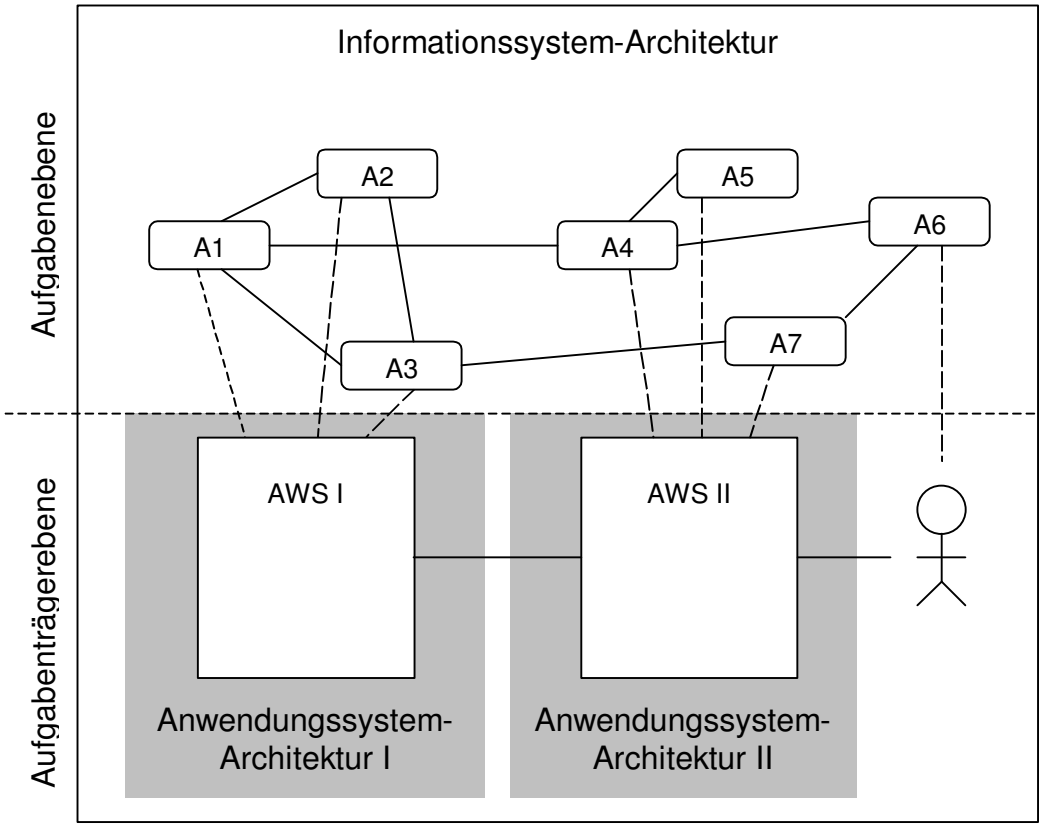


Abbildung 2.17: Informationssystem- und Anwendungssystem-Architektur

2.3.2 Informationssystem-Architektur

Nach den bisherigen Ausführungen umfasst eine **Informationssystem-Architektur** zumindest eine Aufgabenebene und eine Aufgabenträgerebene [Sinz02a, 1055]. Die Aufgabenebene entspricht der Ebene des Fachkonzepts [Sinz02a, 1057]. Die Aufgabenträgerebene wird unterteilt in eine Modellebene für personelle Aufgabenträger und eine Modellebene für maschinelle Aufgabenträger. Letztgenannte entspricht der Ebene des Software-Konzepts [Sinz02a, 1057]. Sowohl die Aufgabenebene als auch die Aufgabenträgerebene kann gegebenenfalls in mehrere Unterebenen aufgeteilt werden [Sinz02a, 1057]. Die Ebene des Software-Konzepts wird im folgenden Abschnitt näher untersucht.

Für die Ebene des Fachkonzepts sind eine Vielzahl von Modellierungsansätzen bekannt, die in Abschnitt 2.2.4.2 anhand der von ihnen unterstützten Sichten auf die Aufgabenmerkmale klassifiziert wurden. Wie erwähnt, besteht ein Modellierungsansatz aus einem Metamodell und einer Metapher. Die von einem Modellierungsansatz unterstützten Sichten stellen zugleich typische Sichten auf der Ebene des Fachkonzepts dar [Sinz02a, 1057]. Bei Bedarf können noch weitere Sichten, wie z. B. eine Struktur- und eine Verhaltenssicht definiert werden. Außerdem stehen für die Ebene des Fachkonzepts vermehrt Muster zur Verfügung. Dazu zählen z. B. die Analyse-

muster von FOWLER [Fowl96] oder einige grundlegende Entwurfsmuster von GAMMA ET AL. [GHVJ96]. Die **Analysemuster** von FOWLER tragen zum Verständnis unterschiedlicher betrieblicher Domänen bei. Dagegen unterstützen die **Entwurfsmuster** von GAMMA ET AL. die Lösungssuche bei konkreten Entwurfsproblemen. Ausführliche Erläuterungen zum Einsatz von Mustern bei der Entwicklung von Anwendungssystemen folgen in Abschnitt 3.1.3.6.

Falls die Ebene des Fachkonzepts in mehrere Unterebenen aufgeteilt ist, werden zur Verbindung von zwei benachbarten Unterebenen jeweils ein Beziehungs-Metamodell und zugehörige Beziehungsmuster benötigt. Ferner ist es für eine ganzheitliche und methodisch durchgängige Entwicklung von Anwendungssystemen erforderlich, dass die Ebene des Fachkonzepts und die Ebene des Software-Konzepts anhand von geeigneten Beziehungs-Metamodellen und Beziehungsmustern miteinander verbunden werden. Im Rahmen der Erläuterungen zum komponentenbasierten Software-Architekturmodell in Kapitel 5 werden Beziehungs-Metamodelle und Beziehungsmuster zur Verknüpfung der Fachkonzept-Ebene mit der Software-Konzept-Ebene vorgestellt.

Aktuell stehen mehrere Konzepte zur Gestaltung von Informationssystem-Architekturen zur Verfügung. Dazu gehören beispielsweise die Architektur integrierter Informationssysteme von SCHEER [Sche98a; Sche98b] und die Unternehmensarchitektur der SOM-Methodik, die in Abschnitt 2.4.1 noch vorgestellt wird. Die Architekturrahmen dieser Konzepte können als Ausprägungen des generischen Architekturrahmens interpretiert werden. Einen ausführlichen Vergleich ausgewählter Architekturrahmen enthält z. B. [Sinz02a].

2.3.3 Anwendungssystem-Architektur

Wie bereits in Abschnitt 2.3.1 erwähnt, stellen **Anwendungssystem-Architekturen** Teilarchitekturen einer Informationssystem-Architektur dar. Das heißt, eine Anwendungssystem-Architektur besteht ebenfalls aus einer Fachkonzept-Ebene und einer Software-Konzept-Ebene. Die Fachkonzept-Ebene einer Anwendungssystem-Architektur entspricht einem Teil der Fachkonzept-Ebene der umfassenden Informationssystem-Architektur. Die Fachkonzept-Ebene einer Informationssystem-Architektur wurde im vorangegangenen Abschnitt bereits erläutert. Im Folgenden wird die Software-Konzept-Ebene einer Anwendungssystem-Architektur näher untersucht. Sie enthält eine Software-Architektur, die einer Ausprägung des generischen Architekturrahmens entspricht. Demnach ist jede Software-Architektur einer Anwendungssystem-Architektur zugeordnet. Abbildung 2.18 veranschaulicht diesen Zusammenhang, wobei die darin enthaltene Software-Architektur vorerst als Black Box dargestellt ist. Da eine Informationssystem-Architektur, wie erwähnt, mehrere Anwendungssystem-Architekturen enthalten kann, weist sie gegebenenfalls auch meh-

rere Software-Architekturen auf. In Abschnitt 2.2.5.3 wurden bereits die grundsätzlichen Schritte zur Gestaltung einer Software-Architektur kurz dargestellt. Anhand dieser Schritte lässt sich auch der Aufbau einer Software-Architektur erklären.

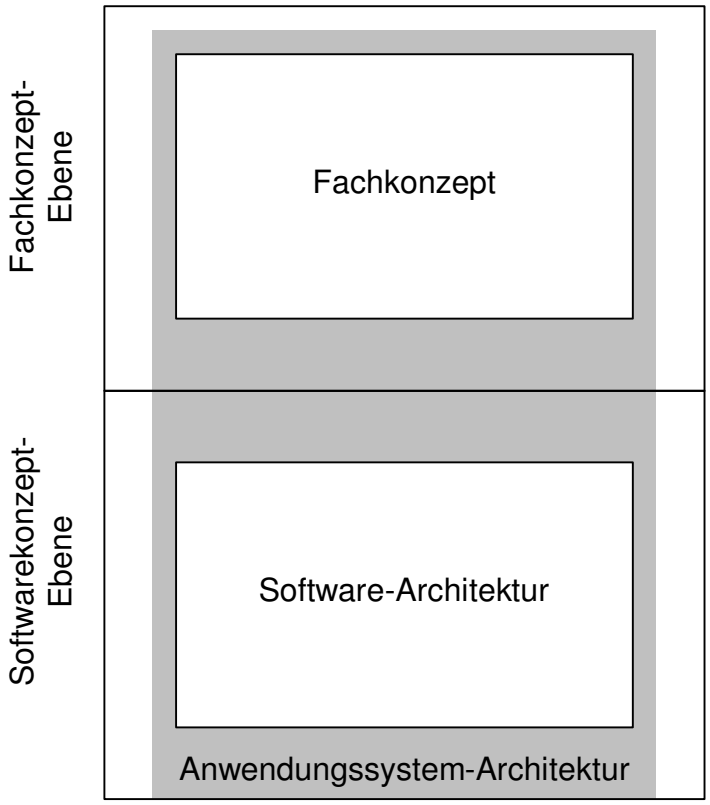


Abbildung 2.18: Anwendungssystem- und Software-Architektur

Schritte zur Gestaltung von Software-Architekturen

Zunächst wird ein Programmsystem in mehrere **Teilsysteme** aufgeteilt. Dafür stehen verschiedene Architekturstile zur Verfügung [ShGa96, 19; Bass01, 394; Somm01, 228; BMR+98, 389 f]. Im vorliegenden Fall müssen jedoch die zur Wahl stehenden Architekturstile auf solche beschränkt werden, die sich mit dem generischen Architekturrahmen beschreiben lassen. Dazu gehören insbesondere die unterschiedlichen Formen von Ebenen-Architekturen, wie z. B. das Modell der abstrakten Maschinen [Somm01, 232 f; FeSi01, 302 f; ShGa96, 25].

Daraufhin wird jedes Teilsystem in **Komponenten** und zugehörige Beziehungen zerlegt. Dies kann entweder datenflussorientiert oder objektorientiert erfolgen [Somm01, 239]. Beide Ansatz-Klassen wurden bereits im Rahmen der Erläuterungen zur Aufgabenstruktur von Informationssystemen in Abschnitt 2.2.4.2 vorgestellt. Aufgrund der inhärenten Vorteile der objektorientierten Ansätze in Bezug auf die Integration von Daten-, Funktions- und Interaktionssicht haben sich diese inzwischen für die Zerlegung von Teilsystemen in Komponenten durchgesetzt [SiMe00, 104 ff; FoBa01, 295]. Die Zerlegung muss im Hinblick auf die bestmögliche Erfüllung der funktionalen

und nicht-funktionalen Anforderungen erfolgen [Bass01, 401 f; Somm01, 227 f; SiMe00, 98 ff; FoBa01, 301]. Zu diesem Zweck stehen Architekturmuster, Entwurfsmuster oder so genannte Entwurfsprinzipien zur Verfügung [BMR+98, 386 ff]. Architekturmuster und Entwurfsmuster werden in Abschnitt 3.1.3.6 noch einmal aufgegriffen, Entwurfsprinzipien sind Gegenstand des Abschnitts 3.4.2. Die resultierenden Komponenten werden zunächst aus der Außensicht und anschließend aus der Innensicht beschrieben (vgl. Abschnitt 2.2.5.3). Die Innensicht einer Komponente spezifiziert ein Lösungsverfahren, das das in der Außensicht anhand von Schnittstellen spezifizierte Verhalten realisiert. Daraus folgt, dass eine Software-Architektur zumindest eine **Ebene für die Außensicht** und eine **Ebene für die Innensicht** aufweisen muss. Zur besseren Unterscheidung zwischen der Sichtweise eines Modellierers, unter dem ein Anwendungssystem auf einer Modellebene beschrieben wird, und den Sichten als Projektionen auf das Metamodell einer Modellebene, sind im Zusammenhang mit Modellebenen eines Architekturmodells die Begriffe „**Außenperspektive**“ und „**Innenperspektive**“ den Begriffen „Außensicht“ und „Innensicht“ vorzuziehen.

Gemäß den in Abschnitt 2.3.1 dargestellten Merkmalen des generischen Architekturrahmens liegen sowohl auf der Außenperspektive- als auch auf der Innenperspektive-Ebene Konstruktionsregeln in Form einer Metapher, eines mit der Metapher abgestimmten Metamodells und ebenenspezifischen Mustern vor. Auf der Außenperspektive-Ebene werden hauptsächlich Architekturmuster, auf der Innenperspektive-Ebene überwiegend Entwurfsmuster oder Idiome eingesetzt. Ferner sind auf beiden Ebenen Sichten als Projektionen auf das jeweilige Metamodell bzw. Modellsystem definiert. Dabei gehören die bereits auf der Fachkonzept-Ebene verwendeten Daten-, Funktions-, Interaktions- und Vorgangssicht zu den typischen Sichten. Daneben können noch weitere Sichten, wie z. B. eine Struktur- und eine Verhaltenssicht abgegrenzt werden.

Damit der oben erwähnte Zusammenhang zwischen beiden Ebenen modelliert werden kann, stehen Beziehungs-Metamodelle zur Verfügung, die die Metaobjekte der Außenperspektive-Ebene mit den Metaobjekten der Innenperspektive-Ebene verknüpfen. Dabei wird die Modellierung des Zusammenhangs durch geeignete Beziehungsmuster unterstützt. Außerdem wurde bereits darauf hingewiesen, dass für eine umfassende und methodisch durchgängige Entwicklung von Anwendungssystemen zusätzliche Beziehungs-Metamodelle und Beziehungsmuster zwischen der Außenperspektive-Ebene der Software-Architektur und der Fachkonzept-Ebene notwendig sind.

Anhand der vorgestellten Merkmale einer Software-Architektur können unterschiedliche Software-Architekturen oder Software-Architektur-Klassen spezifiziert werden. Dafür werden den genannten Merkmalen unterschiedliche Ausprägungen zugeord-

net. Das im weiteren Verlauf der Arbeit vorzustellende komponentenbasierte Software-Architekturmodell stellt eine solche Software-Architektur-Klasse dar. Die zu den Merkmalen gewählten Ausprägungen werden in Kapitel 3 und 5 methodisch begründet.

2.4 Einführung in die Methodik des Semantischen Objektmodells (SOM)

Die Methodik des **Semantischen Objektmodells (SOM)** nach Ferstl/Sinz baut auf den in den vorangegangenen Abschnitten vorgestellten Konzepten und Modellen zur Konstruktion von Informations- und Anwendungssystemen auf. Unter einer **Methodik** wird in diesem Zusammenhang ein Modellierungsansatz zusammen mit einer Informationssystemarchitektur und einem damit abgestimmten Vorgehensmodell verstanden [FeSi01, 180]. Die Bezeichnung „Semantisches Objektmodell“ drückt aus, dass die vorliegende Methodik vollständig auf dem Grundkonzept der Objektorientierung beruht und mit dessen Hilfe betriebliche Bedeutungszusammenhänge explizit erfasst werden können [FeSi98, 339]. Die allgemeine methodische Begründung der SOM-Methodik beruht auf den systemtheoretischen Grundlagen, die in Abschnitt 2.1.1 kurz vorgestellt wurden [FeSi01, 180; FeSi98, 339].

Der Modellierungsansatz der SOM-Methodik kann anhand der in Abschnitt 2.2.3 eingeführten Merkmale Modellierungsreichweite, Modellierungszweck und Modellumfang charakterisiert werden.

Modellierungsreichweite des SOM-Ansatzes

Die Modellierungsreichweite des SOM-Ansatzes umfasst nicht nur das betriebliche Informationssystem, sondern das gesamte betriebliche System [FeSi01, 180; FeSi95, 209]. Es wird dabei als System interagierender Geschäftsprozesse verstanden, die betriebliche Leistungen erstellen oder deren Erstellung lenken [FeSi95, 209; FeSi01, 180]. Zur Durchführung der Geschäftsprozesse weist das betriebliche System Personal, Anwendungssysteme oder Maschinen und Anlagen auf [FeSi01, 182; FeSi98, 340].

Modellierungszweck des SOM-Ansatzes

Die SOM-Methodik ist eine umfassende Methodik zur Analyse und zielorientierten Gestaltung betrieblicher Systeme [FeSi94, 6]. Deswegen können mit dem Modellierungsansatz der SOM-Methodik Beschreibungs- und normative Gestaltungsmodelle erstellt werden. Dies bestimmt den Modellierungszweck des SOM-Ansatzes.

Modellumfang des SOM-Ansatzes

Der Modellumfang des SOM-Ansatzes wird bestimmt durch die zugehörige Architektur und muss somit differenziert werden in den Umfang der Modellebenen sowie den

Umfang der Sichten auf die einzelnen Modellebenen [FeSi95, 210 f]. In Bezug auf die Modellebenen unterstützt der SOM-Ansatz den Unternehmensplan, Geschäftsprozessmodelle und Anwendungssystemspezifikationen als Teilmodelle eines betrieblichen Systems [FeSi95, 210]. Auf jeder genannten Ebene wird zwischen einer Struktursicht und einer Verhaltenssicht unterschieden.

Betriebliches System im SOM-Ansatz

Im SOM-Ansatz wird das in Abschnitt 2.1.2 beschriebene Verständnis eines betrieblichen Systems weiter konkretisiert. Demnach ist ein betriebliches System ein **offenes**, **zielorientiertes** und **sozio-technisches System** [FeSi98, 340]. Es ist offen, weil es mit Kunden, Lieferanten und anderen Geschäftspartnern interagiert und dabei Leistungen in Form von Gütern, Dienstleistungen und Zahlungsmitteln austauscht [FeSi98, 340]. Das Verhalten des betrieblichen Systems ist an den Sach- und Formalzielen des Unternehmens ausgerichtet [FeSi98, 340]. Die Sachziele spezifizieren die Art und den Zweck der Leistungserstellung sowie der Leistungsverwertung [FeSi01, 66]. Mit den Formalzielen wird die Güte der Leistungen und des Leistungsprozesses in technischer und wirtschaftlicher Hinsicht vorgegeben [FeSi01, 66]. Schließlich stehen als Aufgabenträger eines betrieblichen Systems Menschen und Maschinen zur Verfügung, die für die Leistungserstellung und deren Lenkung miteinander kooperieren [FeSi01, 65]. Maschinelle Aufgabenträger des Informationssystems als Teilsystem eines betrieblichen Systems sind Anwendungssysteme, deren Spezifikation durch den SOM-Ansatz unterstützt wird.

Dieses Verständnis eines betrieblichen Systems beschreibt jedoch nur dessen Außensicht. Die Innensicht eines betrieblichen Systems wird im SOM-Ansatz charakterisiert als ein **verteiltes System**, bestehend aus autonomen, lose gekoppelten Komponenten, die zur Erreichung der Ziele des betrieblichen Systems miteinander kooperieren [FeSi01, 182; FeSi98, 340].

Die beschriebenen Sichtweisen auf ein betriebliches System bilden auch die übergreifende Metapher für die Modellierung im SOM-Ansatz [FeSi01, 182]. Sie wird im Zusammenhang mit den Erläuterungen zur SOM-Architektur noch weiter konkretisiert.

Informationssystem-Architektur und Vorgehensmodell der SOM-Methodik

Zur Bewältigung der hohen Komplexität eines betrieblichen Systems verfügt die SOM-Methodik über eine Informationssystem-Architektur, die mehrere Modellebenen umfasst. Da die Modellierung mit dem SOM-Ansatz auch die Leistungserstellung eines betrieblichen Systems mit einbezieht, ist sie zu einer **Unternehmensarchitektur** erweitert [Sinz02a, 1061]. Ferner weist die SOM-Methodik, wie erwähnt, ein mit der

Unternehmensarchitektur abgestimmtes Vorgehensmodell auf. Unternehmensarchitektur und Vorgehensmodell werden in den folgenden Abschnitten kurz vorgestellt.

2.4.1 Unternehmensarchitektur

Die Unternehmensarchitektur der SOM-Methodik, die in Abbildung 2.19 dargestellt ist, kann als Ausprägung des in Abschnitt 2.3.1 vorgestellten generischen Architekturrahmens interpretiert werden. Sie beschreibt ein betriebliches System aus Sicht der **globalen Unternehmensaufgabe**. Folglich umfasst sie eine Modellebene für die Außensicht, eine für die Innensicht und eine für die Aufgabenträger des betrieblichen Systems.

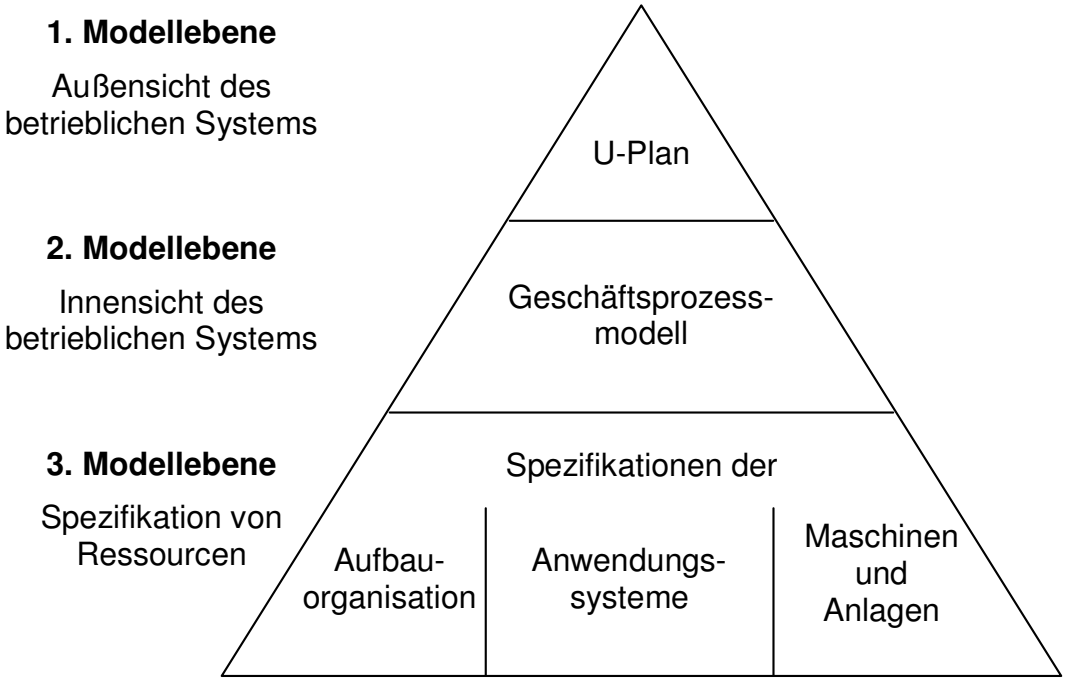


Abbildung 2.19: Unternehmensarchitektur der SOM-Methodik [FeSi01, 181]

Erste Modellebene der SOM-Umternehmensarchitektur

Auf der ersten Modellebene wird der **Unternehmensplan** spezifiziert. Er ist ein Modellsystem der Außensicht eines betrieblichen Systems [FeSi98, 212; FeSi01, 181]. Somit enthält er das Aufgabenobjekt der globalen Unternehmensaufgabe in Form einer abgegrenzten Diskurswelt und Umwelt, die Sach- und Formalziele der Unternehmensaufgabe und die Leistungsbeziehungen zwischen Diskurswelt und Umwelt [FeSi01, 181; FeSi98, 341; FeSi95, 212]. Außerdem können anhand der globalen Unternehmensaufgabe Anforderungen an die Aufgabenträger abgeleitet werden, die mit den verfügbaren Aufgabenträgerkapazitäten abgeglichen werden müssen [FeSi98, 341].

Zweite Modellebene der SOM-Umternehmensarchitektur

Die zweite Modellebene enthält das Modellsystem der Innensicht eines betrieblichen Systems [FeSi01, 181; FeSi98, 342; FeSi95, 212]. Es spezifiziert das Lösungsverfahren für die Realisierung des Unternehmensplans in Form von **Geschäftsprozessen**, die untereinander und mit der Umwelt Leistungen austauschen [FeSi01, 181; FeSi98, 342; FeSi95, 212]. Die dabei zugrundegelegte Metapher basiert auf der in Abschnitt 2.4 vorgestellten übergreifenden Metapher für die Modellierung der Innensicht im SOM-Ansatz. Folglich beschreibt das Modellsystem ein verteiltes System von autonomen Geschäftsprozessen, die über Leistungsbeziehungen lose gekoppelt sind und auf diese Weise zur Erreichung der im Unternehmensplan angegebenen Sach- und Formalziele miteinander kooperieren [FeSi01, 181; FeSi98, 342; FeSi95, 212].

Dritte Modellebene der SOM-Umternehmensarchitektur

Auf der dritten und letzten Modellebene werden die Modellsysteme der Aufgabenträger eines betrieblichen Systems spezifiziert. Sie führen die auf der zweiten Modellebene spezifizierten Geschäftsprozesse durch. Wie erwähnt stehen als **personelle Aufgabenträger** das Personal und als **maschinelle Aufgabenträger** Maschinen und Anlagen sowie Anwendungssysteme zur Verfügung. Die SOM-Methodik unterstützt insbesondere die Spezifikation von Anwendungssystemen [FeSi01, 182]. Dabei wird ebenfalls die in Abschnitt 2.4 vorgestellte übergreifende Metapher für die Modellierung der Innensicht im SOM-Ansatz zugrundegelegt. Folglich werden auf dieser Modellebene objektorientierte und objektintegrierte verteilte Anwendungssysteme spezifiziert, die jeweils bestimmte automatisierbare Informationssystemanteile eines oder mehrerer Geschäftsprozesse automatisieren [FeSi95, 212]. Genauere Ausführungen hierzu folgen in Abschnitt 2.4.1.2.

Charakteristische Merkmale der SOM-Umternehmensarchitektur

Auf jeder der genannten Ebenen wird das zu modellierende betriebliche System vollständig unter einem bestimmten Blickwinkel beschrieben [FeSi01, 180]. So sind lokale Änderungen auf jeder Ebene ohne Auswirkungen auf die Gesamtarchitektur möglich [FeSi98, 342]. Andererseits können die Modellsysteme auf den einzelnen Ebenen auch anhand von explizit ausweisbaren Beziehungen untereinander abgestimmt werden [FeSi01, 182].

Darüber hinaus sind auf jeder der genannten Ebenen eine Sicht für strukturorientierte Merkmale und eine Sicht für verhaltensorientierte Merkmale definiert [Sinz02a, 1062].

Schließlich wird sowohl auf der zweiten als auch auf der dritten Modellebene ein verteiltes System spezifiziert, das aus autonomen, lose gekoppelten Komponenten besteht, die zur Erreichung der Ziele des betrieblichen Systems miteinander kooperieren [FeSi98, 342; FeSi01, 182]. Auf diese Weise sind der geschäftsprozessorientierte

Modellierungsansatz der zweiten Ebene und der objektorientierte Modellierungsansatz der dritten Ebene miteinander verbunden.

Die Spezifikation des Unternehmensplans auf der erstgenannten Modellebene erfolgt in der SOM-Methodik meist informal [FeSi95, 213]. Dagegen steht für die Beschreibung der Modellsysteme auf den übrigen zwei Modellebenen eine semi-formale und vorwiegend diagrammgestützte Notation zur Verfügung [FeSi95, 210]. Folglich können für diese beiden Modellebenen auch Metamodelle und Beziehungs-Metamodelle angegeben werden. Diese werden in den folgenden Abschnitten kurz vorgestellt.

2.4.1.1 Ebene des Geschäftsprozessmodells

Ein **Geschäftsprozess** wird in der SOM-Methodik anhand von drei Merkmalen definiert, die in drei unterschiedlichen Sichten auf ihn zum Ausdruck kommen [FeSi01, 185 f; FeSi95, 214]:

- In der **Leistungssicht** erstellt ein Geschäftsprozess eine oder mehrere **betriebliche Leistungen** und übergibt diese an die ihn beauftragenden Geschäftsprozesse. Ferner beauftragt ein Geschäftsprozess seinerseits andere Geschäftsprozesse mit der Erstellung und der Lieferung von Leistungen.
- In der **Lenkungssicht** koordiniert ein Geschäftsprozess die an der Erstellung und Übergabe von Leistungen beteiligten betrieblichen Objekten anhand von **betrieblichen Transaktionen**. Dabei werden als Koordinationsformen das **Verhandlungsprinzip** und das **Regelungsprinzip** verwendet. Das Verhandlungsprinzip entspricht einer **nicht-hierarchischen Koordinationsform** und besteht aus den aufeinander folgenden Transaktionen Anbahnung, Vereinbarung und Durchführung eines Leistungsaustauschs. Dagegen entspricht das Regelungsprinzip einer **hierarchischen Koordinationsform** und umfasst die Transaktionen Steuerung und Kontrolle einer Aufgabendurchführung. Genauere Erläuterungen hierzu enthalten [FeSi01, 189 f; FeSi95, 217].
- In der **Ablaufsicht** stellt ein Geschäftsprozess einen ereignisgesteuerten Ablauf der den betrieblichen Objekten zugeordneten Aufgaben in Form von Vorgängen dar.

Die beiden erstgenannten Sichten beziehen sich auf das Systemmerkmal Struktur, die letztgenannte Sicht bezieht sich auf das Systemmerkmal Verhalten [FeSi01, 185 f; FeSi95, 214]. Demzufolge werden die beiden erstgenannten Sichten in einer gemeinsamen strukturorientierten Sicht und die letztgenannte Sicht in einer verhaltensorientierten Sicht auf ein Geschäftprozessmodellsystem erfasst.

Metapher auf der Ebene des Geschäftsprozessmodells

Die Metapher auf der Ebene des Geschäftsprozessmodells konkretisiert die in Abschnitt 2.4 eingeführte übergreifende Metapher für die Modellierung der Innensicht im SOM-Ansatz. Danach liegt Geschäftsprozessmodellsystemen die Metapher eines verteilten Systems zugrunde, bestehend aus autonomen lose gekoppelten betrieblichen Objekten, die anhand von Transaktionen in Bezug auf eine gemeinsame Zielerfüllung koordiniert werden. Die Bildung autonomer, lose gekoppelter betrieblicher Objekte folgt dem Konzept der Objektorientierung, die Koordination der betrieblichen Objekte folgt dem Konzept der Transaktionsorientierung [FeSi01, 187].

Ein **betriebliches Objekt** umfasst eine Menge von betrieblichen Aufgaben, deren Aufbau dem in Abschnitt 2.1.3 vorgestellten Konzept einer betrieblichen Aufgabe entspricht (vgl. Abbildung 2.20). Die dem betrieblichen Objekt zugeordneten betrieblichen Aufgaben verfolgen gemeinsame Sach- und Formalziele und operieren auf einem gemeinsamen Aufgabenobjekt [FeSi01, 187 f; FeSi95, 214].

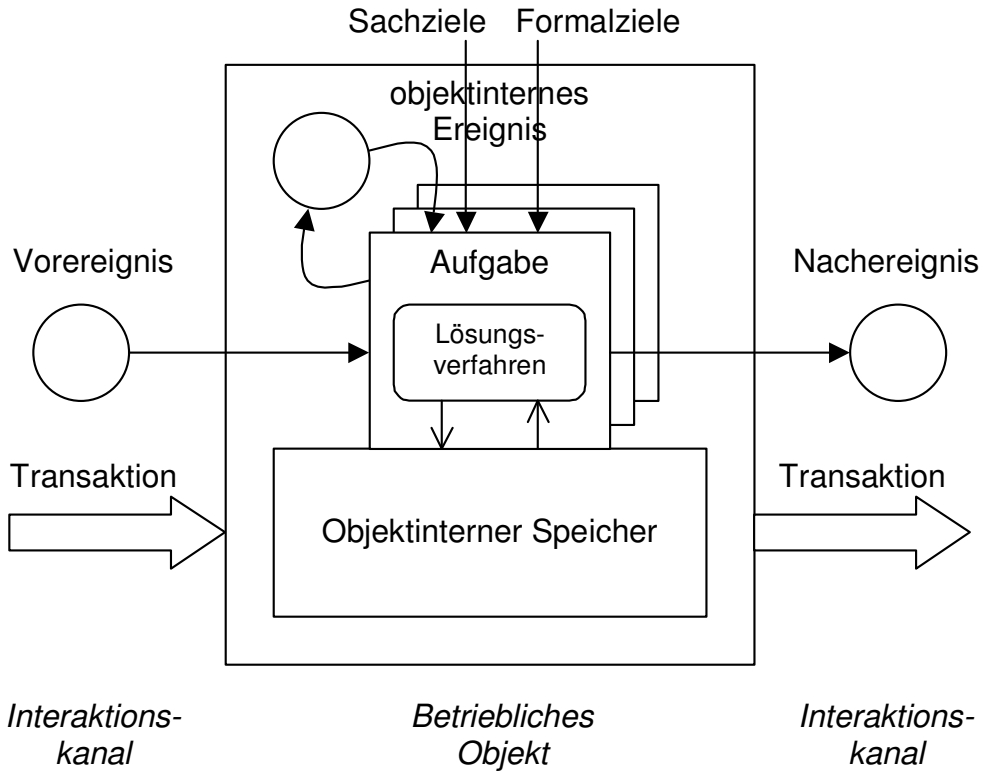


Abbildung 2.20: Konzept betrieblicher Objekte in SOM [FeSi01, 187]

Betriebliche Objekte sind durch Transaktionen lose gekoppelt. Eine **Transaktion** ist ein Kommunikationskanal, auf dem **Leistungspakete** oder **Lenkungsnachrichten** transportiert werden. Ein eingehendes Leistungspaket oder eine eingehende Lenkungsnachricht löst ein Vorereignis am empfangenden Objekt zur Durchführung einer entsprechenden Aufgabe aus. Während der Aufgabendurchführung können Nacher-

eignisse produziert werden, die, wiederum an Leistungspakete oder Lenkungsnachrichten gebunden, über Transaktionen an andere betriebliche Objekte gesandt werden. Jede Transaktion definiert ein fachliches Kommunikationsprotokoll für den Austausch von Leistungspaketen und Lenkungsnachrichten, das von den Aufgaben, die die Transaktion durchführen, verwendet wird. Weitergehende Erläuterungen zum transaktionsorientierten Konzept der SOM-Methodik enthalten [FeSi01, 188 ff; FeSi95, 214 ff].

Ein Geschäftsprozessmodell kann mehrstufig verfeinert werden, um seine Koordination aufdecken zu können. Dabei lassen sich sowohl die betrieblichen Objekte als auch die Transaktionen verfeinern. Zum einen verfeinert die Anwendung des Verhandlungsprinzip eine Transaktion in eine **Anbahnungs-,** eine **Vereinbarungs-** und eine **Durchführungstransaktion**. Zum anderen zerlegt die Anwendung des Regelungsprinzips ein betriebliches Objekt in ein **Reglerobjekt** und ein **Regelstreckenobjekt** und deckt dabei eine **Steuer-** und eine **Kontrolltransaktion** zwischen den entstandenen Objekten auf. Die Zerlegungen von betrieblichen Objekten und Transaktionen lässt sich in den meisten Fällen rekursiv weiterführen. Alle zulässigen Zerlegungsformen sind in der SOM-Methodik anhand von **Zerlegungsregeln** vorgegeben. Die Zerlegungsregeln und weitere Ausführungen zur schrittweisen Verfeinerung von Geschäftsprozessmodellen können [FeSi01, 190 f; FeSi95, 216 f; FeSi98, 343] entnommen werden.

Metamodell der Ebene des Geschäftsprozessmodells

Abbildung 2.21 zeigt das integrierte Metamodell der Ebene des Geschäftsprozessmodells. Es wurde auf der Grundlage des in Abschnitt 2.2.3 vorgestellten Meta-Metamodells spezifiziert und enthält die Metaobjekte betriebliches Objekt, betriebliche Transaktion, Leistung, Aufgabe, Umweltereignis, objektinternes Ereignis und deren Beziehungen zueinander. Alle genannten Metaobjekte wurden im Verlauf dieses Abschnitts bereits eingeführt. Ein betriebliches Objekt kann entweder ein Umweltobjekt oder ein Diskursweltobjekt sein. Auf die gleiche Weise wird eine betriebliche Transaktion in eine Anbahnungs-, eine Vereinbarungs-, eine Durchführungs-, eine Steuer- oder eine Kontrolltransaktion spezialisiert. Ein betriebliches Objekt kann mit einer bis beliebig vielen betrieblichen Transaktionen verbunden sein. Dagegen verbindet eine betriebliche Transaktion genau zwei betriebliche Objekte. Nähere Erläuterungen zum Metamodell der Ebene des Geschäftsprozessmodells enthalten [FeSi98, 343 f; FeSi95, 216; FeSi94, 6 f].

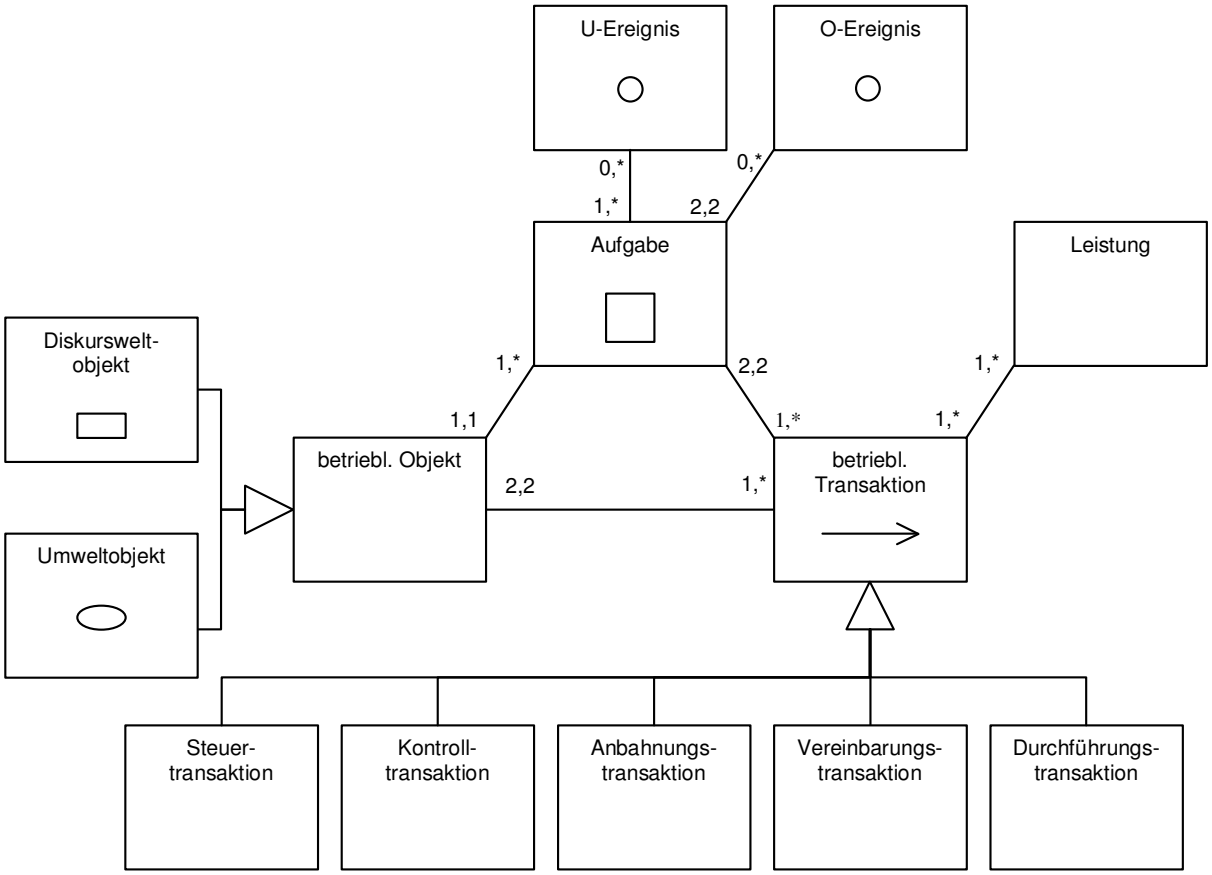


Abbildung 2.21: SOM-Metamodell für Geschäftsprozessmodelle [FeSi01, 199]

Wie erwähnt, sind auf dieser Modellebene eine strukturorientierte und eine verhaltensorientierte Sicht definiert. Die strukturorientierte Sicht auf ein Geschäftsprozessmodellsystem wird **Interaktionsschema (IAS)** genannt [FeSi01, 186]. In der zugehörigen Projektion auf das integrierte Metamodell sind die Metaobjekte betriebliches Objekt und betriebliche Transaktion einschließlich aller jeweiligen Spezialisierungen aufgenommen [FeSi94, 7]. Die verhaltensorientierte Sicht auf ein Geschäftsprozessmodellsystem wird als **Vorgangs-Ereignis-Schema (VES)** bezeichnet [FeSi01, 186]. Inhalt dieser Projektion auf das integrierte Metamodell sind die Metaobjekte Aufgabe, Umweltereignis und objektinternes Ereignis sowie das Metaobjekt Transaktion, das transaktionsgebundene Ereignisse repräsentiert [FeSi94, 8].

Darüber hinaus ist es möglich, die Erstellung von Geschäftsprozessmodellsystemen anhand von elementaren Strukturmustern, wie z. B. dem Verhandlungs- und dem Regelungsprinzip, hybriden Lenkungs-Strukturmustern und domänenspezifischen Strukturmustern zu unterstützen [Sinz02a, 1062].

2.4.1.2 Ebene der fachlichen Anwendungssystemspezifikation

Mit der SOM-Methodik wird ein Konzept zur expliziten Verknüpfung von betrieblichen Anwendungssystemen und Geschäftsprozessmodellsystemen zur Verfügung gestellt [FeSi98, 350]. Es beruht auf dem in Abschnitt 2.1.4 vorgestellten Konzept der Auto-

matisierung betrieblicher Aufgaben, ergänzt um das Konzept der Automatisierung betrieblicher Transaktionen [FeSi01, 200]. Das letztgenannte Konzept unterscheidet **vollautomatisierte** und **nicht-automatisierte Transaktionen**. Eine Transaktion ist vollautomatisiert, wenn sie durch ein Mensch-Computer- oder ein Computer-Computer-Kommunikationssystem durchgeführt wird, und nicht-automatisiert, wenn sie anhand eines Mensch-Mensch-Kommunikationssystem vollzogen wird [FeSi01, 200]. Analog zur Automatisierung betrieblicher Aufgaben muss vor der Festlegung des Automatisierungsgrades einer Transaktion deren Automatisierbarkeit festgestellt werden. Eine **Transaktion** ist **automatisierbar**, wenn für die Nachrichtenübertragung und die Durchführung des Kommunikationsprotokolls ein Mensch-Computer- oder ein Computer-Computer-Kommunikationssystem angebbar ist [FeSi01, 200; FeSi98, 351].

Die Automatisierbarkeit und die Automatisierungsgrade der betrieblichen Aufgaben sowie der betrieblichen Transaktionen werden im Interaktionsschema des Geschäftsprozessmodellsystems festgelegt [FeSi01, 200 f]. Daraufhin erfolgt die Abgrenzung eines Anwendungssystems anhand der automatisierten und teilautomatisierten Aufgaben eines oder mehrerer betrieblicher Objekte einschließlich der Ereignisse, die an die zugehörigen automatisierten Transaktionen gebunden sind [FeSi01, 202].

Metapher auf der Ebene der fachlichen Anwendungssystemspezifikation

Der Modellierung auf dieser Ebene liegt ebenfalls eine Konkretisierung der übergreifenden Metapher für die Modellierung der Innensicht im SOM-Ansatz zugrunde. Demzufolge wird jedes abgegrenzte Anwendungssystem als verteiltes System betrachtet, bestehend aus autonomen, lose gekoppelten Objekten, die zur Erreichung der gemeinsamen Ziele miteinander kooperieren. Alle abgegrenzten Anwendungssysteme bilden zusammen wiederum ein verteiltes System. Dabei ist jedes Anwendungssystem ebenfalls autonom und arbeitet zur Erreichung der gemeinsamen Ziele mit anderen Anwendungssystemen durch den Austausch von Nachrichten zusammen. Kein Anwendungssystem besitzt die globale Kontrolle über das gesamte Anwendungssystem. Zusammengefasst bedeutet dies, dass Anwendungssysteme als objektorientierte und objektintegrierte verteilte Systeme spezifiziert werden [FeSi01, 202 f; FeSi98, 353 f]. Genauere Ausführungen dazu folgen im weiteren Verlauf dieser Arbeit.

Die objektorientierte Spezifikation eines Anwendungssystems erfolgt in Form eines **konzeptuellen Objektschemas (KOS)** und eines darauf aufbauenden **Vorgangsobjektschema (VOS)** [FeSi01, 203].

Konzeptuelles Objektschema (KOS)

Ein konzeptuelles Objektschema besteht aus **konzeptuellen Objekttypen (KOT)**, die untereinander in verschiedenartigen Beziehungen stehen. Jeder KOT kapselt den Zustand einer automatisierten Aufgabe eines betrieblichen Objekts sowie die Zustände der korrespondierenden betrieblichen Transaktionen und Leistungen [FeSi98, 353]. Außerdem kapselt er die Operatoren zur direkten und exklusiven Bearbeitung der Zustände [FeSi98, 353]. Die initiale Struktur eines konzeptuellen Objektschemas lässt sich direkt aus dem Interaktionsschema und dem Vorgangs-Ereignisschema der Ebene des Geschäftsprozessmodells ableiten. Im weiteren Verlauf des Abschnitts wird ein Metamodell für die Beziehung zwischen der Ebene des Geschäftsprozessmodells und der Ebene der fachlichen Anwendungssystemspezifikation eingeführt, das unter anderem die Regeln für die Ableitung eines initialen KOS aus einem Geschäftsprozessmodellsystem enthält.

Vorgangsobjektschema (VOS)

Ein Vorgangsobjektschema enthält **Vorgangsobjekttypen (VOT)**, die die Kooperation der konzeptuellen Objekttypen und bzw. oder anderer Vorgangsobjekttypen bei der Durchführung einer voll- oder teilautomatisierten Aufgabe koordinieren [FeSi98, 354]. Somit legen sie den Workflow im Anwendungssystem fest [FeSi98, 354]. Die initiale Struktur eines VOS kann ebenso direkt aus dem Geschäftsprozessmodellsystem abgeleitet werden. Sie ist zum größten Teil identisch mit einem Ausschnitt aus dem Vorgangs-Ereignis-Schema, der die Aufgaben eines oder mehrerer betrieblicher Objekte umfasst [FeSi98, 354]. Die Regeln für diese Ableitung werden ebenfalls im noch folgenden Beziehungs-Metamodell beschrieben.

Metamodell der Ebene der fachlichen Anwendungssystemspezifikation

In Abbildung 2.22 wird das integrierte Metamodell der Ebene der fachlichen Anwendungssystemspezifikation dargestellt. Es basiert wiederum auf dem in 2.2.3 eingeführten Meta-Metamodell. Zu den Metaobjekten des integrierten Metamodells gehören ein Objekttyp, eine Beziehung und deren jeweilige Spezialisierungen. Ein Objekttyp kann ein **objektspezifischer**, ein **leistungsspezifischer** oder ein **transaktionspezifischer konzeptueller Objekttyp** oder aber ein **Vorgangsobjekttyp** sein. Auf die gleiche Weise wird eine Beziehung spezialisiert in eine ***ist_ein-***, eine ***interagiert_mit-*** und eine ***ist_Teil_von*-Beziehung**. Einem Objekttyp sind Operatoren und Attribute für die Bearbeitung und Speicherung des Zustands zugeordnet. Ferner schreibt das Metamodell vor, dass ein Objekttyp zwar mit keiner bis beliebig vielen Beziehungen verbunden sein kann, eine Beziehung jedoch genau zwei Objekttypen verbindet.

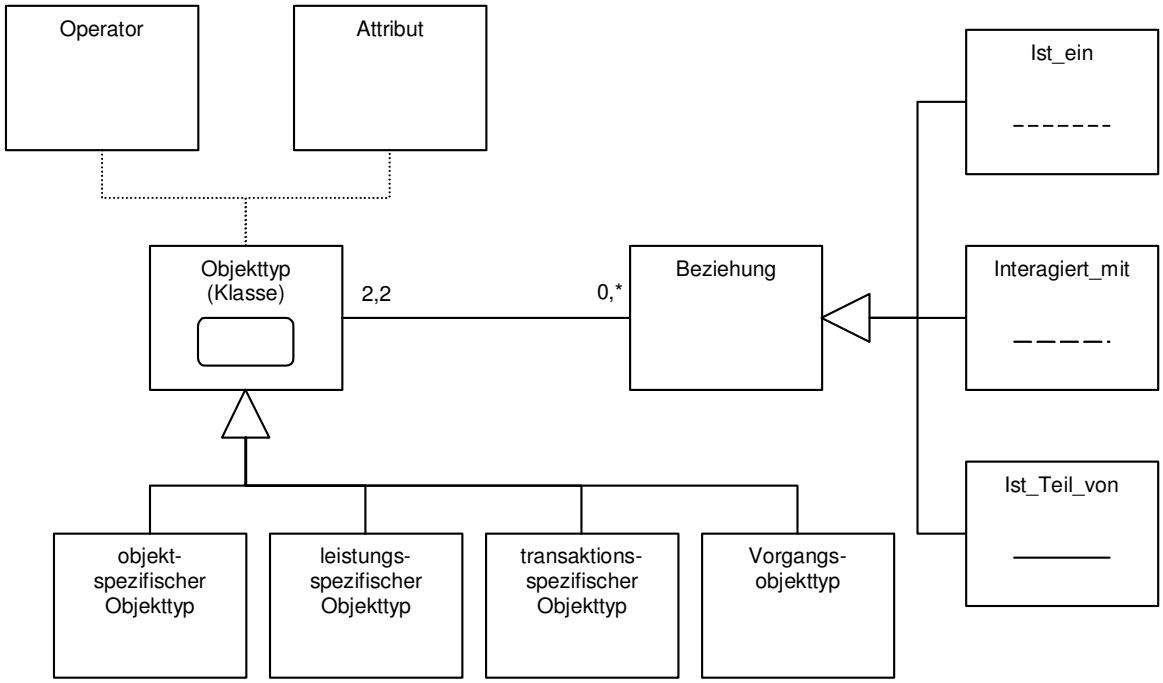


Abbildung 2.22: SOM-Metamodell für die Spezifikation von Anwendungssystemen [FeSi01, 213]

Ein KOS stellt die strukturorientierte Sicht und ein VOS die verhaltensorientierte Sicht auf das Modellsystem eines Anwendungssystems dar. Die zu einem KOS gehörende Projektion auf das integrierte Metamodell umfasst die Metaobjekte objektspezifischer, leistungsspezifischer und transaktionspezifischer konzeptueller Objekttyp sowie das Metaobjekt Beziehung einschließlich aller zugehörigen Spezialisierungen [FeSi98, 352]. Dagegen enthält die zu einem VOS gehörende Projektion auf das integrierte Metamodell die Metaobjekte Vorgangsobjekttyp und *interagiert_mit*-Beziehung [FeSi98, 353].

Für die Spezifikation eines Anwendungssystems stehen zusätzlich eine Vielzahl von Mustern aus dem Bereich des objektorientierten Entwurfs zur Verfügung (siehe z. B. [GHVJ96], [BMR+98]) [Sinz02a, 1062].

Beziehungs-Metamodell zur Integration der zweiten und dritten Modellebene

Schließlich stellt die SOM-Methodik ein Beziehungs-Metamodell zur Integration des Metamodells der Ebene des Geschäftsprozessmodells und des Metamodells der Ebene der fachlichen Anwendungssystemspezifikation zur Verfügung [FeSi98, 353]. Aus Übersichtlichkeitsgründen wird es im Folgenden in Form einer Tabelle dargestellt.

Ebene des Geschäftsprozessmodells	Ebene der fachlichen Anwendungssystemspezifikation
Betriebliches Objekt	Objektspezifischer Objekttyp
Leistung	Leistungsspezifischer Objekttyp und *interagiert_mit*-Beziehungen
Transaktion	Transaktionsspezifischer Objekttyp und *interagiert_mit*-Beziehungen
Aufgabe	Vorgangsobjekttyp
Transaktion / Ereignis	*interagiert_mit*-Beziehung zwischen Vorgangsobjekttypen

Tabelle 2.1: SOM-Beziehungs-Metamodell Geschäftsprozessmodelle – Spezifikation von Anwendungssystemen

Anhand dieses Beziehungs-Metamodells und ergänzenden Ableitungsregeln, die in [FeSi01, 202 ff] beschrieben werden, kann aus einem Geschäftsprozessmodellsystem, bestehend aus einem IAS und einem VES, unmittelbar ein initiales objektorientiertes Anwendungssystem, bestehend aus VOTs und KOTs abgeleitet werden. Weitere Erläuterungen dazu enthalten z. B. [FeSi01, 202 ff; FeSi98, 352 ff].

2.4.2 Vorgehensmodell

Die Modellierung erfolgt in der SOM-Methodik anhand eines Vorgehensmodells, das auf die Unternehmensarchitektur abgestimmt ist. Es umfasst drei Ebenen, die mit denen der Unternehmensarchitektur korrespondieren. Außerdem berücksichtigt es die struktur- und die verhaltensorientierten Sichten auf jeder Modellebene der Unternehmensarchitektur. In Abbildung 2.23 wird das Vorgehensmodell in Form des Buchstabens V visualisiert. Der linke Schenkel des Buchstabens V weist die strukturorientierten Sichten, der rechte Schenkel die verhaltensorientierten Sichten auf [FeSi01, 183; FeSi95, 213]. Mit den Abständen zwischen den beiden Schenkeln werden die Freiheitsgrade bei der Gestaltung der korrespondierenden Sichten auf den einzelnen Modellebenen symbolisiert [FeSi01, 184; FeSi95, 213]. Folglich nehmen diese Freiheitsgrade von der obersten zur untersten Modellebene ab.

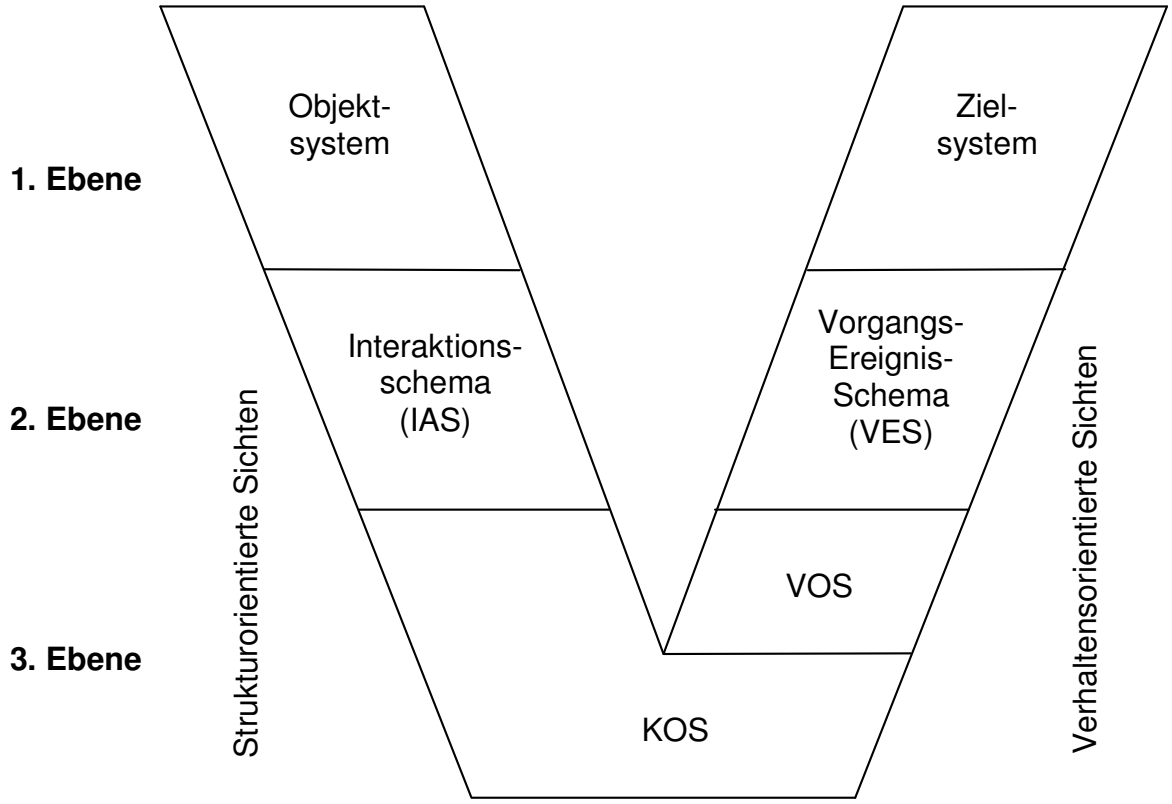


Abbildung 2.23: Vorgehensmodell der SOM-Methodik [FeSi01, 183]

Idealerweise erfolgt der Ablauf der Modellierung von der obersten zur untersten Modellebene, wobei die Modellierungsergebnisse in den korrespondierenden Sichten einer Ebene und zwischen den Sichten benachbarter Ebenen abzustimmen sind [FeSi01, 183; FeSi95, 213].

Auf der obersten Ebene wird der Unternehmensplan anhand eines **Objektsystems** in der strukturorientierten Sicht und eines **Zielsystems** in der verhaltensorientierten Sicht spezifiziert [FeSi01, 183; FeSi95, 213]. Im Objektsystem wird die Diskurswelt und ihre relevante Umwelt abgegrenzt sowie die zugehörigen Leistungsbeziehungen festgestellt. Korrespondierend dazu enthält das Zielsystem sowohl die Sach- und Formalziele des Unternehmens als auch die Strategien und Rahmenbedingungen für deren Umsetzung.

Die mittlere Ebene beschreibt die Spezifikation eines Interaktionsschemas als strukturorientierte Sicht und eines Vorgangs-Ereignis-Schemas als verhaltensorientierte Sicht auf ein Geschäftsprozessmodellsystem [FeSi01, 183]. Aufgrund der Möglichkeit, ein IAS und ein VES in verschiedenen Detaillierungsstufen darzustellen, müssen diese im Verlauf der Modellierung aufeinander abgestimmt werden [FeSi95, 213].

Auf der untersten Ebene erfolgt schließlich die fachliche Spezifikation von Anwendungssystemen zur Unterstützung der Geschäftsprozesse anhand eines konzeptuel-

len Objektschemas als strukturorientierte Sicht und eines Vorgangsobjektschemas als verhaltensorientierte Sicht [FeSi01, 184; FeSi95, 213].

Wie erwähnt, stellt die vorgestellte Reihenfolge der Modellierung von der obersten zur untersten Ebene einen idealtypischen Ablauf dar und kann nur dann angewendet werden, wenn für ein Unternehmen ein neuer Unternehmensplan erstellt und hierfür Geschäftsprozesse und zugehörige Anwendungssysteme zu spezifizieren sind [FeSi01, 184]. Dies ist in der Praxis jedoch selten der Fall. Stattdessen müssen häufig Geschäftsprozesse in Bezug auf die Ziele des Unternehmens verbessert oder bestehende Anwendungssysteme bereits überarbeiteten Geschäftsprozessen zugeordnet werden [FeSi01, 184; FeSi95, 213; FeSi98, 342 f]. Deswegen ist es im Hinblick auf die Ausgangssituation und den Modellierungszweck möglich, von diesem Ablauf abzuweichen, wobei jedoch die Abstimmung des gesamten Modellsystems immer von der obersten zur untersten Ebene erfolgen muss [FeSi01, 184; FeSi95, 213; FeSi98, 342 f].

3 Grundlagen des softwaretechnischen Entwurfs und der Implementierung wiederverwendbarer und verteilbarer Anwendungssysteme

In der vorliegenden Arbeit wird ein Software-Architekturmodell vorgestellt, das den softwaretechnischen Entwurf betrieblicher Anwendungssysteme unter besonderer Berücksichtigung der Formalziele Wiederverwendung, Wiederverwendbarkeit und Verteilbarkeit unterstützt. Auf der Grundlage des daraus resultierenden softwaretechnischen Modellsystems folgt schließlich die Implementierung des Anwendungssystems. In Anlehnung an die häufig vorzufindende Begriffsauffassung und aus Gründen der besseren Lesbarkeit wird im Folgenden der Begriff „Modell“ vereinfachend anstelle von „Modellsystem“ verwendet.

Wie im Verlauf des Abschnitts noch gezeigt wird, sind die genannten Formalziele inzwischen obligatorisch für die Entwicklung von Anwendungssystemen geworden. Es stellt sich somit nicht mehr die Frage, *ob* wiederverwendbare und verteilbare Anwendungssysteme erstellt werden sollen, sondern *wie* die Wiederverwendbarkeit und Verteilbarkeit von Anwendungssystemen bestmöglich realisiert werden kann. Zur Klärung dieser Frage müssen zunächst die mit diesen Formalzielen verbundenen Anforderungen an die Anwendungssystementwicklung im Allgemeinen und an den softwaretechnischen Entwurf und an die Implementierung von Anwendungssystemen im Speziellen eindeutig identifiziert werden. Damit beschäftigen sich die Abschnitte 3.1 bis 3.3.

Im darauf folgenden Abschnitt 3.4 werden Entwurfsprinzipien und -modelle zur Erfüllung der identifizierten Anforderungen vorgestellt und zueinander in Beziehung gesetzt. Sie dienen unter anderem als methodische Begründung des präskriptiven Grundkonzepts der Komponentenorientierung, das im weiteren Verlauf der Arbeit noch vorgestellt wird.

Schließlich werden in Abschnitt 3.5 ausgewählte Erzeugnisarten, die im softwaretechnischen Entwurf und in der Implementierung zur Verfügung stehen, hinsichtlich ihrer Unterstützung der identifizierten Anforderungen bewertet und anhand dieser Ergebnisse die Verwendung von Software-Komponenten im softwaretechnischen Entwurf und in der Implementierung von Anwendungssystemen begründet.

3.1 Formalziele Wiederverwendung und Wiederverwendbarkeit

Das erste Formalziel, das im Folgenden untersucht wird, ist die Wiederverwendbarkeit von Anwendungssystemen und -teilsystemen. Für den Einsatz wiederverwendbarer Anwendungssysteme und -teilsysteme in der Anwendungssystementwicklung muss darüber hinaus ein entsprechendes Vorgehen existieren. Deswegen gehört zu einer vollständigen Untersuchung des Formalziels Wiederverwendbarkeit auch die Untersuchung des Formalziels Wiederverwendung, das sich auf das Vorgehen der Entwicklung von Anwendungssystemen bezieht.

Als Voraussetzung für die Untersuchung der beiden Formalziele ist ein eindeutiges Verständnis der Begriffe „Wiederverwendbarkeit“ und „Wiederverwendung“ zwingend erforderlich.

3.1.1 Begriffliche Grundlagen

Für den Begriff „Wiederverwendung“ im Rahmen der Entwicklung von Anwendungssystemen existieren mehrere Definitionen aus unterschiedlichen Blickwinkeln (siehe dazu auch [Same97, 10; Schr01, 8 f; Küff94, 26 f, Hamm99, 78 ff]). Aus diesen Definitionen ist die Kernaussage der Wiederverwendung erkennbar, die sich folgendermaßen formulieren lässt: „Unter **Wiederverwendung** versteht man die Benutzung existierender Entwicklungsergebnisse in einem neuen Kontext“ [Raue96, 2]. Entwicklungsergebnisse können dabei auch treffender als **Entwicklungserzeugnisse** bezeichnet werden. Somit bezieht sich das Formalziel Wiederverwendung auf das Vorgehen der Entwicklung von Anwendungssystemen [Hamm99, 81]. Dies bezeichnet man auch als ***Development with Reuse*** [Same97, 172; Raue96, 3].

Dagegen bezieht sich das Formalziel **Wiederverwendbarkeit** auf das Ergebnis der Entwicklung von Anwendungssystemen [Hamm99, 81]. Es verlangt, dass gezielt wiederzuverwendende Entwicklungserzeugnisse erstellt werden sollen. Dies erfolgt im Rahmen des so genannten ***Development for Reuse*** [Same97, 172; Raue96, 3]. Ein Entwicklungserzeugnis ist wiederverwendbar, wenn es nützlich und benutzbar ist. Die **Nützlichkeit** wird bestimmt durch die fachlichen und inhaltlichen Bedürfnisse, die ein Entwicklungserzeugnis erfüllen sollte. Ergänzend dazu drückt die **Benutzbarkeit** aus, in wie weit ein Entwicklungserzeugnis seine Wiederverwendung unterstützt [MiMM95, 540]. Beide Forderungen werden im Folgenden genauer bestimmt. Die Formulierung des Formalziels Wiederverwendbarkeit erfolgt üblicherweise als zu forderndes Qualitätsmerkmal und zählt zu den nicht-funktionalen Anforderungen an ein zu entwicklendes Anwendungssystem.

Wiederverwendung von wiederverwendbaren Entwicklungserzeugnissen kann prinzipiell in jeder Phase der Entwicklung eines Anwendungssystems betrieben werden [Raue96, 4; Hamm99, 81]. Die vorliegende Arbeit legt ihren Fokus jedoch auf die

Phase des softwaretechnischen Entwurfs. Sachziel der Arbeit ist, ein Software-Architekturmodell zur Verfügung zu stellen, mit dem wiederverwendbare und verteilbare betriebliche Anwendungssysteme entworfen und auf der Grundlage des Entwurfs realisiert werden können. Daher untersuchen die folgenden Betrachtungen die Wiederverwendung und die Wiederverwendbarkeit von Entwicklungserzeugnissen nicht nur im softwaretechnischen Entwurf, sondern auch in der Implementierung.

3.1.2 Bedeutung der Wiederverwendung in der Software-Entwicklung

Eine **systematische Wiederverwendung** in der Software-Entwicklung verspricht insbesondere eine Verbesserung der Software-Qualität und eine Verbesserung der Produktivität der Software-Entwicklung [Endr88, 86; Raue96, 2; Same97, 11 ff]. Bereits seit Ende der 1960er Jahre sind die Vorteile einer systematischen Wiederverwendung bekannt [McIl69, 138 ff]. Seitdem wurden mehrere Ansätze zur Entwicklung und zum Einsatz wiederverwendbarer Software ausgearbeitet.

Bis vor ein paar Jahren konnten Software-Produzenten mithilfe der Wiederverwendung von Entwicklungserzeugnissen und den damit erzielbaren Verbesserungen der Produktqualität und der Entwicklungsproduktivität Wettbewerbsvorteile gegenüber Konkurrenten erzielen. Inzwischen ist die Wiederverwendung für Software-Unternehmen, die sich längerfristig am Software-Markt behaupten wollen, obligatorisch geworden. Dies liegt zum einen an Gründen, die bereits seit längerem bekannt sind, jedoch in letzter Zeit immer wichtiger für die Branche wurden, und zum anderen an Gründen, die sich erst durch die Entwicklungen in der Informations- und Kommunikationstechnologie und der Software-Technologie herauskristallisiert haben [Schr01, 10; HaMe02, 331]:

- Die Lebenszyklen von Produkten im Allgemeinen und von Software-Produkten im Speziellen verkürzen sich fortwährend. Dies ist auf der einen Seite durch die sich immer häufiger ändernden fachlichen Anforderungen begründet, die an sie gestellt werden, und auf der anderen Seite durch die hohe Innovationsgeschwindigkeit sowohl im Bereich der Informations- und Kommunikationstechnologien als auch im Bereich der Entwicklungs- und Ausführungsplattformen.
- Der Kostendruck auf Unternehmen wird aufgrund der Globalisierung der Wirtschafts-, Handels- und Informationsbeziehungen stetig größer .
- Die Größe und die Komplexität von Software-Systemen steigen weiter an.
- Die Kenntnisse und Fertigkeiten durchschnittlicher Software-Entwickler nehmen tendenziell ab, während die an sie gestellten Anforderungen ständig wachsen.
- Die Manager von Software-Entwicklungsprojekten unterschätzen meistens die dafür benötigten Ressourcen.

- Die Strukturen der Software-Industrie gleichen sich denen etablierter Branchen an. Dies lässt sich in einer stärkeren Arbeitsteilung und einer reduzierten Fertigungstiefe erkennen. „Insbesondere Hersteller von hochintegrierten Anwendungssystemen decken nicht mehr die gesamte Wertschöpfung ihrer Produkte selbst ab, sondern kaufen fallweise Komponenten zu“ [HaMe02, 331].

3.1.3 Aspekte des Formalziels Wiederverwendung

Die Software-Wiederverwendung wird üblicherweise anhand der **Wiederverwendungsobjekte** und der **Wiederverwendungsverfahren** untersucht [MiMM95, 528]. Eine genauere Betrachtung, die neben dem Wiederverwendungsobjekt und dem Wiederverwendungsverfahren weitere wesentliche Merkmale der Software-Wiederverwendung berücksichtigt, wird in [Same97, 22 ff] beschrieben. Die darin enthaltenen sechs **Aspekte der Software-Wiederverwendung** wurden erstmals von PRIETO-DÍAZ [Prie93, 62] formuliert. Anhand dieser ist es möglich, sowohl den Stand der Forschung und des praktischen Einsatzes der Wiederverwendung einzuschätzen als auch Probleme und Aspekte der Wiederverwendung zu analysieren [Prie93, 62]. Tabelle 3.1 gibt einen Überblick über die sechs Aspekte, die auch *Facets* genannt werden. In den darauf folgenden Abschnitten werden sie kurz erläutert.

Aspekt	**Ausprägungen**
Substanz	Konzept, Artefakt, Vorgehen
Bereich	vertikal, horizontal, intern, extern
Vorgehen	geplant, systematisch, ungeplant
Technik	kompositionell, generativ
Nutzungsart	Black Box, White Box
Erzeugnis	Algorithmus, Funktions- oder Routinebibliothek, Klassenbibliothek, Framework, Software-Komponente, Standardsoftware, Muster

Tabelle 3.1: Aspekte der Wiederverwendung (in Anlehnung an [Prie93, 62; Same97, 22])

3.1.3.1 Aspekt Substanz

Der Aspekt **Substanz** identifiziert das Wesen eines wiederzuverwendenden Entwicklungserzeugnisses.

Nach PRIETO-DÍAZ [Prie93, 62] können grundsätzlich **Konzepte**, **Artefakte** oder **Vorgehen** wiederverwendet werden. Ein generischer Algorithmus ist ein typisches Beispiel für ein wiederzuverwendendes Konzept. Komponenten und Klassen sind Bei-

spiele für Artefakte. Vorgehensmodelle für die Entwicklung wiederverwendbarer Software können exemplarisch für Vorgehen genannt werden [Prie93, 62 f].

3.1.3.2 Aspekt Bereich

Der Aspekt **Bereich** legt den Anwendungsbereich der Wiederverwendung fest.

Der Anwendungsbereich ist entweder auf eine bestimmte Domäne beschränkt oder umfasst mehrere unterschiedliche Domänen. Eine Wiederverwendung mit erstgenanntem Anwendungsbereich bezeichnet man auch als **vertikale Wiederverwendung**, eine mit letztgenanntem Anwendungsbereich als **horizontale Wiederverwendung** [Prie93, 63; Same97, 23]. Gängige Domänen sind z. B. Flugreservierungssysteme oder Finanzanwendungssysteme [Same97, 159].

Zusätzlich lässt sich ein interner und ein externer Anwendungsbereich unterscheiden. Bei einem **internen Anwendungsbereich** findet die Wiederverwendung nur innerhalb eines Anwendungssystems statt, bei einem **externen Anwendungsbereich** erstreckt sie sich über mehrere Anwendungssysteme [Same97, 23].

3.1.3.3 Aspekt Vorgehen

Der Aspekt **Vorgehen** bestimmt die verwendete Vorgehensweise bei der Wiederverwendung.

Die Vorgehensweise kann einerseits geplant und systematisch, andererseits ungeplant erfolgen [Endr88, 87; Prie93, 63 f; Same97, 41; Raue96, 3; Schr01, 14].

Eine **ungeplante Wiederverwendung** findet eher zufällig, opportunistisch und nichtwiederholbar statt und wird meistens mit existierendem Code durchgeführt [Endr88, 87; Prie93, 63; Schr01, 14; Raue96, 3]. Allerdings können auch Entwürfe, Spezifikationen und andere Entwicklungserzeugnisse ungeplant wiederverwendet werden. Mit anderen Worten wird eine „Wiederverwendung von bestehenden Entwicklungserzeugnissen durch das Wiederfinden über das Erinnerungsvermögen der Entwickler und anschließendem Einbau mit ‚Copy, Paste and Modify' beschrieben" [Küff94, 31]. Die Entscheidung zur Wiederverwendung findet in dieser Form somit ad hoc nach der Erstellung des Entwicklungserzeugnisses statt.

Dem gegenüber unterscheidet man bei der **geplanten Wiederverwendung** zwischen der gezielten Erstellung wiederzuverwendender Entwicklungserzeugnisse und der Erstellung neuer Entwicklungserzeugnisse unter dem systematischen Rückgriff auf bereits erstellte wiederverwendbare Entwicklungserzeugnisse [Raue96, 3]. Das bedeutet, dass die Entscheidung zur Wiederverwendung bereits von vornherein feststeht und somit bei der Entwicklung des wiederzuverwendenden Entwicklungserzeugnisses entsprechende Vorkehrungen getroffen werden können [Endr88, 86]. Die Erstellung des wiederzuverwendenden Entwicklungserzeugnisses kann dabei eben-

falls unter systematischem Rückgriff auf bereits erstellte wiederverwendbare Entwicklungserzeugnisse erfolgen.

3.1.3.4 Aspekt Technik

Der Aspekt **Technik** stellt das verwendete Verfahren der Wiederverwendung dar.

Es lassen sich zwei grundsätzliche Verfahren der Wiederverwendung unterscheiden, die alternativ oder auch gemeinsam verwendet werden können: Das kompositionelle und das generative Verfahren [Prie93, 64 f; Same97, 24; BiPe89, xxi; BiRi89, 2, Küff94, 28; MiMM95, 528]. Beiden Verfahren liegt das Prinzip der Abstraktion zugrunde, das eine wesentliche Forderung bei der Erzeugung wiederverwendbarer Entwicklungserzeugnisse darstellt [Same97, 24].

Beim **kompositionellen Verfahren** wird ein Anwendungssystem anhand mehrerer unveränderbarer Bausteine, auch Komponenten genannt, zusammengefügt oder erweitert [Prie93, 64 f; Same97, 24 f; BiRi89, 2]. Auf die gleiche Weise lassen sich auch benötigte komplexere Komponenten aus vorhandenen einfacheren Komponenten zusammensetzen. Die Komponenten stehen entweder bereits in einem so genannten ***Repository*** zur Verfügung, oder sie müssen anderenfalls neu erstellt werden [Same97, 24 f]. Im erstgenannten Fall findet eine Wiederverwendung statt, bei der die wiederzuverwendenden Komponenten an den neuen Kontext in geeigneter Weise angepasst werden müssen.

Das **generative Verfahren** findet bisher nur innerhalb einer Domäne statt. Dabei werden bei der Entwicklung oder Erweiterung eines Anwendungssystems keine Komponenten aus einem *Repository* wiederverwendet, sondern domänenspezifisches Entwurfswissen [Eise98, 79]. Dieses Entwurfswissen ist in Form von so genannten ***Patterns*** in einem Programm, auch ***Generator*** genannt, gespeichert [Prie93, 65; BiPe89, xxi; BiRi89, 3; Same97, 26]. Falls ein benötigtes *Pattern* nicht vorliegt, muss es, analog zum kompositionellen Verfahren, neu erstellt und zum *Generator* hinzugefügt werden. Der *Generator* erzeugt bzw. erweitert schließlich anhand der *Patterns* und einer gegebenen Menge elementarer Bausteine ein domänenspezifisches Anwendungssystem. Erfolgreiche Anwendungen des generativen Verfahrens, wie z. B. die *Application Generators* oder die *Language-Based Generators*, sind nur in der Implementierung bekannt [BiPe89, xxii ff; Same97, 26 f]. Die *Patterns* für die *Generator*s können vor der eigentlichen Anwendungssystementwicklung von Domänenexperten sorgfältig erstellt werden [Same97, 28].

Allgemein kann das generative Verfahren als Wiederverwendung von bewährtem Entwurfswissen bei der Erstellung oder Erweiterung von Anwendungssystemen charakterisiert werden. Dabei spielen die wiederzuverwendenden *Patterns* im Erstellungs- bzw. Erweiterungsprozess eines Anwendungssystems eine aktive Rolle, wäh-

rend die wiederzuverwendenden Komponenten im kompositionellen Verfahren beim Erstellungs- bzw. Erweiterungsprozess passiv sind.

Damit die genannten Vorteile beider Verfahren bestmöglich genutzt werden können, lässt sich das kompositionelle Verfahren mit dem generativen Verfahren kombinieren. Dabei werden allgemeinere Komponenten des kompositionellen Verfahrens als elementare Bausteine für das generative Verfahren genutzt [Same97, 28; Bigg98, 169 ff].

Hinsichtlich der Vorgehensweise stellen beide Verfahren eine geplante und systematische Wiederverwendung dar. Sie verbinden das *Development with Reuse* mit dem *Development for Reuse*, da bei der Entwicklung eines Anwendungssystems einerseits vorhandene Komponenten bzw. *Patterns* wiederverwendet werden können und andererseits fehlende Komponenten bzw. *Patterns* zur Wiederverwendung neu erstellt werden.

3.1.3.5 Aspekt Nutzungsart

Der Aspekt **Nutzungsart** beschreibt, wie ein Entwicklungserzeugnis wiederverwendet wird.

Es existieren grundsätzlich zwei orthogonale Nutzungsarten der Wiederverwendung: Die Black-Box- und die White-Box-Wiederverwendung [Prie93, 65; Same97, 28 ff; GHVJ96, 26; Grif98, 18 f]. Dazwischen können zusätzlich die Grey-Box- und die Glass-Box-Wiederverwendung eingeordnet werden [Same97, 28 ff].

Bei der so genannten **Black-Box-Wiederverwendung** sind die internen Details eines wiederzuverwendenden Entwicklungserzeugnisses, wie z. B. das Lösungsverfahren, weder sichtbar noch bekannt und deshalb auch nicht änderbar [Prie93, 65; Same97, 29; GHJV96, 26; Szyp98, 33]. Ein solches Entwicklungserzeugnis wird unverändert (*as is*) wiederverwendet [Same97, 29]. Deswegen muss es eine wohldefinierte Schnittstelle besitzen, die alle notwendigen Informationen für dessen Wiederverwendung enthält [Same97, 29; GHJV96, 26; Szyp98, 33]. So lässt sich das Entwicklungserzeugnis schließlich ohne Rücksicht auf bestimmte interne Details wiederverwenden. Das bedeutet umgekehrt, dass eine eventuelle Änderung oder sogar ein Austausch von bestimmten internen Details keine Auswirkungen auf die Wiederverwendbarkeit des entsprechenden Entwicklungserzeugnisses hat [Same97, 29; GHJV96, 27]. In der Implementierung entspricht eine Black-Box-Wiederverwendung einer Realisierung des Geheimnisprinzips, das in Abschnitt 3.4.2 noch näher beschrieben wird.

Die genau gegensätzliche Nutzungsart ist die **White-Box-Wiederverwendung**. Das bedeutet, dass die internen Details eines Entwicklungserzeugnisses bekannt, sichtbar und änderbar sind und zum Zwecke der Wiederverwendung üblicherweise auch

verändert werden [Prie93, 65; Same97, 28 ff; GHVJ96, 26; Szyp98, 33]. Solche Entwicklungserzeugnisse werden demnach nicht unverändert wiederverwendet, sondern durch Eingriff in die internen Details an den neuen Kontext angepasst. Dadurch entsteht ein neues, eigenständiges Entwicklungserzeugnis, das separat getestet und gewartet werden muss. Dies ist insbesondere dann hinderlich, wenn von einem wiederzuverwendenden Entwicklungserzeugnis mehrere eigenständige White-Box-Exemplare existieren und ein Detail, das allen Exemplaren gemeinsam ist, einer Änderung oder Korrektur bedarf [Same97, 29 f]. Außerdem besteht das Risiko, dass die Nutzung eines wiederzuverwendenden Entwicklungserzeugnisses zu stark von seinen sichtbaren internen Details abhängig gemacht wird. Grund dafür ist, daß das wiederzuverwendende Entwicklungserzeugnis häufig nur durch Untersuchung seiner internen Details verständlich wird [Szyp98, 34]. Dies hat zur Folge, dass Änderungen dieser Details zu Schwierigkeiten bei der Nutzung der Entwicklungserzeugnisexemplare führen können [Szyp98, 34].

Die reine Black-Box-Wiederverwendung verspricht eine höhere Flexibilität, Qualität und bessere Wiederverwendbarkeit der Entwicklungserzeugnisse als die White-Box-Wiederverwendung, da sie unverändert übernommen werden und ihre internen Details nach außen nicht preisgeben [Prie93, 65; Same97, 30]. Allerdings muss eine Black-Box-Wiederverwendung sorgfältig geplant und im Entwurf eines Anwendungssystems stets berücksichtigt werden [Same97, 30]. Demgegenüber ist eine reine White-Box-Wiederverwendung einfacher und deshalb auch kostengünstiger durchzuführen [Prie93, 65, Same97, 30]. Dabei muss man jedoch die oben genannten Nachteile in Kauf nehmen. Aufgrund der unterschiedlichen Vor- und Nachteile der beiden grundsätzlichen Nutzungsarten wurden zwei Mittelwege entwickelt.

Ein Mittelweg ist die **Grey-Box-Wiederverwendung**. Dabei sind von einem wiederzuverwendenden Entwicklungserzeugnis nur wenige, abgesicherte interne Details bekannt, sichtbar und änderbar [Same97, 30; Szyp98, 33].

Ein weiterer Mittelweg ist die **Glass-Box-Wiederverwendung**. Bei ihr sind die internen Details eines wiederzuverwendenden Entwicklungserzeugnisses zwar bekannt und sichtbar, können jedoch nicht geändert werden [Same97, 30; Szyp98, 33]. Diese Nutzungsart wird insbesondere dann angewendet, wenn einerseits Maßnahmen zur Vertrauensbildung in bzw. zum Wissenstransfer durch wiederzuverwendende Entwicklungserzeugnisse durchgeführt, aber gleichzeitig die genannten Nachteile der White-Box-Wiederverwendung vermieden werden sollen [Same97, 20]. Allerdings kann bei einer Glass-Box-Wiederverwendung, analog zur White-Box-Wiederverwendung, die Nutzung des wiederzuverwendenden Entwicklungserzeugnisses zu stark von seinen sichtbaren internen Details abhängig gemacht werden.

3.1.3.6 Aspekt Erzeugnis

Der Aspekt **Erzeugnis** bezeichnet die zugrundeliegende Art des wiederzuverwendenden Entwicklungserzeugnisses.

Seitdem bekannt ist, dass mithilfe der Wiederverwendung eine Qualitäts- und Produktivitätssteigerung bei der Entwicklung von Anwendungssystemen erreicht werden kann, sind mehrere Arten wiederverwendbarer Entwicklungserzeugnisse entstanden. Zur Beschreibung und Bewertung dieser Arten bedarf es einer geeigneten Kategorisierung.

Ansätze zur Kategorisierung wiederverwendbarer Entwicklungserzeugnisse

Einen der ersten Vorschläge enthält [Endr88, 88 ff]. Darin werden die Erzeugnisarten eingeteilt in die Kategorien Programme, Schablonen und Bausteine. Diese Kategorisierung beschränkt sich jedoch nur auf Erzeugnisarten in der Implementierung und lässt dabei Erzeugnisarten im softwaretechnischen Entwurf außer Betracht. Somit ist dieser Kategorisierungsansatz für die vorliegende Arbeit unbrauchbar.

In [Küff94, 66 ff] wird eine ähnliche Einteilung vorgeschlagen. Die Erzeugnisarten werden hierbei in die Kategorien Module, Schablonen, Entwürfe (*Designs*), Entwurfswissen und Spezifikationen untergliedert. Zwar beziehen die Kategorien Entwürfe und Entwurfswissen auch Erzeugnisarten des softwaretechnischen Entwurfs mit ein und beseitigen damit den Mangel des oben erläuterten Vorschlags. Jedoch sind die vorliegenden Kategorien nicht orthogonal zueinander. Zum Beispiel existieren designabhängige Schablonen und Programme [Küff94, 83 f].

Auch der Vorschlag, der in [Same97, 31 ff] beschrieben wird, bezieht Erzeugnisarten des softwaretechnischen Entwurfs mit ein. Hierbei werden so genannte Typen wiederverwendbarer Artefakte unterschieden: Daten, Architekturen, Entwürfe und Programme. Zur Erläuterung der unterschiedlichen Typen führt SAMETINGER beispielhafte Ausprägungen an: Für wiederverwendbare Daten werden standardisierte Datenformate genannt, für Architekturen eine Menge von Entwurfs- und Implementierungsvereinbarungen, die sich mit der logischen Organisation eines Software-Produkts befassen, für Entwürfe häufig verwendete Anwendungssystementwürfe und für Programme ausführbarer Code. Allerdings sind auch die aufgeführten Typen dieses Vorschlags weder orthogonal noch vollständig. Es stellt sich beispielsweise die Frage, welchem Typ Software-Komponenten zuzuordnen sind.

Ein anderer Weg der Kategorisierung wird in [MiMM95, 529] vorgestellt. Er löst die Probleme der oben genannten Kategorisierungsansätze, indem er auf einer höheren Abstraktionsebene angesiedelt ist. Dieser Kategorisierungsvorschlag benennt drei Kriterien, die einzeln oder in unterschiedlichen Kombinationen angewendet werden können:

- Die **Phase** des **Entwicklungsprozesses**, in der ein wiederzuverwendendes Entwicklungserzeugnis erstellt oder wiederverwendet wird,
- die **Abstraktionsebene** eines **wiederzuverwendenden Entwicklungserzeugnisses** (z. B. abstrakte Beschreibung versus konkrete Realisierung eines wiederzuverwendenden Algorithmus) und
- das **Wesen** des **wiederzuverwendenden Entwicklungserzeugnisses** (z. B. Artefakt versus Konzept).

Diese drei Kriterien stehen orthogonal zueinander und können somit für eine Kategorisierung der Erzeugnisarten verwendet werden. Hinsichtlich des erstgenannten Kriteriums werden die in Abschnitt 2.2.5.3 beschriebenen Phasen Planung, Fachentwurf, softwaretechnischer Entwurf, Implementierung und Einführung unterschieden. Da sich das Ziel der vorliegenden Arbeit, nämlich die Konzeption eines komponentenbasierten Software-Architekturmodells, auf den softwaretechnischen Entwurf bezieht und die Ergebnisse des softwaretechnischen Entwurfs als Grundlage für die darauffolgende Implementierung dienen, betrachten die folgenden Ausführungen auch nur Erzeugnisarten in diesen beiden Phasen. Darin werden derzeit insbesondere folgende Erzeugnisarten diskutiert und eingesetzt [Same97, 31 ff; Eise98, 76; Schr01, 17 ff; FSH+98, 31; MeHL97, 29 ff; Hamm99, 92 ff].

Algorithmus

Algorithmen stellen Lösungen für allgemeine, nicht-komplexe und immer wiederkehrende Problemstellungen dar und werden insbesondere im Rahmen der Implementierung wiederverwendet. Dabei können sie entweder in Textform als Beschreibungen bzw. Pseudocode oder in realisierter Form als generische Implementierung oder *Templates* vorliegen [Same97, 31]. Beispiele dafür sind Sortier- oder Suchalgorithmen [Same97, 32; Meye90, 31].

Funktions- oder Routinebibliothek

Der Begriff „**Routine**“ kann als Oberbegriff für Prozedur, Funktion, Unterprogramm, etc. verwendet werden [Meye90, 36]. Eine Routine implementiert eine wohldefinierte Operation in einer bestimmten Programmiersprache, die in verschiedenen Programmen wiederverwendet werden kann [Meye90, 36; Same97, 32]. Somit sind Routinen ausschließlich in der Implementierung verwendbar. Sie werden einerseits zur Lösung komplexerer Problemstellungen entwickelt und bestehen dann gegebenenfalls aus mehreren implementierten Algorithmen. Ein Beispiel dafür sind **Routinebibliotheken** für wissenschaftliche Berechnungen [Meye90, 36]. Andererseits können sie auch aus einer Top-Down-Zerlegung eines Anwendungssystems bei der Spezifikation entstehen, wie beispielsweise Ein- und Ausgaberoutinen [Meye90, 36]. Die Wiederverwendung von Routinen unterliegt einigen Einschränkungen, die in [Meye90, 36 f] zu-

sammengefasst sind. Ein wesentliches Problem bei der Verwendung von Routinen ist die Handhabung komplexer Datenstrukturen. Sie müssen gegebenenfalls auf mehrere sie verwendende Routinen aufgeteilt werden, womit die Eigenständigkeit und Unabhängigkeit der beteiligten Routinen verletzt wird. Dieses Problem kann mithilfe des in Abschnitt 3.5.1.1 beschriebenen Grundkonzepts der Objektorientierung behoben werden, da hierbei die Datenstrukturen im Vordergrund stehen und die Funktionen als Operatoren der Datenstrukturen spezifiziert werden.

Klassenbibliothek

Das Grundkonzept der Objektorientierung, in dem die **Klasse** definiert ist (vgl. Abschnitt 3.5.1.1), stellt einen vorläufigen Schlusspunkt einer Entwicklung von einer prozeduralen Sichtweise zu einer objektorientierten Sichtweise auf ein Anwendungssystem dar [Booc94, 30 ff]. Somit können **Klassenbibliotheken** als objektorientierte Ausgabe von Routinebibliotheken angesehen werden und sind demzufolge ebenfalls nur in der Implementierung einsetzbar [Same97, 32; GHJV96, 36]. Klassen versprechen aufgrund der Konzepte Vererbung, Polymorphie und dynamisches Binden des Grundkonzepts der Objektorientierung eine bessere Abstraktion, Änderbarkeit und Anpassbarkeit und damit eine bessere Wiederverwendbarkeit als Routinen [Same97, 32]. Allerdings können diese Konzepte bei unbedachter und übermäßiger Verwendung die Wiederverwendbarkeit von Klassen auch einschränken bzw. vollständig verhindern. Diese Problematik wird zusammen mit weiteren Defiziten des Grundkonzepts der Objektorientierung in Abschnitt 3.5.1.1 ausführlich diskutiert.

Framework

Ein **Framework** stellt einen wiederverwendbaren softwaretechnischen Entwurf für eine kontextabhängige Problemstellung in Form einer Implementierung dar. Dafür besteht es aus der Struktursicht aus einer Menge von abstrakten und konkreten Klassen, die durch Interaktionsbeziehungen miteinander verknüpft sind [GHJV96, 37; Pree97, 7; Stri92, 29; Szyp98, 137; Hamm99, 97]. Die Existenz der Interaktionsbeziehungen ist ein wesentliches Unterscheidungsmerkmal zu Klassenbibliotheken. Aus der Verhaltenssicht besitzen die Objekte der Framework-Klassen über die Interaktionsbeziehungen ein definiertes Kooperationsverhalten [GHJV96, 37; Pree97, 7; Hamm99, 97]. Ein Framework stellt somit eine Software-Architektur für eine durch die Problemstellung definierte Klasse von Anwendungssystemen zur Verfügung [GHJV96, 37; Szyp98, 137; BMR+98, 427]. Diese kann an die Anforderungen eines bestimmten Anwendungssystems angepasst werden, indem die abstrakten Klassen des Frameworks durch anwendungsspezifische Unterklassen spezialisiert werden [GHJV96, 37; Hamm99, 97]. Die angesprochenen abstrakten Klassen befinden sich an bestimmten vordefinierten Stellen im Framework und werden auch ***Hot Spots*** genannt [Pree97, 7; Hamm99, 97]. Über die Interaktionsbeziehungen wird der Steue-

rungsfluss eines darauf basierenden Anwendungssystems bestimmt. Dies führt hinsichtlich der Wiederverwendung im Vergleich zu einer Klassenbibliothek zu einer Umkehrung der Steuerung (vgl. Abbildung 3.1): Bei der Wiederverwendung von Klassen einer Klassenbibliothek wird die Architektur einschließlich der Ablaufsteuerung des Anwendungssystems implementiert und daraus die wiederzuverwendenden Klassen aufgerufen. Die Wiederverwendung eines Frameworks entspricht dem umgekehrten Fall: Das Framework gibt die Architektur inklusive der Ablaufsteuerung des Anwendungssystems vor und ruft von sich aus die anwendungspezifischen Unterklassen auf, die implementiert werden müssen [Hamm99, 97; Pree97, 19 f; John97, 12]. Eine ausführlichere Diskussion über die Wiederverwendbarkeit von Frameworks im Vergleich zu anderen Wiederverwendungsprodukten enthält der Abschnitt 3.5.2.

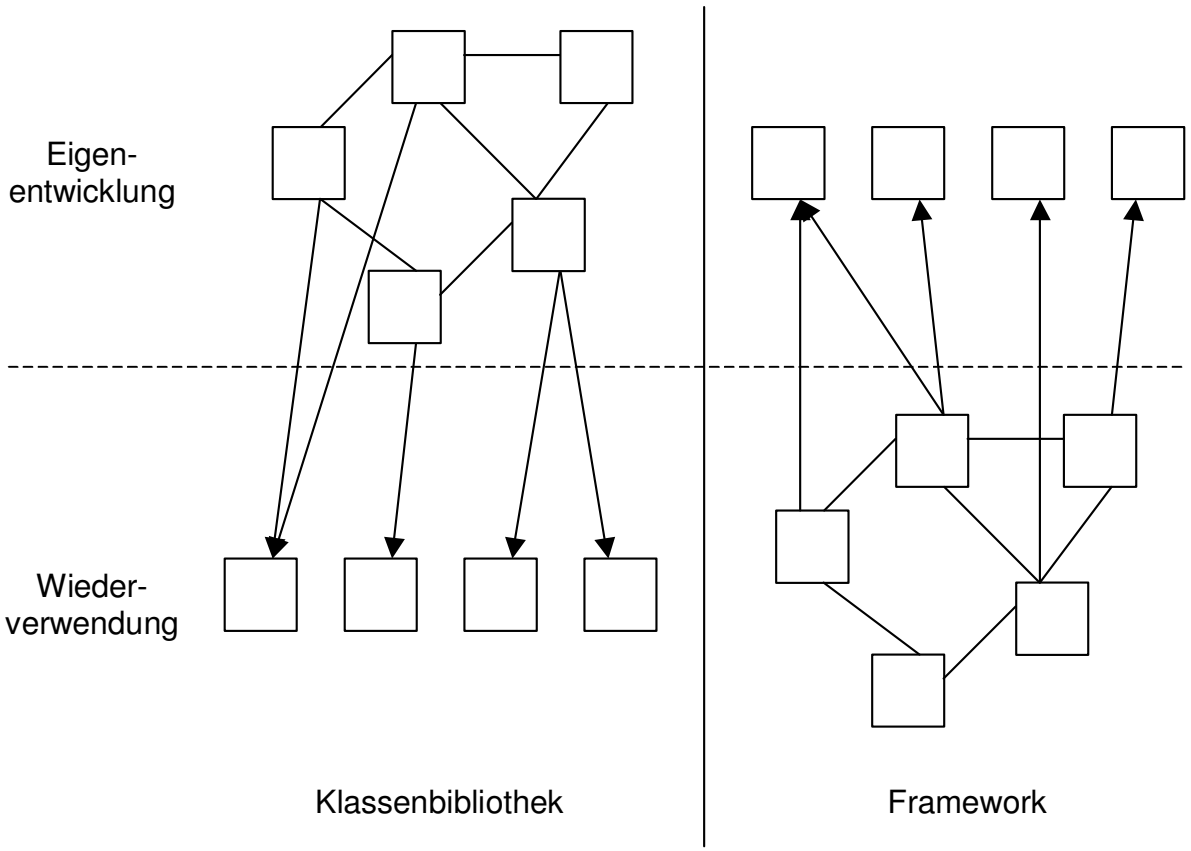


Abbildung 3.1: Framework- contra Klassenbibliothek-Wiederverwendung (in Anlehnung an [Pree97, 17])

Software-Komponente

Zum Begriff „Software-Komponente“ existieren eine Reihe von unterschiedlichen Definitionen (siehe z. B. [Szyp98, 34; Same97, 68; Pree97, 5 f; Grif98, 31; Hopk00, 27; Whit02, 18 ff; CoHe01, 6 ff]). Sie reichen vom allgemeinen wiederverwendbaren Software-Teilsystem [CaLo00, 83], über einen Oberbegriff für bereits vorhandene Erzeugnisarten, wie Modul oder Klasse [Wegn89, 46; Küffm94, 53; John97], bis zu

einer eigenständigen Erzeugnisart als Weiterentwicklung der Erzeugnisart Klasse [Hopk00, 27; WeSa01, 36; JäHe02, 273 f; Grif98, 32 f]. Aufgrund dieser Begriffsvielfalt muss für ein erstes Verständnis der kleinste gemeinsame Nenner aller verwendeten Definitionen gefunden werden. Dieser beschreibt eine **Software-Komponente** als wiederverwendbares Software-Teilsystem, das mit anderen Software-Teilsystemen zu einem Gesamtsystem verbunden werden kann. Die Wiederverwendbarkeit des Software-Teilsystems in dieser Definition impliziert noch weitere Eigenschaften, auf die im Laufe des Kapitels näher eingegangen wird. Allerdings ist diese Definition aufgrund ihrer Generalität nicht operational genug für weitere Untersuchungen. Deswegen wird in Abschnitt 4 auf der Basis der in diesem Abschnitt erarbeiteten Kriterien ein für das Ziel der Arbeit geeignetes Grundkonzept der Komponentenorientierung erarbeitet, in dem auch ein zweckmäßigerer Komponentenbegriff enthalten ist.

Standardsoftware

Standardsoftware wird allgemein definiert als „Anwendungssoftware, die auf der Grundlage prognostizierter Anforderungen (anonymer Markt) für wiederholt vorkommende Aufgabenstellungen bei unterschiedlichen Anwendern entwickelt wird.“ [Ambe99b, 40]. Folglich werden beim Einsatz von Standardsoftware im Rahmen der Entwicklung von Anwendungssystemen nicht einzelne Klassen oder Komponenten wiederverwendet, die zusammengesetzt werden müssen, sondern eine auslieferungsfertige Anwendungssoftware in implementierter Form. Im Vergleich zu einem Framework liegt Standardsoftware jedoch nicht in Quellcode, sondern in Bytecode vor und kann nur durch Konfigurierung, Parametrisierung oder durch Anfügen zusätzlicher Komponenten an spezifische Anforderungen angepasst werden [Ambe99b, 41]. Standardsoftware wird für allgemeine, meist betriebliche Problemstellungen entwickelt und üblicherweise von einer Vielzahl von Anwendern eingesetzt. Sie ist immer eine Vorratsfertigung von vorwiegend größeren Software-Unternehmen. In Abschnitt 3.5 wird näher auf die Vor- und Nachteile des Einsatzes von Standardsoftware bei der Entwicklung von Anwendungssystemen eingegangen.

Muster

Muster, die im Zusammenhang mit der Entwicklung von Anwendungssystemen auch *Patterns* genannt werden, wurden bereits in Kapitel 2 kurz angesprochen. Ein **Muster** ist im allgemeinen Sprachgebrauch entweder eine Anleitung zum Erzeugen einer bewährten Lösung für ein immer wiederkehrendes Problem (Vorlage) oder eine Darstellung einer bewährten Lösung für ein immer wiederkehrendes Problem (Vorbild) oder eine ausschließliche Darstellung der Struktur einer bewährten Lösung für ein immer wiederkehrendes Problem (sich wiederholende Struktur) [Quib96, 326]. In allen drei Definitionen stellt ein Muster eine bewährte Lösung für ein immer wiederkeh-

rendes Problem dar. Bei genauerer Betrachtung sind obige Definitionen auch nicht substitutiv, sondern definieren komplementäre Sichten auf dasselbe Problem-Lösungs-Paar. Eine Vorlage entspricht einer generativen Sicht, ein Vorbild und eine sich wiederholende Struktur einer deskriptiven Sicht.

Anfänge der Musterverwendung in der Anwendungssystementwicklung

Die Verwendung von Mustern zur Unterstützung der Entwicklung von Anwendungssystemen wurde durch den Architekten CHRISTOPHER ALEXANDER inspiriert, der in seinen Büchern „A Timeless Way Of Building“ [Alex79] und „A Pattern Language“ [Alex77] eine neue, zeitlose Art des Entwerfens von Gebäudearchitekturen mithilfe von Mustern beschreibt. Seit Mitte der 1990er Jahre werden Muster hauptsächlich in der objektorientierten Anwendungssystementwicklung eingesetzt. Neben anderen bedeutenden Arbeiten gelten insbesondere das Buch von GAMMA ET AL. [GHJV96] und die Web-Site der Hillside-Group [Hill03], die sich als erste Web-Site mit Mustern beschäftigte, als Auslöser für die weltweite Verbreitung. Inzwischen werden jährliche Konferenzen zum Thema Muster in der objektorientierten Anwendungssystementwicklung abgehalten, auf denen bereits bekannte und bewährte Muster gesammelt und systematisiert werden sowie die Formulierung neuer Muster angeregt wird (siehe z. B. [CoSc95; VlCK96; MaRB98; Risi98]). Üblicherweise wird jedes neue Muster vor seiner Veröffentlichung auf einer dieser Konferenzen vorgestellt und diskutiert.

Nutzen der Musterverwendung in der Anwendungssystementwicklung

Der Nutzen, den der Einsatz von Mustern für die Anwendungssystementwicklung erbringt, ist vielfältig. Zum einen ermöglichen Muster einen qualitativ hochwertigen und verständlichen Entwurf eines Anwendungssystems mit klar definierten Eigenschaften, da sie Abstraktionen auf einer Ebene oberhalb von Klassen und Objekten bzw. Komponenten identifizieren und spezifizieren [GHJV96, 4; BMR+98, 6]. Auf diese Weise verlieren sich Entwickler nicht im Detail, sondern entwerfen zunächst eine Gesamt- bzw. Teilarchitektur, die dann konkretisiert wird. Beispielsweise unterstützt das *Model-View-Controller*-Muster [KrPo88; BMR+98, 3 f] die Erfüllung der nichtfunktionalen Eigenschaften Wiederverwendbarkeit und Anpassbarkeit. Ferner ist das Suchen und Verwenden von passenden Mustern meistens effizienter als das „Wiederentdecken“ von eigenen Lösungen. Muster schaffen außerdem ein gemeinsames Vokabular, das eine effiziente Kommunikation zwischen den Entwicklern erlaubt [BMR+98, 6; GHJV96, 432; Quib96, 327]. Darüber hinaus eignen sich Muster auch zur Dokumentation von Software-Architekturen bzw. Frameworks. Dabei reduzieren sie nachweislich den Lern- und Einarbeitungsaufwand für die Nutzer von Software-Architekturen oder Frameworks [GHJV96, 38].

Merkmalsbezogene Definition eines Musters

Für die Verwendung von Mustern zur Unterstützung der Anwendungssystementwicklung ist die oben angegebene allgemeine Definition zu abstrakt. Deshalb wird im Folgenden eine merkmalsbezogene Definition eines Musters zu Grunde gelegt. Danach besitzt ein Muster

- einen eindeutigen und prägnanten **Namen**,
- eine Beschreibung eines immer wiederkehrenden Entwurfsproblems in Form von **Entwurfsobjekt** und **Entwurfssachzielen**,
- eine generische Beschreibung des **Kontextes**, in dem das Entwurfsproblem auftritt,
- eine Beschreibung aller bei der Lösung des Entwurfsproblem zu beachtenden **Formalziele**, auch Kräfte genannt,
- eine Beschreibung eines **generischen Lösungsverfahrens**, das das beschriebene Entwurfsproblem in seinem Kontext und unter Beachtung der beschriebenen Formalziele löst und
- eine Beschreibung einer **generischen Entwurfsstruktur**, die das Ergebnis der Anwendung des generischen Entwurfsverfahrens visualisiert,
- eine Beschreibung der **Beziehungen** zu anderen Mustern und
- gegebenenfalls eine Beschreibung mehrerer **konkreter Entwurfssituationen**, in denen das Muster erfolgreich angewendet wurde.

Beschreibungsform eines Musters

Ausschlaggebend für die Nützlichkeit eines Musters ist dessen Beschreibungsform [BMR+98, 19; VöSW02, 9]. Die Auswahl der Beschreibungsform wird bestimmt durch den Anspruch und die Komplexität des Musters sowie durch den Adressatenkreis, der das Muster lesen und verstehen soll [BMR+98, 19; VöSW02, 10]. ALEXANDER wählte für seine Muster eine eher prosaische Form, die jedoch alle genannten Merkmale eines Musters erfasst [Alex77; Alex79]. Diese Beschreibungsform wird auch ***alexandrinische Grundform*** genannt. Im Bereich der objektorientierten Anwendungssystementwicklung wird häufig die von GAMMA ET AL. gewählte Beschreibungsform, auch ***Gang-of-Four (GoF)*-Form** genannt, verwendet [GHJV96]. Sie ist strukturierter als die alexandrinische Grundform und ergänzt sie um einen Bereich für Konsequenzen aus der Anwendung des Musters und einen Bereich für Verweise auf alternative Muster. Weitere Beschreibungsformen, die im Bereich der objektorientierten Anwendungssystementwicklung häufiger verwendet werden, sind die **BUSCHMANN-Form** [BMR+98] und die **COPLIEN-Form** [Copl91]. Beide sind der GAMMA-Form

inhaltlich sehr ähnlich und unterscheiden sich zumeist nur in den Bezeichnern. In der vorliegenden Arbeit wird zur Dokumentation der Muster des noch folgenden Software-Architekturmodells eine eigene, strukturierte Beschreibungsform gewählt, die sich an den oben genannten Merkmalen eines Musters orientiert.

Beziehungsarten zwischen Mustern

Muster existieren nicht isoliert voneinander, sondern müssen zur Lösung von Entwurfsproblemen in der Regel zueinander in Beziehung gesetzt werden [BMR+98, 16]. Die am häufigsten verwendeten Beziehungsarten sind die Spezialisierung, die Alternative bzw. Variante und die Kombination [BMR+98, 17 ff; GHJV96, 16 f; VöSW02, 11 f]. Muster, die für einen bestimmten Problembereich in Beziehung gesetzt wurden, bilden ein **Mustersystem** für diesen Problembereich [BMR+98, 357 f; VöSW02, 8 f]. Eine restriktive Variante eines Mustersystems ist eine **Mustersprache** [VöSW02, 9]. Anhand einer Mustersprache sollen Entwickler zu einem sprachspezifischen Ziel geführt werden. Dafür ist es erforderlich, dass die Muster einer Mustersprache in einer bestimmten Reihenfolge angewendet werden. Folglich muss jedes Muster einer Mustersprache seinen Ort in dieser Reihenfolge angeben können. Mustersprachen sind mächtiger als Mustersysteme, da die Muster einer Mustersprache nicht nur einzelne Probleme lösen, sondern zusätzlich die Erfüllung eines übergreifenden, sprachspezifischen Ziels unterstützen [VöSW02, 9].

Musterkategorien

Damit Muster für die Anwendungssystementwicklung einfacher aufgefunden und eingeordnet werden können, ist eine geeignete Kategorisierung der Muster notwendig. Eine der bekanntesten Kategorisierungen, die jedoch nur Muster für den softwaretechnischen Entwurf von Anwendungssystemen berücksichtigt, ist die von BUSCHMANN ET AL. [BMR+98, 360 f]. Sie unterscheidet zwischen

- **Architekturmustern**, die ein grundsätzliches Strukturierungsprinzip von Software-Systemen widerspiegeln,
- **Entwurfsmustern**, die Subsysteme oder Komponenten eines Software-Systems oder den Beziehungen zwischen ihnen verfeinern und
- **Idiomen**, die für eine bestimmte Programmiersprache spezifische Muster auf einer niedrigen Abstraktionsebene darstellen.

Innerhalb solcher Kategorien können Muster noch weiter klassifiziert werden. Beispielsweise ordnen GAMMA ET AL. [GHJV96, 14 f] ihre Entwurfsmuster anhand von zwei Dimensionen. Die erste Dimension Aufgabe besitzt die Ausprägungen **Erzeugungsmuster**, **Strukturmuster** und **Verhaltensmuster**. Orthogonal dazu weist die zweite Dimension Gültigkeitsbereich die Ausprägungen **klassenbasierte Muster** und **objektbasierte Muster** auf. Eine Erweiterung der BUSCHMANN-Kategorisierung wird

in den Tagungsbänden der erwähnten Musterkonferenzen verwendet (siehe z. B. [CoSc95; VlCK96; MaRB98]). Sie umfasst

- domänenunabhängige/-spezifische Entwurfsmuster,
- domänenunabhängige/-spezifische Architekturmuster,
- Implementierungsmuster und Idiome,
- Muster für den Entwicklungsprozess und die -organisation und
- Muster für die Musterentwicklung und -verbreitung.

An dieser Kategorisierung orientiert sich auch die vorliegende Arbeit.

Beziehungen zwischen Mustern und Frameworks

Wie bereits angedeutet, ergänzen sich Muster und Frameworks konzeptionell gegenseitig: Da Muster Konzepte für den softwaretechnischen Entwurf wiederverwendbar machen, können sie sowohl zur Dokumentation als auch zur Anpassung eines Frameworks an spezifische Problemstellungen verwendet werden [GHJV96, 37 f; FSH+98, 34]. Allerdings beschreiben Muster, im Gegensatz zu den bisher genannten Erzeugnisarten, generische Entwurfsverfahren und stellen keine konkreten Entwurfsergebnisse dar, die in der Implementierung wiederverwendet werden können.

3.1.3.7 Wiederverwendungsunterstützende Aspektausprägungen

Für jede der genannten Aspekte lässt sich eine Ausprägung oder eine Ausprägungskombination angeben, die angestrebt werden sollte, um eine bestmögliche Wiederverwendung im softwaretechnischen Entwurf und in der Implementierung zu erreichen.

Entgegen der ursprünglichen Meinung von PRIETO-DÍAZ [Prie93, 62] sind die genannten Aspekte jedoch nicht orthogonal zueinander. Vielmehr determiniert eine ausgewählte Erzeugnisart einen bestimmten Satz an Ausprägungen der übrigen Aspekte. Diese wiederum sind orthogonal zueinander, sind also unabhängig voneinander angebbar. Darum wird zur folgenden Bestimmung der Aspektausprägungen der Aspekt Erzeugnis ausgenommen. In der Formulierung von Forderungen bezüglich der Entwicklung wiederverwendbarer und verteilbarer Anwendungssysteme, die in Abschnitt 3.3 folgt, wird unter anderem auf die folgenden Ausprägungen Bezug genommen. Außerdem dienen diese Forderungen der Ausarbeitung eines präskriptiven Grundkonzepts der Komponentenorientierung in Abschnitt 4.

Aspekt Substanz

Hinsichtlich der Substanz können die Ausprägungen Konzepte, Artefakte und Vorgehen unterschieden werden. Wiederzuverwendende Vorgehen, wie z. B. Vorgehens-

modelle zur Entwicklung wiederverwendbarer Software, schließen alle Phasen der Entwicklung ein. Da sich die folgenden Ausarbeitungen jedoch nur auf die softwaretechnische Entwurfsphase und die Implementierungsphase beziehen, ist diese Ausprägung zu umfassend und wird im Folgenden nicht weiter berücksichtigt. In den genannten Phasen befinden sich allerdings Konzepte und Artefakte, die wiederverwendbar sein können und sollen. Deshalb gehören diese zu den ausgewählten Ausprägungen dieses Aspekts.

Aspekt Bereich

Der Bereich der Wiederverwendung kann entweder vertikal in einer bestimmten Domäne oder horizontal über mehrere Domänen hinweg bestehen. Durch eine vertikale Wiederverwendung kann ein höherer Wiederverwendungsgrad erreicht werden als durch eine horizontale [Prie93, 63; Same97, 23]. Andererseits wird durch eine horizontale Wiederverwendung eine höhere Wiederverwendungshäufigkeit erzielt. Folglich müssen für eine bestmögliche Wiederverwendung beide Bereiche berücksichtigt werden.

Außerdem wird zwischen einem internen Bereich innerhalb eines Anwendungssystems und einem externen Bereich über mehrere Anwendungssysteme hinweg unterschieden. Auch in diesem Fall gilt, dass diese Ausprägungen nicht kompetitiv, sondern komplementär zueinander sind und folglich beide für eine bestmögliche Wiederverwendung einbezogen werden.

Aspekt Vorgehen

Wie man bereits aus den vorangegangenen Erläuterungen zum Vorgehen der Wiederverwendung erkennen kann, ist die geplante Wiederverwendung im Vergleich zur ungeplanten Wiederverwendung vorteilhafter. Die ungeplante Wiederverwendung wirft offensichtlich eine Reihe von Problemen auf, die z. B. in [Endr88, 87 f] skizziert sind. Deswegen wird als Ausprägung für eine bestmögliche Wiederverwendung die geplante, systematische Wiederverwendung gewählt.

Aspekt Technik

Bei der Technik der Wiederverwendung kann grundsätzlich ein kompositionelles oder ein generatives Verfahren zum Einsatz kommen. Aus den obigen Erläuterungen zu beiden Verfahren lässt sich erkennen, dass jedes Verfahren spezifische Vorteile bei der Wiederverwendung aufweist. Folglich ist eine Kombination aus beiden Verfahren für eine Wiederverwendung am besten geeignet.

Aspekt Nutzungsart

Bezüglich der Nutzungsart hat die Black-Box-Wiederverwendung gegenüber der White-Box-Wiederverwendung klare Vorteile. Wie auch schon im Rahmen der voran-

gegangenen Erläuterungen festgestellt wurde, führt die Black-Box-Wiederverwendung zu einer loseren Kopplung und weniger Abhängigkeiten zwischen den Entwicklungserzeugnissen [Grif98, 19]. Die beiden angesprochenen Alternativen, nämlich die Grey-Box- und die Glass-Box-Wiederverwendung, stellen Kompromisse dar, die zu einer Aufweichung des reinen Black-Box-Ansatzes führen und folglich das Wiederverwendungspotenzial eines Entwicklungserzeugnisses von vornherein schmälern. In der vorliegenden Arbeit soll jedoch ein Ansatz ausgearbeitet werden, mit dem eine kompromisslose Umsetzung der Wiederverwendung und Verteilung von Entwicklungserzeugnissen möglich sein soll. Deswegen wird im Rahmen dieser Arbeit die Black-Box-Wiederverwendung als Nutzungsart für eine bestmögliche Wiederverwendung bestimmt.

Tabelle 3.2 fasst die anzustrebenden Aspektausprägungen für eine bestmögliche Wiederverwendung im softwaretechnischen Entwurf und in der Implementierung in übersichtlicher Form zusammen.

Aspekt	**Wiederverwendungsunterstützende Ausprägungen**
Substanz	Konzept, Artefakt
Bereich	vertikal, horizontal, intern, extern
Vorgehen	geplant, systematisch
Technik	kompositionell, generativ
Nutzungsart	Black Box

Tabelle 3.2: Wiederverwendungsunterstützende Aspektausprägungen

3.1.4 Qualitätsmodelle als Ordnungsrahmen für das Formalziel Wiederverwendbarkeit

Nachdem in Abschnitt 3.1.3 das Formalziel Wiederverwendung, das sich auf das Vorgehen bei der Entwicklung von Anwendungssystemen bezieht, näher untersucht wurde, befasst sich dieser Abschnitt mit dem Formalziel Wiederverwendbarkeit, das das Ergebnis der Entwicklung von Anwendungssystemen betrifft.

Neben der Wiederverwendbarkeit existieren eine Reihe von weiteren Qualitätsmerkmalen softwaretechnischer Entwicklungserzeugnisse. Sie sind in so genannten **Qualitätsmodellen** systematisiert. Eines der ersten Qualitätsmodelle wurde 1976 von BOEHM ET AL. [BoBL76] aufgestellt und von MCCALL/MATSUMOTO [McMa80] weiterentwickelt [FrLS00, 18]. Diese beiden Qualitätsmodelle sind aufgrund ihrer allgemeinen Formulierung inzwischen am weitesten verbreitet und fanden auch Eingang in den ISO-Standard 9126 für die Evaluation von Software-Produkten [ISO01]. Einen

knappen Vergleich zwischen den genannten Qualitätsmodellen und dem ISO-Standard 9126 enthält [FrLS00, 19]. Allen drei ist eine dreistufige Beschreibungsform der Qualitätsmerkmale gemein (vgl. Abbildung 3.2) [FrLS00, 18].

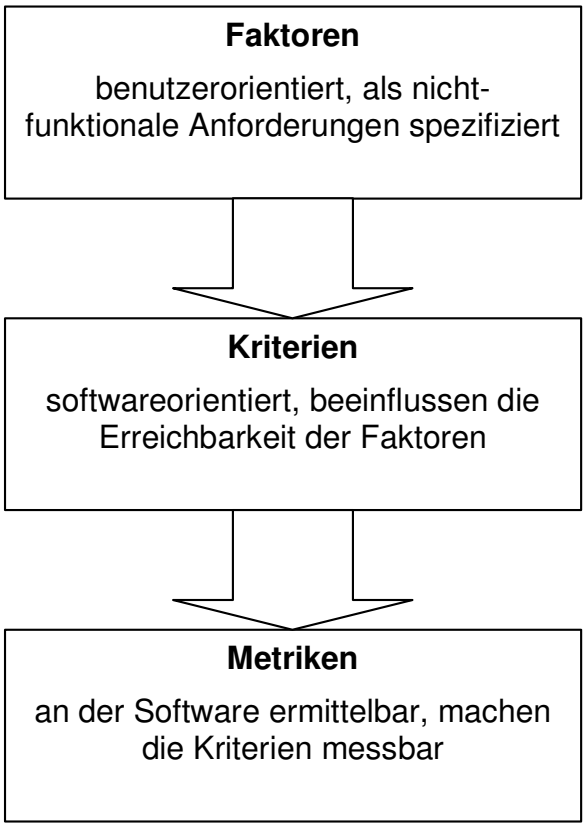


Abbildung 3.2: Dreistufiges Qualitätsmodell (in Anlehnung an [FrLS00, 18])

Auf der obersten Stufe werden die Qualitätsmerkmale als nicht-funktionale Anforderungen aus Benutzersicht beschrieben. Zu diesen so genannten **Faktoren** gehören beispielsweise die Wiederverwendbarkeit, die Änderbarkeit und die Portabilität von Software-Systemen oder -Teilsystemen [Pres87, 454]. Den Faktoren werden wiederum so genannte **Kriterien** zugeordnet. Sie sind aus softwaretechnischer Sicht formuliert und beeinflussen die Erreichbarkeit des entsprechenden Faktors. Die unterste Stufe bilden die so genannten **Metriken**, die an der Software ermittelbar sind und die Kriterien messbar machen.

