Ein Software-Framework für die Entwicklung betrieblicher Anwendungssysteme auf der Basis von Geschäftsprozessmodellen

Dissertation

zur Erlangung des akademischen Grades

Dr. rer. pol.

vorgelegt an der

Fakultät für Wirtschaftsinformatik und Angewandte Informatik
der
Otto-Friedrich-Universität Bamberg

von
Christian Weichelt

Bamberg,
März 2009

Gutachter:

Prof. Dr. Otto K. Ferstl

Prof. Dr. Elmar J. Sinz

Tag der Disputation: 05. Februar 2010

Meinen Eltern

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

ACID	Atomicity Consistency Isolation Durability
AOP	Aspect-oriented Programming
API	Application Programming Interface
ASF	Apache Software Foundation
AT	Aufgabenträger
AwS	Anwendungssystem
BP-MDSD	Business Process Model Driven Software Development
CAS	Central Authentication Service
CCK	Computer-Computer-Kommunikation
CORBA	Common Object Request Broker Architecture
CUA	Common User Access
DBMS	Datenbankmanagementsystem
DBVS	Datenbankverwaltungssystem
DOM	Document Object Modell
E-Typ	Entity Typ
EAI	Enterprise Application Integration
EDV	Elektronische Datenverarbeitung
EJB	Enterprise Java Beans

ER-Typ	Entity-Relationship Typ
ERP	Enterprise Resource Planning
EStdIT	Entwicklungsstandard für IT-Systeme des Bundes
EUS	Entscheidungssunterstützungssystem
HQL	Hibernate Query Language
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IDL	Interface Definition Language
IOS	Interface Objektschema
IOT	Interface Objekttyp
IS	Informationssystem
ISV	Independent Software Vendors
IV	Informationsverarbeitung
JAR	Java Archive
J2EE	Java 2 Enterprise Edition
JVM	Java Virtual Machine
KO	Konzeptuelles Objekt
KOS	Konzeptuelles Objektschema
KOS-P	Konzeptuelles Objektschema persistenter Klassen
KOT	Konzeptueller Objekttyp
MCI	Mensch-Computer-Interface
MCK	Mensch-Computer-Kommunikation
MDA	Model Driven Architecture
MDSD	Model Driven Software Development

MMK	Mensch-Mensch-Kommunikation
OM	Objektmanager
ooAM	objektorientiertes Architekturmodell
OMG	Object Management Group
OMT	Object Modeling Technique
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
OOSA	Object-Oriented Systems Analysis
OOSE	Object-Oriented Software Engineering
OOUI	Object-Oriented User Interface
PIM	Platform Independent Model
PSM	Platform Specific Model
R-Typ	Relationship Typ
SAX	Simple API for XML
SERM	Strukturiertes Entity Relationship Modell
SOM	Semantisches Objektmodell
SQL	Structured Query Language
TOS	Technisches Objektschema
UML	Unified Modeling Language
VO	Vorgangsobjekt
VOS	Vorgangsobjektschema
VOT	Vorgangsobjekttyp
XML	Extensible Markup Language

XP	Extreme Programming
XSD	XML Schema Definition Language
XSL	Extensible Stylesheet Language
XSLT	XSL Transformations

1 Einleitung

Bereits Ende der 1950er Jahre übernahmen Software-Anwendungen als elektronische Büroangestellte - so genannte *Electronic Clerks* [Rose67] - die automatisierte Durchführung einfacher Verwaltungsaufgaben. Die enorm gestiegene Leistungsfähigkeit von Hardware und die im Zuge dessen rasch zunehmende Komplexität der übertragenen Aufgaben führten aber in den folgenden Jahren zu großen Schwierigkeiten bei Entwurf und Produktion von Software, so dass man sehr bald von einer **Software-Krise** [Dijk72] sprach. Im Zuge dieser Krise entstand der Ansatz des **Software Engineering**[1], dessen Aufgabe die Herstellung von qualitativ hochwertiger Software nach ingenieursmäßigen Prinzipien ist.

Seither ist der Einsatz von Software in Unternehmen selbstverständlich geworden und **betriebliche Anwendungssysteme** übernehmen die automatisierte Durchführung einer Vielzahl betrieblicher Aufgaben. Diese sind vor dem Hintergrund zunehmender Internationalisierung und verstärkten Wettbewerbs nicht nur äußerst komplex, sondern unterliegen zudem permanenten Anpassungen [Krüg98]. Mit der Folge, dass vor allem die fachlichen Anforderungen an betriebliche Anwendungssysteme bei gleichzeitiger Zunahme der Nachfrage[2] in einem Maße gewachsen sind, dass deren technische Umsetzung und Pflege trotz großer Fortschritte des Software Engineering sehr aufwendig ist [Boeh08].

Erschwerend kommt hinzu, dass sich sowohl Unternehmen als auch Software-Hersteller einem vor allem durch steigende Entwicklungskosten verursachten erhöhten Kostendruck gegenüber sehen. Einen Ausweg scheint die (Teil-)Auslagerung (*Outsourcing*) der Produktion durch *Offshoring*[3] in Länder mit deutlich niedrigeren Löhnen zu bieten - mit enormen gesellschaftlichen

[1] Grundstein war eine gleichnamige Konferenz, die 1968 in Garmisch-Partenkirchen zum Zwecke der Diskussion aktueller Probleme bei der Entwicklung von Software stattfand (siehe u. a. [NaRa68], [Rand79] sowie rückblickend [McCl08]).

[2] Bis zum Jahr 2013 wird im Vergleich zu 2003 mit einer weiteren Zunahme der Nachfrage um das zwei- bis dreifache gerechnet [Alk$^+$03, 259ff].

[3] *Offshore*-Entwicklung bezeichnet aus Sicht des *Onshore*-Marktes die Produktion in fernen Niedriglohnländer [BoSc04, 9] und ist das Ergebnis des *Offshoring*-Prozesses.

und ökonomischen Folgen für Hochlohnländer wie Deutschland[4], bei fraglichem Erfolg für die Verbesserung der **Software-Qualität**. Denn die Verkürzung der Entwicklungszeit sowie die Senkung der Entwicklungskosten führt nicht zwingend zu besseren Anwendungssystemen, sondern ist häufig mit dem Risiko sinkender Qualität verbunden[5] [Alk+03, 266].
Der beschriebenen Situation kann folglich nicht bzw. nicht ausschließlich über eine Reduzierung der Entwicklungskosten begegnet werden. Vielmehr ist es notwendig, den gesamten Lebenszyklus - von der fachlichen Konzeption über die technische Implementierung bis zum Betrieb der Anwendung - sowie die (fachliche) Qualität der zu realisierenden betrieblichen Anwendungssysteme zu betrachten. Lösungen für deren Entwicklung und Implementierung zu erarbeiten ist eine zentrale Aufgabe der Wirtschaftsinformatik [Sche88, 1ff], die sie im Rahmen der Gestaltung **betrieblicher Informationssysteme** wahrnimmt. Mit einem Software-Framework will die vorliegende Arbeit vor diesem Hintergrund einen Beitrag zur Unterstützung der software-technischen Realisierung betrieblicher Anwendungssysteme leisten.

1.1 Motivation und Problemstellung

Gegenstand der Untersuchung ist die Systementwicklung als Aufgabe, deren Sachziel die Entwicklung eines Anwendungssystems (AwS) gemäß vorgegebener Anforderungen ist [FeSi06, 459ff]. Als Formalziele sind bei der Durchführung Zeit- und Kostenvorgaben zu beachten sowie Qualitätsanforderungen an den Systementwicklungsprozess selbst.
Insgesamt handelt es sich bei der Systementwicklung um eine sehr umfangreiche sowie komplexe Aufgabe. Ihr Lösungsverfahren wird daher zumeist in einzelne Phasen unterteilt (vgl. [FeSi06, 461]):

- **Anforderungsanalyse und -definition** zur Ableitung und Spezifikation der zu entwickelnden Anwendungsfunktionen des AwS
- **Software-Design** zur Spezifikation der Software-Architektur des AwS

[4] Für eine Betrachtung der gesellschaftlichen Auswirkungen siehe u. a. [BoSc04] sowie die dort angegebene Literatur.
[5] Vgl. hierzu auch die Variablen der Realisierbarkeit (*viability variables*) in [Fran03, 4]. FRANKEL argumentiert, dass hochwertige, langlebige Software teuer in der Herstellung ist. Es bestehen damit Zielkonflikte zwischen Qualität, Langlebigkeit und Produktionskosten der Software, die er als *viability variables* der Software-Entwicklung bezeichnet.

- **Realisierung** zur eigentlichen Programmierung der Anwendungsfunktionen des AwS

Das fachliche Modell des Anwendungssystems dient dabei nicht nur der ganzheitlichen Analyse und Definition der fachlichen Anforderungen an das zu entwickelnde Anwendungssystem, sondern ist für den anschließenden software-technischen Realisierungsprozess eine wichtige Grundlage, um den Umfang und die Komplexität der Systementwicklungsaufgabe verringern zu können. Doch gerade beim Übergang von fachlicher zu software-technischer Beschreibungsebene fehlen geeignete Software-Komponenten, um die Mächtigkeit der fachlichen Spezifikation zu nutzen. Entwurf und Evolution verfügbarer Software-Komponenten erfolgen in der Regel *bottom up* durch Abstraktion von einem wiederkehrenden software-technischen Problem, das sie lösen. Die Ableitung und Verfeinerung der zu realisierenden Komponenten erfolgt hingegen *top down*.

Die zu Grunde liegende Problemstellung lässt sich anhand des **Software-Systemmodells** verdeutlichen, welches das Modell von Nutzer- und Basismaschine (vgl. [FeSi06, 302]) um eine Verfahrens- sowie Systemumgebung erweitert [Fer+96b, 46f] (siehe Abbildung 1.1). Die Nutzermaschine kann fachlich abgeleitet werden, sie spezifiziert die Anwendungsfunktionen des zu entwickelnden Anwendungssystems. Ihre software-technische Realisierung erfolgt durch Erstellung eines Programms unter Verwendung der verfügbaren Basismaschine. Der Aufwand der Programmerstellung und damit der Aufwand der Systementwicklungsaufgabe hängt entscheidend vom Komplexitätsabstand zwischen Nutzermaschine und Basismaschine ab, der als **semantische Lücke** bezeichnet wird [FeSi06, 305]. Um diese gering zu halten, müssen die Komponenten der Basismaschine denen der Nutzermaschine in Art und Umfang ähnlich sein [FeSi06, 305].

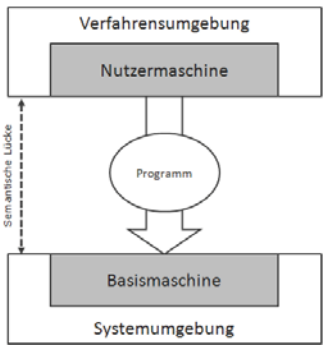

Abbildung 1.1: Software-Systemmodell [Fer+96b, 46]

Vor diesem Hintergrund können drei Hauptgründe für die mangelnde software-technische Nutzung des fachlichen Modells angeführt werden: die Komponenten verfügbarer Basismaschinen bieten zumeist ausschließlich technische Funktionalität an, sie beziehen sich nur auf eine Teilmenge der zu realisierenden Anwendungsfunktionen oder die Spezifikation der Nutzermaschine selbst ist nicht ausreichend mächtig, da sie nicht fachlich abgeleitet, sondern lediglich software-technisch motiviert ist.