3.1.4.1 Änderbarkeit als unterstützender Faktor

Wiederverwendbare Entwicklungserzeugnisse können im Allgemeinen nicht unverändert in andere Umgebungen übernommen werden. Folglich spielt der Faktor **Änderbarkeit** eine wesentliche Rolle für die Wiederverwendbarkeit von Entwicklungserzeugnissen. Nach DIN ISO 9126 versteht man unter Änderbarkeit den „Aufwand, der zur Durchführung vorgegebener Änderungen notwendig ist“ [Balz98, 260]. Damit Entwicklungserzeugnisse in anderen Umgebungen wiederverwendbar sind, müssen gegebenenfalls Änderungen in Form von Anpassungen und Erweiterungen durchge-

führt werden. In Anlehnung an [FSH+98] soll unter **Anpassung** eine „Einschränkung der vorhandenen Gesamtmenge der potentiellen Struktur- und Verhaltenseigenschaften [eines Entwicklungserzeugnisses] auf eine Teilmenge“ [FSH+98, 151 f] verstanden werden. Die Anpassung, oder auch Adaptierung, wird zum Teil selbst schon als eine Form der Wiederverwendung identifiziert [Endr88, 88 f]. Da heutige Entwicklungserzeugnisse jedoch in der Regel nicht allein durch Anpassung wiederverwendbar sind, kann diese Form nur als ein Teilfaktor zur Unterstützung der Wiederverwendbarkeit gesehen werden. Im Gegensatz zum Begriff „Anpassung“ steht der Begriff „**Erweiterung**“ für „die Eigenschaft, das Verhalten des [Entwicklungserzeugnisses] unter Beibehaltung der vorhandenen Verhaltenseigenschaft durch Bereitstellung neuer Verhaltenseigenschaften erweitern zu können“ [FSH+98, 153]. Diese beiden Änderungsformen werden häufig auch als eigene Faktoren unter den Bezeichnungen Anpassbarkeit und Erweiterbarkeit spezifiziert. In der vorliegenden Arbeit werden sie aber aus den genannten Gründen dem Faktor Änderbarkeit zugeordnet.

Die Änderbarkeit und die Wiederverwendbarkeit stehen also in einem bestimmten Zusammenhang: Damit ein Entwicklungserzeugnis wiederverwendbar sein kann, muss es auch änderbar sein. Andererseits können Entwicklungserzeugnisse, die nicht für die Wiederverwendung konstruiert worden sind, auch änderbar sein.

3.1.4.2 Weitere wiederverwendbarkeitsunterstützende Faktoren

In der Implementierung muss ein Entwicklungserzeugnis weitere, nicht-funktionale Eigenschaften aufweisen, damit es wiederverwendet werden kann. Dazu gehören in Anlehnung an McCall [McMa80] die Faktoren Interoperabilität und Portierbarkeit.

Der Faktor **Interoperabilität** wird allgemein definiert als der Aufwand, der notwendig ist, um zwei Software-Systeme oder -Teilsysteme miteinander zu verbinden [Pres87, 435; Balz82, 13 f]. Die Erreichbarkeit dieses Faktors wird von softwaretechnischer Seite beeinflusst durch die Kriterien Datengemeinsamkeit, Kommunikationsgemeinsamkeit und Modularität [Pres87, 454]. Mit der **Datengemeinsamkeit** wird eine durchgängige Verwendung standardisierter Datenstrukturen und -typen innerhalb eines Software-Systems oder -Teilsystems gefordert [Pres87, 435]. Analog dazu fordert die **Kommunikationsgemeinsamkeit** eine Nutzung standardisierter Schnittstellen, Protokolle und Kommunikationswege zur Kommunikation zwischen verschiedenen Software-Systemen oder -Teilsystemen [Pres87, 454]. Das Kriterium Modularität beeinflusst nicht nur die Erreichbarkeit des Faktors Interoperabilität, sondern auch die Erreichbarkeit des Faktors Wiederverwendbarkeit und wird daher im folgenden Abschnitt erläutert.

Unter dem Faktor **Portierbarkeit** versteht man den notwendigen Aufwand zur Überführung eines Software-Systems oder -Teilsystems von einer Hardware- und bzw.

oder Software-Umgebung in eine andere [Pres87, 434; Balz82, 12; FrLS00, 19; Balz98, 260]. Die Erreichbarkeit der Portierbarkeit wird wiederum beeinflusst von den Kriterien Basismaschinenunabhängigkeit, Modularität und Nachvollziehbarkeit.

Das Kriterium **Basismaschinenunabhängigkeit** bedingt die Strukturierung eines Entwicklungserzeugnisses nach dem Modell der Nutzer- und Basismaschine (siehe Abschnitt 2.2.5.1). Durch eine solche mehrschichtige Struktur kann ein Entwicklungserzeugnis unter anderem unabhängig von bestimmten Programmiersprachen, Betriebssystemen oder Hardware-Plattformen verwendet werden [FeSi01, 289]. Die Kriterien Modularität und Nachvollziehbarkeit beeinflussen auch die Erfüllbarkeit des Faktors Wiederverwendbarkeit und werden folglich im nächsten Abschnitt beschrieben.

Die **Portierung** selbst als Tätigkeit wird zum Teil bereits als eine Form der Wiederverwendung identifiziert [Endr88, 88 f]. Jedoch gilt auch in diesem Fall, dass sie allein keine Wiederverwendung im geforderten Sinne darstellt, sondern nur dazu beitragen kann.

Neben den Faktoren Interoperabilität und Portierbarkeit, die direkten Einfluss auf die Wiederverwendbarkeit eines Entwicklungserzeugnisses in der Implementierung haben, existieren weitere Faktoren, die zusätzlich von einem wiederverwendbaren Entwicklungserzeugnisses in der Implementierung verlangt werden können. Dazu gehören z. B. die Wartbarkeit, die Zuverlässigkeit und die Effizienz [Pres87, 433 ff; Balz82, 10 ff; Balz98, 257 ff; BMR+98, 398 ff]. Soweit die Kriterien, die die Erfüllbarkeit dieser Faktoren beeinflussen, messbar und damit quantifizierbar sind, können dafür konkrete Ausprägungen angegeben werden. Dabei wird unterschieden zwischen der Spezifikation geforderter Kriterien und Kriterienausprägungen und der Dokumentation realisierter Kriterien und Kriterienausprägungen. Je besser die realisierten und dokumentierten Kriterien und Kriterienausprägungen eines wiederzuverwendenden Entwicklungserzeugnisses in der Implementierung zu den geforderten Kriterien und Kriterienausprägungen passen, desto eher wird das Entwicklungserzeugnis wiederverwendet.

3.1.4.3 Wiederverwendbarkeitsunterstützende Kriterien

In der Literatur zur Software-Wiederverwendung werden eine Reihe von Kriterien angegeben, die generellen Einfluss auf den Qualitätsfaktor Wiederverwendbarkeit haben [FSH+98, 147ff; Pres87, 452 ff; Küff94, 42 ff; Same97, 173 ff]. Folglich dienen sie auch dem Erreichen der in Abschnitt 3.1.3 beschriebenen Aspekte und Aspektausprägungen, die eine bestmögliche Wiederverwendung gewährleisten. Beispielsweise dienen die im Folgenden genannten Kriterien Abstraktion, Generalität und hierarchische Strukturierung dem Wiederverwendungsaspekt Bereich mit den geforder-

ten Ausprägungen vertikal, horizontal, intern und extern. Eine ausführliche Gegenüberstellung der Kriterien und der wiederverwendbarkeitsunterstützenden Faktoren mit den Wiederverwendungsaspekten und den zugehörigen Ausprägungen enthält der Abschnitt 3.3. Die folgende Zusammenstellung konzentriert sich auf die Erläuterung der Kriterien die unterstützenden Einfluss auf den Faktor Wiederverwendbarkeit haben.

Damit ein Entwicklungserzeugnis bestmöglich wiederverwendet werden kann, sollte es [FSH+98, 147 ff; Pres87, 452 ff; Küff94, 43; Same97, 173 ff]

- ein hohes Abstraktionsniveau aufweisen (**Kriterium der Abstraktion**) [Küff94, 43; Same97, 24; Meye96, 32].

  Dies bezieht sich auf die Realisierungsnähe eines Entwicklungserzeugnisses [FSH+98, 148; Same97, 24; Endr88, 87]. Je realisierungsferner ein Entwicklungserzeugnis ist, desto besser kann es wiederverwendet werden [Endr88, 87].

- möglichst allgemein gehalten sein (**Kriterium der Generalität**).

  Danach sollte ein Entwicklungserzeugnis vorwiegend domänenunabhängige Eigenschaften aufweisen und von domänenspezifischen Eigenschaften abstrahieren [Küff94, 43]. „Die Anwendungsbreite ist umgekehrt proportional zum Umfang des eingebetteten Domänenwissens“ [FSH+98, 148]. Allerdings versprechen domänenspezifischere Entwicklungserzeugnisse eine höhere Produktivitätssteigerung bei der Wiederverwendung, da der Anpassungsaufwand an eine bestimmte Domäne entfällt [BiRi89, 4]. Deswegen muss ein Entwicklungserzeugnis auch Möglichkeiten zur Spezialisierung, z. B. durch hierarchische Strukturierung, aufweisen können.

- hierarchisch strukturiert werden können (**Kriterium der hierarchischen Strukturierung**).

  Mithilfe einer hierarchischen Strukturierung kann die Komplexität eines Entwicklungserzeugnisses bewältigt werden (siehe Abschnitt 3.4.1.1 und [FSH+98, 149]). Dies unterstützt unter anderem die im Folgenden geforderte Verständlichkeit und Nachvollziehbarkeit von Entwicklungserzeugnissen. Außerdem kann eine hierarchische Strukturierung für die noch zu fordernde Anpassung und Erweiterung eines wiederzuverwendenden Entwicklungserzeugnisses genutzt werden [FSH+98, 149; Küff94, 44].

- modular sein (**Kriterium der Modularität**).

  Entsprechend den noch folgenden Ausführungen in Abschnitt 3.4.2 ist ein Entwicklungserzeugnis modular, wenn es zumindest nach dem Geheimnisprinzip strukturiert ist, einen bestimmten Belang abbildet und nach innen eine maximale

funktionale Kohärenz und nach außen eine minimale Bindung besitzt (siehe Abschnitt 3.4.2 und [PaCW89, 142 ff]).

- intuitiv verständlich sein (**Kriterium der intuitiven Verständlichkeit**).

  Die fachlichen Inhalte eines Entwicklungserzeugnisses sollten für ein erstes Verständnis kurz und klar beschrieben sein [FSH+98, 148]. Dies erleichtert das Auffinden eines wiederverwendbaren Entwicklungserzeugnisses [Same97, 187].

- einfach nachvollziehbar sein (**Kriterium der Nachvollziehbarkeit**).

  Dies bezieht sich auf ein tieferes Verständnis eines Entwicklungserzeugnisses beim Einsatz in einer anderen Umgebung. Hierbei sollte der Aufwand für den Einsatz eines Entwicklungserzeugnisses den Aufwand für eine Neuimplementierung nicht übersteigen [FSH+98, 148]. Dafür ist eine ausreichende Selbstdokumentation notwendig mit der seine Objekte, Annahmen, Einschränkungen, Eingaben, Ausgaben, Teile und sein Zustand bestimmt und verifiziert werden kann [Balz82, 13; Pres87, 435].

- standardisiert und formalisiert sein (**Kriterium der Standardisierung und Formalisierung**).

  Beide Merkmale dienen der Verständlichkeit und Nachvollziehbarkeit eines Entwicklungserzeugnisses. Die Formalisierung ist für die Prüfung auf Konsistenz und Vollständigkeit des Entwicklungserzeugnisses notwendig. Je höher der Formalisierungsgrad eines Entwicklungserzeugnisses, desto einfacher ist seine Wiederverwendung, aber desto aufwändiger ist seine Erstellung [FSH+98, 149].

3.2 Formalziel Verteilbarkeit

Neben den Formalzielen Wiederverwendung und Wiederverwendbarkeit rückt in der letzten Zeit ein weiterer Aspekt in den Vordergrund, der auch Auswirkungen auf die zu entwickelnden Anwendungssysteme hat: Die Verteilbarkeit von Anwendungssystemen über verschiedene, gegebenenfalls weltweit vernetzte Rechnersysteme. Der folgende Abschnitt klärt zunächst die Bedeutung des Begriffs „Verteilbarkeit“ in Bezug auf die vorliegende Arbeit.

3.2.1 Begriffliche Grundlagen

Der Begriff „**Verteiltes System**“ stammt ursprünglich aus dem Bereich der Architekturen von Rechnersystemen. Deshalb beziehen sich die meisten Definitionen verteilter Anwendungssysteme auf bestimmte technische Eigenschaften der Verteilung. So definieren z. B. MÜHLHÄUSER und TANENBAUM/VAN STEEN ein verteiltes (Datenverarbeitungs-) System insbesondere anhand der verteilten Hardware-Komponenten, die untereinander in Beziehung stehen [Mühl02, 673 f; TaSt02, 2]. Dem gegenüber be-

tont HAASE in seiner Definition eines verteilten Anwendungssystems die Prozesse, die auf verteilten Hardware-Komponenten ablaufen [Haas01, 1].

Der in dieser Arbeit zugrunde gelegte Begriff eines verteilten Systems wird zunächst unabhängig von einem bestimmten Anwendungsgebiet formuliert. Dafür müssen die wesentlichen Merkmale der aus dem Bereich der Architekturen von Rechnersystemen stammenden Definitionen herausgearbeitet und entsprechend verallgemeinert werden.

Demnach ist ein verteiltes System insbesondere durch folgende Struktur- und Verhaltensmerkmale charakterisiert [Ensl78; FeSi94, 3 f; Sinz02a, 1065]:

- Aus der Außensicht stellt es ein **integriertes System** dar, das gemeinsame Ziele verfolgt.
- Aus der Innensicht besteht es aus mehreren **autonomen Komponenten**, die zur Erreichung der gemeinsamen Ziele miteinander kooperieren. Dabei besitzt keine Komponente die globale Kontrolle über das Gesamtsystem.
- Die Verteilung des Systems auf mehrere Komponenten ist für Nutzer des Systems nicht erkennbar. Dieses Merkmal wird auch **Verteilungstransparenz** genannt.

Diese allgemeine Definition ist gleichermaßen auf Informationssysteme, Anwendungssysteme und die zugehörigen Rechner- und Kommunikationssystemen anwendbar [FeSi94, 5]. Somit kann das Konzept des verteilten Systems als übergreifende Metapher auf allen Ebenen einer Informationssystem-Architektur genutzt werden. Dies trägt wesentlich zur methodischen Durchgängigkeit bei der Entwicklung von Anwendungssystemen bei.

Ein Beispiel für die Verwendung der allgemeinen Definition eines verteilten Systems als übergreifende Metapher stellt die in Abschnitt 2.4 erläuterte SOM-Methodik dar. Wie dort bereits gezeigt wurde, enthält die Unternehmensarchitektur der SOM-Methodik das Konzept des verteilten Systems auf allen Ebenen (vgl. Abbildung 3.3) [FeSi94, 5; Sinz02a, 1066]:

- Auf der Aufgabenebene wird ein verteiltes betriebliches System als eine Menge von kooperierenden Geschäftsprozessen spezifiziert. Jeder Geschäftsprozess ist wiederum in Haupt- und Serviceprozesse zerlegbar und somit auch verteilt.
- Auf der Aufgabenträgerebene werden verteilte Anwendungssysteme als maschinelle Aufgabenträger für die automatisierten Aufgaben der verteilten Geschäftsprozesse spezifiziert. Diese Spezifikation lässt sich weiter unterteilen in eine fachliche und eine softwaretechnische Spezifikation.

- Ergänzend zu diesen beiden Ebenen werden auf einer dritten Ebene verteilte Rechner- und Kommunikationssysteme als anwendungsneutrale Basissysteme für die verteilten Anwendungssysteme spezifiziert. Die Realisierung der Basissysteme erfolgt durch virtuelle Maschinen in Form von Systemsoftware, wie z. B. Betriebssysteme und Datenbankmanagementsysteme und Middleware-Systeme, wie z. B. objekt- und komponentenbasierte Entwicklungs- und Ausführungsplattformen. Diese Systeme werden in Abschnitt 5.3.6.2 noch näher untersucht.

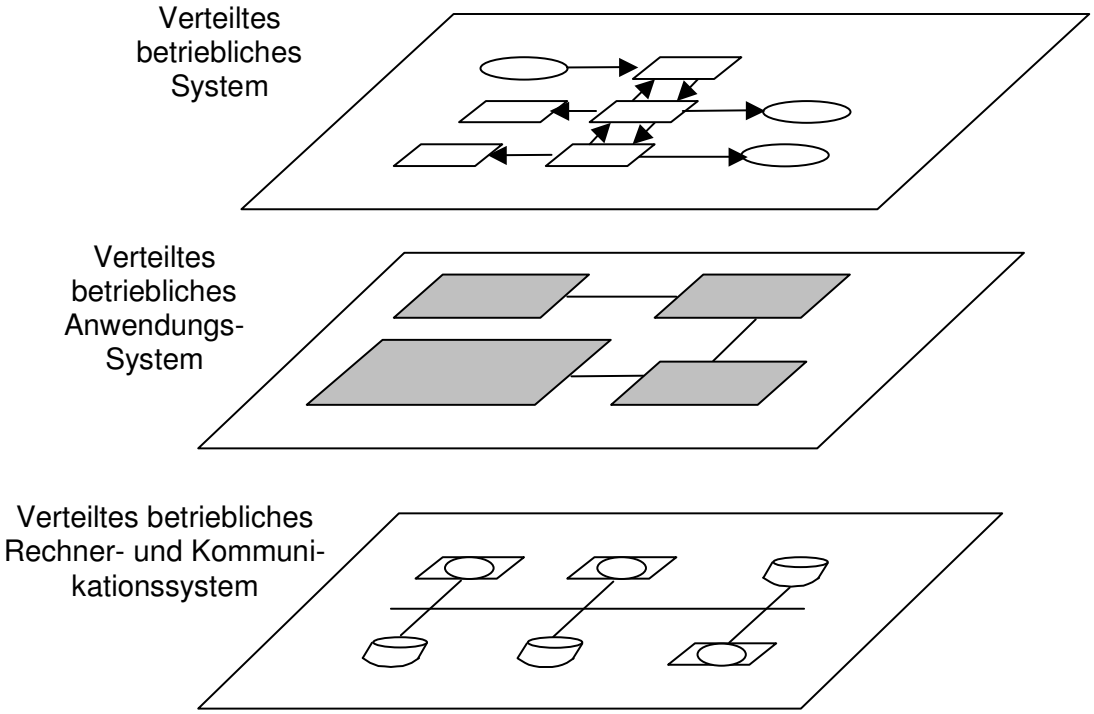


Abbildung 3.3: Ebenen eines verteilten Systems (in Anlehnung an FeSi94, 4])

3.2.2 Bedeutung der Verteilung in der Software-Entwicklung

Der Einsatz von verteilten Anwendungssystemen ist für Unternehmen inzwischen nicht nur optional, sondern für das Bestehen auf dem jeweiligen Markt obligatorisch geworden. Die Begründung dafür liegt in den Veränderungen der wirtschaftlichen und gesellschaftlichen Rahmenbedingungen, die durch die Entwicklung im Bereich der Informations- und Kommunikationstechnologie sowohl ermöglicht als auch forciert werden [Zimm99a, 2].

Zunehmende Leistungssteigerungen und Miniaturisierungen der Kommunikations- und Informationsverarbeitungssysteme führen zum weltweiten Wettbewerb zwischen Unternehmen verschiedener Größen, da die Vermarktung von Produkten und Dienstleistungen global und rund um die Uhr erfolgen kann. Dies hat zur Folge, dass der Preis und die Leistung eines Produkts oder einer Dienstleistung weltweit verglichen werden kann. Deshalb muss ein Unternehmen, das sich im weltweiten Wettbewerb

befindet, versuchen, die Produktionskosten seiner angebotenen Produkte oder Dienstleistungen zu reduzieren, um weltweit konkurrenzfähig zu bleiben.

Neben den veränderten wirtschaftlichen Rahmenbedingungen durch den weltweiten Wettbewerb sind auch Veränderungen in der Gesellschaft feststellbar, die ebenfalls durch die informations- und kommunikationstechnologischen Entwicklungen zunächst ermöglicht wurden und inzwischen forciert werden. Hierbei ist insbesondere die Einstellung zur Arbeit zu nennen, die mittlerweile mehr von postmateriellen Werten als von materiellen Werten geprägt ist [Zimm99a, 1]. Dazu gehören die Bedürfnisse nach Eigenverantwortung, Selbständigkeit, Selbstverwirklichung, Partizipation und Kommunikation [Zimm99a, 1].

Die genannten veränderten wirtschaftlichen und gesellschaftlichen Rahmenbedingungen erfordern den Einsatz von verteilten Anwendungssystemen. Die Fortschritte in der Informations- und Kommunikationstechnologie sorgen außerdem dafür, dass diese Systeme ihre Dienste zuverlässig, sicher, schnell und kostengünstig anbieten können.

3.2.3 Integration als Voraussetzung für das Formalziel Verteilbarkeit

Ein wesentliches Merkmal eines verteilten Systems ist, wie oben beschrieben, die Integration seiner Komponenten zur Verfolgung gemeinsamer Ziele. Dies trifft folglich auch auf verteilte Anwendungssysteme zu. Zur Erläuterung der Integration von verteilten Anwendungssystemen müssen zunächst die im Rahmen der Aufgabenanalyse durchzuführende Zerlegung der Gesamtaufgabe eines Informationssystems in Teilaufgaben und die anschließende Automatisierung der entstandenen Teilaufgaben näher betrachtet werden.

Bei der Zerlegung einer Gesamtaufgabe entstehen die Aufgabenobjekte und die Aufgabenziele der Teilaufgaben aus der Zerlegung des Aufgabenobjekts und der Aufgabenziele der Gesamtaufgabe. Dabei wird ein **Kommunikationskanal** zwischen den entstehenden Teilaufgaben definiert [FeSi01, 215]. Die Zerlegungen der Aufgabenobjekte und der Aufgabenziele müssen nicht disjunkt erfolgen. Dadurch kann es vorkommen, dass Aufgaben mit überlappenden Aufgabenobjekten bei der Automatisierung unterschiedlichen Anwendungssystemen zugeordnet werden. Dies führt zu einer redundanten Speicherung der gemeinsamen Teile der beiden Aufgabenobjekte in den beteiligten Anwendungssystemen [FeSi01, 215]. Analog dazu können auch überlappende Aufgabenziele unterschiedlichen Anwendungssystemen zugeordnet werden. Dies resultiert in einer redundanten Realisierung der gemeinsamen Aktionen der Lösungsverfahren in den entsprechenden Anwendungssystemen [FeSi01, 215]. Außerdem müssen die Anschlussstellen des Kommunikationskanals zwischen den Teilaufgaben zwangsläufig in beiden Anwendungssystemen redundant vorhanden

sein. Dies wird als **Schnittstellen-Redundanz** bezeichnet [FeSi01, 215]. Für die Erfüllung der Gesamtzielsetzung des Informationssystems ist es ferner notwendig, dass die den Anwendungssystemen zugeordneten Teilaufgaben zielgerichtet miteinander kooperieren [Raue96, 43]. Dazu bedarf es entsprechender Leistungs- und Kommunikationsbeziehungen zwischen den Teilaufgaben und einer geeigneten Koordination der Teilaufgaben [MKR+00, 3 ff].

Folglich ist die Automatisierung derart durchzuführen, dass die entstehenden Redundanzen kontrolliert und die zugeordneten Teilaufgaben zielgerichtet koordiniert werden können. Bei Betrachtung der Automatisierung als Meta-Gestaltungsaufgabe bilden diese Bedingungen Formalziele, die den Lösungsraum, der durch das Sachziel der Voll- bzw. Teilautomatisierung der Aufgaben eines Informationssystems aufgespannt wird, einschränkt [Fers92, 3 ff; FeSi01, 220]. Diese Formalziele werden zusammengefasst zum Formalziel Integration [Fers92, 4]. Allgemein bezeichnet man unter **Integration** „das Bemühen, bisher getrennte Teilsysteme zusammenzuführen“ [Raue96, 43]. Dabei kann einerseits top down vorgegangen werden, indem ein System zielgerichtet in Teilsysteme zerlegt wird und Beziehungen zwischen den Teilsystemen definiert werden. Andererseits lässt sich eine Integration auch bottom up durchführen, wobei Beziehungen zwischen isolierten Teilsystemen definiert werden.

3.2.3.1 Integrationsgrad und Integrationsziele

Analog zum Automatisierungsgrad, der die Sachzielerfüllung der Automatisierungsaufgabe beschreibt, kann ein **Integrationsgrad** angegeben werden, der die Erfüllung des Formalziels Integration der Automatisierungsaufgabe wiedergibt. Der Integrationsgrad wird anhand eines Ausprägungstupels von ausgewählten Struktur- und Verhaltensmerkmalen eines integrierten Anwendungssystems beschrieben [Fers92, 12; FeSi01, 217]. Zur Bestimmung eines optimalen Integrationsgrades muss jedem Merkmal eine beabsichtigte Zielausprägung zugeordnet werden [FeSi01, 217]. Diese Zielausprägungen nennt man auch **Integrationsziele** bezüglich der Struktur- und Verhaltensmerkmale. Die ausgewählten Struktur- und Verhaltensmerkmale sind geeignet, gegebene Integrationskonzepte zu vergleichen und zu bewerten [Fers92, 12].

Zunächst werden die strukturorientierten Merkmale Redundanz und Verknüpfung eines integrierten Anwendungssystems beschrieben. Sie beziehen sich auf die Systemkomponenten eines Anwendungssystems und auf die Kommunikationskanäle zwischen ihnen [FeSi01, 217]. Systemkomponenten sind in diesem Zusammenhang beispielsweise Datenstrukturen, Funktionen oder Objekte.

Redundanz

Die **Redundanz** gibt an, inwieweit die einer bestimmten Anwendungssystemfunktion zugeordneten Systemkomponenten mehrfach vorkommen [FeSi01, 218]. Sie ist im

Zusammenhang mit Anwendungssystemen nach den Komponententypen zu differenzieren. Folglich versteht man unter **Datenredundanz** redundante Datenobjekttypen bzw. Datenattribute. Diese sind normalerweise auf die angesprochene Überlappung von Aufgabenobjekten zurückzuführen. Demgegenüber bezeichnet **Funktionsredundanz** redundante Aktionen überlappender Lösungsverfahren und resultiert meistens aus der erwähnten nicht-disjunkten Zerlegung von Aufgabenzielen [FeSi01, 219; Fers92, 13].

Eine Entfernung redundanter Systemkomponenten beeinträchtigt im Normalfall die Funktionsfähigkeit der betreffenden Anwendungssystemfunktion und damit des Anwendungssystems nicht [FeSi01, 218]. Redundante Systemkomponenten sind vielmehr Quellen für mögliche Inkonsistenzen, die aufgedeckt und gegebenenfalls korrigiert werden müssen. Außerdem deuten sie auf eine mangelnde Wirtschaftlichkeit der Ressourcennutzung hin [FeSi01, 218]. Diese Gründe sprechen auf der einen Seite für eine Vermeidung von Redundanz. Auf der anderen Seite kann auch eine Erhaltung von Redundanz erwünscht sein. Beispielsweise ist ein Anwendungssystem mit redundanten Systemkomponenten tolerant gegenüber Komponentenausfällen. Außerdem sind Leistungssteigerungen des Anwendungssystems durch parallele Nutzung redundanter Komponenten möglich. Darüber hinaus lässt sich die Strukturkomplexität des Anwendungssystems durch die Verwendung von Systemkomponenten mit redundanten Merkmalen reduzieren [FeSi01, 218]. Deswegen ist die Bestimmung des beabsichtigten Integrationsziels bezüglich der Redundanz nur durch eine fallweise Bewertung der genannten Argumente möglich. Auf jeden Fall muss die zu erzielende Redundanz kontrollierbar sein [Fers92, 13].

Verknüpfung

Das Merkmal **Verknüpfung** beschreibt die Kommunikationsstruktur eines integrierten Anwendungssystems, d. h. die Art und die Anzahl der Kommunikationskanäle zwischen den Systemkomponenten [FeSi01, 219]. Die für die Gesamtaufgabe eines Informationssystems erforderliche Anzahl von Kommunikationskanälen resultiert aus der Abgrenzung der Systemkomponenten und deren Grad an Redundanz [FeSi01, 219]. Das Integrationsziel bezüglich des Merkmals Verknüpfung ist die Kontrolle der Kommunikationsstruktur [Fers92, 13]. Dafür muss ein eigenständiges Kommunikationssystem zur Verfügung stehen, das die Kommunikationsstruktur mit einem minimalen Verknüpfungsgrad zwischen den Systemkomponenten realisiert. Es ist außerdem dafür verantwortlich, dass zum einen die Nachrichten zwischen den beteiligten Systemkomponenten ausreichend schnell und sicher übertragen werden und zum anderen die Kommunikationsstruktur robust bezüglich Störungen und Erweiterungen gestaltet ist [MKR+00, 5; FeSi01, 219].

Die nächsten beiden Merkmale beziehen sich auf das Verhalten eines integrierten Anwendungssystems. Für die Erfassung der ganzheitlichen und zielbezogenen Ausrichtung der Teilaufgaben, die aus der Zerlegung der Gesamtaufgabe des Informationssystems entstanden sind, stehen die Integrationsmerkmale Konsistenz und Zielorientierung zur Verfügung [FeSi01, 217].

Konsistenz

Das Merkmal **Konsistenz** bezieht sich im Allgemeinen auf zulässige Systemzustände hinsichtlich des Verhaltens. Ein zum Verhalten gehörender Zustand wird als konsistenter Systemzustand, ein nicht zum Verhalten eines Systems gehörender Zustand als inkonsistenter Systemzustand bezeichnet [FeSi01, 219]. Im Zusammenhang mit Anwendungssystemen werden zwei Formen der Konsistenz anhand von entsprechenden Integritätsbedingungen formuliert. Die **semantischen Integritätsbedingungen** sorgen für konsistente Systemzustände aus Modellierungssicht, die **operationalen Integritätsbedingungen** sorgen in einem System, das parallele Zustandsübergänge zulässt, für konsistente Systemzustände vor und nach Zustandsübergängen [FeSi01, 219]. Die permanente Einhaltung der semantischen und operationalen Integrität stellt somit das Integrationsziel bezüglich des Merkmals der Konsistenz dar [FeSi01, 220].

Zielorientierung

Die zielbezogene Koordination der Teilaufgaben auf jeder Zerlegungsebene wird mit dem Merkmal **Zielorientierung** erfasst. Dafür ist eine als **Vorgangssteuerung** bezeichnete Einrichtung notwendig. Deren Funktionsweise entspricht im Wesentlichen der einer Aktionensteuerung, die im Rahmen des Aufgabenkonzepts in Abschnitt 2.1.3 bereits erläutert wurde. Allerdings werden mithilfe einer Vorgangssteuerung keine Aktionen eines Vorgangs koordiniert, sondern ganze Vorgänge, also Aufgabendurchführungen. Die Lenkung der Vorgänge eines Anwendungssystems anhand einer solchen eigenständigen Vorgangssteuerung ist folglich das Integrationsziel bezüglich des Merkmals Zielorientierung. Zur Bewältigung dieser Aufgabe muss die Vorgangssteuerung eine Zielausrichtung aller Komponenten und Teilsysteme des Anwendungssystems vornehmen [FeSi01, 220; Fers92, 13].

Aufgabenträgerbezug

Wie in Abschnitt 2.2.5.1 bereits erläutert wurde, sind Anwendungssysteme schichtweise aufgebaut. Demnach basieren Anwendungssysteme als maschinelle Aufgabenträger selbst wieder auf niedrigeren Aufgabenträgern, den Basismaschinen. Damit jedoch die bisher genannten Integrationsziele unabhängig von diesen niedrigeren Aufgabenträgern verfolgt werden können, wird ein zusätzliches, als **Aufgabenträgerbezug** bezeichnetes Integrationsmerkmal eingeführt [Fers92, 12 ff]. Das dies-

bezügliche Integrationsziel ist die Verringerung der Abhängigkeit von Eigenschaften niedrigerer Aufgabenträger bei der Aufgabendefinition [FeSi01, 218]. Dies kann grundsätzlich durch den Einsatz **abstrakter Maschinen** erreicht werden, die ein Anwendungssystem vor Technologieentwicklungen und spezifischen Leistungsparametern der niedrigeren Aufgabenträger abschirmen [FeSi01, 220; Fers92, 14]. So kann beispielsweise eine Portierung eines Anwendungssystems über mehrere Rechnergenerationen oder zwischen Rechnern unterschiedlicher Leistung und unterschiedlicher Hersteller ermöglicht werden.

Tabelle 3.3 fasst die genannten Integrationsmerkmale und zugehörigen Integrationsziele in übersichtlichere Form zusammen.

Merkmalsgruppe		**Ziel: Einhaltung einer vorgegebenen Ausprägung des Merkmals ...**
Struktur	Redundanz	Datenredundanz Funktionsredundanz
	Verknüpfung	Kommunikationsstruktur
Verhalten	Konsistenz	semantische Integrität operationale Integriät
	Zielorientierung	Vorgangssteuerung
Aufgabenträgerbezug		Unabhängigkeit vom Aufgabenträger

Tabelle 3.3: Integrationsmerkmale und -ziele (in Anlehung an [FeSi01, 218])

3.2.3.2 Integrationskonzepte

Zur Erreichung eines vorgegebenen Integrationsgrads als Formalziel der Meta-Gestaltungsaufgabe Automatisierung sind eine Reihe von Lösungskonzepten, so genannte **Integrationskonzepte** bekannt [FeSi01, 220]. Die am häufigsten verwendeten Integrationskonzepte sind die Funktionsintegration, die Datenintegration und die Objektintegration [FeSi01, 220; Fers92, 14]. Wie man bereits an den Bezeichnungen der Konzepte erkennen kann, orientieren sie sich zur Erfüllung des vorgegebenen Integrationsgrads an unterschiedlichen Integrationsbereichen. Bei der Funktionsintegration steht die Integration der Aufgabenziele bzw. Lösungsverfahren im Vordergrund, bei der Datenintegration die der Aufgabenobjekte und bei der Objektintegration werden beide Integrationsbereiche einbezogen. Deswegen erreicht jedes

Konzept bezüglich der Integrationsziele auch einen anderen Zielerreichungsgrad [FeSi01, 220 f].

Datenflussorientierte Funktionsintegration

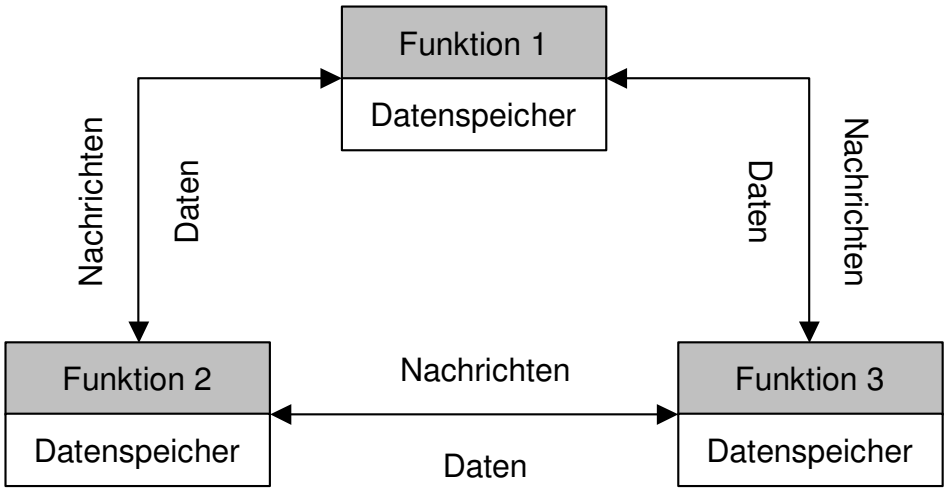


Abbildung 3.4: Datenflussorientierte Funktionsintegration [FeSi01, 222]

Die **datenflussorientierte Funktionsintegration** ist das historisch älteste Integrationskonzept. Es betrachtet ein Anwendungssystem als ein Netzwerk aus Teilsystemen, die jeweils aus einem Datenspeicher einschließlich Ein- und Ausgabekanälen und aus einer Funktion bestehen. Der Datenspeicher und die Ein- und Ausgabekanäle repräsentieren das Aufgabenobjekt, die Funktion spezifiziert implizit die Aufgabenziele [FeSi01, 222 und 92]. Abbildung 3.4 zeigt ein Beispiel eines solchen Netzwerks. Die Teilsysteme tauschen über Kommunikationskanäle Daten oder Nachrichten aus [FeSi01, 222; Fers92, 15]. Dieses Integrationskonzept wurde hauptsächlich dafür eingesetzt, bestehende Automationsinseln zu verbinden und Kommunikationskanäle unter Vermeidung von Medienbrüchen zu automatisieren [FeSi01, 222]. Deswegen wird primär das Integrationsziel bezüglich der Kommunikationsstruktur verfolgt [FeSi01, 222]. Dem gegenüber ist aufgrund von Erfahrungen mit installierten Systemen aus der Praxis bekannt, dass Daten- und Funktionsredundanzen sowie Verletzungen der Integrität bei diesem Integrationskonzept häufig auftreten [Fers92, 15 f].

Datenintegration

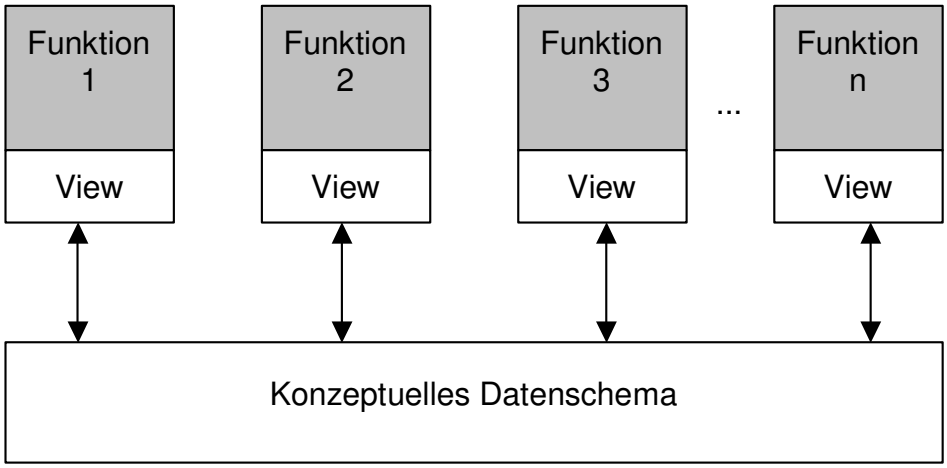


Abbildung 3.5: Datenintegration (in Anlehnung an [FeSi01, 224])

Das Integrationskonzept **Datenintegration** ist eng verbunden mit dem Einsatz von Datenbanksystemen und Mehrplatz-Dialogsystemen in den 1970er und 1980er Jahren. Es beruht auf einer Zusammenfassung von Datenbeständen und Datenkanälen in Form von Realisierungen eines konzeptuellen Datenschemas. Das **konzeptuelle Datenschema** definiert dabei einen gemeinsamen Zustandsraum, auf dem die Funktionen über so genannte **externe Sichten (Views)** operieren (vgl. Abbildung 3.5). Die externen Sichten stellen dabei Teilmengen des konzeptuellen Datenschemas dar [FeSi01, 224]. Somit dienen die **Datenobjekttypen** des konzeptuellen Datenschemas einerseits als Zustandsspeicher und andererseits als Datenkanal zwischen den Funktionen [Fers92, 17 f]. In Analogie zu Konzepten aus dem Bereich der Rechnerarchitekturen kann eine solche Funktionsverknüpfung auch als **enge Kopplung** bezeichnet werden. Im Unterschied dazu bezeichnet eine **lose Kopplung** eine Funktionsverknüpfung durch Nachrichtenaustausch über ein gemeinsames Kommunikationssystem [FeSi01, 225; Fers92, 19]. Mit der Datenintegration sollen insbesondere die Probleme der Datenredundanz und der Integritätsverletzungen der datenflussorientierten Funktionsintegration behoben werden. Die Kontrolle der Datenredundanz wird anhand von Methoden der Datenmodellierung gestaltet. Zur Wahrung der semantischen Integrität werden explizite semantische Intergritätsbedingungen definiert und deren Einhaltung überwacht. Für die Kontrolle der operationalen Integrität ist die Transaktionsverwaltung eines zugrunde liegenden Datenbankverwaltungssystems zuständig [MKR+00, 8]. Aufgrund der angesprochenen engen Kopplung können jedoch komplexe Kommunikationsbeziehungen zwischen den Funktionen auftreten. Diese sind der Kontrolle der Kommunikationsstruktur abträglich und führen häufig zu fehlerhaftem Systemverhalten [MKR+00, 8].

Objektintegration

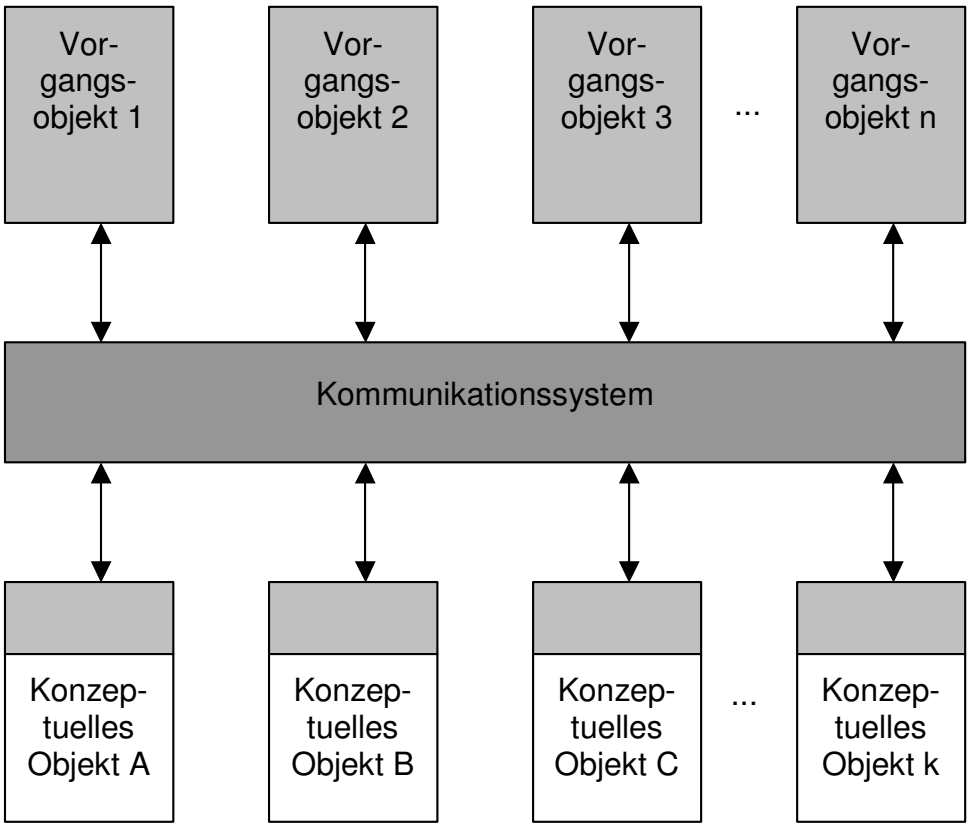


Abbildung 3.6: Objektintegration (in Anlehnung an [MKR+00, 9])

Die **Objektintegration** kann als eine Weiterentwicklung der datenflussorientierten Funktionsintegration angesehen werden. Der wesentliche Unterschied liegt darin, dass die Funktionen der Teilsysteme abhängig von der Art der beteiligten Aufgabenobjekte lose oder eng gekoppelt werden. Funktionen mit einem gemeinsamen Aufgabenobjekt werden über einen gemeinsamen Speicher eng gekoppelt, Funktionen mit unterschiedlichen Aufgabenobjekten werden über Kommunikationskanäle, die dem Nachrichtenaustausch zwischen den Funktionen dienen, lose gekoppelt [FeSi01, 93]. Dies entspricht im Allgemeinen einer Realisierung eines grundlegenden Merkmals des Grundkonzepts der Objektorientierung, nämlich der gezielten Interaktion von Objekten über Nachrichtenaustausch, wobei Objekte ihren Zustand und zugehörige Operatoren kapseln. In Abschnitt 3.5.1.1 wird das Grundkonzept der Objektorientierung ausführlich vorgestellt.

Zusätzlich unterscheidet das Konzept der Objektintegration zwischen konzeptuellen Objekten und Vorgangsobjekten (vgl. Abbildung 3.6). **Konzeptuelle Objekte** repräsentieren die Aufgabenobjekte der Teilaufgaben des Informationssystems. Ergänzend dazu führen **Vorgangsobjekte** Teilaufgaben durch, indem sie Aktionen auf ihrem jeweiligen Aufgabenobjekt, den konzeptuellen Objekten, ausführen lassen. Dies geschieht anhand von **Nachrichten**, die das Vorgangsobjekt an die konzeptuellen

Objekte übermittelt und dort entsprechende Operatorenausführungen auslösen. Im Vergleich zum Konzept der Datenintegration wird in einem objektintegrierten Anwendungssystem eine Anwendungsfunktion durch ein oder mehrere Vorgangsobjekte und den von den Vorgangsobjekten kontrollierten konzeptuellen Objekten realisiert [Fers92, 22].

Gleichartige konzeptuelle Objekte lassen sich anhand von **konzeptuellen Objekttypen** beschreiben. Analog dazu werden gleichartige Vorgangsobjekte anhand von **Vorgangsobjekttypen** repräsentiert. Vorgangsobjekttypen entsprechen somit Teilaufgaben, Beziehungen zwischen Vorgangsobjekttypen korrespondieren mit der Zerlegungsstruktur der Gesamtaufgabe [FeSi01, 227]. Dadurch unterstützt diese Form der Vorgangssteuerung eine Erreichung der Ziele der Gesamtaufgabe. Die konzeptuellen Objekttypen bilden ein **konzeptuelles Objektschema**, das in der Datensicht weitgehend mit einem konzeptuellen Datenschema korrespondiert [FeSi01, 228]. Folglich sind vergleichbare Zielerreichungsgrade bezüglich der Merkmale Datenredundanz und Integrität möglich [FeSi01, 228]. Die zu einem Vorgangsobjekttyp zugeordneten konzeptuellen Objekttypen werden zu einem Teilschema zusammengefasst. Die Teilschemata unterschiedlicher Vorgangsobjekttypen überlappen in der Regel, so dass ein konzeptuelles Objekt an unterschiedlichen Vorgängen beteiligt sein kann [Fers92, 22].

Die Kontrolle der Funktionsredundanz kann im Konzept der Objektintegration auf mehreren Wegen erfolgen. In [FeSi01, 228] werden dafür die Merkmale Generalisierung und Vererbung des Grundkonzepts der Objektorientierung verwendet. Eine andere Möglichkeit, redundante Aktionen in unterschiedlichen Lösungsverfahren zu kontrollieren, ist, diese in eigenen Vorgangsobjekttypen auszulagern und Verwendungsbeziehungen zu den Vorgangsobjekttypen, aus denen sie ausgelagert wurden, einzuführen.

Die Kommunikation aller Objekte erfolgt über ein eigenes Kommunikationssystem (vgl. Abbildung 3.6). Dadurch ist auch die Kontrolle der Kommunikationsstruktur des integrierten Anwendungssystems gewährleistet. Durch die zunehmende Verfügbarkeit von standardisierten Plattformen als Grundlage für objektintegrierte Anwendungssysteme wird ferner das Integrationsziel bezüglich des Merkmals Aufgabenträgerbezug unterstützt.

Zusammenfassend lässt sich feststellen, dass das Integrationskonzept Objektintegration die Integrationsziele bezüglich aller genannten Integrationsmerkmale am besten unterstützt. Dies beruht auf der Tatsache, dass das Konzept der Objektintegration eine Weiterentwicklung der anderen Konzepte darstellt. Dazu wurden Methoden entwickelt, mit denen auch die von den anderen Konzepten nicht unterstützten Integrationsziele erreicht werden können [Fers92, 25].

3.2.3.3 Zusätzliche Merkmale eines objektintegrierten verteilten Anwendungssystems

Da die Objektintegration die genannten Integrationsziele am Besten erfüllt, wird dieses Integrationskonzept in der vorliegenden Arbeit auch einem verteilten System zugrunde gelegt. Daraus resultieren folgende zusätzliche Merkmale eines verteilten Systems [FeSi94, 4; Sinz02a, 1066]:

- Die Komponenten eines verteilten Systems sind über gemeinsame Kommunikationskanäle **lose gekoppelt**. Sie interagieren durch den Austausch von **Nachrichten**, die sie über die gemeinsamen Kommunikationskanäle schicken, miteinander.
- Darüber hinaus **kapselt** jede Komponente eines verteilten Systems einen lokalen **Zustandspeicher** und zugehörige **Operatoren**.

Wie bereits erläutert, kann das Konzept des verteilten Systems auf allen Ebenen einer Informationssystem-Architektur angewendet werden. Dabei können auf jeder Ebene ein ebenenspezifischer Verteilungsgrad und ebenenspezifische Entwurfsziele zur Abgrenzung bzw. Integration von Anwendungssystemen angegeben werden. Der **ebenenspezifische Verteilungsgrad** verringert sich, je mehr Komponenten eng gekoppelt werden [FeSi94, 4]. Durch die **ebenenspezifischen Entwurfsziele** wird festgelegt, welche Komponenten auf einer Ebene eng zu koppeln sind. Folglich bestimmen sie den entsprechenden Verteilungsgrad einer Ebene. Die ebenenspezifischen Entwurfsziele hängen voneinander ab, so dass beim Entwurf eines Anwendungssystems ein simultanes Entscheidungsproblem auftritt. Dieses kann nur aufgelöst werden, indem der Entwurf eines Anwendungssystems schrittweise von der obersten Ebene zur untersten Ebene durchgeführt wird [FeSi94, 12].

Auf der Ebene des verteilten Anwendungssystems, die auch der hauptsächliche Untersuchungsgegenstand der vorliegenden Arbeit ist, entsprechen die Entwurfsziele den Integrationszielen aus Abschnitt 3.2.3.1. Demzufolge bestimmt das gewählte Integrationskonzept den Verteilungsgrad eines verteilten Anwendungssystems. In der vorliegenden Arbeit wird ein verteiltes Anwendungssystem aus den in Abschnitt 3.2.3.2 genannten Vorteilen gegenüber anderen Integrationskonzepten objektintegriert. Die folgende Tabelle fasst noch einmal die Integrationsmerkmale, die zugehörigen Integrationsziele und die jeweiligen Unterstützungstechniken zur Erfüllung der Integrationsziele eines objektintegrierten verteilten Anwendungssystems zusammen.

Merkmal	Ziel	Grund	Technik
Redundanz	Vermeidung	Konsistenz und Wirtschaftlichkeit	Datenredundanz: Methoden der Datenmodellierung Funktionsredundanz: zusätzliche Vorgangsobjekttypen
	Erhaltung	Fehlertoleranz und Performanz	Siehe Merkmal Konsistenz
Verknüpfung	Kontrolle	Minimaler Verknüpfungsgrad Schnelle und sichere Nachrichtenübertragung Robustheit bezüglich Störungen und Erweiterungen	Eigenständiges Kommunikationssystem
Konsistenz	Erhaltung der semantischen Integrität	konsistente Systemzustände aus Modellierungssicht	Kommunikation mithilfe von Nachrichten
	Erhaltung der operationalen Integrität	Bei parallelen Zustandsübergängen: konsistente Systemzustände vor und nach Zustandsübergängen	Transaktionsverwaltung der zugrunde liegenden Basissysteme
Zielorientierung	Zielbezogene Koordination der Teilaufgaben	Erreichung der Ziele der Gesamtaufgabe	Vorgangssteuerung mithilfe von Vorgangsobjekttypen
Aufgabenträgerbezug	Unabhängigkeit vom Aufgabenträger	Verfolgung bisher genannter Integrationsziele unabhängig von niedrigeren Aufgabenträgern	Nutzung von abstrakten Maschinen, hier: standardisierten Plattformen

Tabelle 3.4: Integrationsmerkmale, -ziele und -techniken objektintegrierter verteilter Anwendungssysteme

3.2.4 Verteilungstransparenz als Voraussetzung für das Formalziel Verteilbarkeit

Wie in Abschnitt 3.2.1 erwähnt, ist die Verteilungstransparenz ein wesentliches Merkmal eines verteilten Systems und somit auch eines verteilten Anwendungssystems. Sie wird bei einem schichtweisen Aufbau eines Anwendungssystems von speziellen Basismaschinen (z. B. Middleware-Systemen, siehe Abschnitt 5.3.6.2) gewährleistet. Zur genaueren Formulierung der Anforderungen an diese Basismaschinen muss die Verteilungstransparenz weiter in **Teiltransparenzen** zerlegt werden.

In der vorliegenden Arbeit wird Bezug auf die Teiltransparenzen des **Referenzmodells für offene verteilte Systeme (Reference Model for Open Distributed Processing, RM-ODP)** genommen. Dieses Referenzmodell wurde durch die ISO (International Standardisation Organization) und die ITU (International Telecommunications Union) standardisiert und wird inzwischen weltweit akzeptiert [Putm01, xxxiv]. Das Ziel des Referenzmodells ist die Unterstützung der Konstruktion offener verteilter Systemarchitekturen. Dafür stellt es reichhaltige und präzise definierte Konzepte für die verteilte Informationsverarbeitung und Techniken zur Spezifikation von Architekuren offener verteilter Systeme zur Verfügung [Putm01, 75 f]. Zur Verteilungstransparenz enthält das RM-ODP ein eigenes Rahmenwerk, das 8 Teiltransparenzen definiert und Konstruktions- bzw. Strukturierungsregeln zur technischen Umsetzung der Teiltransparenzen aufweist [Putm01, 377 ff; TaST02, 5 f]. Diese 8 Teiltransparenzen stellen laut RM-ODP jedoch nur eine initiale Menge an Transparenzen dar. Entwickler, die anhand dieses Referenzmodells Architekturen verteilter Anwendungssysteme konstruieren, können die Teiltransparenzen um weitere, anwendungsspezifische Teiltransparenzen ergänzen [Putm01, 380]. Im vorliegenden Fall der Entwicklung verteilter betrieblicher Anwendungssysteme müssen beispielsweise zusätzliche Teiltransparenzen definiert werden, die einer Verwendung des Anwendungssystems durch eine Vielzahl von Nutzern mit unterschiedlichen Nutzungsrechten Rechnung trägt. Dies ist beispielweise bei betrieblichen Anwendungssystemen der Fall, die Vertriebs- oder Abrechnungsaufgaben unterstützen. Diese zusätzlichen Teiltransparenzen sind die Nebenläufigkeitstransparenz, die Sicherheitstransparenz und die Skalierungstransparenz.

In Tabelle 3.5 werden die einzelnen Teiltransparenzen mit dem jeweiligen Zweck für den Betrieb verteilter betrieblicher Anwendungssysteme kurz beschrieben und in Tabelle 3.6 entsprechende Nutzenbeispiele dazu angegeben. Dabei wird der Begriff „Nutzer“ differenziert verwendet. Falls die Außensicht eines verteilten Anwendungssystems im Vordergrund steht, sind mit dem Begriff „Nutzer“ sowohl Anwendungssysteme als auch Personen gemeint. Wird speziell die Innensicht betrachtet, werden unter dem Begriff „Nutzer“ ausschließlich Komponenten des verteilten Anwendungs-

systems verstanden. Dieses Begriffsverständnis wird auch in den folgenden Abschnitten zugrunde gelegt, solange auf kein anderes Verständnis explizit hingewiesen wird.

Durch die Verteilungstransparenz mit ihren Teiltransparenzen kann die Realisierung von Qualitätskriterien und zugehörigen Ausprägungen, die ein verteiltes Anwendungssysteme aus Nutzersicht erfüllen soll, vor dem Nutzer verborgen werden. Dazu gehören unter anderem die Mobilität, Sicherheit, Skalierbarkeit, Fehlertoleranz, Nebenläufigkeit und Synchronisation [Putm01, 17 ff; CoDK01, 16 ff; TaST02, 4 ff; Beng00, 26 ff]. Außerdem soll mithilfe der Transparenzen auch die Verteilung ausgenutzt werden [Beng00, 27]. Für die Realisierung der Verteilungstransparenz sorgen, wie bereits erwähnt, spezielle Basismaschinen, auf die in Abschnitt 5.3.6 näher eingegangen wird.

Nr.	Teiltransparenz	Beschreibung	Zweck
1	Zugriffstransparenz	Verbergen von Details des Zugriffs auf einen von einer Komponente angebotenen Dienst vor dem Nutzer. Dazu gehören unterschiedliche Aufrufmechanismen, Datenformate und Schnittstellen.	Unterstützung der Interaktion zwischen Komponenten
2	Ortstransparenz	Verbergen des physischen Orts einer Komponente vor dem Nutzer	Unterstützung des Auffindens einer Komponente ohne Kenntnis der Systemtopologie zum Zwecke einer darauffolgenden Interaktion
3	Migrationstransparenz	Verbergen eines Ortswechsels einer Komponente vor dem Zugriff durch einen Nutzer	Unterstützung der Mobilität einer Komponente vor der Nutzung
4	Relokationstransparenz	Verbergen eines Ortswechsels einer Komponente während der Nutzung durch einen Nutzer	Unterstützung der Mobilität einer Komponente während der Nutzung
5	Replikationstransparenz	Verbergen der Redundanz einer Komponente vor dem Nutzer	Ermöglichung des Klonens von Komponenten

Nr.	Teiltransparenz	Beschreibung	Zweck
6	Transaktions-transparenz	Verbergen der koordinierenden Aktivitäten von Komponenten im Rahmen von Transaktionen vor dem Nutzer	Konsistenzerhaltung von Komponenten, die sich in überlappenden oder konkurrierenden Ausführungen befinden
7	Fehlertranspa-renz	Verbergen von Systemfehlern und entsprechenden Behebungsmechanismen vor der Komponente selbst und dem Nutzer	Beibehaltung eines konsistenten und stabilen Zustands der Komponente bei einem Fehler
8	Persistierungs-transparenz	Verbergen des Weiterbestehens einer Komponente bei ihrer Deaktivierung vor dem Nutzer	Unterstützung der Persistierung von Komponenten
9	Nebenläufig-keitstransparenz	Verbergen der Teilung einer Komponente auf mehrere Nutzer vor den Nutzern	Unterstützung der mehrfachen konkurrierenden Nutzung von Komponenten
10	Sicherheits-transparenz	Verbergen der Mechanismen für die Sicherheit von Komponenten, Interaktionen und dem System vor dem Nutzer	Unterstützung der Sicherheit von Komponenten, Interaktionen und dem System
11	Skalierungs-transparenz	Verbergen der Anpassung der Anzahl redundanter Komponenten bei veränderlicher Nutzerzahl vor dem Nutzer	Unterstützung der Anpassung des Systems an veränderliche Nutzerzahlen

Tabelle 3.5: Verteilungstransparenzen (in Anlehnung an [Putm01, 393 f; TaST02, 5 f; Beng00, 26 ff])

	Teiltransparenz										
Nutzen	1	2	3	4	5	6	7	8	9	10	11
Minimierung des Entwicklungsaufwands einer Komponente (Schnittstellen, Zugriffsart, Zugriffssicherheit)	+										
Verringerung der Fehleranfälligkeit des Systems	+										
Erhöhung der Systemrobustheit[1]	+	+				+					
Minimierung der Systemänderungen	+	+									
Entkopplung des physischen Auffindens von der Nutzung einer Komponente		+									
Ermöglichung eines Systemlastausgleichs			+	+	+						+
Erhöhung der Systemverfügbarkeit[1]			+	+	+				+	+	+
Erhöhung der Systemsicherheit			+								
Erhöhung der Systemzuverlässigkeit[1]				+	+	+		+		+	+
Erhöhung der Skalierbarkeit des Systems					+				+		+
Erhaltung der Konsistenz						+					
Erhöhung der Fehlertoleranz[1]							+				
Erhöhung der Systemverlässlichkeit[1]							+				
Ermöglichung einer Ressourcenteilung								+	+		
Verbesserung der Erholungsmöglichkeit nach Systemfehlern								+			
Erhöhung der Systemsicherheit										+	
Erhaltung der Systemintegrität										+	

+ = trifft zu

[1] Zu den Begriffen Systemrobustheit, Systemzuverlässigkeit, Systemverfügbarkeit, Systemverlässlichkeit und Fehlertoleranz, siehe z. B. [TaST02; CoDK01; Putm01; GrRe93; BMR+98]

Tabelle 3.6: Nutzen der Verteilungstransparenzen (in Anlehnung an [Putm01, 393 f; TaST02, 5 f])

3.2.5 Faktoren zur Unterstützung der Verteilbarkeit

Neben der geforderten Verteilungstransparenz muss ein verteiltes Anwendungssystem zusätzliche nicht-funktionale Anforderungen aufweisen, die zwar wesentliche Voraussetzungen für dessen Verteilbarkeit darstellen, aber auch unabhängig von einer möglichen Verteilung an ein Anwendungssystem gestellt werden können. Dazu gehören insbesondere die Qualitätsfaktoren Heterogenität und Offenheit [Putm01, 17 f; CoDK01, 16 ff; TaST02, 8 f].

Heterogenität bezeichnet im vorliegenden Zusammenhang die Unterschiedlichkeit von zugrundeliegenden Basismaschinen, wie z. B. Programmiersprachen, Middleware-Systemen, Betriebssystemen oder Computer- bzw. Netzwerkhardware. Die Forderung nach Heterogenität bedeutet somit, dass verteilte Anwendungssysteme gegebenenfalls auf einer Vielzahl unterschiedlicher Basismaschinen aufgebaut werden sollen. [CoDK01, 16 f; Putm01, 18]. Diese Forderung entspricht im Wesentlichen dem in Abschnitt 3.2.3.1 genannten Integrationsziel bezüglich des Merkmals Aufgabenträgerbezug, nämlich der Verringerung der Abhängigkeit von spezifischen Eigenschaften niedrigerer Aufgabenträger. Deswegen kann dieser Forderung auch mit dem gleichen Lösungskonzept begegnet werden: Der Nutzung abstrakter Maschinen, die verteilte Anwendungssysteme von Technologieentwicklungen und spezifischen Leistungsparametern der Basismaschinen abschirmen. Auf spezielle Realisierungsformen dieses Lösungskonzepts wird in Abschnitt 5.3.6 noch näher eingegangen.

Ein **offenes verteiltes Anwendungssystem** ist ein System, das auf mehrere Arten erweitert und reimplementiert werden kann [CoDK01, 17]. Dadurch wird die Portabilität und Interoperabilität eines verteilten Systems bzw. seiner Komponenten ermöglicht [CoDK01, 17; TaST02, 8; Putm01, 6]. Wie bereits in Abschnitt 3.1.4.2 erläutert, versteht man unter Portabilität den Aufwand, der für den Einsatz eines Systems oder einer Komponente in einer anderen Hardware- oder Software-Umgebung als der ursprünglich vorgesehenen notwendig ist. Die Portabilität hängt somit eng mit der Erfüllung des bereits im letzten Absatz erwähnten Integrationsziels Unabhängigkeit von niedrigeren Aufgabenträgern zusammen. Ebenfalls in Abschnitt 3.1.4.2 wurde die Interoperabilität als der notwendige Aufwand zur Verbindung zweier unterschiedlicher Systeme oder Komponenten charakterisiert. Der dabei zugrundeliegende Zweck ist zum einen die Koexistenz der unterschiedlichen Systeme oder Komponenten und zum anderen die Erzeugung einer zielgerichteten Kooperation zwischen ihnen [TaSt02, 8; Putm01, 17]. Die Offenheit eines verteilten Anwendungssystems kann erreicht werden durch eine standardisierte Beschreibung und Veröffentlichung seiner Schnittstellen. Darauf wird in Abschnitt 4 näher eingegangen.

Ein weiterer häufig genannter Qualitätsfaktor, den verteilte Anwendungssysteme aufweisen sollen, ist die Leistungsfähigkeit bzw. Performanz [CoDK01, 44 f; Beng00, 28]. Diese wird im Rahmen der Arbeit nicht weiter betrachtet, da die Leistungsfähigkeit eines Systems aus einer Kombination an Ausprägungen der bisher genannten Qualitätsfaktoren und -kriterien resultiert und somit keinen eigenständigen Qualitätsfaktor darstellt.

3.3 Resultierende Forderungen an den softwaretechnischen Entwurf und an die Implementierung von wiederverwendbaren und verteilbaren Anwendungssystemen

In diesem Abschnitt werden nun die bisher erörterten Merkmale, Kriterien und Faktoren, die eine Wiederverwendung und Verteilung von Entwicklungserzeugnissen ermöglichen bzw. unterstützen als grundlegende Forderungen für die folgenden Ausarbeitungen formuliert.

Bis auf wenige Ausnahmen adressieren diese Forderungen gleichermaßen Entwicklungserzeugnisse im softwaretechnischen Entwurf und Entwicklungserzeugnisse in der Implementierung. Damit die Durchgängigkeit beim Übergang vom softwaretechnischen Entwurf zur Implementierung gewahrt bleibt, sollten die Entwicklungserzeugnisse in der Implementierung auf dem gleichen Grundkonzept basieren, wie die Entwicklungserzeugnisse im softwaretechnischen Entwurf. Dabei ist das Entwicklungserzeugnis im softwaretechnischen Entwurf auf einem höheren Abstraktionsniveau als in der Implementierung. Je konkreter ein Entwicklungserzeugnis im softwaretechnischen Entwurf ist, desto geringer ist die Lücke, die zwischen diesem und der Implementierung geschlossen werden muss. Allerdings nimmt bei zunehmender Konkretisierung eines Entwicklungserzeugnisses auch dessen Wiederverwendbarkeit ab (vgl. Abschnitt 3.1.4.3). Deswegen müssen Entwicklungserzeugnisse, die sowohl im softwaretechnischen Entwurf als auch in der Implementierung auf dem gleichen Grundkonzept basieren, in beiden Phasen so abstrakt wie nötig und dabei so konkret wie möglich sein. Eine Auswahl der derzeit am häufigsten verwendeten Erzeugnisarten, die dieser Anforderung genügen, enthält Abschnitt 3.5. Dort wird auch deren Wiederverwendbarkeits-, Verteilbarkeits- und Wiederverwendungsunterstützung anhand der im Folgenden aufgestellten Forderungen bewertet. Diese Forderungen lassen sich unterteilen in solche, die an das Vorgehen des softwaretechnischen Entwurfs und der Implementierung gestellt werden, und solche, die das Entwicklungserzeugnis erfüllen soll.

Erstgenannte Forderungen resultieren aus den in Abschnitt 3.1.3.7 ausgewählten Aspektausprägungen des Formalziels Wiederverwendung, das sich auf die Durchführung der Konstruktionsaufgabe Entwicklung von Anwendungssystemen bezieht.

Zu den letztgenannten Forderungen gehören die Kriterien und Faktoren, die zur Erfüllung des Formalziels Wiederverwendbarkeit, das an diese Entwicklungserzeugnisse gestellt wird, beitragen. Sie wurden in Abschnitt 3.1.4 erörtert. Darüber hinaus resultieren weitere Forderungen aus den in Abschnitt 3.2 beschriebenen Merkmalen verteilter Anwendungssysteme und aus den damit zusammenhängenden Integrationszielen. Sie ermöglichen bzw. unterstützen die Erfüllung des Formalziels Verteilbarkeit, das ebenfalls an Entwicklungserzeugnisse im softwaretechnischen Entwurf und in der Implementierung gestellt wird.

Durch die Erfüllung der Forderungen, die hinsichtlich des Formalziels Wiederverwendbarkeit an das Entwicklungserzeugnis gerichtet sind, wird auch die Realisierung der Forderungen, die sich auf die Wiederverwendung im softwaretechnischen Entwurf und in der Implementierung beziehen, ermöglicht bzw. unterstützt. Dieser, bereits in Abschnitt 3.1.4.3 angedeutete Zusammenhang wird in Tabelle 3.7 noch einmal in übersichtlicher Form zusammengefasst.

	Forderungen an die Wiederverwendung			
Forderungen an das Entwicklungserzeugnis	Abdeckung aller Bereiche	kompositionell	generativ	Black Box
hohes Abstraktionsniveau	+	+	+	+
Allgemeingültigkeit	+	+	+	+
hierarchische Strukturierung	+	+	+	+
Modularität	+	+		+
Verständlichkeit	+	+	+	+
Nachvollziehbarkeit	+	+	+	+
standardisierte und formalisierte Beschreibung	+	+	+	+
Anpassbarkeit	+	+	+	+
Erweiterbarkeit	+	+	+	+
Interoperabilität	+			
Portierbarkeit	+			
Qualität	+			+

+ = unterstützt

Tabelle 3.7: Forderungen an das Vorgehen und an die Entwicklungserzeugnisse im softwaretechnischen Entwurf und in der Implementierung

In Abschnitt 4 werden die erarbeiteten Forderungen schließlich zur Formulierung eines präskriptiven Grundkonzepts der Komponentenorientierung herangezogen, das als Grundlage für das komponentenbasierte Software-Architekturmodell verwendet wird.

3.4 Prinzipien und Modelle zur Unterstützung der Wiederverwendung, Wiederverwendbarkeit und Verteilbarkeit

Für den softwaretechnischen Entwurf und die Implementierung qualitativ hochwertiger Anwendungssysteme wurden seit den 1960er Jahren verschiedene Prinzipien und Modelle entwickelt, die zum Teil aufeinander aufbauen. Dieser Abschnitt stellt Prinzipien und Modelle vor, die insbesondere die Wiederverwendung, Wiederverwendbarkeit und Verteilbarkeit von Anwendungssystemen unterstützen. Einige wurden bereits im Rahmen der Ausführungen zur Wiederverwendung und zur Integration bzw. Verteilung kurz skizziert. Im Folgenden werden sie nun genauer erläutert und durch weitere Prinzipien ergänzt.

3.4.1 Zugrundeliegende Beziehungsarten

Die Prinzipien und Modelle für den softwaretechnischen Entwurf und die Implementierung basieren auf den grundsätzlichen Beziehungsarten Abstraktion und Assoziation, die zwischen bestimmten Objekten der Betrachtung bestehen können. Deswegen werden diese zunächst erläutert.

3.4.1.1 Hierarchische Beziehungsart Abstraktion

Zur Entwicklung qualitativ hochwertiger Anwendungssysteme muss deren hohe Komplexität, die auf die ständig wachsende Zahl an Anforderungen und die stetige Weiterentwicklung der zugrundeliegenden Technologien zurückzuführen ist, beherrschbar gemacht werden. Ein bewährtes Prinzip zur Bewältigung von Komplexität ist die Abstraktion.

Die **Abstraktion** kann unterschieden werden in Abstraktionsprozess und Abstraktionsergebnis. In der Literatur bezeichnet der Begriff „Abstraktion“ häufig sowohl den Prozess als auch das Ergebnis. Dies führt jedoch zwangsläufig zu Missverständnissen, weshalb hier explizit zwischen Abstraktionsprozess und Abstraktionsergebnis unterschieden wird. Der **Abstraktionsprozess** wird allgemein definiert als „das stufenweise Heraussondern bestimmter Merkmale“ unterschiedlicher Objekte der Betrachtung mit dem Ziel, das „Gleichbleibende und Wesentliche“ dieser Objekte zu erkennen (vgl. [Broc86, 84; Meye88, 175 f.]). Das Wesentliche hängt dabei „einerseits von der fachlichen Fragestellung, andererseits von Aufmerksamkeit, Einsicht und Bildung“ ab [Broc86, 84]. Dadurch wird die Komplexität eines bestimmten Sachverhalts, die durch die Anzahl und Beziehungen seiner unterschiedlichen Objekte

gegeben ist, zielorientiert reduziert. Der komplexitätsreduzierte Sachverhalt bildet schließlich das **Abstraktionsergebnis**. Die Umkehrung der Abstraktion ist die so genannte **Konkretisierung**.

Die Komplexitätsreduktion als wesentliches Merkmal der Abstraktion findet sich auch in der allgemeinen Definition der Modellbildung als Merkmal der Vereinfachung wieder (vgl. Abschnitt 2.2.3), so dass jeder Abstraktionsprozess gleichzeitig auch eine allgemeine Modellbildung darstellt.

Bereits in der allgemeinen Definition der Abstraktion wird das **stufenweise Vorgehen** angesprochen. Dies ist insbesondere bei der Abstraktion von realen Systemen, die im Rahmen der allgemeinen Systemtheorie in Abschnitt 2.1.1 vorgestellt wurden, notwendig, da deren hohe Komplexität häufig nicht in einem einzigen Abstraktionschritt zu einer handhabbaren Komplexität reduziert werden kann. Bei einem stufenweisen Vorgehen bilden alle Objekte auf einer Stufe eine so genannte **Abstraktionsebene**. Dabei stehen die Objekte benachbarter Abstraktionsebenen in einer bestimmten **semantischen Beziehung (Relation)** zueinander. Aufgrund der Komplexitätsreduktion stellt diese semantische Beziehung auch gleichzeitig eine Rangordnung zwischen den Objekten benachbarter Abstraktionsebenen dar. Ein abstraktes Objekt ist definitionsgemäß ranghöher, als die Objekte, von denen abstrahiert wurde. Die Existenz einer Rangordnung zwischen Objekten und die Anordnung von Objekten gleicher Rangordnung auf einer Ebene sind auch die wesentlichen Eigenschaften von Hierarchien, so dass man in diesem Zusammenhang von **Abstraktionshierarchien** sprechen kann [Balz82, 31].

Auf der Basis der allgemeinen Definition, der semantischen Beziehung und der hierarchischen Struktur sind nun verschiedene **Abstraktionsformen** identifizierbar. Zunächst werden damit die Abstraktionsformen erklärt, die jeder Mensch bei der Beschreibung der Wirklichkeit zur Komplexitätsreduktion durchführt. Dazu gehören die Konzeptbildung, die Typisierung, die Klassifizierung und die Außensichtbildung [MaOd99, 36 ff; FeSi01, 18].

Konzeptbildung

Mit der semantischen Relation *ist_vom_Konzept* wird die **Konzeptbildung** beschrieben. Grundsätzlich ist ein **Konzept** „eine Idee oder Vorstellung, die wir in unserem Bewusstsein auf Dinge oder Objekte anwenden“ [MaOd99, 38]. Das bedeutet, dass ein Konzept sowohl eine **Intension**, nämlich eine Idee oder Vorstellung, als auch eine **Extension** in Form einer Menge konkreter Objekte, auf die die Intension zutrifft, besitzt (vgl. Abbildung 3.7) [MaOd99, 41]. Es können jedoch auch Konzepte definiert werden, zu denen es vorerst keine zugehörige Extension gibt bzw. geben soll. Der Zweck darin liegt in der Festlegung, dass Objekte dieser Art entweder nicht existieren sollen oder vorausgeahnt werden können [MaOd99, 40].

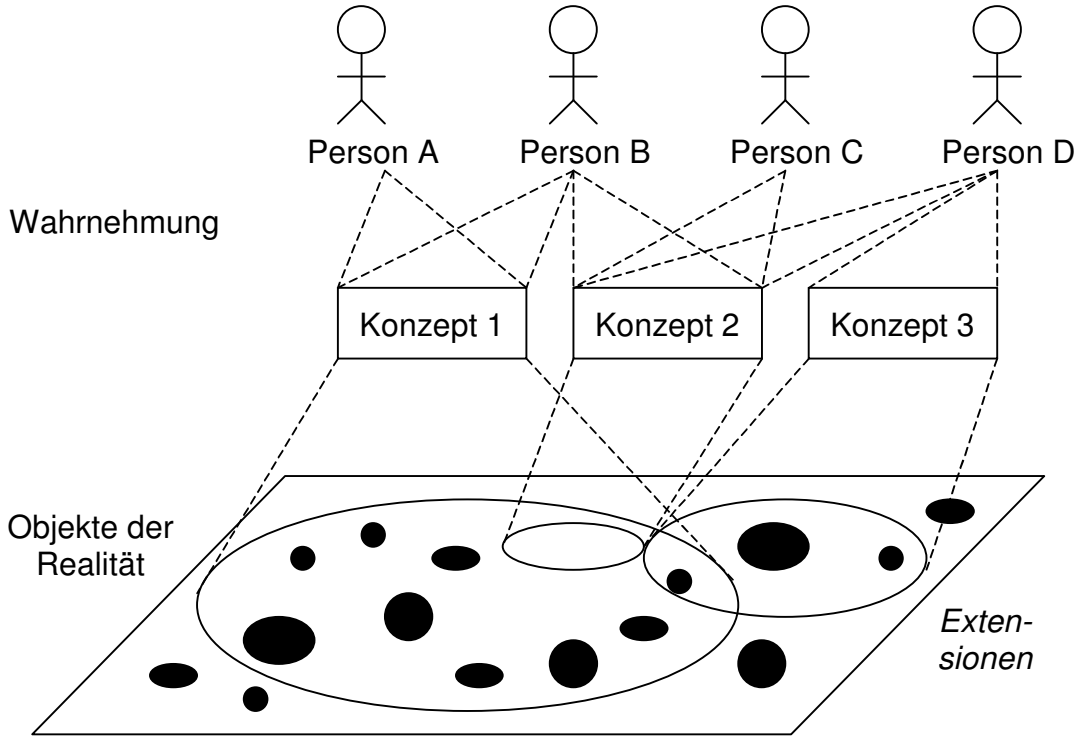


Abbildung 3.7: Konzeptbildung (in Anlehnung an [MaOd99, 39])

Wie man unschwer erkennen kann, stellt die Konzeptbildung eine Komplexitätsreduktion dar. Damit ist die Ebene der Intension ranghöher als die Ebene der Extension, d. h. die beiden Ebenen stehen in einer hierarchischen Beziehung zueinander. Aufgrund der hierarchischen Mehrstufigkeit ist jedes Konzept wiederum ein Objekt, das anhand eines übergeordneten Konzepts beschrieben werden kann. Somit ist ein Konzept ein Objekt auf einer höheren Abstraktionsstufe.

Typisierung

Die semantische Relation *ist_vom_Typ* beschreibt eine **Typisierung**. Ein **Typ** wird allgemein bezeichnet als „a precise characterization of structural or behavioral properties which a collection of entities all share“ [Booc94, 65] und ist somit inhaltlich gleichzusetzen mit einem Konzept [MaOd99, 62]. Infolgedessen verfügt ein Typ ebenfalls über eine Intension und über eine Extension, die in einer hierarchischen Beziehung zueinander stehen [MaOd99, 63]. Die Abstraktionsebene, auf der Typen beschrieben werden, heißt **Typebene**. Analog wird die Ebene, auf der sich die zugehörigen konkreten Objekte befinden, **Ausprägungsebene** genannt.

Klassifizierung

Mit der Abstraktionsform **Klassifizierung** wird ein gegebenes Objekt, auf das ein bestimmtes Konzept bzw. ein bestimmter Typ zutrifft, der Extension des Konzepts bzw. Typs zugeordnet [MaOd99, 56]. D. h., die Klassifizierung ist „der Vorgang oder das Ergebnis der Anwendung eines Konzepts (bzw. eines Typs) auf ein Objekt“

[MaOd99, 109]. Demzufolge ergänzt sie die Abstraktionsformen Konzeptbildung bzw. Typisierung um die Zuordnung bzw. das Entfernen von konkreten Objekten zu oder von der jeweiligen Extension [MaOd99, 56]. Die klassifizierten Objekte werden häufig auch **Instanzen** genannt. Dementsprechend heißt die semantische Relation der Klassifizierung auch *ist_Instanz_von*. Anstatt existierende Objekte zu klassifizieren, können zu jedem Konzept bzw. zu jedem Typ auch neue Objekte erzeugt und der Extension zugeordnet werden. Dieser Vorgang stellt die Umkehrung der Klassifizierung dar und wird **Erzeugung** oder auch **Instantiierung** genannt.

Außensichtbildung

Zur Bewältigung der Komplexität eines einzelnen Objekts hat sich als weitere Abstraktionsform die so genannte **Außensichtbildung** bewährt, die bereits im Rahmen der Erläuterungen zur allgemeinen Systemtheorie kurz skizziert wurde (vgl. Abschnitt 2.1.1). Danach wird ein Objekt als System interpretiert, das aus der Außensicht oder aus der Innensicht betrachtet werden kann. Die **Innensicht** des Systems beschreibt die Struktur und das Verhalten seiner Komponenten. Aus der **Außensicht** sind nur die Schnittstellen des Systems bekannt, die sein äußeres Verhalten gegenüber der Umwelt spezifizieren. Die Innensicht eines Systems oder Teilsystems muss mit der zugehörigen Außensicht verträglich sein, d. h. das äußere Verhalten realisieren [FeSi01, 18]. Dies legt die semantische Relation der Außensichtbildung zu *realisiert* fest. Die Umkehrung der Außensichtbildung ist die **Innensichtbildung**.

Zur Erläuterung der weiteren Abstraktionsformen werden anstatt konkreter Objekte die Konzepte bzw. Typen verwendet, die auf sie zutreffen. Trotzdem beziehen sich die folgenden Abstraktionsformen auf die konkreten Objekte eines Konzepts bzw. Typs, und nicht auf die Konzepte bzw. Typen selbst. Konzepte bzw. Typen sind somit stellvertretend für alle Objekte, die sie beschreiben, zu verstehen. Da die Begriffe „Konzept“ und „Typ“ inhaltlich gleich sind, wird im Folgenden, aus Gründen der besseren Lesbarkeit, ausschließlich der Begriff „Typ“ verwendet.

Die folgenden Abstraktionsformen stehen orthogonal zueinander und werden deswegen auch als **Grundformen der Abstraktion** bezeichnet [Booc94, 13 ff].

Generalisierung

Die semantische Relation *ist_ein* beschreibt eine **Generalisierung**. Dabei wird für Typen, die Gemeinsamkeiten aufweisen, ein übergeordneter Typ erstellt, der diese Gemeinsamkeiten enthält und somit diese Typen verallgemeinert [MaOd99, 111]. Die Umkehrung der Generalisierung ist die **Spezialisierung** [MaOd99, 112].

Aggregation

Mit der semantischen Relation *ist_Teil_von* wird eine **Aggregation** beschrieben. Hierbei wird für verschiedene untergeordnete Typen ein übergeordneter Typ gebil-

det, der sich konzeptionell aus den untergeordneten Typen zusammensetzt [MaOd99, 115]. Der übergeordnete Typ heißt Ganzes oder **Aggregat**, die untergeordneten Typen heißen **Teile**. Als Umkehrung der Aggregation gilt die **Zerlegung** bzw. **Dekomposition**. In der Software-Technik wird darüber hinaus zwischen den Begriffen „Aggregation“ und „Komposition“ unterschieden (vgl. Abschnitt 3.5.1.1). Im allgemeinen Sprachgebrauch gelten sie aber meistens als synonym.

Die vorgestellten Abstraktionsformen wurden jeweils aus Sicht der niedrigeren Abstraktionsebene beschrieben. Dieses Verfahren eignet sich insbesondere zur Analyse und Beschreibung von komplexen Systemen und wird **Bottom-Up-Verfahren** genannt. Zur Gestaltung und Konstruktion von komplexen Systemen, wie z. B. betrieblichen Informations- und Anwendungssystemen bietet sich der umgekehrte Weg an [Balz82, 73]. Hierbei geht man von einer abstrakten Vorstellung des zu konstruierenden Systems aus und konkretisiert diese stufenweise zum realen System. Dieses so genannte **Top-Down-Verfahren** hat den Vorteil, dass sich der Systementwickler bzw. -nutzer auf jeder Ebene auf das jeweils Wesentliche konzentrieren kann und sich nicht im Detail verliert [Balz82, 72]. In der Literatur wird dieses Verfahren auch häufig „**schrittweise Verfeinerung**“ genannt (vgl. Abschnitt 3.4.2).

Durch den Einsatz der Abstraktion werden aufgrund der dabei durchgeführten Komplexitätsreduktion folgende Vorteile für den Systementwickler bzw. -nutzer erreicht [Balz82, 29]:

- Erkennen, ordnen, klassifizieren und gewichten von wesentlichen Merkmalen,
- Erkennen von allgemeinen Charakterstika
- Trennen des Wesentlichen vom Unwesentlichen.

3.4.1.2 Nicht-hierarchische Beziehungsart Assoziation

Wie bereits erläutert, stellen die Abstraktionsformen Konzeptbildung, Typisierung, Klassifizierung, Generalisierung und Aggregation hierarchische Beziehungen zwischen Objekten dar. Neben diesen hierarchischen Beziehungen können auch nicht-hierarchische Beziehungen zwischen Objekten existieren, die nach den Bedürfnissen des Betrachters entsprechend definierbar sind [MaOd99, 95]. Solche Beziehungen werden **Assoziationen** genannt [MaOd99, 95]. Eine Assoziation kann wiederum durch einen Typ beschrieben werden, wobei jede konkrete Beziehung zwischen Objekten, die durch diese Assoziation beschrieben wird, eine Ausprägung des Typs darstellt [MaOd99, 96 ff]. Falls die durch eine Assoziation verbundenen Objekte ebenfalls durch ihre Typen beschrieben werden, können zu einer Assoziation so genannte **Kardinalitäten** angegeben werden. Diese schreiben vor, wieviele konkrete Objekte der an der Assoziation beteiligten Typen minimal und maximal in Beziehung stehen sollen bzw. dürfen [MaOd99, 95 f].

3.4.2 Prinzipien zur Unterstützung der Wiederverwendung, Wiederverwendbarkeit und Verteilbarkeit

Die oben genannten Abstraktionsformen bilden die Grundlage für weitere Prinzipien, die zum Teil aufeinander aufbauen. Sie wurden ursprünglich für den Entwurf und die Implementierung von Programmen entwickelt, können jedoch für den softwaretechnischen Entwurf und die Implementierung von Anwendungssystemen verallgemeinert werden. Hierbei ist zunächst das Geheimnisprinzip zu nennen, das als grundlegendes Entwurfsprinzip gilt. Es kann anhand des Prinzips der Trennung von Schnittstelle und Implementierung und bzw. oder des Prinzips der Kapselung realisiert werden [BMR+98, 394]. Eine besonders ausführliche Realisierungsform des Geheimnisprinzips ist das Prinzip der Modularisierung. Im Zusammenhang mit dem Prinzip der Modularisierung entstanden auch das Prinzip der Trennung von Belangen (*Separation of Concerns*) und das Prinzip der maximalen inneren Kohärenz und der minimalen äußeren Bindung (vgl. [BMR+98, 392 ff.; Balz82, 190 ff]). Die Zusammenhänge zwischen den genannten Prinzipien werden in Abbildung 3.8 verdeutlicht.

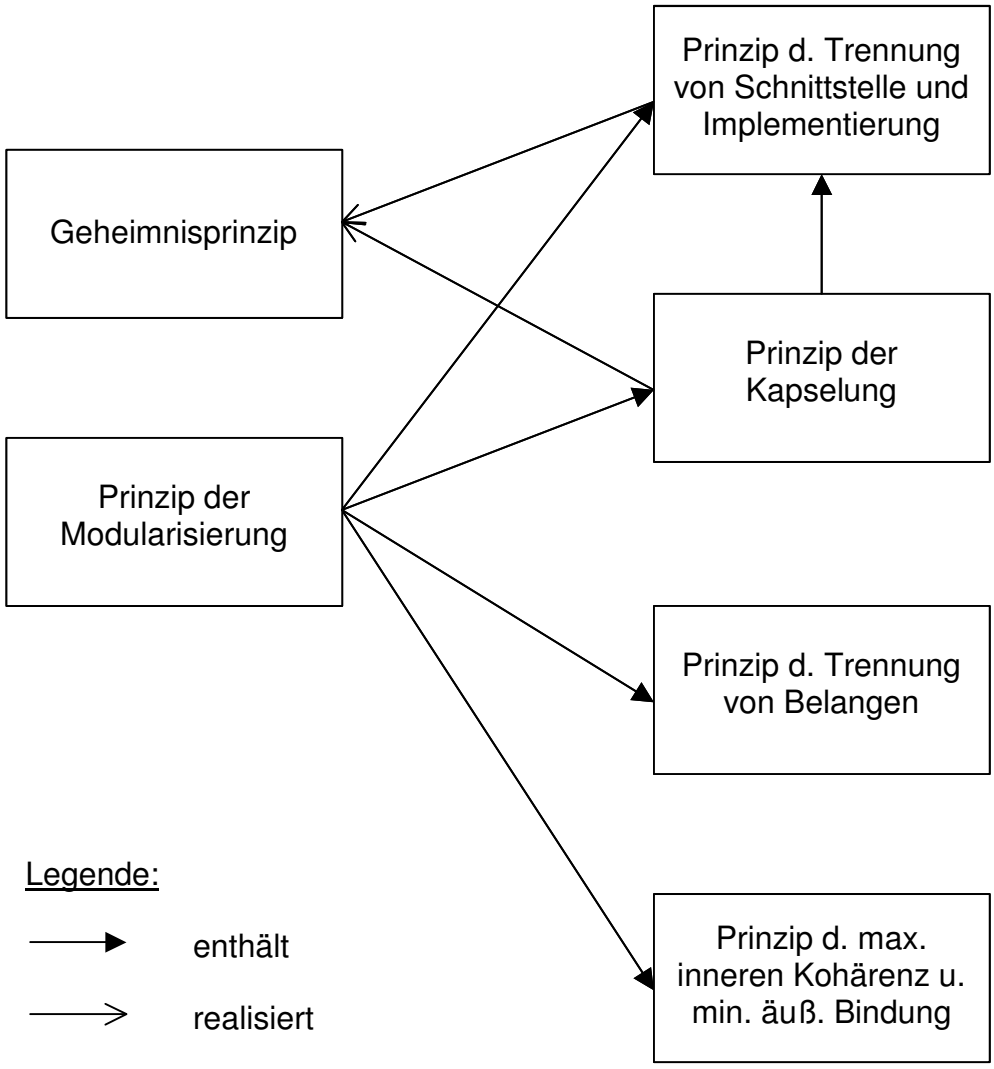


Abbildung 3.8: Zusammenhänge zwischen den Entwurfsprinzipien

Geheimnisprinzip

Das **Geheimnisprinzip** fordert im Allgemeinen einen modularen Aufbau eines Programms mit der Besonderheit, dass jedes Modul seine Implementierung nach außen hin verbirgt. Diese Implementierung ist von der des gesamten Anwendungssystems zu unterscheiden. Das Verbergen geschieht durch die Definition einer so genannten Schnittstelle, die einfach gehalten werden und nur unveränderliche Informationen besitzen soll. Mithilfe der Schnittstelle soll das Modul nach außen verständlich beschrieben und die veränderlichen oder detaillierten Informationen im Modul gekapselt werden [PaCW87, 163; Parn02, 403]. Dies erhöht unter anderem die Wiederverwendbarkeit der Module [PaCW89, 141 ff; PaCW87, 162 f].

Die möglichen Arten der Unterteilung von Programmen in Module ist nicht Gegenstand des Geheimnisprinzips. Hinweise dazu sind im Prinzip der Trennung von Belangen und im Prinzip der Modularisierung enthalten.

Durch die gemeinsame Anwendung des Geheimnisprinzips und des Prinzips der schrittweisen Verfeinerung entsteht ein hierarchischer Aufbau des Programms aus Modulen, wobei jedes Modul eine Schnittstelle aufweist und einen oder mehrere von außen nicht zugängliche Module kapselt. Das Prinzip der schrittweisen Verfeinerung fordert generell, dass die Entwicklung von Programmen in sequentiellen Schritten durchgeführt und dabei eine Programmieraufgabe in jedem Schritt in eine Reihe von untergeordneten Programmieraufgaben zerlegt werden soll. Diese können daraufhin unabhängig voneinander entwickelt werden [Wirt71, 227].

Prinzip der Trennung von Schnittstelle und Implementierung

Das Geheimnisprinzip kann durch die Anwendung des **Prinzips der Trennung von Schnittstelle und Implementierung** realisiert werden. Wie bereits erwähnt, ist die Implementierung eines Moduls von der Implementierung des gesamten Anwendungssystems zu unterscheiden. Im einzelnen verlangt dieses Prinzip, dass

- die Schnittstelle die Funktionalität und die Verwendungsweise eines Moduls definiert und von außen zugänglich ist,
- die Implementierung die Funktionen und Datenstrukturen für die in der Schnittstelle angebotene Funktionalität enthält und von außen nicht zugänglich ist. Darüberhinaus kann die Implementierung noch weitere Funktionen und Datenstrukturen enthalten, die jedoch nur innerhalb des Moduls verwendet werden (vgl. [BMR+98, 396]).

Die Implementierung ist also für das Verhalten des Moduls zuständig, das die entsprechende Schnittstelle nach außen anbietet. Dafür bedient sie sich einer bestimmten Datenstruktur. Die Schnittstelle schirmt wiederum die Implementierung vor Zugriffen von außen ab.

Prinzip der Kapselung

Der eben genannte Aufbau ist auch im allgemeinen **Prinzip der Kapselung** enthalten: „Encapsulation is the process of compartmentalizing the elements of an abstraction, that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation." [Booc94, 50].

Prinzip der Modularisierung

Das **Prinzip der Modularisierung** hängt eng mit dem Geheimnisprinzip und seinen Realisierungsformen zusammen. Ein Anwendungssystem wird anhand von bestimmten Kriterien, auf die später noch eingegangen wird, in in sich abgeschlossene Module zerlegt, die unabhängig voneinander entwickelt, getestet und gewartet werden können [Dene91, 212 ff; Balz82, 214 ff; Pres87, 228 f]. Dafür müssen die Module so entworfen sein, dass sie möglichst wenige Verbindungen untereinander aufweisen [Meye90, 19]. Dies wird durch das Prinzip der maximalen inneren Kohärenz und der minimalen äußeren Bindung erreicht, das im Anschluss an diese Ausführungen noch näher beschrieben wird. Außerdem müssen sie zur Erzielung der Unabhängigkeit das Geheimnisprinzip realisieren, d. h. die Implementierung eines Moduls muss nach außen hin verborgen sein und alle notwendigen Spezifikationsinformationen werden in einer einfach gehaltenen und wohldefinierten Schnittstelle beschrieben [Balz82, 214 f; Dene91, 212 f; Meye90, 11 ff]. Nach BUSCHMANN ET AL. [BMR+98] dient das Prinzip der Modularisierung dazu, „die Komplexität eines Systems durch Einführung von wohldefinierten und dokumentierten Grenzen [...] zu meistern" [BMR+98, 394]. Daraus resultiert die wesentliche Aufgabe, zu entscheiden, wie das System in Module zerlegt werden soll, und wie die Teile die logische Struktur einer Anwendung bilden [BMR+98, 394]. Dies wird im Prinzip der Trennung von Belangen behandelt, auf das in diesem Abschnitt noch näher eingegangen wird.

PARNAS übertrug das Prinzip der Modularisierung in den 1970er Jahren von den Ingenieurdisziplinen in die Entwicklung von Programmsystemen und konkretisierte es diesbezüglich [Parn02, 400]. Dabei forderte PARNAS, Module zu entwickeln, deren Schnittstellen einfach gehalten sind und deren Implementierungen, die meist häufigen Änderungen unterliegen, dem Nutzer verborgen bleiben [Parn02, 403 f]. Historisch gesehen führte das Prinzip der Modularisierung schließlich zur Formulierung des allgemeineren Geheimnisprinzips und des Prinzips der Kapselung als eines seiner möglichen Realisierungsformen.

Prinzip der maximalen inneren Kohärenz und der minimalen äußeren Bindung

Aus dem Prinzip der Modularisierung und dem Geheimnisprinzip entstand das **Prinzip der maximalen inneren Kohärenz und der minimalen äußeren Bindung** (*Cohesion and Coupling*) [CoYo79; Pres87, 230 ff]. Die äußere Bindung konzentriert sich

auf Beziehungen zwischen Modulen, die innere Kohärenz auf Beziehungen innerhalb eines Moduls. Mit der **äußeren Bindung** wird die Stärke des Zusammenhangs zwischen zwei Modulen gemessen [Pres87, 232]. Je stärker der Zusammenhang ist, desto schwerer sind die beteiligten Module zu verstehen, zu ändern und zu korrigieren [BMR+98, 395]. Mit der **Kohärenz** wird die Stärke der Bindung zwischen den Elementen eines Moduls (Datenobjekttypen, Operatoren und Anweisungen) gemessen [Balz82, 219; BMR+98, 395]. So kann auch die Qualität einer Modularisierung beurteilt werden [Balz82, 219]. Je stärker die Bindung zwischen den Elementen eines Moduls ist, desto höher ist die Kontextunabhängigkeit, Wiederverwendbarkeit und Erweiterbarkeit des Moduls [Balz82, 228]. Entlang der Skala der Bindungsstärke können 8 so genannte **Bindungsarten** unterschieden werden [Balz82, 219]. Die Bindungsarten mit der höchsten Bindungsstärke sind die informale und die funktionale Bindung. Eine **funktionale Bindung** liegt vor, wenn alle Elemente eines Moduls an der Realisierung eines einzigen, abgeschlossenen Vorgangs beteiligt sind [Balz82, 226]. Von einer **informalen Bindung** spricht man, wenn alle Operatoren eines Moduls auf einer einzigen Datenstruktur operieren [Balz82, 227]. Abstrakte Datentypen, die in Abschnitt 3.5.1 beschrieben werden, sind Beispiele für Module mit informaler Bindung. Zur Erhöhung der Bindungsstärke werden beide Bindungsarten miteinander kombiniert. Daraus resultieren Module, die einen einzigen, abgeschlossenen Vorgang mithilfe eines Programms realisieren, das auf Operatoren zugreift, die auf einer einzigen Datenstruktur operieren.

Zur Verdeutlichung der Tatsache, dass mit diesem Prinzip insbesondere auf die Bindungsstärke Bezug genommen wird, hat sich als Übersetzung des englischen Begriffs „Coupling“ in diesem Zusammenhang der Begriff „Bindung“ anstelle des Begriffs „Kopplung“ durchgesetzt.

Prinzip der Trennung von Belangen

Wie bereits angedeutet, wird im Prinzip der Modularisierung nicht festgelegt, nach welchen Kriterien ein Anwendungssystem in Module zerlegt werden soll. Auf dieses Problem geht unter anderem PARNAS in seinem Artikel „On the Criteria To Be Used in Decomposing Systems into Modules“ ein [Parn72].

Darin distanziert er sich von der damalig und teilweise auch heute noch gebräuchlichen Meinung, dass Module gleichzusetzen sind mit Unterprogrammen. Das dabei verwendete Zerlegungskriterium orientiert sich an den hauptsächlichen Verarbeitungsschritten eines Programms [Parn72, 1056]. Dieses, auch als **funktionale Dekomposition** bekannte Verfahren wird heutzutage noch vereinzelt für den Entwurf von Anwendungssystemen angewendet.

Dem gegenüber identifiziert PARNAS Module eher als „responsibility assignment[s]“ [Parn72, 1054]: „The *modularization* includes the design decisions which must be

made *before* the work on independent modules can begin" [Parn72, 1054; Hervorhebungen im Original]. Diese Entwurfsentscheidungen werden schließlich anhand des Geheimnisprinzips innerhalb des Moduls vor den Nutzern versteckt: „Each module [...] is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings" [Parn72, 1056; Hervorhebung durch den Verfasser]. PARNAS erkennt die wesentlichen Vorteile dieses Vorgehens gegenüber der Zerlegung in Verarbeitungsschritte in den folgenden Punkten [Parn72, 1055 f]:

- Entwurfsentscheidungen, die einen bestimmten Aspekt des Programmsystems betreffen, sind in einem Modul gekapselt. Dadurch beschränken sich Änderungen solcher Entwurfsentscheidungen und ihre Auswirkungen nur auf ein Modul und verteilen sich nicht auf das gesamte Programm. Dieser Vorteil fördert unter anderem eine höhere Änderbarkeit und Abgeschlossenheit der Module.
- Einfach gehaltene Modulschnittstellen enthalten noch keine umfangreichen Entwurfsentscheidungen, die bei der Entwicklung der zugehörigen Module berücksichtigt werden müssten. Deswegen können Module mit einfach gehaltenen Schnittstellen früher und unabhängiger voneinander entwickelt werden. Dies verbessert vor allem die Verständlichkeit und Abgeschlossenheit der Module.
- Da ein Modul alle Entwurfsentscheidungen kapselt, die zu einem Aspekt des Programmsystems gehören, kann es ohne Kenntnis der Funktionsweise anderer Module entwickelt und verstanden werden. Auch dies verbessert insbesondere die Verständlichkeit und Abgeschlossenheit der Module.

Dieses Vorgehen unterstützt somit auch die Wiederverwendbarkeit der Module, da in Abschnitt 3.1.4 die Verständlichkeit von Entwicklungserzeugnissen im Kriterium der Nachvollziehbarkeit, die Abgeschlossenheit von Entwicklungserzeugnissen im Kriterium der Modularität und die Änderbarkeit von Entwicklungserzeugnissen als eigener wiederverwendungsunterstützender Faktor formuliert wurden. Allerdings klärt PARNAS in seinem Artikel nicht, welche Entwurfsentscheidungen zu einem Aspekt gehören und demnach in einem Modul gekapselt werden sollen.

Die Zerlegung eines Anwendungssystems anhand von bestimmten Aspekten, wie z. B. Verantwortlichkeiten in in sich abgeschlossene Teile ist allgemein bekannt als **Prinzip der Trennung von Belangen** (*Separation of Concerns*) [BMR+98, 394 f; SiMe00, 101 ff].

3.4.3 Modelle zur Unterstützung der Wiederverwendung, Wiederverwendbarkeit und Verteilbarkeit

Auf der Grundlage der genannten Prinzipien wurden in den letzten Jahren mehrere Modelle zur Strukturierung von Anwendungssystemen entwickelt. Davon sind für die

vorliegende Arbeit insbesondere diejenigen von Interesse, die die Wiederverwendung, Wiederverwendbarkeit und Verteilbarkeit von Anwendungssystemen unterstützen.

Modell der Nutzer- und Basismaschine

Eine spezielle Realisierungsform des Prinzips der Trennung von Belangen stellt das in Abschnitt 2.2.5.1 eingeführte Modell der Nutzer- und Basismaschine dar. Es trennt die **fachlichen Belange**, repräsentiert durch die Nutzermaschine, von den **technischen Belangen**, repräsentiert durch die Basismaschine. Dabei berücksichtigen die fachlichen Belange funktionale Anforderungen an ein Anwendungssystem, während die technischen Belange nicht-funktionale Anforderungen adressieren. Diese Form der Trennung erfolgt somit vertikal.

Die Abstraktionsform des Modells der Nutzer- und Basismaschine ist die der Außensicht- bzw. Innensichtbildung. Als zugehöriges Abstraktionsverfahren eignet sich das Top-Down-Verfahren, weil dieses, wie bereits erläutert, den Entwurf von komplexen Systemen, wie z. B. betrieblichen Anwendungssystemen am besten unterstützt.

Dieses Modell unterstützt die Wiederverwendbarkeit und Verteilbarkeit eines Anwendungssystems nicht nur durch die Reduzierung der Komplexität, sondern auch durch die Merkmale Portabilität und Interoperabilität [FeSi01, 289 ff].

ADK-Strukturmodell

Orthogonal zur vertikalen Trennung von Belangen anhand des Modells der Nutzer- und Basismaschine, kann eine horizontale Trennung von Belangen, z. B. eine Trennung von fachlichen Belangen, durchgeführt werden. Vorschläge für geeignete Trennungskriterien enthält das komponentenbasierte Software-Architekturmodell, das in Abschnitt 5.1 vorgestellt wird.

Ein übergeordneter Ansatz zur horizontalen Trennung von Belangen ist das **ADK-Strukturmodell**, das ein Anwendungssystem orthogonal zum Modell der Nutzer- und Basismaschine in Teilsysteme unterteilt (vgl. Abbildung 3.9) [FeSi01, 289 f]. Korrespondierend zum Verständnis eines Anwendungssystems als Aufgabenträger für informationsverarbeitende Aufgaben, orientiert sich die Unterteilung am Aufgabenkonzept, das in Abschnitt 2.1.3 vorgestellt wurde. Folglich wird ein Anwendungssystem in mindestens ein Teilsystem für das Aufgabenobjekt, mindestens ein Teilsystem für das Lösungsverfahren und mindestens ein Teilsystem für die Interaktionskanäle gegliedert. Daraus ergeben sich **Teilsysteme für die Datenverwaltung (D)**, **Teilsysteme für die Anwendungsfunktionen (A)** und **Teilsysteme für die Kommunikation (K)**. Darüberhinaus werden die Teilsysteme für die Kommunikation weiter zerlegt in die Bereiche Kommunikation mit Personen (K_P) und Kommunikation mit Maschinen (K_M). Die Teilsysteme sind in sich abgeschlossen, von einander unabhängig und

über Nachrichtenaustausch miteinander lose gekoppelt [FeSi01, 289 f]. Demzufolge realisiert das ADK-Strukturmodell auch das Prinzip der Modularisierung (vgl. Abschnitt 3.4.2).

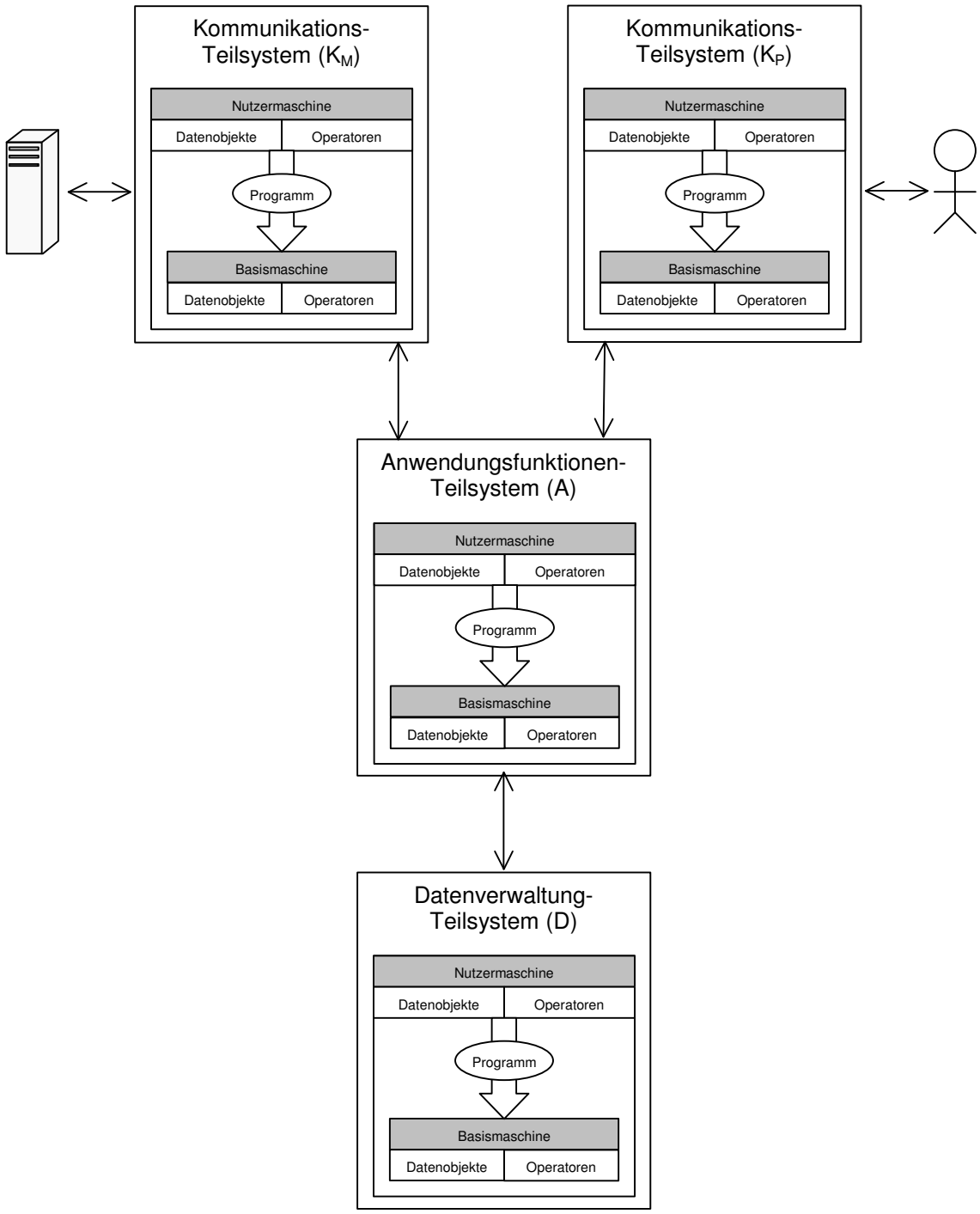


Abbildung 3.9: ADK-Strukturmodell eines Anwendungssystems (in Anlehnung an [FeSi01, 290])

Die einzelnen Teilsysteme sind gemäß dem Modell der Nutzer- und Basismaschine aufgebaut und bestehen jeweils aus einer Nutzermaschine und einer Folge von Nutzer- und Basismaschinen mit den zugehörigen Programmen, die die Nutzermaschine realisieren. Die logische Kommunikation zwischen den Teilsystemen mittels Nach-

richtenaustausch erfolgt über die jeweiligen Nutzermaschinen, für den physischen Transport der Nachrichten sind die jeweiligen Basismaschinen verantwortlich [FeSi01, 290].

Somit weist das ADK-Strukturmodell analog zum Modell der Nutzer- und Basismaschine das Merkmal der Komplexitätsreduzierung und das Merkmal der Flexibilität auf. Darüber hinaus unterstützt dieses Modell das Ziel der Standardisierung und schafft damit die Voraussetzung für das im Rahmen des Modells der Nutzer- und Basismaschine erläuterte Ziel der Portierbarkeit [FeSi01, 290 f]. Die Unterstützung der Standardisierung liegt in der Abtrennung des A-Teilsystems begründet, das normalerweise den meisten Änderungen unterliegt. Dadurch beschränkt sich der hauptsächliche Wartungsaufwand auf dieses Teilsystem und verteilt sich nicht auf das Gesamtsystem. Folglich können die anderen Teilsysteme größtenteils standardisiert sein bzw. werden. Beispielsweise sind für das Teilsystem der Datenverwaltung standardisierte Datenbanksysteme einsetzbar. Durch die Trennung der Teilsysteme Anwendungsfunktionen, Datenverwaltung und Kommunikation kann für jedes eine spezifische Basismaschine verwendet werden. Dies ermöglicht den Einsatz von standardisierten Basismaschinen mit standardisierten Schnittstellen für jedes Teilsystem und somit auch die Portierbarkeit des Anwendungssystems (vgl. Abschnitt 3.1.4.2). Beispiele standardisierter Basismaschinen sind Entwicklungs- und Ausführungsplattformen für die Anwendungsfunktionen, Datenbankverwaltungssysteme (DBVS) für die Datenverwaltung und User-Interface-Management-Systeme (UIMS) für die Kommunikation mit Personen.

Mehrschicht-Architekturansätze

Anhand des ADK-Strukturmodells lassen sich auch die derzeit häufig verwendeten **Mehrschicht-Architekturen**, auch Multi-Tier- oder N-Tier-Architekturen genannt, erläutern (siehe z. B. [NMM+00, 8 f; BMR+98, 49; SiMe00, 112 ff.; DePe02, 18 f; Mons01, 6]). Sie unterteilen ein Anwendungssystem grundsätzlich in eine **Präsentationsschicht**, eine **Anwendungsschicht** und eine **Datenhaltungsschicht** (vgl. Abbildung 3.10). Dies entspricht im ADK-Strukturmodell der Unterteilung in die Teilsysteme Kommunikation mit Personen, Anwendungsfunktionen und Datenverwaltung. Davon ausgehend kann jede Schicht weiter verfeinert werden [NMM+00, 8]. Die meisten Ansätze beschränken sich darauf, die Präsentationsschicht in eine Schicht für die Präsentationslogik und in eine Schicht für die Präsentationsobjekte zu zerlegen. Neuere Ansätze zerlegen analog dazu die Anwendungsschicht in eine Schicht für die Anwendungslogik und in eine für die Anwendungsobjekte [BMR+98, 49]. Der Ansatz von NOACK ET AL. [NSK97, 55] ist einer der weitreichendsten Ansätze: Darin wird sowohl die Präsentations- als auch die Anwendungsschicht auf die

beschriebene Art und Weise zerlegt. Außerdem wird die Datenhaltungsschicht in eine logische und eine physische Datenzugriffsschicht verfeinert.

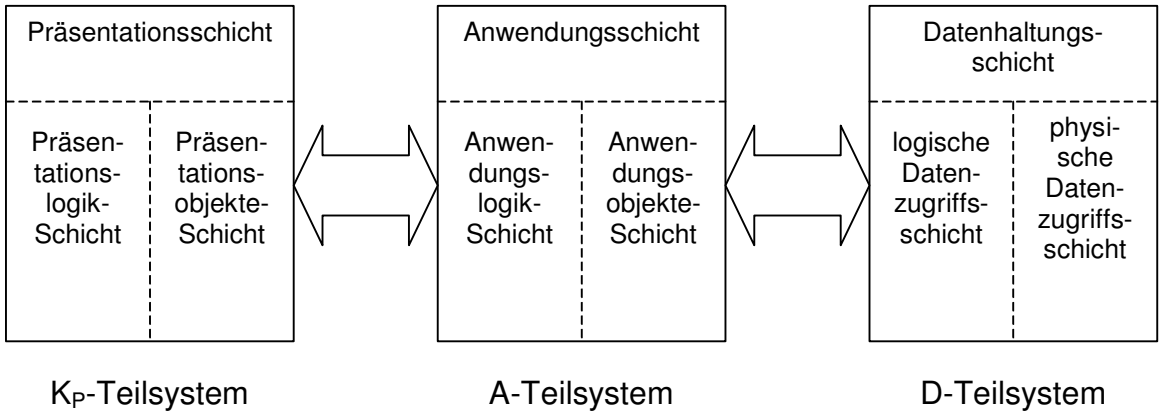


Abbildung 3.10: Mehrschicht-Architekturen

3.4.4 Unterstützte Forderungen

Die erläuterten Prinzipien und Modelle unterstützen die Erfüllung der in Abschnitt 3.3 aufgestellten Forderungen, die an ein wiederverwendbares und verteilbares Entwicklungserzeugnis gestellt werden, entweder vollständig oder zumindest zum Teil. In Tabelle 3.8 werden diese Forderungen zusammen mit den entsprechenden Prinzipien und Modellen nochmals aufgeführt. Dabei werden die Prinzipien, die in einem Modell enthalten sind, nicht mehr explizit aufgeführt.

Forderung	**Prinzipien/Modelle**	**Unterstützung**
Hohes Abstraktionsniveau	Abstraktionsprinzip	+
Allgemeingültigkeit	Abstraktionsprinzip	+
hierarchische Strukturierung	Abstraktionsprinzip	+
Modularität	Abstraktionsprinzip, Geheimnisprinzip, Prinzip der Trennung von Schnittstelle und Implementierung, Prinzip der Kapselung, Prinzip der Modularisierung, Prinzip der Trennung von Belangen, Prinzip der maximalen inneren Kohärenz und der minimalen äußeren Bindung	+
Verständlichkeit	Abstraktionsprinzip	O

Forderung	**Prinzipien/Modelle**	**Unterstützung**
Nachvollziehbarkeit	Abstraktionsprinzip, Geheimnisprinzip, Prinzip der Trennung von Schnittstelle und Implementierung, Prinzip der Kapselung	O
Anpassbarkeit	Abstraktionsprinzip, Geheimnisprinzip, Prinzip der Trennung von Schnittstelle und Implementierung, Prinzip der Kapselung, Prinzip der Modularisierung, Prinzip der Trennung von Belangen, Prinzip der maximalen inneren Kohärenz und der minimalen äußeren Bindung	+
Erweiterbarkeit	Abstraktionsprinzip, Geheimnisprinzip, Prinzip der Trennung von Schnittstelle und Implementierung, Prinzip der Kapselung, Prinzip der Modularisierung, Prinzip der Trennung von Belangen, Prinzip der maximalen inneren Kohärenz und der minimalen äußeren Bindung	+
Interoperabilität	Modell der Nutzer- und Basismaschine	O
Portierbarkeit	Modell der Nutzer- und Basismaschine, ADK-Strukturmodell	+
Qualität	Abstraktionsprinzip, Geheimnisprinzip, Prinzip der Trennung von Schnittstelle und Implementierung, Prinzip der Kapselung, Prinzip der Modularisierung, Prinzip der Trennung von Belangen, Prinzip der maximalen inneren Kohärenz und der minimalen äußeren Bindung	O

+ = vollständig O = zum Teil

Tabelle 3.8: Unterstützung der Forderungen durch Prinzipien und Modelle

In dieser Tabelle sind noch keine Prinzipien zur Unterstützung der Forderungen bezüglich des Formalziels Verteilbarkeit enthalten. Zur Erfüllung der Forderungen, die sich aus den Integrationszielen ergeben, wurden in Abschnitt 3.2.3.2 verschiedene Integrationskonzepte vorgestellt. Die darauf folgenden Untersuchungen zeigten, dass die Objektintegration die Integrationsziele bezüglich der genannten Integrationsmerkmale im Vergleich zu anderen häufig verwendeten Integrationskonzepten am besten unterstützt. Deswegen wird für das präskriptive Grundkonzept der Komponentenorientierung, das in Abschnitt 4 vorgestellt wird, die Objektintegration als Konzept gefordert und ein verteiltes Anwendungssystem objektintegriert beschrieben.

Für die Forderungen, die aus dem Merkmal Verteilungstransparenz resultieren, ist zumindest eine Strukturierung eines wiederverwendbaren und verteilbaren Anwendungssystems nach dem Nutzer- und Basismaschinenmodell erforderlich, da, wie erwähnt, spezielle Basismaschinen für die Erfüllung dieser Forderungen sorgen. Auf diese speziellen Basismaschinen und die durch sie realisierten Lösungskonzepte zur Erfüllung der Teiltransparenzen wird im Rahmen der Vorstellung des komponentenbasierten Software-Architekturmodells in Abschnitt 5.3.6 näher eingegangen.

3.5 Unterstützung der Wiederverwendung, Wiederverwendbarkeit und Verteilbarkeit ausgewählter Erzeugnisarten

Anhand der in Abschnitt 3.3 formulierten Forderungen, die eine bestmögliche Wiederverwendung, Wiederverwendbarkeit und Verteilbarkeit von Entwicklungserzeugnissen im softwaretechnischen Entwurf und in der Implementierung ermöglichen bzw. unterstützen sollen, werden nun verschiedene Erzeugnisarten hinsichtlich ihrer Unterstützung der Wiederverwendung, Wiederverwendbarkeit und Verteilbarkeit untersucht. Dafür muss zunächst eine geeignete Auswahl an Erzeugnisarten getroffen werden.

Durch die Einführung der Wiederverwendung in die Anwendungssystementwicklung haben sich zwei Extremformen herausgebildet: Zum einen die individuelle Entwicklung von Anwendungssystemen (Individualsoftware) ohne jegliche Nutzung der geplanten, systematischen Wiederverwendung. Diese Form ist aus den in Abschnitt 3.1.2 genannten Gründen für heutige Unternehmen nicht mehr wählbar. Zum anderen die Übernahme und Anpassung von Standardanwendungssystemen (Standardsoftware). Sie repräsentieren die Wiederverwendung eines kompletten, lauffähigen Anwendungssystems und wurden im Rahmen des Wiederverwendungsaspekts Erzeugnis in Abschnitt 3.1.3.6 bereits erläutert. Insbesondere im Bereich der Entwicklung betrieblicher Anwendungssysteme wird verstärkt auf diese Form gesetzt.

Standardsoftware

Mit dem Einsatz von Standardsoftware sind eine Reihe von Vorteilen verbunden. Zum einen sind die Kosten der Einführung und der Anpassung in der Regel geringer als die einer vollständigen Neuentwicklung [Ambe99b, 41]. Außerdem besitzt Standardsoftware einen hohen Reifegrad und damit unter anderem die Möglichkeit zum Erwerb von zusätzlichem betriebswirtschaftlichem bzw. organisatorischem Know-How [HaNe01, 152]. Da sie meist von größeren Software-Unternehmen für den Massenmarkt hergestellt wird, können auch professionelle Schulungen und Support-Dienstleistungen angeboten werden. Nicht zuletzt trägt Standardsoftware auch zur inner- und zwischenbetrieblichen Integration betrieblicher Aufgaben bei, da sie be-

reits ein integriertes Anwendungssystem darstellt und gewöhnlich in mehreren Unternehmen eingesetzt ist.

Auf der anderen Seite ist die Einführung von Standardsoftware in ein Unternehmen nicht ganz unproblematisch. Zum einen können oftmals die Geschäftsprozesse eines Unternehmens nicht optimal in der Standardsoftware abgebildet werden [Raue96, 3]. Dies führt oftmals dazu, dass die Geschäftsprozesse eines Unternehmens der ausgewählten Standardsoftware angepasst werden und nicht umgekehrt [Raue96, 3; Szyp98, 5]. Zum anderen müssen zur unternehmensspezifischen Anpassung der Standardsoftware üblicherweise entweder zusätzliche individuelle Komponenten angebaut oder umfangreiche Parametersätze durchgetestet werden [MeHL97, 15 ff]. Darüber hinaus benötigen kleinere und mittelständische Unternehmen oft nur einen Bruchteil der von einer Standardsoftware angebotenen Funktionen [MeHL97, 15]. Außerdem muss die Standardsoftware eine Integrationsfähigkeit durch Standardschnittstellen und Schnittstellen zu Altsystemen, eine Portierbarkeit z. B. auf unterschiedliche Hardware-Plattformen und eine Unterstützung unterschiedlicher Sprachen gewährleisten [Ambe99b, 41]. Beispielsweise kann die Integration mit vorhandenen Altsystemen einen nicht unerheblichen Aufwand bedeuten. Nicht zuletzt muss durch eine gut funktionierende Betreuung und Schulung der Benutzer versucht werden, Vertrauen in und Aktzeptanz für die Standardsoftware aufzubauen. Dies alles kann dazu führen, dass die Kosten für die Einführung und Anpassung der Standardsoftware deren Beschaffungskosten übersteigen [Raue96, 3]. Als ein möglicher Ausweg kann darauf verzichtet werden, die Standardsoftware eng an die Spezifika des Unternehmens, wie z. B. an die Strategien, Strukturen und Prozesse anzupassen [MeHL97, 15]. Dies beeinträchtigt jedoch den Nutzen der Standardsoftware für ein Unternehmen erheblich [MeHL97, 15].

Mittelwege zwischen Individual- und Standardsoftware

Folglich müssen Mittelwege zwischen der Entwicklung von Indivdualsoftware und der Einführung von Standardsoftware gefunden werden. Aus softwaretechnischer Sicht bieten sich derzeit dafür die in Abschnitt 3.1.3.6 genannten Erzeugnisarten an. Dazu gehören Algorithmus, Funktions- oder Routinebibliothek, Klassenbibliothek, Analyse- und Entwurfsmuster, Framework und Software-Komponente.

Ein weiteres Auswahlkriterium ist die in Abschnitt 3.3 bereits angesprochene Existenz eines gemeinsamen Grundkonzepts für Entwicklungserzeugnisse im softwaretechnischen Entwurf und in der Implementierung. Wie dort bereits beschrieben, fördert dies eine Zusammenführung der beiden Phasen.

Damit fallen Algorithmen und Funktions- oder Routinebibliotheken aus der Auswahl heraus, da sie ausschließlich in der Implementierung eingesetzt werden können. Ebenso können Entwurfsmuster nur im softwaretechnischen Entwurf verwendet wer-

den. Allerdings werden sie als Bestandteile der vorzustellenden Software-Architektur in Abschnitt 5.1 wieder aufgegriffen.

Klasse

Das Grundkonzept der Objektorientierung, das im Folgenden behandelt wird, definiert die Erzeugnisart Klasse, die sowohl im softwaretechnischen Entwurf als auch in der Implementierung verwendet werden kann. Die häufigsten Einsatzbereiche von Klassen sind im softwaretechnischen Entwurf objektorientierte Anwendungssystementwürfe und in der Implementierung Klassenbibliotheken. Letztgenannte werden häufig zur Implementierung von objektorientierten Anwendungssystementwürfen verwendet. Beide Einsatzbereiche bilden ebenfalls Erzeugnisarten, die jedoch nur auf jeweils eine Phase beschränkt sind.

Framework

Eine weiterer Einsatzbereich von Klassen sind Frameworks, die, wie bereits in Abschnitt 3.1.3.6 skizziert, als Erzeugnisart den softwaretechnischen Entwurf und die Implementierung umfassen. Folglich werden Frameworks, neben Klassen und Software-Komponenten, in beiden Phasen eingesetzt.

Software-Komponente

Software-Komponenten wurden in Abschnitt 3.1.3.6 als wiederverwendbare Software-Teilsysteme definiert, die mit anderen Software-Teilsystemen zu einem Gesamtsystem verbunden werden können. Weiterhin ist unbestritten, dass Software-Komponenten sowohl im softwaretechnischen Entwurf als auch in der Implementierung existieren: „[...] components are complex design-level entities, that is, both abstractions and implementations“ [BrWa98, 41]. Dies ist jedoch, wie erwähnt, eine Minimaldefinition, die aus der Vielfalt der inzwischen verfügbaren Definitionen gebildet wurde und für detaillierte Untersuchungen nicht operational genug ist. Das gilt speziell für die vorgesehene Bewertung der Erzeugnisart Software-Komponente hinsichtlich der Wiederverwendungs-, Wiederverwendbarkeits- und Verteilbarkeitsunterstützung. Zu diesem Zweck fehlt ein allgemein anerkanntes Grundkonzept der Komponentenorientierung, analog zum noch folgenden Grundkonzept der Objektorientierung. In Kapitel 4 wird auf der Basis der bisherigen und der noch folgenden Ergebnisse ein präskriptives Grundkonzept der Komponentenorientierung entwickelt, in dessen Mittelpunkt ein methodisch hergeleitetes Software-Komponenten-Konzept steht. Das Grundkonzept ist wiederum Grundlage für das in Kapitel 5 beschriebene Software-Architekturmodell, mit dem speziell betriebliche komponentenbasierte Anwendungssysteme entworfen werden können.

Somit bezieht sich die folgende Untersuchung der Wiederverwendungs-, Wiederverwendbarkeits- und Verteilbarkeitsunterstützung auf die Erzeugnisarten Klasse und

Framework. Dafür werden die zugrunde liegenden Konzepte kurz vorgestellt und daraufhin anhand der in Abschnitt 3.3 aufgestellten Forderungen bewertet.

3.5.1 Erzeugnisart Klasse

Zur Erläuterung der Erzeugnisart Klasse ist es zunächst erforderlich, das Konzept der Datenabstraktion mit abstrakten Datentypen kurz darzustellen.

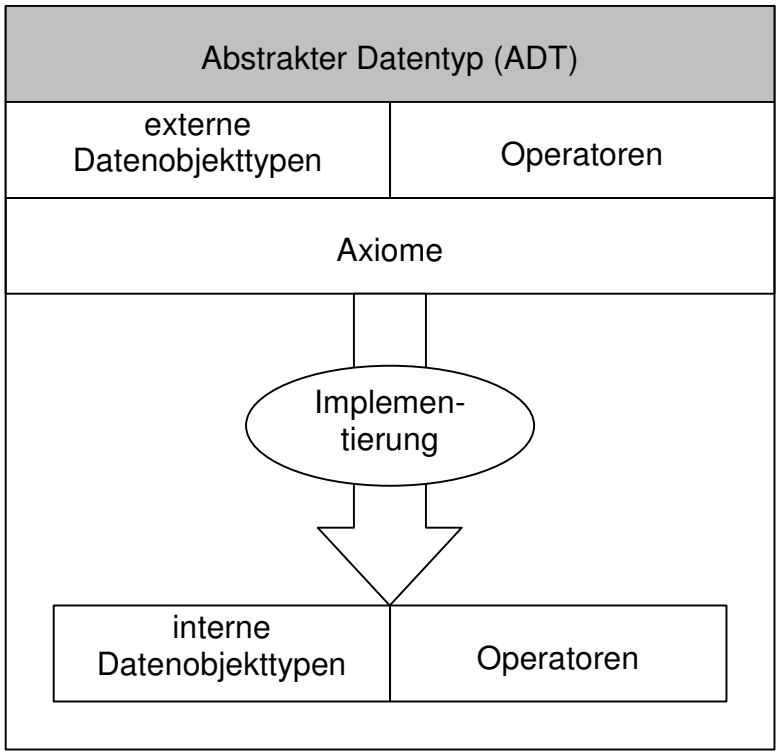


Abbildung 3.11: Abstrakter Datentyp (ADT) [FeSi01, 294]

Im Modell der Nutzer- und Basismaschine, das in Abschnitt 2.2.5.1 beschrieben wurde, werden die Datenobjekte und Operatoren der Nutzermaschine durch ein Programm realisiert, das wiederum auf den Datenobjekten und Operatoren der Basismaschine operiert. Daraus folgt, dass die Realisierung eines Datenobjekttyps einschließlich der zugehörigen Operatoren der Nutzermaschine nach außen hin verborgen bleibt. Dies kann wiederum anhand des Modells der Nutzer- und Basismaschine erläutert werden und führt schließlich zur Definition abstrakter Datentypen (vgl. Abbildung 3.11).

Abstrakte Datentypen sind Datentypen, die ihre innere Struktur verbergen und nur ihren Namen und die auf sie anwendbaren Operatoren nach außen hin bekanntgeben. Die Definition des Datentyps erfolgt also allein durch den Namen und durch die veröffentlichten Operatoren. Diese werden mithilfe eines Programms, welches auf die Datentypen der Innensicht zugreift, realisiert [FeSi01, 294 f; FeSi84, 156 ff].

3.5.1.1 Untersuchung anhand des Grundkonzepts der Objektorientierung

Eine bedeutende Weiterentwicklung des abstrakten Datentyps stellt der Objekttyp dar [FeSi01, 299; JCJÖ92, 49; MaOd92, 156 ff]. Das Aufkommen des Objekttyps begründete zugleich einen Paradigmenwechsel in der Anwendungssystementwicklung vom funktionalen zum objektorientierten Paradigma. Zunächst wurden objektorientierte Techniken nur in der Implementierung im Rahmen von objektorientierten Programmiersprachen eingesetzt [Booc94, 35; FoSc00, 2; BrSi02, 4; JäHe02, 67; MaOd99, 24]. Im Laufe der Zeit haben sie sich auch im softwaretechnischen Entwurf (*Object-oriented Design (OOD)*) und schließlich auch in der Analyse von Anwendungssystemen (*Object-oriented Analysis (OOA)*) etabliert [Booc94, 35 ff.; BrSi02, 4]. Dies führte zu einer Vielzahl unterschiedlicher objektorientierter Analyse- und Entwurfsansätze (siehe z. B. [Booc94; JCJÖ92; MaOd92; Meye90; RBP+93; WiWW90; FeSi01]). Eine historische Betrachtung der verbreitesten Ansätze enthält z. B. [FoSc00, 2 f]. Ein ausführlicher Vergleich dieser Ansätze ist beispielweise in [MaOd99, 357 ff] aufgeführt. Allen Ansätzen ist jedoch ein Grundverständnis der Objektorientierung gemein, das im Folgenden beschrieben wird. Die Unterschiede beschränken sich im Wesentlichen auf die Notation, die Terminologie oder auf spezifische Erweiterungen des genannten Grundverständnisses [MaOd99, 357; Fowl93, 2]. Dabei sind die Bereiche objektorientierte Entwurfsansätze und objektorientierte Programmiersprachen nicht klar getrennt, sondern die Grenzen zwischen ihnen sind eher fließend [JäHe02, 270]. Grundsätzlich gilt zwar die Abgrenzung zwischen dem softwaretechnischen Entwurf als Programmierung-im-Großen und der Implementierung als Programmierung-im-Kleinen auch bei der Entwicklung objektorientierter Anwendungssysteme. Demzufolge könnten in der folgenden Beschreibung alle Konzepte, die sich auf die Implementierung beziehen, eindeutig dem Bereich der objektorientierten Programmiersprachen, und alle Konzepte, die sich auf den softwaretechnischen Entwurf beziehen, eindeutig dem Bereich der objektorientierten Entwurfsansätze zugerechnet werden. Eine solche eindeutige Zuordnung ist allerdings nicht immer möglich. Dies liegt daran, dass der Ursprung des **Grundkonzepts der Objektorientierung**, wie bereits erwähnt, in der Programmierung zu finden ist, und deshalb in objektorientierten Entwurfsansätzen, die ebenfalls auf dem Grundkonzept der Objektorientierung basieren, auch Konzepte und Techniken der Implementierung enthalten sind. Demzufolge wird in der folgenden Beschreibung des Grundkonzepts der Objektorientierung eine explizite Differenzierung zwischen diesen beiden Bereichen nur dann vorgenommen, wenn die Zuordnung eindeutig ist.

Objektorientiertes Anwendungssystem

Das Grundkonzept der Objektorientierung, auf dem alle objektorientierten Entwicklungsansätze und Programmiersprachen beruhen, unterteilt ein Anwendungssystem

in eine Menge von Objekten, die lose gekoppelt über Nachrichten miteinander interagieren [Booc94, 17; FeSi01, 298]. Die Interaktion erfolgt dabei nach dem **Client-/Server-Prinzip**: Ein Client-Objekt beauftragt ein Server-Objekt mit der Ausführung eines Dienstes. Das Server-Objekt führt den angefragten Dienst, gegebenenfalls unter Zuhilfenahme von Diensten anderer Server-Objekte aus und gibt das Ergebnis der Dienstausführung an das Client-Objekt zurück. Aus dieser Beschreibung ist erkennbar, dass jedes Objekt sowohl die Rolle eines Clients als auch die eines Servers annehmen kann.

Objekt

Ein **Objekt** stimmt im grundsätzlichen Aufbau mit einem Exemplar eines abstrakten Datentyps, auch **Datenkapsel** genannt, überein [FeSi01, 300; Meye90, 64]. Es verbirgt, analog zur Datenkapsel, seinen Zustand, der sich aus den Zuständen der internen Datenobjekte zusammensetzt, und die Realisierungen der zugehörigen Operatoren, die für das Verhalten des Objekts zuständig sind (vgl. Abbildung 3.12). Der Zugriff auf den Zustand des Objekts erfolgt ausschließlich über die **Operatorschnittstellen**, auf die von außen zugegriffen werden kann. Sie umfassen üblicherweise den Namen des Operators, die Liste der Parameterdatentypen und den Rückgabedatentyp [GHJV96, 18]. Folglich basiert ein Objekt auf der Anwendung eines der Hauptprinzipien des Grundkonzepts der Objektorientierung, dem Prinzip der Kapselung [Booc94, 49 ff].

Darüber hinaus besitzt ein Objekt eine **Identität**, mit deren Hilfe es von allen anderen Objekten unterschieden werden kann, selbst wenn der Zustand und das Verhalten gleich sind [Booc94, 91 ff].

Im Unterschied zu Datenkapseln kommunizieren Objekte über den Austausch von Nachrichten miteinander (vgl. Abbildung 3.12). Eine Nachricht an ein Objekt löst dort die Durchführung eines Operators aus, wenn dessen Schnittstellenstruktur mit der Nachrichtenstruktur übereinstimmt [GHJV96, 18].

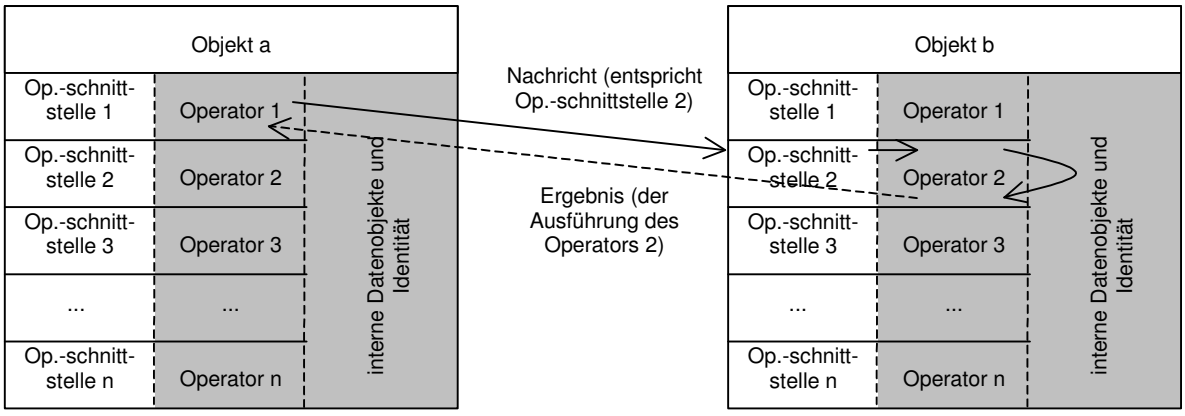


Abbildung 3.12: Objekte und Nachrichtenaustausch

Durch die Verwendung von Nachrichten wird die konzeptionelle Realisierung des Prinzips der Trennung von Schnittstelle und Implementierung deutlicher als bei abstrakten Datentypen, da nach außen hin nur die **Nachrichtendefinitionen**, die von einem Objekt verarbeitet werden, bekannt sind, und nicht die internen Datenobjekte und Operatoren [FeSi01, 301 f].

Objekttyp

Ein weiterer Unterschied zum abstrakten Datentyp ist die deutliche Differenzierung zwischen einem Objekt als Exemplar und einem Objekttyp als dessen Spezifikation [FeSi01, 299]. Der **Objekttyp** spezifiziert zumindest die internen Datenobjekttypen, **Attribute** genannt, und die Implementierung der Operatoren, **Methoden** genannt (vgl. Abbildung 3.13). Allerdings bleiben diese Spezifikationen nach außen hin verborgen und es werden nur die Operatorenschnittstellen, auch als **Methodensignaturen** bezeichnet [GHJV96, 18], und ein Name für den Objekttyp bekanntgegeben.

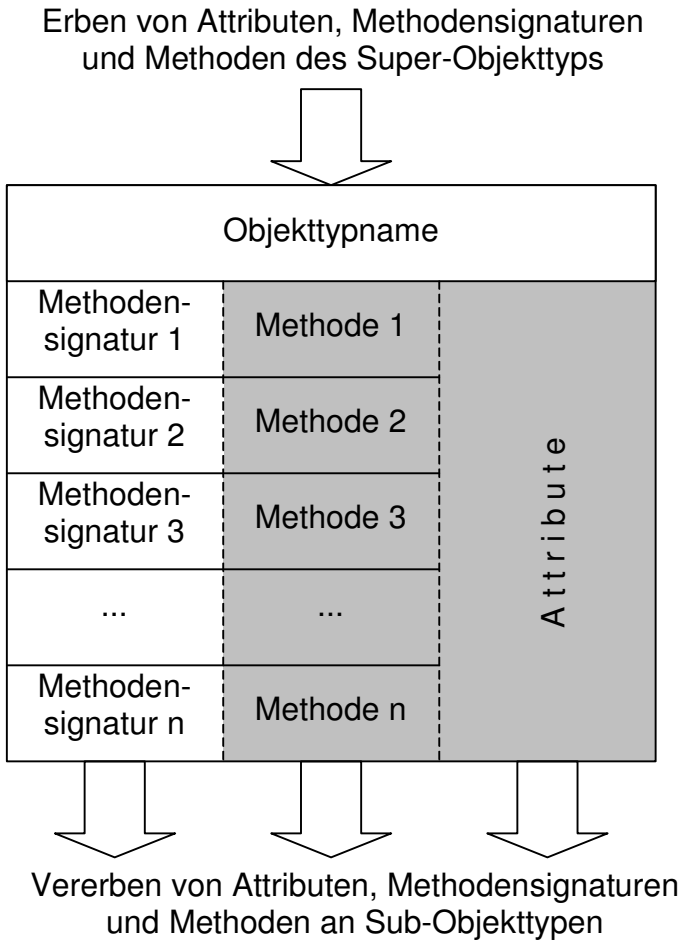


Abbildung 3.13: Objekttyp

Somit ist ein Objekttyp, analog zum abstrakten Datentyp, allein durch den Namen und durch die Methodensignaturen definiert. Die Methodensignaturen bilden dabei

die Schnittstelle des Objekttyps. Von einem Objekttyp können beliebig viele Objekte, auch **Instanzen** genannt, erzeugt werden.

Klasse

Umgekehrt werden Objekte, die eine gemeinsame Struktur und ein gemeinsames Verhalten aufweisen, in einer so genannten **Klasse** zusammengefasst [Booc94, 103]. Eine Klasse umfasst sowohl die Objekte als auch deren Spezifikation in Form des Objekttyps. Falls zu einer Klasse kein konkreter Objekttyp spezifiziert ist, handelt es sich um eine nicht-typisierte Klasse. In der Literatur zu objektorientierten Ansätzen und Programmiersprachen wird üblicherweise nicht zwischen den Begriffen „Klasse“ und „Objekttyp“ unterschieden, sondern in beiden Fällen der Begriff „Klasse“ verwendet. Zur Beibehaltung der Originalterminologie und zur Verbesserung der Lesbarkeit liegt dieses Begriffsverständnis auch den folgenden Kapiteln zugrunde. Der Begriff „Objekttyp“ wird somit nur noch im Zusammenhang mit der SOM-Methodik verwendet.

Für die Verwaltungsfunktionen einer Klasse zeichnet sich ein singuläres Objekt der Klasse verantwortlich, die so genannte **Klasseninstanz**. Sie ist hauptsächlich für die Erzeugung neuer Objekte zuständig [FeSi01, 299]. Im Unterschied zu den beschriebenen konkreten Klassen können auch so genannte **abstrakte Klassen** definiert werden. Sie sind in erster Linie dadurch charakterisiert, dass sie keine Objekte besitzen und auch keine Objekte erzeugen können [Booc94, 511]. Deswegen können Teile einer abstrakten Klasse auch unspezifiziert bleiben, wie z. B. die Implementierung einiger oder aller Methoden. Eine abstrakte Klasse wird durch eine oder mehrere spezialisierte Unterklassen konkretisiert [Booc94, 511]. Dabei spielt die Implementierungsvererbung eine wesentliche Rolle. Die Spezialisierung und die Vererbung von Klassen sind unter anderem Themen der folgenden Ausführungen.

Das Klassenkonzept des Grundkonzepts der Objektorientierung vermischt allerdings zwei von WEGNER identifizierte orthogonale Konzepte der Objektbasierung [Wegn87]. Eine Klasse enthält zum einen die Implementierung von Objekten der Klasse und definiert zum anderen ihren Typ in Form des Namens und der Schnittstelle [JäHe02, 268]. Demzufolge wird bei der Programmierung einer Klasse in einer objektorientierten Programmiersprache auch eine konkrete Implementierung ihrer Schnittstelle festgelegt. Wie bereits dargelegt, enthält die Schnittstelle selbst nur die Methodensignaturen der Klasse. Sämtliche Zusatzinformationen z. B. über das Verhalten der Objekte zur Laufzeit müssen aus der mitgegebenen Implementierung der Klasse herausgelesen werden [Szyp02, 2]. Insofern ist die mitgegebene Implementierung die einzige vollständige Dokumentation einer programmierten Klasse und deswegen ist eine Unabhängigkeit des aufrufenden Objekts von der Implementierung des aufgerufenen Objekts nicht mehr gewährleistet [GHJV96, 25]. Mit anderen Worten, das Ge-

heimnisprinzip wird im Klassenkonzept nicht konsequent genug umgesetzt, da die konzeptionelle Trennung von Schnittstelle und Implementierung darin faktisch wieder aufgehoben wird. Dies wird aber erst bei der Programmierung einer Klasse in einer objektorientierten Programmiersprache deutlich. Bei einer konsequenten Umsetzung des Prinzips der Trennung von Schnittstelle und Implementierung könnten Objekte eines bestimmten Typs, der durch die Schnittstelle und den Namen beschrieben wäre, unterschiedliche Implementierungen aufweisen, die gegebenenfalls auch zur Laufzeit gegenseitig ersetzbar wären [Szyp98, 72]. Dies würde die Anpassbarkeit und Erweiterbarkeit und damit die Wiederverwendbarkeit der Objekte dieses Typs erhöhen. Allerdings müsste dafür die Schnittstelle möglichst vollständig, genau und eindeutig spezifiziert sein [Meye90, 121 ff; Szyp98, 73]. Aufgrund der Zusammenführung von Schnittstelle und Implementierung im Klassenkonzept ist die Ersetzbarkeit der Implementierung nur durch die Anwendung der noch zu erläuternden Schnittstellen- und Implementierungsvererbung erreichbar. Eine solche Wiederverwendung einer Klasse in der Implementierung entspricht jedoch, wie noch gezeigt wird, einer White-Box-Nutzung [Szyp98, 33 f; GHVJ96, 26; Pers03, 2]. Dies führt zu Problemen bei der Wiederverwendung von Klassen und Klassenbibliotheken und somit zu einer Verringerung ihrer Wiederverwendbarkeit.

Im Grundkonzept der Objektorientierung werden alle in Abschnitt 3.4.1.1 genannten Abstraktionsformen zur Verfügung gestellt. Die Konzeptbildung bzw. Typisierung und die Klassifizierung sind durch das Klassenkonzept realisiert. Ein charakterisierender Bestandteil des objektorientierten Konzepts ist die Realisierung der Abstraktionsformen Generalisierung und Aggregation.

Untertypen- und Unterklassenbildung

Die Spezialisierung als Umkehrung der Generalisierung wird durch zwei voneinander unabhängige Formen gewährleistet: Die Untertypenbildung (*Subtyping*) und die Unterklassenbildung (*Subclassing*) [Szyp98, 97; Pers03, 2]. Durch die **Untertypenbildung** entsteht ein neuer Typ, der die Methodensignaturen des übergeordneten Typs und üblicherweise noch zusätzliche aufweist [GHVJ96, 18]. Analog dazu entsteht durch die **Unterklassenbildung** eine neue Klasse, die die Methodensignaturen und die zugehörigen Implementierungen der übergeordneten Klasse und im Normalfall weitere Methodensignaturen, -implementierungen und Attribute aufweist (vgl. Abbildung 3.14) [FeSi01, 300]. Da Beziehungen zu Objekten anderer Klassen in der Implementierung einer Klasse als Objektreferenzen enthalten sind, werden auch diese in die Unterklasse übernommen. Eine Besonderheit der Unterklassenbildung ist, dass die Methodenimplementierungen der übergeordneten Klasse (auch **Oberklasse** oder **Superklasse** genannt) in der **Unterklasse** (auch **Subklasse** genannt) überschrieben, d. h. ersetzt werden können [GHVJ96, 22; RBP+93, 78]. Im Unterschied

zum **Überschreiben** bezeichnet man mit **Überladen** eine Änderung der Art und Anzahl der Parameter einer Methodensignatur der Oberklasse bzw. des übergeordneten Typs in der Unterklasse bzw. im Untertyp [Booc94, 115 f]. Die ursprüngliche Methodensignatur wird in der Unterklasse bzw. im Untertyp dadurch aber nicht ersetzt, sondern die Schnittstelle des Untertyps bzw. der Unterklasse wird durch die neue Methodensignatur erweitert. Das Überladen kann bei der Untertypenbildung und bei der Unterklassenbildung auftreten, wobei der übergeordnete Typ bzw. die übergeordnete Klasse unverändert bleibt. Dem gegenüber kann das Überschreiben nur bei der Unterklassenbildung durchgeführt werden, wobei die betreffende Methode der Oberklasse in der Unterklasse verändert wird.

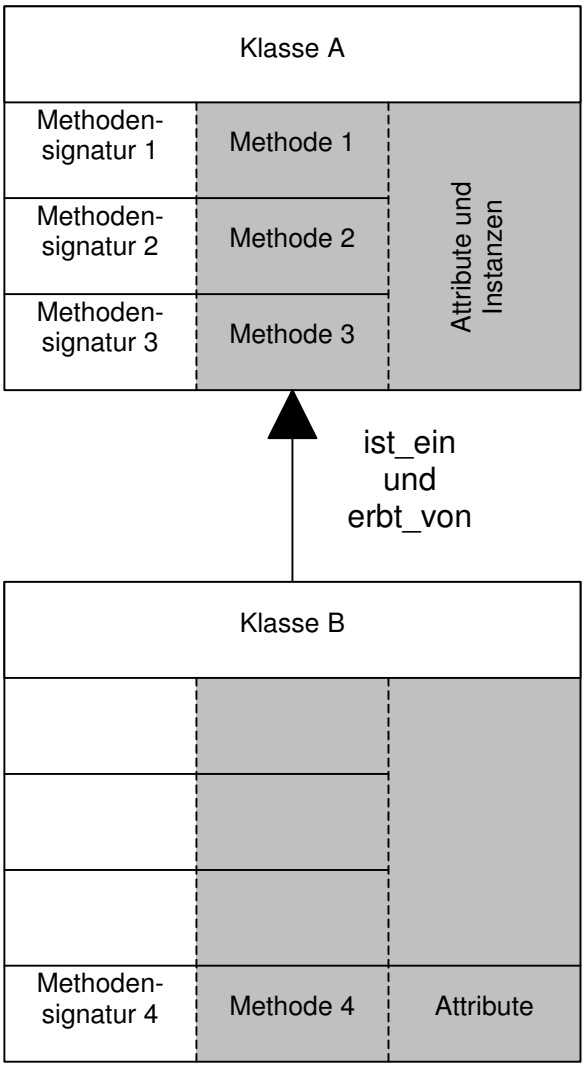


Abbildung 3.14: Unterklassenbildung/Vererbung

Vererbung

Die Untertypenbildung und die Unterklassenbildung wird im Grundkonzept der Objektorientierung durch die so genannte **Vererbung** realisiert (vgl. Abbildung 3.14). An diesem Mechanismus wird der in der Programmierung liegende Ursprung des ob-

jektorientierten Konzepts deutlich: Durch die Vererbung müssen nur die Unterschiede (*delta changes* [Pree97, 4]) zum übergeordneten Typ bzw. zur übergeordneten Klasse im Untertyp bzw. in der Unterklasse programmiert werden. Die unveränderten Methodensignaturen des übergeordneten Typs bzw. die unveränderten Methoden und Attribute der übergeordneten Klasse werden scheinbar automatisch in den spezialisierten Typ bzw. in die spezialisierte Klasse „vererbt", d. h. übernommen [Pree97, 4 f]. Die Vererbung ermöglicht in der Implementierung eine **Programmierung durch Unterscheidung**, bei der unveränderter Programmcode nicht wiederholt reimplementiert werden muss, sondern durch einen geeigneten Mechanismus wiederverwendet werden kann [JoFo88]. D. h. der Vererbungsmechanismus und alle damit zusammenhängenden Techniken und Konzepte wurden ursprünglich für die Code-Wiederverwendung entworfen [Will02, 7; Grif98, 28; BrSi02, 7] und fanden auf diesem Weg Eingang in das Grundkonzept der Objektorientierung und damit auch in die objektorientierten Analyse- und Entwurfsansätze. Gemäß der Spezialisierungsform, die durch die Vererbung unterstützt wird, kann man unterscheiden zwischen der **Schnittstellenvererbung** und der **Implementierungsvererbung** [Szyp98, 97; GHJV96, 23; Pers03, 2]. Die Implementierungsvererbung wird als White-Box-Wiederverwendung bezeichnet, da ein Zugang zur Implementierung für die Wiederverwendung notwendig ist [Pers03, 2]. Deswegen spicht man auch davon, dass die Vererbung die Kapselung bzw. das Geheimnisprinzip verletzt [BrSi02, 4; Szyp02, 1; PfSz02, 5; Pers03, 2; Will02, 4].

Die Implementierungsvererbung ist ein charakteristischer Bestandteil des Grundkonzepts der Objektorientierung. Selbst die Spezifikation einer neuen Klasse erfolgt nicht eigenständig wie die eines neuen abstrakten Datentyps, sondern durch Spezialisierung einer übergeordneten Klasse mittels Implementierungsvererbung [GHJV96, 21]. Ein Ansatz oder eine Programmiersprache, der bzw. die zum größten Teil das Grundkonzept der Objektorientierung realisiert, aber nicht über den Mechanismus der Implementierungsvererbung verfügt, wird nicht objektorientiert, sondern objektbasiert genannt [Booc94, 516].

Polymorphie

Ebenfalls charakteristisch für das Grundkonzept der Objektorientierung ist das mit der Vererbung zusammenhängende Merkmal der Polymorphie. Eine Unterklasse, die durch Implementierungsvererbung von einer Oberklasse abgeleitet wurde, besitzt auch einen neuen Typ, der verschieden von dem der Oberklasse ist. Das liegt zum einen daran, dass bei der Spezialisierung durch Implementierungsvererbung üblicherweise die Schnittstelle durch neue Methodensignaturen erweitert wird und dadurch ein neuer Typ entsteht. Zum anderen muss die Unterklasse einen anderen Namen als die Oberklasse besitzen und dies führt zwangsläufig zu einem anderen

Typ der Klasse. Da durch die Implementierungsvererbung eine Unterklasse nie unabhängig von ihrer Oberklasse sein kann, sondern die Oberklasse faktisch ein Teil der Unterklasse ist, sind Objekte, die aus der Unterklasse erzeugt wurden, sowohl vom Typ der Unterklasse als auch vom Typ der Oberklasse und allen darüber liegenden Klassen. Diese Teil-Ganzes-Beziehung zwischen Oberklasse und Unterklasse bezieht sich jedoch nur auf die Typen der Klassen, nicht aber auf die Menge ihrer Instanzen. Somit besitzt ein Objekt einer Unterklasse nicht nur einen Typ, sondern es kann mehrere Typen aufweisen. Dies wird nach CARDELLI/WEGNER [CaWe85, 475] mit **Polymorphie** bezeichnet. Häufig beschreibt man jedoch das Merkmal der Polymorphie anhand seines Effekts in objektorientierten Programmen: Falls die Implementierung einer geerbten Methode in einer Unterklasse überschrieben wird, kann ein Methodenaufruf in Form einer Nachricht an ein Objekt der Unterklasse zu einem anderen Ergebnis führen, als derselbe Methodenaufruf an ein Objekt der Oberklasse (siehe z. B. [Booc94, 517; RBP+93, 2]). Folglich ist die Auswahl der Methodenimplementierung zu einer bestimmten Nachricht vom Typ des empfangenden Objekts und damit vom Programmablauf abhängig. Falls das empfangende Objekt keine eigene Implementierung der aufgerufenen Methode besitzt, leitet es die Nachricht entlang der Hierarchie bis zur ersten gefundenen Implementierung nach oben weiter. Somit wird in einem objektorientierten Programm die Auswahl der Methodenimplementierung erst zur Laufzeit des Programms entschieden, was auch als **spätes bzw. dynamisches Binden** bezeichnet wird (vgl. z. B. [Booc94, 71; GHJV96, 19; RBP+93, 387].

Wie bereits erwähnt, lassen sich durch die Implementierungsvererbung auch verschiedene Implementierungen zu einer bestimmten Schnittstelle ersetzen. Dafür wird eine Unterklasse nicht um zusätzliche Methodensignaturen, -implementierungen und Attribute erweitert, sondern es werden darin nur die zu ersetzenden Methodenimplementierungen überschrieben. Trotzdem besitzt eine solche Unterklasse einen anderen Typ als die Oberklasse, da sich die Namen der beiden Klassen unterscheiden. Folglich sind Objekte einer solchen Unterklasse ebenfalls polymorph.

Einfach- und Mehrfachvererbung

Falls eine Unterklasse nur von einer Oberklasse Attribute, Nachrichtendefinitionen und Operatoren erbt, spricht man von einer **Einfachvererbung**. Existieren mehrere Oberklassen, von denen einen Unterklasse erben kann, liegt eine **Mehrfachvererbung** vor [Booc94, 123 ff]. Eine Einfachvererbung lässt sich anhand einer Baumstruktur beschreiben, eine Mehrfachvererbung anhand eines quasi-hierarchischen Graphen [FeSi01, 300 f].

Aggregation

Zur Abstraktionsform Aggregation existieren im Grundkonzept der Objektorientierung mehrere Formen. Sie können anhand von drei Merkmalen unterschieden werden [MaOd99, 255]:

- Das Merkmal **Konfiguration** bezieht sich auf die funktionellen oder strukturellen Beziehungen zwischen dem Aggregat und seinen Teilen.
- Von dem Merkmal **Gleichartigkeit** wird gesprochen, wenn die Teile das selbe darstellen wie das Aggregat.
- Das Merkmal **Unveränderbarkeit** drückt aus, wenn sich die Teile nicht vom Aggregat trennen lassen, ohne dass dieses zerstört wird.

Einen Überblick über die daraus folgenden Formen der Aggregation enthält [MaOd99, 254 ff].

Objektkomposition und Delegation

Die genannten Aggregationsformen setzen Objekte der beteiligten Klassen in Beziehung. Auf Objektebene werden diese Aggregationsformen durch die Objektkomposition realisiert. Diese nimmt darüber hinaus eine Sonderstellung im Grundkonzept der Objektorientierung ein. Zusammen mit dem Konzept der Delegation, das nach WEGNER [Wegn87] zu den orthogonalen Konzepten der Objektbasierung zählt, lassen sich damit alle Beziehungsarten zwischen Objekten und somit auch zwischen Klassen erklären. Bei der **Objektkomposition** umschließt ein zusammengesetztes Objekt konzeptionell seine Teilobjekte (vgl. Abbildung 3.15). Deswegen wird das zusammengesetzte Objekt auch äußeres Objekt und die Teilobjekte auch innere Objekte genannt [Szyp98, 117]. Nachrichten, die entweder an das zusammengesetzte Objekt oder an die Teilobjekte gerichtet sind, werden zunächst vom zusammengesetzten Objekt empfangen. Falls dieses keine passende Methode für die empfangene Nachricht besitzt oder die passende Methode den Aufruf nicht vollständig abarbeiten kann, sendet es die Nachricht nach Bedarf an das Teilobjekt bzw. die Teilobjekte weiter. Dies wird mit **Delegation** bezeichnet [RBP+93, 298; Szyp98, 117]. Dabei kann zwischen der Delegation i. e. S. und dem Weiterleiten (*Forwarding*) unterschieden werden [Szyp98, 117]. Bei der **Delegation i. e. S.** behält das äußere Objekt nach dem Weitersenden der Nachricht die Kontrolle über die vollständige Abarbeitung der aufgerufenen Methode. Im Falle des **Weiterleitens** gibt das äußere Objekt die Kontrolle nach dem Weitersenden der Nachricht bis zur vollständigen Abarbeitung der Methode an das innere Objekt ab. [Szyp98, 117 ff].

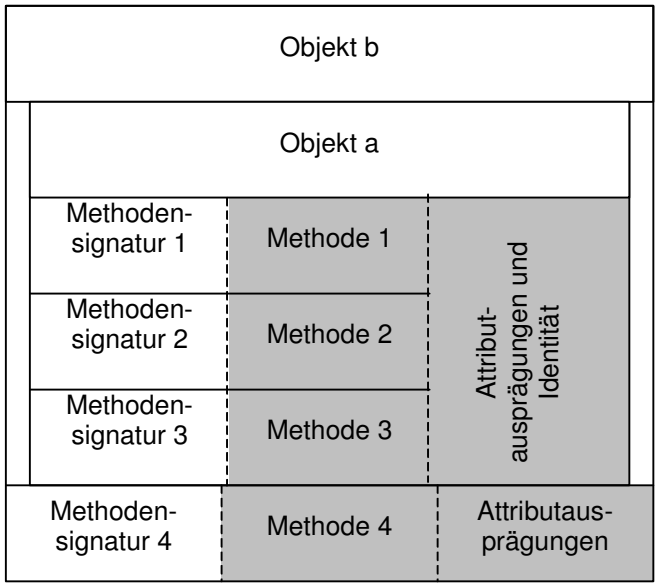


Abbildung 3.15: Objektkomposition

Anhand der Objektkomposition und der Delegation kann beispielsweise auch die Implementierungsvererbung erklärt werden [RBP+93, 297 f; Szyp98, 117 ff; Will02, 5 ff]. Dabei entspricht die Unterklasse der Klasse, aus dem das äußere Objekt erzeugt wurde, und die Oberklasse der Klasse, aus dem das innere Objekt stammt. Wenn eine Methode am Objekt der Unterklasse aufgerufen wird und sich die Implementierung der aufgerufenen Methode vollständig oder zum Teil in der Oberklasse befindet, leitet das Objekt der Unterklasse den Aufruf, analog zur Delegation, an das Objekt der Oberklasse implizit weiter. Dabei behält es jedoch die Kontrolle über die Abarbeitung der Methode, d. h. es handelt sich hierbei um eine Delegation i. e. S. Nun wird jedoch bei der Instantiierung eines Objekts einer Unterklasse nur ein einziges Objekt erzeugt, das sowohl die Rolle des Objekts der Unterklasse als auch die Rolle des Objekts der Oberklasse und aller darüberliegenden Klassen einnimmt (***Possession of a common self***) [Szyp98, 117; Pers03, 2]. Folglich findet die erläuterte Delegation i. e. S. zwischen dem Objekt der Unterklasse und dem Objekt der Oberklasse im selben Objekt statt, d. h. es erfolgt ein **impliziter Selbstaufruf** [Szyp98, 117; Pers03, 2]. Dies führt zu einem weiteren Problem der Implementierungsvererbung: Aufgrund des Selbstaufrufs durch implizite Delegation i. e. S. existiert eine enge Bindung zwischen der Unterklasse und der Oberklasse. Das kann dazu führen, dass eine Änderung oder Weiterentwicklung der Implementierung der Oberklasse, auch wenn sie verhaltensbewahrend ist, die Funktionstüchtigkeit der Unterklasse beeinträchtigt bzw. zerstört [Szyp98, 103; PfSz02, 4; Pers03, 4]. Dieser Effekt wird auch ***semantic base class problem*** genannt [Szyp98, 103; MiSe97; PfSz02, 4; Pers03, 4]. „[..] The bottom line is that whenever a base class is revised, any (unmodified) code taking advantage of the base class through subclassing may unexpectedly start to malfunc-

tion, although care has been taken to ensure that the modifications in the base class will be entirely behaviour-preserving“ [Pers03, 4].

Assoziation

Neben den genannten Abstraktionsformen wird im Grundkonzept der Objektorientierung auch die Assoziation als frei definierbare, nicht-hierarchische Beziehung zwischen Objekten der beteiligten Klassen realisiert. Für jede Assoziation kann angegeben werden, ob sie **unidirektional** oder **bidirektional** sein soll [Booc94, 108 f; RBP+93, 299 f]. Außerdem kann jede Assoziation mit Kardinalitäten versehen werden. Sie bestimmen die Minimal- und Maximalanzahl der Objekte, die durch diese Assoziation in Beziehung stehen sollen bzw. dürfen [Booc94, 109; MaOd99, 95]. Schließlich ist es, wie erwähnt, möglich, Assoziationen selbst als Typ bzw. Klasse zu beschreiben [MaOd99, 96 ff]. Dabei stellen die Instanzen dieser Klasse unveränderbare Verknüpfungen von Objekten der durch die Assoziation verbundenen Klassen dar [MaOd99, 105].

Objektgranularität

Ein zusätzliches Hindernis für die Wiederverwendbarkeit von Klassen bzw. Objekten ist deren feine Granularität [Same97, 33; WiJo90, 116]. Ursprünglich sollte das Grundkonzept der Objektorientierung Konstrukte zur Verfügung stellen, die Objekte der realen Welt abbilden können [JäHe02, 267; Booc94, 17; MaOd99, 25]. Deswegen wurden diese Konstrukte auch Objekte genannt. Von diesem Denkansatz versprach man sich in der Anwendungssystementwicklung einen fließenden, lückenlosen Übergang von der Aufgabe der fachlichen Analyse zur Aufgabe des softwaretechnischen Entwurfs [Booc94, 39 f; RBP+93, 4 f; MaOd99, 24 f]. In der Praxis erfüllten sich diese Versprechungen bis heute jedoch kaum. Das liegt hauptsächlich daran, dass das Grundkonzept der Objektorientierung, wie erwähnt, über die Programmierung in die Anwendungssystementwicklung eingebracht wurde. So wird ein Objekt im Grundkonzept der Objektorientierung immer noch eher als Implementierung eines abstrakten Datentyps verstanden (vgl. z. B. [Szyp98, 32; WeSa01, 36; Meye90, 64; MaOd99, 447; MaOd92, 451]) anstatt als Abbildung eines Objekts der realen Welt. Daraus folgt, dass zumindest im softwaretechnischen Entwurf Klassen hinsichtlich einer einfachen Umsetzbarkeit in einer bestimmten objektorientierten Programmiersprache spezifiziert werden und nicht hinsichtlich einer Kapselung fachlicher Belange [DeMe01, 1; Geig02, 1]. Dies reduziert sie jedoch auf das Wesen eines programmiersprachlichen Elements, eben einer Implementierung eines abstrakten Datentyps [BrSi02, 7]. Für den Entwurf programmiersprachlicher Elemente gelten allerdings andere Ziele als für den Entwurf fachlicher Objekte. Beispielsweise schreibt [Grif98, 32]: „Programmiersprachliche Objekte sind meist klein genug, um in einem einzigen überschaubaren Arbeitsvorgang erzeugt werden zu können – ansonsten

gelten sie oft als ‚schlecht' entworfen". Deswegen sind solche Klassen in der Regel zu klein, um fachliche Belange eigenständig abbilden zu können [DeMe01, 1; Geig02, 2; Grif98, 32]. Zur Abbildung eines fachlichen Belangs müssen üblicherweise Objekte mehrerer Klassen zusammenarbeiten [DeMe01, 1; Geig02, 1; Grif98, 32]. Dies verletzt allerdings das Prinzip der Trennung von Belangen. Darüber hinaus werden Klassen zur Abbildung eines fachlichen Belangs meistens durch Implementierungsvererbung miteinander verknüpft. Daraus ergeben sich tiefe und weit verzweigte Klassenhierarchien, in denen die untersten Klassen zwar fachliche Belange abbilden können, jedoch die dafür notwendige Funktionalität von mehreren Klassen in der Hierarchie erben. Dies wirft mehrere Probleme auf: Zum einen sind die Klassen, die gegebenenfalls fachliche Belange abbilden könnten, nicht eigenständig wiederverwendbar, da die fachliche Funktionalität auf mehrere Klassen in der Klassenhierarchie „verstreut" ist [DeMe01, 1; Geig02, 2; ElFB01, 30]. In diesem Zusammenhang wird auch noch einmal deutlich, dass die Implementierungsvererbung die Realisierung des Geheimnisprinzips verletzt und damit die Wiederverwendbarkeit einzelner Klassen beeinträchtigt. Zum anderen wird insbesondere die Anpassbarkeit und Erweiterbarkeit einzelner Klassen durch weit verzweigte und tiefe Vererbungshierarchien erschwert. Dieses Problem potenziert sich bei einer Mehrfachvererbung.

Ein Klassenentwurf, der sich primär an programmiertechnischen Aspekten orientiert, schränkt überdies die Verteilbarkeit von Klassen ein. Da Objekte solcher Klassen zur Abbildung eines fachlichen Belangs einen erhöhten Kommunikationsbedarf haben, sollten sie nicht auf mehreren Rechnersystemen verteilt werden, sondern sich nur auf einem Rechnersystem befinden [PfSz02, 4; Szyp98, 130 f].

Zusammenfassend lässt sich feststellen, dass Klassen in derzeitigen Software-Entwürfen und Implementierungen aufgrund einer hauptsächlich programmiertechnischen Abgrenzung nicht für sich allein wiederverwendbar und verteilbar sind. Außerdem wird durch eine eher programmiertechnische Abgrenzung von Klassen im softwaretechnischen Entwurf der fließende, lückenlose Übergang zwischen der fachlichen Analyse und dem softwaretechnischen Entwurf, den man sich durch den Einsatz der Objektorientierung versprochen hatte, wieder aufgehoben.

3.5.1.2 Bewertung

Ziel des Abschnitts 3.5 ist die Bewertung der Wiederverwendungs-, Wiederverwendbarkeits- und Verteilbarkeitsunterstützung ausgewählter Erzeugnisarten, die im softwaretechnischen Entwurf und in der Implementierung verwendet werden können.

Klassen sind sowohl in objektorientierten Entwürfen als auch in Klassenbibliotheken enthalten. Allerdings handelt es sich bei einer Klasse in einem objektorientierten Entwurf und einer Klasse in einer Klassenbibliothek nicht um dasselbe Entwicklungs-

erzeugnis, wie es beispielsweise bei einem Framework der Fall ist. Deshalb muss für die Bewertung der Erzeugnisart Klasse zwischen dem softwaretechnischen Entwurf und der Implementierung differenziert werden. Die resultierende Wiederverwendungs-, Wiederverwendbarkeits-, und Verteilbarkeitsunterstützung der Erzeugnisart Klasse bestimmt schließlich auch die entsprechenden Unterstützung der Erzeugnisarten objektorientierter Entwurf und Klassenbibliothek. Die Bewertung erfolgt auf Basis des beschriebenen Grundkonzepts der Objektorientierung und dessen Realisierungen in Form von objektorientierten Entwurfsansätzen bzw. Programmiersprachen (vgl. Tabelle 3.9).

Forderung	**Unterstützung (Entwurf)**	**Unterstützung (Implemen.)**
Wiederverwendung		
Konzepte	O	O
Artefakte	+	+
bereichsübergreifend	O	O
planbar	+	+
kompositionell	+	+
generativ	O	O
Black-Box	-	-
Wiederverwendbarkeit		
hohes Abstraktionsniveau	O	-
Allgemeingültigkeit	+	+
hierarchische Strukturierung	+	+
Modularität	O	O
Verständlichkeit	+	-
Nachvollziehbarkeit	+	-
standardisierte und formalisierte Beschreibung	+	O
Anpassbarkeit	+	+
Erweiterbarkeit	+	+
Interoperabilität	n. a.	O

Forderung	Unterstützung (Entwurf)	Unterstützung (Implemen.)
Portierbarkeit	n. a.	O
Qualität	-	O
Verteilbarkeit		
Kontrolle der Daten- und Funktionsredundanz	+	+
Kontrolle der Verknüpfungen	+	+
Erhaltung der semantischen und operationalen Integrität	+	+
Zielbezogenheit	+	+
Verteilungstransparenz	-	+

+ = vollständig O = teilweise - = keine
n. a. = nicht anwendbar

Tabelle 3.9: Bewertung der Erzeugnisart Klasse

Klassen und Objekte stellen Artefakte dar. Für objektorientierte Entwurfsansätze und objektorientierte Programmiersprachen stehen inzwischen umfangreiche Sammlungen von wiederverwendbaren Konzepten in Form von Entwurfs- bzw. Implementierungsmuster zur Verfügung (siehe Abschnitt 3.1.3.6). Allerdings sind diese Muster noch keine integralen Bestandteile der Entwurfsansätze bzw. Programmiersprachen.

Klassen können grundsätzlich domänenunabhängig, aber auch domänenspezifisch wiederverwendet werden. Genauso ist eine Wiederverwendung von Klassen innerhalb eines Anwendungssystems ohne Schwierigkeiten möglich. Lediglich die Wiederverwendung in mehreren Anwendungssystemen ist aus genannten Gründen problematisch.

Eine Wiederverwendung von Klassen kann grundsätzlich geplant werden. Zur Wiederverwendung wird die kompositionelle Technik eingesetzt. Durch die zusätzliche Verwendung der bereits erwähnten Entwurfs- und Implementierungsmuster ist auch eine Kombination mit der generativen Technik möglich. Allerdings lässt das Grundkonzept der Objektorientierung, wie gezeigt, keine Black-Box-Wiederverwendung von Klassen im softwaretechnischen Entwurf und in der Implementierung zu, sondern nur eine White-Box-Wiederverwendung.

Im softwaretechnischen Entwurf können Klassen auf einem hohen Abstraktionsniveau beschrieben werden. Dagegen werden sie in der Implementierung nur anhand ihrer konkreten Realisierung spezifiziert. Außerdem ist es grundsätzlich möglich, Klassen allgemeingültig, d. h. domänenunabhängig zu konzipieren.

Durch die Umsetzung der Abstraktionsformen Generalisierung/Spezialisierung und Aggregation/Zerlegung im Grundkonzept der Objektorientierung wird die hierarchische Strukturierung von Klassen in beiden Phasen unterstützt. Die Modularität von Klassen wird, wie oben beschrieben, im Grundkonzept der Objektorientierung nicht konsequent genug umgesetzt. Das gilt folglich sowohl für Klassen im softwaretechnischen Entwurf als auch in der Implementierung.

Eine intuitiv verständliche Beschreibbarkeit des fachlichen Inhalts einer Klasse wird im softwaretechnischen Entwurf durch geeignete Diagrammelemente in den standardisierten Beschreibungssprachen ausreichend unterstützt. Dagegen kann eine Klasse in der Implementierung ihren fachlichen Inhalt nicht wiedergeben. Zur geforderten Nachvollziehbarkeit von Klassen ist eine ausreichende Selbstdokumentation notwendig. Zu diesem Zweck stehen in einigen objektorientierten Programmiersprachen so genannte ***Reflection*-Mechanismen** zur Verfügung [Flan00, 160 f; Barn02, 72 ff]. Diese Mechanismen bieten jedoch keine ausreichende Selbstdokumentation im Sinne der Ausführungen in Abschnitt 3.1.4.3 an. Im softwaretechnischen Entwurf existieren im Rahmen der standardisierten Beschreibungssprachen genügend Möglichkeiten zur ausreichenden Selbstdokumentation von Klassen.

Eine standardisierte und formalisierte Beschreibung von Klassen ist im softwaretechnischen Entwurf durch den von der OMG (Object Management Group) angenommenen Notationsstandard **Unified Modeling Language (UML)** möglich. Abschnitt 5.1 widmet sich ausführlich der UML. In der Implementierung existieren zur Zeit keine Bestrebungen zur Standardisierung von objektorientierten Programmiersprachen. Allerdings ist eine Beschreibung von Klassen in einer Programmiersprache genügend formalisiert.

Die Anpassbarkeit und die Erweiterbarkeit von Klassen im softwaretechnischen Entwurf und in der Implementierung wird durch die Implementierungsvererbung und die Objektkomposition mit expliziter Delegation unterstützt. Beide Techniken sind im Grundkonzept der Objektorientierung verankert.

Wie in Abschnitt 3.1.4.2 festgestellt, beziehen sich die im Folgenden genannten Forderungen Interoperabiliät und Portierbarkeit ausschließlich auf Entwicklungserzeugnisse in der Implementierung. Sie bedürfen keiner Spezifikation im softwaretechnischen Entwurf. Zur Gewährleistung der Interoperabilität zwischen Objekten, die aus verschiedenen Klassen erzeugt wurden, stehen unterschiedliche Kommunikationssysteme als so genannte **Middleware-Systeme** zur Verfügung. Abschnitt 5.3.6.2

untersucht unter anderem den Begriff „Middleware-System“ und die derzeit meist verwendeten Kommunikationssysteme. Als problematisch erweist sich gegenwärtig, dass die verschiedenen Kommunikationssysteme untereinander nicht interoperabel sind und folglich Objekte über die Grenzen eines Kommunikationssystems hinaus auch nicht interoperabel sein können. Diesem Problem versucht man gegenwärtig mit der Web-Service-Initiative zu begegnen, die ebenfalls in Abschnitt 5.3.6 behandelt wird.

Für die Portierbarkeit von Klassen sorgen häufig so genannte ***Virtual Machines*** (siehe dazu z. B. [Flan00, 4; Barn02, 30]), die, wie objektorientierte Programmiersprachen, von Entwicklungs- und Ausführungsplattformen für objektorientierte Anwendungssysteme zur Verfügung gestellt werden.

Zur Untersuchung der Forderung Qualität muss unterschieden werden zwischen der Realisierung von Qualitätskriterien in der Implementierung und deren Spezifikation und Dokumentation im softwaretechnischen Entwurf. Zur Realisierung von Qualitätskriterien einer Klasse können zum Teil ebenfalls Entwicklungs- und Ausführungsplattformen für objektorientierte Anwendungssysteme oder spezielle Middleware-Systeme verwendet werden. Die Spezifikation und Dokumentation der Qualitätskriterien im softwaretechnischen Entwurf wird derzeit nicht unterstützt.

Die letztgenannten Forderungen in der Tabelle beziehen sich auf das Formalziel Verteilbarkeit eines Anwendungssystems. Dabei resultieren die ersten vier aus der Notwendigkeit zur Integration der wiederverwendbaren und verteilbaren Entwicklungserzeugnisse im Anwendungssystem. Da die Verwendung von Klassen und daraus erzeugten Objekten offensichtlich das in Abschnitt 3.2.3.2 erläuterte Integrationskonzept Objektintegration realisiert, sind die Bewertungen der Forderungen bezüglich der Verteilbarkeit mit den entsprechenden Zielerreichungsgraden dieses Integrationskonzepts identisch. Dieses Ergebnis gilt selbstverständlich sowohl für Klassen im softwaretechnischen Entwurf als auch für Klassen in der Implementierung.

Die Verteilungstransparenz als letztgenannte Forderung an ein verteilbares Anwendungssystem fasst die Forderungen nach den in Abschnitt 3.2.4 genannten Teiltransparenzen zusammen. Anhand der Verteilungstransparenz soll insbesondere die Realisierung der Qualitätskriterien und der zugehörigen Ausprägungen, die ein verteiltes Anwendungssysteme aus Nutzersicht erfüllen soll, vor dem Nutzer verborgen werden. Diese Qualitätskriterien und -ausprägungen müssen, soweit sie quantifizierbar sind, zunächst im softwaretechnischen Entwurf spezifiziert werden. Dies wird derzeit nicht unterstützt. Für die transparente Realisierung der Qualitätskriterien und -ausprägungen in der Implementierung sind, wie bereits erwähnt, spezielle Basismaschinen zuständig. Diese stehen in Form verschiedener Middleware-Systeme, die in Abschnitt 5.3.6.2 erläutert werden, zunehmend zur Verfügung.

3.5.2 Erzeugnisart Framework

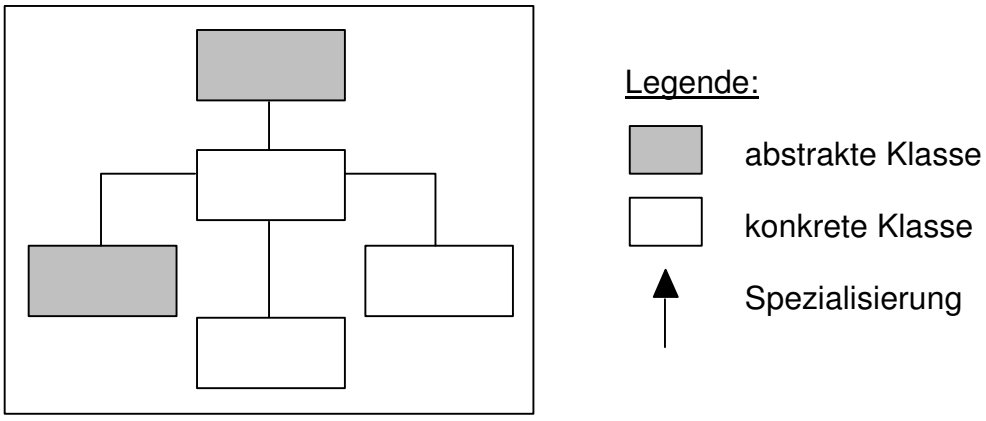
Die Erzeugnisart Framework wurde bereits kurz in Abschnitt 3.1.3.6 vorgestellt. Frameworks stehen in einer engen Beziehung zu Klassenbibliotheken: „Frameworks unterscheiden sich von gewöhnlichen Klassenbibliotheken dadurch, dass Frameworks zusätzlich die Architektur vorgeben“ [Pree97, 19]. Die Architektur stellt einen softwaretechnischen Entwurf für eine kontextabhängige Problemstellung dar, indem sie „die Struktur im Großen, [die] Unterteilung in Klassen und Objekte, die jeweiligen zentralen Zuständigkeiten, die Zusammenarbeit der Klassen und Objekte sowie den Kontrollfluß“ definiert [GHVJ96, 37]. Folglich umfassen Frameworks, im Unterschied zu Klassenbibliotheken, sowohl den softwaretechnischen Entwurf als auch die Implementierung objektorientierter Anwendungssysteme [John97, 11]. Aus diesem Grund unterstützt ein Framework nicht nur die Wiederverwendung von Code, sondern insbesondere die Wiederverwendung von Entwürfen [John97, 13; Pree97, v; Szyp98, 137; Pree95, 55; GHVJ96, 137]. JOHNSON vertritt sogar die Ansicht, dass objektorientierte Programmiersprachen als Notationssprache für den Entwurf von Frameworks verwendet werden können und somit keine speziellen Werkzeuge für die Erstellung und Verwendung von Frameworks notwendig sind [John97, 11].

Frameworks, die statt eines einzelnen Lösungsentwurfs ein vollständiges generisches Anwendungssystem zur Verfügung stellen, werden auch **Application-Frameworks** genannt [Pree95, 59; BMR+98, 391]. Außerdem können Frameworks entweder **domänenspezifisch** oder **domänenunabhängig** sein [Szyp98, 137; Pree95, 55]. Zunächst wurden Application-Frameworks für domänenunabhängige Problemstellungen, wie z. B. für grafische Benutzeroberflächen, entwickelt [Pree95, 55; John97, 12 f; Grif98, 114]. Mit der Veröffentlichung des Application-Frameworks San Francisco durch die IBM, das inzwischen Teil des IBM-WebSphere-Produktprogramms geworden ist, wurden auch domänenspezifische Application Frameworks für die Entwicklung betrieblicher Anwendungssysteme verfügbar [IBM02].

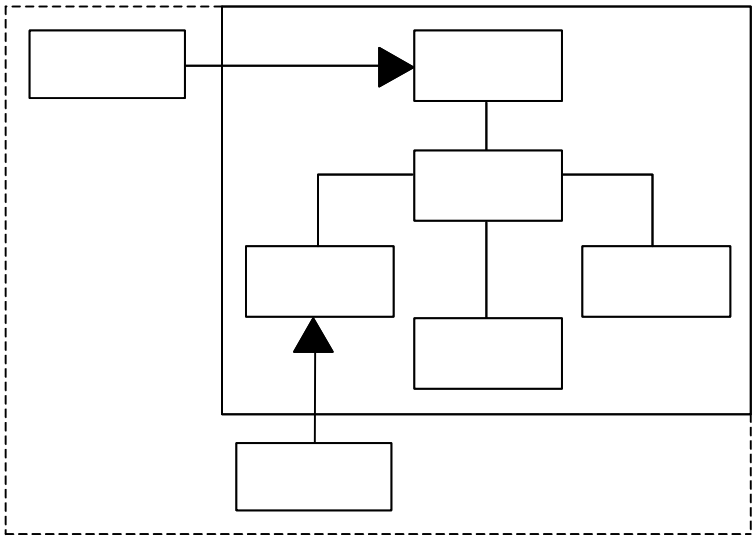
3.5.2.1 Untersuchung

Wie bereits in Abschnitt 3.1.3.6 erläutert, werden Frameworks bei ihrer Wiederverwendung hauptsächlich durch die Spezialisierung von abstrakten Klassen, die sich an vorgegebenen Stellen im Framework befinden, angepasst und erweitert (vgl. Abbildung 3.16 a)). Dies impliziert den Einsatz der Implementierungsvererbung und alle damit verbundenen Probleme für die Wiederverwendung, die in Abschnitt 3.5.1.1 ausführlich erörtert wurden. Solche Frameworks nennt man demzufolge auch **White-Box-Frameworks** [Pree97, 19; Szyp98, 137; Grif98, 115]. Durch eine häufige Wiederverwendung von Frameworks werden die darin enthaltenen abstrakten Klassen Stück für Stück konkretisiert [Pree97, 20; Grif98, 114 f]. Das bedeutet, dass die An-

passung und Erweiterung eines solchen Frameworks nicht zwangsläufig anhand von Spezialisierung und Implementierungsvererbung erfolgt, sondern sich auch durch Klassenbeziehungsformen, die auf Objektkomposition und expliziter Delegation basieren, realisieren lässt [Pree97, 20; Szyp98, 137]. Frameworks, die ausschließlich aus konkretisierten und dadurch direkt wiederverwendbaren Klassen bestehen, werden auch **Black-Box-Framework** genannt (vgl. Abbildung 3.16 b)) [Pree97, 31; Szyp98, 137; Grif98, 115].



a) White-Box-Framework



b) Black-Box-Framework

Abbildung 3.16: White-Box- und Black-Box-Framework

Folglich entwickeln sich aus White-Box-Frameworks mit zunehmendem Reifegrad Black-Box-Frameworks [Pree97, 21]. Da die konkretisierten Klassen auf der Basis von Objektkomposition und expliziter Delegation als Black Box wiederverwendet werden können und jeweils einen fachlichen Belang abbilden, weisen sie Eigen-

schaften auf, die auch von Software-Komponenten gefordert werden (siehe Abschnitt 4). Dies führt dazu, dass die in einem Black-Box-Framework enthaltenen Klassen vereinzelt bereits als Software-Komponenten interpretiert werden [Pree97, 30 ff; John97, 14; Wesk99, 7 ff; Same97, 33 f]. Dabei dient das Framework als anwendungsbezogene Infrastruktur für die Software-Komponenten, die deren Kommunikation und zielgerichtete Koordination sicherstellt [John97, 14; Same97, 34; Griff 98, 116; Lars00, 26; NiLu97; CHJK02, 38 f]. Allerdings impliziert diese Interpretation, dass die Software-Komponenten nur im Kontext ihres Frameworks problemlos wiederverwendet werden können [Grif98, 116; John97, 14; CHJK02, 38]. Außerdem ist die notwendige Abgrenzung zwischen einem solchen Software-Komponenten-Konzept und dem Klassenkonzept der Objektorientierung nicht vorhanden. Deswegen wird diese Interpretation in der vorliegenden Arbeit nicht weiterverfolgt.

Frameworks und Entwurfsmuster

Ebenfalls eng verbunden mit dem Framework-Konzept ist das Konzept der Entwurfsmuster [GHVJ96, 38 f; John97, 14 f; Pree97, 24 f; Szyp98, 137 ff]. Ein Framework wird vereinzelt als eine konkrete Implementierung eines Entwurfsmusters verstanden (siehe z. B. [John97, 15; BMR+98, 392]). Dieses Verständnis greift jedoch zu kurz, da ein Framework üblicherweise mehrere Entwurfsmuster, die zur Problemlösung miteinander in Beziehung stehen, implementiert [GHVJ96, 39; Szyp98, 138; John97, 15]. Tatsächlich fand die Verwendung von Entwurfsmustern auf dem entgegengesetzten Weg Eingang in die objektorientierte Anwendungssystementwicklung. Zur Dokumentation des GUI-Frameworks ET++ wurde eine Beschreibungsform gesucht, die abstrakter ist, als die verwendete Programmiersprache [Pree97, 25]. Daraus entstand das derzeit bekannteste Entwurfsmustersystem von GAMMA ET AL. [GHVJ96]. In dieser Arbeit werden unter anderem drei Merkmale identifiziert, anhand derer Frameworks und Entwurfsmuster unterschieden werden können [GHVJ96, 38 f]:

- Entwurfsmuster sind abstrakter als Frameworks. Frameworks liegen in Form von Code vor, Entwurfsmuster in Form von allgemein verständlichem Text. Deswegen ist die Dokumentation einer Problemlösung in Form eines Entwurfsmusters allgemeiner, umfangreicher und verständlicher als in Form eines Frameworks. Andererseits lässt sich ein Framework unmittelbar einsetzen und ausführen. Ein Entwurfsmuster muss zunächst geeignet implementiert werden.
- Entwurfsmuster sind kleiner als Frameworks. Wie bereits erwähnt, enthält ein Framework üblicherweise die Implementierung mehrerer Entwurfsmuster. Die Umkehrung gilt jedoch nie.

- Entwurfsmuster sind weniger spezialisiert als Frameworks. Frameworks sind stets für einen konkreten Einsatzbereich entworfen und können nur im Rahmen dieses Einsatzbereiches angepasst und erweitert werden. Entwurfsmuster hingegen lassen sich normalerweise in unterschiedlichen Einsatzbereichen verwenden.

Zusammenfassend kann man feststellen, dass Entwurfsmuster sowohl zur Dokumentation als auch zum Entwurf von Frameworks geeignet sind [GHVJ96, 38; Pree95, 63]. Im letztgenannten Fall werden reife und bewährte Entwürfe in das Framework eingebracht [Pree95, 63]. Dies steigert die Wiederverwendbarkeit des Frameworks. Darüber hinaus dienen Entwurfsmuster auch der Anpassung von Frameworks an spezielle Bedürfnisse [Pree95, 63; BMR+98, 391 f]. Dabei wird die gleiche Vorgehensweise angewendet, wie beim Entwurf eines Frameworks.

3.5.2.2 Bewertung

Anhand der bisherigen Ausführungen wird die Wiederverwendungs-, Wiederverwendbarkeits- und Verteilbarkeitsunterstützung der Erzeugnisart Framework wie folgt bewertet (vgl. Tabelle 3.10).

Forderungen	**Unterstützung**
Wiederverwendung	
Konzepte	O
Artefakte	+
bereichsübergreifend	+
planbar	+
kompositionell	+
generativ	-
Black-Box	O
Wiederverwendbarkeit	
hohes Abstraktionsniveau	-
Allgemeingültigkeit	O
hierarchische Strukturierung	+
Modularität	O
Verständlichkeit	-
Handhabbarkeit	O

Forderungen	Unterstützung
standardisierte und formalisierte Beschreibung	O
Anpassbarkeit	O
Erweiterbarkeit	O
Interoperabilität	O
Portierbarkeit	O
Qualität	O
Verteilbarkeit	
Kontrolle der Daten- und Funktionsredundanz	+
Kontrolle der Verknüpfungen	+
Erhaltung der semantischen und operationalen Integrität	+
Zielbezogenheit	+
Verteilungstransparenz	O

+ = vollständige O = teilweise - = keine

Tabelle 3.10: Bewertung der Erzeugnisart Framework

Frameworks sind wiederverwendbare Artefakte. Sie können mithilfe von Entwurfsmustern entworfen und bzw. oder dokumentiert werden. Allerdings existieren auch Frameworks, die keine Entwurfsmuster enthalten. Deshalb werden Konzepte in Form von Entwurfsmustern bei der Wiederverwendung eines Frameworks nicht zwangsläufig wiederverwendet.

Der Einsatzbereich eines Frameworks kann grundsätzlich domänenunabhängig oder domänenspezifisch sein. Außerdem können Frameworks sowohl innerhalb eines Anwendungssystems als auch anwendungssystemübergreifend wiederverwendet werden.

Die Wiederverwendung eines Frameworks findet meistens geplant und systematisch statt. Dabei wird die kompositionelle Technik verwendet. Durch den Einsatz von Entwurfsmustern kann ein Framework zwar entworfen und dokumentiert werden, die Wiederverwendung eines Frameworks ist jedoch nicht generativ.

White-Box-Frameworks werden ausschließlich durch Implementierungsvererbung angepasst und erweitert. Dies entspricht offensichtlich einer White-Box-Wiederverwendung. Demgegenüber können Black-Box-Frameworks auch ohne Imp-

lementierungsvererbung erweitert und angepasst werden. Da Black-Box-Frameworks jedoch nur über die schrittweise Spezialisierung von White-Box-Frameworks entstehen, wird die Black-Box-Wiederverwendung nur teilweise unterstützt.

Ein Framework weist aufgrund seiner Spezifikation in Code-Form ein realisierungsnahes und somit niedriges Abstraktionsniveau auf. Neben domänenunabhängigen existieren auch domänenspezifische Frameworks, die demzufolge nicht mehr allgemeingültig sind. Deswegen wird das Kriterium Allgemeingültigkeit nur teilweise unterstützt.

Durch Spezialisierung und Zerlegung der zugänglichen Framework-Klassen können Frameworks hierarchisch strukturiert werden. Da ein Framework keine einheitliche Schnittstelle aufweist und White-Box-Frameworks ausschließlich durch Implementierungsvererbung angepasst und erweitert werden, wird das Kriterium Modularität von Frameworks nicht vollständig unterstützt.

Wie bereits erwähnt, beschränkt sich die Dokumentation eines Frameworks auf seine Implementierung, gegebenenfalls ergänzt durch Entwurfsmuster. Deshalb ist eine intuitive Verständlichkeit der fachlichen Inhalte des Frameworks nicht gewährleistet. Die mangelnde Dokumentation und die hohe Komplexität aufgrund tiefer Spezialisierungshierarchien und vielen Interaktionsbeziehungen beeinträchtigen auch die Nachvollziehbarkeit eines Frameworks. Dies wird häufig als einer der schwerwiegendsten Nachteile von Frameworks erkannt [Grif98, 117 f; Eise98, 76; Schr01, 31; Kort01, 56; ScLi99, 87].

Da der Quellcode, gegebenenfalls ergänzt durch Entwurfsmuster, die einzige Dokumentation eines Frameworks darstellt, liegt zwar eine weitestgehend formalisierte Beschreibung vor, die aber aufgrund eines fehlenden Programmiersprachenstandards nicht standardisiert ist. Dies führt unter anderem dazu, dass Frameworks, die in unterschiedlichen Programmiersprachen entwickelt wurden, zunächst nicht interoperabel sind [Eise98, 76; Pree97, 9]. Sie können jedoch, wie Objekte verschiedener Programmiersprachen, über Kommunikationssysteme interoperabel gemacht werden. Für Frameworks, die in der gleichen Sprache programmiert wurden, stellt die Interoperabilität selbstverständlich kein Problem dar. Falls für Frameworks eine Entwicklungs- und Ausführungsplattform existiert, die neben der Programmiersprache eine *Virtual Machine* zur Verfügung stellt, ist auch die Portierbarkeit solcher Frameworks sichergestellt.

Zur Realisierung von Qualitätskriterien stehen für Frameworks zum Teil ebenfalls Entwicklungs- und Ausführungsplattformen oder verschiedene Middleware-Systeme zur Verfügung. Allerdings können die Qualitätskriterien aufgrund der Quellcode-Form eines Frameworks nicht ausreichend dokumentiert werden.

Wie bereits dargelegt, werden White-Box-Frameworks durch Implementierungsvererbung und Black-Box-Frameworks üblicherweise durch Objektkomposition mit expliziter Delegation angepasst und erweitert. Jedoch beeinträchtigt die Code-Basierung eines Frameworks dessen Anpassbarkeit und Erweiterbarkeit, da Anpassungen und Erweiterungen nur dann unproblematisch sind, wenn sie in der gleichen Programmiersprache durchgeführt werden [Pree97, 9; Eise98, 76]. Außerdem kann es auftreten, dass zu wenige oder ungeeignete Anpassungsstellen im Framework vorhanden sind [Eise98, 76].

Da durch die in einem Framework enthaltenen Klassen und Interaktionsbeziehungen das Integrationskonzept Objektintegration realisiert wird, sind auch hier die Bewertungen der erstgenannten vier Forderungen bezüglich der Verteilbarkeit mit den entsprechenden Zielerreichungsgraden dieses Integrationskonzepts identisch.

Aufgrund der Quellcode-Form eines Frameworks können Qualitätskriterien, die transparent realisiert werden sollen, nicht ausreichend spezifiziert werden. Für die tatsächliche transparente Realisierung der Qualitätskriterien und -ausprägungen stehen jedoch zunehmend niedrigere Basismaschinen in Form von verschiedenen Middleware-Systemen zur Verfügung.

4 Ein präskriptives Grundkonzept der Komponentenorientierung

Im vorangegangenen Kapitel wurden zunächst die Formalziele Wiederverwendung, Wiederverwendbarkeit und Verteilbarkeit im softwaretechnischen Entwurf und in der Implementierung von Anwendungssystemen erörtert und Forderungen zur bestmöglichen Erfüllung dieser Formalziele aufgestellt. Auf der Grundlage dieser Forderungen folgten anschließend Untersuchungen und Bewertungen der Erzeugnisarten Klasse und Framework hinsichtlich ihrer jeweiligen Wiederverwendungs-, Wiederverwendbarkeits- und Verteilbarkeitsunterstützung.

Wie die Ergebnisse dieser Untersuchungen und Bewertungen zeigen, unterstützt weder die eine noch die andere Erzeugnisart die formulierten Forderungen vollständig. Deswegen ist eine alternative Erzeugnisart erforderlich, die eine vollständige Unterstützung der Forderungen sicherstellt. Da für die Erzeugnisart Software-Komponente noch kein einheitliches Begriffsverständnis vorliegt (vgl. Abschnitt 3.1.3.6), besteht die Möglichkeit, anhand eines präskriptiven **Grundkonzepts der Komponentenorientierung** ein Software-Komponenten-Konzept zu entwickeln, das insbesondere die vollständige Unterstützung der im vorangegangenen Kapitel erarbeiteten Forderungen zum Ziel hat. Dieses Grundkonzept soll darüber hinaus nicht nur zur Konstruktion betrieblicher Anwendungssysteme, sondern auch zur Konstruktion von Anwendungssystemen für andere Domänen eingesetzt werden können.

4.1 Stand der Entwicklungen in der Komponentenorientierung

Eine wesentliche charakteristische Eigenschaft von Software-Komponenten ist deren Wiederverwendbarkeit. Bereits im Jahre 1968 wurde auf einer Nato-Konferenz zum Thema Software-Engineering von McIlroy festgestellt, dass eine (Wieder-) Verwendung von Software-Bauteilen, nach dem Vorbild von Bauteilen entsprechender Fertigungstechniken anderer Industriebereiche, notwendig für die Fortentwicklung der gesamten Software produzierenden Branche sei [McIll69]. In diesem Zusammenhang schlug er vor, einen eigenen Industriezweig zu schaffen, der sich ausschließlich mit der Fertigung von solchen Software-Bausteinen beschäftigt [McIl69]. Die dabei erstmals formulierte Idee der wiederverwendbaren Software-Komponenten prägte daraufhin viele erfolgsversprechende Entwicklungen auf dem Gebiet der Software-Entwicklung, wie z. B. die Objektorientierung oder auch die Komponentenorientierung. Im Unterschied zu McIlroy, der den Begriff „Software-Komponente“ als Oberbegriff für alle wiederverwendbaren und rekombinierbaren Software-Bauteile verwendete, wird der Begriff heutzutage, in unterschiedlich konkretisierten Ausprägungen, als zentrales Element eines eigenständigen Konzepts angesehen. Allerdings besteht, wie bereits erwähnt, noch keine Einigkeit über das Begriffsverständnis und über das zugehörige Grundkonzept. Der derzeit diesbezüglich stattfindende Dis-

kurs ähnelt dem Methoden- und Begriffs-„Streit“, der in den Anfangsjahren der objektorientierten Software-Entwicklung stattfand [FoSc00, 3].

Anwendungsunabhängige Komponententechnologien

MCILLROYS vorgeschlagene Metapher eines Bauteils, das mit Bauteilen anderer Hersteller zu einem integrierten Fertigprodukt zusammengefügt werden kann, lässt sich verhältnismäßig problemlos auf Software-Komponenten übertragen, die anwendungsunabhängig sind und deshalb wenig Semantik besitzen. Dazu gehören unter anderem Software-Komponenten für die Benutzerkommunikation, die Datenhaltung und die Rechnerkommunikation. Deswegen fanden so genannte Komponententechnologien für grafische Benutzeroberflächen, wie z. B. Visual Basic- und ActiveX-Controls von Microsoft oder JavaBeans von Sun, bis heute eine große Verbreitung [LiSe97, 93; Hopk00, 29]. Auch wenn es sich um eine stark vereinfachte Form von Software-Komponenten handelt, erfuhr die komponentenorientierte Anwendungssystementwicklung durch den Erfolg dieser herstellerspezifischen Komponententechnologien einen initialen Schub [Szyp98, 19; Fran99, 12; MeMi99, 35; KoBo98, 36; BrWa98, 42]. Leider entstand dadurch auch ein diffuses Verständnis einer Software-Komponente, da jeder Hersteller seine eigene Meinung vetrat und weiterhin vertritt [KoBo98, 35]. Deswegen ist ein methodisch hergeleitetes und einheitliches Verständnis einer Software-Komponente, wie es in der vorliegenden Arbeit aufgebaut wird, notwendig für weitere Untersuchungen im Bereich der komponentenorientierten Anwendungssystementwicklung.

Ansätze zur Entwicklung fachlicher Komponenten

Aufgrund der erfolgreichen Wiederverwendung von Komponenten in der Software-Technik wurde dieses Prinzip auch im Fachentwurf angewendet [Digr98, 62; Sinz99g]. So sollten nicht nur fachliche Entwurfsergebnisse wiederverwendbar und erweiterbar gemacht, sondern durch eine phasenübergreifende komponentenorientierte Sichtweise der Fachentwurf und der softwaretechnische Entwurf verbunden werden [FSH+98, 86; Wesk99, 5 f]. Es folgten mehrere Ansätze zur Entwicklung fachlicher Komponenten. Hierbei sind insbesondere der **Business-Object-Ansatz** als der bekannteste und der **Application-Object-Ansatz** als der methodisch vielversprechendste Ansatz hervorzuheben [Wesk99; Digr98; FSH+97].

Jedes **Business Object** bzw. **Application Object** kapselt ein fachliches Entwurfsergebnis und besitzt notwendige Eigenschaften zu seiner Wiederverwendung in unterschiedlichen Kontexten [Wesk99, 8 f; FSH+97, 37 ff].

Im Business-Object-Ansatz können Business Objects sowohl im Fachentwurf als auch im softwaretechnischen Entwurf definiert werden. Einem Business Object im Fachentwurf ist genau ein Business Object im softwaretechnischen Entwurf zuge-

ordnet. Allerdings liegen im Fachentwurf keine klaren Abgrenzungskriterien für Business Objects vor [FSH+98, 86]. Es wird vielmehr versucht, ausgehend von systemtechnisch beschriebenen Komponenten durch Abstraktion fachliche Bausteine zu erhalten [FSH+98, 86].

Dagegen ist die Entwicklung und die Wiederverwendung von Application Objects im gleichnamigen Ansatz auf den Fachentwurf beschränkt, da deren Abgrenzung ausschließlich unter Bezugnahme fachlicher Aspekte erfolgt (vgl. Abbildung 4.1) und softwaretechnische Belange darin noch keine Berücksichtigung finden. Deshalb können sie in der Regel nicht unverändert in den softwaretechnischen Entwurf übernommen werden. Beispielweise ist die Kapselung von Vorgangsobjekttypen und zugehörigen konzeptuellen Objekttypen in einem Application Object aus softwaretechnischer Sicht problematisch, wenn die persistenten Anteile der konzeptuellen Objekttypen von einem Datenbanksystem verwaltet werden sollen, das sich auf einem anderen Rechnersystem befindet als das restliche Anwendungssystem.

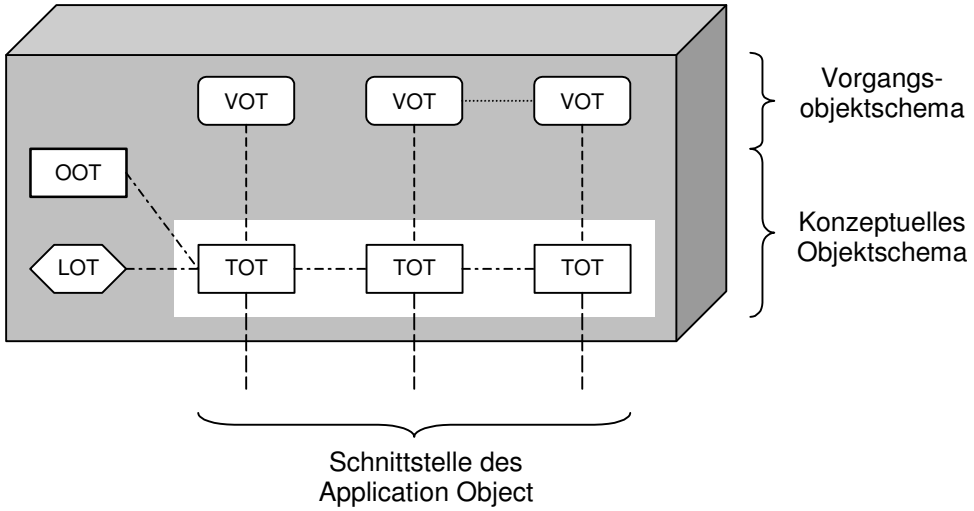


Abbildung 4.1: Mikro-Architektur eines Application Object [FSH+97]

Dennoch müssen zur Abgrenzung von Software-Komponenten für das Anwendungsfunktionen-Teilsystem eines Anwendungssystems fachliche Aspekte berücksichtigt werden. Ein möglicher Lösungsweg für dieses Problem wird in Kapitel 5 vorgestellt. Hierbei wird eine vom Fachentwurf ausgehende Komponentenabgrenzung im softwaretechnischen Entwurf empfohlen. Ein weiteres Problem solcher Software-Komponenten ist deren hoher semantischer Gehalt. Dadurch weisen sie höhere Ansprüche beim Entwurf, bei der Spezifikation, beim Wiederauffinden, bei der Anpassung und beim Kombinieren mit anderen Software-Komponenten in unterschiedlichen Anwendungssystemen auf.

Anwendungsorientierte Komponententechnologien

Das derzeit gesteigerte Interesse am Einsatz von Software-Komponenten zur Entwicklung des Anwendungsfunktionen-Teilsystems eines Anwendungssystems wurde ebenfalls durch eine Initiative eines Software-Herstellers geweckt. Die Firma SUN veröffentlichte 1997 die erste Version ihrer **Enterprise-JavaBeans-Spezifikation**, auf deren Basis erstmals Software-Komponenten speziell für die Anwendungsschicht in Mehrschichtarchitekturen erstellt werden konnten und können (siehe Abschnitt 6.1.5 und [SUN02a]). Der kommerzielle Erfolg von darauf basierenden Software-Produkten belebte die Fachdiskussionen über das Thema komponentenorientierter Anwendungssystementwicklung, da anhand dieser Spezifikation gezeigt werden konnte, dass sich die Wiederverwendung von Software-Komponenten nicht nur auf die Kommunikations- oder Datenhaltungsteilsysteme eines Anwendungssystems beschränken muss.

Komponentenorientierte Anwendungssystementwicklung

Mit den Herausforderungen der komponentenorientierten Anwendungssystementwicklung beschäftigen sich bereits mehrere Autoren aus der Theorie und der Praxis der Anwendungssystementwicklung (siehe z. B. [Szyp98; Same97; Grif98; Whit02]). Im Laufe der Zeit entwickelte sich daraus eine eigene Disziplin der Anwendungssystementwicklung, die ***Component-Based Software Engineering (CBSE)*** genannt wird [BrWa98; KoBo98]. Gelegentlich findet man dafür auch die allgemeinere Bezeichnung ***Component-Based Development (CBD)*** vor [Whit02, 186; Clem01, 192; MeMi99, 35]. Mit *CBSE* wird insbesondere das ingenieurmäßige Vorgehen bei der Entwicklung komponentenbasierter Anwendungssysteme betont [BrWa98, 37]. Komponentenbasierte Anwendungssysteme selbst werden häufig als ***Component-Based Systems (CBS)*** [BrBu00, 54] oder, in Anlehnung an den Begriff „Software“, als ***Componentware*** bezeichnet [Zimm99b, 47 f]. „Componentware [bezeichnet] Softwaresysteme, die sich aus einzelnen, interagierenden Komponenten zusammensetzen, deren Entwicklung unabhängig voneinander erfolgt ist oder zumindest für den wiederholten Einsatz in einer Familie von Anwendungen geplant wurde“ [Grif98, 592]. Zum Teil wird der Begriff *Componentware* aber auch allgemeiner verwendet. Dann charakterisiert er alle Software-Produkte, die im Rahmen einer komponentenorientierten Anwendungssystementwicklung verwendet werden können: „Componentware refers to software assets, that are useful for *CBD*, and can be bought and sold“ [Whit02, 186]. Dazu gehören selbstverständlich Software-Komponenten, aber z. B. auch Komponenten-Frameworks [NiLu97, 12 ff]. Im Zusammenhang mit den zunehmenden Forschungstätigkeiten im Bereich der komponentenorientierten Anwendungssystementwicklung wurde auch der Komponentenbegriff weiter konkretisiert. Einige Konkretisierungen beziehen sich insbesondere auf die Vermarktbarkeit

und Erwerbbarkeit von fertigen Software-Komponenten und bezeichnen diese auch als ***Commercial off-the-shelf (COTS)*** (siehe z. B. [Trac01; CaLo00]). Andere Konkretisierungen beschreiben eher die technischen Eigenschaften von Software-Komponenten (siehe z. B. [Szyp98, 276; Digr98, 62]). Insgesamt konnte man sich jedoch, wie bereits erwähnt, bis heute auf keine einheitliche Definition einigen. Mehr oder weniger ausführliche Vergleiche unterschiedlicher Komponentendefinitionen enthalten beispielsweise [BrWa98, 38 f; BrBu00, 55; Fran99, 12; Hopk00, 27].

Abgrenzung zwischen Software-Komponente und Objekt

Selbst die Zusammenhänge und Unterschiede zwischen einem Objekt und einer Software-Komponente konnten bisher nicht eindeutig geklärt werden. Einigkeit besteht darin, dass Objekt und Software-Komponente nicht dieselbe Erzeugnisart bezeichnen [Meye99, 139; NiLu97, 9 f; KoBo98, 35; Zimm99b, 48; BrWa98, 39]. Außerdem ist unbestritten, dass Objekte auf der Ausprägungsebene und Software-Komponenten auf der Typebene zu finden sind [Szyp98, 30 f; Fran99, 12; NiLu97, 10]. Das bedeutet, dass Software-Komponenten eher vergleichbar sind mit Klassen des Grundkonzepts der Objektorientierung.

4.2 Fundament des präskriptiven Grundkonzepts der Komponentenorientierung

Zur Erläuterung des präskriptiven Grundkonzepts ist es erforderlich, zwischen einer Software-Komponente und einem komponentenbasierten Anwendungssystem zu unterscheiden. Beide werden im softwaretechnischen Entwurf spezifiziert und in der Implementierung realisiert. Außerdem muss im softwaretechnischen Entwurf und in der Implementierung zwischen dem Typ und den Instanzen einer Software-Komponente bzw. eines komponentenbasierten Anwendungssystems differenziert werden. Dabei ist eine Software-Komponente selbst, wie erwähnt, auf der Typebene zu finden. Folglich werden aus einer Software-Komponente konkrete Software-Komponenten-Instanzen erzeugt [Szyp98, 30; Fran99, 12; NiLu97, 16 f].

Daraus ergeben sich folgende orthogonale Merkmale, die den **Beschreibungsraum des Grundkonzepts** aufspannen (vgl. Abbildung 4.2):

- **Beschreibungsobjekt** mit den Ausprägungen Software-Komponente und komponentenbasiertes Anwendungssystem,
- **Entwicklungsphase** mit den Ausprägungen softwaretechnischer Entwurf und Implementierung,
- **Abstraktionsebene** mit den Ausprägungen Typ und Instanz und
- **Systemsicht** mit den Ausprägungen Struktur und Verhalten.

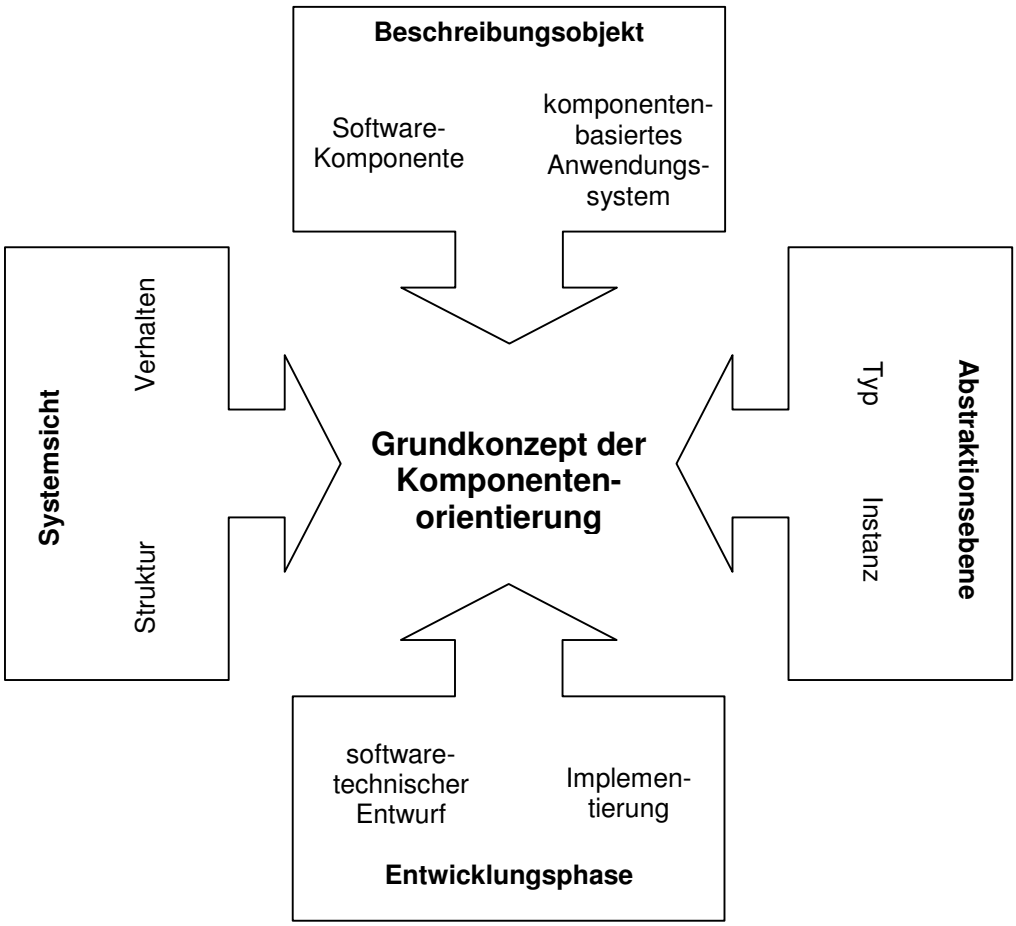


Abbildung 4.2: Beschreibungsdimensionen des Grundkonzepts der Komponentenorientierung

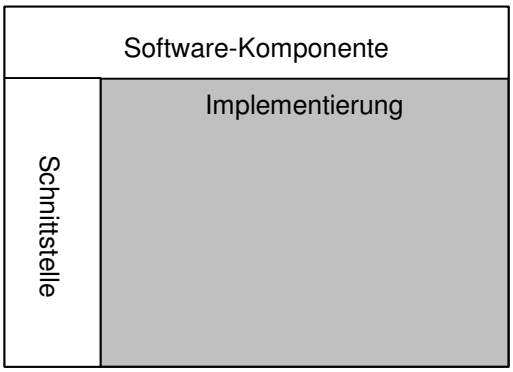
Im Folgenden wird zunächst das Beschreibungsobjekt Software-Komponente in beiden Entwicklungsphasen, auf beiden Abstraktionsebenen und aus beiden Systemsichten erläutert. Anschließend wird in derselben Weise das Beschreibungsobjekt komponentenbasiertes Anwendungssystem dargestellt.

Software-Komponenten sollen bestmöglich wiederverwendbar und verteilbar sein. Deswegen müssen sie im softwaretechnischen Entwurf die in Abschnitt 3.3 genannten Forderungen und die in Abschnitt 3.4 vorgestellten Prinzipien realisieren.

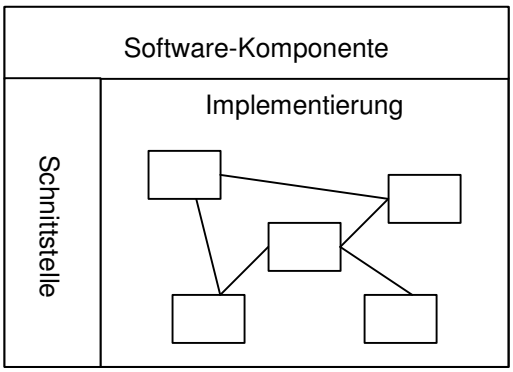
Eine wichtige Forderung wird dadurch erfüllt, dass bei der komponentenorientierten Entwicklung von Anwendungssystemen zwischen der Entwicklung wiederverwendbarer Software-Komponenten (*Development for Reuse*) und der Entwicklung mit wiederverwendbaren Software-Komponenten (*Development with Reuse*) unterschieden wird. Somit findet die Wiederverwendung von Software-Komponenten geplant und systematisch statt. Daraus ergeben sich zwei Sichtweisen auf eine Software-Komponente.

Sichtweisen auf eine Software-Komponente

Zum einen ist die **Sichtweise der Verwender** von Software-Komponenten auf diese beim *Development with Reuse* zu nennen (vgl. Abbildung 4.3 a)). In dieser Sichtweise besteht weitgehend Einigkeit darüber, dass eine Software-Komponente aufgrund der Trennung von Schnittstelle und Implementierung **implementationstransparent** ist [MKR+00, 24]. Deshalb kann keine Aussage darüber getroffen werden, ob Software-Komponenten, die zur Wiederverwendung bereit stehen, auf der Basis von Klassen, Prozeduren oder anderen Elementen der Programmierung implementiert wurden [Szyp98, 31].



a) Software-Komponenten aus der Sichtweise des Verwenders



b) Software-Komponenten aus der Sichtweise des Erstellers

Abbildung 4.3: Software-Komponente aus unterschiedlichen Sichtweisen

Zum anderen ist die **Sicht der Ersteller** von Software-Komponenten auf diese beim *Development for Reuse* zu erwähnen (vgl. Abbildung 4.3 b)). Hier werden zwei unterschiedliche Standpunkte vertreten. Auf der einen Seite lässt sich die Erstellung von Software-Komponenten unabhängig von einem bestimmten Entwurfs- und Implementierungsparadigma betrachten [Hopk00, 28]. Auf der anderen Seite, die auch mehrheitlich vertreten wird, ist die Erstellung von Software-Komponenten eng mit

dem Grundkonzept der Objektorientierung verknüpft [Meye99, 139; Zimm99b, 47]. Dabei werden Klassen, die zur Abbildung eines gemeinsamen fachlichen Belangs in verschiedenartigen Beziehungen zueinander stehen, in einer Software-Komponente gekapselt [NiLu97, 9 f; Zimm99b, 48; Szyp98, 32]. Dadurch können einerseits ausgereifte und allgemein akzeptierte objektorientierte Entwurfsansätze und Programmiersprachen zur Erstellung von Software-Komponenten weiterverwendet werden und andererseits die Wiederverwendbarkeit und Verteilbarkeit der resultierenden Entwurfs- und Implementierungsergebnisse durch die Kapselung in Software-Komponenten verbessert werden. Da dieser Standpunkt ein weitaus größeres Potenzial zur Unterstützung mittels einer durchgängigen Methodik aufweist, wird er im Folgenden zugrunde gelegt.

4.3 Software-Komponente im softwaretechnischen Entwurf

Die Ausführungen zum Beschreibungsobjekt **Software-Komponente** beginnen auf der Typebene im softwaretechnischen Entwurf. Dies stellt auch gleichzeitig den Ausgangspunkt für alle weiteren Ausführungen des Grundkonzepts der Komponentenorientierung dar, da

- eine Software-Komponente die Struktur und das Verhalten ihrer konkreten Instanzen spezifiziert,
- mehrere in Beziehung stehende Software-Komponenten die Struktur eines komponentenbasierten Anwendungssystems bilden, und deren Verhaltensspezifikationen schließlich auch das Verhalten dieses Anwendungssystems bestimmen und
- das Ergebnis des softwaretechnischen Entwurfs einer Software-Komponente bzw. eines komponentenbasierten Anwendungssystems eine Abstraktion seiner Implementierung darstellt.

4.3.1 Strukturmerkmale einer Software-Komponente im softwaretechnischen Entwurf

Eine Software-Komponente im softwaretechnischen Entwurf ist anhand eines einzigen fachlichen Belangs abgegrenzt und somit in sich abgeschlossen. Sie besteht im Kern aus einem Typspezifikationsteil und einem entsprechenden Implementierungsteil.

Typspezifikationsteil

Der **Typspezifikationsteil** umfasst einen fachlich bestimmten Namen, eine Verwaltungsschnittstelle, eine fachliche Schnittstelle und eine Deklaration nicht-funktionaler Eigenschaften und Eigenschaftswerte (vgl. Abbildung 4.4). Nur für Dokumentationszwecke besitzt eine Software-Komponente zusätzlich einen **Teil** für die **Kurzbeschreibung** des zugehörigen fachlichen Belangs.

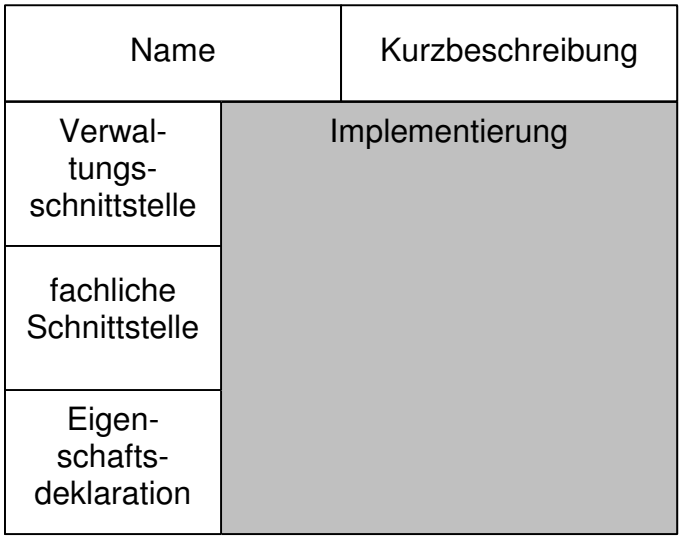


Abbildung 4.4: Software-Komponente beim *Development with Reuse*

Mithilfe des **fachlich bestimmten Namens** und der Kurzbeschreibung des zugehörigen fachlichen Belangs ist eine intuitive Verständlichkeit der Software-Komponente gegeben. Die Verwendung einer standardisierten Sprache erhöht die intuitive Verständlichkeit und erleichtert auch das Auffinden im Rahmen der Wiederverwendung. In Abschnitt 5.2.2.3 wird eine dafür geeignete standardisierte Sprache vorgeschlagen.

Die **Deklaration** von **nicht-funktionalen Eigenschaften** und **Eigenschaftswerten** trennt die technischen Belange von den fachlichen Belangen. Sie enthält quantifizierbare Qualitätskriterien, wie z. B. die Antwortzeit und die Speichernutzung, und deren Ausprägungen. Durch diese kann die Erreichbarkeit bestimmter, für den Verwender wichtiger Qualitätsfaktoren beeinflusst werden. Aufgrund der deklarativen Beschreibungsform muss der Nutzer einer Software-Komponente das Lösungsverfahren zur Gewährleistung der nicht-funktionalen Eigenschaften und der zugehörigen Eigenschaftswerte nicht kennen. Für die Erbringung der nicht-funktionalen Eigenschaften und Eigenschaftswerte sind geeignete **Dienste der zugrundeliegenden Basismaschinen** verantwortlich. Nicht-funktionale Eigenschaften mit diskreten Wertebereichen, wie z. B. die Persistenzeigenschaft, werden konkret spezifiziert. Dagegen erfolgt die Spezifikation nicht-funktionaler Eigenschaften mit kontinuierlichen Wertebereichen anhand eines einzuhaltenden Ausprägungsintervalls. Auf diese Wei-

se lässt sich eine zu starke Basismaschinenabhängigkeit vermeiden [Szyp98, 46]. Auch für die deklarative Beschreibungsform gilt, dass durch die Verwendung einer standardisierten Notation die Nachvollziehbarkeit der Software-Komponente weiter erhöht werden kann. Ein diesbezüglicher Vorschlag enthält ebenfalls der Abschnitt 5.2.2.3.

Über die **Verwaltungsschnittstelle** lassen sich Software-Komponenten-Instanzen erzeugen, auffinden, passivieren, reaktivieren und zerstören. Dagegen können über die **fachliche Schnittstelle** ausschließlich fachliche Operatoren ausgeführt werden. Diese Trennung der Verwaltungsschnittstelle von der fachlichen Schnittstelle entspricht im Wesentlichen einer Realisierung des Entwurfsmusters ***Abstrakte Fabrik*** [GHVJ96, 107 ff]. Durch die Anwendung dieses Entwurfsmusters kann ein System, im vorliegenden Fall eine Software-Komponente, unter anderem davon unabhängig gemacht werden, wie seine Instanzen erzeugt, zusammengesetzt und repräsentiert werden [GHVJ96, 109]. Über die Verwaltungsschnittstelle einer Software-Komponente können jedoch nicht nur Erzeugungsoperatoren ausgeführt werden, sondern alle Operatoren, die sich auf den Lebenslauf von Software-Komponenten-Instanzen beziehen.

Sowohl in der Verwaltungsschnittstelle als auch in der fachlichen Schnittstelle werden, analog zur Klasse im Grundkonzept der Objektorientierung, die Operatoren der Software-Komponente anhand ihrer **Signaturen** spezifiziert. Diese bestehen aus einem intuitiv verständlichen Operatorennamen, einer optionalen Liste von Parametern, die sich aus jeweils einem Parameternamen und einem Parameterdatentyp zusammensetzt, und einem Rückgabedatentyp. Attribute der Software-Komponente werden nicht expliziert, sondern nur anhand entsprechender Zugriffsoperatorensignaturen in der fachlichen Schnittstelle spezifiziert. Dadurch lässt sich das Prinzip der Kapselung beibehalten.

Implementierungsteil

Der **Implementierungsteil** stellt die Innensicht einer Software-Komponente dar und ist getrennt von den bisher genannten Strukturelementen, die die Außensicht der Software-Komponente beschreiben. Der Verwender kann nur über die einfach gehaltenen und wohldefinierten Schnittstellen der Außensicht auf den Implementierungsteil zugreifen. Auf diese Weise wird das Prinzip der Trennung von Schnittstelle und Implementierung, das Prinzip der Kapselung und damit auch das Geheimnisprinzip im softwaretechnischen Entwurf realisiert. Zur Umsetzung des Prinzips der Modularisierung sowie des Prinzips der maximalen inneren Kohärenz und der minimalen äußeren Bindung durch informale und funktionale Bindung befinden sich im Implementierungsteil **Operatoren**, die auf einer abgeschlossenen Menge von **Attributen** operieren und einen einzigen fachlichen Belang abbilden.

Aus Sicht des Erstellers enthält der Implementierungsteil eine in sich abgeschlossene objektorientierte Klassenstruktur (vgl. Abbildung 4.5). Diese wird mithilfe des Grundkonzepts der Objektorientierung, das in Abschnitt 3.5.1.1 ausführlich behandelt wurde, erstellt. Daraus folgt, dass einerseits ausgereifte und allgemein akzeptierte objektorientierte Entwurfsansätze zur Implementierung einer Software-Komponente verwendet werden können, aber andererseits die problematischen Merkmale des Grundkonzepts der Objektorientierung, wie Klassenvererbung und Vererbungshierarchien, auf die interne Implementierung einer Software-Komponente beschränkt bleiben. Diese Kapselung von Klassenvererbung und Vererbungshierarchien innerhalb einer Software-Komponente ist eine zentrale Forderung, die bereits mehrfach an das Konzept einer Software-Komponente gestellt wurde (siehe z. B. [PfSz02, 5; Pers03, 4]).

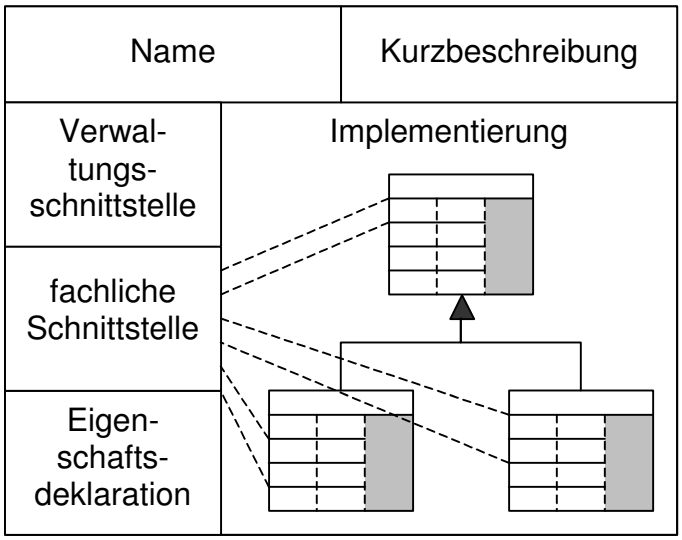


Abbildung 4.5: Software-Komponente beim *Development for Reuse*

Für den Entwurf der Klassenstrukturen können Entwurfsmuster als wiederverwendbare Konzepte eingesetzt werden. Somit wird beim *Development for Reuse* bereits eine generative Wiederverwendung anhand von Mustern durchgeführt. Die Attribute der in einer Software-Komponente enthaltenen Klassen stellen gleichzeitig die Attribute der Software-Komponente dar. Analog dazu sind die öffentlichen Operatoren und Operatorensignaturen der in einer Software-Komponente enthaltenen Klassen auch mit den Operatoren und Operatorensignaturen der Software-Komponente identisch.

4.3.2 Verhaltensmerkmale einer Software-Komponente im softwaretechnischen Entwurf

Wie bereits erläutert, beschreibt die Typspezifikation einer Software-Komponente neben der Struktur auch das Verhalten ihrer Instanzen aus der Außensicht. Da die Typspezifikation einer Software-Komponente den Namen, die Schnittstellen und die

Deklaration nicht-funktionaler Eigenschaften und Eigenschaftswerte umfasst, müssen alle zur Nachvollziehbarkeit erforderlichen Informationen darin enthalten sein.

Vertragsbasierte Typspezifikation

Zu diesem Zweck können die Schnittstellen und die Deklaration als Bestandteile eines Vertrags zwischen Nutzer und Software-Komponente betrachtet werden [Meye90, 125; Szyp98, 43; Wolf96, 63; BJP+99, 38; CoTu00, 180; CHJK02, 37; Fran99, 14; DeRo99, 50]. Wie in der realen Geschäftswelt regelt ein solcher **Vertrag** die Rechte und Pflichten, die von beiden Vertragspartnern erfüllt werden müssen [Meye90, 125; Wolf96, 63]. Dabei nimmt er die zur Erfüllung der Rechte und Pflichten notwendigen Verfahren nicht vorweg [BJP+99, 38]. Somit kann die Wiederverwendbarkeit einer Software-Komponente in einem bestimmten Kontext ohne Kenntnis einer konkreten Implementierung und vor einem konkreten Einsatz bestimmt werden [BJP+99, 38].

Ein softwaretechnischer Vertrag besteht aus einer Menge von **Zusicherungen**, die angeben, was die Vertragspartner tun müssen, aber nicht, wie sie es tun sollen [Meye90, 121]. Dabei werden verschiedene Arten von Zusicherungen unterschieden.

Vor- und Nachbedingungen

Zum einen können zur semantischen Spezifikation eines Operators Vor- und Nachbedingungen angegeben werden [Meye90, 122]. Eine **Vorbedingung** drückt diejenigen Eigenschaften aus, die vor dem Aufruf eines Operators immer gelten müssen. In der **Nachbedingung** werden die vom Operator bei seiner Beendigung gewährleisteten Eigenschaften beschrieben [Meye90, 122]. Vorbedingungen sind demnach die Pflichten, die ein Nutzer einer Software-Komponenten-Instanz erbringen muss, wenn er einen bestimmten Operator verwenden will. Auf der anderen Seite sind Nachbedingungen die Pflichten, die eine Software-Komponenten-Instanz erfüllen muss, nachdem ein bestimmter Operator verwendet wurde [Wolf96, 64; Szyp98, 43]. Vor- und Nachbedingungen werden in Form von **logischen Ausdrücken**, die sich auf Zustände der Software-Komponenten-Instanz oder anderer Software-Komponenten-Instanzen beziehen, spezifiziert. Dabei ist ein bestimmter Zustand einer Software-Komponenten-Instanz definiert durch ihre aktuellen Attributwerte. Neben der besseren Nachvollziehbarkeit der Software-Komponente hat die Spezifikation von Vor- und Nachbedingungen den Vorteil, dass sowohl der Nutzer auf die Erfüllung der Nachbedingungen als auch die Software-Komponenten-Instanz auf die Erfüllung der Vorbedingungen vertrauen können [Szyp98, 43; Meye90, 126]. Letzteres kommt folglich auch dem Ersteller einer Software-Komponente zu gute. Wenn in den Vorbedingungen eines Operators Nachbedingungen anderer Operatoren enthalten sind, können auch Reihenfolgebeziehungen zwischen Operatoren einer Software-Komponente bzw. zwischen Operatoren verschiedener Software-Komponenten hergestellt wer-

den. Zur genaueren Spezifikation der Synchronisation von Operatorenausführungen können die logischen Ausdrücke der Vor- und Nachbedingungen beispielsweise durch temporale Operatoren ergänzt werden [Szyp98, 45; CoTu00, 186]. Im weiteren Verlauf des Abschnitts wird darauf noch näher eingegangen.

Die Operatorensignaturen in den Schnittstellen stellen ebenfalls schon einfache Zusicherungen dar [Szyp98, 77]. Die Datentypen der Eingabe- und Durchlaufparameter einer Operatorensignatur sind Vorbedingungen des Operators und die Datentypen der Ausgabe- und Durchlaufparameter und des Rückgabewertes sind die zugehörigen Nachbedingungen [Szyp98, 77].

Invarianten

Neben den Vor- und Nachbedingungen existieren Eigenschaften, die unabhängig von bestimmten Operatoren immer gewährleistet sein müssen. Diese globalen Eigenschaften werden **Invarianten** genannt und sind von allen Operatoren einer Software-Komponenten-Instanz und vom Nutzer zu beachten [Meye90, 133; Wolf96, 64; BJP+99, 39; DeRo99, 50]. Sie können sowohl funktional als auch nicht-funktional sein. Die **funktionalen Invarianten** werden ebenfalls durch logische Ausdrücke, die sich auf Zustände der Software-Komponenten-Instanz oder anderer Software-Komponenten-Instanzen beziehen, spezifiziert. Zu den **nicht-funktionalen Invarianten** gehören die deklarierten nicht-funktionalen Eigenschaften einschließlich der zugehörigen Eigenschaftswerte. Sie bestimmen maßgeblich das Gesamtverhalten einer Software-Komponenten-Instanz [CHJK02, 37]. In Grenzfällen kann die Verschlechterung oder gänzliche Abwesenheit einer nicht-funktionalen Eigenschaft die Erbringung einer funktionalen Eigenschaft beeinträchtigen oder sogar verhindern [Szyp98, 46]. Beispiele dazu sind unter anderem in [Szyp98, 44] aufgeführt.

Übererfüllung von Zusicherungen

Die genannten Zusicherungen sind als Mindesteigenschaften zu verstehen. So kann bespielsweise ein Nutzer einer Software-Komponenten-Instanz restriktivere Vorbedingungen oder Invarianten aufweisen, als von der Software-Komponenten-Instanz verlangt. Analog dazu kann eine Software-Komponenten-Instanz nach der Ausführung eines Operators zusätzliche Eigenschaften vorweisen, die nicht durch die Nachbedingungen und Invarianten spezifiziert wurden [Szyp98, 73 f]. Durch diese Flexibilität ist es möglich, unterschiedliche Nutzer und unterschiedliche Software-Komponenten-Instanzen zu verwenden, solange diese zumindest die vereinbarten Eigenschaften zusichern [Szyp98, 76].

Da die Schnittstellen und die Deklaration, zusammen mit dem Namen, den Typ der Software-Komponente spezifizieren, lässt sich auch die gesamte Typspezifikation als Vertrag ansehen [Szyp98, 78]. Aufgrund der oben erläuterten Möglichkeit der Über-

erfüllung von Verträgen, ist demzufolge auch eine Typspezifikation übererfüllbar. Dies ist bei der Untertypenbildung der Fall, die in Abschnitt 4.5 noch näher erläutert wird [Szyp98, 78; BBK+97, 22 f].

Deklaration von Ausnahmeverhalten

Anhand der Operatorensignaturen, der zugehörigen Vor- und Nachbedingungen und der funktionalen Invarianten wird in der Regel nur das Normalverhalten der Instanzen einer Software-Komponente beschrieben. Abweichungen vom Normalverhalten können sehr vielfältig sein und werden mit Rücksicht auf die Verständlichkeit und Nachvollziehbarkeit einer Software-Komponente nicht explizit spezifiziert. Stattdessen wird jede Operatorsignatur in der Schnittstelle durch **Deklarationen möglicher Ausnahmen**, die während der Durchführung des zugehörigen Operators auftreten können, ergänzt.

Vertragsebenen von Software-Komponenten

Die genannten Vertragsbestandteile lassen sich anhand der von [BJP+99] vorgeschlagenen Aufteilung von Verträgen für Software-Komponenten in vier Ebenen weiter systematisieren (vgl. Abbildung 4.6). Die Aufteilung orientiert sich dabei an der steigenden Verhandelbarkeit der Vertragsinhalte [BJP+99, 38].

Abbildung 4.6: Vertragsebenen von Software-Komponenten (in Anlehnung an [BJP+99, 39])

Die unterste Ebene enthält die **Basisverträge**, die benötigt werden, damit eine Software-Komponente grundsätzlich funktioniert. Sie bestimmen die Operatorennamen einschließlich der zugehörigen Ein- und Ausgabeparameter, der Rückgabewerttypen und der Ausnahmedeklarationen [BJP+99, 38].

Auf der nächsthöheren Ebene werden die **Verhaltensverträge** spezifiziert, die das Vertrauen in die Software-Komponente in sequentiellen Umgebungen erhöhen [BJP+99, 38]. Sie enthalten die Vor- und Nachbedingungen zu den Operatorensignaturen und die funktionalen Invarianten [BJP+99, 39].

Die vorletzte Ebene umfasst die **Synchronisationsverträge**, die das Vertrauen in die Software-Komponente in verteilten oder nebenläufigen Umgebungen erhöhen [BJP+99, 38]. In der Literatur zum Entwurf von Software-Komponenten werden mehrere Ansätze diskutiert, wie Anforderungen an die Synchronisation von Operatorenausführungen in der Schnittstelle einer Software-Komponente so exakt wie nötig und so verständlich wie möglich spezifiziert werden können. Sie basieren häufig auf einer speziellen Form der **temporalen Logik**, die schließlich zur Formulierung temporaler Integritätsbedingungen dient [Szyp98, 45; CoTu00, 186]. Unterschiedlich ausführliche Vergleiche dieser Ansätze sind beispielsweise in [BJP+99; Gies00; CoTu00] aufgeführt.

Auf der obersten Ebene befinden sich die **Qualitätsverträge**, die die nicht-funktionalen Invarianten einer Software-Komponente in Form von Deklarationen nicht-funktionaler Eigenschaften und Eigenschaftswerte beinhalten. Dazu gehören beispielsweise der Durchsatz und das Antwortzeitverhalten [BJP+99, 40; Gies00, 4; CHJK02, 37; CoTu00, 182]. Wie erwähnt können für nicht-funktionale Eigenschaften mit kontinuierlichen Wertebereichen auch einzuhaltende Ausprägungsintervalle angegeben werden. Daraus lässt sich bereits erkennen, dass die Vertragsinhalte auf dieser Ebene üblicherweise verhandelbar sind [BJP+99, 38].