Die Systementwicklung ist aber nicht isoliert zu betrachten, sondern ganzheitlich als Teilaufgabe der Gestaltungsaufgabe *Automatisierung betrieblicher Informationssysteme* (vgl. [Fers92, 4]). Ihr Sachziel *Vorgangsautomatisierung* leitet sich aus der Aufgabenebene des betrieblichen Informationssystems ab. Mit diesem korrespondieren die Formalziele Korrektheit, Integration, Echtzeitverhalten sowie Flexibilität [Fers92, 11], die sich auf Merkmale des (zu entwickelnden) Anwendungssystems und damit auf das Aufgabenobjekt der eigentlichen Systementwicklungsaufgabe beziehen.

Diese Merkmale des Anwendungssystems müssen bereits Grundlage dessen fachlicher Ableitung sein. In der Phase *Realisierung* müssen sie umgesetzt werden. Ein entsprechendes Lösungsverfahren muss ausgehend vom fachlichen Modell des Anwendungssystems den Entwicklungsprozess beim Übergang von fachlicher zu software-technischer Beschreibungsebene und anschließender Realisierung geeignet unterstützen. Ein solches Lösungsverfahren birgt ein großes Optimierungspotenzial für die Systementwicklungsaufgabe. Vor allem vor dem Hintergrund einer besseren Automatisierbarkeit der Aufgabendurchführung, wie sie von einer modellgetriebenen Software-Entwicklung (MDSD) (u. a. in [Fran03], [StVö05]) angestrebt wird.

1.2 Zielsetzung und Lösungsansatz der Arbeit

Ziel der Arbeit ist daher die Konzeption und Realisierung einer flexiblen Basismaschine, die ausgehend von einem geeigneten fachlichen Modell des Anwendungssystems anwendungsneutrale Software-Komponenten zu dessen vollständiger software-technischer Abbildung anbietet, um die Systementwicklungsaufgabe hinsichtlich Umfang und Komplexität zu verringern.

Aufbauend auf den Fachkonzepten der Modellierungsmethodik **Semantisches Objektmodell** (SOM) (siehe hierzu [FeSi06, 185ff], die dort angegebene Literatur sowie Kapitel 5.3) sowie den Software-Komponenten des **objektorientierten Architekturmodells** (ooAM) [Ambe93]

wird als Basismaschine ein zugehöriges Software-Framework entwickelt.
SOM ist ein umfassender geschäftsprozess- und objektorientierter Ansatz zur Modellierung betrieblicher Systeme und zur fachlichen Spezifikation von Anwendungssystemen. Das der SOM-Methodik zu Grunde liegende Geschäftsprozessverständnis betrachtet einen Geschäftsprozess nicht nur als ereignisgesteuerten Ablauf von Aufgabendurchführungen, sondern bezieht „die auf Unternehmensziele ausgerichtete Leistungserstellung, deren Lenkung sowie die zur Leistungserstellung und Lenkung eingesetzten Ressourcen mit ein“ [FeSi06, 130]. Geschäftsprozesse sind damit das Bindeglied zwischen Unternehmensstrategie und Systementwicklung [Gada01, 30]. Die Konzepte der SOM-Methodik zur fachlichen Spezifikation von Anwendungssystemen nehmen Bezug auf die Aufgabenebene des Informationssystems und die dort formulierten Ziele. Für den Übergang zur software-technischen Beschreibungsebene definiert die Software-Architektur des ooAM Software-Komponenten und damit die Komponenten einer zu realisierenden Nutzermaschine.
Diese sind die Grundlage für den Lösungsansatz dieser Arbeit, das Software-Framework **moccabox**. Generell definiert und implementiert ein Software-Framework[6] anwendungsneutrale Software-Komponenten sowie deren Kooperationsverhalten (siehe Abbildung 1.2).

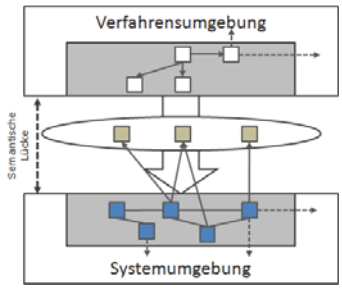

Abbildung 1.2: Programmerstellung mit Framework-Einsatz (nach [Fer+96b, 46])

Frameworks liegt das Konzept der Wiederverwendung auf Entwurfs- sowie Code-Ebene zu Grunde [John97, 10f]. Durch Wiederverwendung der Komponenten verringert sich der Umfang der Systementwicklungsaufgabe:

- Die Software-Architektur des Anwendungssystems muss nicht im Rahmen des Entwicklungsprozesses entworfen werden, da sie vom Software-Framework definiert und implementiert wird.

[6] Zu Frameworks vgl. Kapitel 7.

- Das Kooperationsverhalten der Komponenten muss nicht durch das Programm abgebildet werden, sondern wird vom Software-Framework implementiert. Dadurch wird eine Trennung der Geschäftslogik, wie sie anwendungsspezifisch umgesetzt wird, von der Komplexität des Software-Systems erreicht.

- Der Umfang der Programmerstellung sinkt, da anwendungsneutrale Funktionalität nicht während der Aufgabendurchführung entwickelt werden muss, sondern vom Software-Framework zur Verfügung gestellt wird.

Mit Bezug zu den Fachkonzepten der SOM-Methodik ist nicht technische Abstraktion *bottom up* Ausgangspunkt des vorliegenden Framework-Entwurfs, sondern die fachliche Ableitung *top down* aus Anforderungen betrieblicher Informationssysteme. Dadurch kann die Mächtigkeit des fachlichen Modells erhalten und die Komplexität der Systementwicklungsaufgabe bei der Realisierung verringert werden. Die erreichte Vereinfachung kann wiederum anhand des Software-Systemmodells verdeutlicht werden:

- Die Programmerstellung ist vereinfacht, da die zu überbrückende semantische Lücke klein ist, bedingt durch den geringen Komplexitätsabstand zwischen den Komponenten des Frameworks und den Konzepten des Anwendungsmodells.

- Die Abbildung des fachlichen Modells auf die Komponenten des Frameworks wird vollständig für alle definierten Fachkonzepte unterstützt, einschließlich des definierten Kooperationsverhaltens (siehe oben). Die Fachkonzepte der SOM-Methodik spezifizieren interaktive Anwendungssysteme. Mit den Komponenten des Frameworks lassen sich die entsprechenden Oberflächen zur Mensch-Computer-Kommunikation (MCK) ohne weitere Programmierung ableiten.

- Die Fachkonzepte der SOM-Methodik unterstützen die fachliche Einhaltung von Integrationszielen [Ambe99, 13] (siehe auch [Rose99]). Das Framework hält software-technische Komponenten bereit, um die Verfolgung der Integrationsziele zur Entwicklungs- und Laufzeit zu unterstützen.

- Das fachliche Modell des Anwendungssystems besitzt Strukturähnlichkeiten und -analogien zur Ebene der Geschäftsprozesse des Unternehmens. Diese Analogien werden von

der Basismaschine Framework abgebildet. Anpassungen der Geschäftsprozesse bleiben durch korrespondierende Anpassungen auf Ebene des Anwendungssystems nachvollziehbar.

- Die Programmerstellung lässt sich bei entsprechend geringer semantischer Lücke und gegebener Vollständigkeit der Abbildbarkeit automatisieren (nicht Teil der Arbeit). Im Sinne der modellgetriebenen Software-Entwicklung stellt das Framework dafür eine wichtige Voraussetzung dar.

Im Ergebnis wird durch den Einsatz des **moccabox**-Framework im Anschluss an die Ableitung der Fachspezifikation mit SOM ein durchgängiges Lösungsverfahren der Systementwicklungsaufgabe unterstützt. Es handelt sich beim **moccabox**-Framework nicht um ein fachliches Domänen-Framework für eine bestimmte Anwendungsdomäne (z.B. Versicherung). Es dient bei der Entwicklung betrieblicher Anwendungssysteme als Komponentenplattform, auf der Komponenten sowohl neu entwickelt als auch bestehende wiederverwendet und zu Anwendungssystemen montiert werden können.

1.3 Gang der Arbeit

Die vorliegende Arbeit ist in drei Teile gegliedert, jeder der Teile enthält mehrere Kapitel. Der Gang der Arbeit kann anhand des bereits eingeführten Software-Systemmodells dargestellt werden (siehe Abbildung 1.3).

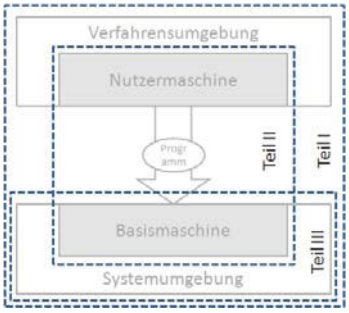

Abbildung 1.3: Gang der Arbeit anhand Software-Systemmodell (nach [Fer+96b, 46])

Teil I setzt den Rahmen der Arbeit. Er führt in den Gegenstand der Untersuchung, die Entwicklung betrieblicher Anwendungssysteme, ein. Kapitel 2 legt die Grundlagen für das

Verständnis von Anwendungssystemen im betrieblichen Kontext. Das darauf folgende Kapitel 3 stellt allgemein Konzepte und Methoden des *Software Engineering* vor, dessen Aufgabe die Spezifikation von Nutzermaschinen und ihre Abbildung auf Basismaschinen mit Hilfe von Programmen ist [Fer+96b, 46]. Darauf aufbauend geht Kapitel 4 speziell auf die Verfolgung von Integrationszielen bei der Ableitung und Realisierung betrieblicher Anwendungssysteme ein, die als ein entscheidendes Qualitätsmerkmal für die Güte der Anwendung betrachtet werden müssen.

Teil II widmet sich dem Ziel der Arbeit, Umfang und Komplexität der Systementwicklungsaufgabe zu verringern. In Kapitel 5 werden Geschäftsprozesse als Ausgangspunkt der fachlichen Modellierung betrieblicher Anwendungssysteme auf Basis der SOM-Methodik eingeführt und als Grundlage für die Spezifikation von Nutzermaschinen beschrieben. Kapitel 6 umfasst software-technische Konzepte zur Realisierung von Nutzermaschinen und führt zum abstrakten Lösungsansatz der Arbeit hin. In Kapitel 7 wird der Entwurf und Einsatz von Frameworks zur Abbildung von Nutzermaschinen beschrieben.

Teil III stellt das konkrete Lösungsverfahren der Arbeit, das Software-Framework **moccabox** vor. In Kapitel 8 wird der fachliche Entwurf des Frameworks dargestellt sowie die Software-Architektur erläutert. Das anschließende Kapitel 9 stellt einige Komponenten der softwaretechnischen Realisierung des **moccabox**-Framework vor sowie die dafür verwendeten Basismaschinen.