Die Aufteilung in vier Vertragsebenen findet inzwischen breite Akzeptanz als allgemeines Modell für Software-Komponenten-Verträge [CoTu00, 181; MeMi99, 36; Gies00, 4]. Deshalb wird sie als ein zentraler Bestandteil in das Grundkonzept der Komponentenorientierung aufgenommen.

Spezifikation von Zusicherungen

Zur Beschreibung der Invarianten und der Vor- und Nachbedingungen wird eine standardisierte und möglichst formale Sprache verwendet. Allerdings wäre eine streng formale Sprache, mit der man Zusicherungen aller Art ausdrücken könnte, unter anderem schwer erlernbar und kaum praktikabel [Meye90, 169; Szyp98, 47 f; CoTu00, 183]. Deswegen wird im softwaretechnischen Entwurf eine Sprache benötigt, mit der sich Zusicherungen nur soweit formalisieren lassen, wie sie intuitiv verständlich bleiben. Alle darüber hinausgehenden Zusicherungsspezifikationen werden dann informell als Kommentare notiert [Meye90, 169]. Somit sind die Zusicherungen in der Typbeschreibung einer Software-Komponente sowohl für den Nutzer als auch für den Ersteller verständlich und nützlich [Meye90, 169]. In Abschnitt 5.2.2.3 wird eine Zusicherungssprache vorgestellt, die diesen Anforderungen genügt.

Spezifikation des Verhaltens aus der Innensicht

Aus der Innensicht wird das Verhalten einer Software-Komponente im softwaretechnischen Entwurf anhand der Beziehungen zwischen den Klassen im Implementierungsteil und den zugeordneten Nachrichtentypen beschrieben. Dabei entsprechen die Nachrichtentypen den Operatorensignaturen der Klassen. Die Operatoren zu den Operatorensignaturen sind, wie erwähnt, gemeinsam mit den Attributen in den Klassen gekapselt. Die Realisierung der deklarierten nicht-funktionalen Eigenschaften und Eigenschaftswerte erfolgt nicht im Implementierungsteil der Software-Komponente, sondern durch geeignete Basismaschinendienste. Auch die Verwaltungsschnittstelle wird nicht im Implementierungsteil der Software-Komponente realisiert, sondern im Implementierungsteil einer so genannten **Software-Komponenten-Fabrik**, mit der sich unter anderem der nächste Abschnitt befasst. Somit wird eine konsequente Trennung von fachlichen und technischen Belangen erreicht.

Ein spezielles Attribut, das jede Software-Komponente besitzt, ist die **Identität**. Jeder Instanz einer Software-Komponente wird bei ihrer Erzeugung durch die Software-Komponenten-Fabrik ein Identitätswert zugeordnet, der zwar ermittelbar, aber unveränderbar ist. Anhand dieses Identitätswerts lassen sich Software-Komponenten-Instanzen stets unterscheiden, auch wenn sie beispielsweise den gleichen Namen und den gleichen Zustand aufweisen.

Zur Spezifikation des Verhaltens stehen auch wiederverwendbare Konzepte in Form von Entwurfsmustern zur Verfügung. Folglich kann auch in diesem Fall das *Development for Reuse* durch eine generative Wiederverwendung mit Mustern ergänzt werden.

4.3.3 Strukturmerkmale einer Software-Komponenten-Instanz im softwaretechnischen Entwurf

Eine **Software-Komponenten-Instanz** wird über die Verwaltungsschnittstelle der Software-Komponente von einer singulären Instanz der bereits erwähnten Software-Komponenten-Fabrik erzeugt, die unabhängig von den Lebensläufen der erzeugten und zu erzeugenden Software-Komponenten-Instanzen existieren muss. Neben der Erzeugung von neuen Software-Komponenten-Instanzen ist sie auch für das Auffinden, Passivieren, Reaktivieren und Zerstören der bereits erzeugten Software-Komponenten-Instanzen verantwortlich und implementiert folglich die entsprechenden Operatorensignaturen der Verwaltungsschnittstelle.

Aus der Außensicht weist eine Software-Komponenten-Instanz einen Namen, eine fachliche Schnittstelle, eine Verwaltungsschnittstelle und eine Deklaration nicht-funktionaler Eigenschaften und Eigenschaftswerte auf. Die Schnittstellen und die Deklaration entsprechen denen der Software-Komponente. Aus der Innensicht enthält

eine Software-Komponenten-Instanz, analog zur Software-Komponente, einen Implementierungsteil, der somit getrennt von den bisher genannten Strukturelementen ist und auf den nur über die fachliche Schnittstelle zugegriffen werden kann. Er wird als Objektstruktur beschrieben, die der Klassenstruktur der zugehörigen Software-Komponente entspricht. Demzufolge stimmen auch die öffentlichen Operatoren und Operatorensignaturen der Objekte in der Objektstruktur mit den Operatoren und Operatorensignaturen der Software-Komponenten-Instanz überein.

Die Kurzbeschreibung des fachlichen Belangs ist, wie bereits erwähnt, nur für Dokumentationszwecke bestimmt und somit kein Bestandteil einer Software-Komponenten-Instanz in der Ausführung.

4.3.4 Verhaltensmerkmale einer Software-Komponenten-Instanz im softwaretechnischen Entwurf

Eine Software-Komponenten-Instanz im softwaretechnischen Entwurf bildet den in der Software-Komponente beschriebenen fachlichen Belang ab, indem sie sich entsprechend der Typspezifikation der zugehörigen Software-Komponente verhält. Dazu gehören einerseits die Operatorensignaturen einschließlich der deklarierten Ausnahmen, der beigefügten Vor- und Nachbedingungen und der funktionalen Invarianten, die in den Schnittstellen festgelegt sind sowie andererseits die deklarierten nichtfunktionalen Eigenschaften und Eigenschaftswerte. Da aus der Außensicht der Implementierungsteil einer Software-Komponenten-Instanz nicht einsehbar ist, kann das Verhalten der Software-Komponenten-Instanz nur aus der Typspezifikation und der fachlichen Kurzbeschreibung der Software-Komponente abgelesen werden. Aus der Innensicht wird das Verhalten der Operatoren anhand der Sequenz von Nachrichten, die über die Beziehungen der Objektstruktur laufen, beschrieben.

Außerdem verfügt eine Software-Komponenten-Instanz über einen Identitätswert und einen Zustand, der durch die aktuellen Attributwerte definiert ist. Der Zustand kann über die fachliche Schnittstelle durch entsprechende Lese- und Schreiboperatoren abgefragt und verändert werden. Die Identität, die die Software-Komponenten-Instanz bei ihrer Erzeugung erhält, ist anhand von speziellen Operatoren, die sich über die Verwaltungschnittstelle aufrufen lassen, zwar ermittelbar, aber nicht änderbar.

4.4 Software-Komponente in der Implementierung

Die Spezifikation einer Software-Komponente im softwaretechnischen Entwurf wird in der Implementierung in Quellcode umgesetzt und dabei konkretisiert. Darüber hinaus realisieren spezielle Basismaschinen die im softwaretechnischen Entwurf spezifizierten Dienste, die für die Wiederverwendbarkeit und Verteilbarkeit von Software-Komponenten und komponentenbasierten Anwendungssystemen notwendig sind.

Alle Basismaschinen, die zusammen die Realisierung und die Ausführung komponentenbasierter Anwendungssysteme gewährleisten, werden in einer **Entwicklungs- und Ausführungsplattform für komponentenbasierte Anwendungssysteme** zusammengefasst.

Eine Software-Komponente in der Implementierung beschreibt ebenfalls die Struktur und das Verhalten ihrer Instanzen. Die Beziehungen zwischen einer Software-Komponente und einem komponentenbasierten Anwendungssystem in der Implementierung entsprechen denen im softwaretechnischen Entwurf.

4.4.1 Strukturmerkmale einer Software-Komponente in der Implementierung

Da eine Software-Komponente in der Implementierung eine konkretisierte Beschreibung ihres softwaretechnischen Entwurfs darstellt, entsprechen ihre Strukturmerkmale im Wesentlichen denen des softwaretechnischen Entwurfs. Somit besteht eine Software-Komponente in der Implementierung ebenfalls aus einem Typspezifikationsteil, bestehend aus einem fachlich bestimmten Namen, einer Verwaltungsschnittstelle, einer fachlichen Schnittstelle und einer Deklaration von nicht-funktionalen Eigenschaften und Eigenschaftswerte sowie einem Implementierungsteil. Außerdem verfügt sie, wiederum nur zu Dokumentationszwecken, über eine Kurzbeschreibung des zugehörigen fachlichen Belangs. Die Inhalte der genannten Teile stimmen mit denen des softwaretechnischen Entwurfs überein, mit dem Unterschied, dass sie in codierter Form vorliegen. Die Verwendung einer standardisierten Codierungssprache erhöht die intuitive Verständlichkeit und die Nachvollziehbarkeit der Software-Komponente. Außerdem erleichtert eine standardisierte Codierungsform auch das Auffinden einer Software-Komponente bei ihrer Wiederverwendung.

Auch in der Implementierung ist der Implementierungsteil aus der Außensicht in der Software-Komponente gekapselt und nur über die codierten Schnittstellen erreichbar. Dadurch bleibt das verwendete Programmierparadigma und die verwendete Programmiersprache vor dem Nutzer verborgen. So ist es dem Nutzer außerdem nicht möglich, zur Anpassung oder Erweiterung der Software-Komponente direkt in das Programm einzugreifen und dieses zu ändern. Die einzigen Möglichkeiten zur Änderung einer Software-Komponente aus Sicht des Nutzers stellen die codierten Schnittstellen und die ebenfalls codierten Deklarationen der nicht-funktionalen Eigenschaften und Eigenschaftswerte dar. Auf diese Weise realisiert eine Software-Komponente in der Implementierung das Prinzip der Kapselung und das Prinzip der Trennung von Schnittstelle und Implementierung, welche beide das Geheimnisprinzip verwirklichen, und kann somit als Black Box wiederverwendet werden.

Der Ersteller einer Software-Komponente muss, zusätzlich zu der Codierung der Schnittstellen, der Deklaration und der Beschreibung, den im softwaretechnischen

Entwurf spezifizierten Implementierungsteil programmieren. Da die softwaretechnische Spezifikation des Implementierungsteils auf dem Grundkonzept der Objektorientierung basiert, kann zu seiner Programmierung eine der zahlreichen objektorientierten Programmiersprachen verwendet werden. Diese objektorientierte Programmiersprache wird von der Entwicklungs- und Ausführungsplattform für komponentenbasierte Anwendungssysteme zur Verfügung gestellt. Die Realisierung des Implementierungsteils der Software-Komponenten-Fabrik sowie der deklarierten nicht-funktionalen Eigenschaften und Eigenschaftswerte übernehmen, wie erwähnt, entsprechende Basismaschinendienste, die ebenfalls von der Entwicklungs- und Ausführungsplattform angeboten werden.

Zur Implementierung von Software-Komponenten können zusätzlich Implementierungsmuster bzw. Idiome verwendet werden (vgl. Abschnitt 3.1.3.6). Auf diese Weise wird auch das *Development for Reuse* in der Implementierung mit einer generativen Wiederverwendung anhand von Mustern unterstützt.

4.4.2 Verhaltensmerkmale einer Software-Komponente in der Implementierung

Das Verhalten einer Software-Komponente wird aus der Außensicht, entsprechend dem softwaretechnischen Entwurf, anhand der Typspezifikation beschrieben. Dazu gehören zum einen die Operatorensignaturen, die deklarierten Ausnahmen, die Vor- und Nachbedingungen und die funktionalen Invarianten in den Schnittstellen sowie zum anderen die nicht-funktionalen Eigenschaften und Eigenschaftswerte in der entsprechenden Deklaration. Die Relevanz dieser letztgenannten Deklaration ist in der Implementierung besonders deutlich erkennbar, da eine Software-Komponente aus Gründen der Verteilbarkeit und Wiederverwendbarkeit erst zur Ausführungszeit in ein Anwendungssystem eingebunden wird und zu diesem Zeitpunkt keine Integrations- oder Modultests mehr durchführbar sind [Gies00, 1 f].

Die Zusicherungen und die deklarierten Ausnahmen in den Schnittstellen liegen ebenfalls in codierter Form vor. Auch in diesem Fall erhöht eine standardisierte Codierungssprache die Verständlichkeit und Nachvollziehbarkeit der Software-Komponente.

Aus der Innensicht wird das Verhalten einer Software-Komponente durch ein objektorientiertes Programm im Implementierungsteil ausgedrückt. Für die Realisierung der deklarierten nicht-funktionalen Eigenschaften und Eigenschaftswerte und der im softwaretechnischen Entwurf spezifizierten Software-Komponenten-Fabrik stehen entsprechende Dienste der erwähnten Entwicklungs- und Ausführungsplattform zur Verfügung.

Analog zur Software-Komponente im softwaretechnischen Entwurf weist auch eine Software-Komponente in der Implementierung ein spezielles Attribut auf, mit dem die verschiedenen Instanzen der Software-Komponente identifiziert werden können.

4.4.3 Strukturmerkmale einer Software-Komponenten-Instanz in der Implementierung

Analog zum softwaretechnischen Entwurf wird eine Software-Komponenten-Instanz über die Verwaltungsschnittstelle der Software-Komponente von einer singulären Instanz der Software-Komponenten-Fabrik erzeugt. Diese wird wiederum von einem speziellen Dienst der Entwicklungs- und Ausführungsplattform für komponentenbasierte Anwendungssysteme erzeugt und bereitgestellt. Für die Interoperabilität und Portabilität von Software-Komponenten-Instanzen in der Implementierung sorgt ebenfalls ein entsprechender Dienst der Entwicklungs- und Ausführungsplattform.

Die Strukturmerkmale einer Software-Komponenten-Instanz in der Implementierung gleichen denen im softwaretechnischen Entwurf. Allerdings liegen alle Teile während der Ausführung nicht mehr in Quellcodeform, sondern in maschinenverarbeitbarer Codeform vor.

4.4.4 Verhaltensmerkmale einer Software-Komponenten-Instanz in der Implementierung

Da eine Software-Komponenten-Instanz in der Implementierung vollständig maschinenverarbeitbar codiert ist, lässt sich ihr Verhalten, wie auch schon im softwaretechnischen Entwurf, nur aus den Operatorensignaturen einschließlich der deklarierten Ausnahmen, der Vor- und Nachbedingungen und der Invarianten, die in den Schnittstellen der Software-Komponente festgelegt sind, erkennen. Zusätzlich bestimmen die nicht-funktionalen Eigenschaften und Eigenschaftswerte in der entsprechenden Deklaration der Software-Komponente das Gesamtverhalten der Software-Komponenten-Instanz.

Ein Zugriff auf eine Operatorsignatur in einer Schnittstelle wird von einem speziellen Basismaschinendienst der Entwicklungs- und Ausführungsplattform an den entsprechenden Operator im Implementierungsteil weitergeleitet, der daraufhin ausgeführt wird. Somit erfolgt die Auswahl der Operatorimplementierung erst zur Zeit des Aufrufs. Dies entspricht dem dynamischen bzw. späten Binden im Grundkonzept der Objektorientierung.

Ein weiterer Basismaschinendienst kontrolliert außerdem die Einhaltung der semantischen Integritätsbedingungen, die in den Schnittstellen der Software-Komponente als codierte Zusicherungen vorliegen.

Darüber hinaus weist die Software-Komponenten-Instanz eine Identität und einen Zustand auf, der durch die aktuellen Attributwerte definiert ist. Der Zustand ist über die fachlichen Operatoren, auf die über die fachliche Schnittstelle zugegriffen werden kann, abfragbar und veränderbar. Die Identität wird der Software-Komponenten-Instanz von der singulären Instanz der Software-Komponenten-Fabrik zugewiesen und ist ermittelbar, aber nicht veränderbar.

4.5 Komponentenbasiertes Anwendungssystem im softwaretechnischen Entwurf

Ein **komponentenbasiertes Anwendungssystem** ist ein verteiltes, objektintegriertes System auf der Basis von Software-Komponenten. Auf diese Weise wird die Erfüllung der in Abschnitt 3.2.3.1 genannten Integrationsziele bestmöglich unterstützt.

Ferner sind die Software-Komponenten, aus denen ein komponentenbasiertes Anwendungssystem besteht, wiederverwendbar. Bei der Entwicklung eines komponentenbasierten Anwendungssystems werden sowohl im softwaretechnischen Entwurf als auch in der Implementierung verfügbare Software-Komponenten wiederverwendet und fehlende Software-Komponenten neu erstellt. Folglich handelt es sich dabei sowohl um ein *Development with Reuse* als auch um ein *Development for Reuse*. Dies führt zu einem Paradigmenwechsel in der Anwendungssystementwicklung von *Make or Buy* zu *Make and Buy*.

Die folgenden Ausführungen betrachten den softwaretechnischen Entwurf eines komponentenbasierten Anwendungssystems mittels verfügbarer Software-Komponenten (*Development with Reuse*). Der softwaretechnische Entwurf von Software-Komponenten selbst (*Development for Reuse*) wurde in den letzten Abschnitten ausführlich behandelt.

Aus der Charakterisierung eines komponentenbasierten Anwendungssystems als verteiltes, objektintegriertes System, das aus wiederverwendbaren Software-Komponenten besteht, folgen bestimmte Struktur- und Verhaltensmerkmale eines komponentenbasierten Anwendungssystems im softwaretechnischen Entwurf und in der Implementierung.

In Abschnitt 3.2 wird ein verteiltes System aus der Außensicht als ein integriertes System beschrieben, das zur Erfüllung einer Gesamtaufgabe gemeinsame Ziele verfolgt. Diese Charakterisierung erfasst bereits die aus der Außensicht wesentlichen Struktur- und Verhaltensmerkmale eines komponentenbasierten Anwendungssystems und seiner Instanzen im softwaretechnischen Entwurf, wie auch in der Implementierung. Deshalb konzentrieren sich die folgenden Erläuterungen ausschließlich auf die Innensicht eines komponentenbasierten Anwendungssystems und seiner Instanzen.

Zur Gewährleistung der Verteilbarkeit der enthaltenen Software-Komponenten benötigt das Anwendungssystem ebenfalls Dienste geeigneter Basismaschinen. Diese Dienste werden wiederum, wie das Anwendungssystem selbst, im softwaretechnischen Entwurf spezifiziert und in der Implementierung realisiert.

Aufgrund des erläuterten Zusammenhangs zwischen dem softwaretechnischen Entwurf und der Implementierung folgt zunächst die Erläuterung der Struktur- und Verhaltensmerkmale eines komponentenbasierten Anwendungssystems in der erstgenannten Phase. Daraufhin werden die entsprechenden Merkmale in der Implementierung beschrieben.

4.5.1 Strukturmerkmale eines komponentenbasierten Anwendungssystems im softwaretechnischen Entwurf

Ein komponentenbasiertes Anwendungssystem weist alle in Abschnitt 3.2.1 und 3.2.3.3 genannten Strukturmerkmale eines verteilten Systems auf. Es besteht aus Software-Komponenten, die über verschiedenartige Beziehungen miteinander lose gekoppelt sind. Folglich stellen die Beziehungen Kommunikationskanäle dar, die ein entsprechender Basismaschinendienst kontrolliert.

Nicht-hierarchische und hierarchische Beziehungsarten

Als Beziehungsart stehen zum einen frei definierbare nicht-hierarchische Beziehungen zwischen verschiedenen Software-Komponenten zur Verfügung. Diese Beziehungen entsprechen der Beziehungsart Assoziation, die in Abschnitt 3.4.1.2 vorgestellt wurde. Dementsprechend lassen sich zu einer Assoziation auch Kardinalitäten angeben, die die Minimal- und Maximalanzahl der an ihr beteiligten Software-Komponenten-Instanzen als Integritätsbedingungen formulieren.

Zum anderen stehen zur Erfüllung der Forderung nach hierarchischer Strukturierbarkeit von Software-Komponenten die beiden Grundformen der Abstraktion, die Generalisierung und die Aggregation, und deren Umkehrungen als Beziehungsarten bereit. Somit kann eine Software-Komponente zu einer anderen generalisiert oder aggregiert, bzw. in eine oder mehrere andere spezialisiert oder zerlegt werden. Da eine Software-Komponente, die wiederverwendet werden soll, nur durch ihre Außensicht beschrieben wird, stimmt eine Generalisierungs- bzw. Spezialisierungsbeziehung mit einer Ober- bzw. Untertypenbildung überein. Zu den Aggregations- bzw. Zerlegungsbeziehungen existieren verschiedene Formen, die denen des Grundkonzepts der Objektorientierung entsprechen (vgl. Abschnitt 3.5.1.1).

Die genannten Beziehungsarten stellen, zusammen mit der Parametrisierung der Operatoren über die Schnittstellen, im softwaretechnischen Entwurf eines komponentenbasierten Anwendungssystems die einzigen Möglichkeiten zur Anpassung oder Erweiterung von Software-Komponenten bei deren Wiederverwendung dar.

Somit wird eine kompositionelle Wiederverwendung von Software-Komponenten bei der Entwicklung von komponentenbasierten Anwendungssystemen unterstützt.

Behandlung von Funktions- und Datenredundanzen

Wie in Abschnitt 3.2.3.1 erläutert, ist es aus Fehlertoleranz- und Performanzgründen sinnvoll, bestimmte Funktions- und Datenredundanzen, die aus der Verteilung des Anwendungssystems entstanden sind, zu erhalten. Zur Sicherstellung der Konsistenz steht ein geeigneter Basismaschinendienst zur Verfügung. Für die Vermeidung von Funktions- und Datenredundanzen sorgen bestimmte Strukturmerkmale, die hier noch nicht erläutert werden können, da sie wiederum auf bestimmten Verhaltensmerkmalen beruhen.

Horizontale Trennung von Belangen

Ein komponentenbasiertes Anwendungssystem ist horizontal unterteilt in **Software-Komponenten** für die **Anwendungsfunktionen,** für die **Datenhaltung** und für die **Kommunikation mit Personen und maschinellen Aufgabenträgern**. Dies entspricht einer Strukturierung nach dem ADK-Strukturmodell, das in Abschnitt 3.4.3 erläutert wurde. Abbildung 4.7 veranschaulicht diese Strukturierung anhand eines Beispiels. Durch diese horizontale Trennung von Belangen kann die Komplexität des Anwendungssystems bewältigt und dessen Anpassbarkeit und Erweiterbarkeit unterstützt werden. Außerdem ermöglicht diese Unterteilung den Einsatz standardisierter Software-Komponenten für die Kommunikation und für die Datenhaltung. Im Zusammenhang mit der vertikalen Strukturierung in Nutzer- und Basismaschinen können auf diese Weise auch standardisierte Basismaschinen für diese Software-Komponenten eingesetzt werden. Dies erhöht unter anderem die Portabilität des Anwendungssystems.

Für den softwaretechnischen Entwurf der Struktur eines komponentenbasierten Anwendungssystems stehen zunehmend wiederverwendbare Konzepte in Form von Entwurfs- und Architekturmustern zur Verfügung. Dadurch kann eine kompositionelle Wiederverwendung von Software-Komponenten mit einer generativen Wiederverwendung anhand von Mustern kombiniert werden.

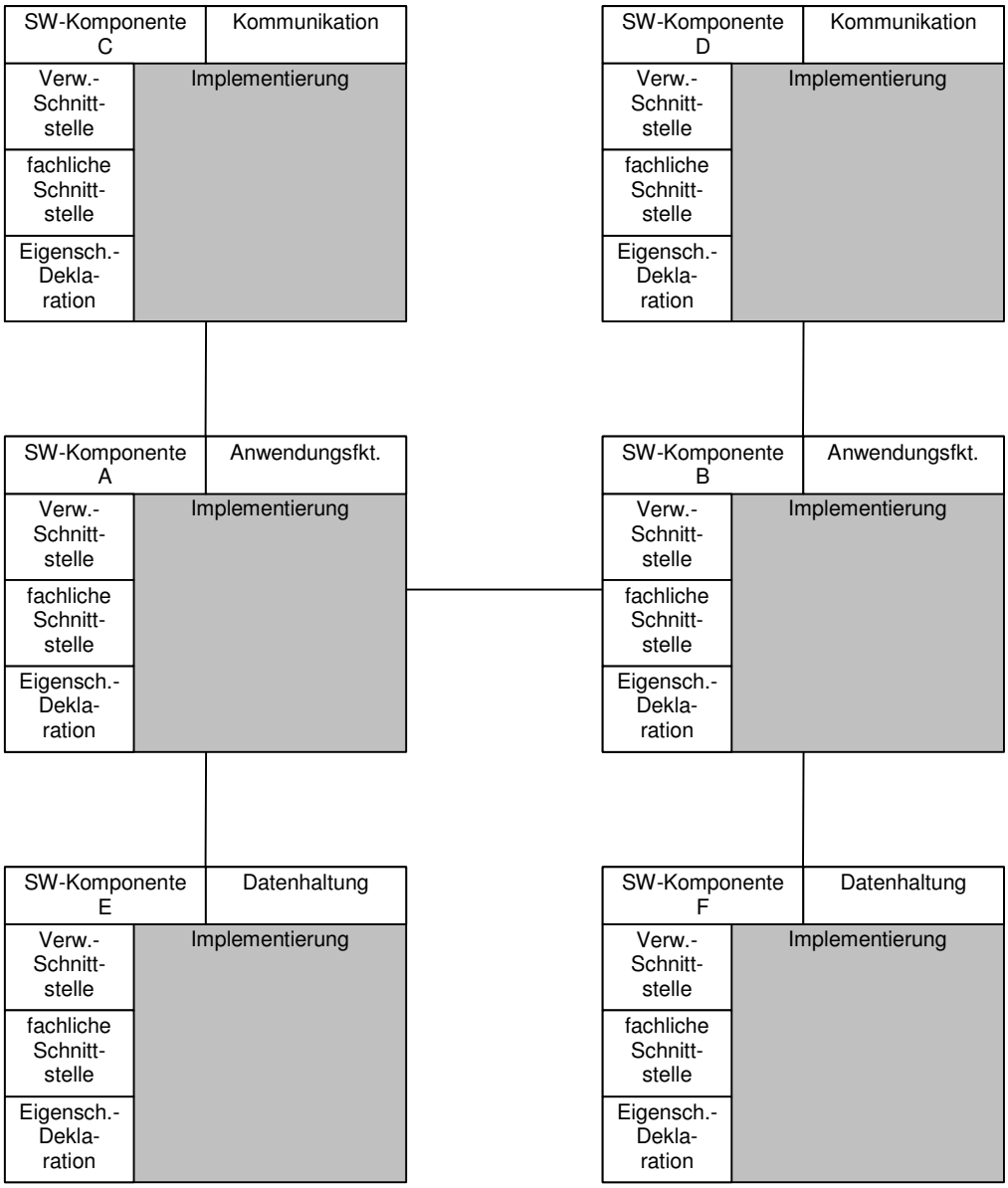


Abbildung 4.7: Exemplarische komponentenbasierte Anwendungssystemstruktur

4.5.2 Verhaltensmerkmale eines komponentenbasierten Anwendungssystems im softwaretechnischen Entwurf

Das Verhalten eines komponentenbasierten Anwendungssystems ergibt sich aus dem Verhalten der Software-Komponenten des Anwendungssystems, die miteinander in Beziehung stehen. Das Verhalten einer Software-Komponente wird anhand ihrer Typspezifikation beschrieben.

Die Beziehungen zwischen den Software-Komponenten stellen Kommunikationskanäle dar, über die die Software-Komponenten-Instanzen gemäß dem in Abschnitt 3.5.1.1 erläuterten Client-/Server-Prinzip zielbezogen zur Erfüllung einer zugeordneten Gesamtaufgabe kooperieren. Die Kooperation erfolgt über den Austausch von

Nachrichten, deren Typspezifikationen den Operatorensignaturen der beteiligten Software-Komponenten entsprechen.

Unterteilung der Software-Komponenten für Anwendungsfunktionen

Damit die Kooperation zielbezogen erfolgt, werden die Software-Komponenten für Anwendungsfunktionen in Entitäts-Software-Komponenten und Vorgangs-Software-Komponenten unterteilt. Abbildung 4.8 veranschaulicht die entsprechende Unterteilung anhand eines Ausschnitts der exemplarischen Anwendungssystemstruktur aus Abbildung 4.7.

Vorgangs-Software-Komponenten stellen Teilaufgaben der Gesamtaufgabe dar, die dem Anwendungssystem zugeordnet ist. Beziehungen zwischen den Vorgangs-Software-Komponenten bilden folglich die Zerlegungsstruktur der Gesamtaufgabe ab.

Entitäts-Software-Komponenten repräsentieren die den Teilaufgaben zugeordneten Aufgabenobjekte. Es ist aber auch möglich, dass eine Entitäts-Software-Komponente mehreren Vorgangs-Software-Komponenten zugeordnet sein kann (vgl. Abschnitt 3.2.3.2 und Abbildung 4.8). Die Entitäts-Software-Komponenten eines komponentenbasierten Anwendungssystems bilden über die oben genannten Beziehungsarten ebenfalls eine Struktur, die in der Datensicht weitgehend mit einem konzeptuellen Datenschema übereinstimmt (vgl. Abschnitt 3.2.3.2). Auf diesem Weg kann unnötige Datenredundanz vermieden und gleichzeitig die Erhaltung der operationalen Integrität ermöglicht werden. Die Sicherstellung dieser beiden Integrationsziele erfolgt durch einen geeigneten Basismaschinendienst.

Eine Anwendungsfunktion des komponentenbasierten Anwendungssystems wird folglich durch eine oder mehrere Vorgangs-Software-Komponenten und den von den Vorgangs-Software-Komponenten kontrollierten Entitäts-Software-Komponenten realisiert. Diese Form der Vorgangssteuerung gewährleistet das Erreichen der mit der Gesamtaufgabe verbundenen Ziele. Dabei sorgen die Software-Komponenten-Instanzen durch den erwähnten Nachrichtenaustausch für die Erhaltung der semantischen Integrität des gesamten Anwendungssystems.

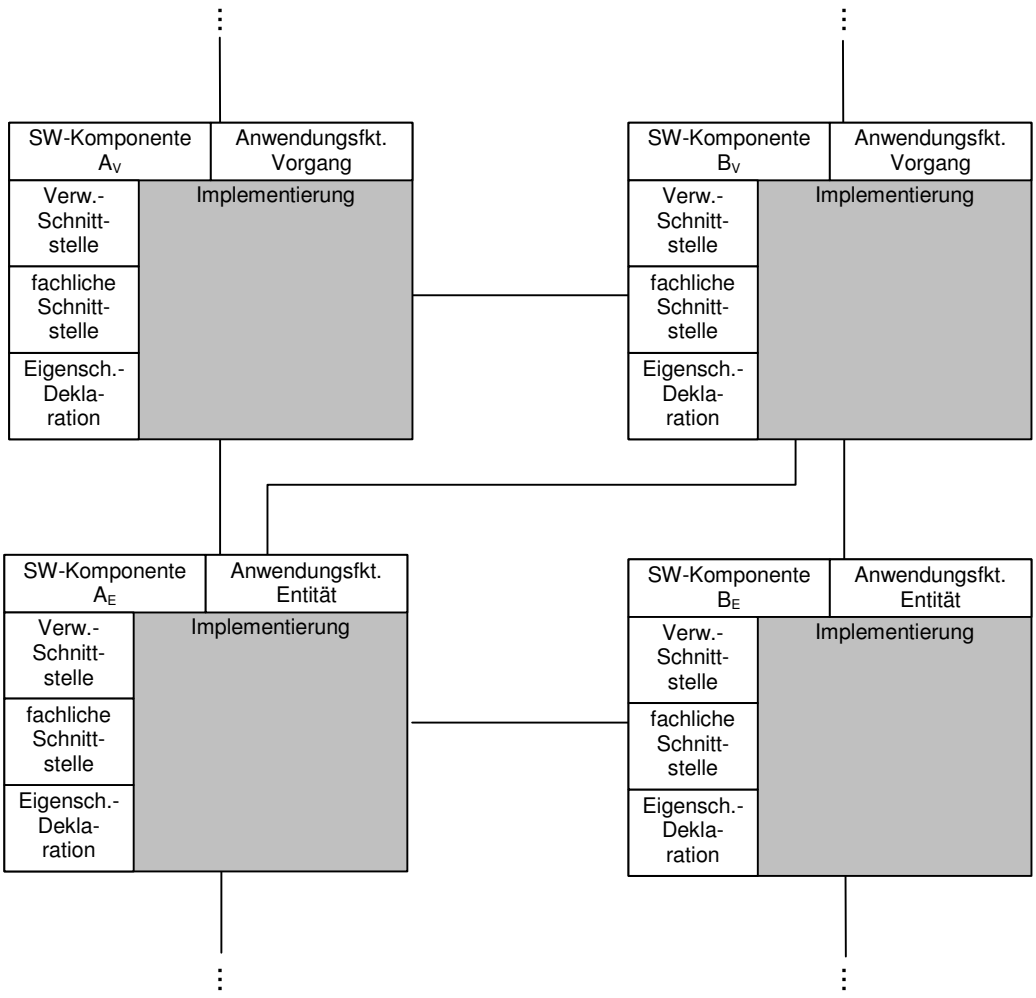


Abbildung 4.8: Exemplarische Unterteilung von Software-Komponenten für Anwendungsfunktionen in Vorgangs- und Entitäts-Software-Komponenten

Mithilfe zusätzlicher Vorgangs-Software-Komponenten lässt sich auch unnötige Funktionsredundanz vermeiden. Dafür werden Aktionen, die in Operatoren mehrerer Vorgangs-Software-Komponenten vorkommen, in einen Operator einer unterstützenden Vorgangs-Software-Komponente, auch **Unterstützungs-Software-Komponente** genannt, ausgelagert. Zwischen den Vorgangs-Software-Komponenten, aus denen die Aktionen ausgelagert wurden, und den Unterstützungs-Software-Komponenten besteht demzufolge eine Assoziationsbeziehung, die eine semantische Relation *verwendet* beschreibt.

Gewährleistung der Verteilungstransparenz

Wie bereits in Abschnitt 3.2.4 erläutert, ist die Verteilungstransparenz ein wesentliches Verhaltensmerkmal eines verteilten Systems. Sie sorgt dafür, dass die nichtfunktionalen Eigenschaften, die das Anwendungssystem aufgrund der Verteilung aufweisen muss, für den Nutzer transparent realisiert werden. Für die Gewährleis-

tung der Verteilungstransparenz sind wiederum spezielle Basismaschinendienste verantwortlich.

Wie in der Struktursicht stehen auch für den softwaretechnischen Entwurf des Verhaltens eines komponentenbasierten Anwendungssystems Entwurfs- und Architekturmuster zur Verfügung. Demnach kann auch in dieser Sicht eine kompositionelle Wiederverwendung mit einer generativen Wiederverwendung kombiniert werden.

4.5.3 Strukturmerkmale einer komponentenbasierten Anwendungssysteminstanz im softwaretechnischen Entwurf

Zur Ausführung wird aus einem komponentenbasierten Anwendungssystem eine konkrete **Anwendungssysteminstanz** erzeugt. Sie weist wiederum konkrete Ausprägungen der Struktur- und Verhaltensmerkmale auf, die durch das Anwendungssystem beschrieben worden sind. Folglich entspricht ein komponentenbasiertes Anwendungssystem einem Typ, der die Struktur- und Verhaltensmerkmale seiner konkreten Instanzen charakterisiert.

Demnach besteht die Struktur einer komponentenbasierten Anwendungssysteminstanz aus Software-Komponenten-Instanzen, die untereinander in Generalisierungs-, Aggregations- oder Assoziationsbeziehungen stehen. Über diese Beziehungen sind die Software-Komponenten-Instanzen lose gekoppelt, d. h. sie verkörpern Kommunikationskanäle zwischen den Software-Komponenten-Instanzen.

Entsprechend der Typebene können **Software-Komponenten-Instanzen** für die **Anwendungsfunktionen,** für die **Datenhaltung** und für die **Kommunikation mit Personen und maschinellen Aufgabenträgern** unterschieden werden. Sie nutzen während ihrer Ausführung spezielle Basismaschinendienste zum kontrollierten Nachrichtenaustausch und zur Erhaltung notwendiger Funktions- und Datenredundanzen.

4.5.4 Verhaltensmerkmale einer komponentenbasierten Anwendungssysteminstanz im softwaretechnischen Entwurf

Auf Instanzebene zeigt sich das Verhalten eines komponentenbasierten Anwendungssystems einerseits in Form eines Nachrichtenaustauschs zwischen den Software-Komponenten-Instanzen über die Kommunikationskanäle und andererseits in Form der dadurch ausgelösten Operatorenausführungen in den empfangenden Software-Komponenten-Instanzen.

Wie bereits im vorigen Abschnitt erläutert, können Software-Komponenten-Instanzen für die Anwendungsfunktionen, für die Datenhaltung und für die Kommunikation mit Personen und maschinellen Aufgabenträgern unterschieden werden. Erstgenannte lassen sich wiederum unterteilen in **Vorgangs-Software-Komponenten-Instanzen** und **Entitäts-Software-Komponenten-Instanzen**. Dabei führen Vorgangs-Software-

Komponenten-Instanzen Teilaufgaben durch, indem sie Operatoren auf ihrem jeweiligen Aufgabenobjekt, den Entitäts-Software-Komponenten-Instanzen, aufrufen. Zur Erhaltung der operationalen Integrität und zur Vermeidung unnötiger Datenredundanzen nutzen die Entitäts-Software-Komponenten-Instanzen geeignete Dienste der zugrundeliegenden Basismaschinen.

Für die Gewährleistung der semantischen Integrität kommunizieren die Software-Komponenten-Instanzen mittels Nachrichtenaustausch über die bestehenden Kommunikationskanäle miteinander. Dieser Nachrichtenaustausch wird grundsätzlich anhand der in Abschnitt 3.5.1.1 erläuterten Komposition mit expliziter Delegation durchgeführt. Dabei erfolgt die Delegation in Form von Weiterleiten (*Forwarding*). Demzufolge werden auch die Generalisierungs- und Aggregationsbeziehungen, deren Umkehrungen sowie die Assoziationsbeziehungen, die zwischen den Software-Komponenten-Instanzen bestehen, durch dieses Prinzip realisiert. Somit verfügt das Grundkonzept der Komponentenorientierung über keinen mit der Klassenvererbung des Grundkonzepts der Objektorientierung vergleichbaren Mechanismus. Dies garantiert eine konsequente Black-Box-Wiederverwendung und eine Vermeidung der Probleme, die durch einen Vererbungsmechanismus verursacht werden können, wie z. B. das *semantic base class*-Problem im Grundkonzept der Objektorientierung.

Die geforderte transparente Verteilung der Software-Komponenten-Instanzen stellen entsprechende Dienste der zugrundeliegenden Basismaschinen sicher.

4.6 Komponentenbasiertes Anwendungssystem in der Implementierung

Ein komponentenbasiertes Anwendungssystem in der Implementierung realisiert das entsprechende Anwendungssystem im softwaretechnischen Entwurf anhand geeigneter Codierungssprachen. Deshalb entsprechen die Struktur- und Verhaltensmerkmale eines komponentenbasierten Anwendungssystems in der Implementierung im Wesentlichen denen des komponentenbasierten Anwendungssystems im softwaretechnischen Entwurf. Die Codierungssprachen werden, wie erwähnt, zusammen mit den im softwaretechnischen Entwurf spezifizierten Diensten von einer Entwicklungs- und Ausführungsplattform für komponentenbasierte Anwendungssysteme zur Verfügung gestellt.

4.6.1 Strukturmerkmale eines komponentenbasierten Anwendungssystems in der Implementierung

Die Struktur eines komponentenorientierten Anwendungssystems in der Implementierung besteht, analog zum softwaretechnischen Entwurf, aus codierten Software-Komponenten und codierten Kommunikationsbeziehungen, über die die Software-

Komponenten lose gekoppelt sind. Dabei verkörpern die Kommunikationsbeziehungen Generalisierungs-, Aggregations- oder Assoziationsbeziehungen zwischen den Software-Komponenten. Außerdem werden, wie im softwaretechnischen Entwurf, Software-Komponenten für Anwendungsfunktionen, für die Datenhaltung und für die Kommunikation mit Personen und maschinellen Aufgabenträgern unterschieden. Die Strukturmerkmale von Software-Komponenten in der Implementierung sind bereits in Abschnitt 4.4.1 erläutert worden.

Ferner stellt die Entwicklungs- und Ausführungsplattform für komponentenbasierte Anwendungssysteme die im softwaretechnischen Entwurf spezifizierten Basismaschinendienste zur Gewährleistung eines minimalen Verknüpfungsgrades zwischen den Software-Komponenten und zur Erhaltung der notwendigen Funktions- und Datenredundanzen bereit.

Zur Unterstützung der Implementierung eines komponentenbasierten Anwendungssystems stehen zunehmend Implementierungsmuster zur Verfügung (vgl. Abschnitt 3.1.3.6). Dadurch wird auch in der Implementierung die kompositionelle Wiederverwendung von Software-Komponenten mit der generativen Wiederverwendung anhand von Mustern kombiniert.

4.6.2 Verhaltensmerkmale eines komponentenbasierten Anwendungssystems in der Implementierung

Die Verhaltensmerkmale eines komponentenbasierten Anwendungssystems in der Implementierung entsprechen denen eines komponentenbasierten Anwendungssystems im softwaretechnischen Entwurf. Danach lässt sich das Verhalten des Anwendungssystems anhand des Verhaltens der über codierte Kommunikationskanäle verbundenen Software-Komponenten beschreiben. Das Verhalten der Software-Komponenten selbst ist in den jeweiligen Typspezifikationen ausgedrückt, die in der Implementierung in codierter Form vorliegen. Zu jeder Kommunikationsbeziehung sind Nachrichtentypen definiert, die den codierten Operatorensignaturen der beteiligten Software-Komponenten entsprechen. Die Instanzen der verbundenen Software-Komponenten kooperieren gemäß dem Client-/Server-Prinzip zielbezogen zur Erfüllung einer zugeordneten Gesamtaufgabe. Auf diese Weise lässt sich auch die semantische Integrität des gesamten Anwendungssystems sicherstellen. Ferner werden die Software-Komponenten für Anwendungsfunktionen in Vorgangs- und Entitäts-Software-Komponenten unterteilt. Für die operationale Integrität der Entitäts-Komponenten und für die Vermeidung unnötiger Datenredundanzen durch die Entitäts-Software-Komponenten sorgt ein entsprechender Basismaschinendienst, der von der Entwicklungs- und Ausführungsplattform zur Verfügung gestellt wird.

Auch die im softwaretechnischen Entwurf spezifizierten Basismaschinendienste zur Gewährleistung der geforderten Verteilungstransparenz werden in der Implementierung von der Entwicklungs- und Ausführungsplattform realisiert.

Die bereits genannten Implementierungsmuster sind auch für die Unterstützung der Implementierung des Verhaltens eines komponentenbasierten Anwendungssystems verwendbar. Somit ist auch hier eine kompositionelle Wiederverwendung mit einer generativen Wiederverwendung kombinierbar.

4.6.3 Strukturmerkmale einer komponentenbasierten Anwendungssysteminstanz in der Implementierung

Die Struktur einer komponentenbasierten Anwendungssysteminstanz in der Implementierung besteht aus Software-Komponenten-Instanzen, die maschinenverarbeitbar codiert sind und über ebenfalls maschinenverarbeitbar codierte Kommunikationskanäle in Generalisierungs-, Aggregations- oder Assoziationsbeziehungen stehen. Dabei können Software-Komponenten-Instanzen für die Anwendungsfunktionen, für die Datenhaltung und für die Kommunikation mit Personen und maschinellen Aufgabenträgern unterschieden werden.

Die Software-Komponenten-Instanzen benötigen für die Kommunikation untereinander und zur Erhaltung notwendiger Funktions- und Datenredundanzen spezielle Dienste, die von der Entwicklungs- und Ausführungsplattform angeboten werden.

4.6.4 Verhaltensmerkmale einer komponentenbasierten Anwendungssysteminstanz in der Implementierung

Das Verhalten einer komponentenbasierten Anwendungssysteminstanz in der Implementierung wird bestimmt durch den Nachrichtenaustausch zwischen den maschinenverarbeitbar codierten Software-Komponenten-Instanzen über die ebenfalls maschinenverarbeitbar codierten Kommunikationskanäle und durch die Ausführungen der Operatoren in den Software-Komponenten-Instanzen aufgrund des Empfangs entsprechender Nachrichten. Bei den Software-Komponenten-Instanzen für Anwendungsfunktionen handelt es sich entweder um Vorgangs-Software-Komponenten-Instanzen oder um Entitäts-Software-Komponenten-Instanzen. Der Austausch von Nachrichten zwischen den Software-Komponenten-Instanzen über die codierten Kommunikationskanäle erfolgt durch Komposition mit explizitem Weiterleiten (*Forwarding*). Dabei sorgt ein spezieller Basismaschinendienst, der von der Entwicklungs- und Ausführungsplattform bereitgestellt wird, für den geregelten Nachrichtenaustausch.

Anhand des Nachrichtenaustauschs zwischen den Software-Komponenten-Instanzen wird auch die semantische Integrität der Software-Komponenten-Instanzen

gewährleistet. Die operationale Integrität und die Vermeidung unnötiger Datenredundanzen stellen wiederum entsprechende Basismaschinendienste sicher, die von der Entwicklungs- und Ausführungsplattform realisiert werden.

Die geforderte Verteilungstransparenz setzt sich aus mehreren Teiltransparenzen zusammen, die durch geeignete Basismaschinendienste gewährleistet werden. Auch diese werden von der Entwicklungs- und Ausführungsplattform zur Verfügung gestellt.

5 Konzeption eines komponentenbasierten Software-Architekturmodells zur Entwicklung betrieblicher Anwendungssysteme

Das Verständnis einer Software-Komponente und eines komponentenbasierten Anwendungssystems des im vorangegangenen Kapitel vorgestellten präskriptiven Grundkonzepts der Komponentenorientierung fließt nun in die Konzeption eines Software-Architekturmodells zur Entwicklung betrieblicher Anwendungssysteme ein.

Abschnitt 5.1 befasst sich zunächst mit dem grundsätzlichen Aufbau des komponentenbasierten Software-Architekturmodells. Daraufhin werden in Abschnitt 5.2 bis Abschnitt 5.4 die Modellebenen und Modellebenenbeziehungen des komponentenbasierten Software-Architekturmodells sukzessive vorgestellt. Abschnitt 5.5 beschreibt schließlich die Einordnung des komponentenbasierten Software-Architekturmodells in eine umfassende Informationssystem-Architektur am Beispiel der Unternehmensarchitektur der SOM-Methodik.

5.1 Aufbau des komponentenbasierten Software-Architekturmodells

Auf der Grundlage des im letzten Kapitel vorgestellten Grundkonzepts der Komponentenorientierung, des in Abschnitt 2.2.5.1 erläuterten Modells der Nutzer- und Basismaschine und des in Abschnitt 2.3.1 skizzierten generischen Architekturrahmens zur Gestaltung von Informationssystem-Architekturen wird nun ein Software-Architekturmodell vorgestellt, das die methodische Entwicklung komponentenbasierter betrieblicher Anwendungssysteme unterstützt.

Entsprechend den Ausführungen in Abschnitt 2.3 bestehen Architekturen im Allgemeinen aus dem Bauplan eines Systems und den Konstruktionsregeln zur Erstellung des Bauplans. Das folgende Software-Architekturmodell stellt auf den Modellebenen Konstruktionsregeln zur Verfügung, mit deren Hilfe verschiedene Software-Entwürfe betrieblicher komponentenbasierter Software-Systeme erstellt werden können. Die Konstruktionsregeln auf jeder Ebene umfassen, gemäß dem in Abschnitt 2.3.1 beschriebenen generischen Architekturrahmen, eine Metapher, ein integriertes Meta-Modell, verschiedene Sichten als Projektionen auf das integrierte Meta-Modell und entsprechende Muster. Außerdem gehören zu den Konstruktionsregeln ein Beziehungs-Metamodell, das die Meta-Objekte einer Modellebene mit den Meta-Objekten einer benachbarten Modellebene verknüpft, und, soweit vorhanden, Beziehungsmuster, die heuristisches Entwurfswissen für die Verknüpfungs- und Transformationsbeziehungen zwischen den beiden Ebenen bereitstellen.

Das Software-Architekturmodell ist als Teil eines umfassenden Architekturkonzeptes für Informationssysteme zu verstehen. Dabei kann jedes Architekturkonzept verwendet werden, das eine Ausprägung des generischen Architekturrahmens zur Gestaltung von Informationssystem-Architekturen darstellt.

5.1.1 Modellierungsziele und resultierende Teilaufgaben

Das Software-Architekturmodell unterstützt einen Entwickler bei der Spezifikation der Architektur eines zu einem Anwendungssystem gehörenden Programmsystems. Dies entspricht dem softwaretechnischen Entwurf eines Anwendungssystem, dessen generelle Schritte bereits in Abschnitt 2.2.5.3 erläutert wurden. Das generelle Formalziel des softwaretechnischen Entwurfs fordert eine eindeutige, vollständige und in der Komplexität bewältigbare Beschreibung des Programmsystems [Hamm99, 63].

Die in Abschnitt 2.2.5.3 erläuterten generellen Schritte des softwaretechnischen Entwurfs müssen hinsichtlich des generischen Architekturrahmens und der Verwendung von Software-Komponenten zu Teilaufgaben konkretisiert werden. Demzufolge wird zunächst das Programmsystem gegebenenfalls mehrstufig in eine hierarchische Anordnung von Zwischenmaschinen einschließlich zugehöriger Programmsysteme zerlegt. Die resultierenden Zwischenmaschinen mit den zugehörigen Programmsystemen bilden die Teilsysteme des zerlegten Programmsystems. Daraufhin werden für jedes Teilsystem die Software-Komponenten aus der Außensicht, deren Beziehungen untereinander und deren Schnittstellen definiert. Schließlich erfolgt die Spezifikation der Innensichten der in einem Teilsystem enthaltenen Software-Komponenten. Die Innensichten bestehen jeweils aus einem Lösungsverfahren, das das in den Schnittstellen spezifizierte Verhalten realisiert.

Sowohl bei der Zerlegung des Programmsystems in Teilsysteme als auch bei der Definition der zu jedem Teilsystem gehörenden Software-Komponenten wird die Abstraktionsform Außensichtbildung angewendet. Das führt zum einen zur Beherrschung der Komplexität des zu entwerfenden Programmsystems und der zugehörenden Software-Komponenten. Zum anderen unterstützt die Außensichtbildung die Wiederverwendung der Software-Komponenten, da beim *Development with Reuse* nur die Außensichten der Software-Komponenten erforderlich sind. Das bedeutet, dass beim *Development with Reuse* die Spezifikation der Innensichten der Software-Komponenten im softwaretechnischen Entwurf entfällt und dafür in der anschließenden Implementierung geeignete Software-Komponenten anhand ihrer Außensichtspezifikationen aufgefunden werden müssen.

Die Implementierung selbst entspricht in diesem Zusammenhang einer tatsächlichen Realisierung des bisher lediglich vorbildhaft modellierten Anwendungssystems

[LeHM95, 75; Hamm99, 64]. Da dabei die Ebene der Modellierung verlassen wird, ist sie nicht Bestandteil des Software-Architekturmodells.

5.1.2 Modellierungsmetapher

In der vorliegenden Arbeit beschreibt die im softwaretechnischen Entwurf verwendete Metapher ein betriebliches Anwendungssystem als objektintegriertes verteiltes System, das die durch die teilautomatisierte Unternehmensaufgabe vorgegebenen Ziele verfolgt. Da das Software-Architekturmodell ausschließlich im softwaretechnischen Entwurf eingesetzt wird, gilt die genannte Metapher für das gesamte Software-Architekturmodell.

Wie erwähnt, wird das Software-Architekturmodell einem umfassenden Architekturkonzept für Informationssysteme als Teilarchitektur zugeordnet. Falls die Metapher dieses umfassenden Architekturkonzepts ebenfalls ein objektintegriertes verteiltes System beschreibt, kann diese somit im softwaretechnischen Entwurf weitergeführt werden. Dies trägt, wie in Abschnitt 3.3 gezeigt, maßgeblich zur methodischen Durchgängigkeit bei der Entwicklung betrieblicher Anwendungssysteme bei.

Ein Beispiel für einen Architekturrahmen, der diese Metapher besitzt, stellt die Unternehmensarchitektur der SOM-Methodik dar (vgl. Abschnitt 2.4). Deshalb wird in Abschnitt 5.5 das komponentenbasierte Software-Architekturmodell exemplarisch in die SOM-Unternehmensarchitektur eingeordnet.

5.1.3 Modellebenen

Die Modellebenen des Software-Architekturmodells orientieren sich an den in Abschnitt 5.1.1 konkretisierten Teilaufgaben des softwaretechnischen Entwurfs. Das bedeutet, dass für jedes identifizierte Teilsystem des Programmsystems eine Teil-Architektur modelliert wird, die eine Ebene für die Außensichten und eine Ebene für die Innensichten der verwendeten Software-Komponenten und deren Beziehungen aufweist. Wie bereits in Abschnitt 2.3.3 erwähnt, sind im Zusammenhang mit Modellebenen eines Architekturmodells die Begriffe „Außenperspektive“ und „Innenperspektive“ den Begriffen „Außensicht“ und „Innensicht“ vorzuziehen, damit zwischen der Perspektive eines Modellierers, unter dem ein Anwendungssystem auf einer Modellebene beschrieben wird, und den Sichten als Projektionen auf das Metamodell einer Modellebene unterschieden werden kann.

Auf der Innenperspektive-Modellebene werden neben den Implementierungen der Software-Komponenten auch die von den Software-Komponenten benötigten Basismaschinendienste spezifiziert. Dabei erfolgt diese Spezifikation nicht für jede Software-Komponente einzeln sondern für ein komponentenbasiertes Teilsystem insgesamt.

Mit dem Software-Architekturmodell wird die softwaretechnische Perspektive eines Systementwicklers auf das zu erstellende Anwendungssystem beschrieben. Ausgangspunkt ist eine fachliche Anwendungssystemspezifikation, die die fachliche Perspektive eines Anwenders auf das zu erstellende Anwendungssystem ausdrückt. Die Lücke, die aufgrund der verschiedenen Perspektiven zwischen den beiden Modellebenen einer umfassenden Informationssystem-Architektur besteht, muss für eine methodische Durchgängigkeit der Entwicklung betrieblicher Anwendungssysteme bestmöglich überbrückt werden. Dafür ist es notwendig, dass die fachliche Anwendungssystemspezifikation in objektorientierter und objektintegrierter Form vorliegt. Falls dies erfüllt ist, werden Vorgangsklassen, die auf einem gemeinsamen Teilschema konzeptueller Klassen operieren, zu einer Vorgangs-Software-Komponente zusammengefasst. Analog dazu werden die entsprechenden konzeptuellen Klassen zu einer Entitäts-Software-Komponente zusammengefasst, soweit sie nicht zu einem weiteren Teilschema gehören, auf dem eine andere Vorgangs-Software-Komponente operiert. In Abschnitt 5.4.3 wird dieser Lösungsansatz genauer erläutert.

5.2 Modellebene der Struktur- und Verhaltensmerkmale eines komponentenbasierten Anwendungssystems aus der Außenperspektive der Software-Komponenten

Methodische Grundlage für diesen Abschnitt sind die Ausführungen zum softwaretechnischen Entwurf von Software-Komponenten und komponentenbasierten Anwendungssystemen im Grundkonzept der Komponentenorientierung.

Zur intuitiven Verständlichkeit und Nachvollziehbarkeit sollen Software-Komponenten im softwaretechnischen Entwurf, gemäß dem Grundkonzept der Komponentenorientierung, standardisiert und in geeigneter Weise formalisiert beschrieben werden. Da jedoch die im Folgenden verwendete standardisierte Beschreibungssprache ein vom Grundkonzept der Komponentenorientierung abweichendes Verständnis einer Software-Komponente und eines komponentenbasierten Anwendungssystems vertritt, muss der eigentliche **komponentenorientierte Entwurf**, der auf dem Grundkonzept der Komponentenorientierung basiert, von der standardisierten und geeignet formalisierten **Entwurfsrepräsentation** getrennt werden.

Für die Entwurfsrepräsentation stehen verschiedene standardisierte Spezifikationen der **Object Management Group (OMG)** zur Verfügung. Die OMG ist eine internationale Organisation, die 1989 gegründet wurde und inzwischen über 800 Mitglieder aus den Bereichen Software-Vertrieb, -Entwicklung und -Nutzung umfasst [OMG02a, ix]. Primäres Ziel ist die Sicherstellung der „reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments“ [OMG02a, ix]. Dafür werden von der OMG objektorientierte Industriestandards erstellt, verabschie-

det und gepflegt [Kort01, 32]. Dies geschieht in einem klar definierten Verfahren, dem so genannten Open Process, an dem sich jedes Mitglied der OMG beteiligen kann [OMG03a]. In diesem Verfahren werden Vorschläge für Spezifikationen eingebracht, verhandelt und schließlich als Standards verabschiedet. Als Rahmen für diese Spezifikationen wurde von der OMG vor kurzem die **Model Driven Architecture (MDA)** verabschiedet (vgl. Abbildung 5.1).

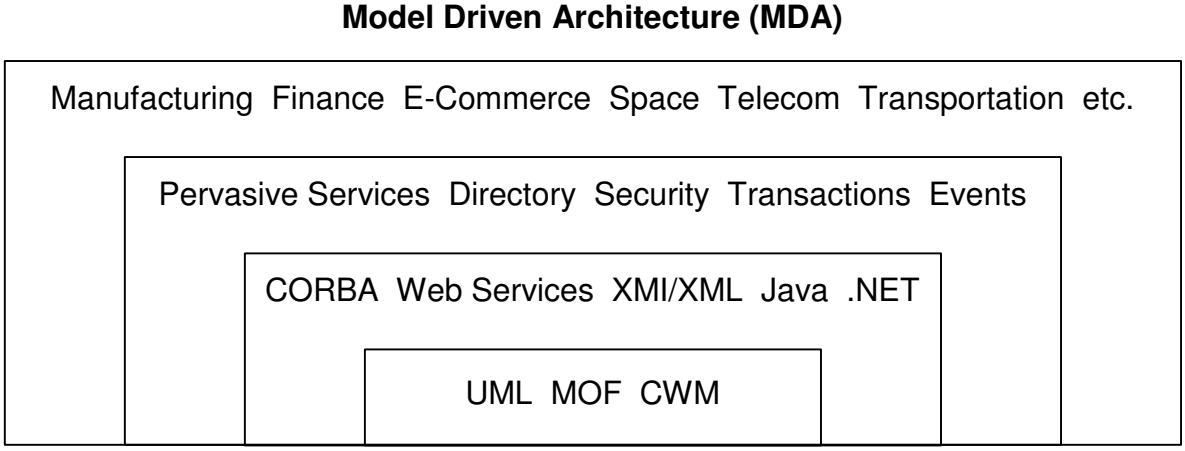


Abbildung 5.1: Model Driven Architecture der OMG (in Anlehnung an [OMG03b])

Sie stellt eine Modellierungsarchitektur dar, mit der das gesamte Vorgehen „from analysis and design, through implementation and deployment, to maintenance and evolution“ unterstützt werden soll [OMG03a]. Die Spezifikationen werden in Form von Meta-Modellen in die MDA eingebracht. Davon verspricht sich die OMG hersteller-, sprach- und middlewareunabhängige Spezifikationen [Kort91, 34].

Neben der breiten Akzeptanz der OMG und ihrer standardisierten Spezifikationen bei Herstellern, Entwicklern und Nutzern von Software-Produkten ist das oben genannte primäre Ziel der OMG ein weiterer Grund, die von ihr zur Verfügung gestellten Spezifikationen für die Repräsentation komponentenbasierter verteilter Anwendungssystementwürfe zu verwenden.

Die grafische Repräsentation komponentenbasierter Entwürfe erfolgt mithilfe der **Unified Modeling Language (UML)**-Spezifikation. Die UML-Spezifikation definiert eine grafische Notationssprache zur Entwicklung objektorientierter Software-Systeme: „The Unified Modeling Language (UML) Specification defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems“ [OMG01, xxii]. Die derzeit zuletzt verabschiedete Version trägt die Versionsnummer 1.4. Seit der Verabschiedung der ersten standardisierten Version 1.1 im Jahre 1997 sind bis zur derzeitigen Version 1.4 eine Vielzahl von Ergänzungen und Erweiterungen zur Unterstützung verschiedener Anwendungsdomänen in die UML-Spezifikation eingebracht worden [Kobr03, 1]. Dies führte dazu, dass die

UML-Spezifikation inzwischen eine hohe Komplexität, eine vielfach unpräzise Semantik, eine beschränkte Anpassbarkeit, eine nicht-standardisierte Anwendbarkeit sowie eine mangelnde Austauschbarkeit von Diagrammen aufweist [Kobr03, 1]. Außerdem vollzog die UML-Spezifikation die Weiterentwicklung des objektorientierten Paradigmas zum komponentenorientierten Paradigma nicht hinreichend nach. Zwar wurden in der Version 1.4 diesbezügliche semantische Präzisierungen berücksichtigt. Diese reichen jedoch für eine vollständige Unterstützung einer komponentenorientierten Anwendungssystementwicklung nicht aus [Kobr03, 1].

Diese Probleme sollen mit der für 2003 angekündigten Version 2.0 weitestgehend beseitigt werden. Dafür wird die UML-2.0-Spezifikation unter anderem formal in einen Teil für den Architekturkern einschließlich diverser Profile und Stereotypen, einen Teil für statische und dynamische Modellelemente, einen Teil für die formale Zusicherungssprache **Object Constraint Language (OCL)** und einen Teil für das UML-Austauschformat **XML Metadata Interchange (XMI)** gegliedert [OeWe03, 36]. Außerdem sind wesentliche Neuerungen im Bereich der Komponentenmodellierung vorgesehen [OeWe03, 37]. Diese grundlegend überarbeitete Version kann allerdings in der vorliegenden Arbeit noch nicht berücksichtigt werden, da deren offizielle Standardisierung zum Zeitpunkt der Ausarbeitung noch ausstand.

Die Metamodelle der UML werden anhand des in der **Meta-Object Facility (MOF)**-Spezifikation definierten Meta-Metamodells beschrieben. Die Beziehung zwischen dem MOF-Meta-Metamodell und der UML-Metamodelle stellt eine in der MOF-Spezifikation dargestellte **4-Ebenen-Architektur** her (vgl. Tabelle 5.1). Deren Ebenen entsprechen den in Abschnitt 2.2.3 beschriebenen Meta-Ebenen von Modellen.

Layer	**Description**	**Example**
meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.	*MetaClass, MetaAttribute, MetaOperation*
metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.	*Class, Attribute, Operation, Component*
model	An instance of a metamodel. Defines a language to describe an information domain.	*StockShare, askPrice, sellLimitOrder, StockQuoteServer*

Layer	Description	Example
user objects (user data)	An instance of a model. Defines a specific information domain.	*<Acme_SW_Share_98789>, 654.56, sell_limit_order, <Stock_Quote_Svr_32123>*

Tabelle 5.1: 4-Ebenen-Architektur der MOF-Spezifikation [OMG01, 2-4]

Für die textuelle Repräsentation der Software-Komponenten eines komponentenbasierten Entwurfs stehen die **Interface Definition Language (IDL)** und die OCL zur Verfügung. Die IDL ist Bestandteil der CORBA (Common Object Request Broker Architecture)-Spezifikation und die OCL ist in der UML-Spezifikation enthalten. Beide Sprachen werden in Abschnitt 5.2.2.3 noch genauer vorgestellt.

Die UML- und die MOF-Spezifikation bilden zusammen mit der Common Warehouse Model (CWM)-Spezifikation die Modellierungssprachen für die Metamodelle der MDA [OMG03b]. Die beiden erstgenannten Spezifikationen werden in Abschnitt 5.2.2.3 respektive Abschnitt 5.2.2.1 ebenfalls noch näher untersucht.

5.2.1 Metapher

Auf beiden Modellebenen des Software-Architekturmodells gilt die in Abschnitt 5.1.2 genannte, übergeordnete Metapher. Sie beschreibt ein betriebliches Anwendungssystem als objektintegriertes verteiltes System, das die durch die teilautomatisierte Unternehmensaufgabe vorgegebenen Ziele verfolgt. Allerdings unterscheiden sich die den Modellebenen zugrundeliegenden Spezifikationsparadigmen. Das Spezifikationsparadigma der **Außenperspektive-Modellebene** ist die Komponentenorientierung.

5.2.2 Metamodelle

In Abschnitt 2.2.3 wurde bereits erläutert, dass ein Metamodell die verfügbaren Arten von Modellbausteinen, die Beziehungsarten zwischen Modellbausteinen, die zugehörigen Verknüpfungsregeln und die Bedeutung der Modellbausteine und der Beziehungsarten festlegt.

Auf der Außenperspektive-Modellebene des Software-Architekturmodells stehen aus den genannten Gründen ein Metamodell für den komponentenorientierten Entwurf und ein zugehöriges Metamodell für die standardisierte und geeignet formalisierte Entwurfsrepräsentation zur Verfügung. Das Erstgenannte wird auch **Kern-Metamodell**, das Letztgenannte **Repräsentations-Metamodell** genannt. Beide basieren wiederum auf jeweils geeigneten Meta-Metamodellen.

5.2.2.1 Meta-Metamodell

Für das Repräsentations-Metamodell wird, wie erwähnt, die UML als Beschreibungssprache verwendet. Die in der zugehörigen Spezifikation enthaltenen Metamodelle basieren auf dem Meta-Metamodell der ebenfalls erwähnten MOF-Spezifikation. „The Meta-Object Facility (MOF) Specification defines an abstract language and a framework for specifying, constructing, and managing technology neutral metamodels“ [OMG02a, xi]. Diese wurde allerdings ursprünglich für die Erstellung von begriffsorientierten Metamodellen konzipiert. Solche **begriffsorientierten Metamodelle** beschreiben mehrstufige Generalisierungen von Begriffen, wobei Beziehungen zwischen Begriffen üblicherweise auf der jeweils höchsten Generalisierungsstufe spezifiziert sind. Diese Darstellungsform ist zwar für die grundsätzliche Definition der UML als Beschreibungssprache geeignet, jedoch nicht für Metamodelle eines dem generischen Architekturrahmen entsprechenden Architekturmodells. Zu diesem Zweck werden **systemorientierte Metamodelle** benötigt, die die Modellelemente und ihre Beziehungen in den Mittelpunkt stellen.

Deswegen wird sowohl für die Erstellung des Kern-Metamodells als auch für die Erstellung des Repräsentations-Metamodells ein Meta-Metamodell verwendet, mit dem systemorientierte Metamodelle entwickelt werden können. Dieses lehnt sich an dem in Abschnitt 2.2.3 vorgestellten Meta-Metamodell an (vgl. Abbildung 5.2). Die wesentlichen Unterschiede zwischen den beiden Meta-Metamodellen werden im Folgenden kurz erläutert.

Zum einen wird eine Aggregationsbeziehung eingeführt. Da die *besitzt*-Beziehung des Meta-Metamodells aus Abschnitt 2.2.3 in diesem Zusammenhang einer Aggregationsbeziehung mit einer gegenseitigen Existenzabhängigkeit entspricht und dies anhand von Kardinalitäten ausgedrückt werden kann, entfällt diese Beziehung.

Außerdem wird die *verbindet*-Beziehung in eine allgemeinere Assoziationsbeziehung umgewandelt, die entweder gerichtet oder ungerichtet sein kann. Zur betrachterspezifischen Definition kann eine Assoziationsbeziehung mit einem Namen versehen werden.

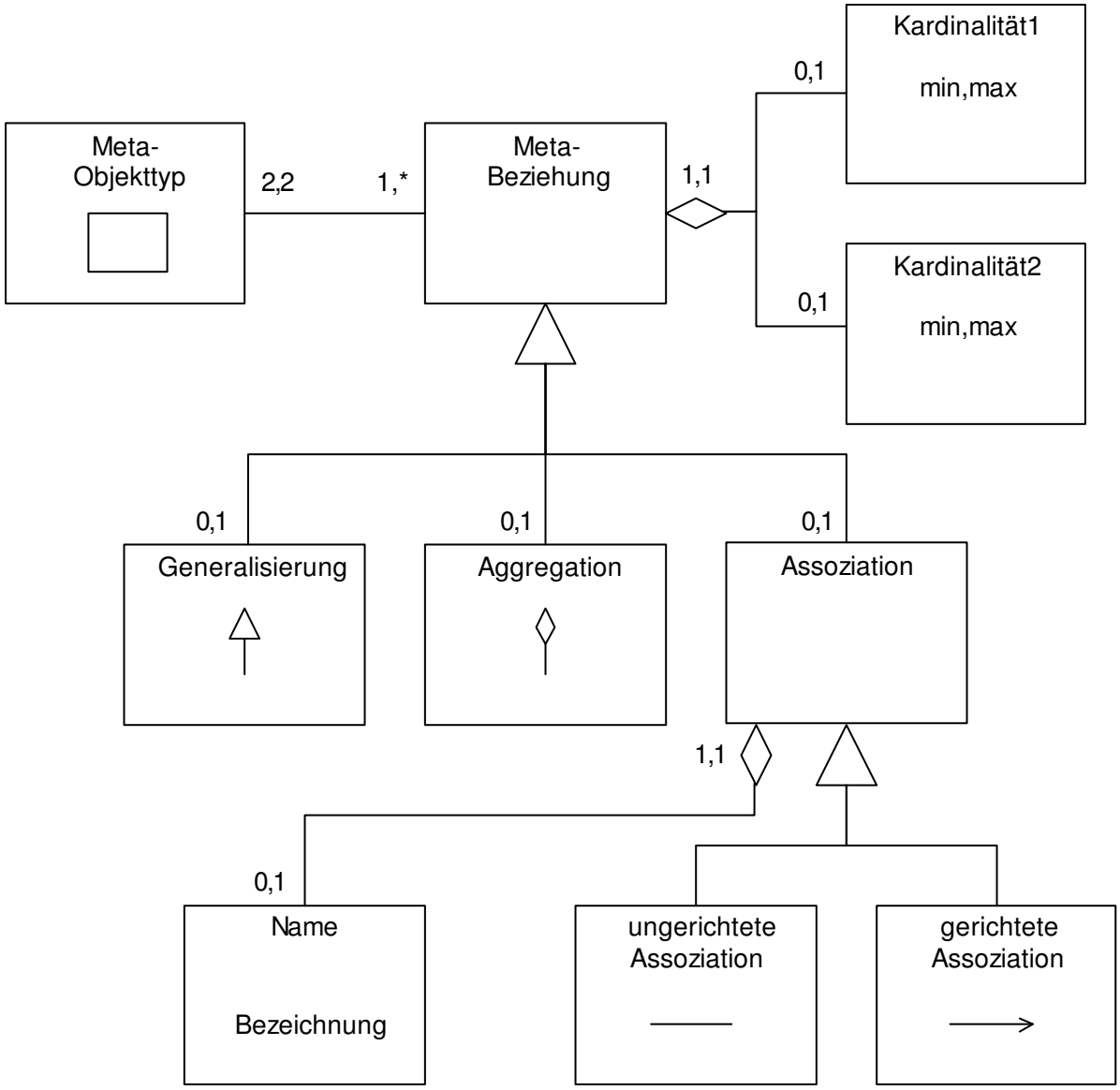


Abbildung 5.2: Meta-Metamodell des Software-Architekturmodells

5.2.2.2 Kern-Metamodell

Das **Kern-Metamodell dieser Modellebene**, das in Abbildung 5.3 dargestellt ist, beschreibt im Wesentlichen die Struktur- und Verhaltensmerkmale einer Software-Komponente aus der Außenperspektive sowie die zulässigen Arten und die minimal bzw. maximal mögliche Anzahl der zwischen den Software-Komponenten modellierbaren Beziehungen.

Eine Software-Komponente muss einen fachlich bestimmten Namen, eine fachliche Schnittstelle, eine Verwaltungsschnittstelle und eine deklarative Beschreibung nichtfunktionaler Eigenschaften und Eigenschaftswerte aufweisen. Diese Merkmale bestimmen, wie erwähnt, auch den Typ der Software-Komponente. Durch eine zusätzliche Beschreibung des fachlichen Belangs kann die Wiederverwendbarkeit der Software-Komponente weiter verbessert werden.

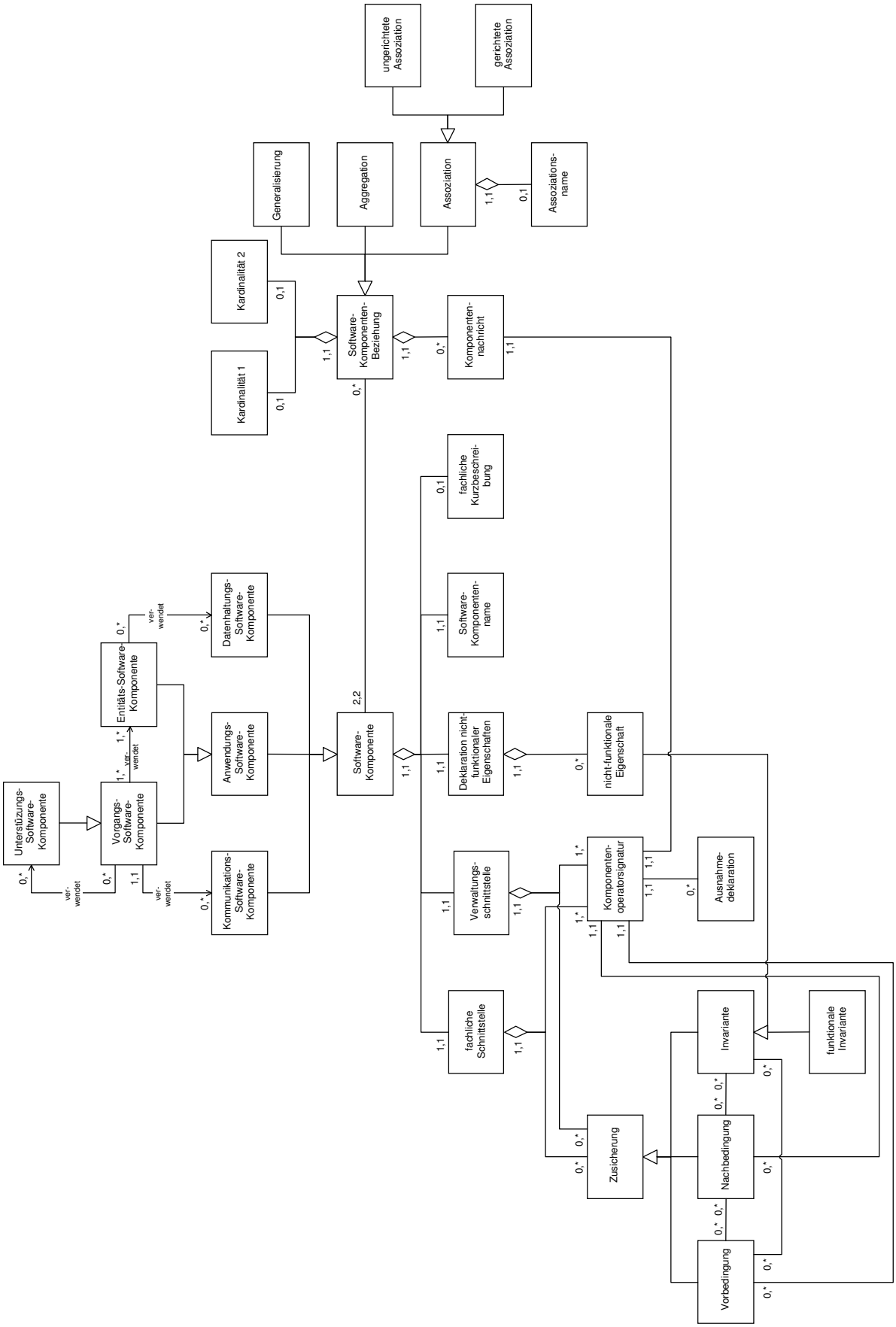


Abbildung 5.3: Kern-Metamodell der Außenperspektive-Modellebene

Sowohl die fachliche als auch die Verwaltungsschnittstelle weisen zumindest eine Operatorsignatur auf. Da nach den Ausführungen im Grundkonzept der Komponentenorientierung Operatorensignaturen bereits einfache Verträge darstellen und diese gegebenenfalls das Verhalten einfacher Operatoren hinreichend beschreiben können, ist die Angabe von zusätzlichen Zusicherungen in den Schnittstellen optional. Die verschiedenen Arten von Zusicherungen und deren Beziehungen untereinander wurden bereits im Grundkonzept der Komponentenorientierung ausführlich erläutert. Die zu den Operatorensignaturen gehörenden Ausnahmedeklarationen sind ebenfalls optional, da das Ausnahmeverhalten einfacher Operatoren bereits durch die Wahl geeigneter Rückgabewerttypen hinreichend spezifiziert werden kann.

Zwischen Software-Komponenten sind beliebig viele Generalisierungs-, Aggregations- und Assoziationsbeziehungen zulässig. Da nach den Ausführungen in Abschnitt 3.4.1.2 die Beziehungen der letztgenannten Beziehungsart in ihren jeweiligen Bedeutungen frei definierbar sind, können sie gerichtet bzw. ungerichtet sein und mit einem verwendungsspezifischen Namen versehen werden. Mit den zu einer Beziehung optional spezifizierbaren Kardinalitäten kann bestimmt werden, wieviele Instanzen der beteiligten Software-Komponenten minimal bzw. maximal in Beziehung stehen sollen bzw. dürfen. Somit stellen spezifizierte Kardinalitäten zusätzliche Integritätsbedingungen dar, deren Einhaltung kontrolliert werden muss. Über eine Beziehung, gleich welcher Art, tauschen die Instanzen der beteiligten Software-Komponenten Nachrichten aus, wobei jeder Nachrichtentyp genau einer Operatorsignatur einer beteiligten Software-Komponente entspricht.

Software-Komponenten können gemäß dem Grundkonzept der Komponentenorientierung weiter unterschieden werden in Kommunikations-, Datenhaltungs- und Anwendungs-Software-Komponenten. In der letztgenannten Klasse stehen wiederum Entitäts-, Vorgangs- und Unterstützungs-Software-Komponenten zur Verfügung. Deren Bedeutungen wurden bereits im Grundkonzept der Komponentenorientierung ausführlich erläutert. Falls eine Vorgangs-Software-Komponente zur Erfüllung der ihr zugeordneten Aufgabe mit Personen oder anderen Anwendungssystemen kommunizieren muss, werden ihr entsprechende Kommunikations-Software-Komponenten zugeordnet. Ferner werden den Entitäts-Software-Komponenten, die persistent deklarierte Attribute besitzen, entsprechende **Datenhaltungs-Software-Komponenten** zugewiesen. Dabei sind 1:1-, 1:N- oder N:M-Zuordnungen zwischen Entitäts-Software-Komponenten und Datenhaltungs-Software-Komponenten möglich. Das heißt, dass die persistent deklarierten Attribute einer Entitäts-Software-Komponente in genau einer Datenhaltungs-Software-Komponente oder in mehreren Datenhaltungs-Software-Komponenten ihre Entsprechung finden. Umgekehrt kann eine Datenhaltungs-Software-Komponente die persistent deklarierten Attribute genau einer Entitäts-Software-Komponente oder mehrerer Entitäts-Software-Komponenten auf-

weisen. Diese Freiheitsgrade bei der Zuordnung von Datenhaltungs-Software-Komponenten zu Entitäts-Software-Komponenten sind hilfreich, wenn bereits ein Datenbanksystem mit einem unternehmensweiten Datenbankschema besteht und in das komponentenbasierte Anwendungssystem integriert werden soll.

5.2.2.3 Repräsentations-Metamodelle

Damit standardisierte und hinreichend formalisierte Repräsentationsprachen die Wiederverwendbarkeit von Software-Komponenten und komponentenbasierten Anwendungssystemen unterstützen können, müssen sie zum einen unabhängig von bestimmten programmiersprachlichen Details sein [HoNo01, 244] und zum anderen sowohl die Außenperspektive als auch die Innenperspektive von Software-Komponenten repräsentieren können. Mit der Repräsentation der Außenperspektive können einem Entwickler beim *Development with Reuse* Auswahl- und Benutzungshinweise zu einer Software-Komponente an die Hand gegeben werden [HoNo01, 244]. Dafür muss sie die Verwendung einer Software-Komponente in unterschiedlichen Kontexten anhand ihrer Schnittstellen und Deklarationen aufzeigen können [HoNo01, 244]. Detaillierte Ausführungen zur Repräsentation der Innenperspektive folgen in Abschnitt 5.3.2.3. Schließlich sollte die gewählte Repräsentationssprache sowohl in der vorangehenden Phase des Fachentwurfs als auch in der nachfolgenden Phase der Implementierung eingesetzt werden können, damit auch bei der Repräsentation von Entwicklungserzeugnissen eine Durchgängigkeit gewährleistet werden kann.

Diese genannten Anforderungen können die UML und die Interface Definition Language (IDL) als standardisierte Repräsentationssprachen erfüllen. Zur Erläuterung der erforderlichen Repräsentations-Metamodelle muss wiederum unterschieden werden zwischen der Repräsentation der Struktur- und Verhaltensmerkmale einer Software-Komponente und der Repräsentation der Struktur- und Verhaltensmerkmale eines komponentenbasierten Anwendungssystems.

Repräsentation der Struktur- und Verhaltensmerkmale einer Software-Komponente aus der Außenperspektive

Für die Repräsentation der Struktur- und Verhaltensmerkmale einer Software-Komponente aus der Außenperspektive sind, wie erwähnt, insbesondere die Schnittstellen und die Deklarationen von Bedeutung. Zu diesem Zweck werden die vier Vertragsebenen einer Software-Komponente aus dem Grundkonzept der Komponentenorientierung in Abschnitt 4.3.2 herangezogen.

Danach erfolgt auf der untersten Ebene der Basisverträge die Spezifikation der Operatorensignaturen, bestehend aus einem Operatornamen, einer Parameterliste und einem Rückgabewerttyp, und der zugehörigen Ausnahmedeklarationen. Zur Gewähr-

leistung der Offenheit und Heterogenität von komponentenbasierten Anwendungssystemen, sollte dies mit einer eigenen Schnittstellenbeschreibungssprache erfolgen, die programmiersprachen- und plattformunabhängig ist [Grif98, 58].

Schnittstellenbeschreibungssprache IDL

Die IDL der CORBA-Spezifikation entspricht den genannten Anforderungen, da sie zur Spezifikation von orts-, plattform- und programmiersprachenunabhängigen Schnittstellen mit den oben angegebenen Inhalten konzipiert wurde [OMG02b, 3-2; DeRo99, 47; OrHE96, 50 f; Turo02, 5]. Darüber hinaus stellt sie einen offenen, von Forschung und Industrie in der Breite getragenen Sprachstandard dar [Turo02, 5]. Die IDL wurde als reine Deklarationssprache entwickelt und verhindert somit eine Spezifikation von Implementierungsdetails [OMG02b, 3-2; OrHE96, 50; Wesk99, 7; Grif98, 58]. Mit ihr können alle auf der Ebene der Basisverträge relevanten Schnittstelleninformationen klar erkennbar und an einer eindeutig definierten Stelle zusammengefasst werden [Grif98, 58]. Außerdem bieten IDL-Beschreibungen den geforderten „sanften“ Übergang vom softwaretechnischen Entwurf einer Software-Komponente zu ihrer Implementierung, indem sie Schnittstellen bereits so konkret beschreiben, dass sie automatisch in Implementierungsgerüste übersetzt werden können [Grif98, 58]. Somit unterstützt die IDL den direkten Weg vom Entwurf einer Software-Architektur, die unter anderem durch die Schnittstellen der Software-Komponenten definiert ist, zum kompilierten Code, der die Implementierung der spezifizierten Schnittstellen der Software-Architektur darstellt [OrHE96, 93]. Dafür müssen in der Implementierungsphase die IDL-Beschreibungen auf eine Zielsprache abgebildet werden [OMG02b, 3-2; Grif98, 58]. Diese als ***Language Mapping*** bezeichnete Aufgabe wird von einem **IDL-Compiler** durchgeführt, der aus einer IDL-Beschreibung ein **clientseitiges** und ein **serverseitiges Implementierungsgerüst** einer Software-Komponente für deren Einsatz in einem verteilten Anwendungssystem erzeugt [OrHE96, 69 ff; DeRo99, 47; Grif98, 145]. Dabei können die Zielsprachen des clientseitigen und des serverseitigen Implementierungsgerüsts auch unterschiedlich sein [DeRo99, 48]. Diese Implementierungsgerüste enthalten neben den in der Zielsprache spezifizierten Operatorensignaturen bereits generierten Code für die orts- und zugriffstransparente Verteilung der Software-Komponente [OrHE96, 69 ff; DeRo99, 47].

Die Erlernbarkeit der IDL wird dadurch unterstützt, dass in der entsprechenden OMG-Spezifikation die Grammatik der IDL anhand von **EBNF (Erweiterte Backus-Naur-Form)**-ähnlichen **Produktionsregeln** definiert wurde [OMG02b, 3-2]. Aus diesen lassen sich die folgenden wesentlichen Bestandteile einer IDL-Schnittstellenbeschreibung ableiten [OMG02b, 3-21]:

- Ein **Schnittstellenkopf**, der aus einem Schnittstellennamen und gegebenfalls dem Namen einer geerbten Schnittstelle besteht, und
- ein **Schnittstellenrumpf**, der aus ein oder mehreren Datentyp-, Konstanten-, Ausnahme-, Attributs- und Operatorsignaturdeklarationen besteht. Dabei ist eine Attributsdeklaration logisch gleichwertig mit einer Kombination aus einer Getter- und einer Setter-Operatorsignaturdeklaration [OMG02b, 3-53].

Seit der Version 3.0 der CORBA-Spezifikation enthält die IDL einen eigenen Sprachbereich für die Beschreibung des Schnittstellenangebots von Software-Komponenten [OMG02b, 3-17]. Beispielsweise sind Sprachkonstrukte zur Unterscheidung von fachlichen Schnittstellen und Verwaltungsschnittstellen vorhanden [OMG02b, 3-63 f]. Außerdem können mit weiteren Sprachkonstrukten auch Ereignisquellen und Ereignissenken zur asynchronen Kommunikationsfähigkeit von Software-Komponenten spezifiziert werden [OMG02b, 3-62 f]. Einen Überblick über diesen speziellen Sprachbereich der IDL enthält [OMG02, 3-58 ff]. Beispiele zu IDL-Schnittstellenbeschreibungen im Allgemeinen sind in den Fallstudien in Kapitel 6 enthalten.

Zusicherungsbeschreibungssprache OCL

Mit der IDL können jedoch nur syntaktische Beschreibungen von Schnittstellen durchgeführt werden. Zur Spezifikation der semantischen Verhaltenseigenschaften von Software-Komponenten, die sich auf der nächsthöheren Vertragsebene befinden, reichen die Sprachkonstrukte der IDL nicht aus [Digr98, 62; CoTu00, 181; Wesk99, 8; DeRo99, 48; OrHE96, 64]. Zu diesem Zweck wird von der OMG die in der UML-Spezifikation enthaltene Object Constraint Language (OCL) empfohlen [OMG01, 6-3; CoTu00, 183 f]. Mit ihr können unter anderem Invarianten, Vorbedingungen und Nachbedingungen formuliert werden [OMG01, 6-3]. Dabei schaffen die zur Verfügung gestellten Sprachkonstrukte der OCL den in Abschnitt 4.3.2 geforderten Kompromiss zwischen einer natürlichsprachlichen Beschreibung von Zusicherungen, die meistens mehrdeutig ist, und einer streng formalen Beschreibung von Zusicherungen, die für Personen ohne ausreichendem mathematischen Hintergrundwissen erfahrungsgemäß schwer verständlich ist [OMG01, 6-2]. Deswegen ist die OCL auch für Ungeübte intuitiv verständlich und entsprechend leicht erlernbar [Fing00, 10]. Gleichwohl sind OCL-Beschreibungen konkret genug, dass sie, analog zu IDL-Beschreibungen, automatisch auf ausgewählte Zielsprachen abgebildet werden können. Zu diesem Zweck sind inzwischen **OCL-Compiler** verfügbar, die nicht nur Quellcoderümpfe aus OCL-Beschreibungen generieren, sondern auch deren Einhaltung zur Ausführungszeit überwachen [Fing00, 15; HuDF00, 5 ff]. Genauere Ausführungen zu den Anforderungen an und die Funktionsweise von OCL-Compilern enthalten beispielsweise [Fing00; HuDF00]. Somit trägt die OCL auch zur geforder-

ten Durchgängigkeit beim Übergang vom softwaretechnischen Entwurf zur Implementierung bei.

Die Grammatik der OCL wird in der entsprechenden OMG-Spezifikation ebenfalls anhand von EBNF-ähnlichen Produktionsregeln dargelegt. Diese definieren unter anderem folgende wesentliche Bestandteile einer OCL-Beschreibung:

- Zunächst wird der **Bezug** der OCL-Beschreibung festgelegt. Dieser ist bei einer Spezifikation von Invarianten eine Software-Komponente und bei einer Spezifikation von Vor- und Nachbedingungen eine Operatorsignatur.
- Daraufhin werden ein oder mehrere auswertbare **OCL-Ausdrücke** spezifiziert. Diese müssen mit einem Schlüsselwort für die entsprechende Zusicherungsart eingeleitet werden. Ein OCL-Ausdruck kann sich auf alle in der UML definierten **Klassifizierer** beziehen. Dazu gehören beispielsweise Klassen, Schnittstellen, typisierte Assoziationen und Datentypen [OMG01, 6-12]. Darüber hinaus können in einem OCL-Ausdruck alle Eigenschaften dieser Klassifizierer, wie z. B. Attribute und Operatoren, verwendet werden [OMG01, 6-12]. Mehrere OCL-Ausdrücke können zu einem komplexen OCL-Ausdruck verknüpft werden. Außerdem stehen zur Verarbeitung von Mengen in einem OCL-Ausdruck entsprechende Operatoren zur Verfügung [OMG01, 6-22 ff].

Anwendungsnahe Beispiele zur Spezifikation von Zusicherungen mit der OCL sind in den Fallstudien in Kapitel 6 beschrieben.

Temporale Erweiterung der Zusicherungsbeschreibungssprache OCL

Die OCL ist in der vorgestellten Form wiederum nur bedingt geeignet, Synchronisationsanforderungen, die auf der zweithöchsten Vertragsebene festgelegt werden, zu beschreiben [CoTu00, 183]. Abgesehen vom Operator *@pre*, der in einer Nachbedingung eines Operators den Wert einer Eigenschaft vor der Operatorausführung zurückgibt [OMG01, 6-21], lassen sich mit der OCL nur statische Integritätsbedingungen spezifizieren [CoTu00, 186]. Damit aber einerseits auch Synchronisationsverträge standardisiert und möglichst formalisiert beschrieben werden können und andererseits die Anzahl der verwendeten Beschreibungssprachen möglichst gering bleibt, wird für die Spezifikation von Synchronisationsanforderungen in Software-Komponentenschnittstellen der Vorschlag von CONRAD und TUROWSKI zur **temporalen Erweiterung der OCL** verwendet [CoTu00].

Dieser Vorschlag differenziert zunächst die Ebene der Synchronisationsverträge in eine **Intra-Komponenten-Abstimmungs-Ebene** und eine **Inter-Komponenten-Abstimmungs-Ebene** [CoTu00, 180]: „Vereinbarungen auf der Intra-Komponenten-Abstimmungs-Ebene regeln die Reihenfolge der Verwendung von Diensten, die von *derselben* [Software-Komponente] angeboten werden. [...] Auf der Inter-

Komponenten-Abstimmungs-Ebene werden Vereinbarungen ergänzt, die Reihenfolgebeziehungen zwischen Diensten *verschiedener* [Software-Komponenten] regeln“ [CoTu00, 180 f; Hervorhebungen im Original]. Zur Spezifikation von Vereinbarungen auf diesen beiden Ebenen wird die OCL um **temporale Operatoren** konsistent erweitert. Theoretische Grundlage für diese Erweiterung ist eine **temporale Logik**, die in [Saak93, 63 ff] zur Formulierung temporaler Integritätsbedingungen in einer objektorientierten Modellierungssprache verwendet wurde [CoTu00, 187]. Es können zwei Formen temporaler Operatoren unterschieden werden: **Vergangenheitsgerichtete temporale Operatoren**, mit denen Zustände in der Vergangenheit referenziert werden können, und **zukunftsgerichtete temporale Operatoren**, die zur Spezifikation von zukünftigen Zuständen verwendet werden können. Allerdings entstehen durch die Verwendung dieser temporalen Operatoren zwei Einschränkungen die beachtet werden müssen [Turo02, 11]: Zum einen sind OCL-Ausdrücke, die temporale Operatoren enthalten, nicht unbedingt zu jedem Zeitpunkt auswertbar. Zum anderen sind in temporalen OCL-Ausdrücken auch Nicht-Abfrage-Operatoren zugelassen [Turo02, 11]. Deswegen können temporale OCL-Ausdrücke nicht unmittelbar in die Implementierungsphase übernommen werden [Turo02, 11]. Ausführliche Erläuterungen zur temporalen Erweiterung der OCL enthält die entsprechende Arbeit von CONRAD und TUROWSKI [CoTu00].

Deklaration der nicht-funktionalen Eigenschaften und Eigenschaftswerte

Auf der obersten Vertragsebene werden schließlich Qualitätsverträge spezifiziert, die die nicht-funktionalen Eigenschaften und Eigenschaftswerte der Software-Komponenten in deklarativer Form enthalten. In [Turo02] wird ein **Rahmen zur Vorgehensweise der Qualitätsspezifikation** vorgestellt. Dieser besteht aus der Festlegung eines Qualitätsordnungsrahmens, aus der Identifizierung der benötigten Qualitätskriterien, aus der Quantifizierung der festgelegten Qualitätskriterien und schließlich aus der Spezifikation der Qualitätseigenschaften (vgl. Abbildung 5.4) [Turo02, 13 ff]. Im vorliegenden Zusammenhang ist insbesondere der letztgenannte Schritt von Interesse. Da die Qualitätseigenschaften auch als formale Ausdrücke beschrieben werden können, bietet sich dafür wiederum die OCL als standardisierte Beschreibungssprache an. Somit können auch auf dieser Vertragsebene die genannten Vorteile der OCL genutzt werden.

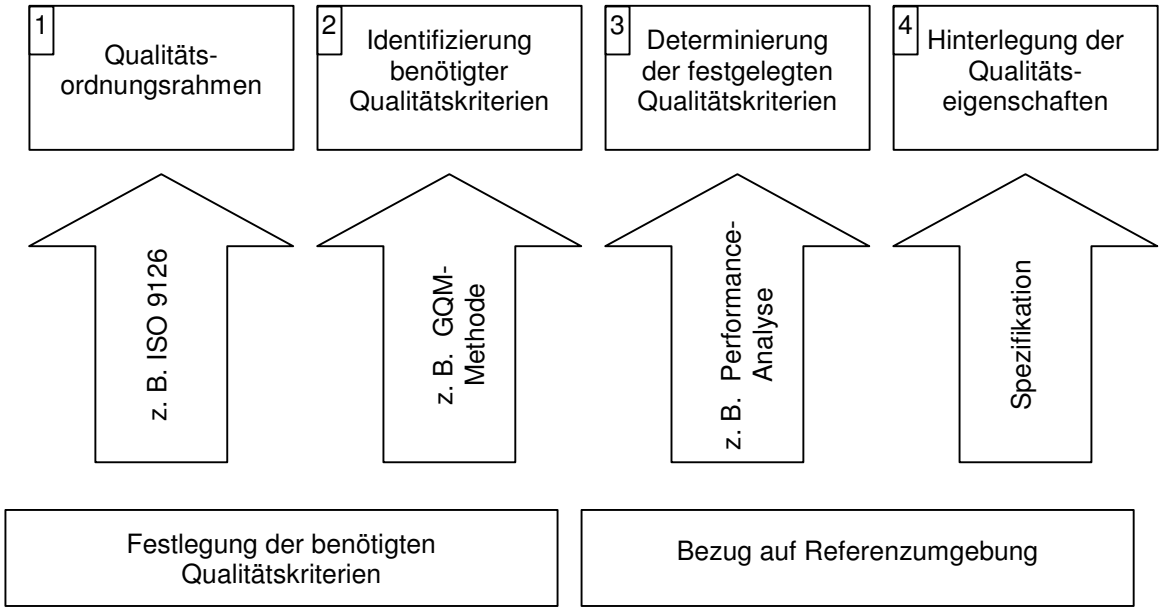


Abbildung 5.4: Schritte zur Qualitätsspezifikation (in Anlehnung an [Turo02, 13])

Kurzbeschreibung des fachlichen Belangs

Zur Repräsentation der Außenperspektive einer Software-Komponente gehört abschließend eine Kurzbeschreibung des zugehörigen fachlichen Belangs. Diese wird vom Nutzer zur intuitiven Verständlichkeit und Nachvollziehbarkeit der Software-Komponente benötigt und sollte aus diesem Grund nicht formalisiert sein. Allerdings unterstützt eine standardisierte Struktur die Erfüllung der genannten Ziele. Da eine Vorgangs-Software-Komponente einschließlich der zugehörigen Kommunikations-Software-Komponenten gemeinsam mit bestimmten Entitäts-Software-Komponenten einschließlich der entsprechenden Datenhaltungs-Software-Komponenten eine betriebliche Aufgabe beschreiben, bieten sich folgende **differenzierte Beschreibungsformen** an:

- Die Kurzbeschreibung einer **Vorgangs-Software-Komponente** enthält die Sach- und Formalziele, die Vor- und Nachereignisse und die grundsätzlichen Schritte des Lösungsverfahrens der zugeordneten betrieblichen Aufgabe. Diese ergeben sich aus den Operatoren der Vorgangs-Software-Komponente. Außerdem enthält sie Verweise auf die erforderlichen Kommunikations-Software-Komponenten und auf bestimmte Entitäts-Software-Komponenten, die die Aufgabenobjekte der betrieblichen Aufgabe repräsentieren.
- Eine **Kommunikations-Software-Komponente** wird anhand der zugeordneten Kommunikationsaufgabe beschrieben. Diese umfasst den oder die Adressaten, den Inhalt und die Form der Kommunikation.

- In der Kurzbeschreibung einer **Entitäts-Software-Komponente** sind die Attribute, deren Bedeutungen und eventuelle Existenzabhängigkeiten zu anderen Entitäts-Software-Komponenten aufgeführt.
- Analog zur Kurzbeschreibung einer Entitäts-Software-Komponente enthält die Kurzbeschreibung einer **Datenhaltungs-Software-Komponente** die Attribute, den Primärschlüssel, die referentiellen Integritätsbedingungen und die semantischen Integritätsbedingungen der zugrundeliegenden Datenbanktabelle.

Repräsentation der Struktur- und Verhaltensmerkmale eines komponentenbasierten Anwendungssystems aus der Außenperspektive

Die Repräsentation der Struktur- und Verhaltensmerkmale eines komponentenbasierten Anwendungssystems aus der Außenperspektive erfordert **Repräsentations-Metaobjekte** und **Repräsentations-Diagrammarten**, die die Struktur und das Verhalten eines komponentenbasierten Anwendungssystems beim *Development for Reuse* und beim *Development with Reuse* ausdrücken können. In der Version 1.4 der UML-Spezifikation, die dieser Arbeit zugrunde gelegt wird, sind für den softwaretechnischen Entwurf von komponentenbasierten Anwendungssystemen jedoch keine speziellen Metaobjekte oder Diagrammarten vorgesehen. Lediglich für die Implementierung steht die Diagrammart **Komponentendiagramm** mit dem zugehörigen UML-Metaobjekt Software-Komponente zur Verfügung [OMG01, 3-170 f]. In Abbildung 5.5 ist ein exemplarisches Komponentendiagramm dargestellt.

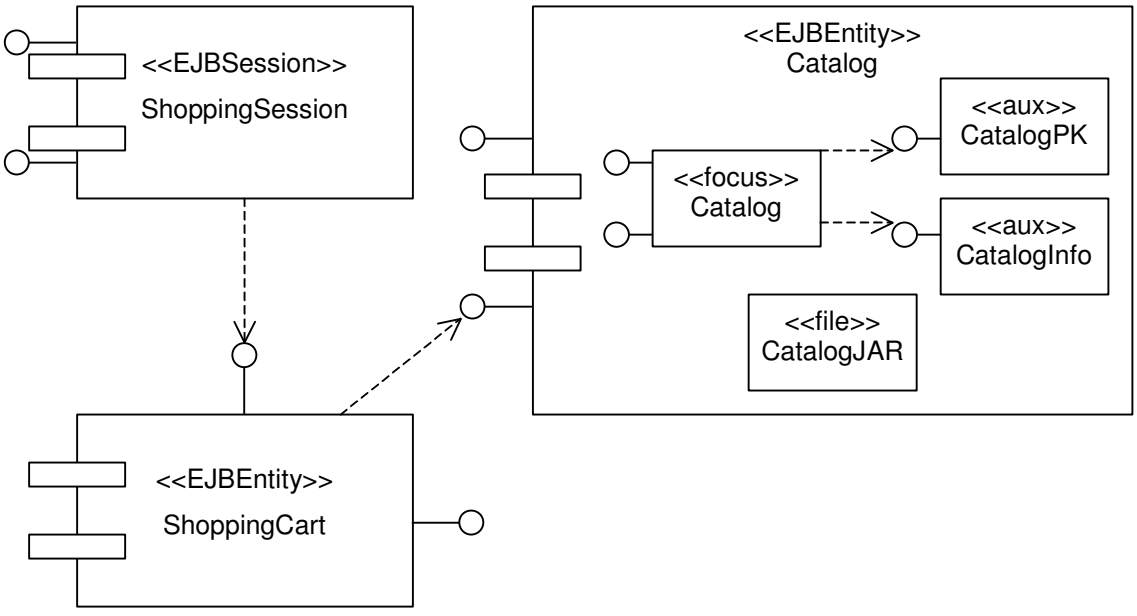


Abbildung 5.5: Exemplarisches UML-Komponentendiagramm (in Anlehnung an [OMG01, 3-171])

Repräsentations-Metaobjekte

Mit dem **Metaobjekt Software-Komponente** lässt sich eine Zusammenstellung programmiersprachlicher Klassen einschließlich zugehöriger Quellcode-, Binärcode- oder ausführbarer Dateien repräsentieren [OMG01, 3-170]. Die Darstellung von Software-Komponenten-Instanzen in Komponentendiagrammen ist nicht vorgesehen. Außerdem kann eine Software-Komponente in einem Komponentendiagramm keine eigenen Attribute oder Operatoren besitzen [OMG01, 3-171]. Die Beziehungen zwischen Software-Komponenten in einem Komponentendiagramm repräsentieren statische Abhängigkeiten, wie z. B. Übersetzerabhängigkeiten, die sich aus der Zusammensetzung von clientseitigen und serverseitigen Software-Komponenten ergeben [OMG01, 3-171]. Somit ist die Diagrammart Komponentendiagramm mit dem zugehörigen UML-Metaobjekt Software-Komponente zur Darstellung von softwaretechnischen Entwürfen komponentenbasierter Anwendungssysteme nicht geeignet.

Dafür bietet sich das **UML-Metaobjekt Subsystem** des Spezifikationsmoduls Model Management an, dessen zugeschriebene Bedeutung der einer Software-Komponente im softwaretechnischen Entwurf am nächsten kommt. Ein Subsystem als Diagrammobjekt repräsentiert eine Zusammenfassung von weiteren, zusammengehörenden Diagrammobjekten, die gemeinsam das Verhalten des Subsystems beschreiben [OMG01, 3-19]. Dafür werden die enthaltenen Diagrammobjekte unterteilt in Spezifikationsobjekte und Realisierungsobjekte. Die **Spezifikationsobjekte** bilden die Schnittstelle des Subsystems, die **Realisierungsobjekte** die zugehörige Implementierung (vgl. Abbildung 5.6) [OMG01, 3-19]. Zur speziellen Repräsentation der Außen- oder der Innenperspektiven eines Subsystems können die Realisierungsobjekte respektive die Spezifikationsobjekte auch ausgeblendet werden [OMG01, 3-20]. Ergänzend zur grafischen Repräsentation der Schnittstelle eines Subsystems durch die enthaltenen Spezifikationsobjekte kann auch eine textuelle Schnittstellenbeschreibung angegeben werden [OMG01, 3-19]. Schließlich ist auch die Darstellung von Instanzen eines Subsystems möglich [OMG01, 3-19]. Aufgrund dieser offensichtlichen semantischen Ähnlichkeiten zwischen einer Software-Komponente im softwaretechnischen Entwurf und dem UML-Metaobjekt Subsystem, hat sich die Verwendung dieses UML-Metaobjekts für die Repräsentation von Software-Komponenten im softwaretechnischen Entwurf weitestgehend durchgesetzt [Kobr00, 34; HoNo01, 246; Zimm99b, 52]. Selbst in der Version 1.4 der UML-Spezifikation wird eine exemplarische Software-Komponente im softwaretechnischen Entwurf anhand eines Subsystems dargestellt (vgl. Abbildung 5.6) [OMG01, 3-23].

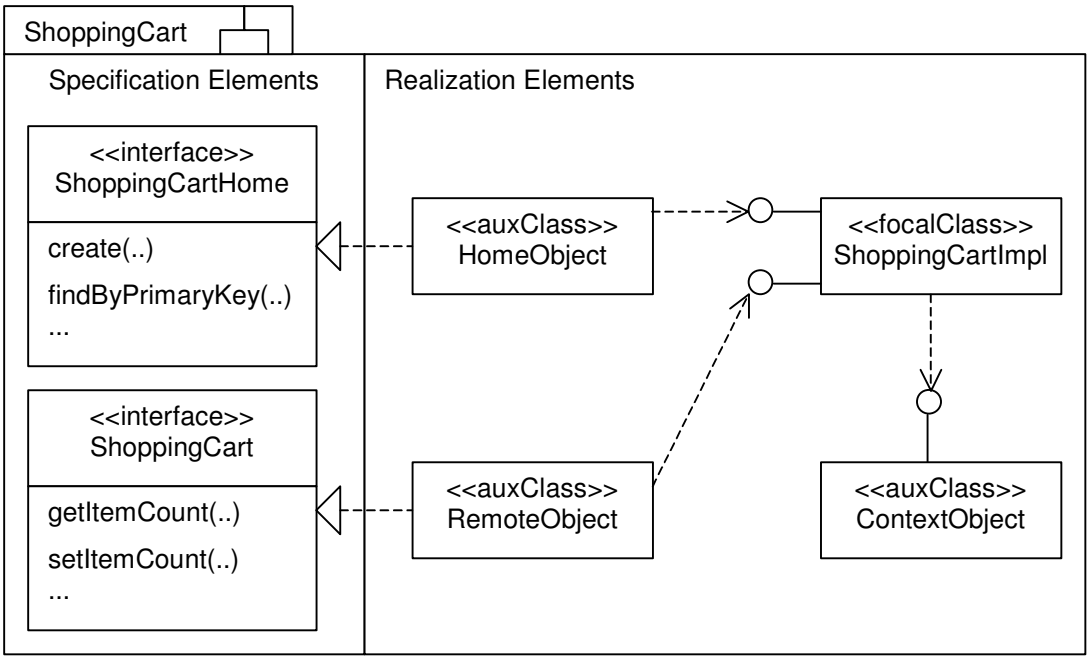


Abbildung 5.6: Repräsentation einer exemplarischen Software-Komponente anhand des UML-Metaobjekts Subsystem (in Anlehnung an [OMG01, 3-23])

Aus den bisherigen Ausführungen ist erkennbar, dass in einem UML-Diagramm auch Realisierungsbeziehungen zwischen den Diagrammobjekten Subsystem und Software-Komponente zugelassen sind. Diese zeigen dann von den Software-Komponenten zu den Subsystemen und werden mit dem Stereotyp *<<trace>>* (englisch für „zurückführen auf“) versehen [HoNo01, 251].

Repräsentations-Diagrammarten

Da die Merkmale der UML-Metaobjekte Subsystem und Subsysteminstanz analog zu den Merkmalen der UML-Metaobjekte Klasse und Objekt sind, stehen zur Repräsentation eines komponentenbasierten Anwendungssystems im softwaretechnischen Entwurf die Diagrammarten Klassendiagramm und Interaktionsdiagramm mit den UML-Metaobjekte Subsystem und Subsysteminstanz zur Verfügung. Zur Diagrammart Interaktionsdiagramm gehören die Diagrammarten Sequenzdiagramm und Kollaborationsdiagramm.

Mit der Diagrammart **Klassendiagramm** wird hauptsächlich die statische Struktur eines Anwendungssystems in der Spezifikationsphase beschrieben. Falls die Klassen bzw. Subsysteme eines Klassendiagramms auch Operatorensignaturen aufweisen, drücken diese, zusammen mit den Beziehungen zwischen ihnen, zusätzlich das statische Verhalten eines Anwendungssystems auf der Typebene aus.

Die Diagrammarten **Sequenzdiagramm** und **Kollaborationsdiagramm** werden zur Darstellung des dynamischen Verhaltens eines Anwendungssystems in der Ausführungsphase verwendet, wobei das Kollaborationsdiagramm zusätzlich die statische

Struktur der Instanzen aufzeigt. Im Grunde entspricht ein Kollaborationsdiagramm einem **Objektdiagramm**, das die statische Struktur eines Anwendungssystems in der Ausführungsphase beschreibt, ergänzt um Nachrichten und deren Reihenfolgebeziehungen [FoSc00, 72]. Zur Verdeutlichung der Reihenfolgebeziehungen zwischen den Nachrichten ist jedoch ein Sequenzdiagramm besser geeignet als ein Kollaborationsdiagramm [FoSc00, 66]. Aufgrund der jeweiligen Vorteile werden beide Diagrammarten zur Repräsentation des dynamischen Verhaltens eines komponentenbasierten Anwendungssystems zur Verfügung gestellt.

Metamodell zur Repräsentation der statischen Struktur und des statischen Verhaltens in der Spezifikationsphase

Abbildung 5.7 zeigt das **Metamodell** zur **Repräsentation der statischen Struktur und des statischen Verhaltens** eines komponentenbasierten Anwendungssystems **in der Spezifikationsphase**. Wie erwähnt, orientiert es sich an der Diagrammart Klassendiagramm der UML-Spezifikation, Version 1.4, wobei es nicht auf dem Meta-Metamodell der MOF basiert, sondern auf dem in Abschnitt 5.2.2.1 vorgestellten systemorientierten Meta-Metamodell. Es legt im Wesentlichen die Repräsentation von Software-Komponenten und die Repräsentation von Software-Komponenten-Beziehungen, die in Art und Anzahl beschränkt sind, fest und wird deswegen auch **Software-Komponenten-Diagramm** genannt.

Software-Komponenten werden, gemäß den oben stehenden Ausführungen, anhand des UML-Metaobjekts Subsystem repräsentiert. Eine detaillierte Beschreibung der Repräsentation von Software-Komponenten erfolgte bereits im ersten Teil des Abschnitts. Für jede im Kern-Metamodell spezifizierte Beziehungsart exisitiert mindestens eine zugehörige UML-Beziehungsart. Aggregationen mit wechselseitigen Existenzabhängigkeiten werden mit einer speziellen UML-Beziehungsart namens **Komposition** dargestellt. Jede repräsentierte Software-Komponenten-Beziehung kann mit höchstens zwei Kardinalitätsangaben, **Multiplizitäten** genannt, und höchstens zwei **Rollenbezeichnungen** versehen werden. Außerdem können ihr höchstens ein Stereotyp, beliebig viele Merkmale und beliebig viele Anmerkungen hinzugefügt werden. **Anmerkungen** werden üblicherweise dafür verwendet, die Bedeutung von besonders relevanten Diagrammobjekten durch informalen Text intuitiv verständlich zu machen.

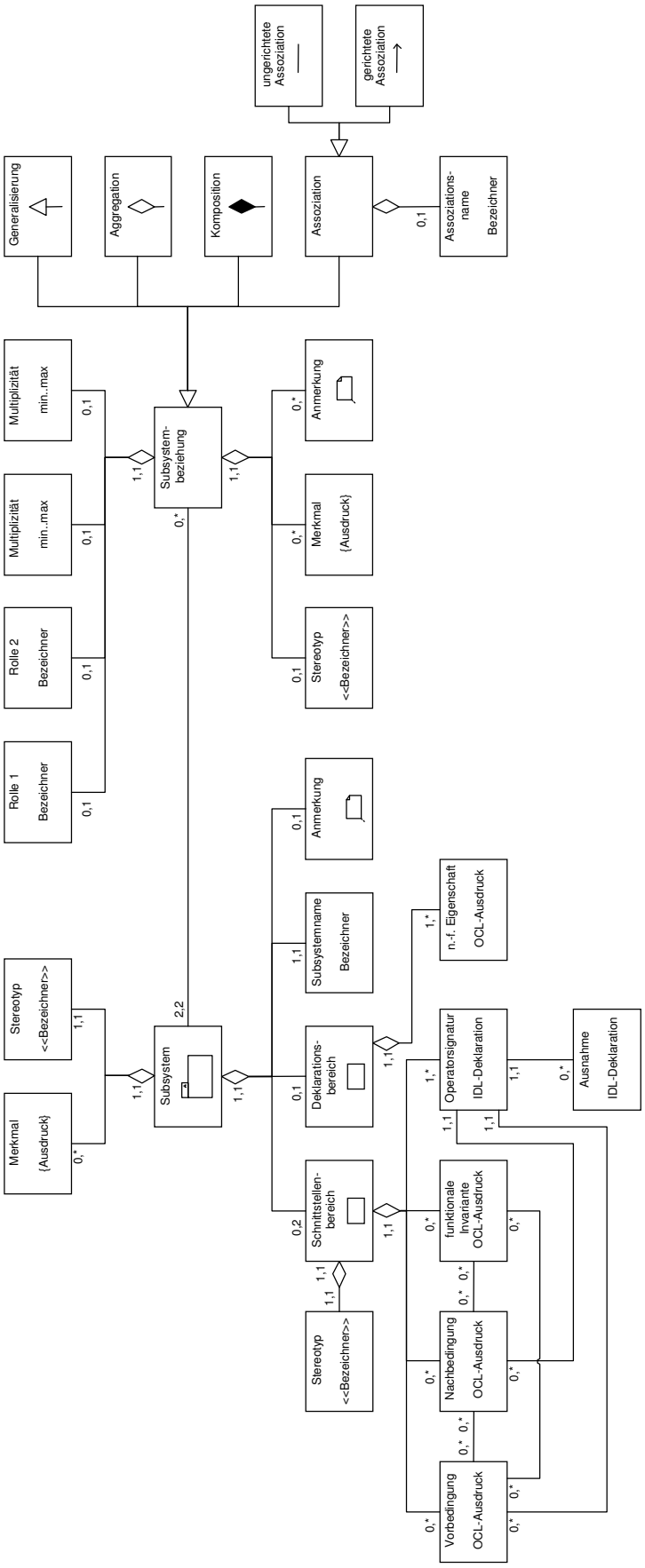


Abbildung 5.7: Repräsentations-Metamodell der statischen Struktur und des statischen Verhaltens eines komponentenbasierten Anwendungssystems in der Spezifikationsphase

Stereotypen und Merkmale gehören zusammen mit Einschränkungen zu den so genannten **generellen Erweiterungsmechanismen** der UML. Mit diesen können die Bedeutungen aller Diagrammobjekte nutzerspezifisch angepasst und erweitert werden [OMG01, 2-74]. Durch die Verwendung von **Stereotypen** lassen sich Diagrammobjekte nutzerspezifisch klassifizieren und anhand von **Merkmalen** und **Einschränkungen** werden ihnen einzelne nutzerspezifische Eigenschaften hinzugefügt [OMG01, 2-75]. Eine abgeschlossene Menge nutzerspezifischer Erweiterungsmechanismen bilden ein so genanntes **UML-Profil** [OMG01, 2-74]. Im vorliegenden Fall ist das Hinzufügen von Einschränkungen zu Software-Komponenten-Beziehungen jedoch nicht vorgesehen, da sämtliche verhaltensrelevante Informationen in den Schnittstellen der beteiligten Software-Komponenten beschrieben werden.

Metamodell zur Repräsentation der statischen Struktur und des dynamischen Verhaltens in der Ausführungsphase

In Abbildung 5.8 wird zunächst das **Metamodell** zur **Repräsentation der statischen Struktur und des dynamischen Verhaltens** eines komponentenbasierten Anwendungssystems **in der Ausführungsphase** dargestellt. Entsprechend den obigen Ausführungen lehnt es sich an der UML-Diagrammart Kollaborationsdiagramm an, wobei es jedoch ebenfalls auf der Grundlage des in Abschnitt 5.2.2.1 vorgestellten systemorientierten Meta-Metamodell erstellt wurde. Die Struktur- und Verhaltensmerkmale eines komponentenbasierten Anwendungssystems in der Ausführungsphase entsprechen denen eines Anwendungssystems in der Spezifikationsphase. Deswegen müssen in diesem Repräsentations-Metamodell die gleichen Beziehungsarten zur Verfügung gestellt werden, wie im Metamodell zur Repräsentation von Anwendungssystemen in der Spezifikationsphase. Eine Software-Komponenten-Instanz wird wiederum durch das UML-Metaobjekt Subsystem dargestellt. Aus Gründen der Übersichtlichkeit besitzt es nur einen Namen und einen Stereotyp, mit dem die Art der Software-Komponenten-Instanz festgelegt wird. Einer Software-Komponenten-Instanz-Beziehung werden mehrere Nachrichten zugeordnet, die den Nachrichtenaustausch zwischen den beteiligten Software-Komponenten-Instanzen im Zeitablauf repräsentieren. Deren Aufbau entspricht der jeweils adressierten Operatorsignatur. Die zeitliche Reihenfolge der Nachrichten wird mit entsprechenden **Sequenznummer** kenntlich gemacht. Zum besseren Verständnis des zeitlichen Ablaufs lässt sich jede Nachricht mit den Vor- und Nachbedingungen, die der aufzurufende Operator aufweist, versehen. Anhand von verschiedenartigen Pfeile wird außerdem festgelegt, ob es sich um eine **synchrone**, eine **asynchrone** oder eine **Rückgabenachricht** handelt.

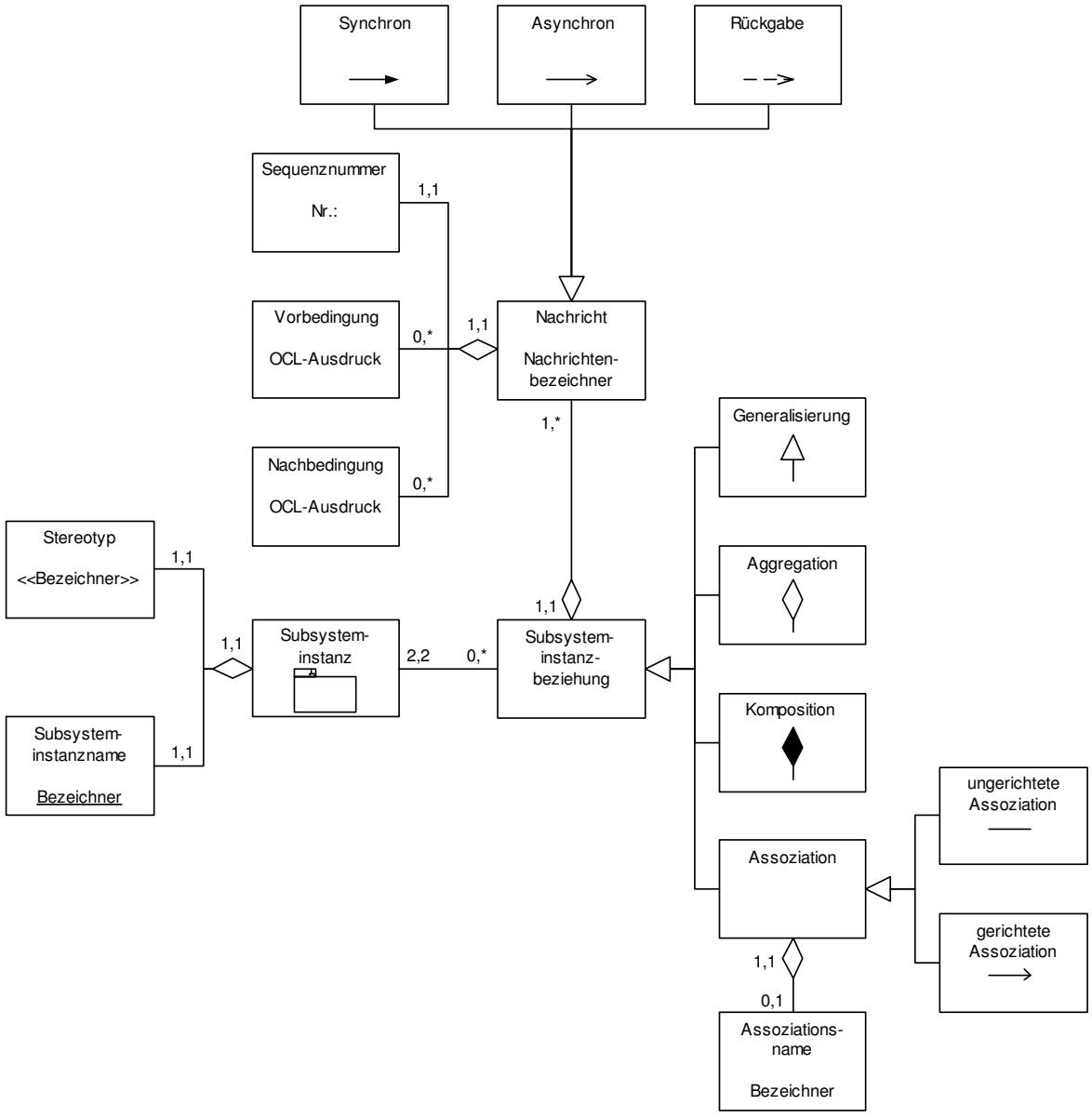


Abbildung 5.8: Repräsentations-Metamodell der statischen Struktur und des dynamischen Verhaltens eines komponentenbasierten Anwendungssystems in der Ausführungsphase

Metamodell zur Repräsentation des dynamischen Verhaltens in der Ausführungsphase

Das letzte vorzustellende **Metamodell** zur **Repräsentation** eines komponentenbasierten Anwendungssystems enthält Abbildung 5.9. Damit lässt sich ebenfalls das **dynamische Verhalten** eines komponentenbasierten Anwendungssystems **in der Ausführungsphase** ausdrücken, wobei jedoch der dynamische Aspekt vergleichsweise stärker betont wird. Es orientiert sich an der UML-Diagrammart Sequenzdiagramm, wobei es aber auch auf dem systemorientierten Meta-Metamodell aus Abschnitt 5.2.2.1 beruht. Die Repräsentation der Software-Komponenten-Instanz selbst durch das UML-Metaobjekt Subsystem tritt in diesem Repräsentations-Metamodell in den Hintergrund. Dafür wird die Zeitachse durch eine so genannte **Lebenslinie**, die

sich unterhalb jeder Software-Komponenten-Instanz befindet, hervorgehoben. Entlang dieser Lebenslinien wird der Nachrichtenaustausch zwischen den entsprechenden Software-Komponenten-Instanzen dargestellt. Die Notation einer Nachricht entspricht im Wesentlichen der des vorangegangenen Metamodells. Allerdings ist die zusätzliche Angabe von Sequenznummern überflüssig und wird deshalb im vorliegenden Repräsentations-Metamodell nicht angeboten. Dafür kann jede Nachricht mit einer informalen Beschreibung versehen werden, die gegebenenfalls Besonderheiten im zeitlichen Ablauf des Nachrichtenaustauschs näher erläutert. Die genannten Lebenslinien repräsentieren die Lebensläufe von Software-Komponenten-Instanzen. Das bedeutet, dass eine Lebenslinie erst ab dem Zeitpunkt der Erzeugung einer Software-Komponenten-Instanz beginnt und bei einer Löschung dieser Software-Komponenten-Instanz mit einem **Kreuzchen** endet. Zur Verdeutlichung der Steuerungsübergänge können die Lebenslinien mit entsprechenden **Aktivierungsbalken** versehen werden.

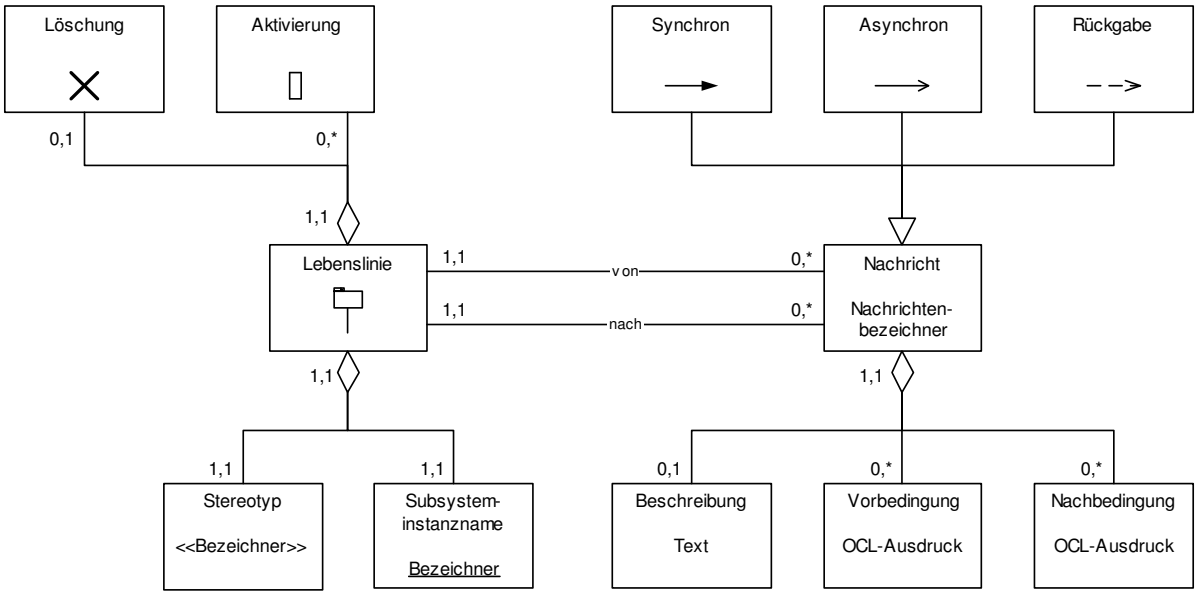


Abbildung 5.9: Repräsentations-Metamodell des dynamischen Verhaltens eines komponentenbasierten Anwendungssystems in der Ausführungsphase

5.2.2.4 Beziehungs-Metamodelle

Das in Abschnitt 5.2.2.2 vorgestellte Kern-Metamodell und die im letzten Abschnitt erläuterten Repräsentations-Metamodelle der Außenperspektive-Modellebene werden anhand von Beziehungs-Metamodellen miteinander verbunden. Demnach verknüpfen diese Beziehungs-Metamodelle, im Gegensatz zu den genannten Beziehungs-Metamodellen des generischen Architekturrahmens, Metaobjekte unterschiedlicher Metamodelle, die sich jedoch auf der gleichen Modellebene befinden. Die dabei verwendete Beziehungsart ist die Assoziation. Aus Gründen der Übersichtlichkeit und der besseren Verständlichkeit wird für dieses Beziehungs-Metamodell und für alle noch folgenden Beziehungs-Metamodelle die tabellarische Darstellung einer gra-

fischen vorgezogen. Eine Beziehungs-Metamodell-Tabelle besteht somit aus je einer Spalte für die Metaobjekte, die in Beziehung gesetzt werden sollen, und je einer Spalte für die zugehörigen Kardinalitäten.

Zunächst wird das in Abbildung 5.3 dargestellte Kern-Metamodell mit dem Metamodell zur Repräsentation der statischen Struktur und des statischen Verhaltens eines komponentenbasierten Anwendungssystems in der Spezifikationsphase in Beziehung gesetzt.

Metaobjekt Kern-Metamodell	Kardinalität 1	Metaobjekt Repräsentations-Metamodell	Kardinalität 2
Kommunikations-Software-Komponente	1,1	Subsystem mit Stereotyp „Kommunikation“	0,1
Datenhaltungs-Software-Komponente	1,1	Subsystem mit Stereotyp „Datenhaltung“	0,1
Vorgangs-Software-Komponente	1,1	Subsystem mit Stereotyp „Vorgang“	0,1
Entitäts-Software-Komponente	1,1	Subsystem mit Stereotyp „Entität“	0,1
Unterstützungs-Software-Komponente	1,1	Subsystem mit Stereotyp „Unterstützung“	0,1
Software-Komponenten-name	1,1	Subsystemname	0,1
Fachliche Schnittstelle	1,1	Schnittstellenbereich mit Stereotyp „Funktion“	0,1
Verwaltungsschnittstelle	1,1	Schnittstellenbereich mit Stereotyp „Verwaltung“	0,1
Fachliche Beschreibung	1,1	Anmerkung	0,1
Komponenten-operatorsignatur	1,1	Operatorsignatur	0,1
Ausnahmedeklaration	1,1	Ausnahme	0,1
Vorbedingung	1,1	Vorbedingung	0,1
Nachbedingung	1,1	Nachbedingung	0,1
Funktionale Invariante	1,1	Funktionale Invariante	0,1

Metaobjekt Kern-Metamodell	Kardinalität 1	Metaobjekt Repräsentations-Metamodell	Kardinalität 2
Deklaration nicht-funktionaler Eigenschaften	1,1	Deklarationsbereich	0,1
Nicht-funktionale Eigenschaft	1,1	Nicht-funktionale Eigenschaft	0,1
Generalisierung	1,1	Generalisierung	0,1
Aggregation	1,1	Aggregation oder Komposition (bei wechselseitiger Existenzabhängigkeit)	0,1
Ungerichtete Assoziation	1,1	Ungerichtete Assoziation	0,1
Gerichtete Assoziation	1,1	Gerichtete Assoziation	0,1
Assoziationsname	1,1	Assoziationsname	0,1
Kardinalität 1	1,1	Multiplizität 1	0,1
Kardinalität 2	1,1	Multiplizität 2	0,1

Tabelle 5.2: Beziehungs-Metamodell zur Verbindung des Kern-Metamodells mit dem Repräsentations-Metamodell der statischen Struktur und des statischen Verhaltens in der Spezifikationsphase

In der folgenden Tabelle werden die Beziehungen zwischen dem Metamodell zur Repräsentation der statischen Struktur und des statischen Verhaltens eines komponentenbasierten Anwendungssystems in der Spezifikationsphase und dem Metamodell zur Repräsentation der statischen Struktur und des dynamischen Verhaltens eines komponentenbasierten Anwendungssystems in der Ausführungsphase dargestellt. Zusammen mit dem Beziehungs-Metamodell in Tabelle 5.2 wird damit auch die Beziehung zwischem dem Kern-Metamodell und dem letztgenannten Repräsentations-Metamodell hergestellt.

Metaobjekt Repräsentations-Metamodell Spezifikation	Kardinalität 1	Metaobjekt Repräsentations-Metamodell Ausführung (stat.)	Kardinalität 2
Subsystem mit Namen und Stereotyp	1,1	Subsysteminstanz mit Namen und Stereotyp	0,*
Generalisierung	1,1	Generalisierung	0,*
Aggregation	1,1	Aggregation	0,*
Komposition	1,1	Komposition	0,*

Metaobjekt Repräsentations-Metamodell Spezifikation	Kardinalität 1	Metaobjekt Repräsentations-Metamodell Ausführung (stat.)	Kardinalität 2
Ungerichtete Assoziation	1,1	Ungerichtete Assoziation	0,*
Gerichtete Assoziation	1,1	Gerichtete Assoziation	0,*
Assoziationsname	1,1	Assoziationsname	0,*
Operatorsignatur	1,1	Nachricht	0,*
Vorbedingung	1,1	Vorbedingung	0,*
Nachbedingung	1,1	Nachbedingung	0,*

Tabelle 5.3: Beziehungs-Metamodell zur Verbindung des Repräsentations-Metamodells der statischen Struktur und des statischen Verhaltens in der Spezifikationsphase mit dem Repräsentations-Metamodell der statischen Struktur und des dynamischen Verhaltens in der Ausführungsphase

Schließlich zeigt die folgende Tabelle die Beziehungen zwischen dem Metamodell zur Repräsentation der statischen Struktur und des statischen Verhaltens eines komponentenbasierten Anwendungssystems in der Spezifikationsphase und dem Metamodell zur ausdrücklichen Repräsentation des dynamischen Verhaltens eines komponentenbasierten Anwendungssystems in der Ausführungsphase. Auch in diesem Fall gilt, dass zusammen mit dem Beziehungs-Metamodell in Tabelle 5.2 ebenso die Beziehungen zwischen dem Kern-Metamodell und dem letztgenannten Repräsentations-Metamodell feststehen.

Metaobjekt Repräsentations-Metamodell Spezifikation	Kardinalität 1	Metaobjekt Repräsentations-Metamodell Ausführung (dyn.)	Kardinalität 2
Subsystem mit Namen und Stereotyp	1,1	Lebenslinie mit Namen und Stereotyp	0,*
Operatorsignatur	1,1	Nachricht	0,*
Vorbedingung	1,1	Vorbedingung	0,*
Nachbedingung	1,1	Nachbedingung	0,*

Tabelle 5.4: Beziehungs-Metamodell zur Verbindung des Repräsentations-Metamodells der statischen Struktur und des statischen Verhaltens in der Spezifikationsphase mit dem Repräsentations-Metamodell des dynamischen Verhaltens in der Ausführungsphase

5.2.3 Sichten

In Abschnitt 2.2.4.1 wurden die relevanten Sichten auf Informations- bzw. Anwendungssysteme vorgestellt. Analog zu objektorientierten Ansätzen werden auf der Außenperspektive-Modellebene durch die Verwendung von Software-Komponenten

und Kommunikationskanälen zur Verbindung von Software-Komponenten die Funktionssicht, die Datensicht und die Interaktionssicht unterstützt. Darüber hinaus kann mit den eingeführten Metamodellen zur Repräsentation des dynamischen Verhaltens eines komponentenbasierten Anwendungssystems in der Ausführungsphase auch die Vorgangssicht dargestellt werden.

Zur Unterstützung der Komplexitätsbewältigung stehen jedoch weitere Sichten zur Verfügung. Auf dieser Modellebene werden sie anhand der Systemmerkmale Struktur und Verhalten abgegrenzt.

Sichten im Kern-Metamodell

Im Kern-Metamodell umfasst die **Struktursicht** zum einen das Metaobjekt Software-Komponente inklusive aller angegebenen Spezialisierungen und die zum Metaobjekt Software-Komponente gehörenden Metaobjekte fachliche Schnittstelle, Verwaltungsschnittstelle, Deklaration nicht-funktionaler Eigenschaften, Software-Komponentenname und fachliche Kurzbeschreibung. Außerdem gehören das Metaobjekt Software-Komponenten-Beziehung einschließlich aller angegebenen Spezialisierungen, die zugehörigen Metaobjekte Kardinalität und Assoziationsname und alle Beziehungen zwischen den genannten Metaobjekten zur Struktursicht des Kern-Metamodells. Die korrespondierende **Verhaltenssicht** enthält die zum Metaobjekt Software-Komponente gehörenden Metaobjekte Komponentenoperatorsignatur, Ausnahmedeklaration und Zusicherung, alle Spezialisierungen des Metaobjekts Zusicherung, das zum Metaobjekt Software-Komponenten-Beziehung gehörende Metaobjekt Komponentennachricht und alle Beziehungen zwischen den aufgeführten Metaobjekten.

Sichten im Metamodell zur Repräsentation der statischen Struktur und des statischen Verhaltens in der Spezifikationsphase

Die Struktur- und die Verhaltenssicht des Metamodells zur Repräsentation der statischen Struktur und des statischen Verhaltens eines komponentenbasierten Anwendungssystems in der Spezifikationsphase sind analog zu den entsprechenden Sichten des Kern-Metamodells. Demnach weist die **Struktursicht** zum einen das Metaobjekt Subsystem einschließlich den zugehörigen Metaobjekten Merkmal, Stereotyp, Subsystemname, Anmerkung, Deklarationsbereich und Schnittstellenbereich mit dem zugeordneten Metaobjekt Stereotyp auf. Darüber hinaus sind in der Struktursicht das Metaobjekt Subsystembeziehung inklusive aller angegebenen Spezialisierungen, die zugehörigen Metaobjekte Merkmal, Stereotyp, Anmerkung, Rolle, Multiplizität und Assoziationsname und alle Beziehungen zwischen den genannten Metaobjekten enthalten. Die **Verhaltenssicht** dieses Repräsentations-Metamodells umfasst die zum Metaobjekt Subsystem gehörenden Metaobjekte Operatorsignatur, Ausnahme, Vorbedingung, Nachbedingung, funktionale Invariante und nicht-funktionale Eigenschaft.

Sichten im Metamodell zur Repräsentation der statischen Struktur und des dynamischen Verhaltens in der Ausführungsphase

Im Metamodell zur Repräsentation der statischen Struktur und des dynamischen Verhaltens eines komponentenbasierten Anwendungssystems in der Ausführungsphase wird die Verhaltenssicht stärker betont als im vorangegangenen Repräsentations-Metamodell. Folglich sind in der entsprechenden **Struktursicht** nur das Metaobjekt Subsysteminstanz mit den zugehörigen Metaobjekten Stereotyp und Subsysteminstanzname, das Metaobjekt Subsysteminstanzbeziehung einschließlich aller Spezialisierungen und dem Metaobjekt Assoziationsname sowie die Beziehungen zwischen diesen Metaobjekten enthalten. Die **Verhaltenssicht** des Metamodells weist das Metaobjekt Nachricht, die entsprechenden Spezialisierungen, die zugehörigen Metaobjekte Sequenznummer, Vorbedingung und Nachbedingung und die Beziehungen zwischen den genannten Metaobjekten auf.

Sichten im Metamodell zur Repräsentation des dynamischen Verhaltens in der Ausführungsphase

Mit dem in Abschnitt 5.2.2.3 zuletzt vorgestellten Repräsentations-Metamodell lässt sich das dynamische Verhalten eines komponentenbasierten Anwendungssystems detailliert darstellen. Zu diesem Zweck schränkt es die Modellierbarkeit von Strukturmerkmalen zu Gunsten der Modellierbarkeit von Verhaltensmerkmalen ein. Demzufolge können alle Metaobjekte dieses Repräsentations-Metamodells der **Verhaltenssicht** zugeordnet werden. Die **Struktursicht** enthält somit keine Metaobjekte.

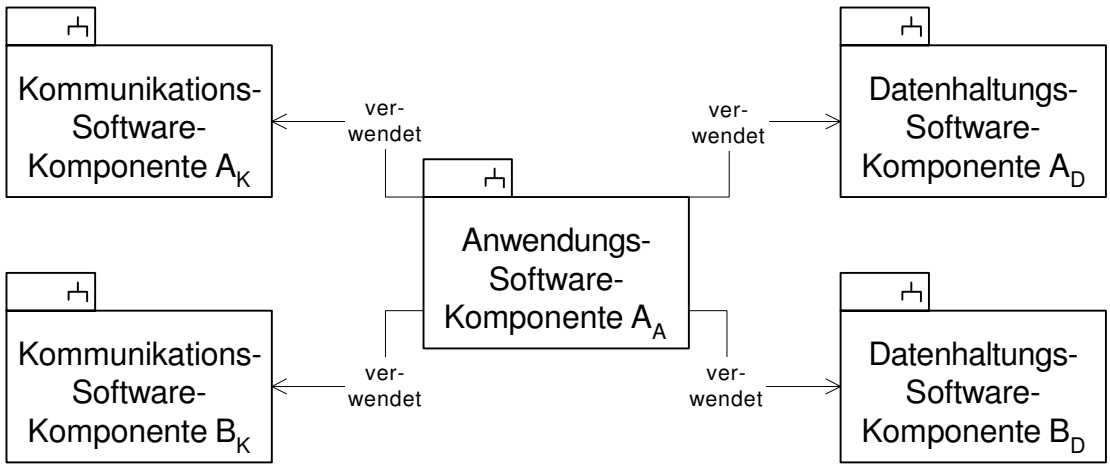
5.2.4 Muster

Zur Unterstützung des softwaretechnischen Entwurfs von Struktur- und Verhaltensmerkmalen komponentenbasierter Anwendungssysteme steht eine ständig wachsende Anzahl von Entwurfs- und Architekturmustern zur Verfügung. Darauf wurde bereits in den Ausführungen zum Grundkonzept der Komponentenorientierung in Kapitel 4 hingewiesen. Dabei handelt es sich entweder um domänenspezifische oder um domänenunabhängige Entwurfs- und Architekturmuster. Grundlegende **domänenunabhängige Entwurfsmuster** sind in [GHJV96; BMR+98] enthalten. Sie werden meistens als elementare Bestandteile in komplexeren Entwurfsmustern oder in Architekturmustern verwendet. Wesentliche **domänenunabhängige Architekturmuster** werden z. B. in [BMR+98] beschrieben. Diese domänenunabhängigen Entwurfs- und Architekturmuster sind häufig auch in **domänenspezifischen Entwurfs- und Architekturmustern** enthalten. Sowohl die domänenunabhängigen als auch die domänenspezifischen Entwurfsmuster können zur Modellierung auf der Außenperspektive-Modellebene verwendet werden.

Darüber hinaus stehen speziell für diese Modellebene des Software-Architekturmodells weitere Entwurfs- und Architekturmuster zur Verfügung, die im weiteren Verlauf dieses Abschnitts vorgestellt werden. Sie beschreiben heuristisches Modellierungswissen, das mit dem Kern-Metamodell nicht ausgedrückt werden kann. Insofern stellen sie eine notwendige Ergänzung zu den Metamodellen der vorangegangenen Abschnitt dar. Die Beschreibungsform dieser Muster orientiert sich, wie bereits in Abschnitt 3.1.3.6 erwähnt, an der alexandrinischen Grundform.

Beim ersten Muster handelt es sich um ein Architekturmuster, das die Strukturierung eines Anwendungssystems in Software-Komponenten für die Anwendungsfunktionen, für die Datenhaltung und für die Kommunikation mit Personen und maschinellen Aufgabenträgern beschreibt.

Name
Anwendungsfunktionen, Datenhaltung und Kommunikation (ADK)
Kontext
Softwaretechnischer Entwurf eines komponentenbasierten Anwendungssystems, das zur Erfüllung seiner Anwendungsfunktionen mit Personen und bzw. oder maschinellen Aufgabenträgern interagieren und die dabei verwendeten Datenobjekte persistent halten soll.
Problem
Ein Anwendungssystem, das aus Software-Komponenten besteht, die sowohl die Anwendungsfunktionalität als auch die Interaktions- und die Datenhaltungsfunktionalität enthalten, ist unflexibel und komplex. Außerdem sind solche Software-Komponenten nicht portierbar, da jede individuell entworfen wird und somit keine Basismaschinen mit standardisierten Schnittstellen eingesetzt werden können. Aufgrund der individuellen Entwicklung solcher Software-Komponenten müssen auch Interaktions- oder Datenhaltungsfunktionen, die auf gleiche Weise in mehreren Software-Komponenten erforderlich sind, jeweils neu entworfen werden. Zudem ist ein hoher Aufwand für die Wartung eines solchen Anwendungssystems erforderlich, da die Anwendungsfunktionalität den häufigsten Änderungen unterliegt und jede Änderung der Anwendungsfunktionalität einer Software-Komponente meistens auch eine Änderung der Interaktions- und der Datenhaltungsfunktionalität dieser Software-Komponente nach sich zieht.
Kräfte
Der softwaretechnische Entwurf eines komponentenbasierten Anwendungssystems muss auf der Basis von Software-Komponenten erfolgen.

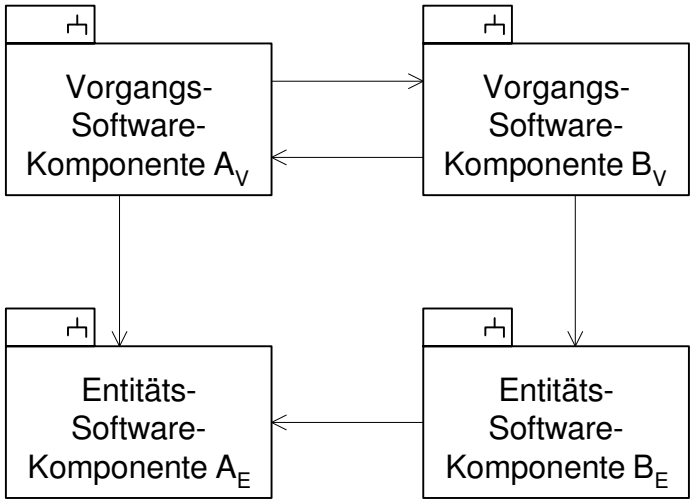
Lösung
Die in den Software-Komponenten enthaltene Anwendungsfunktionalität wird anhand von speziellen Software-Komponenten für die Anwendungsfunktionen (so genannte Anwendungs-Software-Komponenten) von der Interaktions- und der Datenhaltungsfunktionalität getrennt. Diesen werden spezielle Software-Komponenten für die Kommunikation mit Personen und maschinellen Aufgabenträgern (so genannte Kommunikations-Software-Komponenten) und spezielle Software-Komponenten für die Datenhaltung (so genannte Datenhaltungs-Software-Komponenten) zugeordnet. Somit wird auch die Interaktions- von der Datenhaltungsfunktionalität getrennt.
Lösungsstruktur

Beziehungen
-

Das nächste Muster gehört zur Kategorie der Entwurfsmuster. Es trennt die Software-Komponenten für die Anwendungsfunktionen in Vorgangs-Software-Komponenten und Entitäts-Software-Komponenten.

Name
Vorgang und Entität (VE)
Kontext
Softwaretechnischer Entwurf der Anwendungsfunktionen eines komponentenbasierten Anwendungssystems. Die Anwendungsfunktionen eines Anwendungssystems sind aus der Zerlegung der Gesamtaufgabe entstanden, die dem Anwendungssystem zugrundeliegt. Demzufolge umfasst jede Anwendungsfunktion die Durchführung einer Teilaufgabe auf einem entsprechenden Teil des Aufgabenobjekts.

Problem
In einem komponentenbasierten Anwendungssystem sind die Anwendungsfunktionen auf mehrere Software-Komponenten verteilt. Ohne zielgerichtete Koordination dieser Software-Komponenten kann die zugrundeliegende Gesamtaufgabe nicht erfüllt werden. Außerdem können unterschiedliche Software-Komponenten übereinstimmende Teile des Aufgabenobjekts enthalten. Diese Datenredundanz kann zu Verletzungen der semantischen und operationalen Integrität führen.
Kräfte
Der softwaretechnische Entwurf der Anwendungsfunktionen eines komponentenbasierten Anwendungssystems muss auf der Basis von Software-Komponenten erfolgen.
Lösung
Die zielgerichtete Koordination der Software-Komponenten wird durch ihre Zerlegung in Vorgangs-Software-Komponenten und Entitäts-Software-Komponenten erreicht. Eine Vorgangs-Software-Komponente kapselt die Durchführung einer Teilaufgabe und benötigt dafür bestimmte Entitäts-Software-Komponenten, die Teile des Aufgabenobjekts darstellen. Demzufolge wird jede Anwendungsfunktion durch eine Vorgangs-Software-Komponente und einer Menge von entsprechenden Entitäts-Software-Komponenten realisiert. Beziehungen zwischen den Vorgangs-Software-Komponenten korrespondieren mit der Zerlegungsstruktur der zugrundeliegenden Gesamtaufgabe. Die Gesamtheit aller Vorgangs-Software-Komponenten und aller Entitäts-Software-Komponenten eines komponentenbasierten Anwendungssystems realisieren die zugrundeliegende Gesamtaufgabe. Durch die Trennung von Vorgangs-Software-Komponenten und Entitäts-Software-Komponenten wird auch die Datenredundanz und die dadurch hervorgerufene Gefahr der Verletzung von Integritätsbedingungen beseitigt, da unterschiedliche Vorgangs-Software-Komponenten, die zur Durchführung ihrer Teilaufgaben das gleiche Teil-Aufgabenobjekt benötigen, auf ein und dieselbe Entitäts-Software-Komponente, die dieses Teil-Aufgabenobjekt realisiert, zugreifen. Die Entitäts-Software-Komponenten eines komponentenbasierten Anwendungssystems werden unter Berücksichtigung der Regeln zur konzeptuellen Datenmodellierung miteinander in Beziehung gesetzt. Dies ergibt eine Struktur, die in der Datensicht weitgehend mit einem konzeptuellen Datenschema übereinstimmt. Dadurch sind vergleichbare Zielerreichungsgrade bezüglich der Merkmale Datenredundanz und Integrität möglich.

Lösungsstruktur



Beziehungen

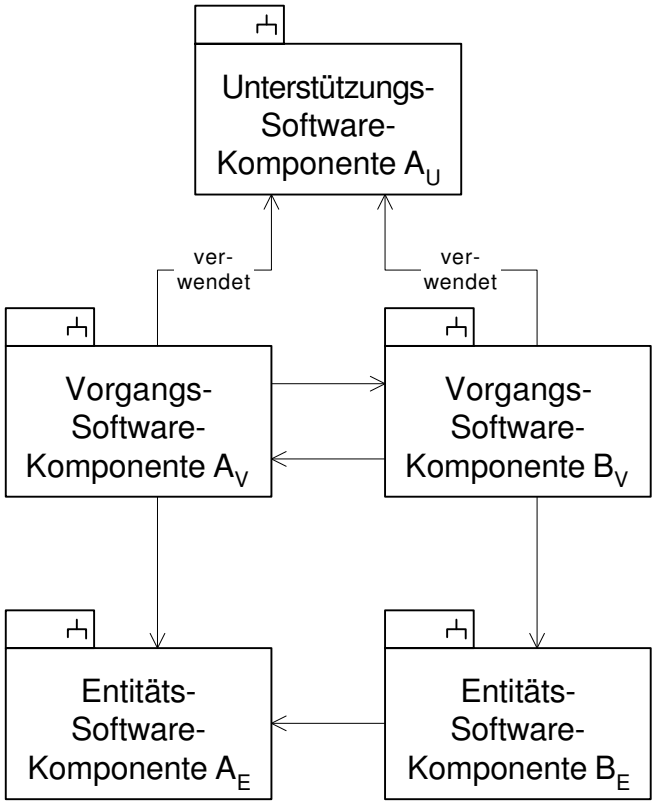
Wenn ein komponentenbasiertes Anwendungssystem zuvor anhand des ADK-Musters in Anwendungs-Software-Komponenten, Kommunikations-Software-Komponenten und Datenhaltungs-Software-Komponenten unterteilt wurde, bezieht sich dieses Entwurfsmuster auf die Anwendungs-Software-Komponenten. Nach der Anwendung dieses Musters ergeben sich folgende Beziehungen zu den Kommunikations-Software-Komponenten und den Datenhaltungs-Software-Komponenten:

- Für jeden Interaktionsbedarf einer Vorgangs-Software-Komponente mit Personen oder maschinellen Aufgabenträgern wird ihr eine Kommunikations-Software-Komponente zugeordnet, die von ihr während der Ausführung gesteuert wird.
- Für alle persistent deklarierten Attribute einer Entitäts-Software-Komponente wird ihr eine entsprechende Datenhaltungs-Software-Komponente zugeordnet. Allerdings können einer Entitäts-Software-Komponente auch mehrere Datenhaltungs-Software-Komponenten zugeordnet werden, wenn die persistent deklarierten Attribute ihre Entsprechungen in mehreren Datenhaltungs-Software-Komponenten finden. Umgekehrt kann eine Datenhaltungs-Software-Komponente mehreren Entitäts-Software-Komponenten zugeordnet sein, wenn sie Attribute besitzt, die mit persistent deklarierten Attributen mehrerer Entitäts-Software-Komponenten korrespondiert.

Das letzte vorzustellende Muster in diesem Abschnitt ist ebenfalls ein Entwurfsmuster, das den Zusammenhang zwischen Vorgangs-Software-Komponenten und Unterstützungs-Software-Komponenten klärt.

Name
Unterstützung
Kontext
Softwaretechnischer Entwurf der Anwendungsfunktionen eines komponentenbasierten Anwendungssystems. Die Anwendungsfunktionen eines Anwendungssystems sind aus der Zerlegung der Gesamtaufgabe entstanden, die dem Anwendungssystem zugrundeliegt. Demzufolge umfasst jede Anwendungsfunktion die Durchführung einer Teilaufgabe auf einem entsprechenden Teil des Aufgabenobjekts. Die Realisierung einer Anwendungsfunktion erfolgt durch eine Vorgangs-Software-Komponente für die Durchführung einer Teilaufgabe und entsprechenden Entitäts-Software-Komponenten für den Teil des Aufgabenobjekts.
Problem
Aus der Zerlegung der zugrundeliegenden Gesamtaufgabe sind Teilaufgaben entstanden, deren Lösungsverfahren sich gegebenenfalls überlappen können. Demzufolge ist es möglich, dass die Operatoren einiger Vorgangs-Software-Komponenten Aktionen enthalten, die auch in Operatoren anderer Vorgangs-Software-Komponenten enthalten sind. Diese Funktionsredundanzen sind Quellen für mögliche Inkonsistenzen und deuten auf eine mangelnde Wirtschaftlichkeit der Ressourcennutzung hin.
Kräfte
Der softwaretechnische Entwurf der Anwendungsfunktionen eines komponentenbasierten Anwendungssystems muss auf der Basis von Software-Komponenten erfolgen.
Lösung
Zur Vermeidung von Funktionsredundanzen werden Aktionen, die in Operatoren mehrerer Vorgangs-Software-Komponenten vorkommen, in einen Operator einer unterstützenden Vorgangs-Software-Komponente (so genannte Unterstützungs-Software-Komponente) ausgelagert. Zwischen den Vorgangs-Software-Komponenten, aus denen die Aktionen ausgelagert wurden, und der Unterstützungs-Software-Komponente werden demzufolge Assoziationsbeziehungen eingeführt, die eine semantische Relation *verwendet* beschreiben.

Lösungsstruktur



Beziehungen

Bevor dieses Muster angewendet werden kann, müssen die Software-Komponenten für die Anwendungsfunktionen eines komponentenbasierten Anwendungssystems anhand des Entwurfsmusters *Vorgang und Entität* in Vorgangs- und in Entitäts-Software-Komponenten zerlegt worden sein.

5.3 Modellebene der Struktur- und Verhaltensmerkmale eines komponentenbasierten Anwendungssystems aus der Innenperspektive

Die **Innenperspektive-Modellebene** steht für die Spezifikation der Implementierungen von Software-Komponenten eines komponentenbasierten Anwendungssystems und der von ihnen benötigten Basismaschinendienste zur Verfügung. Grundlage für die Spezifikation der Implementierungen sind die spezifizierten Außenperspektiven dieser Software-Komponenten auf der vorangegangenen Modellebene. Zur Spezifikation der Basismaschinendienste, die von den Software-Komponenten benötigt werden, existieren mehrere Entwurfs- und Architekturmuster, die Gegenstand des Abschnitts 5.3.4 sind.

Wie auf der Außenperspektive-Modellebene steht auch auf dieser Modellebene ein Kern-Metamodell und mehrere Repräsentations-Metamodelle zur Verfügung. Eine standardisierte und möglichst formalisierte Repräsentation der Entwurfsergebnisse unterstützt deren Kommunikation und effizienten Austausch zwischen den Entwicklern während des *Development for Reuse*. Als Repräsentationssprachen werden ebenfalls ausgewählte standardisierte Spezifikationen der OMG verwendet. Die besondere Eignung der OMG-Spezifikationen für die Repräsentation softwaretechnischer Entwurfsergebnisse wurde bereits in der Einführung zu Abschnitt 5.2 hervorgehoben. Dieser Abschnitt befasste sich auch mit der Notwendigkeit zur Trennung von Kern- und Repräsentations-Metamodell .

5.3.1 Metapher

In Abschnitt 5.2.1 wurde bereits darauf hingewiesen, dass die Metaphern beider Modellebenen mit der übergeordneten Metapher des Software-Architekturmodells übereinstimmen. Somit beschreibt auch die Metapher dieser Modellebene ein betriebliches Anwendungssystem als objektintegriertes verteiltes System, das die durch die teilautomatisierte Unternehmensaufgabe vorgegebenen Ziele verfolgt. Allerdings findet beim Übergang von der Außenperspektive-Modellebene zur Innenperspektive-Modellebene ein Wechsel des verwendeten Spezifikationsparadigmas statt. Das Spezifikationsparadigma der Außenperspektive-Modellebene ist die Komponentenorientierung, das der Innenperspektive-Modellebene ist die Objektorientierung. Damit eine durchgängige Verwendung des objektorientierten Paradigmas auf dieser Modellebene gewährleistet werden kann, erfolgt die Spezifikation der Basismaschinendienste ebenfalls objektorientiert.

5.3.2 Metamodelle

Die Abschnitte 5.3.2.2 und 5.3.2.3 beschreiben das Kern-Metamodell und die Repräsentations-Metamodelle der Innenperspektive-Modellebene. Zu diesem Zweck sind wiederum geeignete Meta-Metamodelle erforderlich.

5.3.2.1 Meta-Metamodell

Aus den in Abschnitt 5.2.2.1 genannten Gründen erfordert das Software-Architekturmodell ausschließlich systemorientierte Metamodelle. Ein Meta-Metamodell, mit dem systemorientierte Metamodelle erstellt werden können, wurde ebenfalls dort vorgestellt. Dieses eignet sich auch für die Beschreibung des Kern-Metamodells und der Repräsentations-Metamodelle der Innenperspektive-Modellebene.

5.3.2.2 Kern-Metamodell

Die Struktur- und Verhaltensmerkmale eines komponentenbasierten Anwendungssystems aus der Innenperspektive werden anhand der Innenperspektiven der Software-Komponenten und der Beziehungen zwischen ihnen beschrieben. Beide wurden auf der vorangegangenen Modellebene aus der Außenperspektive spezifiziert. Da gemäß dem Grundkonzept der Komponentenorientierung Software-Komponenten objektorientiert implementiert werden, stellt das Kern-Metamodell dieser Modellebene Metaobjekte zur Spezifikation von Klassenschemata zur Verfügung. Dazu gehören im Wesentlichen Metaobjekte zur Beschreibung von Klassen und zur Beschreibung der zulässigen Arten von Beziehungen zwischen Klassen (vgl. Abbildung 5.10).

Entsprechend dem Grundkonzept der Objektorientierung, das in Abschnitt 3.5.1.1 erläutert wurde, besitzt eine Klasse einen Namen und weist üblicherweise mehrere Operatoren auf, die sich auf die Attribute der Klasse beziehen. Die Operatoren bestehen jeweils aus einer Operatorsignatur und einer zugehörigen Operatorimplementierung. Für jede Software-Komponenten-Art exisitiert eine entsprechende Klassenart. Demnach stehen für Vorgangs-Software-Komponenten und Unterstützungs-Software-Komponenten **Vorgangsklassen**, für Entitäts-Software-Komponenten **konzeptuelle Klassen**, für Datenhaltungs-Software-Komponenten **Datenhaltungsklassen** und für Kommunikations-Software-Komponenten **Kommunikationsklassen** zur Verfügung. Darüberhinaus werden für den Entwurf von Basismaschinendiensten **Dienstklassen** bereitgestellt, deren Bedeutungen im Rahmen der Erläuterungen zu den ebenenspezifischen Mustern in Abschnitt 5.3.4 geklärt wird. Dieser Entwurf erfolgt, wie erwähnt, nicht für jede Software-Komponente einzeln, sondern für ein komponentenbasiertes Anwendungssystem insgesamt. Deswegen stehen die Dienstklassen eines solchen Entwurfs auf der Innenperspektive-Modellebene nur untereinander in Beziehung. Damit diese Einschränkung entsprechend spezifiziert werden kann, enthält das Metamodell eine spezialisierte Beziehungsart, die so genannte **Dienstklassenbeziehung**. Zur Spezifikation von Beziehungen zwischen Klassen der übrigen Klassenarten stehen einerseits die Beziehungsart **Intra-Komponenten-Beziehung** und andererseits die Beziehungsart **Inter-Komponenten-Beziehung** zur Verfügung. Erstgenannte wird für Beziehungen zwischen Klassen innerhalb einer Software-Komponente verwendet, Letztgenannte für Beziehungen zwischen Klassen unterschiedlicher Software-Komponenten. Diese Differenzierung ist notwendig, da die Objekte der Klassen innerhalb einer Software-Komponente unmittelbar miteinander interagieren, die Objekte der Klassen unterschiedlicher Software-Komponenten jedoch nur mittelbar über die Schnittstellen und Beziehungen der Software-Komponenten, in denen sie enthalten sind.

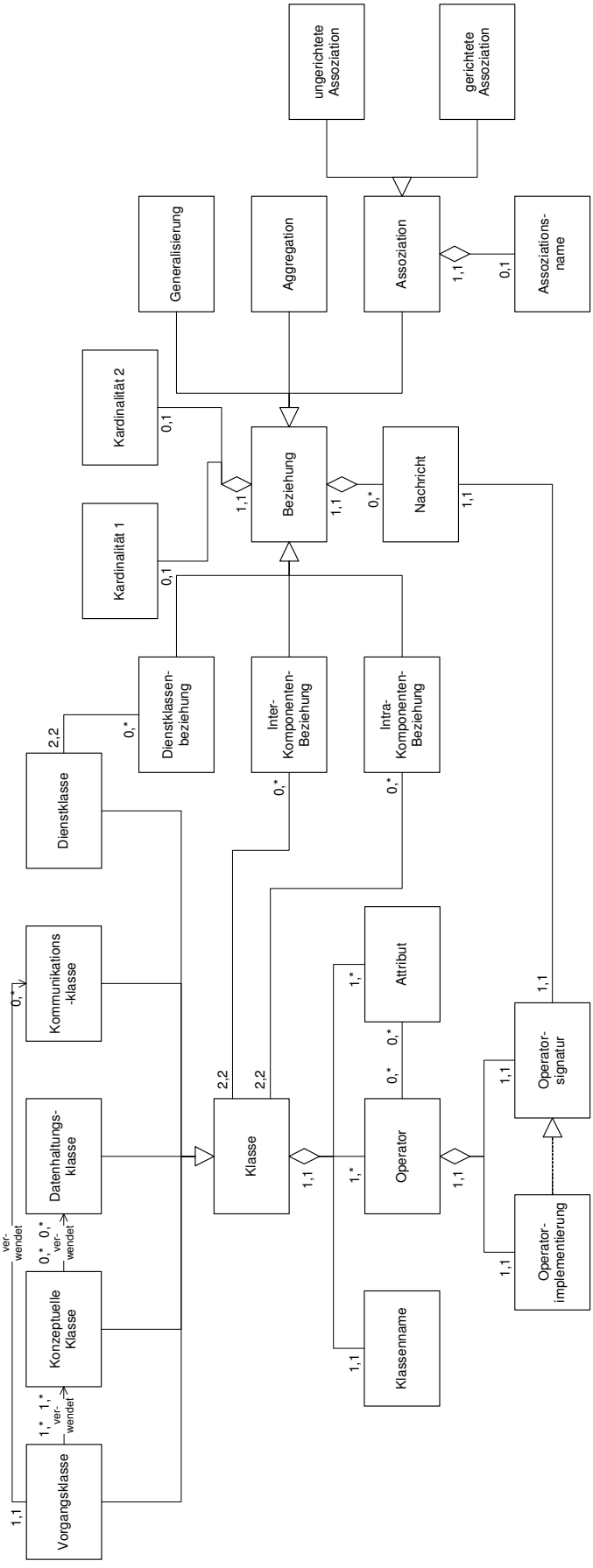


Abbildung 5.10: Kern-Metamodell der Innenperspektive-Modellebene

Generell können jeder Beziehung maximal zwei Kardinalitätsangaben und mehrere Nachrichtendefinitionen zugewiesen werden. Die Nachrichtendefinitionen entsprechen bekanntermaßen den Operatorensignaturen der an der Beziehung beteiligten Klassen. Ferner kann jede Beziehung entweder als Generalisierung, als Aggregation oder als Assoziation spezifiziert werden, wobei jede Assoziation entweder gerichtet oder ungerichtet ist und, falls erforderlich, einen Namen aufweist.

5.3.2.3 Repräsentations-Metamodelle

Damit während des *Development for Reuse* die objektorientiert spezifizierten Implementierungen von Software-Komponenten und die von ihnen benötigten, ebenfalls objektorientiert spezifizierten Basismaschinendienste einfach und effizient ausgetauscht werden können, müssen sie in einer standardisierten und geeignet formalisierten Form vorliegen. Dafür wird eine standardisierte Sprache zur Notation objektorientierter Entwürfe benötigt, die zum Zwecke einer durchgängigen Repräsentation von Entwicklungserzeugnissen möglichst in allen Entwicklungsphasen eingesetzt werden kann. Diese Anforderungen kann die UML erfüllen. Im Gegensatz zur Außenperspektive-Modellebene bestehen auf dieser Modellebene keine besonderen Anforderungen an die Repräsentation von Klassen oder Objekten. Deswegen ist die UML als Repräsentationssprache auch hinreichend und eine weitere Differenzierung zwischen der Repräsentation einer Software-Komponente und der Repräsentation eines komponentenbasierten Anwendungssystems nicht erforderlich.

Die Beschreibung der folgenden UML-Metamodelle erfolgt aus den bereits genannten Gründen wiederum nicht auf Basis der MOF, sondern auf Basis des Meta-Metamodells, das in Abschnitt 5.2.2.1 vorgestellt wurde. Die Metamodelle berücksichtigen alle Sprachelemente, die in der UML-Spezifikation 1.4 für den softwaretechnischen Entwurf betrieblicher Anwendungssysteme zur Verfügung stehen.

Repräsentations-Diagrammarten

Analog zur Außenperspektive-Modellebene werden zur Repräsentation der objektorientierten Implementierungen von Software-Komponenten und zur Repräsentation der objektorientiert spezifizierten Basismaschinendienste die Diagrammarten Klassendiagramm, Sequenzdiagramm und Kollaborationsdiagramm verwendet. Die Diagrammart Klassendiagramm eignet sich zur Darstellung der statischen Struktur und des statischen Verhaltens eines objektorientierten Systems in der Spezifikationsphase. Dagegen werden die Diagrammarten Sequenzdiagramm und Kollaborationsdiagramm hauptsächlich zur Darstellung des dynamischen Verhaltens eines objektorientierten Systems in der Ausführungsphase verwendet.

Metamodell zur Repräsentation der statischen Struktur und des statischen Verhaltens in der Spezifikationsphase

In Abbildung 5.11 ist das systemorientierte Repräsentations-Metamodell zur Erstellung von Klassendiagrammen dargestellt. Es besteht im Wesentlichen aus einem Metaobjekt für die Repräsentation von Klassen und einer Metabeziehung für die Repräsentation von Klassenbeziehungen. Eine Klasse weist zumindest einen Namen auf. Die Darstellung der Attribute und der Operatoren einer Klasse ist optional. Wie bereits in Abschnitt 5.2.2.3 erläutert wurde, können mithilfe von Stereotypen, Merkmalen und Einschränkungen die Bedeutungen aller Diagrammelemente nutzerspezifisch angepasst und erweitert werden. Außerdem stehen zur informalen Erläuterung besonders relevanter Diagrammelemente das UML-Metaobjekt Anmerkung zur Verfügung. Eine Klassenbeziehung ist entweder eine Generalisierung, eine Aggregation, eine Komposition, eine gerichtete oder eine ungerichtete Assoziation. Zusätzlich können mit der UML auch Realisierungsbeziehungen zwischen Klassen und Schnittstellen und so genannte **Abhängigkeitsbeziehungen** zwischen Klassen oder Schnittstellen spezifiziert werden. Letztere werden zur Darstellung von Situationen benötigt, in denen Änderungen der unabhängigen Klasse oder Schnittstelle Änderungen der abhängigen Klassen oder Schnittstelle bedingen können [OMG01, 3-90]. Im Unterschied zu den übrigen Beziehungsarten, die jeweils eine dauerhafte Verbindung zwischen Objekten der beteiligten Klassen repräsentieren, werden Abhängigkeitsbeziehungen zur Darstellung von nicht-dauerhaften Beziehungen zwischen Objekten der beteiligten Klassen verwendet [FoSc00, 49]. Generell kann einer Klassenbeziehung ein Name, höchstens zwei Rollen und höchstens zwei Multiplizitäten hinzugefügt werden.

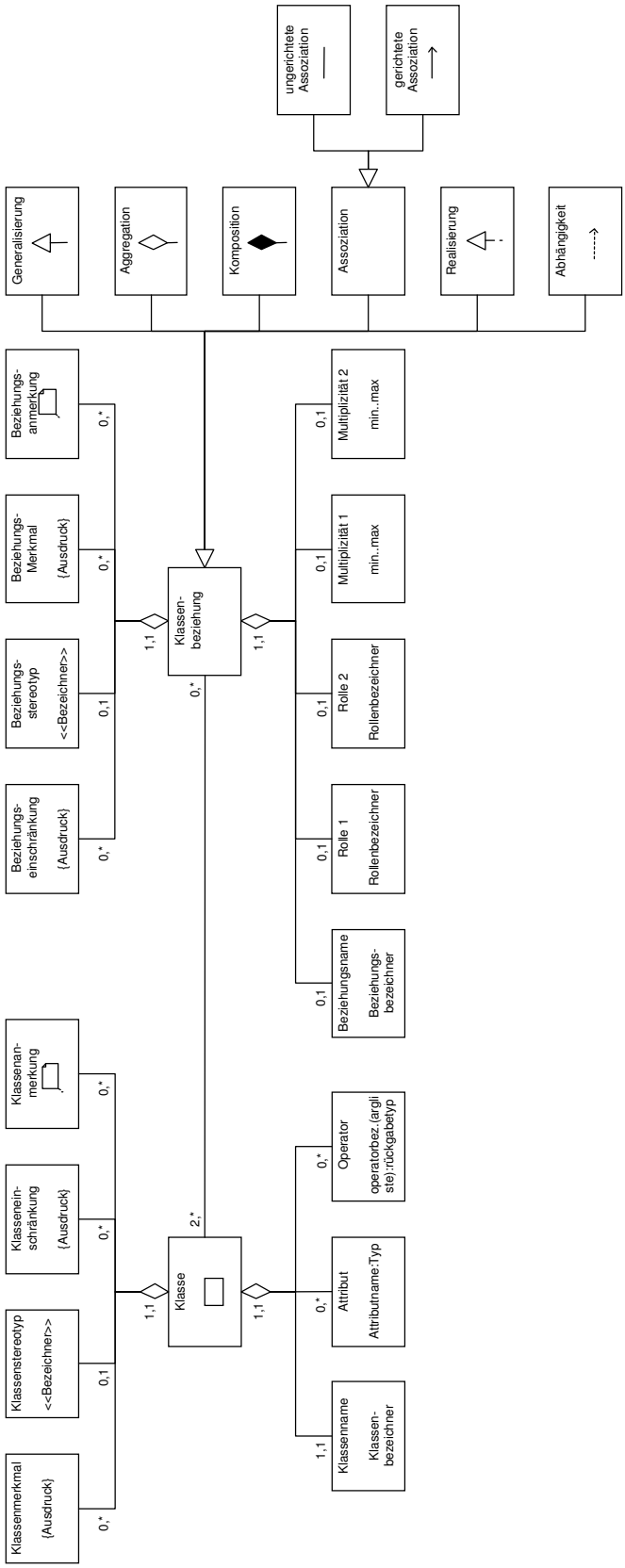


Abbildung 5.11: Repräsentations-Metamodell der statischen Struktur und des statischen Verhaltens eines objektorientierten Anwendungssystems in der Spezifikationsphase

Metamodell zur Repräsentation der statischen Struktur und des dynamischen Verhaltens in der Ausführungsphase

Abbildung 5.12 enthält das systemorientierte Repräsentations-Metamodell zur Erstellung von Kollaborationsdiagrammen. Wie bereits in Abschnitt 5.2.2.3 erwähnt, beruhen die in einem Kollaborationsdiagramm dargestellten Struktur- und Verhaltensmerkmale eines objektorientierten Anwendungssystems auf den in einem entsprechenden Klassendiagramm spezifizierten Struktur- und Verhaltensmerkmalen dieses Systems. Demnach weist dieses Metamodell hauptsächlich ein Metaobjekt für die Darstellung von Objekten und eine Metabeziehung für die Darstellung von Objektbeziehungen auf. Ein Objekt besitzt zumindest einen Namen und gegebenenfalls einen Stereotyp, mit dem die Art des Objekts angezeigt wird. Ferner kann ein Objekt mehrere Attributwerte besitzen. Zulässige Objektbeziehungsarten sind die Generalisierung, die Aggregation, die Komposition, die ungerichtete Assoziation, die gerichtete Assoziation und die Abhängigkeitsbeziehung. Eine Objektbeziehung verfügt eventuell über einen Namen, einen Stereotyp und höchstens zwei Rollen. Außerdem sind ihr üblicherweise mehrere Nachrichten zugeordnet, wobei jede Nachricht entweder synchron, asynchron oder eine Rückgabe sein kann. Zur Darstellung der zeitlichen Reihenfolge werden die Nachrichten der Objektbeziehungen mit Sequenznummern versehen. Ferner kann jede Nachricht anhand von Einschränkungen und Bedingungen ergänzt werden.

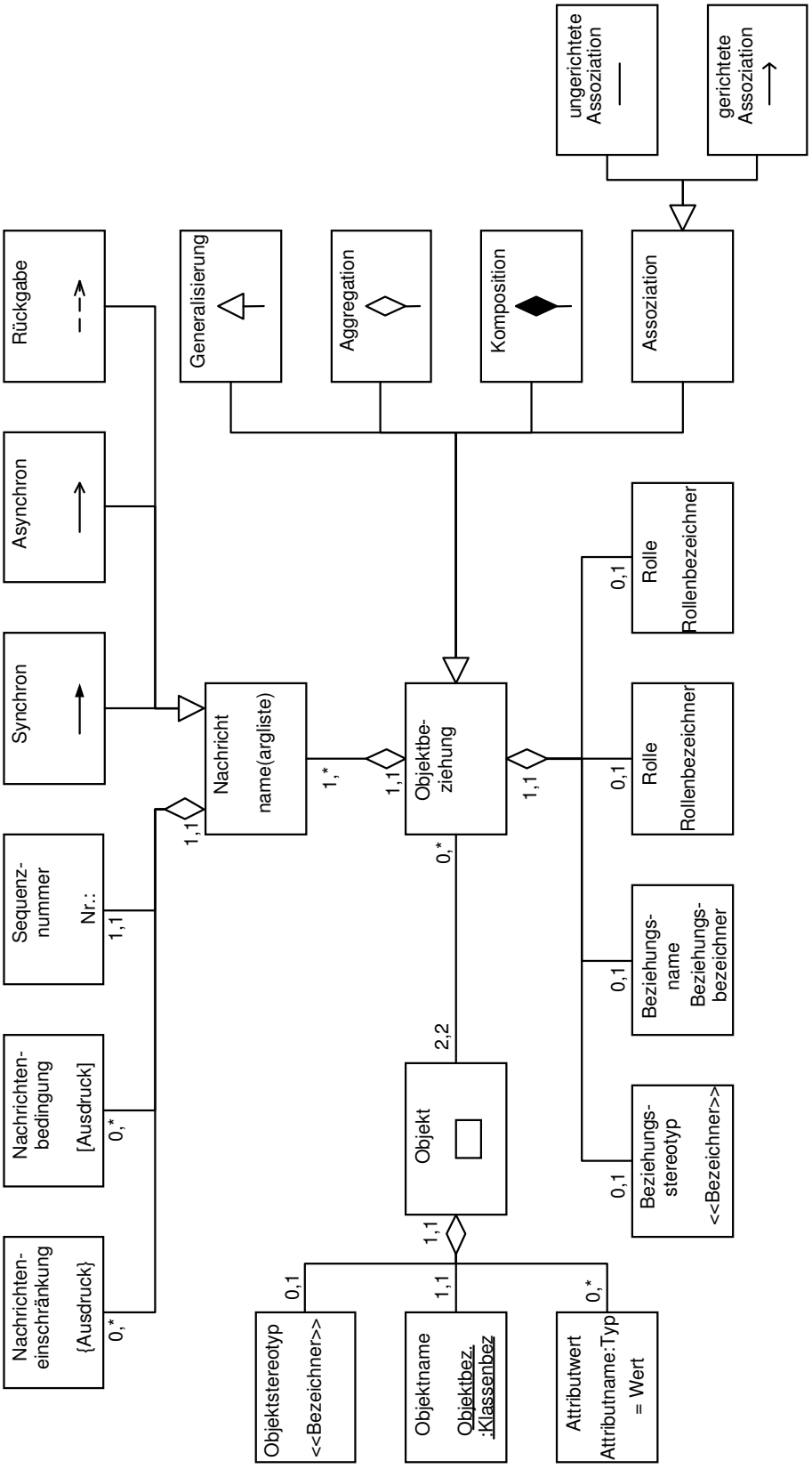


Abbildung 5.12: Repräsentations-Metamodell der statischen Struktur und des dynamischen Verhaltens eines objektorientierten Anwendungssystems in der Ausführungsphase

Metamodell zur Repräsentation des dynamischen Verhaltens in der Ausführungsphase

Das letzte Repräsentations-Metamodell dieser Modellebene zeigt Abbildung 5.13. Es wird zur Erstellung von Sequenzdiagrammen benötigt, mit deren Hilfe insbesondere das dynamische Verhalten objektorientierter Systeme in der Ausführungsphase dargestellt werden kann. Im Unterschied zu Kollaborationsdiagrammen, die die strukturellen Beziehungen zwischen den Objekten eines Systems betonen, wird in Sequenzdiagrammen die zeitliche Abfolge des Nachrichtenaustauschs zwischen diesen Objekten hervorgehoben. Zu diesem Zweck weist jedes Objekt eine so genannte Lebenslinie auf, die sich unterhalb des Objekts befindet. Diese Lebenslinie beginnt ab der Erzeugung des Objekts und endet mit seiner Löschung, die mit einem dafür vorgesehenen Symbol gekennzeichnet wird. Zwischen den Lebenslinien wird der Nachrichtenaustausch zwischen den Objekten dargestellt. Wenn ein Objekt durch den Empfang einer Nachricht aktiviert wird, kann dies mit einer entsprechenden Kennzeichnung seiner Lebenslinie kenntlich gemacht werden. Die Notation einer Nachricht entspricht im Wesentlichen der in Kollaborationsdiagrammen verwendeten, mit dem Unterschied, dass die Spezifikation von Sequenznummern entfällt, aber dafür mögliche Besonderheiten im zeitlichen Ablauf des Nachrichtenaustauschs durch informale Beschreibungen näher erläutert werden können.

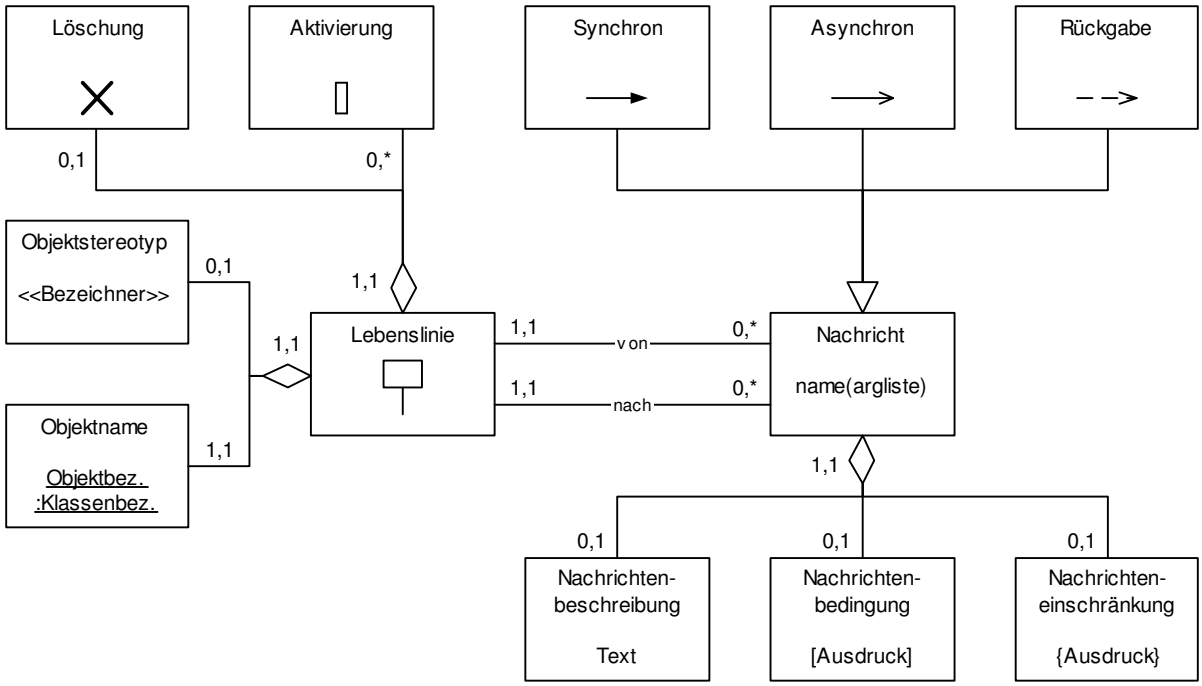


Abbildung 5.13: Repräsentations-Metamodell des dynamischen Verhaltens eines objektorientierten Anwendungssystems in der Ausführungsphase

5.3.2.4 Beziehungs-Metamodelle

Analog zur Außenperspektive-Modellebene werden die im letzten Abschnitt vorgestellten Repräsentations-Metamodelle anhand von Beziehungs-Metamodellen mit dem Kern-Metamodell aus Abschnitt 5.3.2.2 verknüpft. Aus den in Abschnitt 5.2.2.4 genannten Gründen erfolgt die Darstellung dieser Beziehungs-Metamodelle nicht grafisch, sondern tabellarisch.

Das erste Beziehungs-Metamodell setzt das Kern-Metamodell mit dem Metamodell zur Erstellung von Klassendiagrammen in Beziehung.

Metaobjekt Kern-Metamodell	Kardinalität 1	Metaobjekt Repräsentations-Metamodell Klassendiagramm	Kardinalität 2
Vorgangsklasse	1,1	Klasse mit Stereotyp „Vorgang“	0,1
Vorgangsklasse	1,1	Klasse mit Stereotyp „Unterstützung“	0,1
Konzeptuelle Klasse	1,1	Klasse mit Stereotyp „Entität“	0,1
Kommunikationsklasse	1,1	Klasse mit Stereotyp „Kommunikation“	0,1
Datenhaltungsklasse	1,1	Klasse mit Stereotyp „Datenhaltung“	0,1
Dienstklasse	1,1	Klasse mit Stereotyp „Dienst“	0,1
Klassenname	1,1	Klassenname	0,1
Operator mit Operatorsignatur und -implementierung	1,1	Operator	0,1
Attribut	1,1	Attribut	0,1
Generalisierung (Dienstklassenbeziehung oder Intra-Komponenten-Beziehung)	1,1	Generalisierung	0,1
Aggregation (Dienstklassenbeziehung oder Intra-Komponenten-Beziehung)	1,1	Aggregation oder Komposition (bei wechselseitiger Existenzabhängigkeit)	0,1

Metaobjekt Kern-Metamodell	Kardinalität 1	Metaobjekt Repräsentations-Metamodell Klassendiagramm	Kardinalität 2
Ungerichtete Assoziation (Dienstklassenbeziehung oder Intra-Komponenten-Beziehung)	1,1	Ungerichtete Assoziation	0,1
Gerichtete Assoziation (Dienstklassenbeziehung oder Intra-Komponenten-Beziehung)	1,1	Gerichtete Assoziation	0,1
Generalisierung (Inter-Komponenten-Beziehung)	1,1	Abhängigkeitsbeziehung	0,1
Aggregation (Inter-Komponenten-Beziehung)	1,1	Abhängigkeitsbeziehung	0,1
Ungerichtete Assoziation (Inter-Komponenten-Beziehung)	1,1	Abhängigkeitsbeziehung	0,1
Gerichtete Assoziation (Inter-Komponenten-Beziehung)	1,1	Abhängigkeitsbeziehung	0,1
Assoziationsname	1,1	Beziehungsname	0,1
Kardinalität 1	1,1	Multiplizität 1	0,1
Kardinalität 2	1,1	Multiplizität 2	0,1

Tabelle 5.5: Beziehungs-Metamodell zur Verbindung des Kern-Metamodells mit dem Metamodell zur Erstellung von Klassendiagrammen

Mit dem folgenden Beziehungs-Metamodell werden das Repräsentations-Metamodell zur Erstellung von Klassendiagrammen und das Repräsentations-Metamodell zur Erstellung von Kollaborationsdiagrammen miteinander verknüpft. Wie bereits in Abschnitt 5.2.2.4 gezeigt, kann in Verbindung mit dem oben vorgestellten Beziehungs-Metamodell damit auch die Beziehung zwischem dem Kern-Metamodell und dem letztgenannten Repräsentations-Metamodell hergestellt werden.

Metaobjekt Repräsentations-Metamodell Klassendiagramm	Kardinalität 1	Metaobjekt Repräsentations-Metamodell Kollaborations-diagramm	Kardinalität 2
Klasse mit Namen und Stereotyp	1,1	Objekt mit Namen und Stereotyp	0,*
Attribut	1,1	Attributwert	0,*
Operator	1,1	Nachricht	0,*
Generalisierung	1,1	Generalisierung	0,*
Aggregation	1,1	Aggregation	0,*
Komposition	1,1	Komposition	0,*
Ungerichtete Assoziation	1,1	Ungerichtete Assoziation	0,*
Gerichtete Assoziation	1,1	Gerichtete Assoziation	0,*
Abhängigkeitsbeziehung	1,1	Abhängigkeitsbeziehung	0,*
Beziehungsname	1,1	Beziehungsname	0,*
Beziehungsstereotyp	1,1	Beziehungsstereotyp	0,*
Beziehungseinschränkung	1,1	Nachrichteneinschränkung	0,*
Rolle 1	1,1	Rolle 1	0,*
Rolle 2	1,1	Rolle 2	0,*

Tabelle 5.6: Beziehungs-Metamodell zur Verbindung des Repräsentations-Metamodells zur Erstellung von Klassendiagrammen und des Repräsentations-Metamodells zur Erstellung von Kollaborationsdiagrammen

Das letzte Beziehungs-Metamodell in diesem Abschnitt veranschaulicht die Beziehungen zwischen dem Repräsentations-Metamodell zur Erstellung von Klassendiagrammen und dem Repräsentations-Metamodell zur Erstellung von Sequenzdiagrammen. Die Beziehungen zwischen dem Kern-Metamodell und dem letztgenannten Repräsentations-Metamodell können auch in diesem Fall aus diesem Beziehungs-Metamodell und dem in Tabelle 5.5 dargestellten Beziehungs-Metamodell abgeleitet werden.

Metaobjekt Repräsentations-Metamodell Klassendiagramm	Kardinalität 1	Metaobjekt Repräsentations-Metamodell Sequenzdiagramm	Kardinalität 2
Klasse mit Namen und Stereotyp	1,1	Lebenslinie mit Objektnamen und -stereotyp	0,*
Operator	1,1	Nachricht	0,*
Beziehungseinschränkung	1,1	Nachrichteneinschränkung	0,*

Tabelle 5.7: Beziehungs-Metamodell zur Verbindung des Repräsentations-Metamodells zur Erstellung von Klassendiagrammen und des Repräsentations-Metamodells zur Erstellung von Sequenzdiagrammen

5.3.3 Sichten

Bevor in diesem Abschnitt die ebenenspezifischen Sichten zur Unterstützung der Komplexitätsbewältigung vorgestellt werden, wird zunächst untersucht, inwieweit die in Abschnitt 2.2.4.1 beschriebenen relevanten Sichten auf Informations- bzw. Anwendungssysteme auf dieser Modellebene unterstützt werden. Da das in Abschnitt 3.5.1.1 erläuterte Grundkonzept der Objektorientierung sowohl im Spezifikationsparadigma als auch in den Metamodellen zum Ausdruck kommt, ist eine Unterstützung aller relevanten Sichten grundsätzlich gewährleistet. Die Daten- und Funktionssicht wird in Klassen bzw. Objekten vereinigt, durch Kommunikationskanäle zwischen Klassen bzw. Objekten ist die Interaktionssicht darstellbar und anhand von Kollaborations- und Sequenzdiagrammen lässt sich außerdem die Vorgangssicht beschreiben.

Damit auch auf dieser Modellebene die Komplexität komponentenbasierter Anwendungssysteme bewältigt werden kann, stehen zusätzliche, ebenenspezifische Sichten zur Verfügung. Die Definition dieser ebenenspezifischen Sichten erfolgt in diesem Fall zweistufig.

Belangsichten

Auf der obersten Stufe können eine funktionale und eine technische Sicht unterschieden werden. Die **funktionale Sicht** umfasst alle Klassen und Klassenbeziehungen, die die Innenperspektiven der auf der Außenperspektive-Modellebene entworfenen Software-Komponenten spezifizieren. Dazu gehören alle Vorgangsklassen, konzeptuellen Klassen, Kommunikationsklassen und Datenhaltungsklassen, einschließlich aller Intra- bzw. Inter-Komponenten-Beziehungen, über die sie verbunden sind. Dagegen enthält die **technische Sicht** nur die Dienstklassen und die zugehörigen Dienstebeziehungen, mit denen die vom komponentenbasierten Anwendungssystem benötigten Basismaschinendienste spezifiziert werden. Anhand der beiden Sichten lassen sich auf der Innenperspektive-Modellebene die fachlichen Belange von den technischen Belangen trennen.

Systemsichten

Die nächste Stufe definiert in beiden Sichten wiederum eine Struktursicht und eine Verhaltenssicht. Sie werden anhand der entsprechenden Systemmerkmale voneinander abgegrenzt. Die Struktursicht im Kern-Metamodell umfasst das Metaobjekt Klasse einschließlich aller zugeordneten Spezialisierungen sowie die Beziehungsart Klassenbeziehung, die zugehörigen Metaobjekte Kardinalität 1 und Kardinalität 2 und alle zugehörigen Spezialisierungen. Die korrespondierende Verhaltenssicht enthält demnach die Metaobjekte Operator, Attribut, Operatorsignatur, Operatorimplementierung und Nachricht. Im Repräsentations-Metamodell zur Erstellung von Klassendiagrammen sind, bis auf die Metaobjekte Attribut und Operator, die der Verhaltenssicht zugerechnet werden können, alle Metaobjekte der Struktursicht zuzuordnen. In der Struktursicht des Repräsentations-Metamodells zur Erstellung von Kollaborationsdiagrammen sind die Metaobjekte Objekt, die zugehörigen Metaobjekte Objektstereotyp und Objektname sowie die Beziehungsart Objektbeziehung einschließlich aller zugehörigen Spezialisierungen und Metaobjekte enthalten. Die Verhaltenssicht dieses Repräsentations-Metamodells weist folglich das Metaobjekt Attributwert und das Metaobjekt Nachricht inklusive aller zugehörigen Spezialisierungen und Metaobjekte auf. Mit dem Repräsentations-Metamodell zur Erstellung von Sequenzdiagrammen ist ausschließlich das dynamische Verhalten eines objektorientierten Systems beschreibbar. Deshalb werden alle Metaobjekte dieses Repräsentations-Metamodells der Verhaltenssicht zugeordnet. Die zugehörige Struktursicht weist somit keine Metaobjekte auf.

5.3.4 Muster

Für die folgenden Erläuterungen der Muster auf der Innenperspektive-Modellebene muss unterschieden werden zwischen Mustern für die funktionale Sicht und Mustern für die technische Sicht.

Muster für die funktionale Sicht

Wie bereits im Grundkonzept der Komponentenorientierung erläutert, lässt sich der objektorientierte Entwurf von Software-Komponenten-Implementierungen durch den Einsatz von Entwurfsmustern unterstützen. Dabei werden Entwurfsmuster entweder zur Erstellung oder zur Dokumentation von Software-Komponenten-Implementierungen eingesetzt [Grif98, 112; CHJK02, 39]. Domänenunabhängige Entwurfsmuster sind beispielweise im Entwurfsmusterkatalog von GAMMA ET AL. [GHJV96], in den PLoP (Pattern Languages of Programs)-Tagungsbänden (siehe z. B. [CoSc95; VlCK96; MaRB98; Risi98]) oder auf speziellen Internetseiten (siehe z. B. [Hill03]) veröffentlicht. Die beiden letztgenannten Quellen enthalten darüber hinaus auch Entwurfsmuster für domänenspezifische Problemstellungen.

Muster für die technische Sicht

Im Unterschied zu den eben genannten Mustern für die funktionale Sicht enthalten die Muster für die technische Sicht heuristisches Entwurfswissen, das für den Entwurf der Basismaschinendienste und der Dienstebeziehungen zwingend notwendig ist und mit dem Kern-Metamodell nicht ausgedrückt werden kann. Erst durch die Verwendung dieser Muster kann gewährleistet werden, dass die entworfenen Basismaschinendienste und Dienstebeziehungen die im Grundkonzept der Komponentenorientierung genannten Anforderungen erfüllen.

Auswahl geeigneter Mustersysteme und -sprachen

Für den Entwurf technischer Struktur- und Verhaltensmerkmale verteilter, komponentenbasierter Anwendungssysteme stehen inzwischen mehrere Architektur- und Entwurfsmuster zur Verfügung (siehe z. B. [Schm02; AlCM02; Mari02]). Allerdings setzen viele den Einsatz bestimmter Entwicklungs- und Ausführungsplattformen voraus. Dies hat zur Folge, dass diese Muster sehr implementierungsnah und somit für den softwaretechnischen Entwurf nicht ausreichend abstrakt sind. Im vorliegenden Fall werden jedoch Muster benötigt, die den objektorientierten Entwurf von Basismaschinendiensten und Ausführungsplattformen für komponentenbasierte Anwendungssysteme unterstützen und dabei so abstrakt wie möglich und so konkret wie nötig sind. Darüber hinaus sollten die Basismaschinendienste und Ausführungsplattformen, die anhand der Muster entworfen werden können, bereits für verschiedene komponentenbasierte Anwendungssysteme erfolgreich realisiert worden sein und sich demzufolge bewährt haben. Diesen Ansprüchen können derzeit nur die Architektur- und Entwurfsmuster von BUSCHMANN ET AL. [BMR+98] und VÖLTER ET AL. [VöSW02] gerecht werden.

BUSCHMANN ET AL. [BMR+98] enthält ein Mustersystem, das ausdrücklich die Erstellung von Software-Architekturen unterstützt [BMR+98, XI]. Dazu stellt es hauptsächlich Architekturmuster zur Strukturierung eines Anwendungssystems in Subsysteme und Entwurfsmuster zur Verfeinerung von Subsystemen zur Verfügung [BMR+98, 12 f]. Einige dieser Muster bauen auf den grundlegenden Entwurfsmustern des Musterkatalogs von GAMMA ET AL. [GHVJ96] auf bzw. sind aus diesem übernommen [BMR+98, 408]. Die Auswahl der übernommenen Entwurfsmuster in diesem Mustersystem orientiert sich an deren Bedeutung für den Entwurf von Software-Architekturen. Alle enthaltenen Muster sind domänen-, programmierparadigmen- und programmiersprachenunabhängig [BMR+98, 12 f].

Die Architektur- und Entwurfsmuster von **VÖLTER ET AL.** [VöSW02] bilden zusammen eine Mustersprache, die die Erstellung einer Infrastruktur für serverseitige Software-Komponenten unterstützt [VöSW02, 27]. Nach VÖLTER ET AL. ist eine Software-Komponente dann serverseitig bzw. verteilt, wenn sie zumindest aus der Entfernung

zugreifbar ist und zu diesem Zweck bestimmte Dienste in Anspruch nimmt [VöSW02, 4]. Wie bereits in Abschnitt 3.1.3.6 erwähnt, sind die Beziehungen zwischen Mustern in einer Mustersprache kohärenter als in einem Mustersystem. Insbesondere müssen die Muster einer Mustersprache in einer vorgegebenen Reihenfolge angewendet werden, damit das übergreifende Entwurfsziel, das mit der Mustersprache erreicht werden soll, auch erreichbar ist [VöSW02, 9]. Die praktische Anwendbarkeit der Mustersprache von VÖLTER ET AL. wird anhand ihrer Verwendung zur Dokumentation der Enterprise JavaBeans-Plattform und im Rahmen einer exemplarischen Fallstudie nachgewiesen [VöSW02, 219 ff].

Musterbasierte Spezifikation einer Ausführungsplattform für komponentenbasierte Anwendungssysteme

Im folgenden werden geeignete Beziehungen zwischen den von einem komponentenbasierten Anwendungssystem benötigten Basismaschinendiensten anhand der Muster von BUSCHMANN ET AL. und VÖLTER ET AL. identifiziert und auf diesem Weg eine gemäß dem Grundkonzept der Komponentenorientierung vollständige **Ausführungsplattform für komponentenbasierte Anwendungssysteme** konstruiert. Die benötigten Basismaschinendienste lassen sich aus dem Grundkonzept der Komponentenorientierung herleiten. Da die verwendeten Muster ausreichend abstrakt sind, wird eine Ausführungsplattform beschrieben, deren Implementierung nicht vorweg genommen wird und somit durch verschiedene Basismaschinen realisiert werden kann. In Abschnitt 5.3.6 werden dafür geeignete Basismaschinenkonzepte vorgestellt. Als zweckmäßige Vorgehensweise für die Konstruktion der Ausführungsplattform bietet sich die schrittweise Verfeinerung an. Das heißt, dass die Ausführungsplattform zunächst aus der Außensicht spezifiziert und daraufhin deren innere Struktur und damit auch deren Verhalten schrittweise festgelegt wird.

Muster zur Spezifikation der Außensicht einer Ausführungsplattform

Die grundsätzliche Aufgabe einer Ausführungsplattform besteht in der Trennung der technischen Belange von den funktionalen Belangen einer Software-Komponente bzw. eines komponentenbasierten Anwendungssystems. Diese Trennung sollte jedoch für den Nutzer einer Software-Komponente bzw. eines komponentenbasierten Anwendungssystems transparent sein. Darüber hinaus sollten die technischen Belange nicht für jede Software-Komponente individuell, sondern für komponentenbasierte Anwendungssysteme insgesamt entworfen werden. Diesen Anforderungen kann mit dem Muster ***Container*** der Mustersprache von VÖLTER ET AL. begegnet werden [VöSW02, 43 ff]. Das zugehörige Lösungsverfahren sieht vor, einem komponentenbasierten Anwendungssystem eine **Ausführungsumgebung** für Software-Komponenten zur Verfügung zu stellen, die deren technische Belange für den Nutzer transparent implementiert [VöSW02, 44]. „Conceptually, it wraps the components,

thus giving clients the illusion of tightly-integrated functional and technical concerns“ [VöSW02, 44].

Die genannte Ausführungsumgebung entspricht der Außensicht der zu konstruierenden Ausführungsplattform. In den folgenden Erläuterungen wird nun die korrespondierende Innensicht schrittweise festgelegt.

Muster zur Spezifikation der Innensicht einer Ausführungsplattform

Zunächst müssen die funktionalen und nicht-funktionalen Anforderungen, denen eine solche Ausführungsplattform begegnen muss, identifiziert werden. Diese bestimmen maßgeblich die Grobstruktur der Ausführungsplattform. Wie erwähnt, implementiert eine Ausführungsplattform die technischen Belange einer unbestimmten Anzahl von unterschiedlichen komponentenbasierten Anwendungssystemen. Dabei exisitiert jedoch eine Anzahl technischer Belange, die alle komponentenbasierten Anwendungssysteme gleichermaßen aufweisen. Allerdings kann jedes komponentenbasierte Anwendungssystem auch zusätzliche technische Belange aufweisen, die von der Ausführungsplattform individuell behandelt werden müssen. Zu diesen funktionalen Anforderungen, die an eine Ausführungsplattform gestellt werden, kommen noch weitere nicht-funktionale Anforderungen. Sie resultieren vorwiegend aus den nicht-funktionalen Anforderungen, die an Software-Komponenten und komponentenbasierte Anwendungssysteme gerichtet werden, da diese die Ausführungsplattform als Basismaschine nutzen. Danach muss eine Ausführungsplattform die Heterogenität der zugrundeliegenden Basismaschinen bewältigen können. Des Weiteren wird von ihr verlangt, dass sie offen, das heißt portabel und interoperabel ist. Schließlich muss sie erweiterbar und anpassbar sein, damit neue Basismaschinen einfach integriert werden können.

Diese funktionalen und nicht-funktionalen Anforderungen werden auch mit dem Architekturmuster ***Microkernel*** des Mustersystems von BUSCHMANN ET AL. adressiert (vgl. Abbildung 5.14) [BMR+98, 171 ff]. Das zugehörige Lösungsverfahren unterteilt eine Anwendungsplattform in einen Mikrokern (*Microkernel*) und gegebenenfalls mehrere interne Server (*Internal Server*), externe Server (*External Server*) und Adapter.

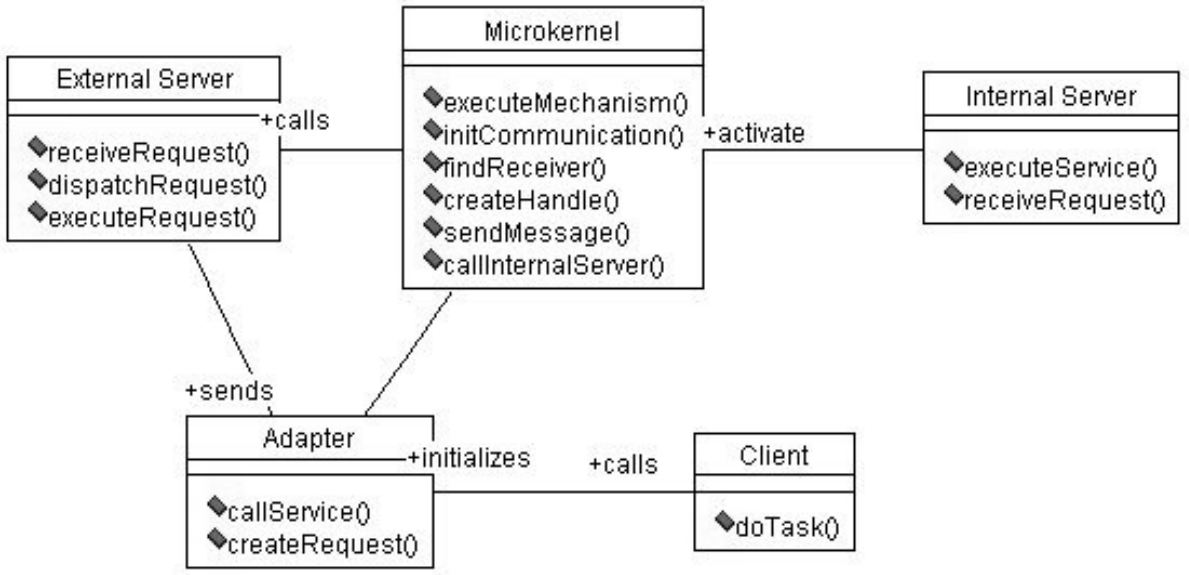


Abbildung 5.14: Architekturmuster *Microkernel* [BMR+03a]

Der **Mikrokern** kapselt die grundlegenden Dienste der Anwendungsplattform. Dazu gehören beispielsweise Mechanismen zur **Interprozesskommunikation** und die Verwaltung und Kontrolle von Ressourcen. Außerdem stellt er Schnittstellen zur Verfügung, über die die anderen Systemkomponenten der Anwendungsplattform auf seine Dienste zugreifen können [BMR+98, 173 f]. In **internen Servern** werden weitere grundlegende Dienste bereitgestellt, die sich nicht im Mikrokern implementieren lassen, ohne seine Größe oder Komplexität unnötig zu erhöhen [BMR+98, 173]. Der Mikrokern ruft die Dienste der internen Server mithilfe von **internen Dienstanforderungen** auf. Somit können interne Server auch zur Kapselung von Basismaschinenabhängigkeiten verwendet werden [BMR+98, 175]. **Externe Server** enthalten anwendungsspezifische Dienste, die auf den grundlegenden Diensten des Mikrokerns aufbauen. Der Zugriff auf die Dienste des Mikrokerns erfolgt über die dafür vorgesehenen Schnittstellen [BMR+98, 173]. Ein externer Server veröffentlicht seine Dienste an externe Nutzer über entsprechende Schnittstellen. Jeder externe Server empfängt Dienstanforderungen von den externen Nutzern anhand der vom Mikrokern bereitgestellten **Kommunikationsmechanismen**, führt die angeforderten Dienste mithilfe der Dienste des Mikrokerns und der internen Server aus und gibt die Ergebnisse an die Nutzer zurück [BMR+98, 176]. Demnach greift ein externer Nutzer nur auf die Schnittstellen der externen Server zu und nicht auf die des Mikrokerns. Wenn jedoch ein externer Nutzer direkt auf die Schnittstellen eines externen Servers zugreift, entsteht eine Abhängigkeit zwischen dem externen Nutzer und den Schnittstellen des externen Servers. Dies verhindert die Portabilität, die Interoperabilität, die Erweiterbarkeit und die Anpassbarkeit der gesamten Anwendungsplattform. Deswegen wird ein **Adapter** zwischen dem externen Nutzer und den Schnittstellen des externen Servers eingeführt, der die Schnittstelle des externen Servers emuliert [BMR+98, 176

f]. Dieser Adapter befindet sich im Adressraum des externen Nutzers und verbirgt somit unter anderem die erforderlichen Mechanismen zur Kommunikation mit dem externen Server vor dem Nutzer [BMR+98, 177]. Der Aufruf eines vom externen Server angebotenen Dienstes an der Schnittstelle des Adapters wird mithilfe der vom Mikrokern bereitgestellten Kommunikationsmechanismen an den externen Server weitergeleitet [BMR+98, 177].

Dieses Architekturmuster wird nun zur Festlegung der Grobstruktur der Ausführungsplattform verwendet. Demnach stellen die Software-Komponenten die externen Server dar, die auf den Mikrokern der Ausführungsplattform zugreifen. Der Mikrokern wiederum verwendet zur Realisierung der technischen Belange mehrere interne Server.

Eine Ausführungsplattform soll jedoch die technischen Belange aller Software-Komponenten eines komponentenbasierten Anwendungssystems implementieren. Deswegen muss sie für die unterschiedlichen technischen Anforderungen dieser Software-Komponenten anpassbar sein.

Dieses Problem wird mit dem Muster ***Glue-Code Layer*** der Mustersprache von VÖLTER ET AL. adressiert [VöSW02, 108 ff]. Im zugehörigen Lösungsverfahren wird vorgeschlagen, dass für jede Software-Komponente ein Adapter erzeugt wird, der die generischen Teile der Ausführungsplattform an die spezifischen technischen Anforderungen der Software-Komponente, die ihr in Form einer entsprechenden Deklaration mitgegeben worden sind, anpasst [VöSW02, 109].

Resultierende Grobstruktur der Ausführungsplattform

Die gemeinsame Anwendung des *Microkernel*- und des *Glue-Code-Layer*-Musters ergibt schließlich die in Abbildung 5.15 gezeigte Grobstruktur der Ausführungsplattform.

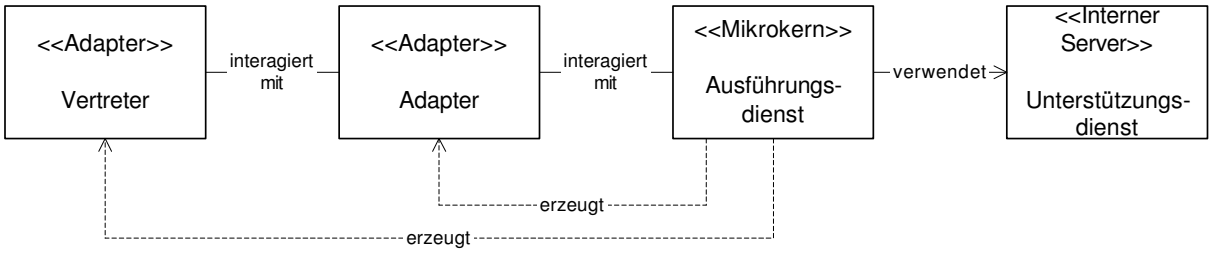


Abbildung 5.15: Grobstruktur der Ausführungsplattform

Zur Repräsentation des Musters sowie aller noch folgenden Muster wird das in Abschnitt 5.3.2.3 vorgestellte systemorientierte Repräsentations-Metamodell zur Erstellung von Klassendiagrammen verwendet. Die Software-Komponente, als externer Server des *Microkernel*-Musters, ist in dieser Grobstruktur nicht enthalten, da sie

selbst kein Basismaschinendienst ist, sondern die aufgeführten Basismaschinendienste nutzt. Sie wird demnach nur auf der Außenperspektive-Modellebene spezifiziert.

Auf der Grundlage der Grobstruktur kann das folgende Verhalten der Ausführungsplattform grob skizziert werden.

Resultierendes Verhalten der Ausführungsplattform

Zunächst wird vom **Ausführungsdienst** eine Software-Komponenten-Vertreter-Instanz (kurz: **Vertreterinstanz**) und eine **Adapterinstanz** erzeugt. Daraufhin kann ein Nutzer auf die von einer Vertreterinstanz emulierte Schnittstelle einer Software-Komponenten-Instanz zugreifen. Die Vertreterinstanz baut eine **Dienstanforderung** auf und fragt über die Adapterinstanz den Ausführungsdienst um eine Kommunikationsverbindung mit der entsprechenden Software-Komponenten-Instanz an. Der Ausführungsdienst stellt die konkrete Adresse der Software-Komponenten-Instanz fest und schickt sie über die Adapterinstanz zurück zur Vertreterinstanz. Nachdem die Vertreterinstanz diese Information bekommen hat, richtet sie eine direkte Kommunikationsverbindung zur Software-Komponenten-Instanz ein. Diese adressiert zur Erbringung der geforderten technischen Funktionalität wiederum über die Adapterinstanz die Schnittstelle des Ausführungsdienstes. Der Ausführungsdienst führt die auf diese Weise angeforderten Dienste entweder direkt aus, oder leitet die Dienstanforderungen an die **Unterstützungsdienste** in geeigneter Form weiter.

Dieser Grobstruktur der Ausführungsplattform werden nun die im Grundkonzept der Komponentenorientierung geforderten Basismaschinendienste eines komponentenbasierten Anwendungssystems zugeordnet. Dabei sind zunächst die Basismaschinendienste von Interesse, die eine einzelne Software-Komponente benötigt.

Zuordnung von Basismaschinendiensten für Software-Komponenten zur Grobstruktur der Ausführungsplattform

Danach müssen geeignete Basismaschinendienste die in der Deklaration der Software-Komponente festgelegten nicht-funktionalen Eigenschaften und Eigenschaftswerte, die sich nicht auf die Verteilung der Software-Komponente beziehen, erbringen. Ferner muss ein geeigneter Basismaschinendienst die Software-Komponenten-Fabrik implementieren. Dies wird dem Ausführungsdienst übertragen, da er unter anderem für die Verwaltung und Kontrolle von Ressourcen verantwortlich ist. Für die Interoperablilität und Portabilität der Software-Komponenten-Instanzen sorgen die Vertreterinstanzen, über die ein Nutzer auf die Software-Komponenten-Instanzen zugreift. Schließlich übernimmt der Ausführungsdienst das dynamische Binden einer Operatorsignatur zu der zugehörigen Operatorimplementierung zur Ausführungszeit und kontrolliert außerdem die Einhaltung der semantischen Integritätsbedingungen,

die in den Schnittstellen der Software-Komponente als codierte Zusicherungen vorliegen.

Zuordnung von Basismaschinendiensten für komponentenbasierte Anwendungssysteme zur Grobstruktur der Ausführungsplattform

Im Folgenden werden die von einem komponentenbasierten Anwendungssystem angeforderten Basismaschinendienste auf die Grobstruktur der Ausführungsplattform abgebildet. Da ein komponentenbasiertes Anwendungssystem ein verteiltes System darstellt, wird den folgenden Erläuterungen eine entsprechende Variante des *Microkernel*-Musters zugrunde gelegt. In dieser Variante, mit der Bezeichnung ***Distributed-Microkernel-System***, agiert ein Mikrokern auch als Nachrichtenübermittler, der für das Verschicken von Nachrichten an entfernte Rechnersysteme oder den Empfang von ihnen verantwortlich ist [BMR+98, 188]. Folglich implementiert jedes Rechnersystem in einem verteilten System sein eigenes Mikrokern-System. Für den Nutzer erscheint das gesamte verteilte System jedoch als ein einziges Mikrokern-System, d. h. die Verteilung bleibt für den Nutzer transparent [BMR+98, 188].

Erste Verfeinerung der Grobstruktur

Damit die Mikrokerne als Nachrichtenübermittler der genannten Art agieren können, benötigen sie zusätzliche Dienste für die Kommunikation. Grundsätzlich sollten diese Dienste entfernte und ortstransparente Aufrufe der von externen Servern bereitgestellten Dienste ermöglichen. Ferner müssen sie dafür sorgen, dass externe Server zur Ausführungszeit ausgewechselt, hinzugefügt oder entfernt werden können. Nicht zuletzt sollten sie system- und implementierungsspezifische Details der Interprozesskommunikation vor den Nutzern der externen Server verbergen.

Der Entwurf solcher Dienste wird durch das Muster ***Broker*** des Mustersystems von BUSCHMANN ET AL. unterstützt (vgl. Abbildung 5.16) [BMR+98, 99 ff]. Das Lösungsverfahren dieses Musters sieht hauptsächlich die Einführung eines **Vermittlers** (*Broker*) vor, der zur Entkopplung von Servern und Clients beiträgt.

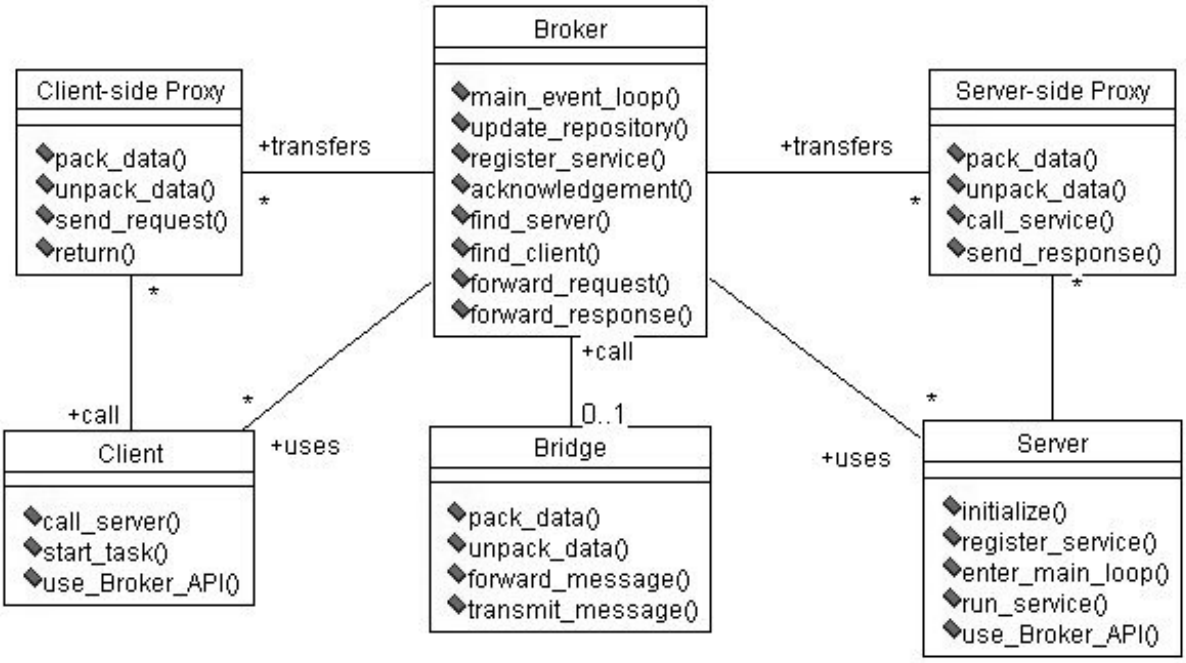


Abbildung 5.16: Architekturmuster *Broker* [BMR+03b]

Eine Software-Komponente kann im vorliegenden Fall sowohl ein Server als auch ein Client sein. Ein Server stellt potenziellen Clients seine Dienste über eine Schnittstelle zur Verfügung. Dafür meldet er sich zunächst selbständig bei einem Vermittler an und übergibt ihm Informationen über seinen Ort und seine Aktivierung [BMR+98, 108]. Danach wartet er auf Dienstanfragen von Clients. Diese schicken ihre Dienstanfragen jedoch nicht direkt an den Server, sondern an den Vermittler, der daraufhin die Lokalisierung des entsprechenden Servers, die Übermittlung der Dienstanfrage sowie die Rückübermittlung der Anworten und Fehlermeldungen übernimmt [BMR+98, 103]. Somit müssen Clients nicht wissen, wo sich der dienstanbietende Server befindet. Dies ist eine wesentliche Voraussetzung für die ortstransparente Kommunikation und das Auswechseln, Hinzufügen und Entfernen von Servern zur Ausführungszeit [BMR+98, 102]. Falls sich der dienstanbietende Server an einem anderen Vermittler als an dem angefragten angemeldet hat, sucht der angefragte Vermittler eine Route zu dem Vermittler, an dem sich der dienstanbietende Server angemeldet hat, und leitet die Anfrage über die gefundene Route weiter [BMR+98, 103]. Die Kommunikation zwischen verschiedenen Vermittlern erfolgt grundsätzlich über eine **Brücke** (*Bridge*), die die system- und implementierungsspezifischen Unterschiede zwischen den Vermittlern ausgleicht [BMR+98, 106]. Ein Client kommuniziert mit einem Vermittler ebenfalls nicht direkt, sondern über einen **clientseitigen Vertreter** des Servers (*Client-side Proxy*). Dieser unterstützt die geforderte Ortstransparenz, da der entfernte Server für den Client wie ein lokaler Server erscheint [BMR+98, 104]. Außerdem verbirgt er system- und implementierungsspezifische Details der Interprozesskommunikation vor dem Client. Dazu gehören beispielsweise

der gewählte Mechanismus der Interprozesskommunikation und das **Externalisieren** von Anfragen bzw. das **Internalisieren** von Ergebnissen und Fehlermeldungen [BMR+98, 105]. Analog dazu erfolgt die Kommunikation zwischen einem Vermittler und einem Server über einen **serverseitigen Vertreter** des Servers (*Server-side Proxy*). Dieser sorgt folglich für den Empfang von Anfragen, deren Internalisierung, den Aufruf des entsprechenden Dienstes, die Externalisierung der Ergebnisse und Fehlermeldung und deren Versenden an den Vermittler [BMR+98, 105].

Von diesem Muster existieren mehrere Varianten, die beispielsweise in [BMR+98, 117 f] erläutert wurden. Für eine möglichst abstrakte Beschreibung der Struktur- und Verhaltensmerkmale der Ausführungsplattform wird jedoch die erläuterte Grundform verwendet. Das *Broker*-Muster enthält das Entwurfsmuster ***Proxy*** für die Unterstützung der Verteilungstransparenz durch client- und serverseitige Vertreter und das Entwurfsmuster ***Forwarder-Receiver*** für das Verbergen von system- und implementierungsspezifischen Details der Interprozesskommunikation als Bausteine. Beide Muster sind ebenfalls im Mustersystem von BUSCHMANN ET AL. enthalten [BMR+98, 263 ff und 308 ff].

Verfeinerte Struktur der Ausführungsplattform

Durch die Anwendung des *Broker*-Musters auf die oben dargestellte Grobstruktur der Ausführungsplattform wird diese verfeinert. Abbildung 5.17 zeigt das Ergebnis der Muster-Anwendung.

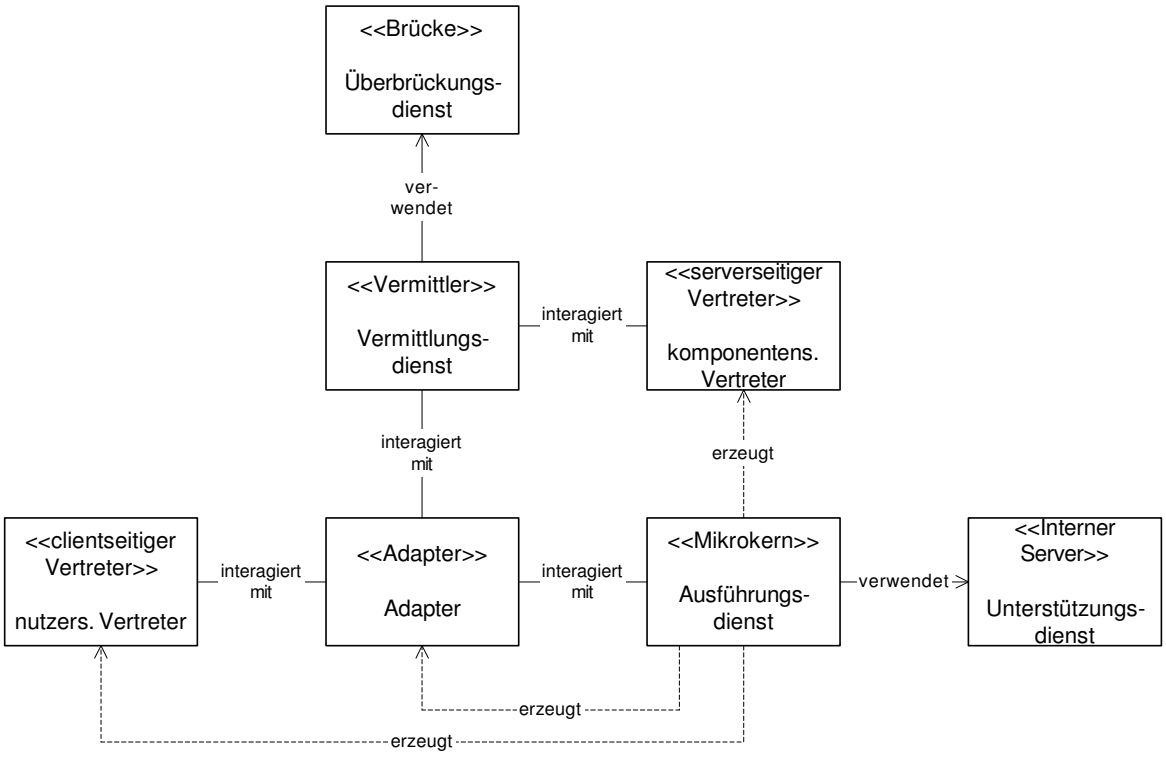


Abbildung 5.17: Verfeinerte Struktur der Ausführungsplattform

Zuordnung von Basismaschinendiensten für komponentenbasierte Anwendungssysteme zur verfeinerten Struktur der Ausführungsplattform

Auf der Grundlage dieser verfeinerten Struktur können nun die von einem komponentenbasierten Anwendungssystem geforderten Basismaschinendienste den Systemkomponenten der Ausführungsplattform zugeordnet werden. Demzufolge ist der Vermittlerdienst für die geforderte Gewährleistung eines minimalen Verküpfungsgrades und die Kontrolle eines geregelten Nachrichtenaustauschs zuständig. Die Sicherstellung der Konsistenz durch Erhaltung der semantischen und operationalen Integrität sowie die Kontrolle der Funktionsredundanz übernimmt der Ausführungsdienst. Spezielle Unterstützungsdienste sorgen für die Kontrolle der Datenredundanz.

Zur Gewährleistung der Verteilungstransparenz wurden in den vorangegangenen Erläuterungen zum *Broker*-Muster bereits geeignete Systemkomponenten aufgeführt. Allerdings sind dabei nicht alle im Grundkonzept der Komponentenorientierung genannten Teiltransparenzen berücksichtigt worden. Dies wird in den folgenden Ausführungen nachgeholt.

Die wichtigsten Teiltransparenzen für verteilte Anwendungssysteme im Allgemeinen und für komponentenbasierte Anwendungssysteme im Speziellen sind die Zugriffs- und die Ortstransparenz. Sie tragen wesentlich zur Nutzung verteilter Ressourcen, wie z. B. Software-Komponenten bei [CoDK01, 24].

Die **Zugriffstransparenz** sorgt dafür, dass die Operatoren, die von verteilten Software-Komponenten-Instanzen angeboten werden, stets mit demselben Zugriffsmechanismus aufrufbar sind. Somit wird zum einen von unterschiedlichen Zugriffsmechanismen bei lokalen und entfernten Software-Komponenten-Instanzen abstrahiert und zum anderen von spezifischen Aufrufstrukturen, Datentypen und Datenstrukturen beim Zugriff auf die Software-Komponenteinstanz [Putm01, 393; CoDK01, 23; TaST02, 5 f; BMR+98, 264]. Für die Zugriffstransparenz sorgt die nutzerseitige Vertreterinstanz, die die Schnittstelle der von ihr vertretenen Software-Komponente emuliert und vor bzw. nach der Kommunikation mit der entsprechenden Software-Komponenten-Instanz noch zusätzliche Vor- bzw. Nachverarbeitungen durchführt. Dabei erfolgt die Kommunikation mit der Software-Komponenten-Instanz über den Vermittlerdienst.

Ziel der **Ortstransparenz** ist das Verbergen des physischen Ortes einer Komponente vor ihrem Nutzer. Dabei muss zwischen dem Zeitpunkt vor dem Binden und dem Zeitpunkt nach dem Binden einer Software-Komponenten-Instanz an einen Nutzer unterschieden werden.

Vor dem Binden sollte ein Nutzer nicht selbst nach einer benötigten Software-Komponenten-Instanz im verteilten Anwendungssystem suchen müssen, sondern dafür einen zentralen Dienst beauftragen können. Dieses Problem wird mit dem Muster ***Naming*** der Mustersprache von VÖLTER ET AL. adressiert [VöSW02, 121 ff]. Das zugehörige Lösungsverfahren sieht die Einrichtung eines **Namensdienstes** vor, der Software-Komponenten und anderen Ressourcen strukturierte Namen zuordnet [VöSW02, 122]. Auf den Namensdienst wird üblicherweise über einen Vermittler zugegriffen. Demnach entspricht der Namensdienst einem internen Server, der dem Vermittlerdienst zugeordnet ist. Da die **Migrationstransparenz** eine Ortsänderung einer Software-Komponenten-Instanz vor einem Nutzer verbirgt, bevor sie von diesem gebunden wird, kann sie gleichfalls durch den Namensdienst gewährleistet werden.

Die Ortstransparenz nach dem Binden wird, wie bereits erläutert, durch die Verwendung eines nutzerseitigen Vertreters realisiert. Im Unterschied zur Migrationstransparenz verbirgt die **Relokationstransparenz** einen Ortswechsel einer Software-Komponenten-Instanz vor einem Nutzer, nachdem sie von ihm gebunden wurde. Deswegen wird auch für diesen Zweck der nutzerseitige Vertreter verwendet.

Ziel der **Replikationstransparenz** ist das Verbergen der Nutzung von redundanten Software-Komponenten-Instanzen zur Erhöhung der Zuverlässigkeit und der Leistungsfähigkeit eines verteilten Anwendungssystems vor dem Nutzer [CoDK01, 23]. Diesen Anforderungen kann mit dem Muster ***Virtual Instance*** der Mustersprache von VÖLTER ET AL. begegnet werden [VöSW02, 92 ff]. Im Lösungsverfahren des Musters wird unterschieden zwischen einer logischen und einer physischen Software-Komponenten-Instanz, die getrennt voneinander existieren [VöSW02, 93]. Die **logische Software-Komponenten-Instanz** wird von der nutzerseitigen Vertreterinstanz realisiert, die **physische Software-Komponenten-Instanz** entspricht der Software-Komponenten-Instanz selbst. Folglich kann ein Nutzer nur auf die logische Software-Komponenten-Instanz zugreifen. Da der Ausführungsdienst unter anderem für die Bereitstellung von Software-Komponenten-Instanzen zuständig ist, übernimmt er auch die bedarfsgesteuerte Erzeugung, Verwaltung und Vernichtung von redundanten Software-Komponenten-Instanzen und deren Zuordnung zu einer Vertreterinstanz. Dabei muss der Ausführungsdienst auch für die Erhaltung der Konsistenz sorgen, die durch die Verwendung von redundanten Software-Komponenten-Instanzen gefährdet sein kann. Zu diesem Zweck steht ihm wiederum ein **Transaktionsdienst** als interner Server zur Verfügung.

Der Transaktionsdienst wird gemeinsam mit dem Ausführungsdienst auch zur Realisierung der **Transaktionstransparenz** benötigt. Dabei sorgt der Ausführungsdienst dafür, dass die erforderliche Inanspruchnahme des Transaktionsdienstes vor dem

Nutzer einer Software-Komponenten-Instanz verborgen bleibt. In analoger Weise werden auch die **Fehlertransparenz**, die **Persistierungstransparenz**, die **Nebenläufigkeitstransparenz** und die **Sicherheitstransparenz** gewährleistet. Dafür stehen dem Ausführungsdienst ein **Fehlerbehandlungsdienst**, ein **Persistierungsdienst**, ein **Synchronisierungsdienst** und ein **Zugriffskontrolldienst** zur Verfügung, die er ebenfalls bedarfsgesteuert aufruft. Alle genannten Dienste stellen in Bezug auf den Ausführungsdienst interne Server dar.

Die **Skalierungstransparenz** verlangt, dass eine Anpassung der Anzahl physischer Software-Komponenten-Instanzen an die aktuelle Auslastung des Anwendungssystems vor dem Nutzer verborgen bleibt. Dies wird, analog zur Replikationstransparenz, von der Vertreterinstanz und dem Ausführungsdienst gemeinsam verwirklicht. Der Ausführungsdienst erhöht oder verringert die Anzahl der Software-Komponenten-Instanzen nach Bedarf und ordnet sie der Vertreterinstanz zu. Der Nutzer greift wiederum nur auf die Vertreterinstanz zu.

Endgültige Struktur der Ausführungsplattform

Abbildung 5.18 zeigt die endgültige Struktur der Ausführungsplattform, in der die genannten Ergänzungen zur Unterstützung der Teiltransparenzen berücksichtigt sind.

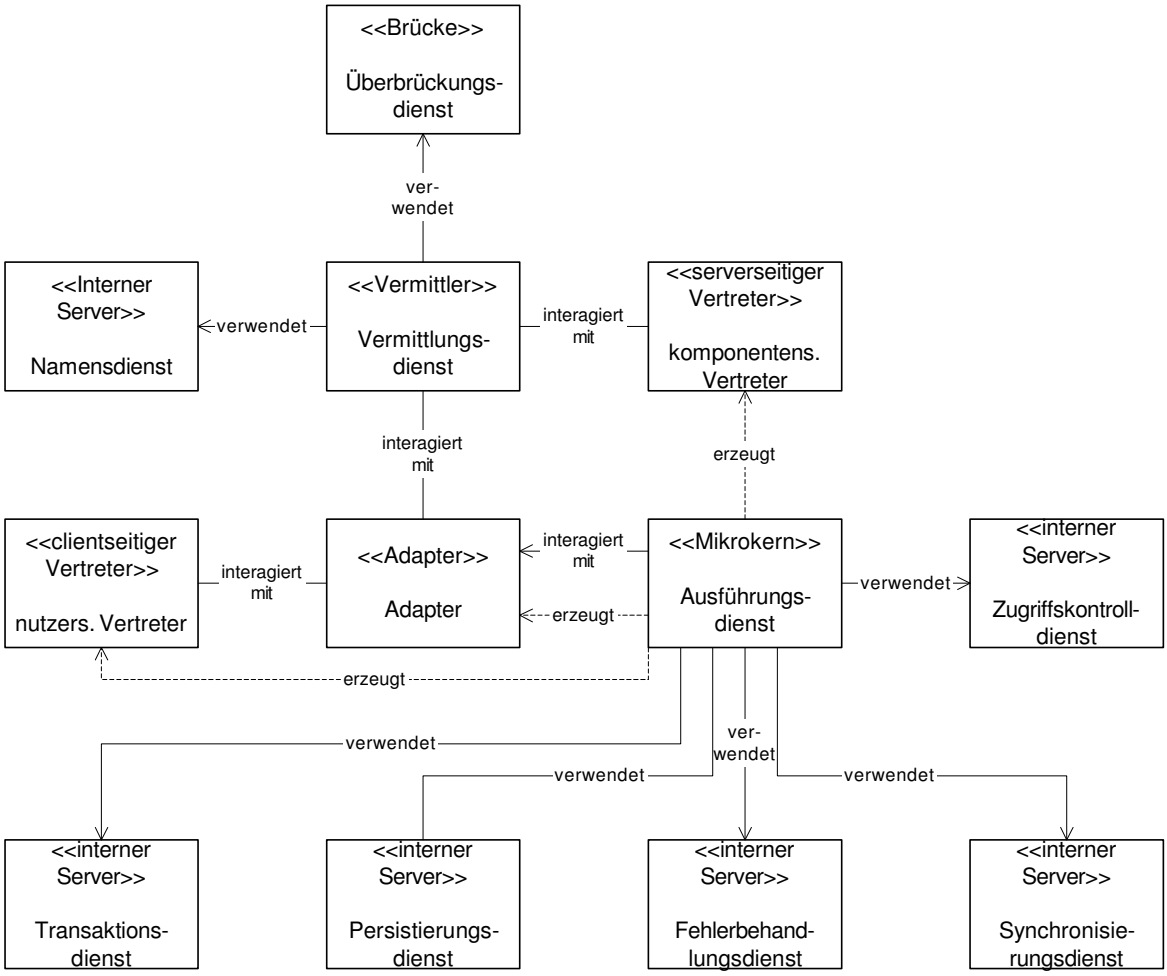


Abbildung 5.18: Endgültige Struktur der Ausführungsplattform

5.3.5 Beziehung zur Modellebene der Außenperspektive

Anhand eines Beziehungs-Metamodells werden nun die Kern-Metamodelle beider Modellebenen integriert. Wie bereits in Abschnitt 5.1.3 erläutert, muss ein auf der Innenperspektive-Modellebene spezifiziertes Anwendungssystem mit einem zugehörigen, auf der Außenperspektive-Modellebene spezifizierten Anwendungssystem verträglich sein, d. h. das auf der Außenperspektive-Modellebene spezifizierte Verhalten realisieren.

5.3.5.1 Beziehungs-Metamodell

Gemäß dem Grundkonzept der Komponentenorientierung weist die Innenperspektive einer Software-Komponente eine objektorientierte Klassenstruktur auf, die den in der Außenperspektive spezifizierten fachlichen Belang realisiert. Die öffentlichen Operatorensignaturen der darin enthaltenen Klassen entsprechen folglich den fachlichen Operatorensignaturen der Außenperspektive dieser Software-Komponente. Ferner realisiert eine in der Innenperspektive spezifizierte objektorientierte Ausführungsplattform die in der Außenperspektive deklarierten nicht-funktionalen Eigenschaften und Eigenschaftswerte sowie die verwaltungsbezogenen Operatorensignaturen. Dies führt zu dem in der folgenden Tabelle enthaltenen Beziehungs-Metamodell.

Metaobjekt Kern-Metamodell der Innenperspektive-Modellebene	Kardinalität 1	Metaobjekt Kern-Metamodell der Außenperspektive-Modellebene	Kardinalität 2
Vorgangsklasse mit Namen, Operatorenimplementierungen und Attributen	1,*	Vorgangs-Software-Komponente	0,1
Vorgangsklasse mit Namen, Operatorenimplementierungen und Attributen	1,*	Unterstützungs-Software-Komponente	0,1
Konzeptuelle Klasse mit Namen, Operatorenimplementierungen und Attributen	1,*	Entitäts-Software-Komponente	1,1
Datenhaltungsklasse mit Namen, Operatorenimplementierungen und Attributen	1,*	Datenhaltungs-Software-Komponente	1,1
Kommunikationsklasse mit Namen, Operatorenimplementierungen und Attributen	1,*	Kommunikations-Software-Komponente	1,1

Metaobjekt Kern-Metamodell der Innenperspektive-Modellebene	Kardinalität 1	Metaobjekt Kern-Metamodell der Außenperspektive-Modellebene	Kardinalität 2
Adapterklasse	0,1	Vorgangs-Software-Komponente	1,1
Adapterklasse	0,1	Unterstützungs-Software-Komponente	1,1
Adapterklasse	0,1	Entitäts-Software-Komponente	1,1
Adapterklasse	0,1	Datenhaltungs-Software-Komponente	1,1
Adapterklasse	0,1	Kommunikations-Software-Komponente	1,1
Serverseitige Vertreterklasse	0,1	Vorgangs-Software-Komponente	1,1
Serverseitige Vertreterklasse	0,1	Unterstützungs-Software-Komponente	1,1
Serverseitige Vertreterklasse	0,1	Entitäts-Software-Komponente	1,1
Serverseitige Vertreterklasse	0,1	Datenhaltungs-Software-Komponente	1,1
Serverseitige Vertreterklasse	0,1	Kommunikations-Software-Komponente	1,1
Generalisierung mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,*	Vorgangs-Software-Komponente	1,1
Generalisierung mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,*	Unterstützungs-Software-Komponente	1,1
Generalisierung mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,*	Entitäts-Software-Komponente	1,1
Generalisierung mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,*	Datenhaltungs-Software-Komponente	1,1

Metaobjekt Kern-Metamodell der Innenperspektive-Modellebene	Kardinalität 1	Metaobjekt Kern-Metamodell der Außenperspektive-Modellebene	Kardinalität 2
Generalisierung mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,*	Kommunikations-Software-Komponente	1,1
Aggregation mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,*	Vorgangs-Software-Komponente	1,1
Aggregation mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,*	Unterstützungs-Software-Komponente	1,1
Aggregation mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,*	Entitäts-Software-Komponente	1,1
Aggregation mit Kardinalitäten (Intra-Komponenten-Beziehung)	0,*	Datenhaltungs-Software-Komponente	1,1
Aggregation mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,*	Kommunikations-Software-Komponente	1,1
Ungerichtete Assoziation mit Kardinalitäten, Namen und Nachricht (Intra-Komponenten-Beziehung)	0,*	Vorgangs-Software-Komponente	1,1
Ungerichtete Assoziation mit Kardinalitäten, Namen und Nachricht (Intra-Komponenten-Beziehung)	0,*	Unterstützungs-Software-Komponente	1,1
Ungerichtete Assoziation mit Kardinalitäten, Namen und Nachricht (Intra-Komponenten-Beziehung)	0,*	Entitäts-Software-Komponente	1,1
Ungerichtete Assoziation mit Kardinalitäten, Namen und Nachricht (Intra-Komponenten-Beziehung)	0,*	Datenhaltungs-Software-Komponente	1,1
Ungerichtete Assoziation mit Kardinalitäten, Namen und Nachricht (Intra-Komponenten-Beziehung)	0,*	Kommunikations-Software-Komponente	1,1

Metaobjekt Kern-Metamodell der Innenperspektive-Modellebene	Kardinalität 1	Metaobjekt Kern-Metamodell der Außenperspektive-Modellebene	Kardinalität 2
Gerichtete Assoziation mit Kardinalitäten, Namen und Nachricht (Intra-Komponenten-Beziehung)	0,*	Vorgangs-Software-Komponente	1,1
Gerichtete Assoziation mit Kardinalitäten, Namen und Nachricht (Intra-Komponenten-Beziehung)	0,*	Unterstützungs-Software-Komponente	1,1
Gerichtete Assoziation mit Kardinalitäten, Namen und Nachricht (Intra-Komponenten-Beziehung)	0,*	Entitäts-Software-Komponente	1,1
Gerichtete Assoziation mit Kardinalitäten, Namen und Nachricht (Intra-Komponenten-Beziehung)	0,*	Datenhaltungs-Software-Komponente	1,1
Gerichtete Assoziation mit Kardinalitäten, Namen und Nachricht (Intra-Komponenten-Beziehung)	0,*	Kommunikations-Software-Komponente	1,1
Generalisierung mit Kardinalitäten und Nachricht (Inter-Komponenten-Beziehung)	0,1	Generalisierung mit Kardinalitäten und Komponentennachricht	1,1
Aggregation mit Kardinalitäten und Nachricht (Inter-Komponenten-Beziehung)	0,1	Aggregation mit Kardinalitäten, Namen und Komponentennachricht	1,1
Ungerichtete Assoziation mit Kardinalitäten, Namen und Nachricht (Inter-Komponenten-Beziehung)	0,1	Ungerichtete Assoziation mit Kardinalitäten, Namen und Komponentennachricht	1,1
Gerichtete Assoziation mit Kardinalitäten, Namen und Nachricht (Inter-Komponenten-Beziehung)	0,1	Gerichtete Assoziation mit Kardinalitäten, Namen und Komponentennachricht	1,1
Operatorsignatur	1,1	Komponentenoperatorsignatur	0,1

Tabelle 5.8: Beziehungs-Metamodell zur Verbindung der Außenperspektive-Modellebene mit der Innenperspektive-Modellebene

5.3.5.2 Beziehungsmuster

Beziehungsmuster stellen im Allgemeinen heuristisches Entwurfswissen bezüglich der Zuordungs- bzw. Transformationsbeziehungen benachbarter Modellebenen bereit. So können Freiheitsgrade bei der Modellierung von Beziehungen zwischen Modellbausteinen der einen Modellebene und Modellbausteinen der anderen Modellebene zielgerichtet reduziert werden [Sinz02a, 1058]. Im vorliegenden Fall existieren potenzielle Freiheitsgrade bei der Zuordnung von Klassen-Teilstrukturen zu Software-Komponenten.