Das abschließende Kapitel 10 fasst die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf weitere Entwicklungen des Frameworks **moccabox** sowie der geschäftsprozessmodellgetriebenen Entwicklung betrieblicher Anwendungssysteme.

1.4 Umfeld der Arbeit

Die vorliegende Arbeit entstand im Rahmen der Tätigkeit als wissenschaftlicher Mitarbeiter von 2001 bis 2005 am Centrum für betriebliche Informationssysteme (Ce-bIS), einem Institut der Universität Bamberg. Das Ce-bIS widmet sich im Bereich der Wirtschaftsinformatik der auftragsbezogenen Forschung und Entwicklung sowie Schulung und Beratung im Bereich

betrieblicher Informationssysteme. Im Bereich der Software-Entwicklung wurden in Projekten mit Kooperationspartnern aus Wirtschaft und Verwaltung die Spezifikation von Anwendungssystemen aus Anforderungen betrieblicher Informationssysteme fachlich abgeleitet (*top-down*-Vorgehen). Für die software-technische Implementierung der automatisierten (Teil-)Aufgaben des jeweiligen Informationssystems wurde das Framework **moccabox** entwickelt und eingesetzt.

1.5 Konventionen

Für ein besseres Verständnis und eine leichtere Lesbarkeit werden zentrale Begriffe in der vorliegenden Arbeit schrittweise eingeführt. Definitionen von Begriffen sind **fett** formatiert. Begriffe, die als Bezeichner verwendet werden, werden *kursiv* formatiert. Begriffe aus Fremdsprachen, sofern sie nicht direkt als Fachbegriff Eingang gefunden haben, werden ebenfalls *kursiv* dargestellt. Besondere Hervorhebungen und Betonungen von Textstellen werden unterstrichen. Programmcode und Bezeichnungen implementierter Klassen werden in `Typewriter` dargestellt. Aus Platzgründen wird der komplette Paketname bei der ersten Erwähnung der Klasse als Fußnote angegeben.
Am Ende der Arbeit befindet sich ein Stichwortverzeichnis, in dem zentrale Begriffe der Arbeit nachgeschlagen werden können. Das Abkürzungsverzeichnis am Anfang der Arbeit beinhaltet alle in der Arbeit verwendeten Abkürzungen.
Wird eine weibliche bzw. männliche Personenbezeichnung verwendet, so schließt das die jeweils andere Personenbezeichnung mit ein. In der Regel wird die männliche Form verwendet.
Werden in einem Zusammenhang mehrere Quellen zitiert, so werden die Quellen in chronologischer Reihenfolge, beginnend mit der ältesten, aufgelistet.

Teil I

Betriebliche Anwendungssysteme

2 Grundlagen betrieblicher Anwendungssysteme

Das Sachziel der Systementwicklungsaufgabe ist die Entwicklung eines betrieblichen Anwendungssystems [FeSi06, 459]. Eine einheitliche begriffliche Definition und Verwendung für die Bezeichnung betriebliches Anwendungssystem existiert in der Literatur nicht (siehe [FeSi06, 9f]). In diesem Kapitel wird das Begriffsverständnis der Arbeit vorgestellt, das auf den in [Sinz97] sowie [FeSi06] beschriebenen Definitionen basiert.

Im betrieblichen Kontext dienen Anwendungssysteme der automatisierten Durchführung betrieblicher Aufgaben. Eine betriebliche Aufgabe ist auf das Unternehmensziel gerichtet, dabei werden Zweck und Art der Aufgabendurchführung durch Sach- sowie Formalziele bestimmt (siehe Abbildung 2.1). Bei der Beschreibung einer Aufgabe wird zwischen Außen- und Innensicht unterschieden [FeSi06, 19f]. Aus Außensicht werden Ziel und Zweck der Aufgabe beschrieben sowie auslösende Vor- und eintretende Nachereignisse. Vom verwendeten Lösungsverfahren - z.B. durch ein Anwendungssystem (automatisiert) oder eine Person (nicht-automatisiert) - wird dabei abstrahiert. Das ermöglicht die Beschreibung der Aufgabe, unabhängig vom gewählten Aufgabenträger. Das Lösungsverfahren wird erst bei der Innensicht der Aufgabe betrachtet.

Informationsverarbeitungsaufgaben werden im betrieblichen Kontext vom betrieblichen Informationssystem ausgeführt. Dessen begriffliche Abgrenzung und Architektur wird in Kapitel 2.1 vorgestellt. Betriebliche Anwendungssysteme stellen hierzu maschinelle Aufgabenträger dar und werden im darauf folgenden Kapitel 2.2 eingeführt.

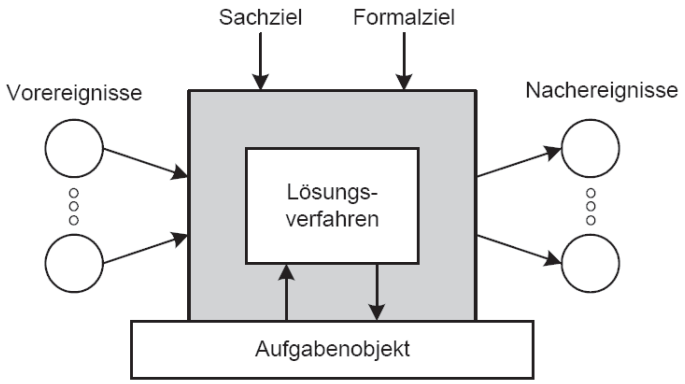

Abbildung 2.1: Betriebliche Aufgabe [FeSi06, 92]

2.1 Betriebliches Informationssystem

Ausgehend von einem betrieblichen System, ist das **betriebliche Informationssystem** dasjenige Teilsystem, welches die Beziehungsart **Information** unterstützt[1]. Primäre Aufgabe des Informationssystems ist die auf die Unternehmensziele ausgerichtete Planung, Steuerung und Kontrolle der betrieblichen Prozesse, einschließlich der Interaktionen mit der Umwelt [Fer+96b, 9]. Neben der Erfüllung dieser Lenkungsaufgabe können vom betrieblichen Informationssystem auch betriebliche Leistungen erzeugt werden, sofern diese aus Informationen bestehen.

Abstimmung der Komponenten

Das Informationssystem der Organisation besteht aus drei Teilsystemen, die aufeinander abgestimmt sind: Aufgabenstruktur, Aufbauorganisation und Anwendungssysteme [Sinz99, 20]. Da die drei Teilsysteme wechselseitig voneinander abhängig sind, muss die evolutionäre Weiterentwicklung der drei Teilsysteme kontinuierlich aufeinander abgestimmt werden. Dies kann nur gelingen, wenn zwischen diesen Teilsystemen Strukturähnlichkeiten und -analogien bestehen. Das im Rahmen dieser Arbeit entwickelte Framework soll eine abgestimmte Entwicklung betrieblicher Anwendungssysteme unterstützen, mittels der durchgängigen Orientierung an den Geschäftsprozessen der Unternehmung.

[1] Komplementär dazu unterstützt das Basissystem die Beziehungsart Nicht-Information (z.B. Güterflüsse, Zahlungsflüsse, physische Dienstleistungen), das in dieser Arbeit nicht weiter betrachtet wird.

Beschreibungsebenen betrieblicher Informationssysteme

Für die Analyse und Gestaltung von betrieblichen Informationssystemen können zwei Beschreibungsebenen unterschieden werden (siehe Abbildung 2.2): die **Aufgabenebene** sowie eine korrespondierende **Aufgabenträgerebene**. Die Aufgabenebene besteht aus der Menge aller Informationsverarbeitungsaufgaben (A_1 bis A_5), die durch Informationsbeziehungen verknüpft sind. Dabei kann es sich, wie oben beschrieben, um Lenkungsaufgaben handeln, aber auch um Durchführungsaufgaben zur Erbringung von Dienstleistungen. Zur kooperativen Ausführung der Informationsverarbeitungsaufgaben werden diese maschinellen (R_1 sowie R_2) oder personellen Aufgabenträgern (P_1 sowie P_2) zugeordnet, die durch Kommunikationsbeziehungen verbunden sind [Sinz97, 2]. Die Menge aller Aufgabenträger bildet die Aufgabenträgerebene. Die Unterscheidung der beiden Ebenen dient dem Ziel, Freiheitsgrade alternativer Realisierungsformen aufdecken und nutzen zu können [Fer+96b, 10].

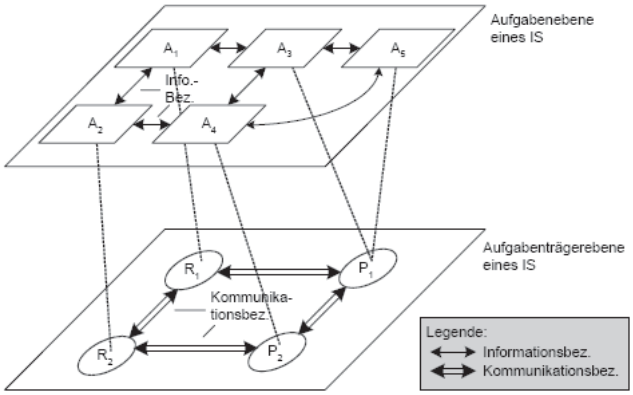

Abbildung 2.2: Aufgabenebene und Aufgabenträgerebene eines IS [FeSi06, 3]

Betriebliche Anwendungssysteme stellen in diesem Zusammenhang maschinelle Aufgabenträger zur Durchführung betrieblicher Informationsverarbeitungsaufgaben dar (z. B. (R_1 sowie R_2) in Abbildung 2.2). Die zugehörigen Informationsbeziehungen und Kommunikationssysteme in betrieblichen Informationssystemen sind in Abbildung 2.3 dargestellt.

Architektur betrieblicher Informationssysteme

Eine Architektur kann allgemein als eine abstrakte, ganzheitliche Betrachtung von Strukturen und Mustern mit Planungscharakter beschrieben werden [Bas+01, 19f]. **Informationssystem-**

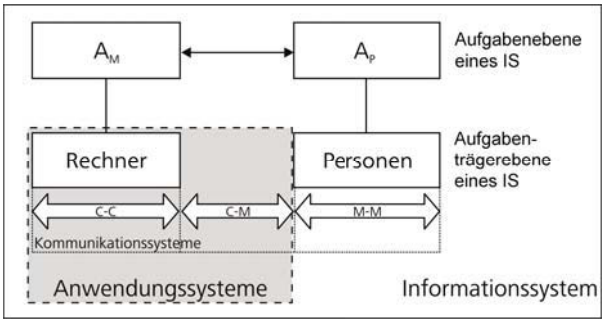

Abbildung 2.3: Informationsbeziehungen und Kommunikationssysteme im IS [FeSi06, 4]

Architekturen[2] stellen ein wichtiges Hilfsmittel zur ganzheitlichen Analyse und Gestaltung sowie zur zielgerichteten Nutzung von Informationssystemen dar [Sinz97, 2]: sie umfassen den Bauplan des jeweiligen Informationssystems sowie die zugehörigen Konstruktionsregeln des Bauplans [FeSi06, 185]. Der Bauplan bildet das zu Grunde liegende Informationssystem einem Metamodell entsprechend als Modellsystem ab. Das Metamodell gibt die verfügbaren Bausteine sowie deren Beziehungen zueinander als Konstruktionsregeln vor.