Grundsätzlich werden diese Freiheitsgrade bereits durch die vorgegebenen *realisiert*-Beziehungen zwischen Klassen-Teilstrukturen und Software-Komponenten eingeschränkt. Für die Zuordnungen von Kommunikationsklassen-Teilstrukturen zu Kommunikations-Software-Komponenten und von Datenhaltungsklassen-Teilstrukturen zu Datenhaltungs-Software-Komponenten existieren auch keine darüber hinausgehenden Beschränkungen. Dagegen unterliegen die Zuordnungen von Teilstrukturen konzeptueller Klassen zu Entitäts-Software-Komponenten und von Vorgangsklassen-Teilstrukturen zu Vorgangs-Software-Komponenten bzw. Unterstützungs-Software-Komponenten der zusätzlichen Einschränkung, dass vornehmlich fachliche Belange zu berücksichtigen sind. Da das Software-Architekturmodell ein Teilmodell eines umfassendes Architekturmodells für Informationssysteme darstellt, werden die zu berücksichtigenden fachlichen Belange durch die fachliche Anwendungssystemspezifikation auf der übergeordneten fachlichen Modellebene bestimmt. Folglich müssen Beziehungsmuster zwischen der fachlichen Modellebene und der Außenperspektive-Modellebene sowie zwischen der fachlichen Modellebene und der Innenperspektive-Modellebene des Software-Architekturmodells formuliert werden, die die erforderliche Einschränkung gewährleisten. Diese Beziehungsmuster werden in Abschnitt 5.4.3 vorgestellt.

5.3.6 Basismaschinen komponentenbasierter Anwendungssysteme

Ein auf der Innenperspektive-Modellebene spezifiziertes komponentenbasiertes Anwendungssystem wird auf der Grundlage von geeigneten Basismaschinen implementiert. Für die Implementierung der funktionalen Sicht stehen, wie erwähnt, objektorientierte Programmiersprachen zur Verfügung. Die technische Sicht enthält eine auf der Grundlage von Architektur- und Entwurfsmustern spezifizierte Ausführungsplattform. Für deren Realisierung sind geeignete Basismaschinen auszuwählen.

5.3.6.1 Bestimmung von Anforderungen

Die folgenden Anforderungen resultieren aus der in Abschnitt 5.3.4 spezifizierten Ausführungsplattform, die auf den im Grundkonzept der Komponentenorientierung geforderten Basismaschinendiensten und der Anwendung ausgewählter Architektur- und Entwurfsmustern basiert. Die Ausführungsplattform unterstützt die transparente Verteilung der Software-Komponenten über mehrere Rechnersysteme. Dabei ist sie offen, das heißt portabel und interoperabel und kann die Heterogenität der zugrundeliegenden Basismaschinen bewältigen. Außerdem ist sie für die einfache Integration neuer Basismaschinen erweiterbar und anpassbar.

Entsprechend der Spezifikation der Ausführungsplattform, erfordert die transparente Kommunikation verteilter Software-Komponenten einen Vermittlungsdienst, einen Namensdienst, einen Überbrückungsdienst und für jede Software-Komponente einen nutzerseitigen und einen komponentenseitigen Software-Komponenten-Vertreter. Die Instanzen der Software-Komponenten-Vertreter sowie der Software-Komponenten selbst werden von einem Ausführungsdienst erzeugt, verwaltet und gelöscht. Der Ausführungsdienst sorgt ferner für die transparente Nutzung der unterstützenden Dienste durch die Software-Komponenten-Instanz. Zu den unterstützenden Diensten gehören ein Transaktionsdienst, ein Persistierungsdienst, ein Fehlerbehandlungsdienst, ein Synchronisierungsdienst und ein Zugriffskontrolldienst. Damit der Ausführungsdienst diese unterstützenden Dienste auf die von der Software-Komponenten-Instanz geforderte Weise aufrufen kann, erzeugt er für jede Software-Komponenten-Instanz eine entsprechende Adapterinstanz, die die zur Software-Komponente gehörenden deklarierten nicht-funktionalen Eigenschaften und Eigenschaftswerte in konkrete Dienstanforderungen übersetzt.

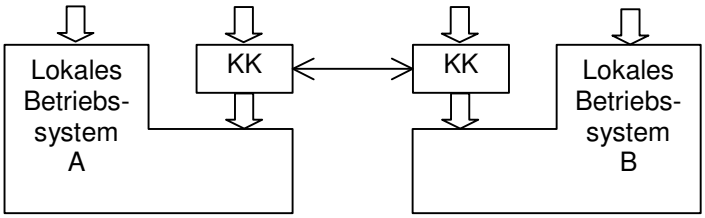
5.3.6.2 Vorstellung geeigneter Basismaschinenkonzepte

Auf der nächstniedrigeren Maschinenebene stehen als **Basismaschinen für verteilte Anwendungssysteme** grundsätzlich netzwerkfähige Betriebssysteme, Netzwerkbetriebssysteme, verteilte Betriebssysteme und middleware-basierende verteilte Systeme zur Verfügung [TaSt02, 22; Beng00, 16 ff; FeSi01, 355 ff]. Die zugrundeliegenden Konzepte der drei erstgenannten Basismaschinen sind in Abbildung 5.19 dargestellt.

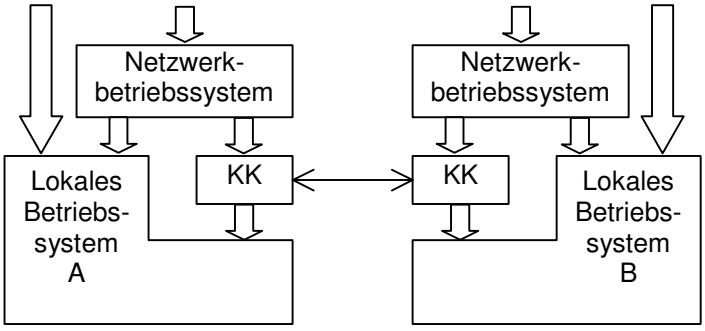
Netzwerkfähige Betriebssysteme sind lokale Betriebssysteme, die über eigene Kommunikationskomponenten (KK) geeignete Kommunikationsfunktionen für Anwendungssysteme bereitstellen. Diese **Kommunikationskomponenten** werden auf der Grundlage der eigenen Betriebssystemfunktionen realisiert [FeSi01, 355]. **Netzwerkbetriebssysteme** bauen auf lokalen Betriebssystemen auf und nutzen deren Funktionen und Kommunikationskomponenten zur Erfüllung von Kommunikations-

funktionen für Anwendungssysteme [TaSt02, 33; FeSi01, 356]. Dagegen ist ein **verteiltes Betriebssystem** ein eigenständiges Betriebssystem, das seine Betriebssystemfunktionen auf mehrere Rechnersysteme verteilt und somit für ein Anwendungssystem als ein einziges virtuelles Betriebssystem erscheint [TaSt02, 25 ff; FeSi01, 356].

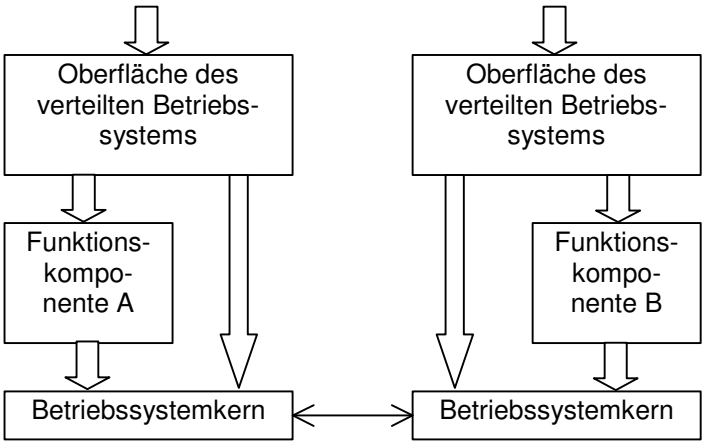
Jede der bisher genannten Basismaschinen hat spezifische Vor- und Nachteile, die beispielsweise in [TaSt02, 41 ff] ausführlich diskutiert werden. Allerdings kann keine dieser Basismaschinen alle im vorangegangenen Abschnitt aufgeführten Anforderungen erfüllen.



a) Netzwerkfähiges Betriebssystem



b) Netzwerkbetriebssystem



c) Verteiltes Betriebssystem

Abbildung 5.19: Basismaschinenkonzepte für verteilte Anwendungssysteme [FeSi01, 355]

Deswegen wird eine weitere Maschinenschicht benötigt, die sich zwischen den Anwendungssystemen und den erwähnten Basismaschinen befindet und die den genannten Anforderungen vollständig entsprechen kann. Diese Maschinenschicht wird allgemein als **Middleware** bezeichnet und unterstützt die Realisierung verteilter Anwendungssysteme durch die Bereitstellung der erforderlichen Dienste [TaSt02, 36; Beng00, 32; FeSi01, 398]. Die Realisierung dieser Dienste übernehmen spezielle Basismaschinen, die anhand der ihnen zugrundeliegenden Konzepte klassifiziert werden. Dieser Abschnitt stellt nur Basismaschinen-Klassen vor, die die im vorangegangenen Abschnitt geforderten Dienste realisieren. Dazu werden die ihnen zugrundeliegenden Konzepte kurz erläutert.

Das erste Konzept ermöglicht die transparente Kommunikation zwischen Software-Komponenten-Instanzen in einem objektintegrierten verteilten Anwendungssystem. Dabei handelt es sich um das Konzept des entfernten Methodenaufrufs (***Remote Method Invocation*, *RMI***). Dieses basiert wiederum auf dem Grundkonzept des entfernten Prozeduraufrufs (***Remote Procedure Call*, *RPC***). Die folgenden Erläuterungen zum Grundkonzept des *RPC* und zum Grundkonzept des *RMI* basieren im Wesentlichen auf den diesbezüglichen Ausführungen in [TaSt02, 68 ff].

Grundkonzept des entfernten Prozeduraufrufs (*RPC*)

Für einen *RPC* muss sich im lokalen Adressraum eines prozedurnutzenden Prozesses ein **nutzerseitiger Vertreter** der entfernten, aufzurufenden Prozedur befinden (vgl. Abbildung 5.20).

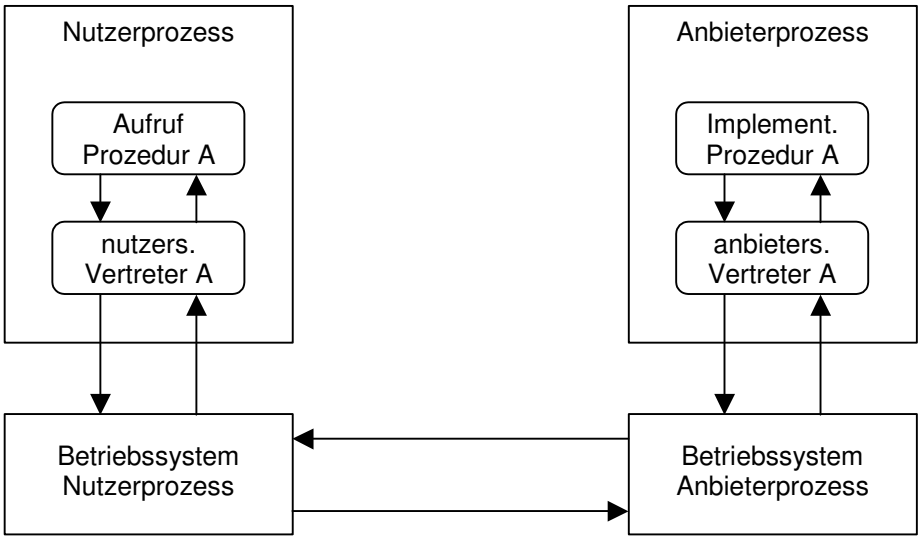


Abbildung 5.20: Grundkonzept *Remote Procedure Call* (*RPC*)

Dieser nutzerseitige Vertreter emuliert die entfernte Prozedur, so dass der Aufruf des nutzerseitigen Vertreters der entfernten Prozedur für den Nutzerprozess einem lokalen Aufruf der Prozedur entspricht. Der nutzerseitige Vertreter verpackt die Parameter des Prozeduraufrufs in eine Nachricht und veranlasst das lokale Betriebssystem, diese Nachricht an das Betriebssystem des prozeduranbietenden Prozesses zu verschicken. Mit dem Verpacken der Parameter, auch ***Marshalling*** oder Externalisieren genannt, werden die Parameter in ein rechnerunabhängiges, standardisiertes Format umgewandelt und somit die Interoperabilität zwischen Nutzerprozess und Anbieterprozess gewährleistet. Bevor das Betriebssystem des Nutzerprozesses die Nachricht an das Betriebssystem des Anbieterprozesses senden kann, muss dieser den Namen der angebotenen Prozedur zusammen mit der Netzwerkadresse des zugrundeliegenden Rechnersystems in einem **Namensverzeichnis** registrieren lassen. Dieses Namensverzeichnis wird von einem unabhängigen Verzeichnisanbieter zur Verfügung gestellt. Daraufhin kann das Betriebssystem des Nutzerprozesses von dem Verzeichnisanbieter anhand des Prozedurnamens die Netzwerkadresse des Rechnersystems, auf dem sich das Betriebssystem des Anbieterprozesses befindet, erfahren und die Nachricht an diese Adresse versenden. Wenn die Nachricht dort angekommen ist, gibt das Betriebssystem des Anbieterprozesses die Nachricht an einen **anbieterseitigen Vertreter** der aufzurufenden Prozedur weiter. Der anbieterseitige Vertreter ist das anbieterseitige Äquivalent des nutzerseitigen Vertreters. Er entpackt die ankommende Nachricht und ruft die entsprechende Prozedur lokal auf. Diese verarbeitet den Aufruf und gibt das Ergebnis an den anbieterseitigen Vertreter zurück. Der anbieterseitige Vertreter verpackt das Ergebnis wiederum in eine Nachricht und weist das lokale Betriebssystem an, die Nachricht an das Betriebssystem des Nutzerprozesses zu senden. Sobald die Nachricht dort angekommen ist, leitet das Betriebssystem des Nutzerprozesses die Nachricht an den nutzerseitigen Vertreter der entfernten Prozedur weiter, der das Ergebnis aus der Nachricht entpackt und es an den Nutzerprozess weitergibt.

Konzept des entfernten Methodenaufrufs (*RMI*)

Das Konzept eines *RMI* ist im Grunde eine Übertragung des *RPC*-Konzepts auf die Interaktion zwischen einem lokalen methodennutzenden Objekt und einem entfernten methodenanbietenden Objekt (vgl. Abbildung 5.21). Demzufolge erfordert ein *RMI* ebenfalls einen nutzerseitigen Vertreter des Anbieterobjekts, einen korrespondierenden anbieterseitigen Vertreter und ein Namensverzeichnis. Beide Vertreter weisen die Schnittstelle des Anbieterobjekts in einer standardisierten **Schnittstellenbeschreibungssprache** (Interface Definition Language, IDL) auf.

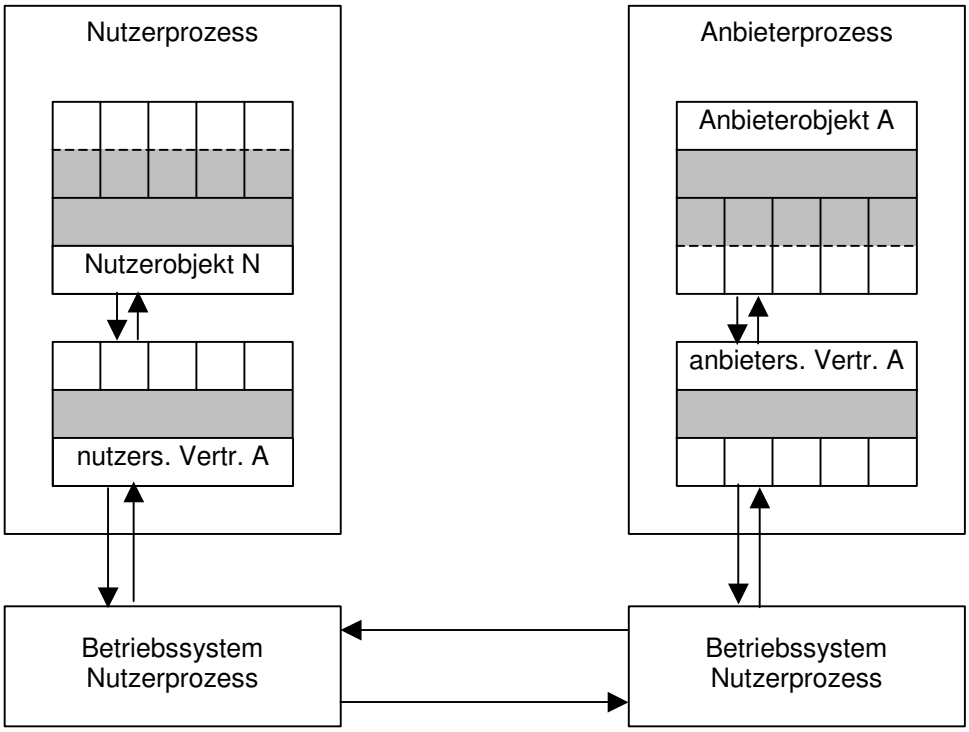


Abbildung 5.21: Grundkonzept *Remote Method Invocation* (*RMI*)

Der nutzerseitige Vertreter ist für das Auffinden des entfernten Objekts mithilfe des Namensverzeichnisses, für das Externalisieren der Parameter und für das Internalisieren des zurückgegebenen Ergebnisses bzw. der Fehlermeldungen zuständig. Der anbieterseitige Vertreter ist für das Internalisieren der Parameter, für den Aufruf der entsprechenden Methode am lokalen Objekt und für das Externalisieren des rückzugebenden Ergebnisses bzw. der Fehlermeldungen verantwortlich. Der Ablauf eines *RMI* entspricht im Wesentlichen dem eines *RPC*. Eine ausführliche Diskussion über die Vor- und Nachteile des *RMI*-Konzepts und die Gemeinsamkeiten und Unterschiede zwischen dem *RPC*- und dem *RMI*-Konzept enthält beispielsweise [TaSt02, 85 ff].

Das beschriebene Konzept des *RMI* kann durch den Einsatz eines **Nachrichtenvermittlers** (*Request Broker*) erweitert werden. Dieser übernimmt für den nutzerseitigen Vertreter das Auffinden des Anbieterobjekts und ist für die Übermittlung der Nachrichten zwischen den beiden Vertretern des Anbieterobjekts verantwortlich. Die Nachrichtenübermittlung kann mithilfe von **Brücken** (*Bridge*) auch über weitere Nachrichtenvermittler hinweg erfolgen. Demnach verbessert der Einsatz eines Nachrichtenvermittlers die Entkopplung eines Nutzerobjekts von einem Anbieterobjekt in einem *RMI*.

Konzept der Nachrichtenvermittler-Architektur CORBA

Die OMG bietet eine standardisierte Spezifikation einer Nachrichtenvermittler-Architektur unter der Bezeichnung **Common Object Request Broker Architecture (CORBA)** an. Der grundsätzliche Aufbau der CORBA ist in Abbildung 5.22 dargestellt.

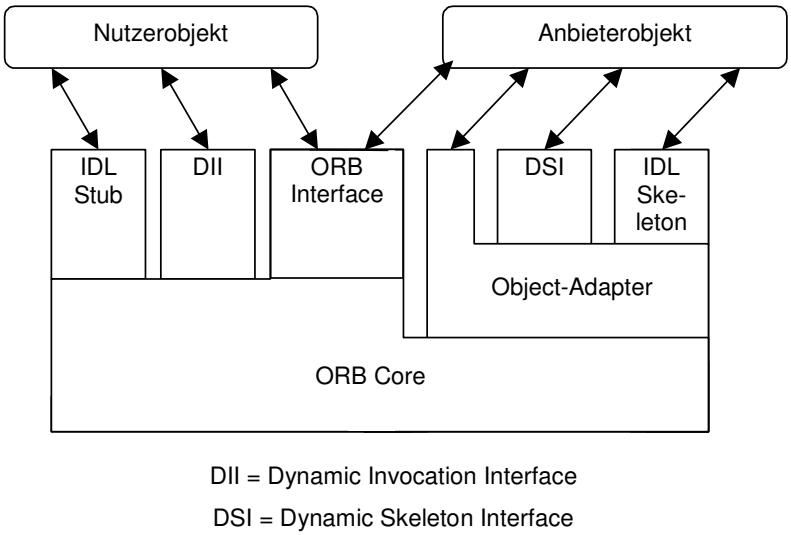


Abbildung 5.22: CORBA-Aufbau [OMG02b, 2-3]

Der **IDL Stub** entspricht dem nutzerseitigen Vertreter, der **IDL Skeleton** dem anbieterseitigen Vertreter und der **ORB Core** dem Nachrichtenvermittler. Das Anbieterobjekt kann auch ohne IDL Skeleton mit dem ORB Core interagieren. Dessen ungeachtet erfolgt die Interaktion zwischen dem Anbieterobjekt und dem ORB Core immer über einen Object Adapter und über das ORB Interface [OMG02b, 2-3]. Über den **Object Adapter** kann unter anderem die Registrierung von Anbieterobjekten, die Erzeugung von Objektreferenzen und das Binden dieser an Anbieterobjekte sowie der Aufruf von Methoden eines Anbieterobjekts über dessen Objektreferenz veranlasst werden [OMG02b, 2-10]. Diese Dienste des ORB Core werden durch einen Object Adapter auf eine Gruppe von ähnlichen Anbieterobjekten angepasst [OMG02b, 2-10]. Demnach existieren in einer CORBA-konformen Nachrichtenvermittler-Architektur üblicherweise mehrere Object Adapter. Das **ORB Interface** bietet sowohl den Nutzer- als auch den Anbieterobjekten zusätzliche Dienste des ORB Core an, die für alle Objekte gleich sind und deswegen nicht über einen Object Adapter angepasst werden müssen [OMG02b, 4-1]. Alternativ zur Verwendung von statischen Vertretern kann ein Aufruf einer Methode an einem Anbieterobjekt auch dynamisch vollzogen werden. Dafür erlaubt das **Dynamic Invocation Interface** dem Nutzerobjekt und das **Dynamic Skeleton Interface** dem ORB Core die dynamische Erzeugung eines Methodenaufrufs an einem Anbieterobjekt [OMG02b, 2-9 f].

Die CORBA-Spezifikation enthält außerdem eine Architektur für die Interoperabilität von unterschiedlichen Object Request Broker. Zu diesem Zweck werden verschiedene **Domänen** eingeführt, die ORB-spezifische Informationen enthalten. Dazu gehören beispielsweise bestimmte Typ-, Sicherheits- und Transaktionsbereiche. ORBs, die sich in der selben Domäne befinden, können direkt miteinander kommunizieren. Dagegen müssen Aufrufe, die zwischen ORBs unterschiedlicher Domänen ausgetauscht werden, über eine Bridge geleitet werden. Diese Bridge sorgt für die notwendige Konvertierung des Aufrufformats.

Neben der Unterstützung der transparenten Kommunikation definiert die CORBA-Spezifikation zusätzliche Dienste für Objekte in einem verteilten Anwendungssystem. Diese Dienste sind selbst wiederum Objekte mit einer IDL-Schnittstelle und einer Implementierung und werden von den Objekten in einem verteilten Anwendungssystem über den ORB Core nach Bedarf aufgerufen. Es können zwei Dienstklassen unterschieden werden. Die so genannten **Object Services** sind grundlegende, domänenunabhängige Dienste für Objekte in einem verteilten Anwendungssystem. Sie sind vergleichbar mit Diensten, die üblicherweise von einem Betriebssystem angeboten werden [TaSt02, 499]. In der Tabelle 5.9 sind die derzeit verfügbaren Object Services zusammengefasst. Dagegen sind die als **Common Facilities** bezeichneten Dienste anwendungsnäher und basieren zum Teil auf den Object Services. Dazu gehören domänenunabhängige Dienste, wie z. B. die Anpassung der Zahlen- und Datenformate an nationale Standards.

Object Service	**Beschreibung**
Collection Service	Dienst zur Gruppierung von Objekten in Form von Listen, Mengen, etc.
Query Service	Dienst zur deklarativen Abfrage von Objektgruppierungen.
Concurrency Service	Dienst, der den nebenläufigen Zugriff auf gemeinsam benutzte Objekte kontrolliert.
Transaction Service	Dienst, der Methodenaufrufe über mehrere Objekte durch flache oder verschachtelte Transaktionen schützt.
Event Service	Dienst für die ereignisorientierte asynchrone Kommunikation.
Notification Service	Dienst für die fortgeschrittene ereignisorientierte asynchrone Kommunikation.
Externalization Service	Dienst zur Externalisierung und Internalisierung von Objekten.

Object Service	Beschreibung
Life Cycle Service	Dienst zur Erzeugung und Zerstörung sowie zum Kopieren und Verschieben von Objekten.
Licensing Service	Dienst zur Lizenzierung von Objekten.
Naming Service	Dienst zur systemweiten Vergabe von Namen für Objekte.
Property Service	Dienst zur Vergabe von Meta-Eigenschaften und zugehörigen Eigenschaftswerten für Objekte.
Trading Service	Dienst zum Veröffentlichen und Auffinden von Objektdiensten.
Persistence Service	Dienst zur persistenten Speicherung von Objekten.
Relationship Service	Dienst, mit dem Beziehungen zwischen Objekten ausgedrückt werden können.
Security Service	Dienst zum Schutz von Übertragungskanälen, zur Autorisieriung von Objektzugriffen und zur Authentifizierung von Objektnutzern.
Time Service	Dienst zur Bereitstellung der aktuellen Zeit innerhalb vorgegebener Genauigkeitsgrenzen

Tabelle 5.9: CORBA Object Services

Konzept der Nachrichten-Warteschlangen (*message queue*)

RPC und *RMI* sind Konzepte, anhand derer eine synchrone Kommunikation zwischen Prozeduren bzw. Objekten in einem verteilten Anwendungssystem durchgeführt werden kann. Zur Realisierung einer asynchronen Kommunikation ist das Konzept der **Nachrichten-Warteschlangen** (*message queue*) aufgrund seiner vielfältigen Einsatzmöglichkeiten am besten geeignet [TaSt02, 115]. Dieses Konzept wird im Grundkonzept der **nachrichtenorientierten Kommunikation** (*message-oriented communication*) in die Kategorie der persistenten asynchronen Kommunikation eingeordnet [TaSt02, 108]. Im Unterschied zu den bisher genannten Konzepten werden Nachrichten-Warteschlangen nicht Prozeduren bzw. Objekten zugeordnet, sondern einem gesamten verteilten Anwendungssystem. Demnach können die oben genannten Konzepte durch das Konzept der Nachrichten-Warteschlangen ergänzt werden. Im vorliegenden Zusammenhang ist insbesondere die Kombination des *RMI*-Konzepts mit dem Konzept der Nachrichten-Warteschlangen von Interesse. Hierzu legt ein nutzerseitiger Vertreter eine Nachricht mit der Adresse einer **Empfänger-Warteschlange** in eine lokale **Sende-Warteschlange** ab. Der zugehörige **Warte-**

schlangen-Verwalter sorgt daraufhin für die Übertragung der Nachricht, gegebenfalls über weitere Sender- und Empfänger-Warteschlangen hinweg, zum Verwalter der lokalen Empfänger-Warteschlange des anbieterseitigen Vertreters. Dieser legt die Nachricht in die lokale Empfänger-Warteschlange ab und benachrichtigt den anbieterseitigen Vertreter über den Empfang der Nachricht. Der anbieterseitige Vertreter holt schließlich die Nachricht aus der lokalen Empfänger-Warteschlange heraus und verarbeitet sie gemäß dem *RMI*-Konzept. Der offensichtliche Vorteil dieses Konzepts ist, dass sowohl der nutzerseitige als auch der anbieterseitige Vertreter durch die Nachrichtenübertragung nicht blockiert werden. Weitergehende Ausführungen zu diesem Konzept sind beispielsweise in [TaSt02, 108 ff] oder [Brit01, 34 ff] enthalten.

Realisierung des entfernten Methoden- bzw. Prozeduraufrufs mit Web Services

Eine mögliche Realisierung des *RMI*- bzw. *RPC*-Mechanismus wird derzeit im Rahmen der Diskussion zum Begriff „**Web Service**“ eingeführt. Dieser Begriff ist bisher noch nicht allgemeingültig bzw. standardisiert definiert [JeMe02, 14; Myer02, 114]. Deswegen existieren inzwischen eine Vielzahl von Interpretationen, die in ihren Abstraktionsgraden stark variieren [JeMe02, 14; Myer02, 114]. Eine ausführliche Zusammenstellung aktueller Begriffsinterpretationen enthält beispielweise [Myer02, 114]. Aufgrund der uneinheitlichen Verwendung des Begriffs „Web Service“ werden in der vorliegenden Arbeit nur die Merkmale vorgestellt, die weitestgehend unstrittig sind und sich auf die in Abschnitt 5.3.6.1 genannten Anforderungen beziehen.

Ein wesentliches Merkmal ist die Verwendung von standardisierten Protokollen und Sprachen des Internetdienstes World Wide Web zur Realisierung des *RPC*- bzw. *RMI*-Mechanismus. So kann als Protokoll für den Transport von Nachrichten im Rahmen eines *RPC* eines der verfügbaren Internetprotokolle, wie z. B. HTTP, SMTP oder FTP verwendet werden [BCF+03, 56; Stal02, 74; BeMW02, 277]. Die Nachrichtenkommunikation wird gemäß dem **Simple Object Access Protocol (SOAP)** durchgeführt. SOAP definiert einen **XML (Extensible Markup Language)**-basierenden *RPC*-Mechanismus, der durch das **World Wide Web Consortium (W3C)** standardisiert wurde [Mitr03, 4 ff; JeMe02, 15; Myer02, 70]. Es wird ergänzt durch eine XML-basierende Schnittstellen-Beschreibungssprache (**Web Service Description Language, WSDL**), deren Struktur sich an der CORBA IDL orientiert und die ebenfalls durch das W3C standardisiert wurde [CGM+03, 5; Stal02, 74; JeMe02, 15; Myer02, 70]. Damit ein Dienst, dessen Schnittstelle mit der WSDL beschrieben wurde, von einem Nutzer verwendet werden kann, muss er von diesem zunächst aufgefunden werden. Dafür steht ihm beispielsweise eine Suchmaschine oder ein globales Verzeichnis, wie das **Universal Service Description, Discovery and Integration (UDDI)** zur Verfügung [BCF+03, 56; JeMe02, 15; Stal02, 74; BeMW02, 278; Myer02, 71]. Anhand des UDDI werden diese Dienste klassifiziert und

über vordefinierte Zugriffsroutinen angeboten [JeMe02, 15; Stal02, 74]. Das dem UDDI zugrundeliegende Konzept orientiert sich an den CORBA Object Services Naming und Property [JeMe02, 15].

Der Zusammenhang zwischen den genannten Protokollen und Sprachen kann anhand eines so genannten **Web Services Stack** beschrieben werden (vgl. Abbildung 5.23) [BCF+03, 15 f; Myer02, 117 f]. Da dafür jedoch noch kein Standard vorliegt, ist der Aufbau des Stacks von Anbieter zu Anbieter verschieden [BCF+03, 15; Myer02, 117 f]. Im Rahmen der Standardisierungsbemühungen des W3C wird derzeit eine so genannte **Web Service Architecture (WSA)** entwickelt, die die genannten web-service-unterstützenden Protokolle und Sprachen in einen Gesamtzusammenhang einordnet und gleichzeitig offen für weitere web-service-unterstützende Technologien ist: „The WSA provides a model and a context for understanding Web Services, and a context for placing Web Services specifications and technologies into relationships with each other and with other technologies outside the WSA“ [BCF+03, 7].

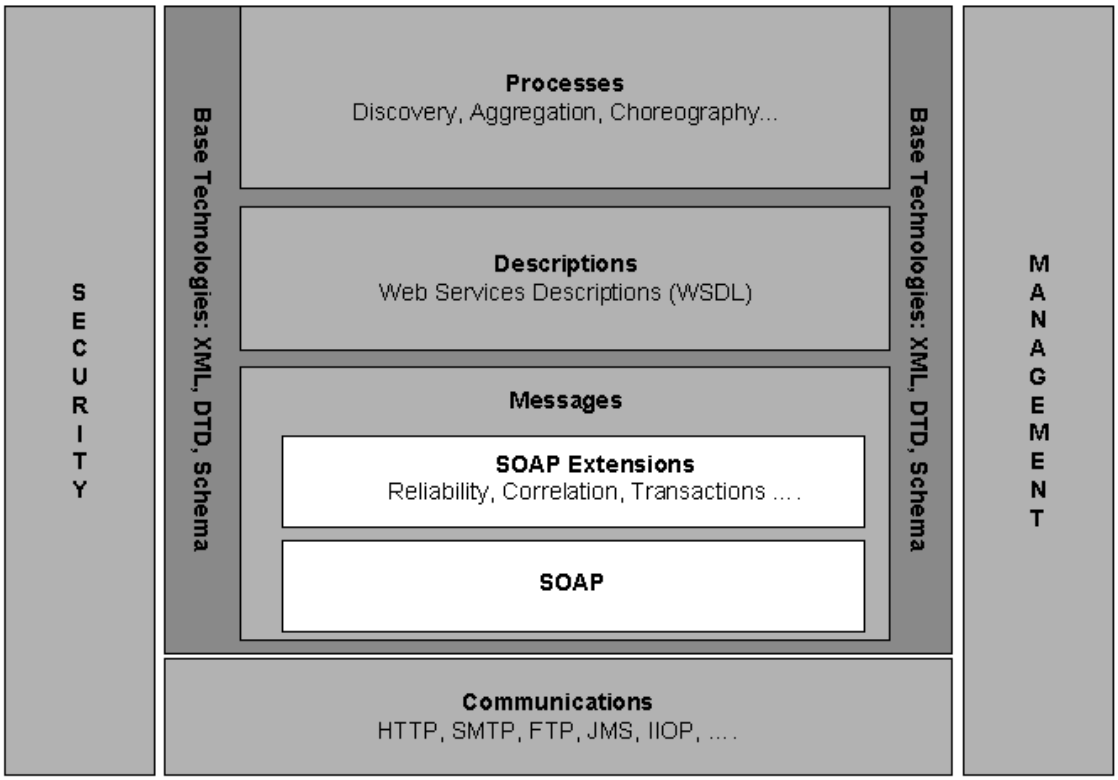


Abbildung 5.23: Web Services Stack [BCF+03, 16]

Wie in Abschnitt 3.5.1.2 bereits angesprochen, werden die genannten Protokolle und Sprachen des Internetdienstes World Wide Web derzeit hauptsächlich zur Herstellung der Interoperabilität zwischen heterogenen Kommunikationssystemen, wie z. B. unterschiedlich realisierten Nachrichtenvermittler-Architekturen verwendet. Die bisherigen Anstrengungen, Kommunikationssysteme unterschiedlicher Anbieter zu integ-

rieren, führten üblicherweise zu weiteren proprietären Integrationstechnologien [Stal02, 73]. Dem gegenüber sind die genannten Protokolle und Sprachen des Internetdienstes World Wide Web standardisiert und auch allgemein akzeptiert, nicht zuletzt weil sie alle auf einer einzigen standardisierten Sprache des Internetdienstes WWW basieren, der XML. Die XML ist aufgrund ihrer Standardisierung, ihrer Flexibilität und ihrer einfachen Struktur als gemeinsame Sprache für den Austausch von Informationen zwischen Unternehmen bereits weit verbreitet. Deswegen bieten sich darauf basierende Protokolle und Sprachen zur Realisierung eines unternehmensübergreifenden Kommunikationssystems an [Stal02, 75]. Da jedoch der Internetdienst WWW mit den zugehörigen Protokollen und Sprachen ursprünglich für die Mensch-Computer-Interaktion, und nicht für den sicheren und leistungsfähigen Datenaustausch zwischen unterschiedlichen Rechnersystemen konzipiert wurde, bestehen gegenwärtig Sicherheits- und Qualitätsprobleme bei der Verwendung der genannten Protokolle und Sprachen zur Integration betrieblicher Anwendungssysteme [NoMe02, 28; BeMW02, 278].

Konzept eines verteilten Transaktionssystems

Eine **verteilte Transaktionsverarbeitung** (*distributed transaction processing*) kann grundsätzlich durch ein verteiltes Datenbanksystem oder durch ein verteiltes Transaktionssystem erreicht werden. Da in der vorliegenden Arbeit die Software-Architektur komponentenbasierter Anwendungssysteme im Vordergrund steht, wird im Folgenden das Konzept eines **verteilten Transaktionssystems** kurz vorgestellt. Für Erläuterungen zum Konzept eines verteilten Datenbanksystems wird auf die diesbezügliche Literatur, wie z. B. [Dada96] oder [OeVa99] verwiesen.

Eine **Transaktion** wird von [GrRe93] als „a collection of operations on the physical and abstract application state“ definiert [GrRe93, 6]. Sie besitzt spezielle Eigenschaften, die häufig im Akronym **ACID (Atomicity, Consistency, Isolation, Durability)** zusammengefasst werden [GrRe93, 6]. Demnach ist eine Transaktion einerseits atomar, d. h. Zustandsänderungen werden vollständig oder gar nicht ausgeführt. Ferner führt eine Zustandsänderung in Form einer Transaktion zu einem konsistenten Folgezustand, der darüber hinaus dauerhaft, d. h. resistent gegen Fehlschläge ist. Schließlich wird eine Transaktion scheinbar isoliert von anderen Transaktionen durchgeführt auch wenn die Transaktionen tatsächlich nebenläufig sind [GrRe93, 6].

Ein **Transaktionssystem** unterstützt die Verwaltung von Transaktionen in einem Anwendungssystem [GrRe93, 5]. Dafür nehmen das Anwendungssystem und die vom Anwendungssystem verwendeten Ressourcen, wie z. B. Datenbanksysteme, die Dienste eines so genannten **Transaction-Processing-Monitor (TP-Monitor)** in Anspruch (vgl. Abbildung 5.24) [GrRe93, 5]. Dieser TP-Monitor sorgt unter anderem für die Gewährleistung der genannten ACID-Eigenschaften einer Transaktion [GrRe93,

5]. Dafür bietet er dem Anwendungssystem als zugrundeliegenden Kommunikationsmechanismus so genannte ***Transactional Remote Procedure Calls* (*TRPC*)** an [GrRe93, 13 f]. Diese entsprechen im Wesentlichen normalen *RPC*s, die jedoch um zusätzliche Merkmale zur Gewährleistung der ACID-Eigenschaften einer Transaktion erweitert wurden [GrRe93, 295 f]. Ausführliche Erläuterungen zu den Merkmalen eines *TRPC* sind in [GrRe93, 295 ff] nachzulesen. Der Aufbau und die speziellen Aufgaben eines TP-Monitors sind ebenfalls in [GrRe93] beschrieben. In einem verteilten Transaktionssystem müssen mehrere TP-Monitore zusammenarbeiten, um die ACID-Eigenschaften einer verteilten Transaktion sicherstellen zu können. Zu diesem Zweck verwenden sie ein spezielles Kommunikationsprotokoll, das verteilte **Zwei-Phasen-Freigabe-Protokoll (*Two-Phase Commit Protocol*, *2PC*)** [GrRe93, 562]. Nähere Ausführungen zu diesem Protokoll sind ebenfalls in [GrRe93, 562 ff] enthalten.

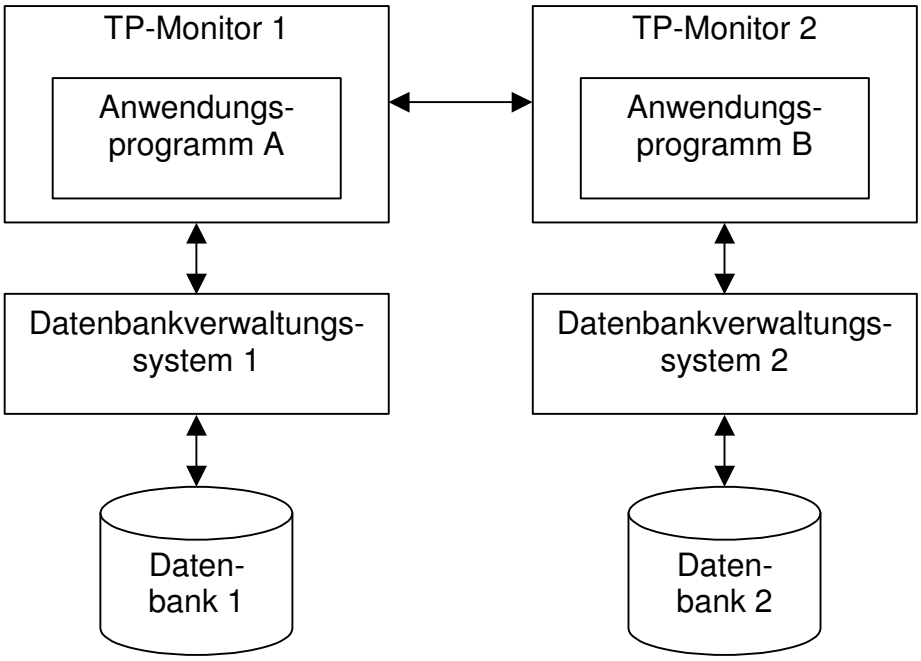


Abbildung 5.24: Transaktionsmonitore

Konzept eines Objekt-Transaktions-Dienstes

Da TP-Monitore nur eine spezielle Form des *RPC*-Mechanismus als Kommunikationsmechanismus anbieten, können sie nicht unmittelbar in einem verteilten objektorientierten oder komponentenbasierten Anwendungssystem eingesetzt werden. Dafür müssten sie Objekten oder Software-Komponenten-Instanzen des verteilten Anwendungssystems einen entsprechenden *RMI*-Mechanismus zur Verfügung stellen. Ferner müssten die Dienste eines TP-Monitors selbst in objektorientierter Form vorliegen. Wie erwähnt bieten CORBA-konforme Vermittler-Architekturen neben der transparenten Kommunikation über *RMI* und *Request Broker* auch zusätzliche objektorientierte Dienste für Objekte bzw. Software-Komponenten-Instanzen an. Darunter befindet sich auch ein Dienst, über den Transaktionen mit verteilten Objekten

bzw. Software-Komponenten-Instanzen ausgeführt werden können. Das zugrundeliegende Konzept dieses als **Object Transaction Service (OTS)** bezeichneten Dienstes wurde gemeinsam mit mehreren Herstellern von TP-Monitoren entwickelt, die ihre Kenntnisse und langjährigen Erfahrungen auf diesem Gebiet einbrachten [OrHE96, 11; Mons01, 16]. Dieser Dienst definiert IDL-Schnittstellen, über die sich mehrere verteilte Objekte in gegebenenfalls mehreren ORBs an einer flachen oder verschachtelten Transaktion beteiligen können [OrHE96, 123]. Darüber hinaus ist es möglich, dass über den OTS auch Anwendungssysteme an einer Transaktion teilnehmen, die weder CORBA-basierend noch objekt- oder komponentenorientiert sind, wie z. B. prozedurale Anwendungssysteme auf der Basis eines TP-Monitors [ZiBe00, 102; OrHE96, 127]. Demnach lassen sich mit OTS auch heterogene Anwendungssysteme über Transaktionen integrieren [ZiBe00, 102].

Aufgrund der genannten Vorteile von CORBA-konformen Vermittler-Architekturen mit Object Transaction Service gegenüber konventionellen TP-Monitoren für die Transaktionsverarbeitung in verteilten objektorientierten oder komponentenbasierten Anwendungssystemen wird vielfach prognostiziert, dass CORBA-konforme Vermittler-Architekturen zukünftig die Rolle von TP-Monitoren übernehmen werden [OrHE96, 11; ZiBe00, 102; Mons01, 16].

Konzept eines Datenbanksystems

Für die Persistierung von Zuständen der Software-Komponenten-Instanzen eines verteilten komponentenbasierten Anwendungssystems stehen **Datenbankverwaltungssysteme** zur Verfügung, über die das Anwendungssystem auf eine oder mehrere **Datenbanken** zugreifen kann. Ein Datenbankverwaltungssystem und die von ihm verwalteten Datenbanken bilden zusammen ein **Datenbanksystem** [FeSi01, 361; ZiBe00, 127]. Die genannten Zugriffe müssen transaktionsgeschützt erfolgen, damit die Konsistenz der Datenbanken gewahrt bleibt. Deswegen wird die Persistierung von einem TP-Monitor oder, im vorliegenden Fall, von einem Transaktionsdienst einer Nachrichtenvermittler-Architektur kontrolliert.

Der Zugriff auf die Datenbanken erfolgt meistens über eine deskriptive Sprache, die von den Datenbankverwaltungssystemen bereitgestellt wird. Für relationale und objektrelationale Datenbanken hat sich die von der ANSI (American National Standards Institute) standardisierte Definitions- und Manipulationssprache **SQL (Structured Query Language)** durchgesetzt. Für objektorientierte Datenbanken steht ein von der ODMG (Object Data Management Group) standardisiertes Sprachpaket zur Verfügung, das jedoch bisher noch keine allgemeine Akzeptanz gefunden hat [ZiBe00, 137].

Konzept eines Datenbank-Gateways

Trotz dieser Standards unterscheiden sich die von den Datenbankverwaltungssystemen angebotenen Sprachen geringfügig. Damit dieser Unterschied für Anwendungssysteme transparent bleibt, wurden so genannte **Datenbank-Gateways** entwickelt [Dada96, 284; FeSi01, 398]. Diese bieten einem Anwendungssystem eine einheitliche Schnittstelle an, über die es auf gegebenenfalls mehrere heterogene, verteilte Datenbankverwaltungssysteme und die von ihnen verwalteten Datenbanken zugreifen kann (vgl. Abbildung 5.25) [Dada96, 284; FeSi01, 398 f]. Die Heterogenität und die Verteilung der Datenbanksystem bleiben somit vor dem Anwendungssystem verborgen.

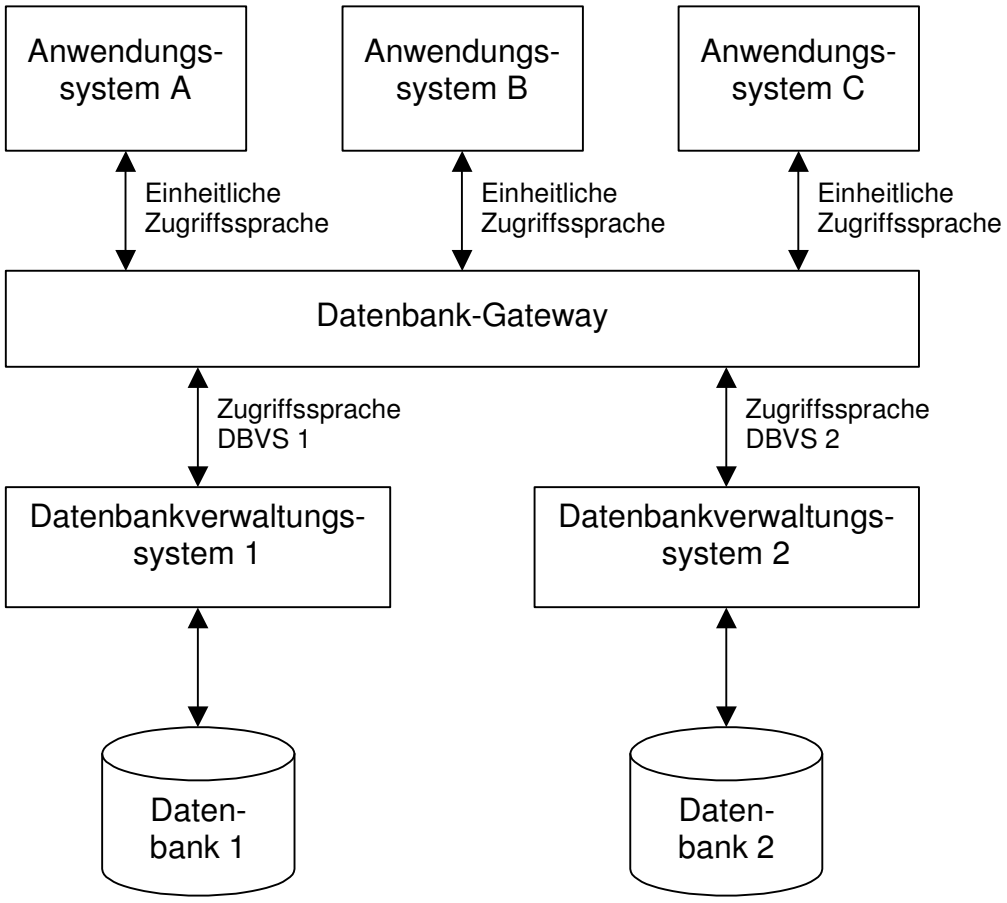


Abbildung 5.25: Datenbank-Gateway

Konzept einer Persistierungsschicht

Ein weiteres Problem stellt die Abbildung von Objekten oder Software-Komponenten-Instanzen in relationale Datenbanken dar. Die objektorientierte Verarbeitung in einem Anwendungssystem unterscheidet sich von der deskriptiven mengenorientierten Verarbeitung in einem relationalen Datenbanksystem. Dieser als ***Impedance Mismatch*** bezeichnete Unterschied kann von den erläuterten Datenbank-Gateways nicht direkt überbrückt werden [ZiBe00, 169]. Demzufolge sorgt eine so genannte **Persistie-**

rungsschicht für die Abbildung von Objekt- oder Software-Komponenten-Instanz-Zuständen auf Tabellenzeilen in einer relationalen Datenbank (vgl. Abbildung 5.26) [ZiBe00, 253 f]. Auf diese Weise werden die funktionalen Belange des Anwendungssystems von den technischen Belangen des Datenbanksystems getrennt. Technische Details des Datenbankzugriffs, wie z. B. Such- und Abbildungsalgorithmen oder leistungsverbessernde Maßnahmen, können anhand einer Persistierungsschicht vor dem Anwendungssystem verborgen werden [ZiBe00, 254]. In CORBA-konformen Vermittler-Architekturen wird eine solche Persistierungsschicht in Form des **Persistence Service** angeboten [OrHE96, 139 ff].

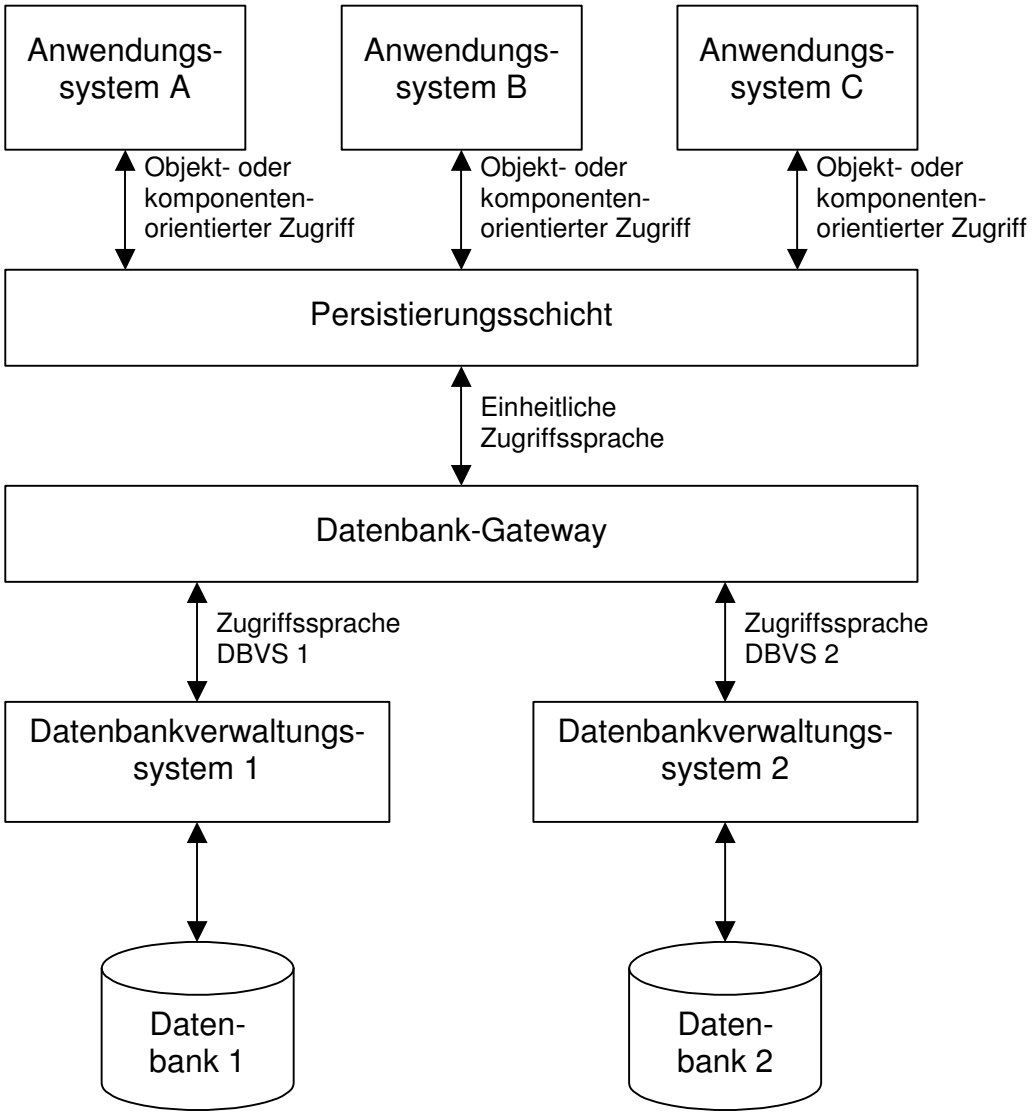


Abbildung 5.26: Persistierungsschicht

Gegenüberstellung der Anforderungen und der vorgestellten Konzepte

Zusammenfassend lässt sich feststellen, dass die in Abschnitt 5.3.6.1 genannten Anforderungen fast vollständig durch den Einsatz einer CORBA-konformen Nachrichtenvermittler-Architektur und eines Datenbank-Gateways erfüllt werden können.

Eine resultierende Gegenüberstellung der Anforderungen mit den beschriebenen Merkmalen dieser beiden Konzepte enthält Tabelle 5.10.

Anforderung	**Konzeptmerkmal**
Vermittlungsdienst	ORB-Core
Namensdienst	Naming Service
Überbrückungsdienst	Bridge
nutzerseitiger Software-Komponenten-Vertreter	IDL-Stub
komponentenseitiger Software-Komponenten-Vertreter	IDL-Skeleton
Ausführungsdienst	-
Erzeugung, Verwaltung und Zerstörung von Software-Komponenten-Instanzen und ihrer Vertreter	Life Cycle Service und Object Adapter
Transaktionsdienst	Transaction Service
Persistierungsdienst	Persistence Service und Datenbank-Gateways
Fehlerbehandlungsdienst	Transactions Service
Synchronisierungsdienst	Concurrency Service
Zugriffskontrolldienst	Security Service
Adapterinstanz	Object Adapter

Tabelle 5.10: Gegenüberstellung der Anforderungen an Basismaschinenkonzepten mit den Merkmalen einer CORBA-konformen Nachrichtenvermittler-Architektur und eines Datenbank-Gateways

Wie aus dieser Tabelle zu entnehmen ist, kann mit den genannten Konzepten die Forderung nach einem Ausführungsdienst nicht erfüllt werden. Eine CORBA-konforme Nachrichtenvermittler-Architektur und ein Datenbank-Gateway liegen meistens in Form von Klassenbibliotheken vor, die in ein verteiltes Anwendungssystem geeignet eingebunden werden müssen. Die Objekte oder Software-Komponenten-Instanzen dieses Anwendungssystems müssen für die geeignete Nutzung der zur Verfügung gestellten Basismaschinendienste selbst sorgen [Mons01, 16]. Eine transparente Erfüllung der deklarierten nicht-funktionalen Eigenschaften und Eigenschaftswerte kann nicht gewährleistet werden. Somit vermischen sich im Code eines

solchen verteilten Anwendungssystem die funktionalen Belange mit den technischen Belangen.

Konzept einer Komponenteninfrastruktur

Diesem Problem wird mit einer neueren Klasse von Basismaschinen auf der Middleware-Schicht begegnet. Gegenwärtig existiert noch keine einheitliche Bezeichnung für diese Klasse. Verbreitete Bezeichnungen sind ***Komponenten-Transaktionsmonitore***, ***Transactional Component Middleware***, ***Object Transaction Monitor*** oder ***Komponenteninfrastruktur*** [Brit01, 66; Mons01, 5; VöSW03, 37]. Solche Basismaschinen stellen eine Infrastruktur für Software-Komponenten zur Verfügung, die für die transparente Erfüllung der geforderten nicht-funktionalen Belange sorgt. Zu diesem Zweck besitzen sie einen so genannten ***Container***, der konzeptuell dem geforderten Ausführungsdienst entspricht [Brit01, 67]. Anhand des *Containers* werden CORBA-konforme Nachrichtenvermittler-Architekturen und Datenbank-Gateways, aber auch andere Middleware-Basismaschinen, wie z. B. TP-Monitore zu einer einheitlichen Infrastruktur integriert und bedarfsorientiert genutzt [Mons01, 16; Brit01, 67]. Eine solche Basismaschine ist auf ein bestimmtes Komponentenkonzept abgestimmt [Mons01, 18; Brit01, 67; VöSW03, 42]. Demzufolge existieren derzeit Infrastrukturen für java-basierende Software-Komponenten (Java 2 Enterprise Edition (J2EE)-konforme Server) und für Microsoft-COM (Component Object Model)-basierende Software-Komponenten (Microsoft COM+-Server).

5.4 Beziehung zwischen Software-Architekturmodell und fachlicher Modellebene

Wie bereits erwähnt, ist das Software-Architekturmodell als Teil eines umfassenden Architekturkonzepts für Informationssysteme konzipiert. Demnach muss eine methodische Durchgängigkeit beim Übergang von der fachlichen Modellebene des gewählten Architekturkonzepts zur softwaretechnischen Modellebene, die in Form des Software-Architekturmodells vorliegt, hergestellt werden. Dies erfordert, wie ebenfalls schon erwähnt, eine fachliche Anwendungssystemspezifikation in objektorientierter und objektintegrierter Form. Damit wird die Lücke, die aufgrund der verschiedenen Perspektiven zwischen diesen beiden Modellebenen besteht, bestmöglich überwunden.

Folglich wird für die weiteren Ausführungen in diesem Abschnitt vorausgesetzt, dass die fachliche Anwendungssystemspezifikation zumindest aus Vorgangsklassen und konzeptuellen Klassen besteht, die, entsprechend den Ausführungen in Abschnitt 3.2.3.2, miteinander in Beziehung stehen und gemeinsam die zugrundeliegende Gesamtaufgabe des Anwendungssystems realisieren.

5.4.1 Exemplarisches Metamodell der fachlichen Modellebene

Ein exemplarisches Metamodell, mit dem objektorientierte und objektintegrierte fachliche Anwendungssystemspezifikationen erstellt werden können, enthält Abbildung 5.27. Dieses Metamodell bildet auch die Grundlage für das Beziehungs-Metamodell, das im folgenden Abschnitt vorgestellt wird.

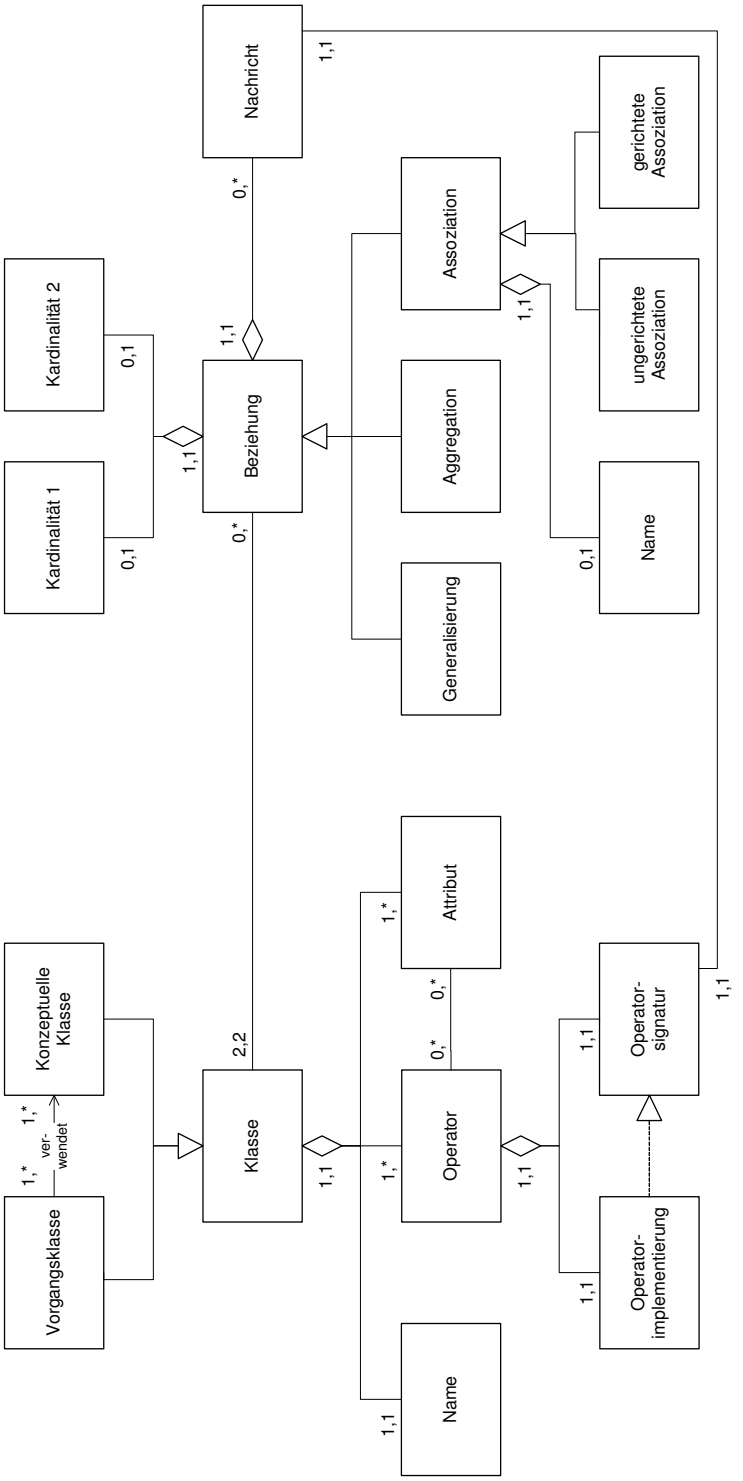


Abbildung 5.27: Exemplarisches Metamodell für objektorientierte und objektintegrierte fachliche Anwendungssystemspezifikationen

5.4.2 Beziehungs-Metamodelle

Sowohl das exemplarische Metamodell der fachlichen Modellebene als auch die Metamodelle der beiden Modellebenen des Software-Architekturmodells enthalten Metaobjekte für die Abbildung von Vorgängen und Metaobjekte für die Abbildung von Aufgabenobjekten.

Grundsätzliches Vorgehen bei der Abgrenzung von Software-Komponenten aus einer fachlichen Anwendungssystemspezifikation

Zunächst werden Teilstrukturen von Vorgangsklassen, die gemeinsam einen fachlichen Belang abbilden, in Vorgangs-Software-Komponenten zusammengefasst. Falls sich die Teilstrukturen unterschiedlicher Vorgangs-Software-Komponenten überlappen, werden die redundanten Teilstrukturen anhand des Entwurfsmusters *Unterstützung* in Unterstützungs-Software-Komponenten ausgelagert.

Die Zusammenfassung in Vorgangs- und Unterstützungs-Software-Komponenten begründet daraufhin die Abgrenzung von Teilstrukturen konzeptueller Klassen, die in Entitäts-Software-Komponenten gekapselt werden. Bei der Abgrenzung von Klassenteilstrukturen zu Software-Komponenten muss grundsätzlich darauf geachtet werden, dass Generalisierungs- und Aggregationsbeziehungen, die zwischen den abzugrenzenden Klassen bestehen und nicht fachlich begründet sind, in den zu bildenden Software-Komponenten gekapselt werden.

Schließlich folgt die Zuordnung der erstellten Entitäts-Software-Komponenten zu den entsprechenden Vorgangs- und Unterstützungs-Software-Komponenten. Das detaillierte Vorgehen bei der Bildung von Vorgangs- und Entitäts-Software-Komponenten auf der Grundlage von Vorgangs- und konzeptuellen Klassen wird in den Beziehungsmustern im folgenden Abschnitt beschrieben.

Die abgegrenzten Teilstrukturen von Vorgangsklassen und konzeptuellen Klassen stellen daraufhin die Implementierungen der gebildeten Vorgangs- bzw. Unterstützungs-Software-Komponenten und Entitäts-Software-Komponenten dar. Die Attribute der einer Software-Komponente zugeordneten Klassen sind zugleich die Attribute der Software-Komponente selbst. Analog dazu stimmen die öffentlichen Operatoren der einer Software-Komponente zugeordneten Klassen mit den Operatoren der Software-Komponente selbst überein. Demzufolge bilden die abgegrenzten Klassen-Teilstrukturen der fachlichen Modellebene die Innenperspektiven der entsprechenden Vorgangs-, Unterstützungs- oder Entitäts-Software-Komponenten auf der Innenperspektive-Modellebene der Software-Architektur.

Die Kommunikations-Software-Komponenten und die Datenhaltungs-Software-Komponenten, die in Abschnitt 5.2.2.2 vorgestellt wurden, haben keine Klassen-Entsprechungen auf der fachlichen Modellebene, da sie sich auf softwaretechnische

Belange beziehen und somit erst im Software-Architekturmodell modelliert werden. Wie bereits im Rahmen des *VE*-Musters in Abschnitt 5.2.4 erläutert, werden sie den Vorgangs-Software-Komponenten und den Entitäts-Software-Komponenten zugeordnet.

Beziehungs-Metamodelle zur Verknüpfung der fachlichen Modellebene mit den Modellebenen des Software-Architekturmodells

Das folgende Beziehungs-Metamodell stellt zunächst die Verknüpfungen der Meta-Objekte der fachlichen Modellebene mit den Meta-Objekten der Außenperspektive-Modellebene des Software-Architekturmodells dar.

Metaobjekt d. Metamodells der fachlichen Modellebene	Kardinalität 1	Metaobjekt des Kern-Metamodells der Außenperspektive-Modellebene	Kardinalität 2
Vorgangsklasse mit Namen, Operatorenimplementierungen und Attributen	1,*	Vorgangs-Software-Komponente	0,1
Vorgangsklasse mit Namen, Operatorenimplementierungen und Attributen	1,*	Unterstützungs-Software-Komponente	0,1
Konzeptuelle Klasse mit Namen, Operatorenimplementierungen und Attributen	1,*	Entitäts-Software-Komponente	1,1
Generalisierung mit Kardinalitäten und Nachricht	0,*	Vorgangs-Software-Komponente	0,1
Generalisierung mit Kardinalitäten und Nachricht	0,*	Unterstützungs-Software-Komponente	0,1
Generalisierung mit Kardinalitäten und Nachricht	0,*	Entitäts-Software-Komponente	0,1
Generalisierung mit Kardinalitäten und Nachricht	0,1	Generalisierung mit Kardinalitäten und Komponentennachricht	0,1
Aggregation mit Kardinalitäten und Nachricht	0,*	Vorgangs-Software-Komponente	0,1
Aggregation mit Kardinalitäten und Nachricht	0,*	Unterstützungs-Software-Komponente	0,1
Aggregation mit Kardinalitäten und Nachricht	0,*	Entitäts-Software-Komponente	0,1

Metaobjekt d. Metamodells der fachlichen Modellebene	Kardinalität 1	Metaobjekt des Kern-Metamodells der Außenperspektive-Modellebene	Kardinalität 2
Aggregation mit Kardinalitäten und Nachricht	0,1	Aggregation mit Kardinalitäten, Namen und Komponentennachricht	0,1
Ungerichtete Assoziation mit Kardinalitäten und Nachricht	0,*	Vorgangs-Software-Komponente	0,1
Ungerichtete Assoziation mit Kardinalitäten und Nachricht	0,*	Unterstützungs-Software-Komponente	0,1
Ungerichtete Assoziation mit Kardinalitäten und Nachricht	0,*	Entitäts-Software-Komponente	0,1
Ungerichtete Assoziation mit Kardinalitäten und Nachricht	0,1	Ungerichtete Assoziation mit Kardinalitäten und Komponentennachricht	0,1
Gerichtete Assoziation mit Kardinalitäten und Nachricht	0,*	Vorgangs-Software-Komponente	0,1
Gerichtete Assoziation mit Kardinalitäten und Nachricht	0,*	Unterstützungs-Software-Komponente	0,1
Gerichtete Assoziation mit Kardinalitäten und Nachricht	0,*	Entitäts-Software-Komponente	0,1
Gerichtete Assoziation mit Kardinalitäten und Nachricht	0,1	Gerichtete Assoziation mit Kardinalitäten und Komponentennachricht	0,1
Operatorsignatur	1,1	Komponentenoperatorsignatur	0,1

Tabelle 5.11: Beziehungs-Metamodell zur Vebindung der fachlichen Modellebene mit der Außenperspektive- Modellebene

Im folgenden Beziehungs-Metamodell werden die Metaobjekte der fachlichen Modellebene mit den Meta-Objekten der Innenperspektive-Modellebene des Software-Architekturmodells verknüpft. Da die abgegrenzten Klassen-Teilstrukturen der fachlichen Modellebene, wie erwähnt, zugleich die Innenperspektiven der entsprechenden Vorgangs-, Unterstützungs- oder Entitäts-Software-Komponenten auf der Innenperspektive-Modellebene der Software-Architektur darstellen, entsprechen sich die Metaobjekte der beiden Modellebene weitestgehend. Auf der Innenperspektive-Modellebene sind zusätzlich Metaobjekte für Kommunikations-, Datenhaltungs- und Dienstklassen vorhanden, die auf der fachlichen Modellebene aus den genannten Gründen nicht zur Verfügung stehen.

Metaobjekt d. Metamodells der fachlichen Modellebene	Kardinalität 1	Metaobjekt d. Kern-Metamodells der Innenperspektive-Modellebene	Kardinalität 2
Vorgangsklasse mit Namen, Operatorenimplementierungen, Operatorensignaturen und Attributen	1,1	Vorgangsklasse mit Namen, Operatorenimplementierungen, Operatorensignaturen und Attributen	1,1
Konzeptuelle Klasse mit Namen, Operatorenimplementierungen, Operatorensignaturen und Attributen	1,1	Konzeptuelle Klasse mit Namen, Operatorenimplementierungen, Operatorensignaturen und Attributen	1,1
Generalisierung mit Kardinalitäten und Nachricht	1,1	Generalisierung mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,1
Generalisierung mit Kardinalitäten und Nachricht	1,1	Generalisierung mit Kardinalitäten und Nachricht (Inter-Komponenten-Beziehung)	0,1
Aggregation mit Kardinalitäten und Nachricht	1,1	Aggregation mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,1
Aggregation mit Kardinalitäten und Nachricht	1,1	Aggregation mit Kardinalitäten und Nachricht (Inter-Komponenten-Beziehung)	0,1
Ungerichtete Assoziation mit Kardinalitäten und Nachricht	1,1	Ungerichtete Assoziation mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,1
Ungerichtete Assoziation mit Kardinalitäten und Nachricht	1,1	Ungerichtete Assoziation mit Kardinalitäten und Nachricht (Inter-Komponenten-Beziehung)	0,1
Gerichtete Assoziation mit Kardinalitäten und Nachricht	1,1	Gerichtete Assoziation mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,1
Gerichtete Assoziation mit Kardinalitäten und Nachricht	1,1	Gerichtete Assoziation mit Kardinalitäten und Nachricht (Inter-Komponenten-Beziehung)	0,1

Tabelle 5.12: Beziehungs-Metamodell zur Vebindung der fachlichen Modellebene mit der Innenperspektive- Modellebene

Das heuristische Entwurfswissen für die Verknüpfungs- und Transformationsbeziehungen zwischen der fachlichen Modellebene und den beiden Modellebenen des Software-Architekturmodells wird in Form von Beziehungsmustern beschrieben, die im folgenden Abschnitt dargestellt werden.

5.4.3 Beziehungsmuster

Das erste Beziehungsmuster beschreibt das Vorgehen bei der Abgrenzung von Vorgangsklassen der fachlichen Modellebene zu Vorgangs-Software-Komponenten des Software-Architekturmodells.

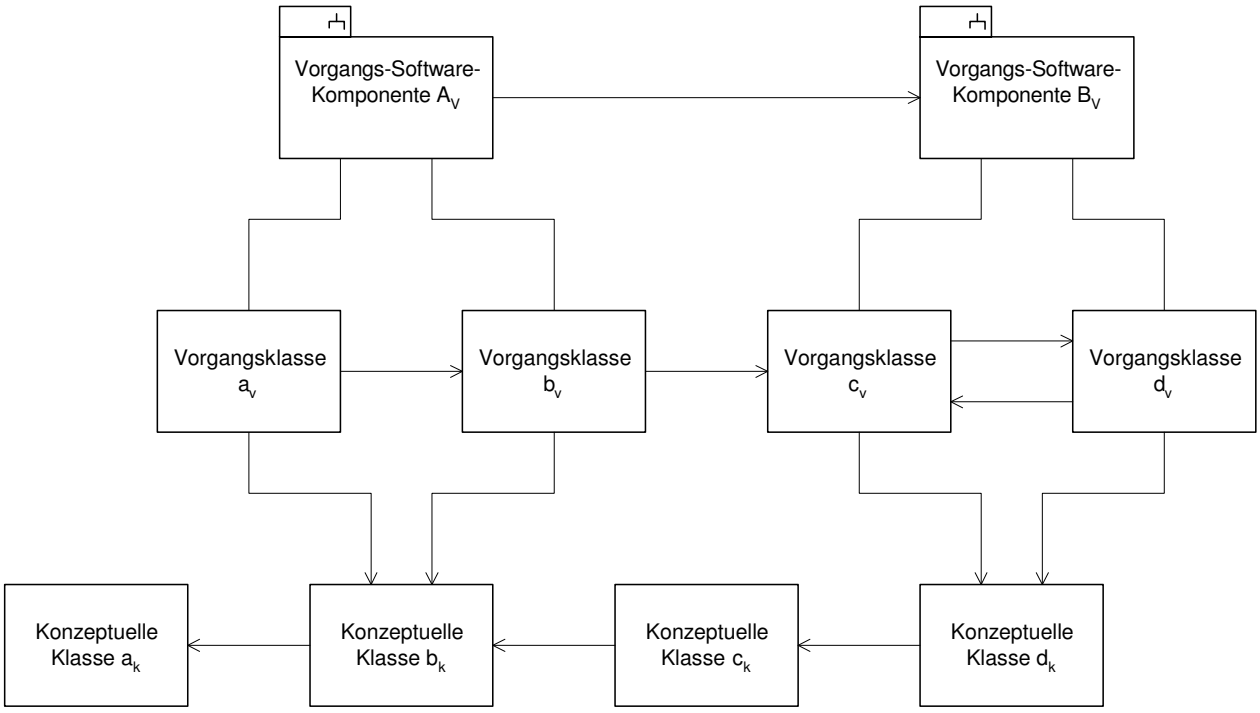
Name
Vorgangsklassen - Vorgangs-Software-Komponente
Kontext
Softwaretechnischer Entwurf der Anwendungsfunktionen eines komponentenbasierten Anwendungssystems. Die Spezifikation der Anwendungsfunktionen liegt in Form einer Struktur von fachlichen Vorgangsklassen und einer zugeordneten Struktur von fachlichen konzeptuellen Klassen vor. Die Struktur der fachlichen konzeptuellen Klassen entspricht der eines konzeptuellen Datenschemas. Insbesondere werden darin Existenzabhängigkeiten zwischen den fachlichen konzeptuellen Klassen berücksichtigt. Für den softwaretechnischen Entwurf der Anwendungsfunktionen eines komponentenbasierten Anwendungssystems stehen Vorgangs-Software-Komponenten und Entitäts-Software-Komponenten zur Verfügung.
Problem
Aus der Struktur der fachlichen Vorgangsklassen müssen zur Bildung von Vorgangs-Software-Komponenten geeignete Teilstrukturen abgegrenzt werden. Diese stellen daraufhin die Implementierungen der gebildeten Vorgangs-Software-Komponenten dar.
Kräfte
Die abgegrenzten Teilstrukturen gelten als geeignet, wenn sie einerseits einen abgeschlossenen fachlichen Belang abbilden und andererseits eine maximale innere Kohärenz und eine minimale äußere Bindung aufweisen.

Lösung

Eine maximale innere Kohärenz und eine minimale äußere Bindung wird durch eine Kombination der informalen Bindungsart und der funktionalen Bindungsart erreicht. Wie in Abschnitt 3.4.2 erläutert, liegt eine funktionale Bindung vor, wenn alle Klassen einer Software-Komponente an der Realisierung eines einzigen, abgeschlossenen Vorgangs beteiligt sind. In dem gleichen Abschnitt wird auch die informale Bindungsart definiert. Sie besteht, wenn alle Operatoren einer Software-Komponenten auf einer einzigen Datenstruktur operieren. Durch die Kombination beider Bindungsarten kann zudem gewährleistet werden, dass die resultierenden Software-Komponenten einen einzigen, abgeschlossenen fachlichen Belang realisieren.

In Bezug auf das genannte Problem im zugehörigen Kontext wird eine informale und funktionale Bindung dadurch erreicht, dass alle Vorgangsklassen, die auf derselben Teilstruktur konzeptueller Klassen operieren, die Implementierung einer Vorgangs-Software-Komponente bilden. Die Teilstrukturen konzeptueller Klassen ergeben sich aus den Existenzabhängigkeiten zwischen den konzeptuellen Klassen. Demnach beginnen die Abgrenzungen mit der konzeptuellen Klasse, die von allen anderen existenzabhängig ist, und enden mit der konzeptuellen Klasse, die von keiner anderen existenzabhängig ist.

Lösungsstruktur



Beziehungen

-

Mit dem folgenden Beziehungsmuster werden konzeptuelle Klassen der fachlichen Modellebene zu Entitäts-Software-Komponenten des Software-Architekturmodells abgegrenzt. Voraussetzung dafür ist eine bereits erfolgte Abgrenzung von Vorgangsklassen zu Vorgangs-Software-Komponenten anhand des gerade beschriebenen Beziehungsmusters *Vorgangsklassen - Vorgangs-Software-Komponente*.

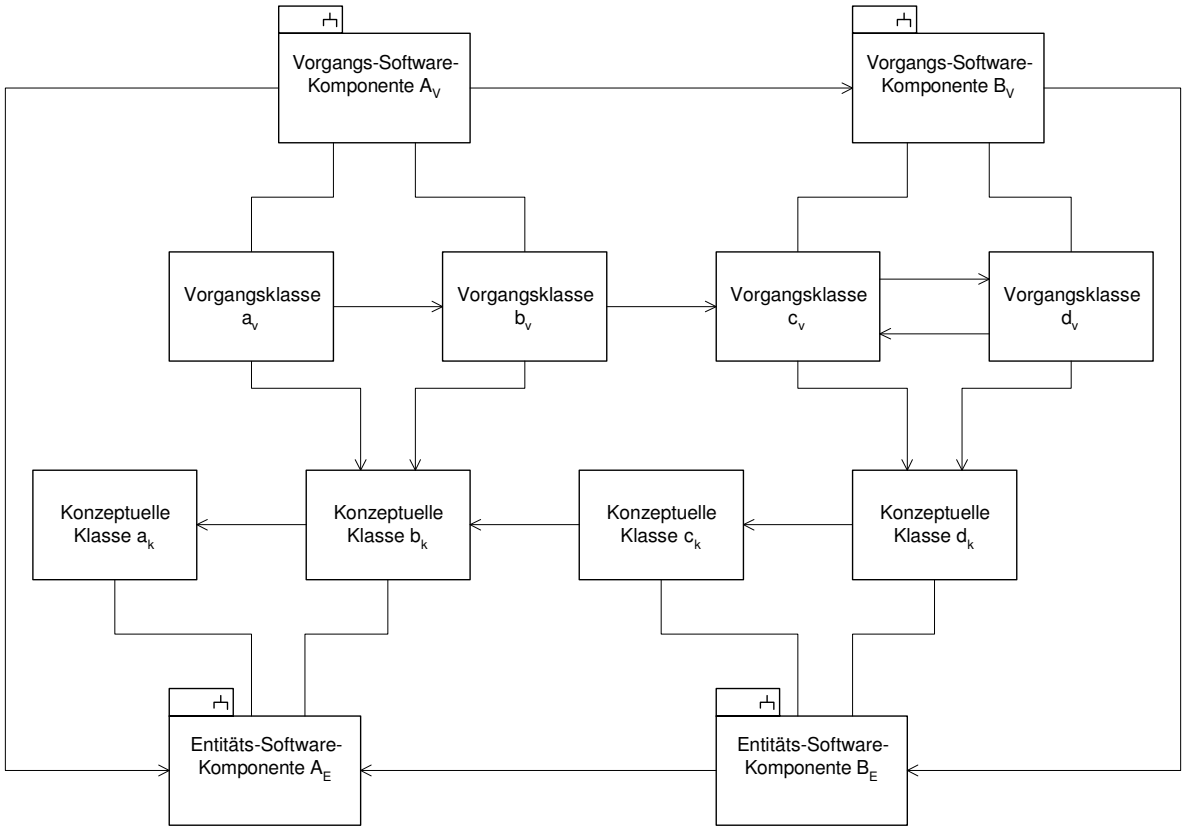
Name
Konzeptuelle Klassen - Entitäts-Software-Komponente
Kontext
Softwaretechnischer Entwurf der Anwendungsfunktionen eines komponentenbasierten Anwendungssystems. Die Spezifikation der Anwendungsfunktionen liegt in Form einer Struktur von fachlichen Vorgangsklassen und einer zugeordneten Struktur von fachlichen konzeptuellen Klassen vor. Die Struktur der fachlichen konzeptuellen Klassen entspricht der eines konzeptuellen Datenschemas. Insbesondere werden darin Existenzabhängigkeiten zwischen den fachlichen konzeptuellen Klassen berücksichtigt. Für den softwaretechnischen Entwurf der Anwendungsfunktionen eines komponentenbasierten Anwendungssystems stehen Vorgangs-Software-Komponenten und Entitäts-Software-Komponenten zur Verfügung. Aus der Struktur der fachlichen Vorgangsklassen wurden bereits Vorgangs-Software-Komponenten anhand des Beziehungsmusters *Vorgangsklassen - Vorgangs-Software-Komponente* abgegrenzt.
Problem
Analog zum Problem, das mit dem Beziehungsmuster *Vorgangsklassen - Vorgangs-Software-Komponente* gelöst wird, sind zur Erstellung von Entitäts-Software-Komponenten geeignete Teilstrukturen aus der Struktur der fachlichen konzeptuellen Klassen abzugrenzen. Die Teilstrukturen werden daraufhin ebenfalls als Implementierungen der gebildeten Entitäts-Software-Komponenten eingesetzt.
Kräfte
Eine abgegrenzte Teilstrukturen ist geeignet, wenn sie einerseits einen abgeschlossenen fachlichen Belang abbildet und andererseits eine maximale innere Kohärenz und eine minimale äußere Bindung aufweist.

Lösung

Wie bereits im Lösungsabschnitt des Beziehungsmusters *Vorgangsklassen - Vorgangs-Software-Komponente* erläutert, lässt sich eine maximale innere Kohärenz und eine minimale äußere Bindung durch eine Kombination der informalen Bindungsart und der funktionalen Bindungsart realisieren. Beide Bindungsformen werden in diesem Lösungsabschnitt ebenfalls näher erläutert.

Im vorliegenden Fall wird eine informale und funktionale Bindung dadurch erreicht, dass jede Teilstruktur konzeptueller Klassen, die der Abgrenzung einer Vorgangs-Software-Komponente diente, eine Implementierung einer Entitäts-Software-Komponente darstellt. Da sich die Abgrenzungen der Vorgangs-Software-Komponenten im Lösungsverfahren des Beziehungsmusters *Vorgangsklassen - Vorgangs-Software-Komponente* an den Existenzabhängigkeiten der konzeptuellen Klassen orientieren, werden Überlappungen zwischen den Entitäts-Software-Komponenten vermieden. Existenzabhängigkeiten zwischen konzeptuellen Klassen unterschiedlicher Entitäts-Software-Komponenten führen demnach zu Existenzabhängigkeiten zwischen diesen Entitäts-Software-Komponenten.

Lösungsstruktur



Beziehungen
Bevor dieses Muster angewendet werden kann, müssen Vorgangs-Software-Komponenten anhand des Beziehungsmusters *Vorgangsklassen - Vorgangs-Software-Komponente* abgegrenzt worden sein.

5.5 Einordnung in die SOM- Unternehmensarchitektur

Wie bereits in Abschnitt 2.4.1 festgestellt wurde, entspricht der Architekturrahmen, der mit der SOM-Unternehmensarchitektur beschrieben wird, einer Ausprägung des generischen Architekturrahmens. Die übergeordnete Metapher der SOM-Unternehmensarchitektur beschreibt ein betriebliches System aus der Innensicht als ein verteiltes System, das aus einer Menge autonomer, lose gekoppelter Objekte besteht, die im Hinblick auf die Erreichung übergeordneter Ziele kooperieren [FeSi01, 182]. Außerdem werden auf der Ebene der fachlichen Anwendungssystemspezifikation Anwendungssysteme als objektorientierte und objektintegrierte verteilte Systeme beschrieben [FeSi01, 202]. Demzufolge erfüllen die SOM-Unternehmensarchitektur und die Ebene der fachlichen Anwendungssystemspezifikation alle in Abschnitt 5.1.2 und 5.1.3 genannten Anforderungen. Deswegen ist die SOM-Unternehmensarchitektur als übergreifende Informationssystem-Architektur für das Software-Architekturmodell geeignet.

5.5.1 Metamodell der Ebene der fachlichen Anwendungssystemspezifikation

Im Rahmen der Erläuterungen zur SOM-Methodik wurde in Abschnitt 2.4.1.2 die Ebene der fachlichen Anwendungssystemspezifikation bereits kurz vorgestellt. Ein Anwendungssystem wird in der SOM-Methodik in Form eines konzeptuellen Objektschemas (KOS) und eines darauf aufbauenden Vorgangsobjektschemas (VOS) spezifiziert.

Metaobjekte des konzeptuellen Objektschemas (KOS)

Ein KOS besteht aus einer Menge von konzeptuellen Objekttypen (KOT), die untereinander in Beziehung stehen. Als Beziehungsarten sind die Assoziation, die Generalisierung oder die Aggregation einsetzbar [FeSi01, 204]. In der Struktur eines KOS werden Existenzabhängigkeiten zwischen den KOTs anhand einer quasi-hierarchischen Anordnung der KOTs berücksichtigt. Der Grad der Existenzabhängigkeit zwischen den KOTs nimmt von links nach rechts zu. Das heißt, dass auf der linken Seite eines KOS die existenzunabhängigen KOTs und rechts davon, unter Berücksichtigung steigender Existenzabhängigkeiten, die restlichen KOTs aneinander gereiht werden. Daraus ergibt sich eine Struktur, die graphentheoretisch einem ge-

richteten azyklischen Graphen entspricht [FeSi01, 143]. Dieses, auf die klare Visualisierung von Existenzabhängigkeiten ausgerichtete Strukturierungskonzept stammt ursprünglich aus dem **Strukturierten Entity-Relationship-Modell (SERM)** nach SINZ [Sinz88; Sinz92; FeSi01, 143 ff]. Mit dem SERM lassen sich konzeptuelle Datenschemata erstellen, die aufgrund dieses Strukturierungskonzepts signifikante Vorteile gegenüber Datenschemata besitzen, die mit dem **Entity-Relationship-Modell (ERM)** erstellt wurden. Ausführliche Erläuterungen zur Modellierung mit dem SERM und zu den Vorteilen des SERM gegenüber dem ERM sind in [Sinz88; Sinz92; FeSi01, 143 ff] nachzulesen. Grundsätzlich kann ein KOS als objektorientierte Erweiterung und Modifikation eines konzeptuellen Datenschemas im SERM interpretiert werden [FeSi01, 203]. Die Datenobjekttypen eines konzeptuellen Datenschemas im SERM entsprechen im Wesentlichen den konzeptuellen Objekttypen eines KOS. Allerdings besitzen Objekttypen des KOS zusätzlich Nachrichtendefinitionen und Operatoren. Außerdem wird nicht zwischen Gegenstands- und Beziehungs-Objekttypen unterschieden. Schließlich erfolgt die Realisierung von Beziehungen zwischen Objekttypen mithilfe von Objekt-Identifikatoren, d. h. unabhängig von den Attributwerten der Instanzen eines Objekttyps [FeSi01, 204]. Somit lässt sich ein konzeptuelles Datenschema im SERM auch als (statische) Datensicht eines KOS interpretieren, bei der nur Attribute und Beziehungen betrachtet werden [FeSi01, 205]. Demzufolge kann ein KOS als Teil der fachlichen Anwendungssystemspezifikation in der SOM-Methodik die Anforderungen der in Abschnitt 5.4.3 formulierten Beziehungsmuster erfüllen.

Metaobjekte des Vorgangsobjektschemas (VOS)

Zur vollständigen Spezifikation eines Anwendungssystems in der SOM-Methodik gehört ein Vorgangsobjektschema, das anhand eines oder mehrerer betrieblicher Objekte, für die ein Anwendungssystem spezifiziert werden soll, abgegrenzt wird [FeSi01, 210]. Ein VOS besteht aus einer Menge von Vorgangsobjekttypen (VOT), die untereinander in Beziehung stehen und auf diesem Weg die Zerlegungsstruktur der Gesamtaufgabe widerspiegeln. Im Unterschied zum KOS lässt sich als Beziehungsart für Beziehungen zwischen VOTs nur die Assoziation mit der semantischen Relation *interagiert_mit* verwenden [FeSi01, 211]. Jedem VOT werden Attribute in Form eines Teilgraphen des zugehörigen KOS zugeordnet. Die Operatoren eines VOTs beschreiben das Zusammenwirken der zugeordneten KOTs bei der Durchführung einer betrieblichen Teilaufgabe [FeSi01, 210].

Zusammenfassend lässt sich feststellen, dass eine fachliche Anwendungssystemspezifikation in der SOM-Unternehmensarchitektur, bestehend aus einem VOS und einem zugehörigen KOS, den Anforderungen in Abschnitt 5.1.2 und 5.1.3 vollständig entspricht. Für die fachliche Anwendungssystemspezifikation liegt in der SOM-

Methodik ein integriertes Metamodell vor, das in Abschnitt 2.4.1.2 bereits kurz dargestellt wurde. Allerdings basiert dieses Metamodell nicht auf dem in Abschnitt 5.2.2.1 eingeführten Meta-Metamodell. Deswegen enthält die Abbildung 5.28 ein überarbeitetes Metamodell für die fachliche Anwendungssystemspezifikation in der SOM-Methodik auf Basis des in Abschnitt 5.2.2.1 vorgestellten Meta-Metamodells.

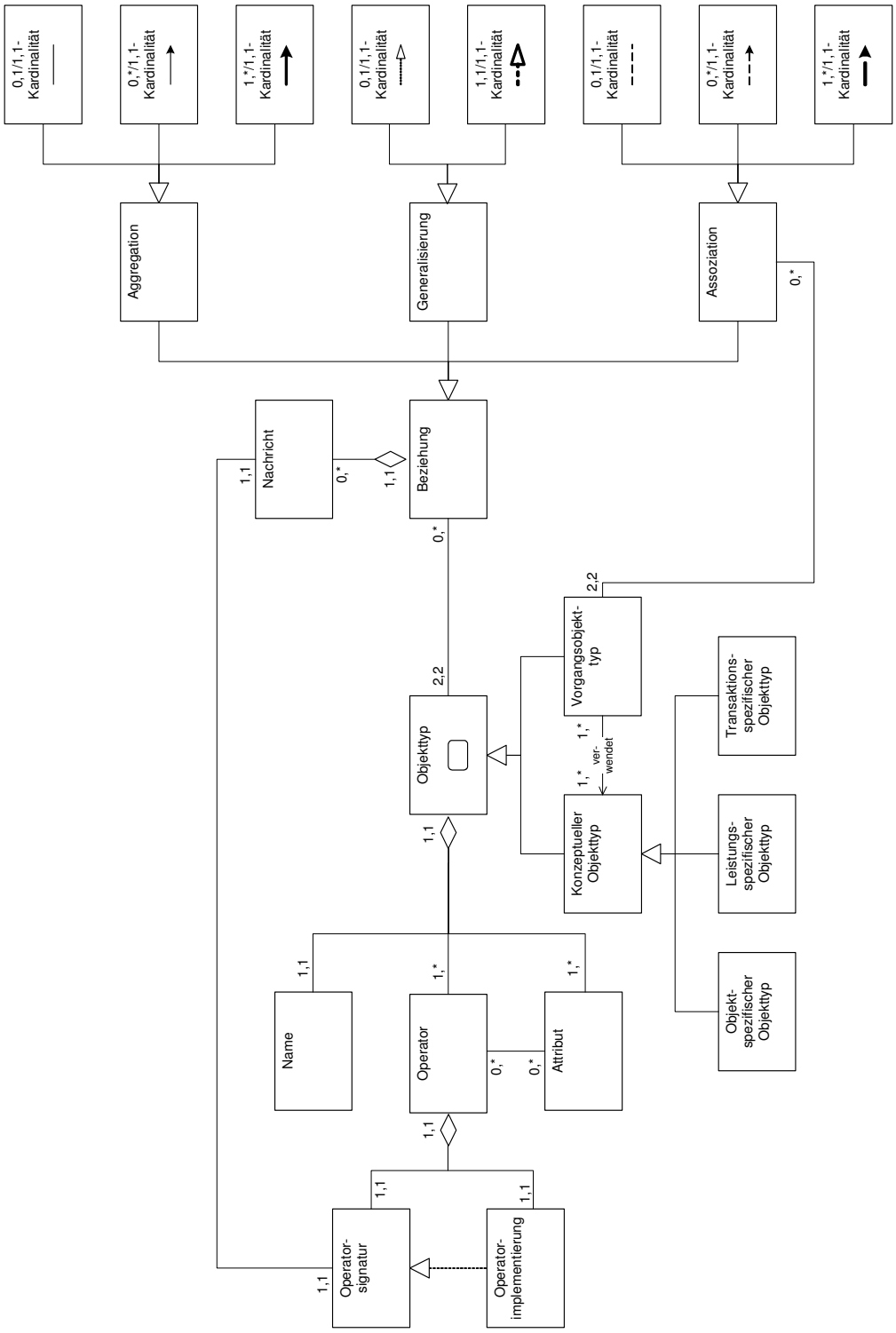


Abbildung 5.28: Überarbeitetes SOM-Metamodell für die fachliche Spezifikation von Anwendungssystemen

5.5.2 Beziehungs-Metamodelle

Analog zu den in Abschnitt 5.4.2 vorgestellten Beziehungs-Metamodellen, die das exemplarische Metamodell für objektorientierte und objektintegrierte fachliche Anwendungssystemspezifikationen mit den Metamodellen der Außenperspektive- und Innenperspektive-Modellebene des Software-Architekturmodells verknüpfen, werden im Folgenden Beziehungs-Metamodelle beschrieben, die das im vorangegangenen Abschnitt dargestellte Metamodell für die fachliche Anwendungssystemspezifikation in der SOM-Methodik mit den Metamodellen der Außenperspektive- und Innenperspektive-Modellebene des Software-Architekturmodells verbinden. Wie erwähnt, erfüllen fachliche Anwendungssystemspezifikationen in der SOM-Methodik die in Abschnitt 5.1.2 und 5.1.3 genannten Anforderungen an Spezifikationen der fachlichen Modellebene. Sie können als Grundlage für den softwaretechnischen Entwurf mit dem Software-Architekturmodell verwendet werden. Dies zeigt sich auch an der großen Übereinstimmung des integrierten Metamodells für die fachliche Anwendungssystemspezifikation in der SOM-Methodik mit dem exemplarischen Metamodell für objektorientierte und objektintegrierte fachliche Anwendungssystemspezifikationen aus Abschnitt 5.4.1. Demzufolge sind auch die in diesem Abschnitt beschriebenen Beziehungs-Metamodelle zu denen in Abschnitt 5.4.2 zum größten Teil gleich.

Zunächst wird das Beziehungs-Metamodell dargestellt, das das Metamodell für die fachliche Anwendungssystemspezifikation in der SOM-Methodik mit dem Kern-Metamodell der Außenperspektive-Modellebene verbindet. Es beschreibt im Wesentlichen die Zuordnung von Teilstrukturen bestimmter Vorgangsobjekttypen zu Vorgangs-Software-Komponenten bzw. Unterstützungs-Software-Komponenten und von Teilstrukturen konzeptueller Objekttypen zu Entitäts-Software-Komponenten. Dies impliziert, dass die Attribute der zugeordneten Objekttypen zugleich die Attribute der Software-Komponente selbst sind und dass die Operatoren der zugeordneten Objekttypen mit den Operatoren der Software-Komponente selbst übereinstimmen.

Metaobjekt des SOM-Metamodells für die fachliche Anwendungssystem-spezifikation	Kardi-nalität 1	Metaobjekt des Kern-Metamodells der Außenperspektive-Modellebene	Kardi-nalität 2
Vorgangsobjekttyp mit Namen, Operatorenimplementierungen und Attributen	1,*	Vorgangs-Software-Komponente	0,1
Vorgangsobjekttyp mit Namen, Operatorenimplementierungen und Attributen	1,*	Unterstützungs-Software-Komponente	0,1

Metaobjekt des SOM-Metamodells für die fachliche Anwendungssystem-spezifikation	Kardi-nalität 1	Metaobjekt des Kern-Metamodells der Außenperspektive-Modellebene	Kardi-nalität 2
Objektspezifischer Objekttyp mit Namen, Operatorenimplementie-rungen und Attributen	1,*	Entitäts-Software-Komponente	0,1
Leistungsspezifischer Objekttyp mit Namen, Operatorenimple-mentierungen und Attributen	1,*	Entitäts-Software-Komponente	0,1
Transaktionsspezifischer Objekt-typ mit Namen, Operatorenimple-mentierungen und Attributen	1,*	Entitäts-Software-Komponente	0,1
Generalisierung mit Kardinalität und Nachricht	0,*	Vorgangs-Software-Komponente	0,1
Generalisierung mit Kardinalität und Nachricht	0,*	Unterstützungs-Software-Komponente	0,1
Generalisierung mit Kardinalität und Nachricht	0,*	Entitäts-Software-Komponente	0,1
Generalisierung mit Kardinalität und Nachricht	0,1	Generalisierung mit Kardinalitä-ten und Komponentennachricht	0,1
Aggregation mit Kardinalität und Nachricht	0,*	Vorgangs-Software-Komponente	0,1
Aggregation mit Kardinalität und Nachricht	0,*	Unterstützungs-Software-Komponente	0,1
Aggregation mit Kardinalität und Nachricht	0,*	Entitäts-Software-Komponente	0,1
Aggregation mit Kardinalität und Nachricht	0,1	Aggregation mit Kardinalitäten, und Komponentennachricht	0,1
Assoziation mit Kardinalität und Nachricht	0,*	Vorgangs-Software-Komponente	0,1
Assoziation mit Kardinalität und Nachricht	0,*	Unterstützungs-Software-Komponente	0,1
Assoziation mit Kardinalität und Nachricht	0,*	Entitäts-Software-Komponente	0,1

Metaobjekt des SOM-Metamodells für die fachliche Anwendungssystem-spezifikation	Kardi-nalität 1	Metaobjekt des Kern-Metamodells der Außenperspektive-Modellebene	Kardi-nalität 2
Assoziation mit Kardinalität und Nachricht	0,1	Gerichtete Assoziation mit Kardinalitäten und Komponenten-nachricht	0,1
Operatorsignatur	1,1	Komponentenoperatorsignatur	0,1

Tabelle 5.13: Beziehungs-Metamodell zur Verbindung des überarbeiteten SOM-Metamodells für die fachliche Anwendungssystemspezifikation mit dem Kern-Metamodell der Außenperspektive-Modellebene

Das folgende Beziehungs-Metamodell beschreibt die Verknüpfung des Metamodells für die fachliche Anwendungssystemspezifikation in der SOM-Methodik mit dem Kern-Metamodell der Innenperspektive-Modellebene. In Abschnitt 5.3.5.1 wurde bereits verdeutlicht, dass die auf der fachlichen Modellebene abgegrenzten Teilstrukturen, die entweder Vorgangs- oder konzeptuelle Objekttypen enthalten und den entsprechenden Software-Komponenten zugeordnet werden, auch gleichzeitig deren Implementierung darstellen. Da im vorliegenden Software-Architekturmodell die Implementierungen von Software-Komponenten eines komponentenbasierten Anwendungssystems auf der Innenperspektive-Modellebene spezifiziert werden, stimmen die Metaobjekte Vorgangsobjekttyp und konzeptueller Objekttyp des Metamodells für die fachliche Anwendungssystemspezifikation mit den entsprechenden Metaobjekten Vorgangsklasse und konzeptuelle Klasse des Metamodells der Innenperspektive-Modellebene überein. Die Beziehungen zwischen Objekttypen einer auf der Ebene der fachlichen Anwendungssystemspezifikation abgegrenzten Teilstruktur entsprechen den Intra-Komponenten-Beziehungen auf der Innenperspektive-Modellebene. Dagegen korrespondieren die Beziehungen zwischen Objekttypen unterschiedlicher Teilstrukturen auf der Ebene der fachlichen Anwendungssystemspezifikation mit den Inter-Komponenten-Beziehungen auf der Innenperspektive-Modellebene.

Metaobjekt des SOM-Metamodells für die fachliche Anwendungssystem-spezifikation	Kardi-nalität 1	Metaobjekt des Kern-Metamodells der Innenperspektive-Modellebene	Kardi-nalität 2
Vorgangsobjekttyp mit Namen, Operatorenimplementierungen, Operatorensignaturen und Attributen	1,1	Vorgangsklasse mit Namen, Operatorenimplementierungen, Operatorensignaturen und Attributen	1,1

Metaobjekt des SOM-Metamodells für die fachliche Anwendungssystem-spezifikation	Kardi-nalität 1	Metaobjekt des Kern-Metamodells der Innenperspektive-Modellebene	Kardi-nalität 2
Objektspezifischer Objekttyp mit Namen, Operatorenimplementierungen, Operatorensignaturen und Attributen	0,1	Konzeptuelle Klasse mit Namen, Operatorenimplementierungen, Operatoren-signaturen und Attributen	1,1
Leistungsspezifischer Objekttyp mit Namen, Operatorenimplementierungen, Operatorensignaturen und Attributen	0,1	Konzeptuelle Klasse mit Namen, Operatorenimplementierungen, Operatoren-signaturen und Attributen	1,1
Transaktionsspezifischer Objekttyp mit Namen, Operatorenimplementierungen, Operatorensignaturen und Attributen	0,1	Konzeptuelle Klasse mit Namen, Operatorenimplementierungen, Operatoren-signaturen und Attributen	1,1
Generalisierung mit Kardinalität und Nachricht	1,1	Generalisierung mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,1
Generalisierung mit Kardinalität und Nachricht	1,1	Generalisierung mit Kardinalitäten und Nachricht (Inter-Komponenten-Beziehung)	0,1
Aggregation mit Kardinalität und Nachricht	1,1	Aggregation mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,1
Aggregation mit Kardinalität und Nachricht	1,1	Aggregation mit Kardinalitäten und Nachricht (Inter-Komponenten-Beziehung)	0,1
Assoziation mit Kardinalität und Nachricht	1,1	Gerichtete Assoziation mit Kardinalitäten und Nachricht (Intra-Komponenten-Beziehung)	0,1
Assoziation mit Kardinalität und Nachricht	1,1	Gerichtete Assoziation mit Kardinalitäten und Nachricht (Inter-Komponenten-Beziehung)	0,1

Tabelle 5.14: Beziehungs-Metamodell zur Verbindung des überarbeiteten SOM-Metamodells für die fachliche Anwendungssystemspezifikation mit dem Kern-Metamodell der Innenperspektive-Modellebene

5.5.3 Beziehungsmuster

Das Metamodell für die fachliche Anwendungssystemspezifikation in der SOM-Methodik entspricht, wie bereits erwähnt, im Wesentlichen dem exemplarischen Metamodell für objektorientierte und objektintegrierte fachliche Anwendungssystemspezifikationen aus Abschnitt 5.4.1. Demzufolge können die in Abschnitt 5.4.3 formulierten Beziehungsmuster für die Modellierung von Verknüpfungs- und Transformationsbeziehungen zwischen der fachlichen Modellebene und den beiden Modellebenen des Software-Architekturmodells auf die Modellierung der Beziehungen zwischen den Objekttyp-Teilstrukturen einer fachlichen Anwendungssystemspezifikation und den Software-Komponenten-Strukturen der Außenperspektive-Modellebene einerseits und den Klassenstrukturen auf der Innenperspektive-Modellebene des Software-Architekturmodells andererseits ausnahmslos übertragen werden.

6 Fallstudien zur Evaluation des komponentenbasierten Software-Architekturmodells

In diesem Abschnitt wird ein erster Nachweis der praktischen Anwendbarkeit des vorgestellten komponentenbasierten Software-Architekturmodells anhand von zwei Fallstudien aus der betrieblichen Praxis erbracht. Die erste Fallstudie dient der Durchführung des *Development for Reuse*. Das heißt, aus dem softwaretechnischen Entwurf des Anwendungssystems für diese Fallstudie resultieren wiederverwendbare Software-Komponenten. Im Rahmen der zweiten Fallstudie wird schließlich das *Development with Reuse* realisiert. Folglich verwendet der zugehörige softwaretechnische Entwurf des Anwendungssystems Software-Komponenten aus dem softwaretechnischen Entwurf der ersten Fallstudie wieder. Damit auch die praktische Realisierbarkeit einer horizontalen Wiederverwendung nachgewiesen wird, stammen die Fallstudien aus unterschiedlichen betrieblichen Domänen.

Die Erstellung der Fachkonzepte für die Anwendungssysteme beider Fallstudien erfolgt anhand der SOM-Methodik, da diese einerseits eine umfassende und methodisch durchgängige Fachkonzeptentwicklung unterstützt und andererseits alle Voraussetzungen für einen lückenlosen Übergang von der Ebene der fachlichen Anwendungssystemspezifikation zum komponentenbasierten Software-Architekturmodell erfüllt.

Abschnitt 6.1 beschreibt die Entwicklung eines komponentenbasierten Anwendungssystems für die erste Fallstudie, Abschnitt 6.2 stellt die entsprechende Entwicklung für die zweite Fallstudie dar.

6.1 Fallstudie Personalvermittlungsunternehmen

Die erste Fallstudie basiert auf den Ergebnissen einer Geschäftsprozessanalyse eines Personalvermittlungsunternehmens, durchgeführt im Rahmen einer Diplomarbeit am Lehrstuhl für Wirtschaftsinformatik der Universität Bamberg [Pane00]. Die analysierten Prozesse wurden hinsichtlich einer möglichen Effizienzsteigerung durch den Einsatz von internet-basierten Anwendungssystemen untersucht und diesbezüglich überarbeitet [Pane00, 44 ff]. Folglich sind die ursprünglichen Prozesse zunächst noch unabhängig von eventuell zu treffenden Automatisierungsentscheidungen. Dies ist eine wesentliche Voraussetzung dafür, dass die SOM-Methodik von der Geschäftsprozessmodellebene über die Ebene der fachlichen Anwendungssystemspezifikation bis zur erweiterten Ebene des softwaretechnischen Entwurfs angewendet werden kann.

6.1.1 Unternehmensplan

Die analysierten Prozesse des Personalvermittlungsunternehmens realisieren einen bereits existierenden Unternehmensplan, der in der SOM-Methodik anhand eines Objektsystems und eines Zielsystems beschrieben wird. Die folgende Tabelle gibt die wesentlichen Ergebnisse der entsprechenden Untersuchungen in der zugrundeliegenden Arbeit wieder [Pane00, 24 f].

<table>
<tr><th>Objektsystem</th><th colspan="2">Zielsystem</th></tr>
<tr><td rowspan="2">Es handelt sich um ein Personalvermittlungsunternehmen.</td><td>Das Sachziel des Personalvermittlungsunternehmens ist die Vermittlung von Bewerbern an Firmen mit freien Stellen.</td><td>Sachziel</td></tr>
<tr><td rowspan="4">Das Personalvermittlungsunternehmen vermittelt Bewerbern freie Stellen bei Firmen. Dieser Prozess wird als Arbeitgebervermittlung identifiziert.

Außerdem vermittelt es Firmen mit freien Stellen vorab beurteilte Bewerber. Dieser Prozess entspricht der Arbeitnehmervermittlung.

Marktforschungsinstitute führen Analysen des Stellen- und Bewerbermarktes für das Personalvermittlungsunternehmen durch.</td><td>Dabei soll die Qualität der Vermittlungsergebnisse vor deren Quantität stehen.

Unter Berücksichtigung des vorgegebenen Qualitätsziels wird für die Vermittlung eine Kombination aus Umsatz- und Gewinnmaximierung angestrebt.

Die Betreuung der Firmen und der Bewerber soll freundlich, zuverlässig und effektiv sein.

Außerdem wird festgelegt, dass Vermittlungen für Bewerber kostenlos sein sollen.

Ferner soll Werbung nur gezielt und in geeigneten Medien durchgeführt werden.</td><td>Formalziele</td></tr>
</table>

Tabelle 6.1: Objekt- und Zielsystem des Personalvermittlungsunternehmens (in Anlehung an [Pane00, 24 f])

6.1.2 Geschäftsprozessmodell

Die folgenden Ausführungen fassen die wesentlichen Ergebnisse der Geschäftsprozessmodellierung des Personalvermittlungsunternehmens zusammen. Grundlegende Erläuterungen zum Ablauf der Geschäftsprozessmodellierung in der SOM-Methodik sind der entsprechenden Literatur zu entnehmen [FeSi01, 185 ff; FeSi95, 214 ff]. Die durchgeführten Zerlegungen des Interaktionsschemas (IAS) stellen Abbildung 6.1 bis Abbildung 6.6 dar. Das mit der letzten Zerlegung des Interaktionsschemas korrespondierende Vorgangs-Ereignis-Schema (VES) ist in Abbildung A.I im Anhang enthalten. Aus Gründen der Übersichtlichkeit wurde auf die Darstellung der übrigen Zerlegungen des Vorgangs-Ereignis-Schemas verzichtet. Aus den gleichen Gründen sind in Tabelle 6.2 und Tabelle 6.3 die durchgeführten Zerlegungen der betrieblichen Transaktionen und der betrieblichen Objekte nochmals in Form von Zerlegungsbäumen zusammengefasst.