Ein für die Analyse und Gestaltung von Informationssystemen geeignetes Modellierungskonzept muss ein Metamodell besitzen, welches die ganzheitliche und detaillierte Beschreibung sowohl von dessen Aufgaben- als auch dessen Aufgabenträgerebene umfasst. Das **Semantische Objektmodell** (siehe u. a. [FeSi06, 185ff]) ist ein Modellierungskonzept, das eine besondere Durchgängigkeit aufweist [Ohle98, 89] und den Zusammenhang zwischen Unternehmenszielen, Geschäftsprozessen und Anwendungssystem herstellt [Stei97, 170]. Seine Fachkonzepte sind in dieser Arbeit zentral für die Beschreibung von Architekturen sowie die Abbildung fachlicher Modelle auf software-technische Komponenten. Eine Einführung und Beschreibung folgt in Kapitel 5.3.

2.2 Betriebliche Anwendungssysteme

Unter einem **betrieblichen Anwendungssystem** wird das gesamte automatisierte Teilsystem eines betrieblichen Informationssystems verstanden [Ambe99, 11]. Um diesem als Aufgabenträger dienen zu können, muss es Objekte sowie Operatoren anbieten, welche mit den Aufgabenobjekten und Verrichtungen der Informationsverarbeitungsaufgaben korrespondieren [Sinz99,

[2] Gleiches gilt auch für Anwendungssystem-Architekturen.

16]. Die zugehörige Systemabgrenzung wird durch folgende Schritten vollzogen (vgl. [Fer+96b, 46]):

- Abgrenzung des betrieblichen Systems gegenüber Umwelt
- Abgrenzung des betrieblichen Informationssystems gegenüber Basissystem
- Abgrenzung des Gesamt-Anwendungssystems gegenüber dem nicht-automatisierten Teil des betrieblichen Informationssystems
- Abgrenzung mehrerer Teil-Anwendungssysteme

Analog zum betrieblichen Informationssystem werden die Aufgabenebene und die zugehörige Aufgabenträgerebene betrieblicher Anwendungssysteme unterschieden [Fer+96b, 11]. Die Aufgabenebene umfasst automatisierte Informationsverarbeitungsaufgaben sowie deren Beziehungen untereinander. Die Aufgabenträgerebene repräsentiert die verwendeten Rechner- und Kommunikationssysteme einschließlich der korrespondierenden System-Software.
Zur Darstellung der Struktur eines Anwendungssystems dient das in Abbildung 2.4 gezeigte **Software-Systemmodell**, welches auf dem Modell der Nutzer- und Basismaschine aufbaut (siehe [FeSi06, 302ff] und Kapitel 3.3.1). Aus Außensicht handelt es sich bei einem Anwendungssystem um eine Nutzermaschine, die in eine **Verfahrensumgebung** eingebettet ist. Die Innensicht zeigt, wie die Objekte und Operatoren der Nutzermaschine durch ein Programmsystem auf eine entsprechende Basismaschine abgebildet werden. Die Basismaschine ist ihrerseits in eine **Systemumgebung** eingebettet. Die Spezifikation der Nutzermaschine sowie deren anschließende Abbildung auf eine Basismaschine durch ein zugehöriges Programm ist Aufgabe der Systementwicklung, die Thema des folgenden Kapitels ist.
In Abgrenzung zu personellen Aufgabenträgern für die Durchführung von Informationsverarbeitungsaufgaben, nennt AMBERG mehrere Nutzeffekte des Einsatzes betrieblicher Anwendungssysteme, die sich allesamt aus den Eignungsmerkmalen maschineller Aufgabenträger ableiten [Ambe99, 12]. Dazu gehören die zeitliche Verfügbarkeit, weitgehend frei wählbare Kapazitäten, Zuverlässigkeit der Aufgabendurchführung sowie die Transparenz der Aufgabenbearbeitung.

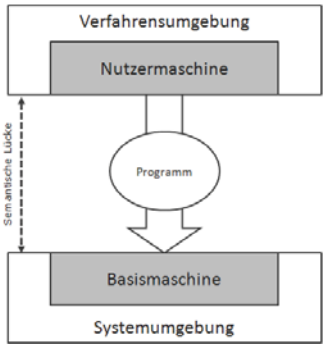

Abbildung 2.4: Software-Systemmodell [Fer+96b, 46]

Merkmale betrieblicher Anwendungssysteme

Sachlich können betriebliche Anwendungssysteme nach ihrem Einsatzbereich untergliedert werden. FERSTL ET. AL unterschieden dabei Entscheidungssunterstützungssysteme (EUS), betriebliche Expertensysteme sowie Büroinformationssysteme [Fer+96b, 37ff]. Orthogonal dazu können betriebliche Anwendungssysteme nach formalen Merkmalen differenziert werden. Dazu gehören Standard- und Individualsoftware, Erweiterbarkeit und Wiederverwendbarkeit, Endbenutzersystem sowie verteiltes betriebliches Anwendungssystem (ebenda).

Interaktive Anwendungssysteme

Unter einem **interaktiven Anwendungssystem** „[...] versteht man ein Computerprogramm, das für Benutzer Eingriffsmöglichkeiten im Kontroll- und Datenfluß aufweist“ [Star96, 34]. Ein Benutzer ist in diesem Kontext „[...] eine Person, die mit einem bestimmten Anwendungssystem und Interaktionsmedien zur Problemlösung interagiert“ [Star96, 34].

Ein interaktives Anwendungssystem ist folglich ein System, das hilft, fachliche Aufgaben in einem Anwendungsbereich zu erledigen. Die Programmausführung wird von Benutzereingaben oder Signalen technischer Systeme beeinflusst [Züll98, 3]. Die kooperative Erfüllung betrieblicher Aufgaben von personellen und maschinellen Aufgabenträgern erfordert Schnittstellen auf Seiten des Anwendungssystems für die Benutzereingabe und entsprechende Rückmeldungen. Aufgaben werden kooperativ durchgeführt, wenn sie teilautomatisiert sind [FeSi06, 208ff] (siehe auch Kapitel 5.3.2). Die Interaktion zwischen personellem Aufgabenträger und Anwendungssystem wird bei einem interaktiven Anwendungssystem durch eine entsprechende

Benutzungsoberfläche[3] abgebildet. Daraus leiten sich auch Anforderungen bezüglich des Echtzeitverhaltens des Systems ab [Fers92, 11f].

In der vorliegenden Arbeit ist immer von interaktiven betrieblichen Anwendungssystemen die Rede. Mit Blick auf die Nutzergruppe wird in diesem Zusammenhang auch von **Endbenutzersystem**[4] gesprochen [Fer+96b, 44]. Dieses Verständnis liegt der Arbeit zu Grunde. Aus Gründen der Lesbarkeit wird im Folgenden lediglich von betrieblichen Anwendungssystemen oder kurz Anwendungssystemen gesprochen.

[3] In der Literatur finden sich sowohl Benutzeroberfläche als auch Benutzungsoberfläche. In der Arbeit wird der Begriff Benutzungsoberfläche verwendet.

[4] In Abgrenzung zu den Nutzergruppen Entwickler und Administrator.

3 Entwicklung betrieblicher Anwendungssysteme

Die Systementwicklung kann als Aufgabe gemäß dem in Kapitel 2 eingeführten Aufgabenmodell interpretiert werden (siehe Abbildung 3.1). Ihr Sachziel ist die Entwicklung eines Anwendungssystems, das vorgegebene Anforderungen erfüllt [FeSi06, 459]. Die Systementwicklungsaufgabe ist ganzheitlich als Teilaufgabe der Gestaltungsaufgabe *Automatisierung betrieblicher Informationssysteme* (vgl. [Fers92, 4]) zu sehen, die sich aus der Zuordnung (teil-)automatisierbarer Aufgaben des Informationssystems zu maschinellen Aufgabenträgern [FeSi06, 4] ableitet. Ist kein geeignetes Anwendungssystem verfügbar, muss es entwickelt werden.

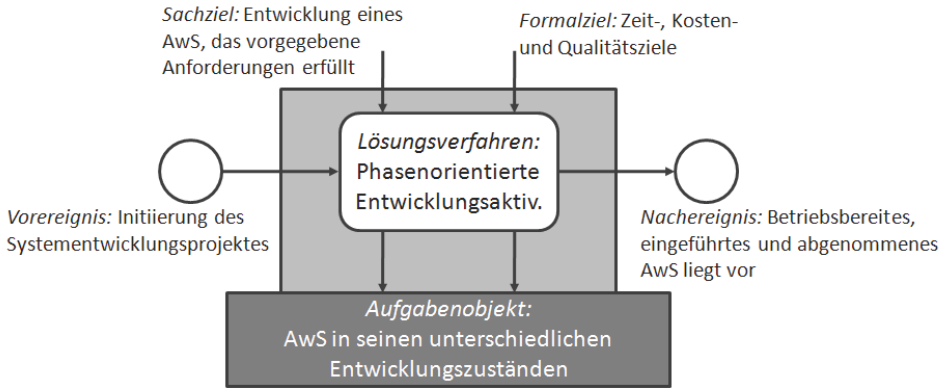

Abbildung 3.1: Aufgabenmodell der Systementwicklung [FeSi06, 459]

Der Aufgabe liegt ein Konstruktionsproblem zu Grunde, das auf Grund seiner Komplexität nicht in einem Schritt gelöst werden kann. Das Lösungsverfahren wird daher in mehrere Phasen zerlegt: Aufgabenanalyse und -definition, Software-Design sowie Realisierung [FeSi06, 461]. Im Zuge des phasenorientierten Vorgehens muss die Lücke zwischen fachlicher Anwendungsdomäne und software-technischer Implementierungsplattform überbrückt werden [Hof+96, 4]. Die Erstellung und Nutzung von Modellen der zu Grunde liegenden Systeme stellt dabei ein

wichtiges Hilfsmittel dar [FeSi06, 122][1]. Die Fachkonzepte des Modells eines Anwendungssystems dienen der Arbeit als Grundlage für den Entwurf einer geeigneten software-technischen Implementierungsplattform.
Grundlagen der Entwicklung und Strukturierung betrieblicher Anwendungssysteme sind Gegenstand dieses Kapitels. In den folgenden Abschnitten wird zunächst in Kapitel 3.1 auf die Software-Entwicklung als Konstruktionsproblem eingegangen. Anschließend wird in Kapitel 3.2 die Disziplin des Software Engineering eingeführt, dessen Aufgabe die Spezifikation von Nutzermaschinen und ihre Abbildung auf Basismaschinen mit Hilfe von Programmen ist [Fer+96b, 46]. In Kapitel 3.3 werden Modelle zur Strukturierung von Software-Systemen vorgestellt. In den Kapiteln 3.4 und 3.5 werden Ergebnis und Vorgehensmodelle zu Erstellung betrieblicher Anwendungssysteme vorgestellt.

3.1 Konstruktion von Systemen

Ein **Konstruktionsproblem** ist dadurch charakterisiert, dass es sich beim **Untersuchungsobjekt** um ein noch nicht existierendes System handelt, dessen Verhalten postuliert wird. Das System wird durch die Angabe einer Umgebung abgegrenzt. **Untersuchungsziel** und damit gesucht ist eine Struktur des Systems, mit der das geforderte Verhalten erreicht wird [Fers79, 44ff]. Durch die Vorgabe zu verwendender Komponenten und Teilstrukturen - z. B. Architekturkonzepte[2] (siehe [Sinz97]) - kann die Menge möglicher Strukturen eingeschränkt werden [Sinz99, 3], mit dem Ergebnis eines vereinfachten weil begrenzteren **Lösungsraums**.

Konstruktion betrieblicher Anwendungssysteme

Bei der Konstruktion eines betrieblichen Anwendungssystems ist das gewünschte Verhalten des zu entwickelnden Software-Systems vorgegeben. Gesucht ist eine Anwendung, die dieses Verhalten bewirkt. Dem kann die Konstruktion eines entsprechenden Modells des realen betrieblichen Systems vorausgehen, anhand dessen Struktur und Verhalten des zu konstruierenden Software-Systems abgeleitet wird.

[1] Siehe auch Kapitel 5.2.
[2] Der Einsatz von Frameworks unterstützt neben der Wiederverwendung von Code auch die Wiederverwendung von Design und somit der zu Grunde liegenden Architekturkonzepte [JoFo88], [John97].

In dieser Arbeit wird speziell die objektorientierte Anwendungssystementwicklung[3] betrachtet, deren Sachziel ein Anwendungssystem in objektorientierter Form ist [Popp94, 12]. Zu diesem Sachziel existieren als gemeinsames Zielsystem verschiedene Formalziele [Fers92, 11]. Diese nehmen Bezug auf die Korrektheit des Systems, sein Echtzeitverhalten, die Integration seiner Komponenten[4], seine Flexibilität sowie die Kosten und den Nutzen der automatisierten Aufgabendurchführung [Fers92, 11f]. Als weitere Formalziele gelten die Kriterien die Software-Qualität (siehe Kapitel 3.4).

3.2 Software Engineering

Um die Lösung von Konstruktionsproblemen technischer Systeme sind vor allem die Ingenieurswissenschaften bemüht [Fers79, 45]. Im Falle von Anwendungssystemen ist dies die Disziplin des **Software Engineering**[5], die bei POMBERGER & BLASCHEK wie folgt definiert wird [PoBl96, 2]:

> „Software Engineering ist die praktische Anwendung wissenschaftlicher Erkenntnisse für die wirtschaftliche Herstellung und den wirtschaftlichen Einsatz qualitativ hochwertiger Software."

Allgemein ist die Aufgabe des Software Engineering also die Spezifikation von Nutzermaschinen und ihre Abbildung auf Basismaschinen mit Hilfe von Programmen [Fer+96b, 46]. Der Begriff des Engineering soll dabei zeigen, dass „[...] auch die Herstellung von Software den Charakter einer Ingenieursdisziplin aufweisen muss" [PoBl96, 2].
Um dem Rechnung zu tragen, muss der Herstellung eine strukturierte und ableitbare Spezifikation sowie Implementierung zu Grunde liegen. Doch gerade bei der software-technischen Realisierung des Fachkonzepts ist die semantische Lücke zu verfügbaren Implementierungsplattformen noch immer zu groß. Die vorliegende Arbeit widmet sich eben dieser Problematik. Dem entspricht auch, dass die Anforderungsanalyse in Zukunft weiter an Bedeutung zunehmen wird, während die software-technische Implementierung immer weniger eine kreative Entwickleraufgabe sein soll [Alk+03, 243ff]. Vor dem Hintergrund der Tatsache, dass die Fehler

[3] Zum Thema der Objektorientierung siehe Kapitel 6.3.
[4] siehe hierzu vor allem Kapitel 4
[5] Zur Definition von Software Engineering siehe auch [ShGa96], [Same97], [Somm07]

häufig bereits in der Anforderungsableitung entstehen, ist das eine Entwicklung hin zu mehr Software-Qualität. Der gesamte Bereich der Anforderungsableitung wird auch als **Requirements Engineering** bezeichnet.

Ausgangspunkt für die Erstellung betrieblicher Anwendungssysteme ist der **Anwendungsbereich**, der den Kontext bildet, in dem das zu entwickelnde System eingesetzt wird. Die Abgrenzung des Anwendungsbereichs erfolgt durch die Modellierung der relevanten Diskurswelt des Unternehmens sowie dessen Beziehungen zur Umwelt[6]. Die Konstruktion des Anwendungssystems erfolgt anschließend auf der Grundlage des Modells des Anwendungsbereichs.

3.3 Software-Architektur

Die Aufgabe des Software Engineering besteht in der Spezifikation von Nutzermaschinen und ihre Abbildung auf Basismaschinen mit Hilfe von Programmen. Diese Nutzermaschinen entsprechen den zu entwickelnden Aufgabenträgern für die Aufgaben eines betrieblichen Informationssystems. Die Basismaschinen spezifizieren die dafür verfügbaren software-technischen Aufgabenträger [Fer+96b, 46f].

Die Realisierung komplexer Anwendungen erfordert aus Gründen der Komplexitätsreduktion eine *separation of concerns* [Evan04, 67]. Darunter wird die Trennung der automatisierten Verarbeitungsaufgaben nach verschiedenen Belangen (z. B. Datenverwaltung, Interaktion mit dem Nutzer) verstanden, die dem Entwickler die Konzentration auf bestimmte Teilbereiche des Designs ermöglicht. Die Struktur eines Software-Systems wird als **Software-Architektur** bezeichnet[7]. Sie wird als eine Menge von zueinander in Beziehung stehenden Komponenten beschrieben und ist eine Art Blaupause für das zu realisierende System und damit ein abstrakter Plan für dessen Konstruktion [Erl04, 56]. Für die komplette Beschreibung einer Architektur werden verschiedene Modelle und Sichten auf diese benötigt[8] [Hof+96, 9]. Als Vorteile des Einsatzes wohlstrukturierter Software-Architektur sind der hohe Abstraktionsgrad, die globale Sicht auf das System sowie die verbesserte Wiederverwendbarkeit zu nennen [Hof+96, 5]. Die

[6] Siehe hierzu Kapitel 5.2 und Kapitel 5.3.

[7] Gleichzeitig wird der Begriff der Software-Architektur häufig für die Disziplin der Strukturierung von Systemen verwendet. In dieser Arbeit wird jedoch das Ergebnis der Strukturierung darunter verstanden. Zur Architektur von Systemen im Allgemeinen siehe auch [MaRe00].

[8] Zum Thema Modellbildung und Sichten siehe auch Kapitel 5.3.

gewählte Architektur hat folglich großen Einfluss auf die Flexibilität des zu konstruierenden Systems.

Es werden verschiedene **Architekturstile** unterschieden, denen jeweils ein anderes Paradigma zu Grunde liegt. Eine Liste mehrerer Stile und deren Kategorisierung findet sich in [ShGa96, 20ff]. Einer dieser Stile ist die Schichtenarchitektur (*Layered Architecture*) eines Systems (siehe u.a. [Dijk68], [ShGa96], [Bas+01], [Bus+00], [Noac01], [Szyp02], [Evan04])[9]. Das zu Grunde liegende **Layer-Muster**[10] wird bei BUSCHMANN ET AL. folgendermaßen beschrieben [Bus+00, 32]:

> „Mit Hilfe des *Layer-Musters* lassen sich Anwendungen strukturieren, die in Gruppen von Teilaufgaben zerlegbar sind und in denen diese Gruppen auf verschiedenen Abstraktionsebenen liegen."

Bei der Entwicklung betrieblicher Anwendungssysteme ist das Schichtenmodell das Ergebnis der Schichtenbildung als Phase des Entwurfsprozesses [Noac01, 269ff]. Die Strukturierung von Anwendungssystemen durch Schichtenbildung dient der *Flexibilität* und der *Reduzierung der Komplexität* [FeSi06, 304]. Eine solche Referenz-Architektur ist in Abbildung 3.2 dargestellt. Betriebssystem und Programmiersprache bilden die Basis der Architektur, auf der ein technisches Basis-Framework aufbaut (z. B. Persistenz, GUI etc.) und Dienste entsprechend der technischen Domäne anbietet[11]. Die Anwendung stützt sich auf ein fachliches Framework, welches Konzepte aus der Anwendungsdomäne[12] (hier *betriebliche Anwendungssysteme*) umfasst, sowie das Basis-Framework.

Im Kontext einer Ableitung der Anforderungen aus der Anwendungsdomäne (*Domain-Driven Design*) zielt die Schichtenbildung vor allem auf die Abgrenzung der zu modellierenden Domäne der Diskurswelt und deren Abbildung in einer Schicht. EVANS sieht dieses Vorgehen einer klaren Trennung von Geschäftslogik und unterstützender Funktionalität als nicht selbstverständlich

[9] Als sehr bekannte Schichtenarchitektur sei hier das OSI-7-Schichtenmodell der *International Standardization Organization* (ISO) (z. B. in [Tane01]) erwähnt.

[10] Häufig werden die Begriffe *Layer* und *Tier* - beides im Deutschen mit *Schicht* bezeichnet - in der Literatur synonym verwendet. *Tier* hat jedoch die Bedeutung einer physischen Trennung der ausführenden Rechner, die bei unterschiedlichen *Layern* nicht notwendigerweise der Fall ist (zu Client-/Server-Architekturen und *Tier* siehe u.a. [Orf+97]).

[11] In dieser Arbeit handelt es sich um Dienste für die Domäne *Business*. Weitere Domänen sind beispielsweise Echtzeit- und eingebettete Systeme [StVö05, 136].

[12] Die Konzepte der *SOM-Methodik* (siehe Kapitel 5.3) sind Interface Objekte, Vorgangsobjekte und konzeptuelle Objekte.

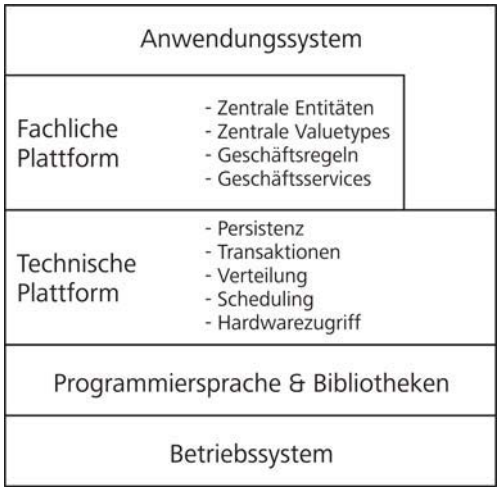

Abbildung 3.2: Schichten des Architektur-Referenzmodells [StVö05, 135]

an, sondern bemerkt hierzu [Evan04, 69]:

> „In an object-oriented program, UI, database, and other support code often gets written directly into the business objects. Additional business logic is embedded in the behavior of UI widgets and database scripts. This happens because it is the easiest way to make things work, in the short run."