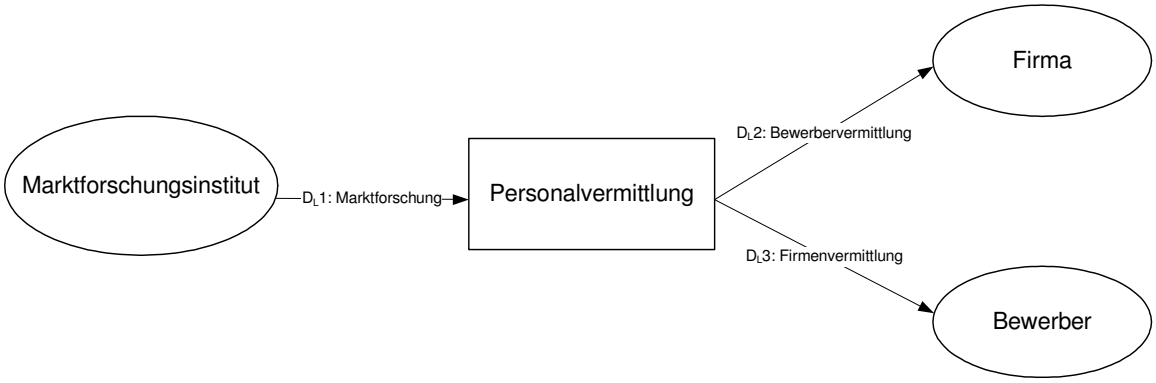


Abbildung 6.1: Initiales Interaktionsschema für den Geschäftsprozess Stellenvermittlung

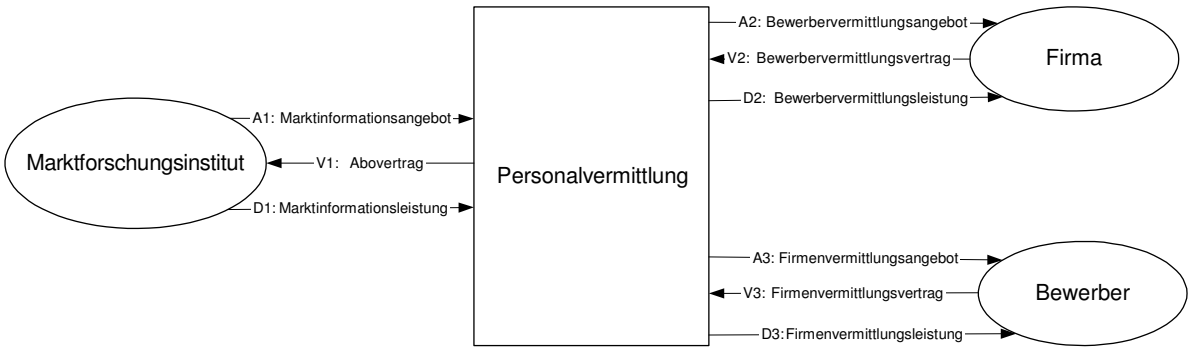


Abbildung 6.2: Interaktionsschema für den Geschäftsprozess Stellenvermittlung (1.Zerlegung)

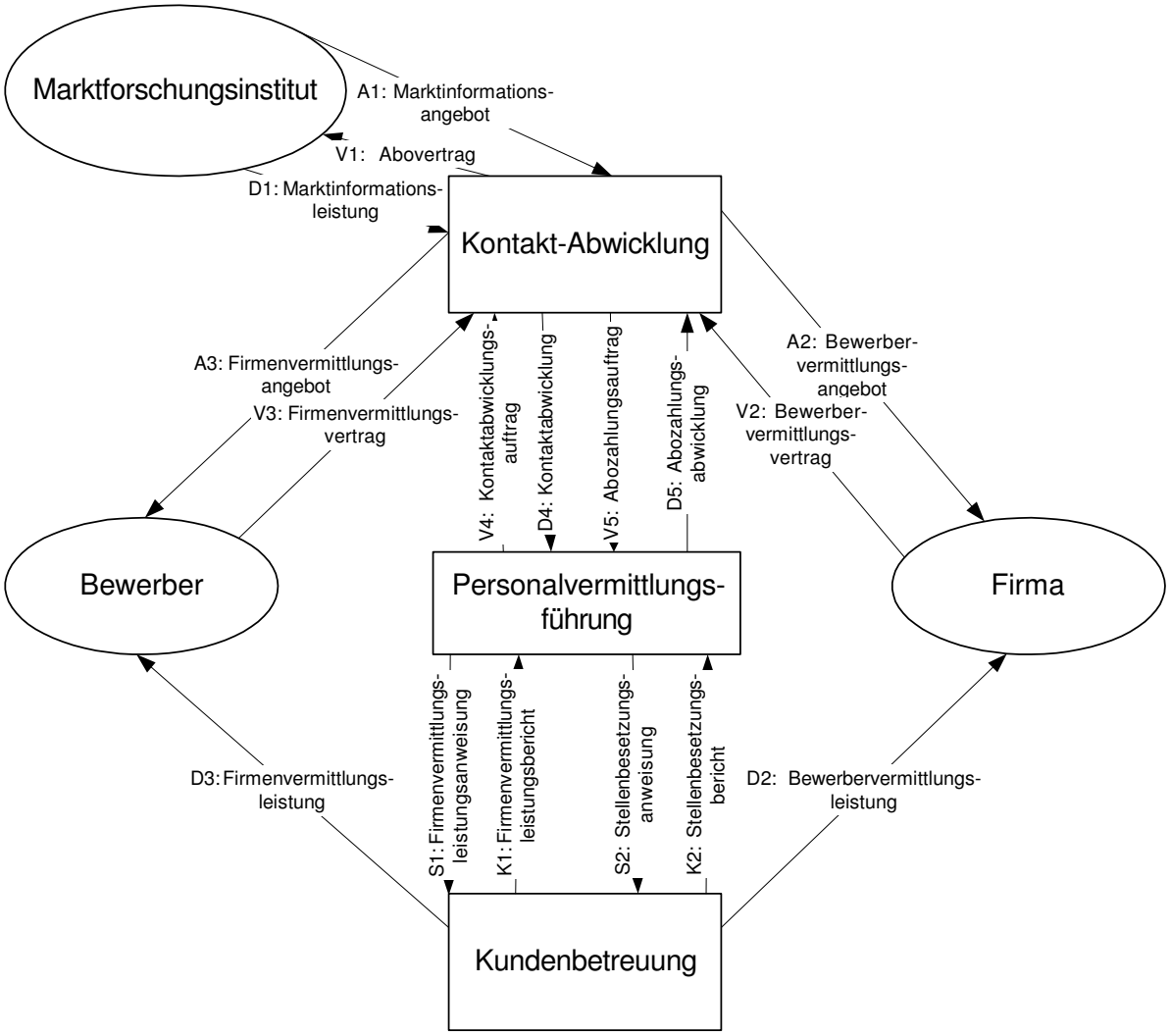


Abbildung 6.3: Interaktionsschema für den Geschäftsprozess Stellenvermittlung (2.Zerlegung)

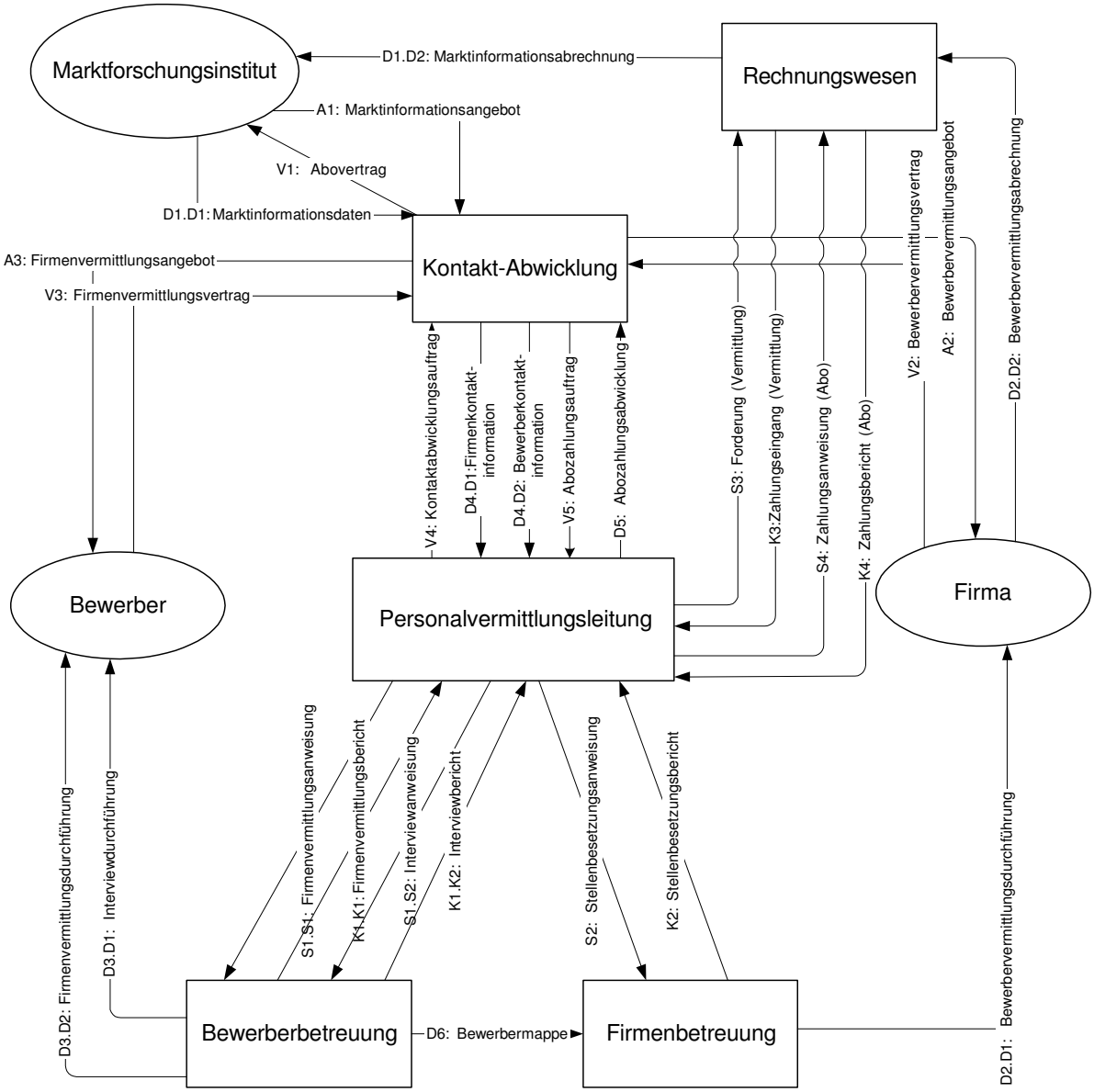


Abbildung 6.4: Interaktionsschema für den Geschäftsprozess Stellenvermittlung (3.Zerlegung)

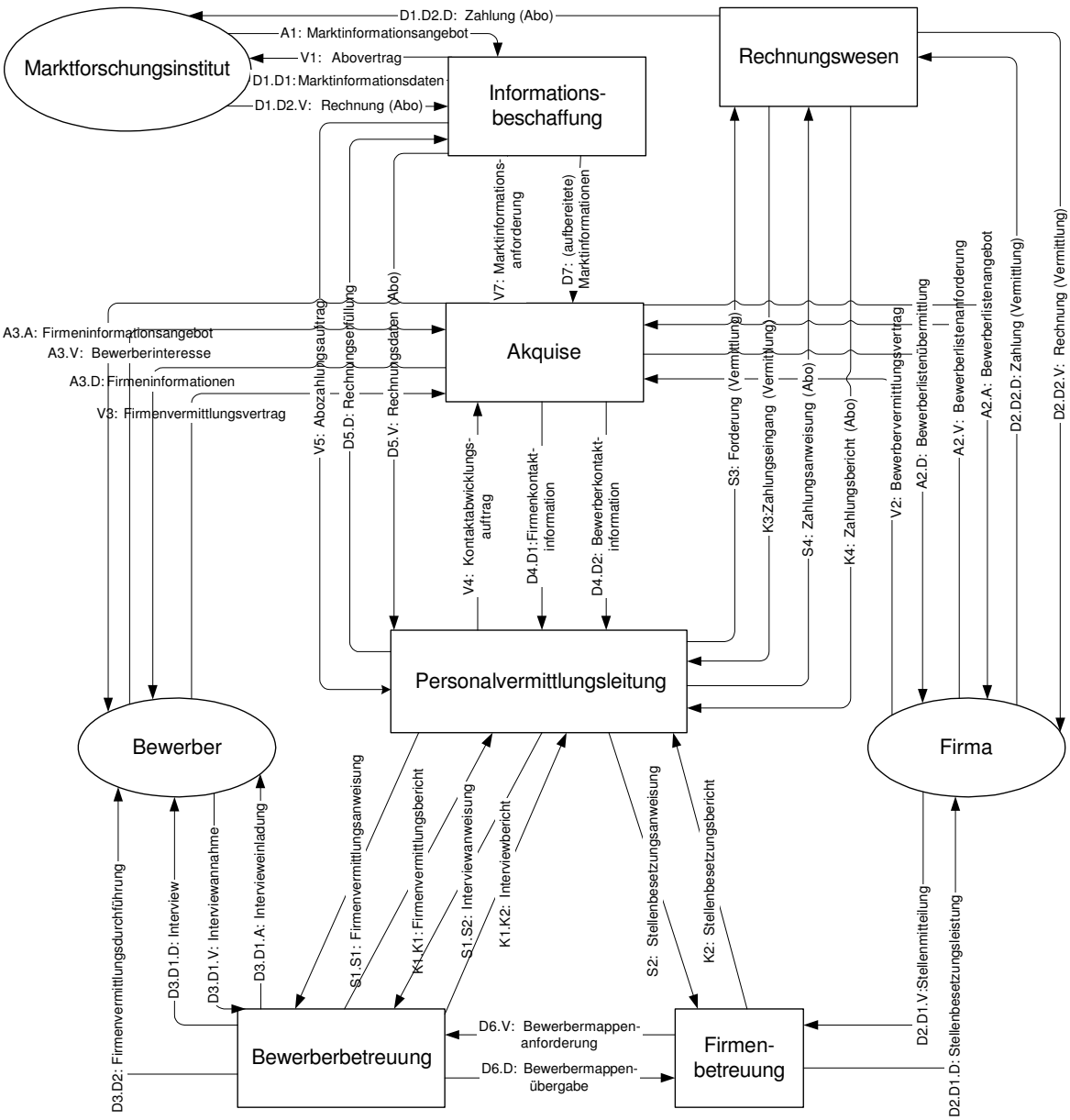


Abbildung 6.5: Interaktionsschema für den Geschäftsprozess Stellenvermittlung (4.Zerlegung)

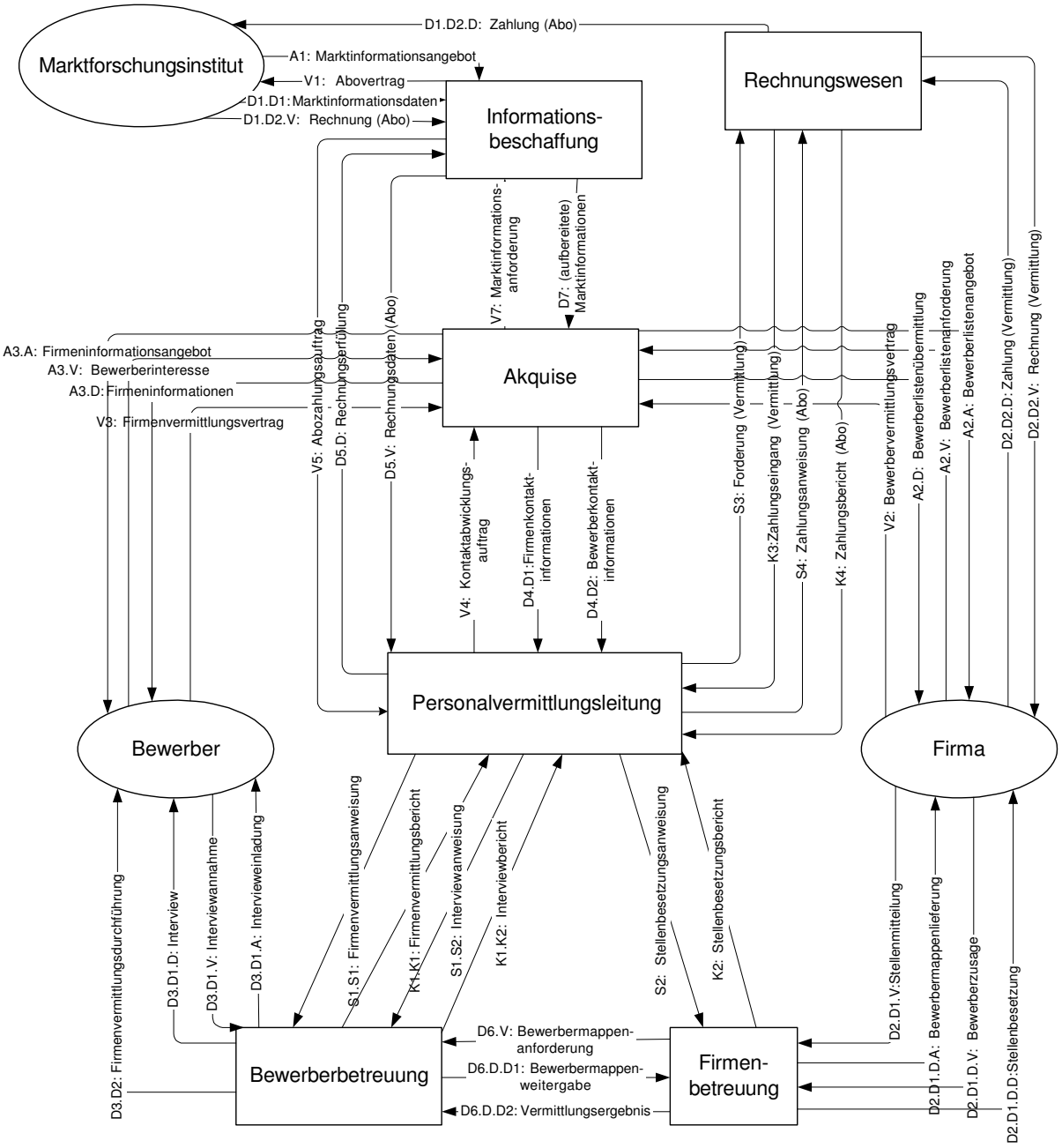


Abbildung 6.6: Interaktionsschema für den Geschäftsprozess Stellenvermittlung (5.Zerlegung)

Transaktionszerlegung
D_L1: Marktforschung
A1: Marktinformationsangebot
V1: Abovertrag
D1: Marktinformationsleistung
D1.D1: Marktinformationsdaten
D1.D2: Marktinformationsabrechnung
D1.D2.V: Rechnung (Abo)
D1.D2.D: Zahlung (Abo)
D_L2: Bewerbervermittlung
A2: Bewerbervermittlungsangebot
A2.A: Bewerberlistenangebot
A2.V: Bewerberlistenanforderung
A2.D: Bewerberlistenübermittlung
V2: Bewerbervermittlungsvertrag
D2: Bewerbervermittlungsleistung
D2.D1: Bewerbervermittlungsdurchführung
D2.D1.V: Stellenmitteilung
D2.D1.D: Stellenbesetzungsleistung
D2.D1.D.A: Bewerbermappenlieferung
D2.D1.D.V: Bewerberzusage
D2.D1.D.D: Stellenbesetzung
D2.D2: Bewerbervermittlungsabrechnung
D2.D2.V: Rechnung (Vermittlung)
D2.D2.D: Zahlung (Vermittlung)
D_L3: Firmenvermittlung
A3: Firmenvermittlungsangebot
A3.A: Firmeninformationsangebot
A3.V: Bewerberinteresse
A3.D: Firmeninformationen
V3: Firmenvermittlungsvertrag
D3: Firmenvermittlungsleistung
D3.D1: Interviewdurchführung
D3.D1.A: Intervieweinladung
D3.D1.V: Interviewannahme
D3.D1.D: Interview
D3.D2: Firmenvermittlungsdurchführung

Tabelle 6.2: Transaktionszerlegung des Geschäftsprozesses Stellenvermittlung

Objektzerlegung
Personalvermittlung
Personalvermittlungsführung
Rechnungswesen
Personalvermittlungsleitung
Kontakt-Abwicklung
Akquise
Informationsbeschaffung
Kundenbetreuung
Bewerberbetreuung
Firmenbetreuung
V4: Kontaktabwicklungsauftrag
D4: Kontaktabwicklung
D4.D1: Firmenkontaktinformationen
D4.D2: Bewerberkontaktinformationen
V5: Abozahlungsauftrag
D5: Abozahlungsabwicklung
D5.V: Rechnungsdaten (Abo)
D5.D: Rechnungserfüllung
D6: Bewerbermappe
D6.V: Bewerbermappenanforderung
D6.D: Bewerbermappenübergabe
D6.D1: Bewerbermappenweitergabe
D6.D2: Vermittlungsergebnis
V7: Marktinformationsanforderung
D7: (aufbereitete) Marktinformationen
S1: Firmenvermittlungsleistungsanweisung
S1.S1: Firmenvermittlungsanweisung
S1.S2: Interviewanweisung
K1: Firmenvermittlungsleistungsbericht
K1.K1: Firmenvermittlungsbericht
K1.K2: Interviewbericht
S2: Stellenbesetzungsanweisung
K2: Stellenbesetzungsbericht
S3: Forderung (Vermittlung)
K3: Zahlungseingang (Vermittlung)
S4: Zahlungsanweisung (Abo)
K4: Zahlungsbericht (Abo)

Tabelle 6.3: Objektzerlegung des Geschäftsprozesses Stellenvermittlung

Das Personalvermittlungsunternehmen bezieht von einem Marktforschungsinstitut im Rahmen eines Abonnements Informationen über stellensuchende Personen und stellenanbietende Firmen. Grundlage dafür ist ein entsprechender Vertrag, der vorher

mit dem Marktforschungsinstitut abgeschlossen wurde und in dem auch die Zahlung des Abonnements durch das Personalvermittlungsunternehmens festgelegt ist. Anhand dieser Informationen wirbt das Personalvermittlungsunternehmen zielgruppenspezifisch in geeigneten Medien für sein Vermittlungsleistungsangebot.

Falls eine stellensuchende Person Interesse an der angebotenen Vermittlungsleistung zeigt, werden ihr vom Personalvermittlungsunternehmen vorab Informationen über die Firmen zugeschickt, für die sie sich im Speziellen interessiert. Wenn sich die stellensuchende Person daraufhin zur Annahme der Vermittlungsleistung entscheidet, schließt das Personalvermittlungsunternehmen mit ihr einen Vermittlungsvertrag ab. Nach Abschluss des Vermittlungsvertrages wird die stellensuchende Person, die sich nun in der Rolle eines Bewerbers befindet, von dem Personalvermittlungsunternehmen zu einem Interview eingeladen. Im Falle eines positiven Interviewergebnisses stellt das Personalvermittlungsunternehmen aus dem Ergebnis und den persönlichen Daten des Bewerbers eine Bewerbermappe zusammen. Anderenfalls informiert das Personalvermittlungsunternehmen den Bewerber über das negative Ergebnis des Interviews und kündigt daraufhin den Vermittlungsvertrag. Diese Maßnahme trägt zur angestrebten Qualitätssicherung der Vermittlungsergebnisse bei.

Wenn sich eine stellenanbietende Firma für die angebotene Vermittlungsleistung interessiert, bekommt sie vom Personalvermittlungsunternehmen zunächst eine Liste mit anonymisierten Bewerbern zugeschickt, deren Profile grundsätzlich zu den von der Firma genannten freien Stellen passen. Falls die Firma daraufhin die angebotene Vermittlungsleistung annehmen möchte, schließt sie mit dem Personalvermittlungsunternehmen ebenfalls einen Vermittlungsvertrag ab. Danach teilt sie dem Personalvermittlungsunternehmen freie Stellen nach Bedarf mit, woraufhin dieses der Firma geeignete Bewerbermappen zuschickt. Falls der Vorgang zur Einstellung eines Bewerbers führt, ist die Firma zur Zahlung der getätigten Vermittlungsleistung verpflichtet.

6.1.3 Fachliche Anwendungssystemspezifikation

Zur fachlichen Spezifikation von Anwendungssystemen für das Personalvermittlungsunternehmen werden zunächst die aus dem Geschäftsprozessmodell resultierenden betrieblichen Aufgaben und Transaktionen hinsichtlich ihrer Automatisierbarkeit untersucht und auf Basis dieser Untersuchungsergebnisse deren Automatisierungsgrade festgelegt werden. Die diesbezüglichen Ergebnisse sind aus Gründen der besseren Lesbarkeit in den Tabellen A.1 und A.2 im Anhang zusammengefasst.

Initiale fachliche Anwendungsspezifikation

Wie bereits in Abschnitt 2.4.1.2 erwähnt, kann man in der SOM-Methodik eine initiale fachliche Spezifikation objektorientierter Anwendungssysteme unmittelbar aus einem

Geschäftsprozessmodell ableiten. Das dafür erforderliche Beziehungs-Metamodell, das die beiden Modellebenen integriert, wurde ebenfalls in diesem Abschnitt vorgestellt. Darüber hinaus sind zur korrekten Ableitung weitere Regeln notwendig (siehe [FeSi01, 202 ff]).

Somit basiert auch die Ableitung der initialen fachlichen Spezifikation objektorientierter Anwendungssysteme aus dem Geschäftsprozessmodell des Personalvermittlungsunternehmens auf dem genannten Beziehungs-Metamodell und den ergänzenden Ableitungsregeln. Ausgehend von der letzten Zerlegungsstufe des im vorangegangenen Abschnitt dargestellten IAS und des korrespondierenden VES werden ein **initiales konzeptuelles Objektschema (KOS)** und **initiale Vorgangsobjektschemata (VOS)** erstellt. Diese sind in Abbildung A.II und A.III im Anhang dargestellt. Die Bedeutungen der konzeptuellen Objekttypen und der Vorgangsobjekttypen wurden bereits in Abschnitt 2.4.1.2 erläutert.

Weitere Ausarbeitung der initialen fachlichen Anwendungsspezifikation

Das initiale KOS und die initialen VOS sind für eine vollständige Spezifikation noch weiter auszuarbeiten. Die entsprechenden Schritte werden in [FeSi01, 207] bzw. [FeSi01, 211] beschrieben. Im Wesentlichen sind zunächst diejenigen Objekttypen aus den initialen Objektschemata zu entfernen, deren korrespondierende Aufgaben und Transaktionen nicht automatisiert werden. Daraufhin ordnet man den verbleibenden Objekttypen Attribute, Operatoren und zugehörige Nachrichtendefinitionen zu. Schließlich erfordert die weitere Ausarbeitung, dass Objekttypen aus unterschiedlichen Gründen zusammengefasst werden. KOTs werden zusammengefasst, wenn sie sich in ihren Attributen und bzw. oder ihren Operatoren decken oder weitgehend überlappen. Dies vermeidet eine Daten- und Funktionsredundanz. Dagegen werden VOTs zusammengefasst, wenn deren korrespondierende Aufgaben zur Wahrung der semantischen Integrität des Anwendungssystems stets gemeinsam durchzuführen sind.

Bei der Ausarbeitung des initialen KOS zeigt sich auf Basis der Tabellen A.1 und A.2, dass die konzeptuellen Objekttypen weder mit nicht-automatisierten Aufgaben noch mit nicht-automatisierten Transaktionen korrespondieren. Folglich muss kein konzeptueller Objekttyp aus dem initialen KOS entfernt werden. Die Zuordnung von Attributen zu den KOTs und die daraufhin durchzuführende Zusammenfassung von KOTs aufgrund von Attributüberlappungen sind aus Gründen der Verständlichkeit und besseren Lesbarkeit in gemeinsamen Tabellen beschrieben (vgl. Tabelle 6.4 und Tabelle 6.5). Da jeder KOT nur Operatoren für das Lesen und das Schreiben seiner Attributwerte aufweist, wird auf die Zuordnung von entsprechenden Nachrichtendefinitionen zu jedem KOT zugunsten der besseren Lesbarkeit verzichtet. In Abbildung A.IV im Anhang wird das resultierende KOS dargestellt. Anhand dieses Diagramms sind

auch die Existenzabhängigkeitsbeziehungen zwischen den KOTs und die zugehörigen Kardinalitäten ersichtlich.

Initiale OOTs und LOTs	**Konsolidierte OOTs und LOTs**
Bewerberbetreuung (BB_ID, Leiter, Anzahl MA, Tel, E-Mail, WWW-Adresse, Kapazität)	dto.
Bewerber (B_ID, Name, Vorname, Tel, Straße, PLZ, Ort, E-Mail, Alter, Qualifikation)	dto.
Bewerbervermittlung (BV_ID, Branche, Beschreibung, Dauer, Preis)	Bewerbervermittlungsprodukt (BV_ID, Branche, Beschreibung, Dauer, Preis)
Marktforschung (MF_ID, Dauer, Erhebungsmenge, Zielgruppe, Gebiet, Preis)	Marktforschungsprodukt (MF_ID, Dauer, Erhebungsmenge, Zielgruppe, Gebiet, Preis)
Marktforschungsinstitut (MFI_ID, Name, Tel, Straße, PLZ, Ort, E-Mail, WWW-Adresse)	dto.
Informations-Beschaffung (IB_ID, Leiter, Anzahl MA, Tel, E-Mail, WWW-Adresse, Kapazität)	dto.
Akquise (Akq_ID, Leiter, Anzahl MA, Tel, E-Mail, WWW-Adresse, Kapazität)	dto.
Personalvermittlungsleitung (PVL_ID, Leiter, Anzahl MA, Tel, E-Mail)	dto.
Firmenvermittlung (FV_ID, Branche, Position, Beschreibung, Dauer, Preis)	Firmenvermittlungsprodukt (FV_ID, Branche, Position, Beschreibung, Dauer, Preis)
Firma (F_ID, Name, Branche, Tel, Straße, PLZ, Ort, E-Mail, WWW-Adresse)	dto.
Firmenbetreuung (FB_ID, Leiter, Anzahl MA, Tel, E-Mail, WWW-Adresse, Kapazität)	dto.
Rechnungswesen (R_ID, Leiter, Anzahl MA, Tel, E-Mail)	dto.

Tabelle 6.4: Initiale und konsolidierte OOTs und LOTs des Personalvermittlungsunternehmens

Initiale TOTs	**Konsolidierte TOTs**
Marktinformationsangebot (MIA_ID, MF_ID, MFI_ID, IB_ID, Datum, Ansprechpartner)	dto.
Kontaktabwicklungsauftrag (KAA_ID, MF_ID, BV_ID, AKQ_ID, PVL_ID, FV_ID, Datum)	dto.
Marktinformationsanforderung (MIAN_ID, KAA_ID, IB_ID, Zeitraum, Umfang, Profil)	dto.
Abovertrag (AV_ID, MIAN_ID, MIA_ID, Unterzeichner, Datum, Zeitraum, Kündigungsfrist, Rabatt, Erhebungsmenge, Preis, Prüfstatus)	dto.
Marktinformationsdaten (MID_ID, AV_ID, Datum, Menge) (aufbereitete) Marktinformationen (MI_ID, MID_ID, Datum, Menge)	Marktinformationsdaten (MID_ID, AV_ID, Datum, Menge)
Rechnung (Abo) (RA_ID, AV_ID, Fälligkeit, Betrag, Datum, KtoNr, BLZ, Status) Abozahlungsauftrag (AZ_ID, RA_ID, Betrag, Fälligkeit) Rechnungsdaten (Abo) (RD_ID, AZ_ID, Fälligkeit, Betrag, Datum, KtoNr, BLZ) Zahlungsanweisung (Abo) (ZAA_ID, R_ID, RD_ID, Fälligkeit)	Kreditorrechnung (KR_ID, R_ID, AV_ID, Fälligkeit, Betrag, Datum, KtoNr, BLZ, Status)
Firmeninformationsangebot (FIA_ID, MI_ID, B_ID, Firma, Branche)	Firmeninformationsangebot (FIA_ID, MID_ID, B_ID, F_ID, Branche)
Bewerberinteresse (BI_ID, FIA_ID, Datum)	dto.
Firmeninformationen (FI_ID, BI_ID, Name, Branche, Größe, Umsatz, Sitz)	Firmeninformationen (FI_ID, BI_ID, F_ID, Größe, Umsatz, Sitz)
Firmenvermittlungsvertrag (FVV_ID, FI_ID, Bewerber, Datum, Dauer, Ansprechpartner) Bewerberkontaktinformationen (BKI_ID, FVV_ID, Name, Vorname, Straße, PLZ, Ort, E-Mail, Tel) Firmenvermittlungsanweisung (FVA_ID, IVB_ID, Name, Datum, Firma)	Firmenvermittlungsvertrag (FVV_ID, FI_ID, B_ID, Datum, Dauer, Ansprechpartner)

Initiale TOTs	**Konsolidierte TOTs**
Interviewanweisung (IVA_ID, BKI_ID, BB_ID, Datum, Zeit, Ort, Art) Intervieweinladung (IVE_ID, IVA_ID, Datum, Zeit, Ort, Art) Interviewannahme (IVAN_ID, IVE_ID, Datum, Zeit, Ort)	Interviewvorbereitung (IVV_ID, FVV_ID, BB_ID, Datum, Zeit, Ort, Art)
Interview (IV_ID, IVAN_ID, Interviewer, Teilnehmer, Datum, Zeit, Ort, Art, Dauer, Ergebnis) Interviewbericht (IVB_ID, IVAN_ID, Status, Erfolg)	Interviewprotokoll (IVP_ID, IVV_ID, Interviewer, B_ID, Datum, Zeit, Ort, Art, Dauer, Ergebnis)
Zahlung (Abo) (ZA_ID, ZAA_ID, Betrag, Währung, KtoNr, BLZ) Zahlungsbericht (Abo) (ZBA_ID, ZAA_ID, Status) Rechnungserfüllung (RE_ID, ZBA_ID, Datum)	Kreditorzahlung (KZ_ID, KR_ID, Betrag, Währung, KtoNr, BLZ, Datum, Status)
Bewerberlistenangebot (BLA_ID, MI_ID, F_ID, Anzahl, Verfügbarkeit) Bewerberlistenanforderung (BLAN_ID, BLA_ID, Anzahl, Verfügbarkeit)	Bewerberlistenangebot (BLA_ID, MI_ID, F_ID, Anzahl, Verfügbarkeit)
Bewerberlistenübermittlung (BLUE_ID, BLAN_ID, Bewerberliste)	Bewerberlistenkopf (BLK_ID, BLA_ID, Datum, Anzahl) Bewerberlistenpositionen (BLP_ID, BLK_ID, B_ID, Verfügbarkeit)
Bewerbervermittlungsvertrag (BVV_ID, BLUE_ID, Firma, Datum, Ansprechpartner, Dauer) Firmenkontaktinformationen (FKI_ID, BVV_ID, Name, Straße, PLZ, Ort, E-Mail, Tel) Stellenbesetzungsanweisung (SBA_ID, FKI_ID, FB_ID, Firma, Datum, Dauer)	Bewerbervermittlungsvertrag (BVV_ID, BLK_ID, FB_ID, F_ID, Datum, Ansprechpartner, Dauer)
Stellenmitteilung (SM_ID, SBA_ID, BVV_ID, Beginndatum, Position, Beschreibung, Dauer)	Stellenmitteilung (SM_ID, BVV_ID, Beginndatum, Position, Beschreibung, Dauer)
Bewerbermappenanforderung (BMA_ID, FVA_ID, SM_ID, Datum, Priorität, Status)	Bewerbermappenanforderung (BMA_ID, FVV_ID, SM_ID, Datum, Priorität, Status)

Initiale TOTs	**Konsolidierte TOTs**
Bewerbermappenweitergabe (BMW_ID, BMA_ID, Datum, Priorität) Bewerbermappenlieferung (BML_ID, BMW_ID, Datum, Status, Priorität)	Bewerbermappe (BM_ID, BMA_ID, IVP_ID, Status, Umfang, Priorität)
Bewerberzusage (BZ_ID, BML_ID, Datum, Beginndatum, Position, Verantwortlicher) Stellenbesetzung (SB_ID, BZ_ID, Bewerber, Firma, Beginndatum, Position, Dauer) Vermittlungsergebnis (VME_ID, BZ_ID, Datum, Status) Firmenvermittlungsdurchführung (FVD_ID, VME_ID, Datum, Ort, Bewerber, Firma) Firmenvermittlungsbericht (FVB_ID, VME_ID, Datum, Ort, Bewerber, Firma, Verantwortlicher) Stellenbesetzungsbericht (SBB_ID, VME_ID, Datum, Ort, Bewerber, Firma, Verantwortlicher)	Stellenvermittlung (SV_ID, BM_ID, B_ID, F_ID, Beginndatum, Position, Dauer, Verantwortlicher, Datum, Ort)
Forderung (Vermittlung) (FOV_ID, FVB_ID, SBB_ID, R_ID, Betrag, Fälligkeit) Rechnung (Vermittlung) (RV_ID, FOV_ID, Betrag, Fälligkeit, KtoNr, BLZ, Status)	Debitorrechnung (DR_ID, SV_ID, R_ID, Betrag, Fälligkeit, KtoNr, BLZ, Status)
Zahlung (Vermittlung) (ZV_ID, RV_ID, Betrag, KtoNr, BLZ) Zahlungseingang (Vermittlung) (ZEV_ID, ZV_ID, Datum, KtoNr, BLZ)	Debitorzahlung (DZ_ID, DR_ID, Betrag, Datum, KtoNr, BLZ)

Tabelle 6.5: Initiale und konsolidierte TOTs des Personalvermittlungsunternehmens

Bei der Ausarbeitung des initialen VOS müssen aus diesem keine Vorgangsobjekttypen entfernt werden, da, wie bereits erwähnt, keine zugehörigen nichtautomatisierten Aufgaben vorliegen. Die Attribute werden den Vorgangsobjekttypen in Form von zugehörigen Teilgraphen des ausgearbeiteten KOS zugeordnet. Dabei überlappen sich die Teilgraphen unterschiedlicher VOS in der Regel, was zu unübersichtlichen und schwer verständlichen Diagrammen führt. Deshalb wird in der vorliegenden Arbeit die Zuordnung von Attributen zu Vorgangsobjekttypen anhand einer Tabelle dargestellt, die in der ersten Spalte jeder Zeile diejenigen Vorgangsobjektty-

pen enthält, deren zuzuordnende Teilgraphen des ausgearbeiteten KOS sich überlappen, und in der zweiten Spalte der gleichen Zeile den konzeptuellen Objekttyp aufweist, der im zuzuordnenden Teilgraphen am existenzabhängigsten, das heißt am weitesten rechts angeordnet ist (vgl. Tabelle 6.6). Auf eine Zusammenfassung von Vorgangsobjekttypen wird noch verzichtet, da diese im Rahmen der noch folgenden Definition von Vorgangs-Software-Komponenten ohnehin stattfindet. Die Zuordnung von Nachrichtendefinitionen zu jedem Vorgangsobjekttyp erfolgt wiederum aus Übersichtlichkeitsgründen anhand einer Tabelle (vgl. Tabelle A.3 im Anhang).

VOTs	**KOTs**
Marktinformationsangebot> >Marktinformationsangebot	Marktinformationsangebot (MIA_ID, MF_ID, MFI_ID, IB_ID, Datum, Ansprechpartner)
Kontaktabwicklungsauftrag> >Kontaktabwicklungsauftrag	Kontaktabwicklungsauftrag (KAA_ID, MF_ID, BV_ID, AKQ_ID, PVL_ID, FV_ID, Datum)
Marktinformationsanforderung> >Marktinformationsanforderung	Marktinformationsanforderung (MIAN_ID, KAA_ID, IB_ID, Zeitraum, Umfang, Profil)
Abovertrag> >Abovertrag	Abovertrag (AV_ID, MIAN_ID, MIA_ID, Unterzeichner, Datum, Zeitraum, Kündigungsfrist, Rabatt, Erhebungsmenge, Preis, Prüfstatus)
Marktinformationsdaten> >Marktinformationsdaten >(aufbereitete) Marktinformationen (aufbereitete) Marktinformationen>	Marktinformationsdaten (MID_ID, AV_ID, Datum, Menge)
Rechnung (Abo)> >Rechnung (Abo) Abozahlungsauftrag> >Abozahlungsauftrag Rechnungsdaten (Abo)> > Rechnungsdaten (Abo) Zahlungsanweisung (Abo)> >Zahlungsanweisung (Abo)	Kreditorrechnung (KR_ID, R_ID, AV_ID, Fälligkeit, Betrag, Datum, KtoNr, BLZ, Status)
Firmeninformationsangebot> >Firmeninformationsangebot	Firmeninformationsangebot (FIA_ID, MID_ID, B_ID, F_ID, Branche)
Bewerberinteresse> >Bewerberinteresse	Bewerberinteresse (BI_ID, FIA_ID, Datum)

VOTs	**KOTs**
Firmeninformationen> >Firmeninformationen	Firmeninformationen (FI_ID, BI_ID, F_ID, Größe, Umsatz, Sitz)
Firmenvermittlungsvertrag> >Firmenvermittlungsvertrag Bewerberkontaktinformationen> >Bewerberkontaktinformationen Firmenvermittlungsanweisung> >Firmenvermittlungsanweisung	Firmenvermittlungsvertrag (FVV_ID, FI_ID, B_ID, Datum, Dauer, Ansprechpartner)
Interviewanweisung> >Interviewanweisung Intervieweinladung> >Intervieweinladung Interviewannahme> >Interviewannahme	Interviewvorbereitung (IVV_ID, FVV_ID, BB_ID, Datum, Zeit, Ort, Art)
Interview> >Interview Interviewbericht> >Interviewbericht	Interviewprotokoll (IVP_ID, IVV_ID, Interviewer, B_ID, Datum, Zeit, Ort, Art, Dauer, Ergebnis)
Zahlung (Abo)> >Zahlung (Abo) Zahlungsbericht (Abo)> >Zahlungsbericht (Abo) Rechnungserfüllung> >Rechnungserfüllung	Kreditorzahlung (KZ_ID, KR_ID, Betrag, Währung, KtoNr, BLZ, Datum, Status)
Bewerberlistenangebot> >Bewerberlistenangebot >Rechnungserfüllung Rechnungserfüllung> Bewerberlistenanforderung> >Bewerberlistenanforderung	Bewerberlistenangebot (BLA_ID, MI_ID, F_ID, Anzahl, Verfügbarkeit)
Bewerberlistenübermittlung> >Bewerberlistenübermittlung	Bewerberlistenkopf (BLK_ID, BLA_ID, Datum, Anzahl) Bewerberlistenpositionen (BLP_ID, BLK_ID, B_ID, Verfügbarkeit)

VOTs	**KOTs**
Bewerbervermittlungsvertrag> >Bewerbervermittlungsvertrag Firmenkontaktinformation> >Firmenkontaktinformation Stellenbesetzungsanweisung> >Stellenbesetzungsanweisung	Bewerbervermittlungsvertrag (BVV_ID, BLK_ID, FB_ID, F_ID, Datum, Ansprechpartner, Dauer)
Stellenmitteilung> >Stellenmitteilung	Stellenmitteilung (SM_ID, BVV_ID, Beginndatum, Position, Beschreibung, Dauer)
Bewerbermappenanforderung> >Bewerbermappenanforderung	Bewerbermappenanforderung (BMA_ID, FVV_ID, SM_ID, Datum, Priorität, Status)
Bewerbermappenweitergabe> >Bewerbermappenweitergabe Bewerbermappenlieferung> >Bewerbermappenlieferung	Bewerbermappe (BM_ID, BMA_ID, IVP_ID, Status, Umfang, Priorität)
Bewerberzusage> >Bewerberzusage Stellenbesetzung> >Stellenbesetzung Vermittlungsergebnis> >Vermittlungsergebnis Firmenvermittlungsdurchführung> >Firmenvermittlungsdurchführung Firmenvermittlungsbericht> >Firmenvermittlungsbericht Stellenbesetzungsbericht> >Stellenbesetzungsbericht	Stellenvermittlung (SV_ID, BM_ID, B_ID, F_ID, Beginndatum, Position, Dauer, Verantwortlicher, Datum, Ort)
Forderung> >Forderung Rechnung (Vermittlung)> >Rechnung (Vermittlung)	Debitorrechnung (DR_ID, SV_ID, R_ID, Betrag, Fälligkeit, KtoNr, BLZ, Status)

VOTs	KOTs
Zahlung (Vermittlung)> >Zahlung (Vermittlung) Zahlungseingang (Vermittlung)> >Zahlungseingang (Vermittlung)	Debitorzahlung (DZ_ID, DR_ID, Betrag, Datum, KtoNr, BLZ)

Tabelle 6.6: Zuordnung von KOTs zu VOTs des Personalvermittlungsunternehmens

In den vorangegangenen Ausarbeitungen der Objektschemata wurde auf die Definition von Operatoren verzichtet. Dies folgt im anschließenden Entwurf von Software-Komponenten.

6.1.4 Softwaretechnischer Entwurf

Das ausgearbeitete KOS und die ausgearbeiteten VOS bilden nun den Ausgangspunkt für den softwaretechnischen Entwurf der Anwendungssysteme. Zum Nachweis der praktischen Anwendbarkeit des vorgestellten komponentenbasierten Software-Architekturmodells genügt die Beschreibung eines Ausschnitts des softwaretechnischen Entwurfs der Anwendungssysteme dieser Fallstudie.

Der weitere Aufbau des Abschnitts orientiert sich an den Ebenen und den Sichten des komponentenbasierten Software-Architekturmodells. Die in Kapitel 5 vorgeschlagene Vorgehensweise für das *Development for Reuse* sieht vor, dass zunächst die Struktur- und Verhaltensmerkmale des komponentenbasierten Anwendungssystems aus der Außenperspektive spezifiziert und daraufhin die zugehörige Spezifikation der Struktur- und Verhaltensmerkmale des komponentenbasierten Anwendungssystems aus der Innenperspektive durchgeführt werden soll.

Im Vergleich dazu müssen beim *Development with Reuse*, womit sich die zweite Fallstudie beschäftigt, nur die Struktur- und Verhaltensmerkmale des komponentenbasierten Anwendungssystems aus der Außenperspektive spezifiziert werden. Die resultierenden Software-Komponenten aus der Außenperspektive dienen daraufhin als konkrete Muster für die Suche nach entsprechenden wiederverwendbaren Software-Komponenten.

Da die Spezifikation des komponentenbasierten Anwendungssystems aus der Außenperspektive auf einer Abgrenzung von Teilstrukturen aus der fachlichen Anwendungssystemspezifikation beruht, muss diese in beiden Fällen durchgeführt werden. Die Teilstrukturen werden allerdings nur beim *Development for Reuse* zur Spezifikation der Struktur- und Verhaltensmerkmale des komponentenbasierten Anwendungssystems aus der Innenperspektive weiterverwendet.

Deshalb können beim *Development for Reuse* die Struktur- und Verhaltensmerkmale des komponentenbasierten Anwendungssystems aus der Außenperspektive und die zugehörigen Struktur- und Verhaltensmerkmale des komponentenbasierten Anwendungssystems aus der Innenperspektive auch simultan spezifiziert werden, so wie im Falle des folgenden softwaretechnischen Entwurf der vorliegenden Fallstudie.

Spezifikation der Struktur- und Verhaltensmerkmale eines komponentenbasierten Anwendungssystems aus der Außenperspektive und aus der Innenperspektive

Anhand der in Abschnitt 5.4.3 vorgestellten Beziehungsmuster *Vorgangsklassen - Vorgangs-Software-Komponenten* und *Konzeptuelle Klassen - Entitäts-Software-Komponenten* werden im ersten Schritt die Vorgangs-Software-Komponenten und die Entitäts-Software-Komponenten abgegrenzt. Gemäß den Musterbeschreibungen determiniert dabei die Abgrenzung der Vorgangs-Software-Komponenten die darauffolgende Definition der Entitäts-Software-Komponenten. Daher werden im Folgenden zunächst die Vorgangs-Software-Komponenten und auf der Grundlage dieses Ergebnisses die Entitäts-Software-Komponenten abgegrenzt.

Abgrenzung von Vorgangs-Software-Komponenten

Die Vorgangsobjekttypen, die eine Vorgangs-Software-Komponente bilden, lassen sich unmittelbar aus der im vorangegangenen Abschnitt vorgestellten Tabelle 6.6 ablesen. Wie bereits erwähnt, stellt diese Tabelle die Zuordnung von Teilgraphen des ausgearbeiteten KOS zu den Vorgangsobjekttypen der VOS dar. Im vorliegenden Zusammenhang ist insbesondere die erste Spalte dieser Tabelle von Interesse, in der je Zeile diejenigen Vorgangsobjekttypen zusammengefasst sind, deren zuzuordnende Teilgraphen des ausgearbeiteten KOS sich überlappen. Diese Vorgangsobjekttypen bilden eine Vorgangs-Software-Komponente.

Vorliegende Arbeit stellt zwei Vorgangs-Software-Komponenten einschließlich den zugehörigen Entitäts-Software-Komponenten vor. Die erste Vorgangs-Software-Komponente besteht aus den Vorgangsobjekttypen *Rechnung (Abo)>*, *>Rechnung (Abo)*, *Abozahlungsauftrag>*, *>Abozahlungsauftrag*, *Rechnungsdaten (Abo)>*, *>Rechnungsdaten (Abo)*, *Zahlungsanweisung (Abo)>* und *>Zahlungsanweisung (Abo)* sowie den Beziehungen untereinander. Der mit dieser Vorgangs-Software-Komponente abgebildete fachliche Belang beschreibt die Bearbeitung von Kreditorrechnungen. Deswegen wird sie mit dem fachlichen Namen *Kreditorrechnungsbearbeitung* bezeichnet. Die zweite Vorgangs-Software-Komponente enthält die Vorgangsobjekttypen *Abovertrag>* und *>Abovertrag* einschließlich den zugehörigen Beziehungen. Mit dem fachlichen Namen *Abovertragsabschluss* wird dem abgebildeten fachlichen Belang Rechnung getragen, der den Abschluss eines Abonnement-Vertrags beschreibt.

Beide Vorgangs-Software-Komponenten stehen miteinander in Beziehung, da sich, gemäß den zugrunde gelegten VOS, der Vorgangsobjekttyp *>Abovertrag* der *Abovertragsabschluss*-Komponente auf den Vorgangsobjekttyp *Rechnung (Abo)>* der *Kreditorrechnungsbearbeitung*-Komponente bezieht.

Wie erwähnt, bilden die Nachrichtendefinitionen der in einer Vorgangs-Software-Komponente gekapselten Vorgangsobjekttypen den Ausgangspunkt für die Ableitung ihrer fachlichen Operatorensignaturen. Demnach basieren die Operatorensignaturen der *Abovertragsabschluss*-Komponente auf den in der Tabelle A.3 im Anhang spezifizierten Nachrichtendefinitionen *erstellenAbovertrag* und *annehmenAbovertrag* und die Operatorensignaturen der *Kreditorrechnungsbearbeitung*-Komponente auf den ebenfalls in Tabelle A.3 aufgeführten Nachrichtendefinitionen *erstellenRechnung*, *annehmenRechnung*, *vorbereitenRechnungsbegleichung, prüfenVertragsdaten, begleichenRechnung*, *prüfenRechnung*, *freigebenZahlung* und *anweisenZahlung*.

Abgrenzung von Entitäts-Software-Komponenten und deren Zuordnung zu den Vorgangs-Software-Komponenten

Diesen Vorgangs-Software-Komponenten werden nun Entitäts-Software-Komponenten zugeordnet, deren Bestimmung von der gewählten Abgrenzung der Vorgangs-Software-Komponenten determiniert wird. Gemäß dem in Abschnitt 5.4.3 vorgestellten Beziehungsmuster *Konzeptuelle Klassen - Entitäts-Software-Komponenten* bilden diejenigen konzeptuellen Objekttypen eine Entitäts-Software-Komponente, die der Abgrenzung einer Vorgangs-Software-Komponente dienten. Der Existenzabhängigste dieser konzeptuellen Objekttypen lässt sich direkt aus der zweiten Spalte der im vorangegangenen Abschnitt vorgestellten Tabelle 6.6 ablesen. Sein Name ist zugleich der Name der Entitäts-Software-Komponente selbst. Da die Abgrenzung der Vorgangs-Software-Komponente von den existenzabhängigen zu den existenzunabhängigen konzeptuellen Objekttypen erfolgt, werden auch Existenzabhängigkeitsbeziehungen zwischen den Entitäts-Software-Komponenten spezifiziert.

Im vorliegenden Fall besteht die Entitäts-Software-Komponente, die der *Kreditorrechnungsbearbeitung*-Komponente zugeordnet wird, aus den konzeptuellen Objekttypen *Kreditorrechnung* und *Rechnungswesen* sowie der zugehörigen Beziehung. Der konzeptuelle Objekttyp *Kreditorrechnung* bestimmt den Namen dieser Entitäts-Software-Komponente. Die *Abovertragsabschluss*-Komponente bekommt eine Entitäts-Software-Komponente zugewiesen, die die konzeptuellen Objekttypen *Abovertrag*, *Marktinformationsangebot*, *Marktinformationsanforderung*, *Kontaktabwicklungsauftrag*, *Marktforschungsinstitut*, *Informationsbeschaffung*, *Marktforschungsprodukt*, *Akquise*, *Personalvermittlungsleitung*, *Bewerbervermittlungsprodukt* und *Firmenvermittlungsprodukt* sowie deren Beziehungen untereinander aufweist. Dabei

bestimmt der konzeptuelle Objekttyp *Abovertrag* den Namen dieser Entitäts-Software-Komponente. Aus der Existenzabhängigkeitsbeziehung zwischen dem konzeptuellen Objekttyp *Kreditorrechnung* und dem konzeptuellen Objekttyp *Abovertrag* im zugrunde liegenden KOS entsteht eine gerichtete Assoziation von der *Kreditorrechnungs*-Komponente zur *Abovertrags*-Komponente.

Spezifikation von Kommunikations-Software-Komponenten auf Basis der Vorgangs-Software-Komponenten

Im nächsten Schritt werden für die Vorgangs-Software-Komponenten, deren korrespondierende Aufgaben für die Teilautomatisierung vorgesehen sind, Kommunikations-Software-Komponenten für die Mensch-Computer-Kommunikation spezifiziert. Grundlage dafür ist demzufolge die Tabelle A.1 im Anhang, die die Automatisierungsgrade der Aufgaben des Personalvermittlungsunternehmens wiedergibt. Aus dieser Tabelle wird ersichtlich, dass im Rahmen des gewählten Ausschnitts die Aufgaben *Abovertrag*>, *>Abozahlungsauftrag*, *>Rechnungsdaten (Abo)* und *>Zahlungsanweisung (Abo)* zu teilautomatisieren sind. Die mit diesen Aufgaben korrespondierenden und gleichlautenden Vorgangsobjekttypen weisen die Nachrichtendefinitionen *erstellenAbovertrag*, *prüfenVertragsdaten*, *prüfenRechnung* und *anweisenZahlung* auf. Der erstgenannte Vorgangsobjekttyp und die zugehörige Nachrichtendefinition ist der *Abovertragsabschluss*-Komponente zugeordnet. Demnach wird für diese Vorgangs-Software-Komponente eine Kommunikations-Software-Komponente mit dem Namen *Abovertragsabschluss-Dialog* spezifiziert, deren Operatorsignatur sich an der genannten Nachrichtendefinition der *Abovertragsabschluss*-Komponente orientiert. Die restlichen Vorgangsobjekttypen und Nachrichtendefinitionen gehören zur *Kreditorrechnungsbearbeitung*-Komponente. Deswegen wird dieser Vorgangs-Software-Komponente ebenfalls eine Kommunikations-Software-Komponente mit dem Namen *Kreditorrechnungsbearbeitung-Dialog* zugeordnet. Die zugehörigen Operatorensignaturen sind ebenfalls aus den entsprechenden Nachrichtendefinitionen abgeleitet. In der vorliegenden Arbeit erfolgt die Beschreibung von Operatoren im softwaretechnischen Entwurf und in der Implementierung aus Gründen der besseren Lesbarkeit nur anhand ihrer Namen und nicht anhand ihrer gesamten Signatur. Somit stellen die folgenden Tabellen die Ableitungen der Operatorensignaturen der genannten Software-Komponenten auch nur anhand der jeweiligen Operatornamen dar.

Operator *Abovertragsabschluss*	**Operatoren *Abovertragsabschluss-Dialog***
erstellenAbovertrag	*eingebenVertragsdaten*

Tabelle 6.7: Operator der Kommunikations-Software-Komponente *Abovertragsabschluss-Dialog*

Operatoren *Kreditorrechnungsbearbeitung*	**Operatoren *Kreditorrechnungsbearbeitung-Dialog***
prüfenVertragsdaten	*ausgebenVertragsdaten*
	eingebenVertragsprüfungsergebnis
prüfenRechnung	*ausgebenRechnungVertrag*
	eingebenRechnungsprüfungsergebnis
anweisenZahlung	*eingebenZahlungsanweisung*

Tabelle 6.8: Operatoren der Kommunikations-Software-Komponente Kreditorrechnungsbearbeitung-Dialog

Spezifikation von Datenhaltungs-Software-Komponenten auf Basis der Entitäts-Software-Komponenten

Schließlich sind für die Entitäts-Software-Komponenten des gewählten Ausschnitts zugehörige Datenhaltungs-Software-Komponenten zu spezifizieren. Dabei wird in der vorliegenden Arbeit davon ausgegangen, dass kein bestehendes Datenbanksystem zu berücksichtigen ist. In diesem Fall ist eine 1:1-Abbildung zwischen Entitäts-Software-Komponenten und Datenhaltungs-Software-Komponenten möglich. Das bedeutet, dass die Datenhaltungs-Software-Komponenten dieses Ausschnitts im Wesentlichen den Entitäts-Software-Komponenten entsprechen, denen sie zugeordnet sind.

Resultierendes komponentenbasiertes Anwendungssystems aus der Außenperspektive und aus der Innenperspektive

Abbildung A.V im Anhang zeigt die resultierende Außenperspektive des zugrunde gelegten Ausschnitts anhand eines Software-Komponenten-Diagramms. Es basiert auf dem in Abschnitt 5.2.2.3 vorgestellten Repräsentations-Metamodell und beschreibt insbesondere die Beziehungen zwischen den spezifizierten Software-Komponenten und deren Schnittstellen. Neben den Vorgangs- und Entitäts-Software-Komponenten für die Anwendungsfunktionen enthält es für jede Vorgangs-Software-Komponenten eine Kommunikations-Software-Komponenten für die Mensch-Computer-Kommunikation. Aus Übersichtlichkeitsgründen wurden die Datenhaltungs-Software-Komponenten, die den Entitäts-Software-Komponenten zugeordnet sind, nicht mit in das Diagramm aufgenommen. Sie stimmen, wie erwähnt, im Wesentlichen mit den jeweiligen Entitäts-Software-Komponenten überein. Außerdem enthalten die Entitäts- und die Kommunikations-Software-Komponenten keine zusätzlichen OCL-Ausdrücke zur Beschreibung ihres Verhaltens, da deren Operatorensignaturen dafür bereits ausreichen.

In Abbildung A.VI im Anhang wird die zugehörige Innenperspektive mittels eines Klassendiagramms beschrieben. Das entsprechende Repräsentations-Metamodell wurde in Abschnitt 5.3.2.3 vorgestellt. Das Klassendiagramm spezifiziert wiederum hauptsächlich die Beziehungen zwischen den Objekttypen und deren Schnittstellen. Da im Software-Komponenten-Diagramm auf die Darstellung der Datenhaltungs-Software-Komponenten aus den genannten Gründen verzichtet wurde, enthält das Klassendiagramm ebenfalls keine Datenhaltungs-Objekttypen.

Die folgenden Tabellen verdeutlichen noch einmal exemplarisch die dem Software-Architekturmodell entsprechende Gestaltung der Software-Komponenten im softwaretechnischen Entwurf der Fallstudie. Dabei werden die zur Spezifikation einer Software-Komponente bereitgestellten Metaobjekte der Kern-Metamodelle sowie der Repräsentations-Metamodelle der statischen Struktur und des statischen Verhaltens auf beiden Modellebenen den entsprechenden Struktur- und Verhaltensmerkmalen der Vorgangs-Software-Komponente *Abovertragsabschluss* gegenübergestellt.

Außenperspektive-Modellebene		
Metaobjekte des Kern-Metamodells	**Metaobjekte des Repräsentations-Metamodells der statischen Struktur und des statischen Verhaltens**	**Merkmale der Vorgangs-Software-Komponente *Abovertragsabschluss***
Vorgangs-Software-Komponente	Subsystem mit Stereotyp „<<Vorgang>>“	Subsystem, *zugehöriger Stereotyp aus Übersichtlichkeitsgründen nicht dargestellt*
Software-Komponentenname	Subsystemname	„Abovertragsabschluss“
fachliche Schnittstelle	Schnittstellenbereich mit Stereotyp „<<Funktion>>“	Schnittstellenbereich mit Stereotyp „<<Funktion>>“
Verwaltungsschnittstelle	Schnittstellenbereich mit Stereotyp „<<Verwaltung>>“	Schnittstellenbereich mit Stereotyp „<<Verwaltung>>“
fachliche Kurzbeschreibung	Anmerkung	*Anmerkung aus Übersichtlichkeitsgründen nicht dargestellt*
Komponentenoperatorsignatur	Operatorsignatur	Abovertragsabschluss erstellen(); void zerstoeren(); Abovertrag erstellenAbovertrag(in sequence miaIds, in sequence mianIds) boolean annehmenAbovertrag(in Abovertrag einAVertrag)

Außenperspektive-Modellebene		
Metaobjekte des Kern-Metamodells	**Metaobjekte des Repräsentations-Metamodells der statischen Struktur und des statischen Verhaltens**	**Merkmale der Vorgangs-Software-Komponente *Abovertragsabschluss***
Ausnahmedeklaration	Ausnahme	raises (ErstellungGescheitert) raises (AbovertragFehlerhaft)
Vorbedingung	Vorbedingung	pre: miaIds -> forAll (miaId|Marktinformationsangebot -> exists (mia|mia.mia_id=miaId)) and mianIds -> forAll (mianId|Marktinformationsanforderung -> exists (mian|mian.mian_id=mianId)) pre: Abovertrag -> exists (av|av.av_id=einAVertrag.av_id)
Nachbedingung	Nachbedingung	post: Abovertrag -> exists (av|av.oclIsNew() and Marktinformationsangebot -> exists (mia|mia.mia_id=av.marktinfoangebot.mia_id) and Marktinformationsanforderung -> exists (mian|mian.mian_id=av.marktinfoanforderung.mian_id)) post: result = Abovertrag -> exists (av|av.av_id=einAVertrag.av_id and av.datum<>"" and av.zeitraum>0 and av.kFrist>0 and av.unterzeichner<>"" and Marktinformationsangebot -> exists (mia|mia.mia_id=av.marktinfoangebot.mia_id) and Marktinformationsanforderung -> exists (mian|mian.mian_id=av.marktinfoanforderung.mian_id) and av.rabatt>=0 and

Außenperspektive-Modellebene		
Metaobjekte des Kern-Metamodells	**Metaobjekte des Repräsentations-Metamodells der statischen Struktur und des statischen Verhaltens**	**Merkmale der Vorgangs-Software-Komponente *Abovertragsabschluss***
		av.preis>=0 and av.erhebungsmenge>0 and av.pruefstatus=false)
funktionale Invariante	Funktionale Invariante	inv: Marktinformationsangebot -> notEmpty() and Marktinformationsanforderung -> notEmpty()
Deklaration nicht-funktionaler Eigenschaften	Schnittstellenbereich mit Stereotyp „<<Nicht-funktionale Eigenschaften>>“	Schnittstellenbereich mit Stereotyp „<<Nicht-funktionale Eigenschaften>>“
nicht-funktionale Eigenschaft	Nicht-funktionale Eigenschaft	transactionManagedOperations=operations() transactionManagedType=REQUIRED

Tabelle 6.9: Gegenüberstellung der Kern- und Repräsentations-Metaobjekte auf der Außenperspektive-Modellebene und der Merkmale der Vorgangs-Software-Komponente *Abovertragsabschluss*

Innenperspektive-Modellebene		
Metaobjekte des Kern-Metamodells	**Metaobjekte des Repräsentations-Metamodells der statischen Struktur und des statischen Verhaltens**	**Merkmale der Vorgangs-Software-Komponente *Abovertragsabschluss***
Vorgangsklasse	Klasse mit Stereotyp „<<Vorgang>>“	Klassen, *zugehörige Stereotypen aus Übersichtlichkeitsgründen nicht dargestellt*
Klassenname	Klassenname	„Abovertrag>“ „>Abovertrag“
Operatorsignatur	Operator	*Operatoren aus Übersichtlichkeitsgründen nicht dargestellt*
Operatorimplementierung	-	-
Attribut	Attribut	*Attribute aus Übersichtlichkeitsgründen nicht dargestellt*

Innenperspektive-Modellebene		
Metaobjekte des Kern-Metamodells	**Metaobjekte des Repräsentations-Metamodells der statischen Struktur und des statischen Verhaltens**	**Merkmale der Vorgangs-Software-Komponente *Abovertragsabschluss***
Intra-Komponenten-Beziehung	Klassenbeziehung, ausgenommen Abhängigkeits- und Realisierungsbeziehung	gerichtete Assoziation zwischen den beteiligten Klassen
Assoziationsname	Beziehungsname	*Beziehungsnamen nicht benötigt*
Nachricht	-	-
Kardinalität 1	Multiplizität 1	*Multiplizitäten aus Übersichtlichkeitsgründen nicht dargestellt*
Kardinalität 2	Multiplizität 2	*Multiplizitäten aus Übersichtlichkeitsgründen nicht dargestellt*

Tabelle 6.10: Gegenüberstellung der Kern- und Repräsentations-Metaobjekte auf der Innenperspektive-Modellebene und der Merkmale der Vorgangs-Software-Komponente *Abovertragsabschluss*

Die Spezifikation der korrespondierenden Verhaltenssichten der beiden Modellebenen folgen im weiteren Verlauf des Abschnitts.

Untersuchung der abgeleiteten Software-Komponenten hinsichtlich ihrer Wiederverwendbarkeit

Zunächst sind jedoch die aus dem KOS und den VOS abgeleiteten Software-Komponenten hinsichtlich ihrer Wiederverwendbarkeit in unterschiedlichen betrieblichen Anwendungssystemen zu untersuchen und gegebenenfalls weiter aufzubereiten. Die Untersuchung erfolgt anhand der in Abschnitt 3.3 formulierten Forderungen an den softwaretechnischen Entwurf wiederverwendbarer und verteilbarer Anwendungssysteme. Tabelle 6.11 fasst das Untersuchungsergebnis übersichtlich zusammen.

	Forderungserfüllung			
Forderung	Abovertrags-abschluss	Kreditor-rechnungs-bearbeitung	Abovertrag	Kreditor-rechnung
hohes Abstraktionsniveau	+	+	+	+
Allgemeingültigkeit	-	-	-	-

	Forderungserfüllung			
Forderung	Abovertrags-abschluss	Kreditor-rechnungs-bearbeitung	Abovertrag	Kreditor-rechnung
hierarchische Strukturierung	+	+	+	+
Modularität	+	+	+	+
Verständlichkeit	+	+	+	+
Nachvollzieh-barkeit	+	+	+	+
standardisierte und formalisier-te Beschreibung	+	+	+	+
Anpassbarkeit	+	+	+	+
Erweiterbarkeit	+	+	+	+
Interoperabilität	+	+	+	+
Portierbarkeit	+	+	+	+
Qualität	+	+	+	+

+ = erfüllt - nicht erfüllt

Tabelle 6.11: Untersuchung der abgeleiteten Software-Komponenten hinsichtlich ihrer Wiederverwendbarkeit

Die Forderungen nach einem hohen Abstraktionsniveau, einer hierarchischen Strukturierung sowie nach Modularität, Verständlichkeit, Nachvollziehbarkeit, Anpassbarkeit und Erweiterbarkeit erfüllen die Software-Komponenten bereits aufgrund ihrer Gestaltung gemäß dem präskriptiven Grundkonzept der Komponentenorientierung. Da sie außerdem mithilfe des komponentenbasierten Software-Architekturmodells entworfen wurden, erfüllen sie auch die Forderungen nach einer standardisierten und formalisierten Beschreibung sowie nach Interoperabilität, Portierbarkeit und Qualität. Allerdings können sie aufgrund ihrer Domänenabhängigkeit die Forderung nach Allgemeingültigkeit noch nicht erfüllen und müssen deshalb diesbezüglich überarbeitet werden.

Als theoretische Grundlage zur Überführung domänenabhängiger in domänenunabhängige Software-Komponenten können u. a. geeignete und bewährte Referenzmodelle herangezogen werden. Im vorliegenden Fall wurde beispielsweise das Referenzmodell von MERTENS [Mert00] als Grundlage gewählt. Bezüglich der Aus-

wahl und der Verwendung von Referenzmodellen, sowie hinsichtlich eventueller Alternativen besteht jedoch noch weiterer Forschungsbedarf.

Überarbeitung der abgeleiteten Software-Komponenten hinsichtlich ihrer Allgemeingültigkeit

Im Folgenden soll exemplarisch die *Kreditorrechnungsbearbeitung*-Komponente für die Wiederverwendung in unterschiedlichen betrieblichen Domänen vorbereitet werden. Dies bestimmt daraufhin auch die notwendigen Maßnahmen zur Überarbeitung der zugehörigen Entitäts-, Kommunikations- und Datenhaltungs-Software-Komponenten.

In der *Kreditorrechnungsbearbeitung*-Komponente sind die Vorgangsobjekttypen *Rechnung (Abo)>*, *>Rechnung (Abo)*, *Abozahlungsauftrag>* und *>Abozahlungsauftrag* für eine Wiederverwendung nicht allgemein genug, da sie auf unternehmensspezifischen Aufgaben basieren. Diese Aufgaben beschreiben die Erstellung, die Annahme und die Vorbereitung zur Zahlung von Rechnungen, die sich speziell auf einen Abonnement-Vertrag für Marktinformationsdaten beziehen. Deshalb werden die genannten Vorgangsobjekttypen in eine **Adapter-Software-Komponente** mit dem fachlichen Namen *Kreditorrechnungsbearbeitung-Adapter* ausgelagert und diese mit der *Kreditorrechnungsbearbeitung*-Komponente in Beziehung gesetzt. Außerdem sind die Namen, die Operatoren und die Operatorensignaturen der in der *Kreditorrechnungsbearbeitung*-Komponente verbleibenden Vorgangsobjekttypen im Hinblick auf die geforderte Allgemeingültigkeit zu überarbeiten. Zum einen ist es notwendig, einer Kreditorrechnung einen allgemeinen Liefervertrag anstelle eines Abonnement-Vertrags zugrunde zu legen. Zum anderen sollte der zum Vorgangsobjekttyp *Rechnungsdaten (Abo)>* gehörende Operator *begleichenRechnung* für die ordnungsgemäße Entgegennahme von Kreditorrechnungen zuständig sein. Zu diesem Zweck wird er in *empfangenRechnung* umbenannt.

Damit diese überarbeitete *Kreditorrechnungsbearbeitung*-Komponente weiterhin in der ursprünglichen Anwendungssystementwicklung verwendet werden kann, sind der *Kreditorrechnungsbearbeitung-Adapter*-Komponente schließlich noch weitere Operatoren hinzuzufügen. Sie sorgen für die Anpassung der überarbeiteten *Kreditorrechnungsbearbeitung*-Komponente an die übrigen Vorgangs-Software-Komponenten. Dazu gehört zum einen der Operator *anpassenDaten*, der die Entitäts-Software-Komponente *Abovertrag* zur allgemeineren Entitäts-Software-Komponente *Liefervertrag* konvertiert, und zum anderen der Operator *begleichenRechnung*, der den Operator *empfangenRechnung* der *Kreditorrechnungsbearbeitung*-Komponente aufruft. Dieser ist, wie erwähnt, für die Entgegennahme von Kreditorrechnungen bestimmt.

Die Überarbeitung der Vorgangs-Software-Komponente *Kreditorrechnungsbearbeitung* wirkt sich auch auf die zugeordnete Entitäts-Software-Komponente *Kreditor-*

rechnung aus. Demnach ist es erforderlich, dass sie sich nicht mehr auf die Entitäts-Software-Komponente *Abovertrag*, sondern auf die allgemeinere Entitäts-Software-Komponente *Liefervertrag* bezieht. Diese stimmt mit der Entitäts-Software-Komponente *Abovertrag* bis auf die abovertragsspezifischen Attribute überein. Beide Entitäts-Software-Komponenten werden der Vorgangs-Software-Komponente *Kreditorrechnungsbearbeitung-Adapter* zugeordnet, die, wie erwähnt, unter anderem für die Konvertierung der *Abovertrag*-Komponente in die *Liefervertrag*-Komponente zuständig ist.

Außerdem folgt aus dem Auslagern des Vorgangsobjekttyps *>Abozahlungsauftrag* in die *Kreditorrechnungsbearbeitung-Adapter*-Komponente, dass eine neue Kommunikations-Software-Komponente mit dem Namen *Kreditorrechnungsbearbeitung-Adapter-Dialog* erstellt und der *Kreditorrechnungsbearbeitung-Adapter*-Komponente zugeordnet werden muss. Sie weist die Operatoren *ausgebenVertragsdaten* und *eingebenVertragsprüfungsergebnis* auf.

Schließlich erfolgt eine Anpassung der Datenhaltungs-Software-Komponenten an die Änderungen der Entitäts-Software-Komponenten. Wie bereits erwähnt, sind in der vorliegenden Arbeit 1:1-Abbildungen zwischen den Entitäts-Software-Komponenten und den zugehörigen Datenhaltungs-Software-Komponenten vorgesehen. Somit wird die Datenhaltungs-Software-Komponente, die der Entitäts-Software-Komponente *Kreditorrechnung* zugeordnet ist, entsprechend geändert und für die zusätzliche Entitäts-Software-Komponente *Liefervertrag* eine korrespondierende Datenhaltungs-Software-Komponente erstellt. Dies bedingt auch eine Anpassung der Existenzabhängigkeitsbeziehungen zwischen den Datenhaltungs-Software-Komponenten.

Überarbeitetes komponentenbasiertes Anwendungssystems aus der Außenperspektive und aus der Innenperspektive

Abbildung A.VII im Anhang enthält ein Software-Komponenten-Diagramm, das die resultierenden Außenperspektiven der überarbeiteten Software-Komponenten beschreibt. Aus den bereits genannten Gründen wird auf die Darstellung der Datenhaltungs-Software-Komponenten und auf die Beschreibung des Verhaltens der Entitäts- und der Kommunikations-Software-Komponenten mittels OCL-Ausdrücken verzichtet. Abbildung A.VIII im Anhang zeigt das Verhalten der Instanzen dieser Software-Komponenten während der Ausführung anhand eines Sequenzdiagramms auf. Es basiert auf dem in Abschnitt 5.2.2.3 eingeführten Repräsentations-Metamodell für die Beschreibung des Verhaltens von Software-Komponenten aus der Außenperspektive.

Die Innenperspektiven der überarbeiteten Software-Komponenten werden in Abbildung A.IX anhand eines Klassendiagramms dargestellt. Aufgrund der vermiedenen Darstellung von Datenhaltungs-Software-Komponenten im Software-Komponenten-

Diagramm weist das Klassendiagramm auch keine Datenhaltungs-Objekttypen auf. Das Verhalten der Objekte dieser Objekttypen wird, wie in Abschnitt 5.3.2.3 erläutert, anhand von Sequenzdiagrammen beschrieben. Abbildung A.10 zeigt ein exemplarisches Sequenzdiagramm, das die Interaktion der zur *Kreditorrechnungsbearbeitung-*, *Kreditorrechnung-*, *Lieferauftrag-* und *Kreditorrechnungsbearbeitung-Dialog-*Komponente gehörenden Objekte darstellt.

6.1.5 Implementierung

Die Ergebnisse des im vorangegangenen Abschnitt durchgeführten softwaretechnischen Entwurfs der Software-Komponenten und der zugehörigen Beziehungen sind nun Grundlage für deren Implementierung. Wie erwähnt, entspricht die Implementierung einer tatsächlichen Realisierung des modellierten Anwendungssystems und ist folglich kein Bestandteil des vorgestellten Software-Architekturmodells. In der vorliegenden Arbeit dient sie lediglich dazu, die praktische Anwendbarkeit des Software-Architekturmodells über den softwaretechnischen Entwurf hinaus nachzuweisen und ist daher vergleichsweise kurz gefasst.

Gewählte Entwicklungs- und Ausführungsplattform

Die Implementierung erfolgt auf Basis der Entwicklungs- und Ausführungsplattform **Java 2 Enterprise Edition (J2EE)**, die unter anderem die bereits angesprochenen Komponentenkonzepte **Enterprise JavaBeans (EJB)** und **JavaBeans (JB)** enthält [SUN02a; SUN97]. EJB ist derzeit das einzige in sich abgeschlossene Konzept für Anwendungs- und Datenhaltungs-Software-Komponenten, das von mehr als einem Hersteller unterstützt und weiterentwickelt wird [VöSW03, 42 f; Fisc02, 52 f].

Sowohl die Komponentenkonzepte als auch die Entwicklungs- und Ausführungsplattform werden ursprünglich als Spezifikationen veröffentlicht [SUN02a; SUN02b]. Sie stellen verpflichtende Vorgaben für Hersteller von J2EE-konformen Entwicklungs- und Ausführungsplattformen sowie EJB- bzw. JB-konformen Programmiersprachen dar. Ziel dieser Spezifikationen ist die herstellerunabhängige Standardisierung von Entwicklungs- und Ausführungsplattformen und Programmiersprachen für die komponentenorientierte Entwicklung von Anwendungssystemen [GrTh00, 209 f]. Allerdings werden die Spezifikationen nicht von einer unabhängigen Standardisierungsorganisation, sondern von einem Unternehmen, SUN Microsystems, herausgegeben und betreut. Dies schränkt die Herstellerunabhängigkeit dieser Spezifikationen wiederum deutlich ein. Im Folgenden werden nur die für die Implementierung des Fallstudienausschnitts notwendigen Merkmale der Entwicklungs- und Ausführungsplattform J2EE und der Programmiersprachen EJB und JB kurz erläutert. Für weitergehende Ausführungen zu J2EE, EJB und JB kann auf die inzwischen ausreichende

Anzahl von Veröffentlichungen zu diesem Thema verwiesen werden (siehe z. B. [SUN02a; SUN97; Mons01; DePe02; GrTh00; Blev01]).

Enterprise-JavaBeans-Spezifikation

Die EJB-Spezifikation stellt im Wesentlichen zwei Komponentenarten zur Verfügung, die **Entity Beans** und die **Session Beans**. Sie unterscheiden sich hauptsächlich darin, dass Instanzen der Entity Beans im Gegensatz zu Instanzen der Session Beans ihren Zustand persistent speichern können [SUN02a, 44; Mons01, 26; DePe02, 42 f; Blev01, 592 ff; GrTh00, 217 f]. „Der relevante Unterschied bei Enterprise JavaBeans ist der persistente Zustand einer Entity Bean; eine Session Bean modelliert Interaktionen, hat aber selbst keinen persistenten Zustand“ [Mons01, 26]. Deswegen sind Entity Beans für die Implementierung von Entitäts-Software-Komponenten und den zugehörigen Datenhaltungs-Software-Komponenten geeignet und Session Beans für die Realisierung von Vorgangs-Software-Komponenten. Entity Beans und Session Beans werden unter dem Begriff „**Enterprise Beans**“ zusammengefasst [SUN02a, 43; DePe02, 42; Blev01, 592; Mons01, 26; GrTh00, 217].

JavaBeans-Spezifikation

Im Gegensatz zum Komponentenkonzept der Enterprise JavaBeans, das insbesondere den Einsatz von Enterprise Beans als Anwendungs-Software-Komponenten vorsieht, ist das der JavaBeans allgemeiner formuliert: „The goal of the JavaBeans APIs is to define a *software component model* for Java, so that third party ISVs [Independent Software Vendors; Anm. d. Verf.] can create and ship Java components that can be composed together into applications by end users“ [SUN97, 7; Hervorhebung im Original]. Dies erfasst einen Bereich, der auf der einen Seite von einfachen grafischen Benutzerschnittstellenelementen und auf der anderen Seite von komplexen Software-Teilsystemen begrenzt wird [SUN97, 7]. Demnach eignen sich JavaBeans unter anderem auch zur Implementierung von Kommunikations-Software-Komponenten für die Mensch-Computer-Kommunikation.

Programmierschnittstellen der gewählten Komponentenkonzepte

Die Programmierschnittstellen beider Komponentenkonzepte basieren auf dem Grundkonzept der Objektorientierung. Folglich werden Enterprise Beans und JavaBeans in Form von objektorientierten Klassen implementiert und können somit auch in Vererbungsbeziehungen zueinander stehen. Das bedeutet, dass mit Enterprise Beans und JavaBeans keine objektorientierten Klassen, die einen gemeinsamen fachlichen Belang abbilden, gekapselt und damit besser wiederverwendet werden können, sondern dass sie selbst objektorientierte Klassen darstellen. In Bezug auf die Ergebnisse der vorangegangenen Kapitel ist dies ein bedeutender Nachteil dieser Programmierschnittstellen und damit auch der ihnen zugrunde liegenden Kom-

ponentenkonzepte. Zudem besteht darin der wesentliche Unterschied zwischen dem in Abschnitt 4 formulierten Grundkonzept der Komponentenorientierung und den beiden Komponentenkonzepten.

Das Komponentenkonzept und die Programmierschnittstelle der Enterprise JavaBeans unterstützen eine getrennte Realisierung der Schnittstelle und der zugehörigen Implementierung einer Enterprise Bean [SUN02a, 46; Mons01, 27; DePe02, 45; Blev01, 596 f; GrTh00, 213 f]. Außerdem wird zwischen einer Verwaltungsschnittstelle (**Home Interface**) und einer fachlichen Schnittstelle (**Remote Interface**) einer Enterprise Bean unterschieden [SUN02a, 46; Mons01, 27; DePe02, 45; Blev01, 599 ff; GrTh00, 213 ff]. Darüber hinaus sieht das Komponentenkonzept der Enterprise JavaBeans die Deklaration von nicht-funktionalen Eigenschaften und Eigenschaftswerten einer Enterprise Bean anhand von so genannten ***Deployment Descriptors*** vor [SUN02a, 36; Mons01, 33 ff; DePe02, 51; Blev01, 601 f]. Diese werden in XML (Extensible Markup Language) formuliert und zum Zeitpunkt der Installation der Enterprise Beans dem Ausführungsdienst, *Container* genannt, übergeben [Mons01, 37; DePe02, 51; Blev01, 601 f]. In den folgenden Ausführungen zur Implementierung des Fallstudienausschnitts werden aus Gründen der Übersichtlichkeit und Verständlichkeit nur die Verwaltungsschnittstellen und die fachlichen Schnittstellen sowie die *Deployment Descriptors* der verwendeten Enterprise Beans gezeigt. Die Beschreibung der den Session Beans zugeordneten JavaBeans erfolgt dagegen anhand der jeweils realisierten grafischen Benutzerschnittstelle.

Implementierung auf Basis der gewählten Entwicklungs- und Ausführungsplattform

Zunächst wird die Implementierung der wiederzuverwendenden Software-Komponenten *Kreditorrechnungsbearbeitung*, *Kreditorrechnungsbearbeitungs-Dialog*, *Kreditorrechnung* und *Liefervertrag* vorgestellt. Ausgangspunkt ist die Vorgangs-Software-Komponente *Kreditorrechnungsbearbeitung*, die anhand einer Session Bean implementiert wird. Abbildung 6.7 zeigt die zugehörigen Schnittstellen.

```
/**
 * Home interface for Enterprise Bean: Kreditorrechnungsbearbeitung
 */
public interface KreditorrechnungsbearbeitungHome extends javax.ejb.EJBHome
{

      public Kreditorrechnungsbearbeitung create() throws
      javax.ejb.CreateException, java.rmi.RemoteException;

}

import reuse.*;
/**
 * Remote interface for Enterprise Bean: Kreditorrechnungsbearbeitung
 */
public interface Kreditorrechnungsbearbeitung extends javax.ejb.EJBObject {

      public boolean empfangenRechnung(Kreditorrechnung kr, ICallback
      callback) throws java.rmi.RemoteException;

      public boolean pruefenRechnung(Kreditorrechnung kr, Liefervertrag lv)
      throws java.rmi.RemoteException;

      public boolean freigebenZahlung(Kreditorrechnung kr) throws
      java.rmi.RemoteException;

      public boolean anweisenZahlung(Kreditorrechnung kr) throws
      java.rmi.RemoteException;

}
```

Abbildung 6.7: Schnittstellen der Session Bean *Kreditorrechnungsbearbeitung*

Die Verwaltungsschnittstelle, *KreditorrechnungsbearbeitungHome* enthält im Wesentlichen den Konstruktor, *create()*, der ein Objekt des Typs der fachlichen Schnittstelle zurückgibt. In der fachlichen Schnittstelle, *Kreditorrechnungsbearbeitung*, sind insbesondere die fachlichen Operatorensignaturen aufgeführt, die im softwaretechnischen Entwurf spezifiziert wurden. Zur Interaktion mit der zugeordneten Kommunikations-Software-Komponente *Kreditorrechnungsbearbeitungs-Dialog*, die mittels einer JavaBean realisiert wird, enthält die Signatur des Operators *empfangenRechnung* einen zusätzlichen Parameter vom Typ *ICallback*, der zur Ausführungszeit eine Instanzreferenz auf die JavaBean aufweist. Folglich wird einer Instanz der Session Bean, die eine *Kreditorrechnungsbearbeitung*-Komponente realisiert, mit dem Aufruf des *empfangenRechnung*-Operators eine Instanzreferenz der JavaBean übergeben, die die *Kreditorrechnungsbearbeitungs-Dialog*-Komponente realisiert. Anhand dieser Instanzreferenz kann die *Kreditorrechnungsbearbeitung*-Session-Bean schließlich die Operatoren der *Kreditorrechnungsbearbeitungs-Dialog*-JavaBean aufrufen.

Die Entitäts-Software-Komponenten *Kreditorrechnung* und *Liefervertrag*, die der Vorgangs-Software-Komponente *Kreditorrechnungsbearbeitung* zugeordnet sind, werden anhand von Entity Beans realisiert. In Abbildung 6.8 sind die Schnittstellen der

Kreditorrechnung-Entity-Bean und in Abbildung 6.9 die der *Liefervertrag*-Entity-Bean dargestellt.

```
/**
 * Home interface for Enterprise Bean: Kreditorrechnung
 */
public interface KreditorrechnungHome extends javax.ejb.EJBHome {

      public Kreditorrechnung create(int KR_ID, Liefervertrag lief) throws
      javax.ejb.CreateException, java.rmi.RemoteException;

      public Kreditorrechnung findByPrimaryKey(KreditorrechnungKey
      primaryKey) throws javax.ejb.FinderException,
      java.rmi.RemoteException;

}

/**
 * Remote interface for Enterprise Bean: Kreditorrechnung
 */
public interface Kreditorrechnung extends javax.ejb.EJBObject {

      public int getkr_ID() throws java.rmi.RemoteException;

      public double getBetrag() throws java.rmi.RemoteException;

      public void setBetrag(double newBetrag) throws
      java.rmi.RemoteException;

      public java.lang.String getFaelligkeit() throws
      java.rmi.RemoteException;

      public void setFaelligkeit(java.lang.String newFaelligkeit) throws
      java.rmi.RemoteException;

      public int getKtoNr() throws java.rmi.RemoteException;

      public void setKtoNr(int newKtoNr) throws java.rmi.RemoteException;

      public int getBlz() throws java.rmi.RemoteException;

      public void setBlz(int newBlz) throws java.rmi.RemoteException;

      public LiefervertragKey getLiefervertragKey() throws
      java.rmi.RemoteException;

      public Liefervertrag getLiefervertrag() throws
      java.rmi.RemoteException, javax.ejb.FinderException;

      public Rechnungswesen getRechnungswesen() throws
      java.rmi.RemoteException;

      public java.lang.String getStatus() throws java.rmi.RemoteException;

      public void setStatus(java.lang.String newStatus) throws
      java.rmi.RemoteException;

}
```

Abbildung 6.8: Schnittstellen der Entity Bean *Kreditorrechnung*

```
/**
 * Home interface for Enterprise Bean: Liefervertrag
 */
public interface LiefervertragHome extends javax.ejb.EJBHome {

      public Liefervertrag create(int lv_ID) throws
      javax.ejb.CreateException, java.rmi.RemoteException;

      public Liefervertrag findByPrimaryKey(LiefervertragKey primaryKey)
      throws javax.ejb.FinderException, java.rmi.RemoteException;

}


/**
 * Remote interface for Enterprise Bean: Liefervertrag
 */
public interface Liefervertrag extends javax.ejb.EJBObject {

      public int getlv_ID() throws java.rmi.RemoteException;

      public java.lang.String getDatum() throws java.rmi.RemoteException;

      public void setDatum(java.lang.String newDatum) throws
      java.rmi.RemoteException;

      public java.lang.String getK_frist() throws java.rmi.RemoteException;

      public void setK_frist(java.lang.String newK_frist) throws
      java.rmi.RemoteException;

      public double getSumme() throws java.rmi.RemoteException;

      public void setSumme(double newSumme) throws
      java.rmi.RemoteException;

      public double getRabatt() throws java.rmi.RemoteException;

      public void setRabatt(double newRabatt) throws
      java.rmi.RemoteException;

      public boolean isPruefstatus() throws java.rmi.RemoteException;

      public void setPruefstatus(boolean newPruefstatus) throws
      java.rmi.RemoteException;

      public Angebot getAngebot() throws java.rmi.RemoteException;

      public Lieferant getLieferant() throws java.rmi.RemoteException;

}
```

Abbildung 6.9: Schnittstellen der Entity Bean Liefervertrag

Die Verwaltungsschnittstellen dieser Entity Beans weisen, dem softwaretechnischen Entwurfsergebnis entsprechend, jeweils eine Konstruktorsignatur und eine Operatorsignatur zum Auffinden einer bestimmten Instanz auf. In den fachlichen Schnittstellen der Entity Beans sind die Operatorensignaturen zum Lesen und zum Schreiben der jeweiligen Attributwerte aufgeführt.

Abbildung 6.10 zeigt die grafische Benutzerschnittstelle der bereits erwähnten *Kreditorrechnungsbearbeitungs-Dialog*-JavaBean. Jedes Element der grafischen Benutzerschnittstelle entspricht einer Klasse, die in dieser JavaBean gekapselt ist und aus der zur Ausführungszeit ein konkretes Objekt erzeugt wird. Anhand dieser JavaBean wird exemplarisch der Zusammenhang zwischen den Operatoren einer Kommunikations-Software-Komponente und den in einer JavaBean enthaltenen Benutzerschnittstellen-Klassen erläutert.

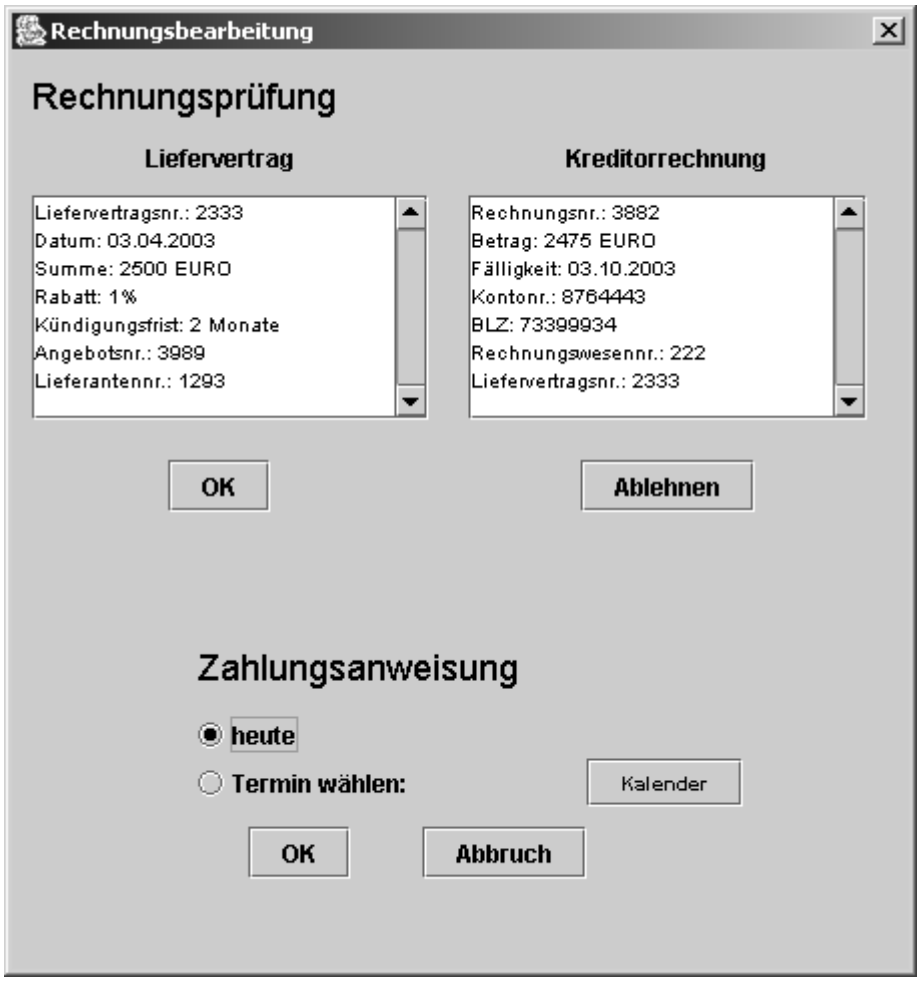


Abbildung 6.10: Grafische Benutzerschnittstelle der JavaBean *Kreditorrechnungsbearbeitungs-Dialog*

Die Objekte der Textfeld-Klassen unter den Beschriftungs-Klassen *Liefervertrag* und *Kreditorrechnung* werden durch den Aufruf des Operators *ausgebenRechnung-Vertrag* gefüllt. Der darauffolgende Aufruf des Operators *eingebenRechnungsprüfungsergebnis* aktiviert die beiden Knopf-Objekte *OK* und *Ablehnen*, durch die ein Benutzer das Ergebnis dieser Operatorausführung bestimmen kann. Schließlich bewirkt der Aufruf des Operators *eingebenZahlungsanweisung* die Aktivierung des Beschriftungs-Objekts *Zahlungsanweisung*, der Auswahlschalter-Objekte *heute* und *Termin wählen* sowie der Knopf-Objekte *Kalender*, *OK* und *Abbruch*. Mit den genann-

ten Auswahlschalter-Objekten und Knopf-Objekten kann ein Benutzer wiederum den Rückgabewert dieses Operators festlegen.

Die bisher erläuterten Enterprise Beans und JavaBeans stellten zunächst die Implementierungen der wiederzuverwendenden Software-Komponenten dar. Da die Implementierungen der übrigen Software-Komponenten analog dazu sind, wird im Folgenden auf eine ausführliche Erläuterung dieser Implementierungen verzichtet und nur auf Besonderheiten, die dabei zu beachten sind, hingewiesen.

In Abbildung 6.11 und Abbildung 6.12 sind die Schnittstellen der Session Beans *Kreditorrechnungsbearbeitung-Adapter* und *Abovertragsabschluss* dargestellt.

```
/**
 * Home interface for Enterprise Bean: KreditorrechnungsbearbeitungAdapter
 */
public interface KreditorrechnungsbearbeitungAdapterHome extends
javax.ejb.EJBHome {

      public KreditorrechnungsbearbeitungAdapter create() throws
      javax.ejb.CreateException, java.rmi.RemoteException;

}

import java.rmi.RemoteException;
import javax.ejb.EJBObject;
import reuse.ICallback;
import fallstudie.*;
import fallstudie.AbovertragDO;

/**
 * Remote interface for Enterprise Bean:
KreditorrechnungsbearbeitungAdapter
 */
public interface KreditorrechnungsbearbeitungAdapter extends
javax.ejb.EJBObject {

      public Kreditorrechnung erstellenRechnung(fallstudie.ICallbackF1
      callback, Abovertrag av) throws java.rmi.RemoteException;

      public boolean annehmenRechnung(Kreditorrechnung kred) throws
      java.rmi.RemoteException;

      public boolean vorbereitenRechnungsbegleichung(Kreditorrechnung kred)
      throws java.rmi.RemoteException;

      public boolean pruefenVertragsdaten(Kreditorrechnung kred) throws
      java.rmi.RemoteException;

      public boolean begleichenRechung(Kreditorrechnung kred, ICallback
      callback) throws java.rmi.RemoteException;

}
```

Abbildung 6.11: Schnittstellen der Session Bean *Kreditorrechnungsbearbeitung-Adapter*

```
/**
 * Home interface for Enterprise Bean: Abovertragsabschluss
 */
public interface AbovertragsabschlussHome extends javax.ejb.EJBHome {

      public Abovertragsabschluss create() throws
      javax.ejb.CreateException, java.rmi.RemoteException;

}


import fallstudie.ICallbackF1;
/**
 * Remote interface for Enterprise Bean: Abovertragsabschluss
 */
public interface Abovertragsabschluss extends javax.ejb.EJBObject {

      public Abovertrag erstellenAbovertrag(java.util.Collection miaIds,
      java.util.Collection mianIds, ICallbackF1 callback) throws
      java.rmi.RemoteException;

      public boolean annehmenAbovertrag(Abovertrag av) throws
      java.rmi.RemoteException;

}
```

Abbildung 6.12: Schnittstellen der Session Bean *Abovertragsabschluss*

Die Verwaltungsschnittstellen beider Session Beans enthalten ausschließlich die obligatorischen Konstruktorsignaturen.

Der in der fachlichen Schnittstelle der *Kreditorrechnungsbearbeitung-Adapter*-Bean enthaltene Operator *erstellenRechnung* nimmt eine Referenz auf eine *Abovertrag*-Komponenten-Instanz entgegen und gibt eine *Kreditorrechnung*-Komponenten-Instanz zurück, die wiederum eine Referenz auf eine *Liefervertrag*-Komponenten-Instanz besitzt. Die notwendige Konvertierung der *Abovertrag*-Komponenten-Instanz in eine *Liefervertrag*-Komponenten-Instanz erfolgt, wie erwähnt, durch den nichtöffentlichen Operator *anpassenDaten*, der von dem Operator *erstellenRechnung* aufgerufen wird. Eine weitere Besonderheit ist der Parameter vom Typ *ICallback* in den Operatorensignaturen *erstellenRechnung* und *begleichenRechnung*. Analog zur Signatur des Operators *empfangenRechnung* der *Kreditorrechnungsbearbeitung*-Bean nimmt er zur Ausführungszeit eine Instanzreferenz auf eine JavaBean entgegen. In der erstgenannten Operatorsignatur handelt es sich um eine Instanzreferenz auf die JavaBean *Kreditorrechnungsbearbeitung-Adapter-Dialog*, die für den Aufruf der zugehörigen Operatoren benötigt wird. In der letztgenannten Operatorsignatur bezieht sich die Instanzreferenz auf die JavaBean *Kreditorrechnungsbearbeitung-Dialog*, die beim Aufruf des Operators *empfangenRechnung* der *Kreditorrechnungsbearbeitung*-Bean weitergegeben wird.

In der fachlichen Schnittstelle der *Abovertragsabschluss*-Bean besitzt die Operatorsignatur *erstellenAbovertrag* ebenfalls einen Parameter vom Typ *ICallback*. Dieser

wird, analog zu den bisherigen Verwendungen, für die Interaktion mit der zugeordneten JavaBean *Abovertragsabschluss-Dialog*, die die gleichnamige Kommunikations-Software-Komponente realisiert, eingesetzt. Die übrigen Operatorensignaturen dieser Schnittstelle entsprechen im Wesentlichen denen des softwaretechnischen Entwurfs der *Abovertragsabschluss*-Komponente.

Den Session Beans *Kreditorrechnungsbearbeitung-Adapter* und *Abovertragsabschluss* ist eine Entity Bean mit dem Namen *Abovertrag* zugeordnet, deren Schnittstellen in Abbildung 6.13 dargestellt sind. Sie realisiert die entsprechende Entitäts-Software-Komponente des softwaretechnischen Entwurfs. Die Verwaltungsschnittstelle der Entity Bean enthält wiederum eine Konstruktorsignatur und eine Operatorsignatur zum Auffinden einer bestimmten Instanz. In der zugehörigen fachlichen Schnittstelle sind, wie in den fachlichen Schnittstellen der bisher vorgestellten Entity Beans, Operatorensignaturen zum Lesen und zum Schreiben der jeweiligen Attributwerte vorhanden.

```
/**
 * Home interface for Enterprise Bean: Abovertrag
 */
public interface AbovertragHome extends javax.ejb.EJBHome {

      public Abovertrag create(int AV_ID) throws javax.ejb.CreateException,
      java.rmi.RemoteException;

      public Abovertrag findByPrimaryKey(AbovertragKey primaryKey) throws
      javax.ejb.FinderException, java.rmi.RemoteException;

}

/**
 * Remote interface for Enterprise Bean: Abovertrag
 */
public interface Abovertrag extends javax.ejb.EJBObject {

      public int getav_ID() throws java.rmi.RemoteException;

      public java.lang.String getDatum() throws java.rmi.RemoteException;

      public void setDatum(java.lang.String newDatum) throws
      java.rmi.RemoteException;

      public String getZeitraum() throws java.rmi.RemoteException;

      public void setZeitraum(String zeitraum) throws
      java.rmi.RemoteException;

      public int getErhebungsmenge() throws java.rmi.RemoteException;

      public void setErhebungsmenge(int newErhebungsmenge) throws
      java.rmi.RemoteException;

      public double getPreis() throws java.rmi.RemoteException;

      public void setPreis(double preis) throws java.rmi.RemoteException;
```

```
        public double getRabatt() throws java.rmi.RemoteException;

        public void setRabatt(double rabatt) throws java.rmi.RemoteException;

        public Marktinformationsangebot getMa_Angebot() throws
        java.rmi.RemoteException;

        public Marktinformationsanforderung getMa_Anforderung() throws
        java.rmi.RemoteException;

        public String getK_Frist() throws java.rmi.RemoteException;

        public void setK_Frist(String k_Frist) throws
        java.rmi.RemoteException;

        public java.lang.String getUnterzeichner() throws
        java.rmi.RemoteException;

        public void setUnterzeichner(java.lang.String newUnterzeichner)
        throws java.rmi.RemoteException;

        public boolean isPruefstatus() throws java.rmi.RemoteException;

        public void setPruefstatus(boolean newPruefstatus) throws
        java.rmi.RemoteException;
}
```

Abbildung 6.13: Schnittstellen der Entity Bean *Abovertrag*

Die grafischen Benutzerschnittstellen der bereits erwähnten JavaBeans *Abovertragsabschluss-Dialog* und *Kreditorrechnungsbearbeitung-Adapter-Dialog* sind in Abbildung 6.14 und Abbildung 6.15 dargestellt und bestehen jeweils aus Benutzerschnittstellen-Objekten, die mit den Operatoren der entsprechenden Kommunikations-Software-Komponente korrespondieren.

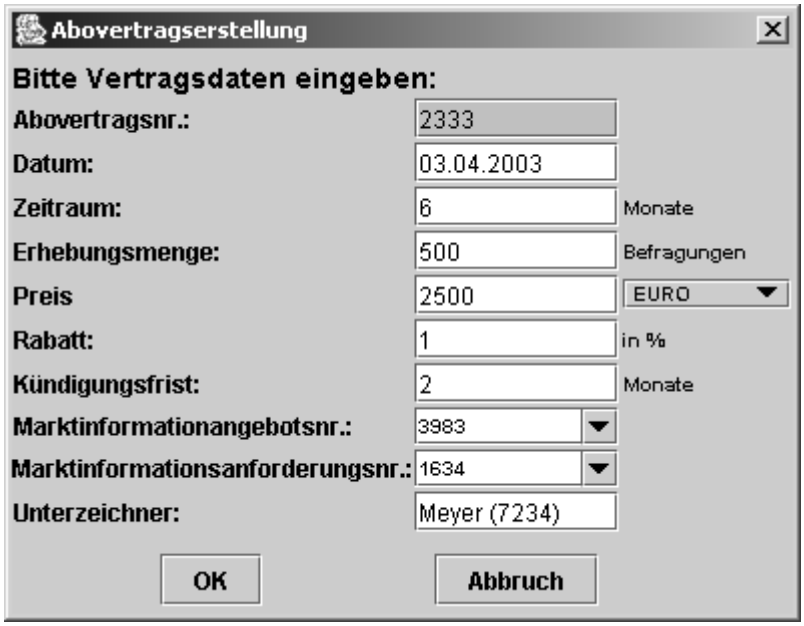


Abbildung 6.14: Grafische Benutzerschnittstelle der JavaBean *Abovertragsabschluss-Dialog*

Abbildung 6.15: Grafische Benutzerschnittstelle der JavaBean *Kreditorrechnungsbearbeitung-Adapter-Dialog*

Die folgenden Tabellen beschreiben die Beziehungen zwischen den Benutzerschnittstellen-Objekten und den Operatoren der entsprechenden Kommunikations-Software-Komponenten in übersichtlicher Form.

Operatoren der Kommunikations-Software-Komponente *Abovertrags-abschluss-Dialog*	**Benutzerschnittstellen-Objekte der JavaBean *Abovertragsabschluss-Dialog***
eingebenVertragsdaten	Beschriftungs-Objekt *Bitte Vertragsdaten eingeben*
	Beschriftungs- und Textfeld-Objekt *Abovertragsnr.*
	Beschriftungs- und Textfeld-Objekt *Datum*
	Beschriftungs- und Textfeld-Objekt *Zeitraum*
	Beschriftungs- und Textfeld-Objekt *Erhebungsmenge*
	Beschriftungs- und Textfeld-Objekt *Preis*
	Beschriftungs- und Textfeld-Objekt *Rabatt*
	Beschriftungs- und Textfeld-Objekt *Kündigungsfrist*
	Beschriftungs- und Kombinationsfeld-Objekt *Marktinformationsangebotsnr.*
	Beschriftungs- und Kombinationsfeld-Objekt *Marktinformationsanforderungsnr.*

Operatoren der Kommunikations-Software-Komponente *Abovertrags-abschluss-Dialog*	**Benutzerschnittstellen-Objekte der JavaBean *Abovertragsabschluss-Dialog***
	Beschriftungs- und Textfeld-Objekt *Unterzeichner*
	Knopf-Objekt *OK*
	Knopf-Objekt *Abbruch*

Tabelle 6.12: Benutzerschnittstellen-Objekte der JavaBean *Abovertragsabschluss-Dialog*

Operatoren der Kommunikations-Software-Komponente *Kreditorrech-nungsbearbeitung-Adapter-Dialog*	**Benutzerschnittstellen-Objekte der JavaBean *Kreditorrechnungsbearbei-tung-Adapter-Dialog***
ausgebenVertragsdaten	Beschriftungs- und Textfeld-Objekt *Abovertragsdaten*
eingebenPrüfungsergebnis	Knopf-Objekt *Akzeptieren*
	Knopf-Objekt *Ablehnen*

Tabelle 6.13: Benutzerschnittstellen-Objekte der JavaBean *Kreditorrechnungsbearbeitung-Adapter-Dialog*

Schließlich enthält Abbildung A.1 im Anhang den zugehörigen *Deployment Descriptor* für die beschriebenen Session- und Entity Beans. Dabei sind insbesondere die Festlegungen der geforderten Persistenzeigenschaften und des notwendigen Transaktionsschutzes der Entity Beans hervorzuheben. Im vorliegenden Fall wird die Wahl der geeigneten Eigenschaftswerte dem Ausführungsdienst (*Container*) der Ausführungsplattform überlassen. Weitere Erläuterungen zum Aufbau eines *Deployment Descriptor* und zu den möglichen Eigenschaften und Eigenschaftswerten, die in einem *Deployment Descriptor* festgelegt werden können, sind z. B. in [SUN02a, 36; Mons01, 33 ff; DePe02, 51; Blev01, 601 f] beschrieben.

6.2 Fallstudie PC-Fertigungsunternehmen

Im Mittelpunkt der zweiten Fallstudie steht ein Fertigungsunternehmen für Personal Computer (PC). Es stellt PCs auftragsbezogen her und liefert sie anschließend an seine Kunden aus. Diese Fallstudie basiert auf der Dokumentation eines Projekts, das die Kopplung von Geschäftsprozessen zweier Fertigungsunternehmen auf Grundlage des Standard-Anwendungssystems SAP R/3 zum Ziel hatte [Kell99, 289 ff]. Im Gegensatz zur ersten Fallstudie lagen die dokumentierten Prozesse diesmal nicht in Form eines Geschäftsprozessmodells, sondern in Form einer informalen Beschreibung vor. Dadurch konnten wiederum die anwendungssystemspezifischen Prozesse des PC-Fertigungsunternehmens besser identifiziert und hinsichtlich einer geforderten Anwendungssystemunabhängigkeit überarbeitet werden. Diese überar-

beiteten Prozesse stellen nun die Grundlage für die Anwendung der SOM-Methodik von der Geschäftsprozessmodellebene über die Ebene der fachlichen Anwendungssystemspezifikation bis zur erweiterten Ebene des softwaretechnischen Entwurfs dar.

6.2.1 Unternehmensplan

Ausgehend von den Beschreibungen der überarbeiteten Prozesse und ergänzenden Informationen aus der zugrundeliegenden Projektdokumentation [Kell99, 289 ff] wird zunächst ein Unternehmensplan für das PC-Fertigungsunternehmen aufgestellt. Dieser umfasst ein Objektsystem und ein Zielsystem, die beide in der folgenden Tabelle zusammengefasst werden.

Objektsystem	**Zielsystem**	
Es handelt sich um ein Fertigungsunternehmen für Personal Computer (PC).	Das Sachziel des Fertigungsunternehmens ist die Herstellung und der Vertrieb von PCs.	**Sachziel**
Das Fertigungsunternehmen liefert eigengefertigte PCs an Kunden aus. Die zur Fertigung eines PCs benötigten Bauteile werden von einem Zulieferer produziert und geliefert.	Die Herstellung soll auftragsbezogen erfolgen. Kunden sollen ihren gewünschten PC innerhalb eines angebotenen Variantenspektrums frei konfigurieren können. Die zur Herstellung eines PCs benötigten Bauteile sollen fremdbezogen werden. Dabei wird ein bedarfsgesteuerter Abruf der benötigten Bauteile aus einem mit dem Lieferanten abgeschlossenen Rahmenvertrag bevorzugt. Auf eine gleichbleibend hohe Qualität der zugelieferten Bauteile wird großer Wert gelegt. Es wird eine Kombination aus Umsatz- und Gewinnmaximierung angestrebt.	**Formalziele**

Tabelle 6.14: Objekt- und Zielsystem des PC-Fertigungsunternehmens

6.2.2 Geschäftsprozessmodell

Das Geschäftsprozessmodell des PC-Fertigungsunternehmens besteht wiederum aus mehreren IAS-Zerlegungen und mehreren mit den IAS-Zerlegungen korrespondieren VES-Zerlegungen. Die IAS-Zerlegungen sind in Abbildung 6.16 bis Abbildung 6.19 dargestellt. Aus den bereits genannten Gründen zeigt Abbildung A.XI im Anhang nur die mit der letzten IAS-Zerlegung korrespondierende VES-Zerlegung. Ferner enthalten Tabelle 6.15 und Tabelle 6.16 die durchgeführten Zerlegungen der betrieblichen Objekte und der betrieblichen Transaktionen nochmals in Form von Zerlegungsbäumen.

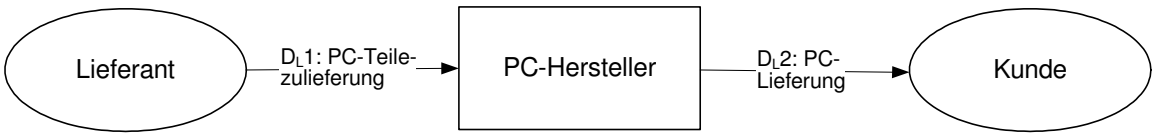


Abbildung 6.16: Initiales Interaktionsschema für den Geschäftsprozess PC-Herstellung und -Vertrieb

Abbildung 6.17: Interaktionsschema für den Geschäftsprozess PC-Herstellung und -Vertrieb (1.Zerlegung)

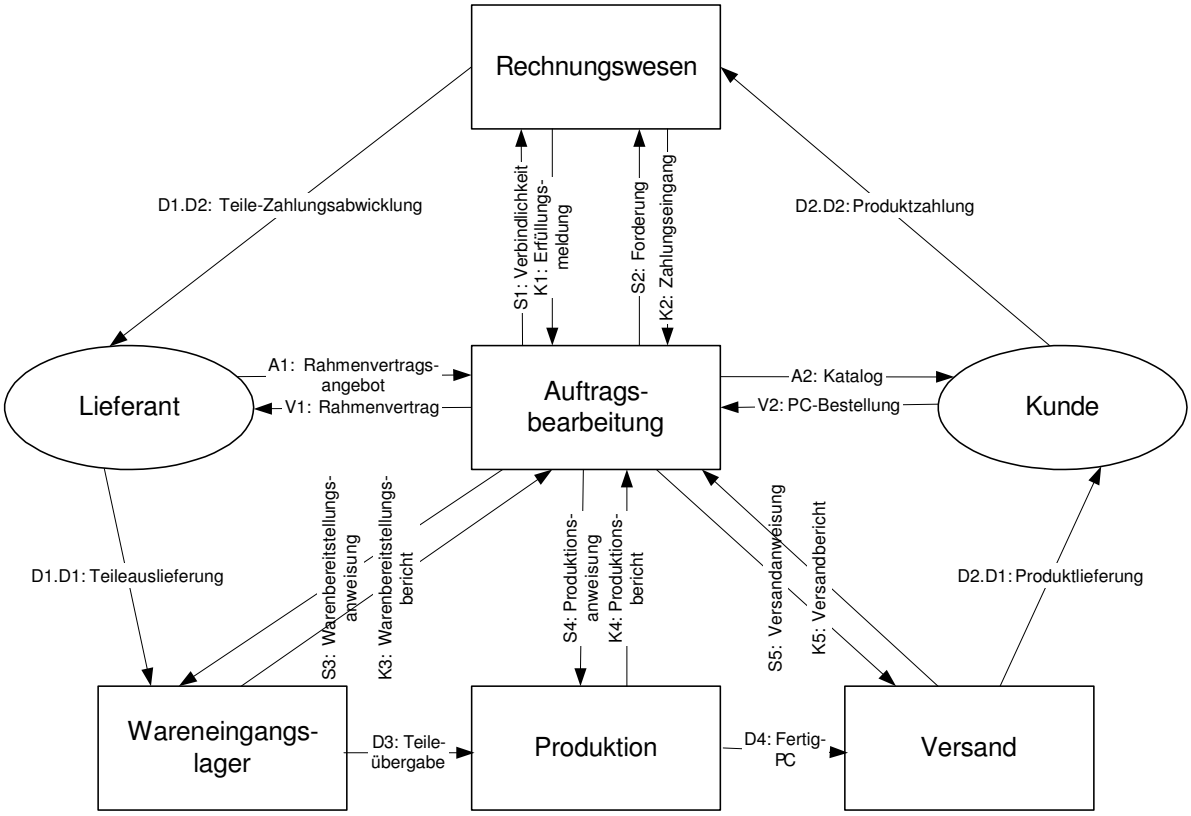


Abbildung 6.18: Interaktionsschema für den Geschäftsprozess PC-Herstellung und -Vertrieb (2.Zerlegung)

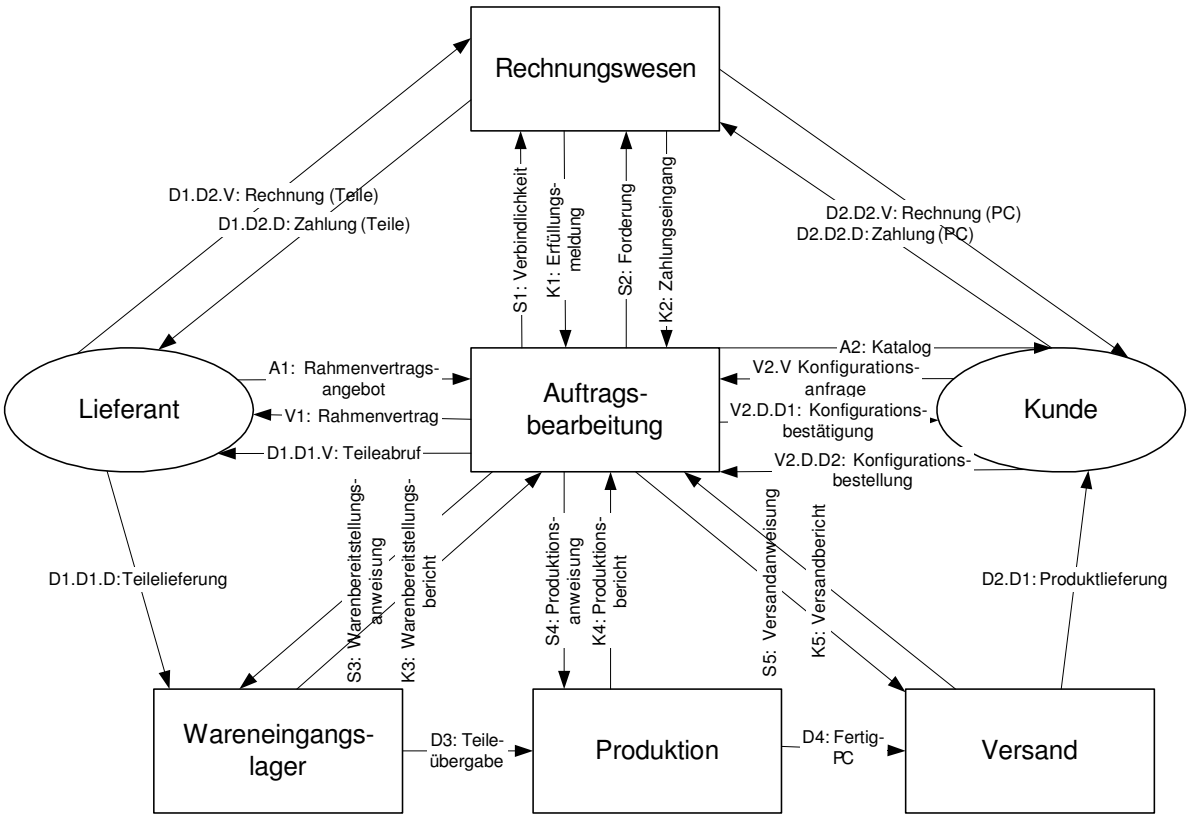


Abbildung 6.19: Interaktionsschema für den Geschäftsprozess PC-Herstellung und -Vertrieb (3.Zerlegung)

Transaktionszerlegung
D_L1: PC-Teilezulieferung
A1: Rahmenvertragsangebot
V1: Rahmenvertrag
D1: Teilezustellung
D1.D1: Teileauslieferung
D1.D1.V: Teileabruf
D1.D1.D: Teilelieferung
D1.D2: Teile-Zahlungabwicklung
D1.D2.V: Rechnung (Teile)
D1.D2.D: Zahlung (Teile)
D_L2: PC-Lieferung
A2: Katalog
V2: PC-Bestellung
V2.V: Konfigurationsanfrage
V2.D: Konfigurationsvertrag
V2.D.D1:Konfigurationsbestätigung
V2.D.D2: Konfigurationsbestellung
D2: PC-Auslieferung
D2.D1: Produktlieferung
D2.D2: Produktzahlung
D2.D2.V: Rechnung (PC)
D2.D2.D: Zahlung (PC)

Tabelle 6.15: Transaktionszerlegung des Geschäftsprozesses PC-Herstellung und -Vertrieb

Objektzerlegung
PC-Hersteller
Rechnungswesen
Auftragsbearbeitung
Wareneingangslager
Produktion
Versand
S1: Verbindlichkeit
K1: Erfüllungsmeldung
S2: Forderung
K2: Zahlungseingang
S3: Warenbereitstellungsanweisung
K3: Warenbereitstellungsbericht
S4: Produktionsanweisung
K4: Produktionsbericht
S5: Versandanweisung
K5: Versandbericht

Objektzerlegung
D3: Teileübergabe
D4: Fertig-PC

Tabelle 6.16: Objektzerlegung des Geschäftsprozesses PC-Herstellung und -Vertrieb

Die wesentlichen Ergebnisse der Geschäftsprozessmodellierung des PC-Fertigungsunternehmens werden im Folgenden zusammengefasst. Ein Kunde bekommt vom PC-Fertigungsunternehmen einen Katalog zugeschickt, in dem alle produzierbaren PC-Konfigurationen aufgeführt sind. Falls sich der Kunde für eine bestimmte PC-Konfiguration entscheidet und die Fertigung dieser Konfiguration technisch möglich ist, gibt er bei dem PC-Fertigungsunternehmen eine entsprechende Bestellung auf. Daraufhin ruft das PC-Fertigungsunternehmen die zur Herstellung des bestellten PCs benötigten Bauteile bei einem Lieferanten ab. Der Abruf erfolgt dabei auf Basis eines Rahmenvertrags, den das PC-Fertigungsunternehmen mit dem Lieferanten vorher abgeschlossen hat. Zur Erfüllung des Abrufs liefert der Lieferant die benötigten Bauteile an das PC-Fertigungsunternehmen. Dieses überprüft zunächst die Qualität der gelieferten Bauteile. Im Falle eines positiven Überprüfungsergebnisses, fertigt das PC-Fertigungsunternehmen den bestellten PC mittels der gelieferten und geprüften Bauteile und stellt den fertiggestellten PC dem Kunden zu.

6.2.3 Fachliche Anwendungssystemspezifikation

Die fachliche Anwendungssystemspezifikation für das PC-Fertigungsunternehmen beginnt wiederum mit der Automatisierbarkeitsanalyse und der darauffolgenden Automatisierung der identifizierten betrieblichen Aufgaben und Transaktionen. Wie in der ersten Fallstudie werden die entsprechenden Ergebnisse in den Tabellen A.4 und A.5 im Anhang dargestellt.

Initiale fachliche Anwendungsspezifikation

Aus dem im vorangegangenen Abschnitt spezifizierten Interaktionsschema und dem zugehörigen Vorgangs-Ereignisschema werden unmittelbar das in Abbildung A.XII im Anhang dargestellte initiale konzeptuelle Objektschema und die in Abbildung A.XIII im Anhang enthaltenen initialen Vorgangsobjektschemata abgeleitet. Sie sind wiederum Grundlage für die folgenden weiteren Ausarbeitungen.

Weitere Ausarbeitung der initialen fachlichen Anwendungsspezifikation

Wie aus den Tabellen A.4 und A.5 ersichtlich, werden auch in diesem Fall alle Aufgaben zumindest teil-automatisiert. Deswegen müssen auch keine konzeptuellen Objekttypen oder Vorgangsobjekttypen aus den initialen Objektschemata entfernt werden.

Tabelle 6.17 und Tabelle 6.18 stellen die Zuordnung von Attributen zu den initialen KOTs und die daraufhin vorgenommene Zusammenfassung von KOTs mit überlappenden Attributen dar. Auf die Definition der von den KOTs zu interpretierenden Nachrichten wird aus den in Abschnitt 6.1.3 genannten Gründen verzichtet. Abbildung A.XIV im Anhang zeigt das resultierende KOS einschließlich den Existenzabhängigkeitsbeziehungen zwischen den KOTs und den zugehörigen Kardinalitäten.

initiale OOTs und LOTs	**konsolidierte OOTs und LOTs**
Versand (V_ID, Verantwortlicher, Telefon)	dto.
Produktion (P_ID, Verantwortlicher, Kapazität)	dto.
Auftragsbearbeitung (AB_ID, Verantwortlicher, Telefon)	dto.
Lieferant (L_ID, Ansprechpartner, Telefon)	dto.
Kunde (K_ID, Name, PLZ, Ort, Strasse)	dto.
Wareneingangslager(WEL_ID, Verantwortlicher, Telefon)	dto.
PC-Lieferung (PCL_ID, Artikelnr., Beschreibung)	dto.
PC-Teilezulieferung (PCTZ_ID, Positionen, Beschreibung)	dto.
Rechnungswesen (R_ID, Verantwortlicher, Telefon, Buchungen)	dto.