Die Strukturierung von Programmen erfolgt durch korrespondierende Modelle. Solche Modelle beschreiben spezifische, vereinfachende Sichten auf diese [FeSi06, 299]. In Kapitel 3.3.1 wird das Nutzer- und Basismaschinenkonzept vorgestellt, in Kapitel 3.3.2 das dazu orthogonale ADK-Strukturmodell. Kapitel 3.3.3 behandelt die domänenbezogene Schichtenarchitektur. Alle drei dienen der Strukturierung von Anwendungssystemen, stellen jedoch - wie AMBERG anmerkt - kein Modell für deren Entwicklung dar. Ein solches Modell ist das objektorientierte Architekturmodell, das auf den vorgenannten Konzepten aufbaut (Kapitel 3.3.4).

3.3.1 Nutzer- und Basismaschine

In Abbildung 3.3 ist das Modell der Nutzer- und Basismaschine dargestellt. Es beschreibt die Struktur einer programmgesteuerten Maschine aus der Außensicht (**Nutzermaschine**) und seiner zugehörigen Innensicht (**Basismaschine** und zugehöriges Programm), wobei das **Programm** die Nutzermaschine unter Verwendung der Basismaschine realisiert [FeSi84, 74-80],

[FeSi06, 302]. Struktur und Verhalten der Nutzermaschine werden durch Datenobjekte und Operatoren beschrieben.

Durch die Schnittstelle der Nutzermaschine wird aus Sicht des Nutzers von ihrer Realisierung abstrahiert[13]. Diese Strukturierung dient der Komplexitätsreduktion. Die Schnittstelle der Basismaschine stellt aus Sicht des Programms wiederum eine Nutzermaschine dar, deren Implementierung vor dem Nutzer verborgen ist. Sie kann ihrerseits unter Verwendung weiterer Basismaschinen realisiert sein, wie in Abbildung 3.3 zu sehen ist.

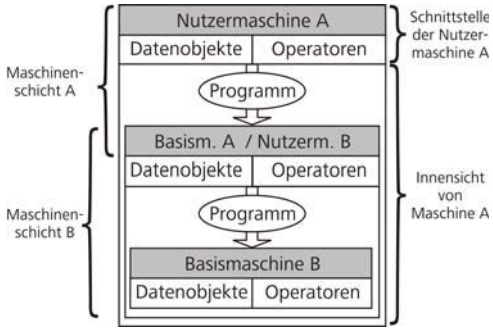

Abbildung 3.3: Schichtung von Nutzer- und Basismaschinen [FeSi06, 303]

Ein betriebliches Anwendungssystem stellt aus Außensicht eine Nutzermaschine dar, die dem Anwender über ihre Schnittstelle Datenobjekte und Operatoren bereitstellt. Diese stimmen mit den Aufgabenobjekten und Verrichtungen der automatisierten (Teil-)Aufgaben des betrieblichen Informationssystems überein [Sinz99, 16f].

Bestandteil der Innensicht des Anwendungssystems sind eine oder mehrere Basismaschinen (z. B. Programmiersprachen, Betriebssysteme etc.), die eine Schnittstelle bieten, auf deren Grundlage die Datenobjekte und Operatoren der Nutzermaschine realisiert werden. Diese werden mittels des Programm(system)s auf die Basismaschinen abgebildet. Das Programm(system) überbrückt die *semantische Lücke* zwischen Nutzer- und Basismaschine [FeSi06, 305], wodurch die Schnittstelle der Nutzermaschine von der Realisierung der Nutzermaschine abstrahiert.

Das im Rahmen dieser Arbeit entwickelte Framework stellt aus Sicht des zu konstruierenden betrieblichen Anwendungssystems (Nutzermaschine) eine Basismaschine dar, durch die von darunter liegenden Basismaschinen wie Betriebssystem, DBMS usw. abstrahiert werden kann.

[13] Vgl. auch das Prinzip der Abstraktion als Operation zur Erreichung von Software-Qualität in [KaBa94].

3.3.2 Das ADK-Strukturmodell

Das ADK-Strukturmodell von FERSTL & SINZ stellt eine zum Schichtenmodell der Nutzer- und Basismaschine orthogonale Gliederung dar (vgl. [FeSi84], [FeSi06, 305ff]). Es unterscheidet die Teilsysteme[14] **Anwendungsfunktionen** (A), **Datenverwaltung** (D) sowie **Kommunikation** (K), die weiter unterteilt wird in Mensch-Computer-Kommunikation (K_P) und Computer-Computer-Kommunikation (K_M) (siehe Abbildung 3.4). Diese Strukturierung ist mit der Strukturierung der betrieblichen Aufgabe vergleichbar (siehe Abbildung 2.1): Der Kommunikationsteil (K) entspricht den Ereignissen, der Datenhaltungsteil (D) dem Aufgabenobjekt und der Anwendungsteil (A) korrespondiert mit dem Lösungsverfahren der Aufgabe.

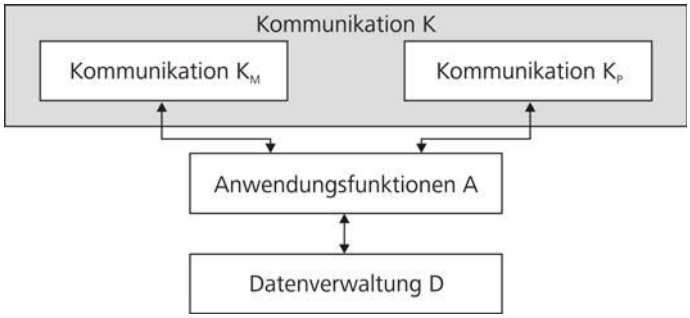

Abbildung 3.4: ADK-Strukturmodell [FeSi06, 305]

Die jeweiligen Teilsysteme können wiederum nach dem Modell von Nutzer- und Basismaschine strukturiert werden (Abbildung 3.5).
Die Trennung in die Teilsysteme A, D, K_P und K_M ermöglicht die Realisierung der jeweiligen Komponenten unter Verwendung spezifischer, häufig standardisierter Basismaschinen für jedes Teilsystem [FeSi06, 306]. Sind die Schnittstellen der Basismaschinen standardisiert, so können Programme flexibel zwischen Basismaschinen mit gleichen Schnittstellen portiert werden. Dadurch werden mit dem Ziel erhöhter Software-Qualität neben *Flexibilität* und *Reduzierung der Komplexität* auch *Standardisierung* und *Portierbarkeit* unterstützt.

- **Kommunikation** stellt die Schnittstelle zum interaktiven Anwendungssystem [Züll98, 3] für personelle Nutzer (MCK) und andere Anwendungssysteme (CCK) dar. Basismaschinen für Realisierung sind beispielsweise Netzwerkbetriebssysteme und User-Interface Management Systeme.

[14] Vgl. auch das Prinzip der Separation als Operation zur Erreichung von Software-Qualität in [KaBa94].

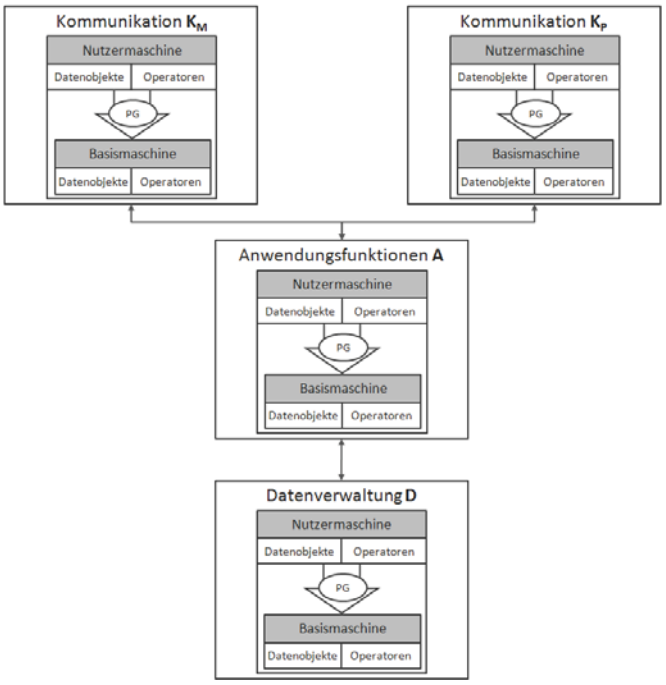

Abbildung 3.5: ADK-Strukturmodell eines Anwendungssystems (nach [FeSi06, 305])

- **Anwendungsfunktionen** unterliegen den häufigsten Änderungen, der Aufwand für Wartung kann hauptsächlich auf dieses Teilsystem konzentriert werden, mit dem Vorteil der Senkung des Wartungsaufwandes für das Gesamtsystem. Allgemeine und spezielle Programmiersprachen sind Beispiele für Basismaschinen der Anwendungsfunktionen.
- **Datenverwaltung** wird auf der Basis von Datenbankverwaltungssystemen (DBVS) umgesetzt.

Das zu entwickelnde Framework stellt für die jeweiligen (Teil-)Nutzermaschinen entsprechende (Teil-)Basismaschinen zur Verfügung. Die Architektur des Framework ist damit strukturgleich mit dem zu konstruierenden betrieblichen Anwendungssystem.

3.3.3 Domänenbezogene Schichtenarchitektur

EVANS stellt im Kontext des *Domain-Driven Design* die in Abbildung 3.6 dargestellte Schichtenarchitektur vor [Evan04, 68-75], in der sich die Konzepte des ADK-Strukturmodells sowie der Nutzer- und Basismaschine wiederfinden. Vor allem der Begriff der Domäne soll hier in Bezug zu ADK und den SOM-Fachspezifikationen gesetzt werden.

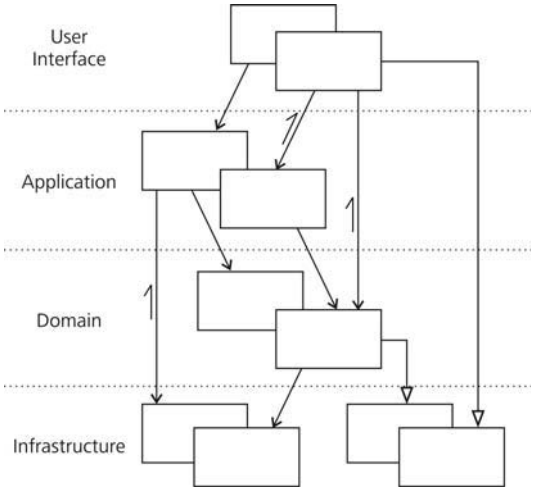

Abbildung 3.6: Software-Architektur-Schichten [Evan04, 68])

EVANS unterscheidet die folgenden Schichten[15]:

- **User Interface / Präsentationsschicht**

  Verantwortlich dem Nutzer Informationen anzuzeigen und seine Kommandos zu interpretieren. Beim Nutzer kann es sich auch um einen anderen Rechner handeln.

- **Schicht der Applikation**

  Definiert die Aufgaben, die das System erledigen soll und dirigiert die Domänenobjekte. In dieser Schicht werden keine Geschäftsregeln gehalten, sondern lediglich Domänenobjekte zur Erfüllung von Aufgaben koordiniert. Die Schicht hat keinen Zustand, der die Geschäftssituation widerspiegelt. Lediglich der Zustand einer Aufgabendurchführung kann wiedergegeben werden.

- **Schicht der Domäne / Modell**

  Verantwortlich für die Repräsentation der Konzepte des Unternehmens, Informationen über die Geschäftssituation und *Business Rules*.

- **Infrastruktur**

  Stellt generische technische Basismaschinen für die höheren Schichten zur Verfügung.

Die Schicht, die die Infrastruktur bereitstellt, ist orthogonal zu den drei anderen Schichten. Sie entspricht dem Konzept der Nutzer- und Basismaschine. Sie stellt den anderen Schichten jeweils

[15] Eine weitere Schichtenarchitektur mit den Schichten *Presentation*, *Domain* und *Data Source* findet sich bei FOWLER [Fowl03, 17-24]. Der Unterteilung wird in der vorliegenden Arbeit aber nicht gefolgt.

spezielle Dienste zu Verfügung. Die Schicht der Domäne geht hier über das Verständnis der Datenverwaltung im ADK-Strukturmodell hinaus, da Applikation und Domäne zusammen die eigentliche Anwendung beschreiben. Diese Einteilung wird im folgenden Kapitel noch etwas deutlicher werden.

3.3.4 Objektorientiertes Architekturmodell (ooAM)

Das **objektorientierte Architekturmodell** (ooAM) zerlegt die Funktionalität eines betrieblichen Anwendungssystems und ordnet jede (Teil-)Funktionalität eindeutig einer Komponente zu. Dabei unterscheidet das ooAM - wie in Abbildung 3.7 dargestellt - fachliche und technische Funktionalität sowie Basisfunktionalität [Ambe93, 33].

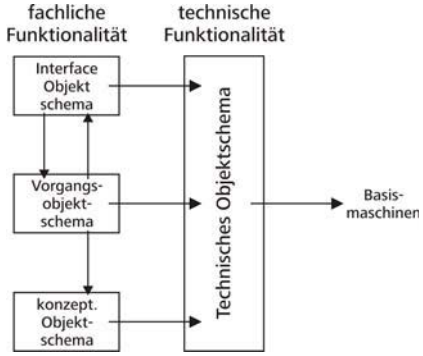

Abbildung 3.7: Das ooAM mit Schnittstellen [Ambe93, 37])

Die **fachliche Funktionalität** beschreibt die anwendungsspezifischen Funktionen, die sich aus fachlichen Anforderungen an ein spezielles AwS ergeben. Die Komponente der fachlichen Funktionalität umfasst konzeptuelle, Vorgangs- und Schnittstellen-Funktionalität. Das Framework **moccabox** realisiert keine fachliche Funktionalität, sondern stellt aus Sicht der Nutzermaschine *betriebliches Anwendungssystem* die Basismaschine zur Realisierung der Applikation dar. Ein Domänen-Framework (z.B. für Versicherungen oder Banken) hingegen stellt vorgefertigte fachliche Funktionalität zur Verfügung.

Die **technische Funktionalität** stellt anwendungsneutrale Funktionalität bereit. Technische Funktionalität muss vollständig unter Verwendung von Basisfunktionalität realisiert werden [Ambe93, 33]. AMBERG trennt sie von der fachlichen Funktionalität dadurch ab, dass sie nicht eindeutig der Vorgangs-, konzeptuellen oder Schnittstellen-Funktionalität zugeordnet werden kann [Ambe93, 34]. Ein Framework für die Entwicklung betrieblicher Anwendungssysteme erbringt

aus Sicht der Anwendung anwendungsneutrale, aber an den fachlichen Konzepten orientierte Funktionalität. Es wird demzufolge für die Architektur des Framework (siehe Kapitel 8) unterschieden zwischen technischer Funktionalität für das Interface-Objektschema, das Vorgangsobjektschema sowie das konzeptuelle Objektschema.

Die **Basisfunktionalität** bezeichnet diejenige anwendungsneutrale Funktionalität, die von verfügbaren Basismaschinen bereitgestellt wird und umfasst z. B. Datenbanksysteme und Programmiersprachen. Im Rahmen der Arbeit werden zusätzlich solche Basismaschinen zur Basisfunktionalität gezählt, die als so genannte *Black-Box*-Komponenten verwendet werden und Basismaschinen für die Realisierung des Framework darstellen (z. B. Persistenzmechanismen, GUI-Frameworks etc.).

Abbildung 3.7 zeigt die Komponenten IOS, VOS, KOS und TOS. Jede der Komponenten besitzt eine klar abgegrenzte Aufgabenstellung [Ambe93, 35]. Die Ableitung der fachlichen Funktionalität, die für ein betriebliches Anwendungssystem gefordert wird, erfolgt durch den Einsatz der Modellierungsmethodik SOM, die in Kapitel 5.3 vorgestellt wird. Da Modellierungsmethodik und Architekturmodell den gleichen fachlichen Konzepte folgen, verringert sich bei deren durchgängigem Einsatz die semantische Lücke zwischen fachlicher Spezifikation und technischer Realisierung.

3.3.5 Service-orientierte Architekturen

Eine neuere Entwicklung stellen **Service-orientierte Architekturen** (**SOA**) dar, die häufig als eine Weiterentwicklung der objektorientierten Strukturierung von Systemarchitekturen erachtet werden (siehe z. B. [Zach05]). Dieser Abschnitt definiert das der Arbeit zu Grunde liegende Verständnis einer SOA und stellt die verwendete objektorientierte Anwendungssystemarchitektur dazu in Beziehung.

Service-orientierte Architekturen wurden erstmals 1996 in einem Bericht der Gartner Group als „style of application partitioning and targeting (placement)“ erwähnt [ScNa96]. Gegenstand der beschriebenen Aufteilung (*partitioning*) sind dabei die namengebenden **Dienste** (englisch ***Services***). In der Literatur finden sich mittlerweile eine Vielzahl verschiedener Definitionen, die je nach Autor einen eher fachlichen (z. B. [Pall01], [LuTy03]) oder technischen Fokus (z. B. [Erl04], [Kra+04]) setzen. Der Dienst als zentrales Element einer SOA ist allen gemeinsam

sowie die notwendigen und in Abbildung 3.8 dargestellten Rollen *Service-Provider*, *Service-Broker* und *Service-Requestor*.

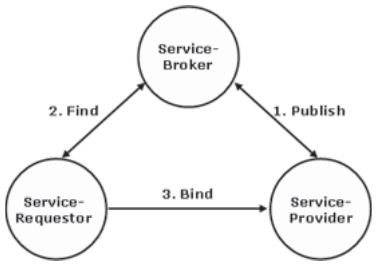

Abbildung 3.8: Service-orientierte Architektur

Zunächst ist eine service-orientierte Architektur ein technologieunabhängiges Konzept, das bezogen auf die Unternehmensarchitektur die Artefakte Services und Prozesse sowie die diese besitzende Organisation umfasst [LuTy03, 53]. Services werden orchestriert zur gemeinsamen Erfüllung der betrieblichen Aufgaben von Geschäftsprozessen, aufbauend auf bewährten Konzepten wie loser Kopplung, Wiederverwendung und Schnittstellen-Programmierung auf. Dabei abstrahiert der SOA-Ansatz - wie er von vielen Autoren definiert wird und wie er auch für diese Arbeit verstanden wird - vom Detaillierungsgrad der software-technischen Realisierung und fokussiert stattdessen die Architektur auf Geschäftsebene (z. B. [Pall01], [LuTy03], [Grou04]). LUBLINSKY und TYOMKIN unterteilen die Unternehmensarchitektur in eine Geschäfts-, eine Applikations-, eine Technologie- sowie eine querschnittliche Informationsperspektive. Eine SOA ordnen sie auf oberster Ebene ein [LuTy03, 53]:

> „SOA is an architectural style that promotes business process orchestration of enterprise-level business services."

Ausgehend von diesem Verständnis, können die eben vorgestellten Ebenen zur Informationssystemarchitektur aus Abbildung 3.9 folgendermaßen in Beziehung gesetzt werden:

- Die *Business*-Perspektive entspricht der Aufgabenebene.
- Die Perspektiven *Application* und *Technology* werden von den maschinellen Aufgabenträgern der Aufgabenträgerebene repräsentiert und stehen in Form von Nutzer- und Basismaschinen zueinander im Verhältnis.

- Die vierte Perspektive *Information* ist querschnittlich zu den vorgenannten und entspricht den Kommunikationssystemen als spezielle maschinelle Aufgabenträger für die Übertragungs- und Speicherungsaufgaben (siehe [FeSi06, 3f]).

Durch die höhere konzeptuelle Abstraktion können bei der Abgrenzung von Services Geschäftsprozesse stärker fokussiert werden. Dies wirkt sich auf die gesamte Informationssystemarchitektur eines Unternehmens aus (siehe u. a. [Bie+05], [Che+05]). Die Aufgabe der Konzeption und Realisierung einer service-orientierten Architektur wird somit zum Teil des betrieblichen Informationsmanagements (siehe dazu [FeSi06, 419–441]). Abbildung 3.9 zeigt nochmals die Architektur eines Informationssystems, wie sie in Kapitel 2.1 vorgestellt wurde[16].

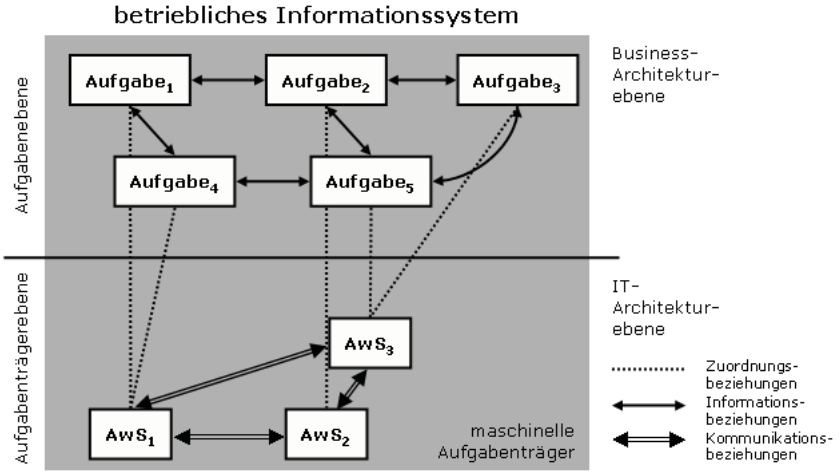

Abbildung 3.9: Informationssystemarchitektur (in Anlehnung an [FeSi06, 3]) und [FeSi06, 444]

Bildet man das Konzept der SOA auf die Informationssystemarchitektur ab, so muss der Ebene der Aufgaben, die durch Geschäftsprozesse ausgeführt werden, und der korrespondierenden Aufgabenträgerebene eine weitere Ebene, nämlich die der *Business Services*, hinzugefügt werden. Diese werden zur Ausführung von Geschäftsprozessen orchestriert und abstrahieren gleichzeitig aus Sicht der Business-Architektur-Ebene von den ausführenden Anwendungssystemen. Dieser Zusammenhang ist in Abbildung 3.10 dargestellt.
Zusammenfassend definiert Gartners *Glossary of Information Technology Acronyms and Terms* eine Service-orientierte Architektur wie folgt [Grou04, 325]:

SOA is an „[...] application topology in which the business logic of the application

[16] Auf die Zuordnung von Aufgaben zu personellen Aufgabenträgern wurde in dieser Abbildung verzichtet.