Tabelle 6.17: Initiale und konsolidierte OOTs und LOTs des PC-Fertigungsunternehmens

initiale TOTs	**konsolidierte TOTs**
Rahmenvertragsangebot(RVA_ID, Gültigkeitszeitraum, AB_ID, L_ID, PCTZ_ID)	Teilekatalog(TK_ID, Version, Gültigkeitszeitraum, AB_ID, L_ID, PCTZ_ID)
Rahmenvertrag (RV_ID, Datum, Ort, Unterzeichner, RVA_ID)	Rahmenvertrag (RV_ID, Datum, Ort, Unterzeichner, TK_ID)
Katalog (KAT_ID, Version, Gültigkeitszeitraum, AB_ID, K_ID, PCL_ID)	PC-Katalog(PCK_ID, Version, Gültigkeitszeitraum, AB_ID, K_ID, PCL_ID)
Konfigurationsanfrage(KFA_ID, Datum, Priorität, Beschreibung, KAT_ID)	Konfigurationsvertrag (KFV_ID, Datum, Beschreibung, Priorität, Status, Fälligkeit, PCK_ID)
Konfigurationsbestätigung(KFB_ID, Datum, KFA_ID)	
Konfigurationsbestellung(KFBLL_ID, Datum, Fälligkeit, Status, KFB_ID)	
Teileabruf (TA_ID, Teileliste, Datum, Abrufstatus, KFBLL_ID, RV_ID)	Lieferauftrag (LA_ID, Teileliste, Datum, Auftragsstatus, KFV_ID, RV_ID)
Verbindlichkeit (VB_ID, Betrag, Fälligkeit, TA_ID, R_ID)	Kreditorrechnung (Teile) (KR_ID, Betrag, Fälligkeit, KtoNr, BLZ, Datum, Status,

initiale TOTs	**konsolidierte TOTs**
Rechnung (Teile) (RECH_ID, Betrag, Fälligkeit, KtoNr, BLZ, Datum, Status, VB_ID)	LA_ID, R_ID)
Zahlung (Teile) (ZAT_ID, Datum, Betrag, RECH_ID)	Kreditorzahlung (Teile) (KZ_ID, Datum, Betrag, Verantwortlicher, KR_ID)
Erfüllungsmeldung (EFM_ID, Datum, Verantwortlicher, ZAT_ID)	
Warenbereitstellungsanweisung (WBA_ID, Bestelldatum, TA_ID, WEL_ID)	Warenbereitstellungsanweisung (WBA_ID, Bestelldatum, LA_ID, WEL_ID)
Produktionsanweisung (PA_ID, Datum, Priorität, WBA_ID, P_ID)	dto.
Teilelieferung (TL_ID, Positionen, Datum, WBA_ID)	Teileliefereinheit(TLE_ID, Positionen, Datum, WBA_ID, PA_ID)
Teileübergabe (TUE_ID, Positionen, Datum, TL_ID, PA_ID)	
Warenbereitstellungsbericht (WBB_ID, Lieferdatum, Verantwortlicher, TUE_ID)	Warenbereitstellungsbericht (WBB_ID, Lieferdatum, Verantwortlicher, TLE_ID)
Versandanweisung (VA_ID, Fälligkeit, Dringlichkeit, WBB_ID, V_ID)	dto.
Fertig-PC (FPC_ID, Bezeichnung, VA_ID, TUE_ID)	Produkt (PRO_ID, Fertigstellungsdatum, Status, Bezeichnung, VA_ID, TLE_ID)
Produktlieferung (PL_ID, Datum, Status, FPC_ID)	
Versandbericht (VSB_ID, Datum, Verantwortlicher, FPC_ID)	Versandbericht (VSB_ID, Datum, Verantwortlicher, PRO_ID)
Produktionsbericht (PRB_ID, Datum, Verantwortlicher, FPC_ID)	Produktionsbericht (PRB_ID, Datum, Verantwortlicher, PRO_ID)
Forderung (FORD_ID, Betrag, Fälligkeit, Datum, VSB_ID)	Debitorrechnung(DR_ID, Betrag, Fälligkeit, Datum, KtoNr, BLZ, Status, VSB_ID)
Rechnung (PC) (RECHPC_ID, Betrag, Fälligkeit, Datum, KtoNr, BLZ, Status, FORD_ID)	
Zahlung (PC) (ZAPC_ID, Datum, Betrag, RECHPC_ID)	Debitorzahlung (PC) (DZ_ID, Betrag, Datum, DR_ID)
Zahlungseingang (ZAEIN_ID, Betrag, Datum, ZAPC_ID)	

Tabelle 6.18: Initiale und konsolidierte TOTs des PC-Fertigungsunternehmens

Die Zuordnung von Teilgraphen des ausgearbeiteten KOS zu den VOTs der initialen Vorgangsobjektschemata enthält Tabelle 6.19. Der dafür notwendige Tabellenaufbau

wurde bereits in Abschnitt 6.1.3 erläutert. Im Hinblick auf die noch folgende Definition von Vorgangs-Software-Komponenten wird auf eine Zusammenfassung von VOTs aus semantischen Gründen verzichtet. Tabelle A.6 im Anhang stellt die von den VOTs angenommenen und verarbeiteten Nachrichten übersichtlich dar.

VOTs	**KOTs**
Rahmenvertragsangebot> >Rahmenvertragsangebot	Teilekatalog(TK_ID, Version, Gültigkeitszeitraum, AB_ID, L_ID, PCTZ_ID)
Rahmenvertrag> >Rahmenvertrag	Rahmenvertrag (RV_ID, Datum, Ort, Unterzeichner, TK_ID)
Katalog> >Katalog	PC-Katalog(PCK_ID, Version, Gültigkeitszeitraum, AB_ID, K_ID, PCL_ID)
Konfigurationsanfrage> > Konfigurationsanfrage Konfigurationsbestätigung> > Konfigurationsbestätigung Konfigurationsbestellung> >Konfigurationsbestellung	Konfigurationsvertrag (KFV_ID, Datum, Beschreibung, Priorität, Status, Fälligkeit, PCK_ID)
Teileabruf> >Teileabruf	Lieferauftrag (LA_ID, Teileliste, Datum, Auftragsstatus, KFV_ID, RV_ID)
Verbindlichkeit > > Verbindlichkeit Rechnung (Teile) > > Rechnung (Teile)	Kreditorrechnung (Teile) (KR_ID, Betrag, Fälligkeit, KtoNr, BLZ, Datum, Status, LA_ID, R_ID)
Zahlung (Teile) > > Zahlung (Teile) Erfüllungsmeldung > > Erfüllungsmeldung	Kreditorzahlung (Teile) (KZ_ID, Datum, Betrag, Verantwortlicher, KR_ID)
Warenbereitstellungsanweisung > Warenbereitstellungsanweisung	Warenbereitstellungsanweisung (WBA_ID, Bestelldatum, LA_ID, WEL_ID)
Produktionsanweisung > > Produktionsanweisung	Produktionsanweisung (PA_ID, Datum, Priorität, WBA_ID, P_ID)
Teilelieferung > >Teilelieferung Teileübergabe > > Teileübergabe	Teileliefereinheit(TLE_ID, Positionen, Datum, WBA_ID, PA_ID)

VOTs	KOTs
Warenbereitstellungsbericht > > Warenbereitstellungsbericht	Warenbereitstellungsbericht (WBB_ID, Lieferdatum, Verantwortlicher, TLE_ID)
Versandanweisung > > Versandanweisung	Versandanweisung (VA_ID, Fälligkeit, Dringlichkeit, WBB_ID, V_ID)
Fertig-PC > > Fertig-PC Produktlieferung > > Produktlieferung	Produkt (PRO_ID, Fertigstellungsdatum, Status, Bezeichnung, VA_ID, TLE_ID)
Versandbericht > > Versandbericht	Versandbericht (VSB_ID, Datum, Verantwortlicher, PRO_ID)
Produktionsbericht > > Produktionsbericht	Produktionsbericht (PRB_ID, Datum, Verantwortlicher, PRO_ID)
Forderung > > Forderung > Rechnung (PC) Rechnung (PC)>	Debitorrechnung(DR_ID, Betrag, Fälligkeit, Datum, KtoNr, BLZ, Status, VSB_ID)
Zahlung (PC)> > Zahlung (PC) Zahlungseingang > > Zahlungseingang	Debitorzahlung (PC) (DZ_ID, Betrag, Datum, DR_ID)

Tabelle 6.19: Zuordnung von KOTs zu VOTs des PC-Fertigungsunternehmens

Aus den in Abschnitt 6.1.3 genannten Gründen wird auch in diesen ausgearbeiteten Objektschemata auf die Definition von Operatoren vorerst verzichtet.

6.2.4 Softwaretechnischer Entwurf

Auf der Grundlage des ausgearbeiteten KOS und der ausgearbeiteten VOS wird nun der softwaretechnische Entwurf von Anwendungssystemen für das PC-Fertigungsunternehmen durchgeführt. Allerdings beschränkt sich die Beschreibung der Ergebnisse auch in diesem Fall auf einen Ausschnitt, der für den Nachweis der praktischen Anwendbarkeit des vorgestellten komponentenbasierten Software-Architekturmodells geeignet ist. Da die Verwendung des komponentenbasierten Software-Architekturmodells bereits im vorangegangenen Abschnitt ausführlich erläutert wurde, verzichten die folgenden Ausführungen auf eine nochmalige Erläuterung der beschriebenen Schritte und legen nur die Ergebnisse der softwaretechnischen Spezifikation des gewählten Ausschnitts dar. Die eigentliche Wiederverwendung der

aus dem softwaretechnischen Entwurf der ersten Fallstudie entstandenen Software-Komponenten im softwaretechnischen Entwurf dieser Fallstudie wird dagegen ausführlich erläutert.

Spezifikation der Struktur- und Verhaltensmerkmale eines komponentenbasierten Anwendungssystems aus der Außenperspektive und aus der Innenperspektive

Die Spezifikation der Vorgangs-Software-Komponenten, die den Ausgangspunkt für die weitere softwaretechnische Spezifikation von komponentenbasierten Anwendungssystemen für das PC-Fertigungsunternehmen darstellen, basiert auf der im vorangegangenen Abschnitt vorgestellten Tabelle 6.19. Sie beschreibt die Zuordnung von Teilgraphen des ausgearbeiteten KOS zu den Vorgangsobjekttypen der VOS. Der Ausschnitt, der im Folgenden näher betrachtet wird, umfasst zwei Vorgangs-Software-Komponenten einschließlich den zugehörigen Entitäts-Software-Komponenten.

Abgrenzung von Vorgangs-Software-Komponenten

Die erste Vorgangs-Software-Komponente wird anhand der Vorgangsobjekttypen *Verbindlichkeit>*, *>Verbindlichkeit*, *Rechnung (Teile)>* und *>Rechnung (Teile)* sowie den zugehörigen Beziehungen abgegrenzt. Da der fachliche Belang, der mit dieser Software-Komponente abgebildet wird, die Erstellung und Bearbeitung von Kreditorrechnungen beschreibt, wird ihr der fachliche Name *Kreditorrechnungserstellung und -bearbeitung* zugewiesen. Die zweite Vorgangs-Software-Komponente kapselt die Vorgangsobjekttypen *Teileabruf>* und *>Teileabruf* sowie die entsprechende Beziehung. In diesem Fall wird der fachliche Name dieser Software-Komponente, *Teileabruf*, unmittelbar von den enthaltenen Vorgangsobjekttypen übernommen.

Diese Vorgangs-Software-Komponenten stehen untereinander in wechselseitiger Beziehung. Ursache dafür sind die Beziehungen zwischen den Vorgangsobjekttypen *Teileabruf>* und *Verbindlichkeit>* sowie *>Teileabruf* und *Rechnung (Teile)>*.

Die Operatorensignaturen der Vorgangs-Software-Komponenten basieren auf den Nachrichtendefinitionen der jeweils zugeordneten Vorgangsobjekttypen. Diese wurden in Tabelle A.6 im Anhang spezifiziert. Demnach besitzt die *Teileabruf*-Komponente die Operatoren *erstellenLieferauftrag* und *annehmenLieferauftrag* und die *Kreditorrechnungserstellung und -bearbeitung*-Komponente die Operatoren *erstellenVerbindlichkeit*, *buchenVerbindlichkeit*, *erstellenRechnung*, *prüfenRechnung* und *begleichenRechnung*.

Abgrenzung von Entitäts-Software-Komponenten und deren Zuordnung zu den Vorgangs-Software-Komponenten

Anschließend werden die Entitäts-Software-Komponenten abgegrenzt und den Vorgangs-Software-Komponenten zugeordnet. Die der *Kreditorrechnungserstellung und -bearbeitung*-Komponente zuzuordnende Entitäts-Software-Komponente enthält die konzeptuellen Objekttypen *Kreditorrechnung* und *Rechnungswesen* einschließlich der zugehörigen Beziehung und besitzt den Namen *Kreditorrechnung*. Der *Teileabruf*-Komponente wird die Entitäts-Software-Komponente mit dem Namen *Lieferauftrag* und den konzeptuellen Objekttypen *Lieferauftrag*, *Rahmenvertrag*, *Konfigurationsvertrag*, *Teilekatalog, PC-Katalog, Auftragsbearbeitung*, *Lieferant*, *Kunde*, *PC-Teilezulieferung* und *PC-Lieferung* sowie den zugehörigen Beziehungen zugeordnet.

Spezifikation von Kommunikations-Software-Komponenten auf Basis der Vorgangs-Software-Komponenten

Daraufhin folgt die Definition der Kommunikations-Software-Komponenten für die Mensch-Computer-Kommunikation, die ebenfalls den Vorgangs-Software-Komponenten zugeordnet werden. Sie basiert auf den Ergebnissen der Automatisierung, die in Tabelle A.4 im Anhang zusammengefasst sind, und den Nachrichtendefinitionen der den Vorgangs-Software-Komponenten zugeordneten Vorgangsobjekttypen. Demnach ist für jede Vorgangs-Software-Komponente eine zugehörige Kommunikations-Software-Komponente zu spezifizieren. Daraus resultieren die Kommunikations-Software-Komponenten *Teileabruf-Dialog* und *Kreditorrechnungserstellung und -bearbeitung-Dialog*. Die Spezifikationen der zugehörigen Operatorensignaturen werden, analog zu Abschnitt 6.1.4, anhand der Operatornamen in den folgenden Tabellen zusammengefasst.

Operator *Teileabruf*	**Operator *Teileabruf-Dialog***
erstellenLieferauftrag	*eingebenAuftragsdaten*

Tabelle 6.20: Operatoren der Kommunikations-Software-Komponente *Teileabruf-Dialog*

Operatoren *Kreditorrechnungserstellung und -bearbeitung*	**Operatoren *Kreditorrechnungserstellung und -bearbeitung-Dialog***
prüfenRechnung	*ausgebenRechnungAuftrag*
	eingebenRechnungsprüfungsergebnis
begleichenRechnung	*eingebenZahlungsanweisung*

Tabelle 6.21: Operatoren der Kommunikations-Software-Komponente *Kreditorrechnungserstellung und -bearbeitung-Dialog*

Spezifikation von Datenhaltungs-Software-Komponenten auf Basis der Entitäts-Software-Komponenten

Im letzten Schritt werden für die Entitäts-Software-Komponenten entsprechende Datenhaltungs-Software-Komponenten spezifiziert und ihnen zugeordnet. Wie bereits im Rahmen der Ausführungen zur letzten Fallstudie erwähnt, muss dabei kein bestehendes Datenbanksystem berücksichtigt werden. Das bedeutet, dass zwischen einer Entitäts-Software-Komponente und einer Datenhaltungs-Software-Komponente eine 1:1-Abbildungsbeziehung einführbar ist. Folglich entsprechen sich die Software-Komponenten im Wesentlichen.

Resultierendes komponentenbasiertes Anwendungssystems aus der Außenperspektive und aus der Innenperspektive

Abbildung A.XV im Anhang enthält die resultierende Außenperspektive des erläuterten Ausschnitts in Form eines Software-Komponenten-Diagramms. Es verdeutlicht insbesondere die Beziehungen zwischen den spezifizierten Software-Komponenten und deren Schnittstellen. Analog zu den bisher gezeigten Software-Komponenten-Diagrammen wird auf die Darstellung der Datenhaltungs-Software-Komponenten aus Gründen der Übersichtlichkeit verzichtet. Ferner weisen die Entitäts- und die Kommunikations-Software-Komponenten keine zusätzlichen OCL-Ausdrücke zur Beschreibung ihres Verhaltens auf, da deren Operatorensignaturen dafür bereits genügen.

Die zugehörige Innenperspektive des Ausschnitts wird in Abbildung A.XVI anhand eines Klassendiagramms beschrieben. Es zeigt hauptsächlich die Beziehungen zwischen den Objekttypen und deren Schnittstellen. Die zu den Datenhaltungs-Software-Komponenten gehörenden Datenhaltungs-Objekttypen wurden aus den genannten Gründen nicht mit aufgenommen.

Zur Verdeutlichung, dass die Software-Komponenten im softwaretechnischen Entwurf dieser Fallstudie entsprechend dem Software-Architekturmodell gestaltet worden sind, werden wiederum die zur Spezifikation einer Software-Komponente bereitgestellten Metaobjekte der Kern-Metamodelle sowie der Repräsentations-Metamodelle der statischen Struktur und des statischen Verhaltens auf beiden Modellebenen den entsprechenden Struktur- und Verhaltensmerkmalen der Vorgangs-Software-Komponente *Teileabruf* gegenübergestellt. Das Ergebnis dieser Gegenüberstellung enthalten die folgenden Tabellen.

Außenperspektive-Modellebene		
Metaobjekte des Kern-Metamodells	**Metaobjekte des Repräsentations-Metamodells der statischen Struktur und des statischen Verhaltens**	**Merkmale der Vorgangs-Software-Komponente *Teileabruf***
Vorgangs-Software-Komponente	Subsystem mit Stereotyp „<<Vorgang>>“	Subsystem, *zugehöriger Stereotyp aus Übersichtlichkeitsgründen nicht dargestellt*
Software-Komponentenname	Subsystemname	„Teileabruf“
fachliche Schnittstelle	Schnittstellenbereich mit Stereotyp „<<Funktion>>“	Schnittstellenbereich mit Stereotyp „<<Funktion>>“
Verwaltungsschnittstelle	Schnittstellenbereich mit Stereotyp „<<Verwaltung>>“	Schnittstellenbereich mit Stereotyp „<<Verwaltung>>“
fachliche Kurzbeschreibung	Anmerkung	*Anmerkung aus Übersichtlichkeitsgründen nicht dargestellt*
Komponentenoperatorsignatur	Operatorsignatur	Teileabruf erstellenTeileabruf(); void zerstoerenTeileabruf(); Lieferauftrag erstellenLieferauftrag(in sequence rvIds, in sequence kvIds) boolean annehmenLieferauftrag(in Lieferauftrag einLAuftrag)
Ausnahmedeklaration	Ausnahme	raises (RahmenvertragFehlerhaft, KonfigurationsvertragFehlerhaft, ErstellungGescheitert) raises (LieferauftragFehlerhaft, AnnahmeGescheitert)
Vorbedingung	Vorbedingung	pre: rvIds -> forAll (rvId|Rahmenvertrag -> exists (rv|rv.rv_id=rvId)) and kvIds -> forAll (kvId|Konfigurationsvertrag -> exists (kv|kv.kv_id=kvId)) pre: Lieferauftrag -> exists (la|la.la_id=einLAuftrag.la_id)

Außenperspektive-Modellebene		
Metaobjekte des Kern-Metamodells	**Metaobjekte des Repräsentations-Metamodells der statischen Struktur und des statischen Verhaltens**	**Merkmale der Vorgangs-Software-Komponente *Teileabruf***
Nachbedingung	Nachbedingung	post: Lieferauftrag -> exists (la|la.oclIsNew() and Rahmenvertrag -> exists (rv|rv.rv_id=la.rahmenvertrag.rv_id) and Konfigurationsvertrag -> exists (kv|kv.kv_id=la.konfigurations-vertrag.kv_id)) post: result = Lieferauftrag -> exists (la|la.la_id=einLAuftrag.la_id and la.teileliste -> notEmpty() and la.datum <> "" and Rahmenvertrag -> exists (rv|rv.rv_id=la.rahmenvertrag.rv_id) and Konfigurationsvertrag -> exists (kv|kv.kv_id=la.konfigurations-vertrag.kv_id) and la.auftragsstatus="angenommen")
funktionale Invariante	Funktionale Invariante	inv: Konfigurationsvertrag -> not-Empty() and Rahmenvertrag -> notEmpty()
Deklaration nicht-funktionaler Eigenschaften	Schnittstellenbereich mit Stereotyp „<<Nicht-funktionale Eigenschaften>>“	Schnittstellenbereich mit Stereotyp „<<Nicht-funktionale Eigenschaften>>“
nicht-funktionale Eigenschaft	Nicht-funktionale Eigenschaft	transactionManagedOperations=operations() transactionManaged-Type=REQUIRED

Tabelle 6.22: Gegenüberstellung der Kern- und Repräsentations-Metaobjekte auf der Außenperspektive-Modellebene und der Merkmale der Vorgangs-Software-Komponente *Teileabruf*

Innenperspektive-Modellebene		
Metaobjekte des Kern-Metamodells	**Metaobjekte des Repräsentations-Metamodells der statischen Struktur und des statischen Verhaltens**	**Merkmale der Vorgangs-Software-Komponente *Teileabruf***
Vorgangsklasse	Klasse mit Stereotyp „<<Vorgang>>“	Klassen, *zugehörige Stereotypen aus Übersichtlichkeitsgründen nicht dargestellt*
Klassenname	Klassenname	„Teileabruf>“ „>Teileabruf “
Operatorsignatur	Operator	*Operatoren aus Übersichtlichkeitsgründen nicht dargestellt*
Operatorimplementierung	-	-
Attribut	Attribut	*Attribute aus Übersichtlichkeitsgründen nicht dargestellt*
Intra-Komponenten-Beziehung	Klassenbeziehung, ausgenommen Abhängigkeits- und Realisierungsbeziehung	gerichtete Assoziation zwischen den beteiligten Klassen
Assoziationsname	Beziehungsname	*Beziehungsnamen nicht benötigt*
Nachricht	-	-
Kardinalität 1	Multiplizität 1	*Multiplizitäten aus Übersichtlichkeitsgründen nicht dargestellt*
Kardinalität 2	Multiplizität 2	*Multiplizitäten aus Übersichtlichkeitsgründen nicht dargestellt*

Tabelle 6.23: Gegenüberstellung der Kern- und Repräsentations-Metaobjekte auf der Innenperspektive-Modellebene und der Merkmale der Vorgangs-Software-Komponente *Teileabruf*

Vor der Spezifikation der jeweils korrespondierenden Verhaltenssichten, werden die aus dem KOS und den VOS abgeleiteten Software-Komponenten im Hinblick auf die angestrebte Wiederverwendung von Software-Komponenten aus der ersten Fallstudie überarbeitet.

Untersuchung des komponentenbasierten Anwendungssystems hinsichtlich einer Wiederverwendung bereits exisitierender Software-Komponenten

Anhand der Schnittstellen und den entsprechenden Beziehungen der Software-Komponenten in den Software-Komponenten-Diagrammen der ersten und der vorliegenden Fallstudie lässt sich erkennen, dass die Vorgangs-Software-Komponente *Kreditorrechnungsbearbeitung* und die zugehörigen Entitäts- und Kommunikations-

Software-Komponenten aus der ersten Fallstudie in der vorliegenden Fallstudie grundsätzlich wiederverwendet werden können. Da jedoch die wiederzuverwendenden Software-Komponenten aus der ersten Fallstudie unverändert (*as is*) in den softwaretechnischen Entwurf der zweiten Fallstudie eingebracht werden sollen, müssen die Vorgangs-Software-Komponente *Kreditorrechnungserstellung und -bearbeitung* sowie die zugeordneten Entitäts- und Kommunikations-Software-Komponenten an die wiederzuverwendenden Software-Komponenten der ersten Fallstudie angepasst werden.

Vorbereitung des komponentenbasierten Anwendungssystems hinsichtlich einer Wiederverwendung bereits exisitierender Software-Komponenten

Die wiederzuverwendende Vorgangs-Software-Komponente *Kreditorrechnungsbearbeitung* bildet, wie der fachliche Name und die fachlichen Operatorensignaturen zeigen, den fachlichen Belang der Bearbeitung von Kreditorrechnungen ab. Dazu gehört die Prüfung der Kreditorrechung auf Plausibilität und, im Falle eines positiven Prüfungsergebnisses, die darauffolgende Veranlassung der Kreditorrechnungsbegleichung. Dies entspricht auch einem Teil des fachlichen Belangs, der mit der Vorgangs-Software-Komponente *Kreditorrechnungserstellung und -bearbeitung* abgebildet wird. Demzufolge müssen für eine Wiederverwendung der *Kreditorrechnungsbearbeitung*-Komponente aus der *Kreditorrechnungserstellung und -bearbeitung*-Komponente diejenigen Operatoren ausgelagert werden, die sich auf die Erstellung der Kreditorrechnung beziehen. Dazu gehören die Operatoren *erstellenVerbindlichkeit*, *buchenVerbindlichkeit* und *erstellenRechnung*. Aus den entsprechenden Vorgangsobjekttypen *Verbindlichkeit>*, *>Verbindlichkeit* und *Rechnung (Teile)>* wird eine zusätzliche Vorgangs-Software-Komponente namens *Kreditorrechnungsbearbeitung-Adapter* erstellt und mit der *Kreditorrechnungsbearbeitung*-Komponente in Beziehung gesetzt. Darüber hinaus muss der Operator *begleichenRechnung* an die entsprechenden Operatoren der *Kreditorrechnungsbearbeitung*-Komponente angepasst werden. Demnach wird der Vorgangsobjekttyp *>Rechnung (Teile)*, der diesen Operator enthält, ebenfalls in die *Kreditorrechnungsbearbeitung-Adapter*-Komponente ausgelagert. Ferner ist es erforderlich, dass die Entitäts-Software-Komponente *Lieferauftrag*, die der Vorgangs-Software-Komponente *Kreditorrechnungserstellung und -bearbeitung* zugeordnet ist, zur Entitäts-Software-Komponente *Liefervertrag*, die zur Vorgangs-Software-Komponente *Kreditorrechnungsbearbeitung* gehört, konvertiert wird. Zu diesem Zweck ist dem in der *Kreditorrechnungsbearbeitung-Adapter*-Komponente enthaltenen Vorgangsobjekttyp *Rechnung (Teile)>* ein weiterer Operator *anpassenDaten* hinzuzufügen.

Die Wiederverwendung der Vorgangs-Software-Komponente *Kreditorrechnungsbearbeitung* bedingt auch die Wiederverwendung der ihr zugeordneten Entitäts-

Software-Komponenten *Kreditorrechnung* und *Liefervertrag* sowie der ihr ebenfalls zugeordneten Kommunikations-Software-Komponente *Kreditorrechnungsbearbeitung-Dialog*.

Für die Konvertierung der Entitäts-Software-Komponente *Lieferauftrag* in die geforderte Entitäts-Software-Komponente *Liefervertrag* ist, wie erwähnt, der in der *Kreditorrechnungsbearbeitung-Adapter*-Komponente enthaltene Operator *anpassenDaten* zuständig. Deswegen muss die *Kreditorrechnungsbearbeitung-Adapter*-Komponente mit beiden Entitäts-Software-Komponenten in Beziehung stehen.

Angesichts der zugrundegelegten 1:1-Abbildungsbeziehungen zwischen den Entitäts-Software-Komponenten und den zugehörigen Datenhaltungs-Software-Komponenten, können die der *Kreditorrechnung*-Komponente und der *Liefervertrag*-Komponente zugeordneten Datenhaltungs-Software-Komponenten in dieser Fallstudie wiederverwendet werden. In diesem Zusammenhang sind keine weiteren Anpassungen notwendig.

Überarbeitetes komponentenbasiertes Anwendungssystems aus der Außenperspektive und aus der Innenperspektive

Das Ergebnis dieser Überarbeitungen zeigt Abbildung A.XVII im Anhang anhand eines Software-Komponenten-Diagramms. Es enthält die Software-Komponenten des zugrundegelegten Ausschnitts und deren Beziehungen untereinander. Die Datenhaltungs-Software-Komponenten sind aus den genannten Gründen nicht darin enthalten. Auf den Verzicht zusätzlicher OCL-Ausdrücke zur Beschreibung des Verhaltens der Entitäts- und die Kommunikations-Software-Komponenten wurde bereits ebenfalls hingewiesen. Abbildung A.XVIII im Anhang beschreibt das Verhalten der Instanzen dieser Software-Komponenten anhand eines Sequenzdiagramms. Beide Diagramme gehören zur Außenperspektive-Modellebene des Software-Architekturmodells.

Auf der zugehörigen Innenperspektive-Modellebene werden die Implementierungen der Software-Komponenten aus der Struktursicht anhand eines Klassendiagramms beschrieben (vgl. Abbildung A.XIX im Anhang). Da im Software-Komponenten-Diagramm auf die Darstellung von Datenhaltungs-Software-Komponenten verzichtet wurde, weist das Klassendiagramm auch keine Datenhaltungs-Objekttypen auf. Die korrespondierende Verhaltenssicht, die die Interaktionen der zu den Objekttypen gehörenden Objekte mithilfe von Sequenzdiagrammen beschreibt, wurde bereits in Abschnitt 6.1.4 an einem Beispiel aufgezeigt. Das gewählte Beispiel bezog sich auf die Interaktion der zur *Kreditorrechnungsbearbeitung*-, *Kreditorrechnung*-, *Liefervertrag*- und *Kreditorrechnungsbearbeitung-Dialog*-Komponente gehörenden Objekte. Da die genannten Software-Komponenten auch im vorliegenden Ausschnitt verwendet werden und die Beschreibung weiterer Sequenzdiagramme keinen zusätzlichen Nutzen

für das Verständnis der durchgeführten Spezifikationen bringt, wird für die Darstellung der auf der Innenperspektive-Modellebene spezifizierten Verhaltenssicht auf Abbildung A.X im Anhang verwiesen.

6.2.5 Implementierung

Die Dokumentation der aus diesem Fallstudienausschnitt resultierenden Enterprise Beans und JavaBeans erfolgt analog zu Abschnitt 6.1.5.

Auf die Implementierung der wiederverwendeten Software-Komponenten *Kreditorrechnungsbearbeitung*, *Kreditorrechnung*, *Liefervertrag* und *Kreditorrechnungsbearbeitungs-Dialog* wurde bereits in Abschnitt 6.1.5 eingegangen. Deswegen beschränken sich die folgenden Ausführungen auf die fallstudienspezifischen Software-Komponenten *Teileabruf*, *Kreditorrechnungsbearbeitung-Adapter*, *Lieferauftrag* und *Teileabruf-Dialog*. Die beiden erstgenannten werden als Session Beans, die mittlere als Entity Bean und die letztgenannte als JavaBean realisiert.

In Abbildung 6.20 und Abbildung 6.21 sind die Schnittstellen der Session Beans, in Abbildung 6.22 die der Entity Bean dargestellt.

```
/**
 * Home interface for Enterprise Bean: Teileabruf
 */
public interface TeileabrufHome extends javax.ejb.EJBHome {

      public Teileabruf create() throws javax.ejb.CreateException,
      java.rmi.RemoteException;

}

 /**
 * Remote interface for Enterprise Bean: Teileabruf
 */
public interface Teileabruf extends javax.ejb.EJBObject {

      public Lieferauftrag erstellenLieferauftrag(java.util.Collection
      rvIds, java.util.Collection kvIds, fallstudie2.ICallbackF2 callback)
      throws java.rmi.RemoteException;

      public boolean annehmenLieferauftrag(Lieferauftrag lf) throws
      java.rmi.RemoteException;

}
```

Abbildung 6.20: Schnittstellen der Session Bean *Teileabruf*

```
/**
 * Home interface for Enterprise Bean: KreditorrechnungsbearbeitungAdapter
 */
public interface KreditorrechnungsbearbeitungAdapterHome extends
javax.ejb.EJBHome {

      public KreditorrechnungsbearbeitungAdapter create() throws
      javax.ejb.CreateException, java.rmi.RemoteException;

}

import fallstudie2.*;
import reuse.ICallback;
/**
 * Remote interface for Enterprise Bean:KreditorrechnungsbearbeitungAdapter
 */
public interface KreditorrechnungsbearbeitungAdapter extends
javax.ejb.EJBObject {

      public String erstellenVerbindlichkeit(Lieferauftrag lf) throws
      java.rmi.RemoteException;

      public boolean buchenVerbindlichkeit(String verbindlichkeit,
      Rechnungswesen rw) throws java.rmi.RemoteException;

      public Kreditorrechnung erstellenRechnung(Lieferauftrag lief) throws
      java.rmi.RemoteException;

      public boolean begleichenRechnung(Kreditorrechnung kr, ICallback
      callback) throws java.rmi.RemoteException;

}
```

Abbildung 6.21: Schnittstellen der Session Bean *Kreditorrechnungsbearbeitung-Adapter*

```
/**
 * Home interface for Enterprise Bean: Lieferauftrag
 */
public interface LieferauftragHome extends javax.ejb.EJBHome {

      public Lieferauftrag create(int lA_ID) throws
      javax.ejb.CreateException, java.rmi.RemoteException;

      public Lieferauftrag findByPrimaryKey(LieferauftragKey primaryKey)
      throws javax.ejb.FinderException, java.rmi.RemoteException;

}

 /**
 * Remote interface for Enterprise Bean: Lieferauftrag
 */
public interface Lieferauftrag extends javax.ejb.EJBObject {

      public int getlA_ID() throws java.rmi.RemoteException;

      public java.lang.String getDatum() throws java.rmi.RemoteException;

      public void setDatum(java.lang.String newDatum) throws
      java.rmi.RemoteException;

      public void setTeileliste(Teileliste teileliste) throws
      java.rmi.RemoteException;

      public Teileliste getTeileliste() throws java.rmi.RemoteException;

      public java.lang.String getAuftragstatus() throws
      java.rmi.RemoteException;

      public void setAuftragstatus(java.lang.String newAuftragstatus)
      throws java.rmi.RemoteException;

      public Konfigurationsvertrag getKonfigurationsvertrag() throws
      java.rmi.RemoteException;

      public Rahmenvertrag getRahmenvertrag() throws
      java.rmi.RemoteException;

}
```

Abbildung 6.22: Schnittstellen der Entity Bean *Lieferauftrag*

Die Verwaltungsschnittstellen aller genannten Enterprise Beans enthalten eine jeweils entsprechende Konstruktorsignatur. Zusätzlich besitzen die Verwaltungsschnittstellen der Entity Bean *Lieferauftrag* eine Operatursignatur zum Auffinden einer bestimmten Instanz.

Die fachlichen Schnittstellen der genannten Enterprise Beans stimmen im Wesentlichen mit denen der entsprechenden Software-Komponenten des softwaretechnischen Entwurfs überein. Es existieren nur marginale Unterschiede, die technisch begründet sind und im Folgenden kurz erläutert werden.

Die Signatur des Operators *erstellenLieferauftrag* in der fachlichen Schnittstelle der *Teileabruf*-Bean weist einen zusätzlichen Parameter des Typs *ICallback* auf. Dieser

wird, wie in Abschnitt 6.1.5 erläutert, im Allgemeinen für die Interaktion mit einer zugeordneten JavaBean benötigt. In diesem Fall handelt es sich um die *Teileabruf-Dialog*-JavaBean. Auch die Signatur des Operators *begleichenRechnung* in der fachlichen Schnittstelle der *Kreditorrechnungsbearbeitungs-Adapter*-Bean besitzt einen Parameter des Typs *ICallback*. Dieser wird vom Operator jedoch nicht direkt verwendet, sondern an den Operator *empfangenRechnung* der *Kreditorrechnungsbearbeitungs*-Bean weitergegeben.

In der fachlichen Schnittstellen der Entity Bean, die die Entitäts-Software-Komponenten *Lieferauftrag* realisiert, sind ausschließlich Operatorensignaturen zum Lesen und zum Schreiben der jeweiligen Attributwerte enthalten.

Die grafische Benutzerschnittstelle der JavaBean *Teileabruf-Dialog* ist in Abbildung 6.23 dargestellt. Ihre Benutzerschnittstellen-Objekte korrespondieren mit dem Operator der entsprechenden Kommunikations-Software-Komponente.

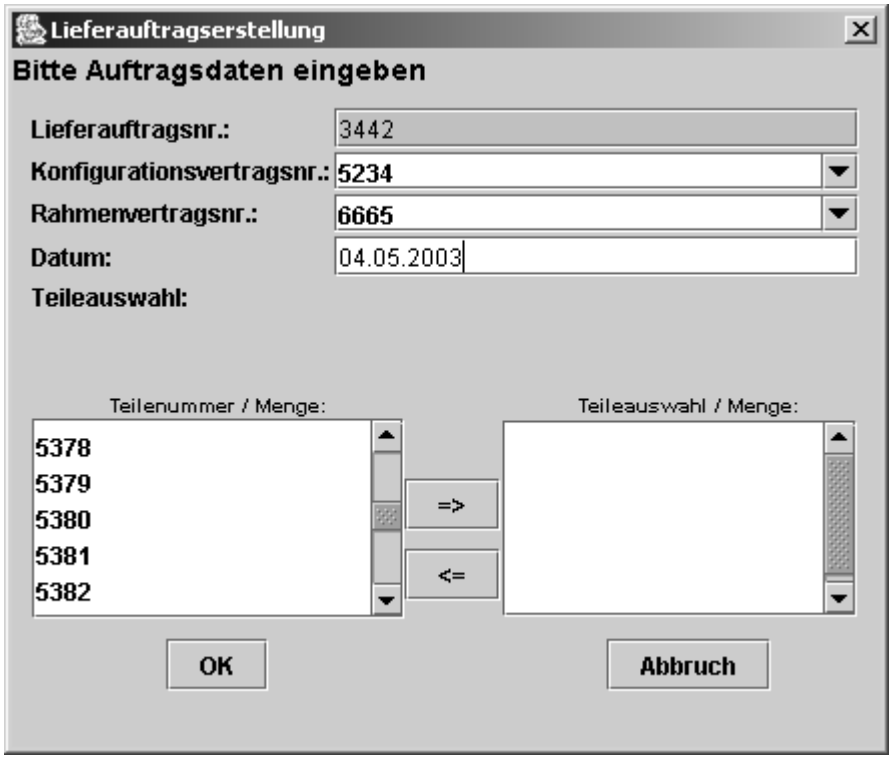


Abbildung 6.23: Grafische Benutzerschnittstelle der JavaBean *Teileabruf-Dialog*

Die folgende Tabelle zeigt übersichtlich die Beziehung zwischen dem Operator der Kommunikations-Software-Komponente *Teileabruf-Dialog* und die zugeordneten Benutzerschnittstellen-Objekte der gleichnamigen JavaBean.

Operator der Kommunikations-Software-Komponente *Teileabruf-Dialog*	**Benutzerschnittstellen-Objekte der JavaBean *Teileabruf-Dialog***
eingebenAuftragsdaten	Beschriftungs-Objekt *Bitte Auftragsdaten eingeben*
	Beschriftungs- und Textfeld-Objekt *Lieferauftragsnr.*
	Beschriftungs- und Kombinationsfeld-Objekt *Konfigurationsvertragsnr.*
	Beschriftungs- und Kombinationsfeld-Objekt *Rahmenvertragsnr.*
	Beschriftungs- und Textfeld-Objekt *Datum*
	Beschriftungs-Objekt *Teileauswahl*
	Beschriftungs- und Listenfeld-Objekt *Teilenummer / Menge*
	Beschriftungs- und Listenfeld-Objekt *Teileauswahl / Menge*
	Knopf-Objekt =>
	Knopf-Objekt <=
	Knopf-Objekt *OK*
	Knopf-Objekt *Abbruch*

Tabelle 6.24: Benutzerschnittstellen-Objekte der JavaBean *Teileabruf-Dialog*

Der für die beschriebenen Enterprise Beans erforderliche *Deployment Descriptor* ist in Abbildung A.2 im Anhang enthalten. Wie bereits in Abschnitt 6.1.5 bemerkt, sind dabei insbesondere die Einstellungen der Persistenzeigenschaften und des Transaktionsschutzes für die Entity Beans beachtenswert, deren Auswahl wiederum dem Ausführungsdienst der Ausführungsplattform überlassen wird.

7 Zusammenfassende Bewertung

In der vorliegenden Arbeit wurde ein Software-Architekturmodell entwickelt, das den softwaretechnischen Entwurf wiederverwendbarer und verteilbarer betrieblicher Anwendungssysteme durch den Einsatz von Software-Komponenten unterstützt. Dabei entsprechen die verwendeten Software-Komponenten einem Komponentenkonzept, das Teil eines in der Arbeit formulierten präskriptiven Grundkonzepts der Komponentenorientierung ist. Zur Erstellung dieses Grundkonzepts wurden Anforderungen ausgearbeitet, die an Entwicklungserzeugnisse und an das Vorgehen im softwaretechnischen Entwurf und in der Implementierung gerichtet sind und die sich aus der geforderten Wiederverwendbarkeit und Verteilbarkeit von Anwendungssystemen und -teilsystemen ergeben.

Sowohl das methodisch fundierte präskriptive Grundkonzept der Komponentenorientierung als auch die Entwicklung eines darauf aufbauenden Software-Architekturmodells tragen zur Lösung der in der Einleitung beschriebenen Problemstellung bei.

In Kapitel 2 folgte zunächst eine konzeptionelle Bestimmung des softwaretechnischen Entwurfs betrieblicher Anwendungssysteme. Dafür wurde die Entwicklung eines gesamten betrieblichen Informationssystems als Konstruktionsaufgabe interpretiert und daraufhin der softwaretechnische Entwurf als eine Teilaufgabe identifiziert. In diesem Zusammenhang zeigte sich die Bedeutung von Architekturmodellen zur Unterstützung einer ganzheitlichen und methodisch durchgängigen Entwicklung von betrieblichen Informationssystemen im Allgemeinen und betrieblichen Anwendungssystemen im Speziellen. Anschließend wurde ein generischer Architekturrahmen zur Gestaltung von Informations- und Anwendungssystemarchitekturen eingeführt, der im weiteren Verlauf der Arbeit als Grundlage für die Konzeption des Software-Architekturmodells diente. Auf diese Weise konnte sichergestellt werden, dass sich das konzipierte Software-Architekturmodell in unterschiedliche Informationssystem-Architekturen, die ebenfalls Ausprägungen des generischen Architekturrahmens sind, einordnen lässt.

Kapitel 3 diente hauptsächlich der Bestimmung von Anforderungen, die aufgrund der Formalziele Wiederverwendung, Wiederverwendbarkeit und Verteilbarkeit an Entwicklungserzeugnisse und an das Vorgehen im softwaretechnischen Entwurf gestellt werden. Allerdings wurden aufgrund einer notwendigen Durchgängigkeit beim Übergang vom softwaretechnischen Entwurf zur Implementierung nur solche Entwicklungserzeugnisse betrachtet, die sowohl im softwaretechnischen Entwurf als auch in der Implementierung exisitieren und dabei auf dem gleichen Grundkonzept basieren. Da im Rahmen der Bestimmung insbesondere auf eine Orthogonalität der Anforderungen geachtet wurde, konnten sie zur folgenden Bewertung der Wiederverwen-

dungs-, Wiederverwendbarkeits- und Verteilbarkeitsunterstützung häufig genutzter Erzeugnisarten und zur methodischen Fundierung des in Kapitel 4 formulierten präskriptiven Grundkonzepts der Komponentenorientierung herangezogen werden.

Im Anschluss an die Bestimmung der Anforderungen wurden Prinzipien und Modelle vorgestellt, deren Beiträge zur Unterstützung der Anforderungen bereits mehrfach nachgewiesen wurden und die deswegen unumstritten sind. Auch sie flossen in das präskriptive Grundkonzept der Komponentenorientierung ein.

Die bereits angesprochene Bewertung der häufig genutzten Erzeugnisarten objektorientierte Klasse und Framework hinsichtlich ihrer jeweiligen Wiederverwendungs-, Wiederverwendbarkeits- und Verteilbarkeitsunterstützung ergab, dass keine Erzeugnisart den aufgestellten Anforderungen vollständig begegnen kann. Die Erzeugnisart Software-Komponente konnte nicht bewertet werden, da für sie noch kein einheitliches und untersuchbares Grundkonzept vorliegt.

Deswegen befasste sich das Kapitel 4 mit der Entwicklung eines präskriptiven Grundkonzepts der Komponentenorientierung, das auf den erarbeiteten Anforderungen und den vorgestellten Prinzipien und Modellen beruht.

Ein wesentliches Merkmal des Grundkonzepts ist die Differenzierung zwischen der Außensicht einer Software-Komponente, in der ihre Implementierung nicht offengelegt wird, und ihrer Innensicht. Im *Development with Reuse* stehen nur die Außensichten der Software-Komponenten zur Verfügung. Dies erleichtert ihre Wiederverwendung. Die zugehörigen Innensichten werden im *Development for Reuse* objektorientiert modelliert. Dies ermöglicht zum einen die Nutzung der unbestrittenen Vorteile des objektorientierten Paradigmas, die es gegenüber anderen Entwicklungsparadigmen aufweist, und zum anderen einen lückenlosen Übergang vom objektorientierten Fachentwurf eines Anwendungssystems zu dessen komponentenorientiertem Software-Entwurf.

Die weiteren Merkmale des präskriptiven Grundkonzepts der Komponentenorientierung wurden anhand der Dimensionen Beschreibungsobjekt, Entwicklungsphase, Abstraktionsebene und Systemsicht systematisiert. Dadurch konnte eine in der Breite und Tiefe hinreichende und vollständige Beschreibung des Grundkonzepts gewährleistet werden.

Das präskriptive Grundkonzept der Komponentenorientierung diente schließlich, zusammen mit dem in Kapitel 2 vorgestellten generischen Architekturrahmen und dem in Kapitel 3 eingeführten Modell der Nutzer- und Basismaschine, als Grundlage für die in Kapitel 5 durchgeführte Konzeption eines komponentenbasierten Software-Architekturmodells zur Entwicklung betrieblicher Anwendungssysteme.

Gemäß dem Grundkonzept der Komponentenorientierung besteht das komponentenbasierte Software-Architekturmodell aus einer Modellebene für die Außenperspektive der Struktur- und Verhaltensmerkmale eines komponentenbasierten Anwendungssystems und einer Modellebene für die zugehörige Innenperspektive. Auf der Modellebene der Außenperspektive findet eine komponentenorientierte, auf der Modellebene der Innenperspektive eine objektorientierte Modellierung statt. Beim *Development with Reuse* wird nur die Modellebene der Außenperspektive genutzt. Dies trägt zu einer erhöhten Wiederverwendbarkeit der Software-Komponenten bei.

Als methodischer Beschreibungsrahmen stehen auf beiden Modellebenen entsprechende Metamodelle und Metaphern zur Verfügung. Zusätzlich werden die Metamodelle auf jeder Modellebene in ein eigentliches Kern-Metamodell und ein zugehöriges Repräsentations-Metamodell zur standardisierten und geeignet formalisierten Darstellung der Modellierungsergebnisse differenziert. Als Repräsentationssprachen wurden die UML, die IDL und die OCL gewählt. Sie tragen aufgrund ihrer Standardisierung, ihrer vergleichweise leichten Erlernbarkeit und ihrer weiten Verbreitung zu einem verbesserten Verständnis und folglich zu einer Erhöhung der Wiederverwendbarkeit der Modellierungsergebnisse bei. Außerdem ermöglicht ihr Einsatz eine breitere Akzeptanz des gesamten Software-Architekturmodells.

Aus Gründen der Komplexitätsbewältigung sind auf beiden Modellebenen jeweils eine Struktursicht und eine Verhaltenssicht als Projektionen auf die jeweiligen Metamodelle definiert. Die Modellebene der Innenperspektive weist darüber hinaus eine funktionale und eine technische Sicht auf. Anhand dieser sind die technischen und die funktionalen Belange getrennt voneinander spezifizierbar.

Auf beiden Modellebenen wird die Modellierung durch geeignete Muster unterstützt. Die gewählte strukturierte Form der Musterbeschreibung erhöht die Verständlichkeit und Anwendbarkeit des damit ausgedrückten heuristischen Entwurfswissens.

Ein speziell für die Außenperspektive-Modellebene formuliertes Muster strukturiert ein komponentenbasiertes Anwendungssystem in Software-Komponenten für die Anwendungsfunktionen, für die Datenhaltung und für die Kommunikation. Zusätzlich werden die Software-Komponenten für die Anwendungsfunktionen anhand eines weiteren Musters in Vorgangs- und Entitäts-Software-Komponenten unterteilt. Beide Muster sind obligatorisch für eine bestmögliche Wiederverwendung und Verteilung der zu erstellenden Software-Komponenten.

Zu den erwähnenswerten Mustern auf der Innenperspektive-Modellebene gehören die der technischen Sicht, die zusammen ein Mustersystem bilden. Mithilfe dieses Mustersystems lässt sich eine für die Wiederverwendbarkeit und Verteilbarkeit von Software-Komponenten erforderliche Ausführungsplattform entwickeln oder auswählen. In diesem Zusammenhang wurden auch Konzepte für Basismaschinen, die sich

grundsätzlich als Ausführungsplattformen für komponentenbasierte Anwendungssysteme eignen, kurz eingeführt.

Zur Verbindung der Außenperspektive- mit der Innenperspektive-Modellebene und zur Verknüpfung des Software-Architekturmodells mit der Modellebene der fachlichen Anwendungssystemspezifikation einer übergreifenden Informationssystem-Architektur stehen geeignete Beziehungs-Metamodelle und Beziehungsmuster zur Verfügung. Voraussetzung für eine Verknüpfung des Software-Architekturmodells mit der fachlichen Modellebene einer Informationssystem-Architektur ist eine objektorientierte Anwendungssystemspezifikation, die aus Vorgangsklassen und konzeptuellen Klassen besteht. Ferner müssen die konzeptuellen Klassen der fachlichen Anwendungssystemspezifikation nach ihren Existenzabhängigkeiten strukturiert sein. Sind diese Voraussetzungen erfüllt, lassen sich anhand von entsprechenden Beziehungs-Mustern aus Vorgangsklassen Vorgangs-Software-Komponenten und aus konzeptuellen Klassen Entitäts-Software-Komponenten methodisch herleiten. Diese determinieren daraufhin den Entwurf der notwendigen Kommunikations- und Datenhaltungs-Software-Komponenten.

Die einfache und lückenlose Integrierbarkeit des Software-Architekturmodells in eine übergreifende Informationssystem-Architektur konnte am Beispiel der in Kapitel 2 eingeführten SOM-Unternehmensarchitektur demonstriert werden.

Schließlich folgte in Kapitel 6 ein erster Nachweis der praktischen Anwendbarkeit des komponentenbasierten Software-Architekturmodells anhand von zwei Fallstudien aus unterschiedlichen betrieblichen Domänen. Dabei sollten insbesondere die Wiederverwendbarkeit und Verteilbarkeit der mit dem Software-Architekturmodell erstellten Software-Komponenten sowie die lückenlose Integration des Software-Architekturmodells in eine übergreifende Informationssystem-Architektur nachgewiesen werden.

Zum Nachweis der lückenlosen Integration wurde jede Fallstudie zunächst vom Unternehmensplan über die zugehörigen Geschäftsprozesse bis zur objektorientierten fachlichen Anwendungssystemspezifikation mit der SOM-Methodik modelliert und das Ergebnis der fachlichen Anwendungssystemspezifikation als Ausgangspunkt für den komponentenbasierten softwaretechnischen Entwurf mithilfe des Software-Architekturmodells genutzt.

Die Wiederverwendbarkeit der Software-Komponenten ließ sich an einer Menge von geeignet abgegrenzten Exemplaren aus der ersten Fallstudie zeigen, die in der zweiten Fallstudie unverändert eingesetzt werden konnten.

Der Nachweis der Verteilbarkeit erfolgte für jede Fallstudie anhand der Implementierung eines Ausschnitts aus dem zugehörigen softwaretechnischen Entwurf. Da zur

Implementierung jedoch noch keine Programmiersprache verfügbar war, die dem präskriptiven Grundkonzept der Komponentenorientierung vollständig entsprach, musste auf eine objektorientierte Programmiersprache zurückgegriffen werden. Somit entstand ein Bruch beim Übergang vom softwaretechnischen Entwurf zur Implementierung der ausgewählten Software-Komponenten. Deswegen ist die Entwicklung einer durchgängig komponentenorientierten Programmiersprache für einen lückenlosen Übergang vom komponentenbasierten Software-Architekturmodell zur Ebene der Implementierung zwingend erforderlich.

Grundsätzlich sind weitere Einsätze des komponentenbasierten Software-Architekturmodells in praxisrelevanten Entwicklungsprojekten für ausgedehntere Evaluationen und für dessen kontinuierliche Weiterentwicklung wünschenswert.

Literaturverzeichnis

[AlCM02] Alur D., Crupi J., Malks D.: Core J2EE Patterns – Die besten Praxislösungen und Design-Strategien. Markt+Technik, München 2002.

[Alex77] Alexander Ch.: A Pattern Language. Oxford University Press, New York 1977

[Alex79] Alexander Ch.: The Timeless Way Of Building. Oxford University Press, New York 1979

[Ambe99a] Amberg M.: Betriebliches Anwendungssystem. In: [BrSp99], S. 28-29.

[Ambe99b] Amberg M.: Standard-Software. In: [BrSp99], S. 706.

[BaCK98] Bass L., Clements P., Kazman R.: Software Architecture in Practice. Addison-Wesley Longman, Boston 1998.

[BaHH00] Barroca L., Hall J., Hall P.: An Introduction and History of Software Architectures, Components, and Reuse. In: Barroca L., Hall J., Hall P. (Hrsg.): Software Architectures – Advances and Applications. Springer, London 2000, S. 1-11.

[Balz82] Balzert H.: Die Entwicklung von Software-Systemen – Prinzipien, Methoden, Sprachen, Werkzeuge. Bibliographisches Institut, Zürich 1982.

[Balz98] Balzert H.: Lehrbuch der Software-Technik, Band 2. Spektrum, Heidelberg 1998.

[Barn02] Barnaby T.: Distributed .NET Programming in C#. Apress, Berkeley 2002.

[Bass01] Bass L.: Software Architecture Design Principles. In: [HeCo01], S. 389-403.

[BCF+03] Booth D., Champion M., Ferris Ch., McCabe F., Newcomer E., Orchard D. (Hrsg.): Web Services Architecture. W3C Working Draft, 14 May 2003. http://www.w3.org/TR/2003/WD-ws-arch-20030514/WD-ws-arch-20030514.html, Abruf am 2003-05-25.

[BeMS98] Bernus P., Mertins K., Schmidt G. (Hrsg.): Handbook on Architectures of Information Systems. Springer, Berlin 1998.

[BeMW02] Beimborn D., Mintert S., Weitzel T.: Web Services und ebXML. In: Wirtschaftsinformatik 44 (2002) 3, S. 277-280.

[Beng00] Bengel G.: Verteilte Systeme. Vieweg, Braunschweig 2000.

[BeSc98] Bernus P., Schmidt G.: Architectures of Information Systems. In: [BeMS98], S. 1-9.

[Bigg98] Biggerstaff T.J.: A Perspective of Generative Reuse. In: Annals of Software Engineering 5 (1998) o. Nr., S. 169-226.

[BiPe89] Biggerstaff T. J., Perlis A. J. (Hrsg.): Software Reusability. Volume I: Concepts an Models. ACM Press, New York 1989.

[BiRi89] Biggerstaff T. J., Richter C.: Reusability Framework, Assessment, and Directions. In: [BiPe89], S. 1-17.

[BJP+99] Beugnard A., Jézéquel J.-M., Plouzeau N., Watkins D.: Making Components Contract Aware. In: IEEE Computer 32 (1999) 7, S. 38-45.

[Blev01] Blevins D.: Overview of the Enterprise JavaBeans Component Model. In: [HeCo01], S. 589-606.

[BMR+98] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.: Pattern-orientierte Software-Architektur – Ein Pattern-System. Addison-Wesley, München 1998.

[BMR+03a] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.: Pattern: Microkernel. http://www.vico.org/pages/PatronsDisseny/Pattern MicroKernel/index.html, Abruf am 2003-07-31.

[BMR+03b] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.: Pattern: Broker. http://www.vico.org/pages/PatronsDisseny/Pattern Broker/index.html, Abruf am 2003-07-31.

[BoBL76] Boehm B. W., Brown J. R., Lipow M.: Quantitative Evaluation of Software Quality. In: Proceedings of the Second International Conference on Software Engineering, San Francisco 1976, S. 592-605.

[BöFu02] Böhm R., Fuchs E.: System-Entwicklung in der Wirtschaftsinformatik. 5. Aufl., vdf, Zürich 2002.

[Booc94] Booch G.: Object-oriented analysis and design with applications. 2. Aufl., Benjamin/Cummings, Redwood City 1994.

[BrBu00] Brereton P., Budgen D.: Component-Based Systems: A Classification of Issues. In: IEEE Computer 33 (2000) 11, S. 54-62.

[Brit01] Britton Ch.: IT Architectures and Middleware – Strategies for Building Large, Integrated Systems. Addison-Wesley, Boston 2001.

[Broc86] Brockhaus-Enzyklopädie. Band 1. 19. Aufl., Brockhaus, Mannheim 1986.

[BrSi00] Brössler P., Siedersleben J. (Hrsg.): Softwaretechnik. Hanser, München 2000.

[BrSi02] Broy M., Siedersleben J.: Objektorientierte Programmierung und Softwareentwicklung. In: Informatik-Spektrum 25 (2002) 1, S. 3-11.

[BrSp99] Broy M., Spaniol O. (Hrsg.): Informatik und Kommunikationstechnik. 2. Aufl., Springer, Berlin 1999.

[BrWa98] Brown A. W., Wallnau K. C.: The Current State of CBSE. In: IEEE Software 15 (1998) 5, S. 37-46.

[CaLo00] Carney D., Long F.: What Do You Mean by COTS? Finally, a Useful Answer. In: IEEE Software 17 (2000) 2, S. 83-86.

[CaWe85] Cardelli L., Wegner P.: On Understanding Types Data Abstraction, and Polymorphism. In: Computing Surveys 17 (1985) 4, S. 471-522.

[CGM+03] Chinnici R., Gudgin M., Moreau J.-J., Weerawarana S.: Web Services Description Language (WSDL), Version 1.2. W3C Working Draft, 3 March 2003. http://www.w3.org/TR/2003/WD-wsdl12-20030303/WD-wsdl12-20030303.html, Abruf am 2003-05-25.

[CHJK02] Crnkovic I., Hnich B., Jonsson T., Kiziltan Z.: Specification, Implementation, and Deployment of Components – Clarifying Common Terminology and Exploring Component-Based Relationships. In: Communications of the ACM 45 (2002) 10, S. 35-40.

[Clem01] Clements P. C.: From Subroutines to Subsystems: Component-Based Software Development. In: [HeCo01], S. 189-198.

[CoDK01] Coulouris G, Dollimore J., Kindberg T.: Distributed Systems – Concepts and Design. 3. Aufl., Addison-Wesley, Harlow 2001.

[CoHe01] Councill B., Heinemann G. T.: Definition of a Software Component and Its Elements. In: [HeCo01], S. 5-19.

[Copl91] Coplien J.O.: Advanced C++ Programming Styles an Idioms. Addison-Wesley, Reading MA 1991

[CoSc95] Coplien J.O., Schmidt D.C. (Hrsg.): Pattern Languages of Program Design. Addison-Wesley, Reading 1995.

[CoTu00] Conrad S., Turowski K.: Vereinheitlichung der Spezifikation von Fachkomponenten auf der Basis eines Notationsstandards. In: Ebert J., Frank U. (Hrsg.): Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik. Beiträge des Workshops „Modellierung 2000“, St. Goar, 5. - 7. April 2000. Fölbach, Koblenz 2000, S. 179-193.

[CoYo79] Constantine L., Yourdon E.: Structured Design. Prentice Hall, Englewood Cliffs 1979.

[Dada96] Dadam P.: Verteilte Datenbanken und Client/Server-Systeme. Springer, Berlin 1996.

[DeMe01] Depke R., Mehner K.: „Separation of Concern" mit Rollen, Subjekten und Aspekten. In: Mehner K., Mezini M., Pulvermüller E., Speck A. (Hrsg.): Aspektorientierung. Workshop der GI-Fachgruppe 2.1.9, Objektorientierte Software-Entwicklung, Paderborn 2001, S. 1-7.

[Dene91] Denert E.: Software-Engineering – Methodische Projektabwicklung. Springer, Berlin 1991.

[DePe02] Denninger S., Peters I.: Enterprise JavaBeans 2.0. 2. Aufl., Addison-Wesley, München 2002.

[DeRo99] Della Torre Cicalese C., Rotenstreich S.: Behavioral Specification of Distributed Software Component Interfaces. In: IEEE Computer 32 (1999) 7, S. 46-53.

[Digr98] Digre T.: Business Object Component Architecture. In: IEEE Software 15 (1998) 5, S. 60-69.

[Eise98] Eisenecker U. W.: Generative Programmierung – Ein neues Paradigma der Softwaretechnik. In: HMD – Theorie und Praxis der Wirtschaftsinformatik 35 (1998) 204, S. 76-83.

[ElFB01] Elrad T., Filman R. E., Bader A.: Aspect-Oriented Programming. Communications of the ACM 44 (2001) 10, S. 29-32.

[Endr88] Endres A.: Software-Wiederverwendung: Ziele, Wege und Erfahrungen. In: Informatik-Spektrum 11 (1988) 2, S. 85-96.

[Ensl78] Enslow P. H.: What is a 'Distributed' Data Processing System? In: IEEE Computer 11 (1978) 1, S. 13-21.

[Fers79] Ferstl O. K.: Konstruktion und Analyse von Simulationsmodellen. Hain, Königstein 1979.

[Fers92] Ferstl O.: Integrationskonzepte Betrieblicher Anwendungssysteme. Fachberichte Informatik 1/92, Universität Koblenz-Landau, Koblenz 1992.

[FeSi84] Ferstl O. K., Sinz E. J.: Software-Konzepte der Wirtschaftsinformatik. De Gruyter, Berlin 1984.

[FeSi94] Ferstl O. K., Sinz E. J.: Multi-Layered Development of Business Process Models and Distributed Business Application Systems – An Object-

Oriented Approach. Bamberger Beiträge zur Wirtschaftsinformatik Nr. 20, Universität Bamberg, Bamberg 1994.

[FeSi95] Ferstl O. K., Sinz E. J.: Der Ansatz des Semantischen Objektmodells (SOM) zur Modellierung von Geschäftsprozessen. In: Wirtschaftsinformatik 37 (1995) 3, S. 209-220.

[FeSi98] Ferstl O. K., Sinz E. J.: SOM – Modeling of Business Systems. In: [BeMS98], S. 389-358.

[FeSi01] Ferstl O. K., Sinz E. J.: Grundlagen der Wirtschaftsinformatik. Band 1. 4. Aufl, Oldenbourg, München 2001.

[Fing00] Finger F.: Entwurf und Implementation eines modularen OCL-Compilers. Diplomarbeit am Lehrstuhl Softwaretechnologie an der Technischen Universität Dresden, Dresden 2000.

[Fisc02] Fischer B.: Konzeption eines generischen Beschreibungsrahmens für Komponentenmodelle zur Software-Entwicklung betrieblicher Anwendungssysteme. Diplomarbeit am Lehrstuhl für Wirtschaftsinformatik, insbes. Systementwicklung und Datenbankanwendung der Universität Bamberg, Bamberg 2002.

[Flan00] Flanagan D.: Java in a Nutshell. 3. Aufl., O'Reilly, Köln 2000.

[FlZü02] Floyd C., Züllighoven H.: Softwaretechnik. In: [RePo02], S. 763-790.

[FoBa01] Foegen M., Battenfeld J.: Die Rolle der Architektur in der Anwendungsentwicklung. In: Informatik-Spektrum 24 (2001) 5, S. 290-301.

[FoSc00] Fowler M., Scott K.: UML konzentriert – Eine strukturierte Einführung in die Standard-Objektmodellierungssprache. 2. Aufl., Addison-Wesley, München 2000.

[Fowl96] Fowler M.: Analysis Patterns – Reusable Object Models. Addison-Wesley, Reading 1996

[Fran99] Frank U.: Componentware – Software-technische Konzepte und Perspektiven für die Gestaltung betrieblicher Informationssysteme. In: Information Management & Consulting 14 (1999) 2, S. 11-18.

[FrLS00] Frühauf K., Ludewig J., Sandmayr H.: Software-Projektmanagement und -Qualitätssicherung. 3. Aufl., vdf, Zürich 2000.

[FSH+97] Ferstl O.K., Sinz E.J., Hammel C., Schlitt M., Wolf St.: Bausteine komponentenbasierter Anwendungssysteme. In: HMD – Theorie und Praxis der Wirtschaftsinformatik 34 (1997) 197, S.24-46

[FSH+98] Ferstl O. K., Sinz E. J., Hammel Ch., Schlitt M., Wolf S., Popp K., Kehlenbeck R., Pfister A., Kniep H., Nielsen N., Seitz A.: WEGA – Wiederverwendbare und erweiterbare Geschäftsprozeß- und Anwendungssystemarchitekturen. Abschlussbericht, Walldorf 1998.

[Geig02] Geiger J.: Aspekt- und subjektorientiertes Programmieren. http://www.informatik.uni-stuttgart.de/ifi/ps/Lehre/HS_Produktlinien/jg-finale.pdf, Abruf am 2002-11-04.

[GHVJ96] Gamma E., Helm R., Johnson R., Vlissides J.: Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley, München 1996.

[Gies00] Giese H.: Contract-based Component System Design. In: Proceedings of the Hawai'i International Conference On System Sciences, Maui, 4.-7. Januar 2000, Maui 2000, S. 1-10.

[Grif98] Griffel F.: Componentware – Konzepte und Techniken eines Softwareparadigmas. 1. Aufl., Dpunkt, Heidelberg 1998.

[GrRe93] Gray J., Reuter A.: Transaction Processing: Concepts and Techniques. Morgan Kaufman, San Francisco 1993.

[GrTh00] Gruhn V., Thiel A.: Komponentenmodelle – DCOM, JavaBeans, Enterprise JavaBeans, CORBA. Addison-Wesley, München 2000.

[Haas01] Haase O.: Kommunikation in verteilten Anwendungen – Einführung in Sockets, Java RMI, CORBA und Jini. Oldenbourg, München 2001.

[HaMe02] Hau M., Mertens P.: Computergestützte Auswahl komponentenbasierter Anwendungssysteme. In: Informatik-Spektrum 25 (2002) 5, S. 331-340.

[Hamm99] Hammel Ch.: Generische Spezifikation betrieblicher Anwendungssysteme. Shaker, Aachen 1999.

[HaNe01] Hansen H. R., Neumann G.: Wirtschaftsinformatik I. 8. Aufl., Lucius & Lucius, Stuttgart 2001.

[HeCo01] Heineman G. T., Councill W. T. (Hrsg.): Component-Based Software Engineering – Putting the Pieces Together. Addison-Wesley, Boston 2001.

[Hein02] Heinrich L. J.: Grundlagen der Wirtschaftsinformatik. In: [RePo02], S. 1039-1054.

[HeRo98] Heinrich L. J., Roithmayr F.: Wirtschaftsinformatik-Lexikon. 6. Aufl., Oldenbourg, München 1998.

[HeSi02] Heinrich L. J., Sinz E. J.: Wirtschaftsinformatik. In: [RePo02], S. 1037-1038.

[Hill03] Hillside.net (Hrsg.): Patterns Home Page – Your Patterns Library. http://www.hillside.net/patterns/index.html, Abruf am 2003-07-13.

[HoNo01] Houston K., Norris D.: Software Components and the UML. In [HeCo01], S. 243-262.

[Hopk00] Hopkins J.: Component Primer. In: Communications of the ACM 43 (2000) 10, S. 27-30.

[HuDF00] Hussmann H., Demuth B., Finger F.: Modular Architecture for a Toolset Supporting OCL. Arbeitsbericht des Lehrstuhls Softwaretechnologie an der Technischen Universität Dresden, Dresden 2000

[IBM02] IBM (Hrsg.): IBM Business Components for WebSphere Application Server – SanFrancisco to WSBC Migration – Overview. ftp://ftp.software.ibm.com/software/websphere/awdtools/businesscomponents/wsbcforsfcustomers.pdf, Abruf am 2003-06-26.

[ISO01] International Standardisation Organization (Hrsg.): ISO/IEC9126-1 (2001): Part I - Quality Model. Genf 2001.

[JäHe02] Jähnichen S., Herrmann S.: Was, bitte, bedeutet Objektorientierung? In: Informatik-Spektrum 25 (2002) 4, S. 266-276.

[JCJÖ92] Jacobson I., Christerson M., Jonsson P., Övergaard G.: Object-Oriented Software Engineering – A Use Case Driven Approach. Addison-Wesley, Workingham 1992.

[JeMe02] Jeckle M., Meier E.: Web-Services – Standards und ihre Anwendungen. In: JavaSPEKTRUM o. Jg. (2002) 1, S. 14-23.

[JoFo88] Johnson R. E., Foote B.: Designing reusable classes. In: Journal of Object-Oriented Programming 1 (1988) 2, S. 22-35.

[John97] Johnson R. E.: Components, Frameworks, Patterns (extended abstract). In: ACM Software Engineering Notes 22 (1997) 3, S. 10-17

[Kell99] Keller G.: SAP R/3 prozeßorientiert anwenden – Iteratives Prozeß-Prototyping mit Ereignisgesteuerten Prozeßketten und Knowledge Maps. 3. Aufl., Addison-Wesley, Bonn 1999.

[KoBo98] Kozaczynski W., Booch G.: Component-Based Software Engineering. In: IEEE Software 15 (1998) 5, S. 34-36.

[Kobr03] Kobryn C.: What to Expect From UML 2.0. http://www.sdtimes.com/opinions/guestview_048.htm, Abruf am 2003-01-14.

[Kort01] Korthaus A.: Komponentenbasierte Entwicklung computergestützter betrieblicher Informationssysteme. Peter Lang, Frankfurt a. M. 2001.

[KrPo88] Krasner G. E., Pope S. T.: A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. ParcPlace Systems, Mountain View 1988.

[Küff94] Küffmann K.: Software-Wiederverwendung – Konzeption einer domänenorientierten Architektur. Vieweg, Braunschweig 1994.

[Lars00] Larsen G.: Component-Based Enterprise Frameworks. In: Communications of the ACM 43 (2000) 10, S. 25-26.

[LeHM95] Lehner F., Hildbrand K., Maier R.: Wirtschaftsinformatik – Theoretische Grundlagen. Hanser, München 1995.

[Lehn01] Lehner F.: Wirtschaftsinformatik, Forschungsgegenstände und Erkenntnisverfahren. In: [Mert01], S. 505-507.

[LiSe97] Linssen O., Seidel F.: Löst Componentware das Wiederverwendungsproblem? In: HMD – Theorie und Praxis der Wirtschaftsinformatik 34 (1997) 197, S. 90-97.

[MaOd92] Martin J., Odell J. J.: Object-oriented analysis and design. Prentice Hall, Englewood Cliffs 1992.

[MaOd99] Martin J., Odell J. J.: Objektorientierte Modellierung mit UML: Das Fundament. Prentice Hall, München 1999.

[MaRB98] Martin R., Riehle D., Buschmann F. (Hrsg.): Pattern Languages of Program Design 3. Addison-Wesley, Reading 1998.

[Mari02] Marinescu F.: EJB design patterns – Advanced patterns, processes, and idioms. Wiley & Sons, New York 2002.

[McIl69] McIlroy M.D.: Mass Produced Software Components. In: [Naur69], S. 138-148.

[McMa80] McCall J. L., Matsumoto M. T.: Software Quality Metrics Enhancements. Final Technical Report RADC-TR-80-109, Volume I, Rome Air Development Center, Rome 1980.

[MeHL97] Mertens P., Holzner J., Ludwig P.: Mittelwege zwischen Individual- und Standardsoftware. In: Abramowicz W. (Hrsg.): Business Information Systems – BIS ’97, International Conference, Poznan 1997, S. 15-44.

[Meie00] Meier J.: Projektmodell. In: [BrSi00], S. 5-20.

[MeMi99] Meyer B., Mingins Ch.: Component-Based Development: From Buzz to Spark. In: IEEE Computer 32 (1999) 7, S. 35-38.

[Mert00] Mertens P.: Integrierte Informationsverarbeitung 1: Administrations- und Dispositionssysteme in der Industrie. 12. Aufl., Gabler, Wiesbaden 2000.

[Mert01] Mertens P. (Hrsg.): Lexikon der Wirtschaftsinformatik. 4. Aufl., Springer, Berlin 2001.

[Meye88] Meyers Enzyklopädisches Lexikon. Band 1. 9. Aufl., Bibliographisches Institut, Mannheim 1988.

[Meye90] Meyer B.: Objektorientierte Softwareentwicklung. Hanser, München 1990.

[Meye99] Meyer B.: On To Components. In: IEEE Computer 32 (1999) 1, S. 139-140.

[MiMM95] Mili H., Mili F., Mili A.: Reusing Software: Issues and Research Directions. In: IEEE Transactions on Software Engineering 21 (1995) 6, S. 528-562.

[MiSe97] Mikhajlov L., Sekerinski E.: The Fragile Base Class Problem and Its Solution. Technical Report No 117, Turku Centre for Computer Science, Turku/Åbo 1997.

[Mitr03] Mitra N. (Hrsg.): SOAP Version 1.2 Part 0: Primer. W3C Proposed Recommendation, 07 May 2003. http://www.w3.org/TR/2003/PR-soap12-part0-20030507/PR-soap12-part0-20030507.html, Abruf am 2003-05-25.

[MKR+00] Mantel S., Knobloch B., Rüffer T., Schissler M., Schmitz K., Ferstl O. K., Sinz E. J.: Analyse der Integrationspotenziale von Kommunikationsplattformen für verteilte Anwendungssysteme. FORWIN-Bericht FWM-2000-009, Bamberg 2000.

[Mons01] Monson-Haefel R.: Enterprise JavaBeans. 2. Aufl., O'Reilly, Köln 2001.

[Mühl02] Mühlhäuser M.: Verteilte Systeme. In: [RePo02], S. 673-707.

[Myer02] Myerson J. M.: The Complete Book of Middleware. CRC Press LLC, Boca Raton 2002.

[Naur69] Naur P., Randell B. (Hrsg.): Software Engineering. Report on a conference sponsored by the NATO Science Comitee. Brüssel 1969.

[NiLu97] Nierstrasz O., Lumpe M.: Komponenten, Komponentenframeworks und Gluing. In: HMD – Theorie und Praxis der Wirtschaftsinformatik 34 (1997) 197, S. 8-23.

[NMM+00] Noack J., Mehmanesh H., Mehmaneche H., Zendler A.: Architekturen für Network Computing. In: Wirtschaftsinformatik 42 (2000) 1, S. 5-14.

[NoMe02] Noel G., Meier J.: Einführung in die apektorientierte Programmierung. In: ObjektSPEKTRUM o. Jg. (2002) 5, S. 28-32.

[NoSK97] Noack J., Schienmann B., Kittlaus H.-B.: Ein Leitfaden zur Anwendungsentwicklung in der Sparkassenorganisation. In: OBJEKTspektrum o. Jg. (1997) 6, S. 52-59.

[OeVa99] Oezsu T. M., Valduriez P.: Principles of Distributed Database Systems. 2. Aufl., Prentice Hall, Upper Saddle River 1999.

[OeWe03] Oestereich B., Weilkiens T.: UML 2.0: Alles wird gut? In: OBJEKTspektrum o. Jg. (2003) 1, S. 36-38.

[OMG96] OMG (Hrsg.): Common Facilities RFP-4: Common Business Objects and Business Objects Facility. OMG, Needham 1996

[OMG01] OMG (Hrsg.): Unified Modeling Language Specification, Version 1.4, September 2001. OMG, Needham 2001.

[OMG02a] OMG (Hrsg.): Meta Object Facility (MOF) Specification, Version 1.4, April 2002. OMG, Needham 2002.

[OMG02b] OMG (Hrsg.): The Common Object Request Broker: Architecture and Specification, Version 3.0, July 2002. OMG, Needham 2002.

[OMG03a] OMG (Hrsg.): OMG Specifications and Process: The Big Picture. http://www.omg.org/gettingstarted/overview.htm, Abruf am 2003-06-26.

[OMG03b] OMG (Hrsg.): MDA® Specifications. http://www.omg.org/mda/specs.htm, Abruf am 2003-06-26.

[OrHE96] Orfali R., Harkey D., Edwards J.: The Essential Distributed Objects Survival Guide. John Wiley & Sons, New York 1996.

[PaCW87] Parnas D. L., Clements P. C., Weiss D. M.: The Modular Structure of Complex Systems. In: [Pete87], S. 162-169.

[PaCW89] Parnas D. L., Clements P.C., Weiss D.M.: Enhancing Reusability with Information Hiding. In: [BiPe89], S. 141-157.

[Pane00] Panek, S.: Fachliche Pattern für die Modellierung von Dienstleistungsprozessen am Beispiel einer Personalvermittlung. Diplomarbeit am

Lehrstuhl für Wirtschaftsinformatik, insbes. Systementwicklung und Datenbankanwendung der Universität Bamberg, Bamberg 2000.

[Parn72] Parnas D. L.: On the criteria to be used in decomposing systems into modules. In: Communications of the ACM 15 (1972) 12, S. 1053-1058.

[Parn02] Parnas D. L.: The Secret History of Information Hiding. In: Broy M., Denert E. (Hrsg.): Software Pioneers – Contributions to Software Engineering. Springer, Berlin 2002, S. 398-409

[Pers03] Persson E.: Business Object Components? http://jeffsutherland.org/oopsla2000/persson/persson.html, Abruf am 2003-01-20.

[Pete87] Peterson G. (Hrsg.): Tutorial: Object-Oriented Computing. Volume 2: Implementations. IEEE Computer Society Press, Washington 1987.

[PfSz02] Pfister C., Szyperski C.: Why Objects Are Not Enough. http://www.fit.qut.edu.au/~szypersk/pub/CUC96a.ps.gz, Abruf am 2002-11-08.

[PoBl93] Pomberger G., Blaschek G.: Software Engineering – Prototyping und objektorientierte Software-Entwicklung. Hanser, München 1993.

[Pree95] Pree W.: Design-Patterns for Object-Oriented Software Development. Addison-Wesley, Workingham 1995.

[Pree97] Pree W.: Komponentenbasierte Softwareentwicklung mit Frameworks. Dpunkt, Heidelberg 1997.

[Pres87] Pressman R. S.: Software Engineering – A Practitioner's Approach. 2. Aufl., McGraw-Hill, New York 1987.

[Prie89] Prieto-Díaz R.: Classification of Reusable Modules. In: [BiPe89], S. 99-123.

[Prie93] Prieto-Diaz R.: Status Report: Software Reusability. In: IEEE Software 10 (1993) 3, S. 61-66.

[Putm01] Putman J. R.: Architecting with RM-ODP. Prentice Hall, Upper Saddle River 2001.

[Quib96] Quibeldey-Cirkel K.: Entwurfsmuster. In: Informatik-Spektrum 19 (1996) 6, S. 326-327.

[Raue96] Raue H.: Wiederverwendbare betriebliche Anwendungssysteme: Grundlagen und Methoden ihrer objektorientierten Entwicklung. Deutscher Universitäts-Verlag, Wiesbaden 1996.

[RBP+93] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W.: Objektorientiertes Modellieren und Entwerfen. Hanser, Prentice Hall, München, London 1993.

[RePo02] Rechenberg P., Pomberger G. (Hrsg.): Informatik-Handbuch. 3. Aufl., Hanser, München 2002.

[Risi98] Rising L. (Hrsg.): The Patterns Handbook. Cambridge University Press, New York 1998.

[Saak93] Saake G.: Objektorientierte Spezifikation von Informationssystemen. Teubner, Stuttgart 1993.

[Same97] Sametinger J.: Software Engineering with Reusable Components. Springer, Berlin 1997.

[Sche98a] Scheer A.W.: ARIS – Vom Geschäftsprozeß zum Anwendungssystem. 3. Aufl., Springer, Berlin 1998.

[Sche98b] Scheer A.W.: ARIS – Modellierungsmethoden, Meta-Modelle, Anwendungen. 3. Aufl., Springer, Berlin 1998.

[Schm02] Schmidt, D. (Hrsg.): Pattern-orientierte Software-Architektur – Muster für nebenläufige und vernetzte Objekte. Dpunkt, Heidelberg 2002.

[Schr01] Schryen G.: Komponentenorientierte Softwareentwicklung in Softwareunternehmen: Konzeption eines Vorgehensmodells zur Einführung und Etablierung. Deutscher Universitäts-Verlag, Wiesbaden 2001.

[Schu01] Schulze D.: Grundlagen der wissensbasierten Konstruktion von Modellen betrieblicher Systeme. Shaker, Aachen 2001.

[ScLi99] Schmitzer B., Ließmann H.: Entwicklung von Komponenten für das betriebliche Rechnungswesen auf Basis des IBM SanFrancisco Frameworks – ein Erfahrungsbericht. In: Turowski K. (Hrsg.): Tagungsband zum 1. Workshop Komponentenorientierte betriebliche Anwendungssysteme (WKBA 1), Otto-von-Guericke-Universität Magdeburg, 30. März 1999. Magdeburg 1999, S. 75-87.

[ScNZ97] Scheer A. W., Nüttgens M., Zimmermann V.: Objektorientierte Ereignisgesteuerte Prozesskette: Methode und Anwendung. In: Scheer A. W. (Hrsg.): Veröffentlichungen des Instituts für Wirtschaftsinformatik. Heft Nr. 141, Saarbrücken 1997.

[Seib01] Seibt D.: Anwendungssystem. In: [Mert01], S. 46-47.

[Shaw87] Shaw M.: Abstraction Techniques in Modern Programming Languages. In: [Pete87], S. 146-161.

[ShGa96] Shaw M., Garlan D.: Software Architectures – Perspective on an Emerging Discipline. Prentice Hall, Upper Saddle River 1996.

[SiMe00] Siedersleben J., Meyer M.: Software-Architektur. In: [BrSi00], S. 95-121.

[Simo62] Simon H. A.: The Architecture of Complexity. In: Proceedings of the American Philosophical Society, Philadelphia 1962, S. 467-482.

[Sinz88] Sinz E. J.: Das Strukturierte Entity-Relationship-Modell (SER-Modell). In: Angewandte Informatik 30 (1988) 5, S. 191-202.

[Sinz92] Sinz E. J.: Datenmodellierung im Strukturierten Entity-Relationship-Modell (SERM). In: Müller-Ettrich G. (Hrsg.): Fachliche Analyse von Informationssystemen. Addison-Wesley, Bonn 1992.

[Sinz96] Sinz E. J.: Ansätze zur fachlichen Modellierung betrieblicher Informationssysteme – Entwicklung, aktueller Stand und Trends. In: Heilmann H., Heinrich L. J., Roithmayr F. (Hrsg.): Information Engineering. Wirtschaftsinformatik im Schnittpunkt von Wirtschafts-, Sozial- und Ingenieurwissenschaften. Oldenbourg, München 1996, S. 123-143.

[Sinz99a] Sinz E. J.: Architektur betrieblicher Informationssysteme. In: [BrSp99], S. 32-33

[Sinz99b] Sinz E. J.: Automatisierung betrieblicher Aufgaben. In: [BrSp99], S. 54.

[Sinz99c] Sinz E. J.: Betriebliches Informationssystem. In: [BrSp99], S. 349.

[Sinz99d] Sinz E. J.: Modellierung betrieblicher Informationssysteme. In: [BrSp99], S. 458-459.

[Sinz99e] Sinz E. J.: Software-Engineering betrieblicher Anwendungssysteme. In: [BrSp99], S. 667-669.

[Sinz99f] Sinz E. J.: Wirtschaftsinformatik. In: [BrSp99], S. 803-804.

[Sinz99g] Sinz E. J.: Anwendungssysteme aus fachlichen Komponenten. In: Wirtschaftsinformatik 41 (1999) 1, S. 3.

[Sinz01a] Sinz E. J.: Modell. In: [Mert01], S. 311-312.

[Sinz01b] Sinz E. J.: Modellierung. In: [Mert01], S. 312-313.

[Sinz02a] Sinz E. J.: Architektur von Informationssystemen. In: [RePo02], S. 1055-1068.

[Sinz02b] Sinz E. J.: Konstruktion von Informationssystemen. In: [RePo02], S. 1069-1084.

[Sinz02c] Sinz E. J.: EbIS-1 - Modellierung von Informationssystemen. Unterlagen zur gleichnamigen Vorlesung an der Universität Bamberg 2002.

[Somm01] Sommerville I.: Software Engineering. 6. Aufl., Pearson Studium, München 2001.

[Spie99] Spies P. P.: System. In: [BrSp99], S. 722-723.

[SSRB02] Schmidt D. C., Stal M., Rohnert H., Buschmann F.: Pattern-orientierte Software-Architektur: Muster für nebenläufige und vernetzte Objekte. 1. Aufl., Dpunkt, Heidelberg 2002

[Stal02] Stal M.: Web Services: Beyond Component-Based Computing. In: Communications of the ACM 45 (2002) 10, S. 71-76.

[Stri92] Stritzinger A.: Reusable Software Components and Application Frameworks: Concepts, Design Principles and Implications. Dissertationen der Johannes Kepler-Universität Linz, Bd. 91, VWGÖ, Wien 1992.

[StWo01] Stafford J. A., Wolf A. L.: Software-Architecture. In: [HeCo01], S. 371-387.

[SUN97] Sun Microsystems (Hrsg.): JavaBeans™, Version 1.01-A. Sun Microsystems, Inc., Mountain View 1997.

[SUN02a] Sun Microsystems (Hrsg.): Enterprise JavaBeans™ Specification, Version 2.1. Sun Microsystems, Inc., Santa Clara 2002.

[SUN02b] Sun Microsystems (Hrsg.): Java™ 2 Enterprise Edition Specification, v1.4. Sun Microsystems, Inc., Santa Clara 2002.

[Szyp98] Szyperski C.: Component Software – Beyond Object-Oriented Programming. Addison-Wesley, Harlow 1998.

[Szyp02] Szyperski C.: Component-Oriented Programming. A Refined Variation on Obect-Oriented Programming. http://www.oberon.ch/resources/component_software/cop.html, Abruf am 2002-11-08.

[TaSt02] Tanenbaum A. S., van Steen M.: Distributed Systems – Principles and Paradigms. Prentice-Hall, Upper Saddle River 2002.

[Trac01] Tracz W.: COTS Myths an Other Lessons Learned in Component-Based Software Development. In: [HeCo01], S. 99-111.

[Turo02] Turowski K. (Hrsg.): Vereinheitlichte Spezifikation von Fachkomponenten. Memorandum des Arbeitskreises 5.10.3 Komponentenorientierte betriebliche Anwendungssysteme. Universität Augsburg, Augsburg 2002.

[VlCK96] Vlissides J.M., Coplien J.O., Kerth N.L. (Hrsg.): Pattern Languages of Program Design 2. Addison-Wesley, Reading 1996.

[VöSW02] Völter M., Schmid A., Wolff E.: Server Component Patterns – Component Infrastructures Illustrated with EJB. John Wiley & Sons, Chichester 2002.

[VöSW03] Völter M., Schmid A., Wolff E.: Kontinent der Komponenten. Serverseitige Komponenteninfrastrukturen – Ein Überblick. In: Javamagazin o. Jg. (2003) 1, S. 37-43.

[Wegn87] Wegner P.: Dimensions of Object-Based Language Design. In: OOPSLA'87 Proceedings, ACM SIGPLAN Notes 22 (1987) 12, S. 168-182.

[Wegn89] Wegner P.: Capital-Intensive Software Technology. In: [BiPe89], S. 43-97.

[WeSa01] Weinreich R., Sametinger J.: Component Models and Component Services: Concepts and Principles. In: [HeCo01], S. 33-48.

[Wesk99] Weske M.: Business-Objekte: Konzepte, Architekturen, Standards. In: Wirtschaftsinformatik 41 (1999) 1, S. 4-11.

[Whit02] Whitehead K.: Component-Based Development: Principles and Planning for Business Systems. Addison-Wesley, London 2002.

[WiJo90] Wirfs-Brock R. J., Johnson R. E.: Surveying current research in object-oriented design. In: Communications of the ACM 33 (1990) 9, S. 104-124.

[Will02] Williams T.: On Inheritance: What It Means and How To Use It. Draft. http://research.microsoft.com/comapps/docs/Inherit.doc, Abruf am 2002-10-31.

[Wirt71] Wirth N.: Program Development by Stepwise Refinement. In: Communications of the ACM 14 (1971) 4, S. 221-227.

[WiWW90] Wirfs-Brock R., Wilkerson B., Wiener L.: Designing Object-Oriented Software. Prentice Hall, Englewood Cliffs 1990.

[WKWI94] Wissenschaftliche Kommission Wirtschaftsinformatik: Profil der Wirtschaftsinformatik. In: Wirtschaftsinformatik 36 (1994) 1, S. 80-81.

[Wolf96] Wolffram J.: „Design by Contract“: Verträge schaffen Qualität. In: OBJEKTspektrum o. Jg. (1996) 6, S. 52-59.

[ZiBe00] Zimmermann J., Beneken G.: Verteilte Komponenten und Datenbankanbindung. Addison-Wesley, München 2000.

[Zimm99a] Zimmermann F.-O.: Betriebliche Informationssysteme in virtuellen Organisationen. Deutscher Universitäts-Verlag, Wiesbaden 1999.

[Zimm99b] Zimmermann V.: Montage von Softwarekomponenten mit Prozeßmodellen. In: Information Management & Consulting 14 (1999) 2, S. 47-55.

Anhang

Betriebliche Aufgaben			
Betriebliches Objekt	**Betriebliche Aufgabe**	**Automatisierbarkeit**	**Automatisierungsgrad**
Bewerber	>Firmeninformationsangebot	T	T
	Bewerberinteresse>	T	T
	>Firmeninformationen	T	T
	Firmenvermittlungsvertrag>	T	T
	>Intervieweinladung	T	T
	Interviewannahme>	T	T
	>Interview	T	T
	>Firmenvermittlungsdurchführung	T	T
Firma	>Bewerberlistenangebot	T	T
	Bewerberlistenanforderung>	T	T
	>Bewerberlistenübermittlung	T	T
	Bewerbervermittlungsvertrag>	T	T
	Stellenmitteilung>	T	T
	>Bewerbermappenlieferung	T	T
	Bewerberzusage>	T	T
	>Stellenbesetzung	V	V
	>Rechnung (Vermittlung)	T	T
	Zahlung (Vermittlung)>	V	V
Marktforschungs-institut	Marktinformationsangebot>	T	T
	>Abovertrag	V	V
	Marktinformationsdaten>	V	V
	Rechnung (Abo)>	V	V
	>Zahlung (Abo)	V	V

Betriebliche Aufgaben			
Betriebliches Objekt	**Betriebliche Aufgabe**	**Automatisierbarkeit**	**Automatisierungsgrad**
Informationsbeschaffung	>Marktinformationsanforderung	V	V
	>Marktinformationsangebot	T	T
	Abovertrag>	T	T
	>Marktinformationsdaten	V	V
	(aufber.) Marktinformationen>	T	T
	>Rechnung(Abo)	V	V
	Abozahlungsauftrag>	V	V
	Rechnungsdaten (Abo)>	V	V
	>Rechnungserfüllung	V	V
Rechnungswesen	>Forderung (Vermittlung)	V	V
	Rechnung (Vermittlung)>	T	T
	>Zahlung (Vermittlung)	V	V
	Zahlungseingang(Vermittlung)>	V	V
	>Zahlungsanweisung (Abo)	T	T
	Zahlung (Abo)>	V	V
	Zahlungsbericht (Abo)>	V	V
Akquise	>Kontaktabwicklungsauftrag	V	V
	Marktinformationsanforderung>	T	T
	>(aufber.) Marktinformationen	V	V
	Firmeninformationsangebot>	T	T
	Bewerberlistenangebot>	T	T
	>Bewerberlistenanforderung	V	V
	Bewerberlistenübermittlung>	T	T
	>Bewerbervermittlungsvertrag	T	T
	Firmenkontaktinformationen>	V	V

Betriebliche Aufgaben			
Betriebliches Objekt	**Betriebliche Aufgabe**	**Automatisierbarkeit**	**Automatisierungsgrad**
	>Bewerberinteresse	V	V
	Firmeninformationen>	T	T
	>Firmenvermittlungsvertrag	T	T
	Bewerberkontaktinformationen>	V	V
Personalvermittlungsleitung	Kontaktabwicklungsauftrag>	V	V
	>Abozahlungsauftrag	T	T
	>Rechnungsdaten (Abo)	T	T
	>Firmenkontaktinformationen	V	V
	>Bewerberkontaktinformationen	V	V
	Interviewanweisung>	V	V
	>Interviewbericht	V	V
	Firmenvermittlungsanweisung>	V	V
	>Firmenvermittlungsbericht	V	V
	Stellenbesetzungsanweisung>	V	V
	>Stellenbesetzungsbericht	V	V
	Forderung (Vermittlung)>	V	V
	>Zahlungseingang (Vermittlung)	V	V
	Zahlungsanweisung (Abo)>	V	V
	>Zahlungsbericht (Abo)	V	V
	Rechnungserfüllung (Abo)>	V	V
Bewerberbetreuung	>Interviewanweisung	V	V
	Intervieweinladung>	V	V
	>Interviewannahme	V	V
	Interview>	T	T
	Interviewbericht>	V	V

Betriebliche Aufgaben			
Betriebliches Objekt	**Betriebliche Aufgabe**	**Automatisierbarkeit**	**Automatisierungsgrad**
	>Firmenvermittlungsanweisung	V	V
	>Bewerbermappenanforderung	V	V
	Bewerbermappenweitergabe>	T	T
	>Vermittlungsergebnis	V	V
	Firmenvermittlungsdurchführung>	V	V
	Firmenvermittlungsbericht>	V	V
Firmenbetreuung	>Stellenbesetzungsanweisung	V	V
	>Stellenmitteilung	V	V
	Bewerbermappenanforderung>	T	T
	>Bewerbermappenweitergabe	V	V
	Bewerbermappenlieferung>	V	V
	>Bewerberzusage	V	V
	Vermittlungsergebnis>	V	V
	Stellenbesetzung>	V	V
	Stellenbesetzungsbericht>	V	V

N = nicht-automatisierbar/nicht-automatisiert

T = teil-automatisierbar/teil-automatisiert

V = voll-automatisierbar/voll-automatisiert

Tabelle A.1: Automatisierbarkeit und Automatisierung der Aufgaben des Personalvermittlungsunternehmens

Betriebliche Transaktionen		
Betriebliche Transaktion	**Automati-sierbarkeit**	**Automatisie-rungsgrad**
A1: Marktinformationsangebot	A	A
V1: Abovertrag	A	A
D1.D1: Marktinformationsdaten	A	A
D1.D2.V: Rechnung (Abo)	A	A
D1.D2.D: Zahlung (Abo)	A	A
A2.A: Bewerberlistenangebot	A	A
A2.V: Bewerberlistenanforderung	A	A
A2.D: Bewerberlistenübermittlung	A	A
V2: Bewerbervermittlungsvertrag	A	A
D2.D1.V: Stellenmitteilung	A	A
D2.D1.D.A: Bewerbermappenlieferung	A	A
D2.D1.D.V: Bewerberzusage	A	A
D2.D1.D.D: Stellenbesetzung	A	A
D2.D2.V: Rechnung (Vermittlung)	A	A
D2.D2.D: Zahlung (Vermittlung)	A	A
A3.A: Firmeninformationsangebot	A	A
A3.V: Bewerberinteresse	A	A
A3.D: Firmeninformationen	A	A
V3: Firmenvermittlungsvertrag	A	A
D3.D1.A: Intervieweinladung	A	A
D3.D1.V: Interviewannahme	A	A
D3.D1.D: Interview	N	N
D3.D2: Firmenvermittlungsdurchführung	A	A
V4: Kontaktabwicklungsauftrag	A	A
D4.D1: Firmenkontaktinformationen	A	A

Betriebliche Transaktionen		
Betriebliche Transaktion	**Automati-sierbarkeit**	**Automatisie-rungsgrad**
D4.D2: Bewerberkontaktinformationen	A	A
V5: Abozahlungsauftrag	A	A
D5.V: Rechnungsdaten (Abo)	A	A
D5.D: Rechnungserfüllung	A	A
D6.V: Bewerbermappenanforderung	A	A
D6.D.D1: Bewerbermappenweitergabe	A	A
D6.D.D2: Vermittlungsergebnis	A	A
V7: Marktinformationsanforderung	A	A
D7: (aufber.) Marktinformationen	A	A
S1.S1: Firmenvermittlungsanweisung	A	A
K1.K1: Firmenvermittlungsbericht	A	A
S1.S2: Interviewanweisung	A	A
K1.K2: Interviewbericht	A	A
S2: Stellenbesetzungsanweisung	A	A
K2: Stellenbesetzungsbericht	A	A
S3: Forderung (Vermittlung)	A	A
K3: Zahlungseingang (Vermittlung)	A	A
S4: Zahlungsanweisung (Abo)	A	A
K4: Zahlungsbericht (Abo)	A	A

N = nicht-automatisierbar/nicht-automatisiert

V = voll-automatisierbar/voll-automatisiert

Tabelle A.2: Automatisierbarkeit und Automatisierung der Transaktionen des Personalvermittlungsunternehmens

Vorgangsobjekt-schema	**Vorgangsobjekttyp**	**Nachrichtendefinition**
Bewerber	>Firmeninformationsangebot	durchsehenFirmeninformations-angebot
	Bewerberinteresse>	erstellenFirmeninformationsan-forderung
	>Firmeninformationen	durchsehenFirmeninformationen
	Firmenvermittlungsvertrag>	erstellenFirmenvermittlungsver-trag
	>Intervieweinladung	prüfenEinladung
	Interviewannahme>	annehmenEinladung
	>Interview	durchführenInterview
	>Firmenvermittlungsdurchfüh-rung	durchführenFirmenvermittlung
Firma	>Bewerberlistenangebot	durchsehenBewerberlistenan-gebot
	Bewerberlistenanforderung>	anfordernBewerberlisten
	>Bewerberlistenübermittlung	durchsehenBewerberlisten
	Bewerbervermittlungsvertrag>	erstellenBerwerbervermittlungs-vertrag
	Stellenmitteilung>	mitteilenStellen
	>Bewerbermappenlieferung	prüfenBewerbermappe
	Bewerberzusage>	zusagenBewerber
	>Stellenbesetzung	durchführenStellenbesetzung
	>Rechnung (Vermittlung)	annehmenRechnung
	Zahlung (Vermittlung)>	freigebenZahlung
Marktforschungs-institut	Marktinformationsangebot>	anbietenMarktinformationen
	>Abovertrag	annehmenAbovertrag
	Marktinformationsdaten>	zusammenstellenMarkt-informationsdaten

Vorgangsobjekt-schema	**Vorgangsobjekttyp**	**Nachrichtendefinition**
	Rechnung (Abo)>	erstellenRechnung
	>Zahlung (Abo)	empfangenZahlung
Informationsbe-schaffung	>Marktinformationsanforde-rung	empfangenAnforderung
	>Marktinformationsangebot	durchsehenMarktinformations-angebot
	Abovertrag>	erstellenAbovertrag
	>Marktinformationsdaten	empfangenMarktinformationsda-ten
	(aufber.) Marktinformationen>	aufbereitenMarktinformationsda-ten
	>Rechnung(Abo)	annehmenRechnung
	Abozahlungsauftrag>	vorbereitenRechnungsbegle-ichung
	Rechnungsdaten (Abo)>	begleichenRechnung
	>Rechnungserfüllung	empfangenRechnungserfüllung
Rechnungswesen	>Forderung (Vermittlung)	empfangenForderung
	Rechnung (Vermittlung)>	erstellenRechnung
	>Zahlung (Vermittlung)	empfangenZahlung
	Zahlungsein-gang(Vermittlung)>	übermittelnZahlungs-eingangsmeldung
	>Zahlungsanweisung (Abo)	anweisenZahlung
	Zahlung (Abo)>	durchführenZahlung
	Zahlungsbericht (Abo)>	versendenZahlungsbericht
Akquise	>Kontaktabwicklungsauftrag	annehmenKontaktabwicklungs-auftrag
	Marktinformationsanforde-rung>	anfordernMarktinformationen
	>(aufber.) Marktinformationen	empfangenMarktinformationen

Vorgangsobjekt-schema	**Vorgangsobjekttyp**	**Nachrichtendefinition**
	Firmeninformationsangebot>	anbietenFirmeninformationen
	Bewerberlistenangebot>	anbietenBewerberlisten
	>Bewerberlistenanforderung	empfangenBewerberlistenanforderung
	Bewerberlistenübermittlung>	übermittelnBewerberliste
	>Bewerbervermittlungsvertrag	annehmenBewerbervermittlungsvertrag
	Firmenkontaktinformationen>	übermittelnFirmenkontaktinformationen
	>Bewerberinteresse	empfangenFirmeninformationsanforderung
	Firmeninformationen>	übermittelnFirmeninformationen
	>Firmenvermittlungsvertrag	annehmenFirmenvermittlungsvertrag
	Bewerberkontaktinformationen>	übermittelnBewerberkontaktinformationen
Personalvermittlungsleitung	Kontaktabwicklungsauftrag>	übermittelnKontaktabwicklungsauftrag
	>Abozahlungsauftrag	prüfenVertragsdaten
	>Rechnungsdaten (Abo)	prüfenRechnung
	>Firmenkontaktinformationen	empfangenFirmenkontaktinformationen
	>Bewerberkontaktinformationen	empfangenBewerberkontaktinformationen
	Interviewanweisung>	anweisenInterview
	>Interviewbericht	empfangenInterviewbericht
	Firmenvermittlungsanweisung>	anweisenFirmenvermittlung

Vorgangsobjekt-schema	**Vorgangsobjekttyp**	**Nachrichtendefinition**
	>Firmenvermittlungsbericht	empfangenFirmenvermittlungs-bericht
	Stellenbesetzungsanweisung>	anweisenStellenbesetzung
	>Stellenbesetzungsbericht	empfangenStellenbesetzungs-bericht
	Forderung (Vermittlung)>	übermittelnForderung
	>Zahlungseingang (Vermitt-lung)	empfangenZahlungseingangs-meldung
	Zahlungsanweisung (Abo)>	freigebenZahlung
	>Zahlungsbericht (Abo)	empfangenZahlungsbericht
	Rechnungserfüllung (Abo)>	übermittelnRechnungserfül-lungsmeldung
Bewerberbetreu-ung	>Interviewanweisung	empfangenInterviewanweisung
	Intervieweinladung>	erstellenEinladung
	>Interviewannahme	empfangenAnnahme
	Interview>	vorbereitenInterview
	Interviewbericht>	übermittelnInterviewbericht
	>Firmenvermittlungsanwei-sung	empfangenFirmenvermittlungs-anweisung
	>Bewerbermappenanforde-rung	empfangenBewerbermappenan-forderung
	Bewerbermappenweitergabe>	weitergebenBewerbermappe
	>Vermittlungsergebnis	empfangenVermittlungsergebnis
	Firmenvermittlungsdurchfüh-rung>	vorbereitenFirmenvermittlung
	Firmenvermittlungsbericht>	übermittelnFirmenvermittlungs-bericht
Firmenbetreuung	>Stellenbesetzungsanweisung	empfangenStellenbesetzungs-anweisung

Vorgangsobjekt-schema	**Vorgangsobjekttyp**	**Nachrichtendefinition**
	>Stellenmitteilung	empfangenStellenmitteilung
	Bewerbermappenanforde-rung>	anfordernBewerbermappe
	>Bewerbermappenweitergabe	empfangenBewerbermappe
	Bewerbermappenlieferung>	übermittelnBewerbermappe
	>Bewerberzusage	empfangenBewerberzusage
	Vermittlungsergebnis>	übermittelnVermittlungsergebnis
	Stellenbesetzung>	vorbereitenStellenbesetzung
	Stellenbesetzungsbericht>	übermittelnStellenbesetzungs-bericht

Tabelle A.3: Nachrichtendefinitionen der Vorgangsobjekttypen des Personalvermittlungsunternehmens

Betriebliche Aufgaben			
Betriebliches Objekt	**Betriebliche Aufgabe**	**Automatisierbarkeit**	**Automatisierungsgrad**
Kunde	>Katalog	T	T
	Konfigurationsanfrage>	T	T
	>Konfigurationsbestätigung	V	V
	Konfigurationsbestellung>	T	T
	>Produktlieferung	T	T
	>Rechnung (PC)	V	V
	Zahlung (PC)>	T	T
Lieferant	Rahmenvertragsangebot>	T	T
	>Rahmenvertrag	T	T
	>Teileabruf	V	V
	Teilelieferung>	V	V
	Rechnung (Teile)>	V	V
	>Zahlung (Teile)	V	V
Rechnungswesen	>Verbindlichkeit	V	V
	>Rechnung (Teile)	T	T
	Zahlung (Teile)>	T	T
	Erfüllungsmeldung>	V	V
	>Forderung	V	V
	Rechnung (PC)>	V	V
	>Zahlung (PC)	V	V
	Zahlungseingang>	V	V
Auftragsbearbeitung	>Rahmenvertragsangebot	T	T
	Rahmenvertrag>	T	T
	Katalog>	T	T
	>Konfigurationsanfrage	V	V

Betriebliche Aufgaben			
Betriebliches Objekt	**Betriebliche Aufgabe**	**Automatisierbarkeit**	**Automatisierungsgrad**
	Konfigurationsbestätigung>	T	T
	>Konfigurationsbestellung	V	V
	Teileabruf>	T	T
	Warenbereitstellungsanweisung>	V	V
	Verbindlichkeit>	V	V
	>Erfüllungsmeldung	V	V
	Produktionsanweisung>	V	V
	>Warenbereitstellungsbericht	V	V
	Versandanweisung>	V	V
	>Produktionsbericht	V	V
	>Versandbericht	V	V
	Forderung>	V	V
	>Zahlungseingang	V	V
Wareneingangslager	>Warenbereitstellungsanweisung	V	V
	>Teilelieferung	T	T
	Teileübergabe>	V	V
	Warenbereitstellungsbericht>	V	V
Produktion	>Produktionsanweisung	V	V
	>Teileübergabe	T	T
	Fertig-PC>	V	V
	Produktionsbericht>	V	V

Betriebliche Aufgaben			
Betriebliches Objekt	**Betriebliche Aufgabe**	**Automati-sierbarkeit**	**Automatisie-rungsgrad**
Versand	>Versandanweisung	V	V
	>Fertig-PC	V	V
	Produktlieferung>	T	T
	Versandbericht>	V	V

N = nicht-automatisierbar/nicht-automatisiert

T = teil-automatisierbar/teil-automatisiert

V = voll-automatisierbar/voll-automatisiert

Tabelle A.4: Automatisierbarkeit und Automatisierung der Aufgaben des PC-Fertigungsunternehmens

Betriebliche Transaktionen		
Betriebliche Transaktion	**Automati-sierbarkeit**	**Automatisie-rungsgrad**
A1: Rahmenvertragsangebot	A	A
V1: Rahmenvertrag	A	A
D1.D1.V: Teileabruf	A	A
D1.D1.D: Teilelieferung	N	N
D1.D2.V: Rechnung (Teile)	A	A
D1.D2.D: Zahlung (Teile)	A	A
A2: Katalog	A	A
V2.V: Konfigurationsanfrage	A	A
V2.D.D1: Konfigurationsbestätigung	A	A
V2.D.D2: Konfigurationsbestellung	A	A
D2.D1: Produktlieferung	N	N
D2.D2.V: Rechnung (PC)	A	A
D2.D2.D: Zahlung (PC)	A	A
D3: Teileübergabe	A	A
D4: Fertig-PC	A	A
S1: Verbindlichkeit	A	A
K1: Erfüllungsmeldung	A	A
S2: Forderung	A	A
K2: Zahlungseingang	A	A
S3: Warenbereitstellungsanweisung	A	A
K3: Warenbereitstellungsbericht	A	A
S4: Produktionsanweisung	A	A
K4: Produktionsbericht	A	A

Betriebliche Transaktionen		
Betriebliche Transaktion	**Automati-sierbarkeit**	**Automatisie-rungsgrad**
S5: Versandanweisung	A	A
K5: Versandbericht	A	A

N = nicht-automatisierbar/nicht-automatisiert

V = voll-automatisierbar/voll-automatisiert

Tabelle A.5: Automatisierbarkeit und Automatisierung der Transaktionen des PC-Fertigungsunternehmens

Vorgangsobjekt-schema	**Vorgangsobjekttyp**	**Nachrichtendefinition**
Kunde	>Katalog	durchsehenKatalog
	Konfigurationsanfrage>	anfragenKonfiguration
	>Konfigurationsbestätigung	empfangenKonfigurationsbestä-tigung
	Konfigurationsbestellung>	bestellenKonfiguration
	>Produktlieferung	annehmenProduktlieferung
	>Rechnung (PC)	annehmenRechnung
	Zahlung (PC)>	freigebenZahlung
Lieferant	Rahmenvertragsangebot>	anbietenRahmenvertrag
	>Rahmenvertrag	annehmenRahmenvertrag
	>Teileabruf	annehmenLieferauftrag
	Teilelieferung>	vorbereitenTeilelieferung
	Rechnung (Teile)>	erstellenRechnung
	>Zahlung (Teile)	empfangenZahlung
Rechnungswesen	>Verbindlichkeit	buchenVerbindlichkeit
	>Rechnung (Teile)	prüfenRechnung
		begleichenRechnung
	Zahlung (Teile)>	freigebenZahlung
	Erfüllungsmeldung>	erstellenErfüllungsmeldung
	>Forderung	buchenForderung
	Rechnung (PC)>	erstellenRechnung
	>Zahlung (PC)	empfangenZahlung
	Zahlungseingang>	bestätigenZahlungseingang
Auftragsbearbei-tung	>Rahmenvertragsangebot	annehmenRahmenvertragsan-gebot
	Rahmenvertrag>	erstellenRahmenvertrag
	Katalog>	erstellenKatalog

Vorgangsobjekt-schema	**Vorgangsobjekttyp**	**Nachrichtendefinition**
	>Konfigurationsanfrage	empfangenKonfigurationsanfrage
	Konfigurationsbestätigung>	bestätigenKonfiguration
	>Konfigurationsbestellung	annehmenKonfigurationsbestellung
	Teileabruf>	erstellenLieferauftrag
	Warenbereitstellungsanweisung>	anweisenWarenbereitstellung
	Verbindlichkeit>	erstellenVerbindlichkeit
	>Erfüllungsmeldung	empfangenErfüllungsmeldung
	Produktionsanweisung>	anweisenProduktion
	>Warenbereitstellungsbericht	empfangenWarenbereitstellungsbericht
	Versandanweisung>	anweisenVersand
	>Produktionsbericht	empfangenProduktionsbericht
	>Versandbericht	empfangenVersandbericht
	Forderung>	erstellenForderung
	>Zahlungseingang	empfangenZahlungeingangsbestätigung
Wareneingangslager	>Warenbereitstellungsanweisung	empfangenWarenbereitstellungsanweisung
	>Teilelieferung	annehmenTeilelieferung
		prüfenTeilelieferung
	Teileübergabe>	vorbereitenTeileübergabe
	Warenbereitstellungsbericht>	erstellenWarenbereitstellungsbericht
Produktion	>Produktionsanweisung	empfangenProduktionsanweisung
	>Teileübergabe	annehmenTeileübergabe

Vorgangsobjekt-schema	**Vorgangsobjekttyp**	**Nachrichtendefinition**
		prüfenTeileübergabe
	Fertig-PC>	übergebenFertigPC
	Produktionsbericht>	erstellenProduktionsbericht
Versand	>Versandanweisung	empfangenVersandanweisung
	>Fertig-PC	annehmenFertigPC
		prüfenFertigPC
	Produktlieferung>	liefernProdukt
	Versandbericht>	erstellenVersandbericht

Tabelle A.6: Nachrichtendefinitionen der Vorgangsobjekttypen des PC-Fertigungsunternehmens

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar id="ejb-jar_ID">
      <display-name>Fallstudie1</display-name>
      <enterprise-beans>
            <session id="Abovertragsabschluss">
                  <ejb-name>Abovertragsabschluss</ejb-name>
                  <home>AbovertragsabschlussHome</home>
                  <remote>Abovertragsabschluss</remote>
                  <ejb-class>AbovertragsabschlussBean</ejb-class>
                  <session-type>Stateless</session-type>
                  <transaction-type>Container</transaction-type>
            </session>
            <session id="KreditorrechnungsbearbeitungAdapter">
                  <ejb-name>KreditorrechnungsbearbeitungAdapter</ejb-name>
                  <home>KreditorrechnungsbearbeitungAdapterHome</home>
                  <remote>KreditorrechnungsbearbeitungAdapter</remote>
                  <ejb-class>KreditorrechnungsbearbeitungAdapterBean</ejb-
                  class>
                  <session-type>Stateless</session-type>
                  <transaction-type>Container</transaction-type>
            </session>
            <session id="Kreditorrechnungsbearbeitung">
                  <ejb-name>Kreditorrechnungsbearbeitung</ejb-name>
                  <home>KreditorrechnungsbearbeitungHome</home>
                  <remote>Kreditorrechnungsbearbeitung</remote>
                  <ejb-class>KreditorrechnungsbearbeitungBean</ejb-class>
                  <session-type>Stateless</session-type>
                  <transaction-type>Container</transaction-type>
            </session>
            <entity id="Abovertrag">
                  <ejb-name>Abovertrag</ejb-name>
                  <home>AbovertragHome</home>
                  <remote>Abovertrag</remote>
                  <ejb-class>AbovertragBean</ejb-class>
                  <persistence-type>Container</persistence-type>
                  <prim-key-class>AbovertragKey</prim-key-class>
                  <reentrant>False</reentrant>
                  <cmp-field>
                        <field-name>k_Frist</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>zeitraum</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>rabatt</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>AV_ID</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>preis</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>mIA_ID</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>mIAN_ID</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>erhebungsmenge</field-name>
```

```
            </cmp-field>
            <cmp-field>
                  <field-name>datum</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>unterzeichner</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>pruefstatus</field-name>
            </cmp-field>
      </entity>
      <entity id="Kreditorrechnung">
            <ejb-name>Kreditorrechnung</ejb-name>
            <home>KreditorrechnungHome</home>
            <remote>Kreditorrechnung</remote>
            <ejb-class>KreditorrechnungBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>KreditorrechnungKey</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-field>
                  <field-name>kr_ID</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>blz</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>betrag</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>faelligkeit</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>status</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>ktoNr</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>lv_ID</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>r_ID</field-name>
            </cmp-field>
            <ejb-ref>
                  <ejb-ref-name>ejb/Liefervertrag</ejb-ref-name>
                  <ejb-ref-type>Entity</ejb-ref-type>
                  <home>LiefervertragHome</home>
                  <remote>Liefervertrag</remote>
                  <ejb-link>Liefervertrag</ejb-link>
            </ejb-ref>
      </entity>
      <entity id="Liefervertrag">
            <ejb-name>Liefervertrag</ejb-name>
            <home>LiefervertragHome</home>
            <remote>Liefervertrag</remote>
            <ejb-class>LiefervertragBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>LiefervertragKey</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-field>
                  <field-name>lv_ID</field-name>
            </cmp-field>
```

```
                  <cmp-field>
                        <field-name>datum</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>k_frist</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>rabatt</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>summe</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>pruefstatus</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>angebot_id</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>lieferant_id</field-name>
                  </cmp-field>
                  <ejb-ref>
                        <ejb-ref-name>ejb/Kreditorrechnung</ejb-ref-name>
                        <ejb-ref-type>Entity</ejb-ref-type>
                        <home>KreditorrechnungHome</home>
                        <remote>Kreditorrechnung</remote>
                        <ejb-link>Kreditorrechnung</ejb-link>
                  </ejb-ref>
            </entity>
      </enterprise-beans>
</ejb-jar>
```

Abbildung A.1: *Deployment Descriptor* des komponentenbasierten Anwendungssystems für das Personalvermittlungsunternehmen

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar id="ejb-jar_ID">
      <display-name>Fallstudie2</display-name>
      <enterprise-beans>
            <session id="Teileabruf">
                  <ejb-name>Teileabruf</ejb-name>
                  <home>TeileabrufHome</home>
                  <remote>Teileabruf</remote>
                  <ejb-class>TeileabrufBean</ejb-class>
                  <session-type>Stateless</session-type>
                  <transaction-type>Container</transaction-type>
            </session>
            <session id="KreditorrechnungsbearbeitungAdapter">
                  <ejb-name>KreditorrechnungsbearbeitungAdapter</ejb-name>
                  <home>KreditorrechnungsbearbeitungAdapterHome</home>
                  <remote>KreditorrechnungsbearbeitungAdapter</remote>
                  <ejb-class>KreditorrechnungsbearbeitungAdapterBean</ejb-
                  class>
                  <session-type>Stateless</session-type>
                  <transaction-type>Container</transaction-type>
            </session>
            <session id="Kreditorrechnungsbearbeitung">
                  <ejb-name>Kreditorrechnungsbearbeitung</ejb-name>
                  <home>KreditorrechnungsbearbeitungHome</home>
                  <remote>Kreditorrechnungsbearbeitung</remote>
                  <ejb-class>KreditorrechnungsbearbeitungBean</ejb-class>
                  <session-type>Stateless</session-type>
                  <transaction-type>Container</transaction-type>
            </session>
            <entity id="Lieferauftrag">
                  <ejb-name>Lieferauftrag</ejb-name>
                  <home>LieferauftragHome</home>
                  <remote>Lieferauftrag</remote>
                  <ejb-class>LieferauftragBean</ejb-class>
                  <persistence-type>Container</persistence-type>
                  <prim-key-class>LieferauftragKey</prim-key-class>
                  <reentrant>False</reentrant>
                  <cmp-field>
                        <field-name>auftragstatus</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>la_ID</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>datum</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>kvertrag_ID</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>rvertrag_ID</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>teileliste</field-name>
                  </cmp-field>
            </entity>
            <entity id="Kreditorrechnung">
                  <ejb-name>Kreditorrechnung</ejb-name>
                  <home>KreditorrechnungHome</home>
                  <remote>Kreditorrechnung</remote>
```

```
            <ejb-class>KreditorrechnungBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>KreditorrechnungKey</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-field>
                  <field-name>kr_ID</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>blz</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>betrag</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>faelligkeit</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>status</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>ktoNr</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>lv_ID</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>r_ID</field-name>
            </cmp-field>
            <ejb-ref>
                  <ejb-ref-name>ejb/Liefervertrag</ejb-ref-name>
                  <ejb-ref-type>Entity</ejb-ref-type>
                  <home>LiefervertragHome</home>
                  <remote>Liefervertrag</remote>
                  <ejb-link>Liefervertrag</ejb-link>
            </ejb-ref>
      </entity>
      <entity id="Liefervertrag">
            <ejb-name>Liefervertrag</ejb-name>
            <home>LiefervertragHome</home>
            <remote>Liefervertrag</remote>
            <ejb-class>LiefervertragBean</ejb-class>
            <persistence-type>Container</persistence-type>
            <prim-key-class>LiefervertragKey</prim-key-class>
            <reentrant>False</reentrant>
            <cmp-field>
                  <field-name>lv_ID</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>datum</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>k_frist</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>rabatt</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>summe</field-name>
            </cmp-field>
            <cmp-field>
                  <field-name>pruefstatus</field-name>
            </cmp-field>
```

```
                  <cmp-field>
                        <field-name>angebot_id</field-name>
                  </cmp-field>
                  <cmp-field>
                        <field-name>lieferant_id</field-name>
                  </cmp-field>
                  <ejb-ref>
                        <ejb-ref-name>ejb/Kreditorrechnung</ejb-ref-name>
                        <ejb-ref-type>Entity</ejb-ref-type>
                        <home>KreditorrechnungHome</home>
                        <remote>Kreditorrechnung</remote>
                        <ejb-link>Kreditorrechnung</ejb-link>
                  </ejb-ref>
            </entity>
      </enterprise-beans>
</ejb-jar>
```

Abbildung A.2: *Deployment Descriptor* des komponentenbasierten Anwendungssystems für das PC-Fertigungsunternehmen

Lebenslauf

Persönliche Daten:	Name:	Christian Robra
	Geburtstag:	02. Januar 1971
	Geburtsort:	Darmstadt
	Staatsangehörigkeit:	deutsch
	Familienstand:	ledig
Schulbildung:	09/1977 - 07/1981	Grundschule Heroldsberg
	09/1981 - 07/1990	Labenwolf-Gymnasium Nürnberg (mathematisch-naturwissen-schaftlicher Zweig)
Schulabschluss:	06/1990	Allgemeine Hochschulreife
Studium:	11/1990 - 07/1992	Informatik an der Universität Erlangen-Nürnberg; Wechsel in den Studiengang Wirtschaftsinformatik
	11/1992 - 05/1998	Wirtschaftsinformatik an der Universität Erlangen-Nürnberg
Studienabschluss:	05/1998	Diplom-Wirtschaftsinformatiker (Dipl.-Wirtsch.-Inf.)
Promotion:	11/1999 - 05/2002	Wirtschaftsinformatik an der Universität Bamberg
Promotionsabschluss:	09/2004	Doctor rerum politicarum (Dr. rer. pol.)

Berufserfahrung:	04/1996 - 05/1997	Freier Mitarbeiter bei der Novell GmbH, München im Bereich Sales Promotion
	09/1998 - 04/1999	IT-Berater bei der SerCon GmbH, München
	06/1999 - 05/2003	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Wirtschaftsinformatik, insbesondere Systementwicklung und Datenbankanwendung (Prof. Dr. E.J. Sinz)
	seit 04/2004	Prozessanalytiker im Bereich Automobilfertigung bei der SALT Solutions GmbH, Oberpfaffenhofen

Unterhaching, 28. Mai. 2005