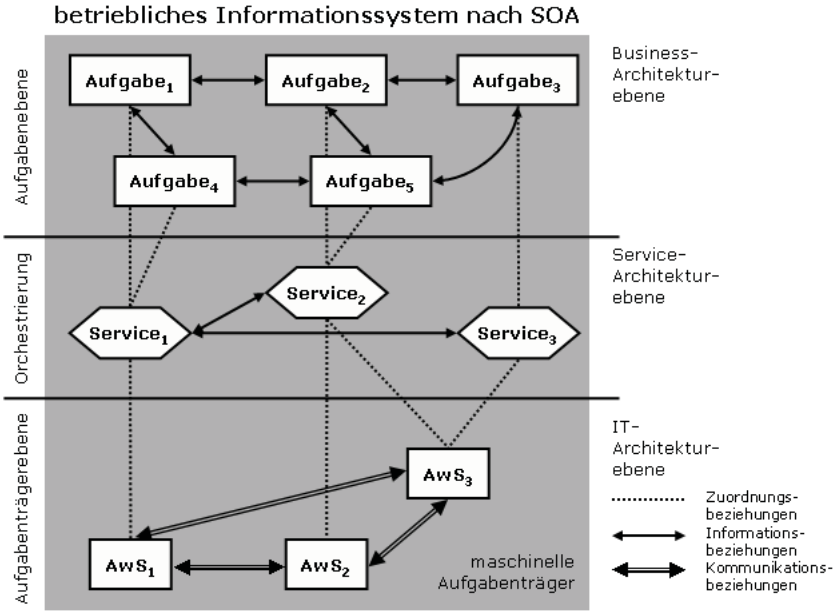

Abbildung 3.10: Informationssystemarchitektur mit SOA (in Anlehnung an [FeSi06, 3]) und [LuTy03, 53f]

> is organized in modules (services) with clear identity, purpose and programmatic-access interfaces. Services behave as 'black boxes': Their internal design is independent of the nature and purpose of the requestor. In SOA, data and business logic are encapsulated in modular business components with documented interfaces."

Demnach werden Services fachlich abgegrenzt und sind '*modular business components*'. Sie haben einen klaren Bezug zu den Geschäftsprozessen des betrieblichen Informationssystems. Dieses Verständnis findet sich auch bei PALLOS, der eine SOA beschreibt als Aggregation von Komponenten, die einem Geschäftszweck genügen [Pall01]. LUBLINSKY und TYOMKIN stellen den SOA-Ansatz im Kontext der Unternehmensarchitektur dar, die auf oberster Ebene Ziele abstrakt repräsentiert, die mit einer geeigneten Organisation und zugehörigen Prozessen verfolgt werden. Zur Umsetzung dieser Prozesse stehen unter anderem Anwendungssysteme zur Verfügung, die wiederum auf speziellen Technologien basieren [LuTy03]. Dieses Verständnis einer Unternehmensarchitektur deckt sich mit dem aus [FeSi06], wie es in Abbildung 5.4 dargestellt ist.

Ausgehend von den Beschreibungsebenen Aufgaben- und Aufgabenträgerebene, werden in der Informationssystemarchitektur automatisierbare Aufgaben durch Zuordnung zu maschinellen Aufgabenträger (Anwendungssystemen) automatisiert [FeSi06, 444]. Gegenstand dieser

Arbeit ist ein Framework, dass die software-technische Realisierung ebendieser betrieblichen Anwendungssysteme unterstützt. Für den Fortgang der Arbeit wird davon ausgegangen, dass der fachliche Entwurf dieser Anwendungssysteme nach geschäftsprozess-orientierten Kriterien gemäß SOM (siehe Kapitel 5.3) abgegrenzt wird und ihre software-technische Realisierung entsprechend dem objektorientierten Architekturmodell (siehe Kapitel 3.3.4) erfolgt. Aus Gründen der durchgängigen Ableitbarkeit sowie der semantischen Nähe von Fachkonzept und software-technischem Konzept baut die Architektur des zu entwickelnden Frameworks auf dem Prinzip der Objektorientierung auf. Die so entwickelten Applikationen können folglich im Rahmen einer service-orientierten Informationssystemarchitektur als maschinelle Aufgabenträger von Services dienen. Dies ist lediglich eine Frage der fachlichen Abgrenzung bei der Modellierung. Zudem können technische Services durchaus Bestandteil eines nach dem ooAM entwickelten Anwendungssystems sein.

3.4 Ergebnis der Anwendungssystementwicklung

Im Rahmen des Software Engineering stellt das fertiggestellte und eingeführte Anwendungssystem die vom Auftragnehmer gegenüber dem Auftraggeber zu erbringende **Leistung** dar [Fer+96b, 47]. Der Software Engineering-Prozess umfasst alle Teilleistungen, die notwendig sind, die Nutzeranforderungen in ein Software-System zu überführen [Hump90, 248]. Diese werden nacheinander dem Auftraggeber zur Verfügung gestellt und von diesem hinsichtlich der vereinbarten Anforderungen geprüft sowie abgenommen. Die einzelnen Leistungen können dabei von verschiedenen Auftragnehmern erbracht werden [Fer+96b, 48].

Die Überprüfung der jeweiligen Teilleistung sowie des fertig gestellten Anwendungssystems ist äußerst schwierig, da kaum objektive Kriterien dafür bestehen. Ein Kriterium der Endabnahme ist die Software-Qualität, die im folgenden vorgestellt wird. Die Qualität des Systems kann danach bemessen werden, inwieweit es die funktionalen und nicht-funktionalen Anforderungen an das System erfüllt.

Software-Qualität als Ergebnis der Anwendungssystementwicklung

Eng verknüpft mit der Architektur eines Software-Systems sind dessen Qualitätsattribute [KaBa94, 2], vor allem bei größeren Systemen. Das hier zu entwickelnde Framework soll neben der Unterstützung des Entwicklungsprozesses auch die Qualität des resultierenden Ergebnisses positiv beeinflussen. In der Literatur herrscht Einigkeit darüber, dass Software bestimmten überprüfbaren Qualitätskriterien entsprechen muss[17] (siehe u. a. [Hump90], [Jaco93], [PoBl96], [Somm07]). Uneinigkeit besteht hinsichtlich einer einheitlichen Definition von **Software-Qualität** [PoBl96, 9]:

> „Es fehlt eine allgemein verbreitete und anerkannte Definition der Begriffe *Qualität* und *Qualitätsmerkmal*, und es ist eine eindeutige, objektive Bewertung der Qualität eines Softwareprodukts nicht möglich. Qualität muss aber notwendigerweise ein Ziel der Software-Entwicklung sein und wirkt sich sowohl auf die *Planung* als auch auf die *Durchführung* und die *Kontrolle* der Softwareherstellung aus."

POMMBERGER & BLASCHEK beschreiben damit den Mangel einer eindeutigen und objektiven Vorgabe, was qualitativ hochwertige Software ist und wie daraus folgend deren Erstellung erreicht wird[18]. Die Qualitätsattribute des Systems müssen bei der Entwicklung systematisch verfolgt werden, denn Software-Qualität stellt sich nicht einfach ein, sondern muss im System explizit umgesetzt werden [Booc94, 348]. Der im Zuge der Realisierung zu entwickelnde Programm-Code muss den Qualitätsanforderungen entsprechen, was vor allem den Umfang der Systementwicklungsaufgabe erhöht. Der Einsatz von Software-Frameworks stellt bereits implementierte und getestete Komponenten zur Verfügung und kann dadurch den Umfang des zu realisierenden Programm-Code verringern.

In einer ersten Annäherung kann die Qualität eines Software-Systems nach der Summe seiner **nicht-funktionalen Eigenschaften** bemessen werden [Star05, 16]. Darunter werden Eigenschaften subsummiert, die von der eigentlichen Funktionalität des Systems unabhängig sind. Dazu zählen Skalierbarkeit, Modifizierbarkeit, Integrationsfähigkeit, Portierbarkeit, Performanz, Zuverlässigkeit, Einfachheit der Erstellung sowie Wiederverwendbarkeit (vgl. hierzu

[17] LUDEWIG betrachtet mangelnde Software-Qualität sogar als ein Kultur-Defizit [Lude96].

[18] Qualität ist immer aus der Perspektive des Kunden zu sehen und ist somit sehr subjektiv (vgl. [Pete00, 99ff]). Es wird hier nur auf allgemeine, objektivierbare Kriterien für Qualität von Software eingegangen.

auch [KaBa94]). Bei interaktiven System muss zudem der Software-Ergonomie besondere Aufmerksamkeit gewidmet werden [BeCu99].
BASS ET AL. identifizieren drei Klassen von Software-Qualitätsattributen [Bas+01, 79]:

1. Kriterien, die das Anwendungssystem selbst betreffen. Es werden solche unterschieden, die zur Laufzeit wahrnehmbar sind (z. B. Performanz, Benutzbarkeit etc.), und solche, die es nicht sind (z. B. Änderbarkeit, Wiederverwendbarkeit etc.).

2. Kriterien, die als geschäftsbezogene Qualität bezeichnet werden können (z. B. Time-to-Market) und die direkt von der verwendeten Architektur bzw. dem eingesetzten Framework beeinflusst werden.

3. Kriterien, die für die Architektur selbst wichtig sind.

Unabhängig von dieser Klassifikation werden im Folgenden die Merkmale für Software-Qualität der *International Standardization Organization* (ISO) vorgestellt [ISO01] sowie die sich daraus ergebenden Konsequenzen für das Framework **moccabox**: Funktionalität, Zuverlässigkeit, Benutzbarkeit, Änderbarkeit, Effizienz und Übertragbarkeit.
Die Architektur eines Software-Systems beeinflusst die Ausprägung dieser Qualitätsmerkmale entscheidend. Der Einsatz eines Frameworks bei der Entwicklung eines Anwendungssystems wiederum bestimmt dessen Software-Architektur (vgl. Kapitel 6.7). KAZMAN beschreibt in [KaBa94] Operationen, die zur Ableitung einer Software-Architektur verwendet werden und stellt diese den Qualitätsattributen gegenüber, wie in Tabelle 3.1 dargestellt.

Unit Operation	**Software Quality Factor**							
	Scal-ability	Modifi-ability	Integr-ability	Port-ability	Perfor-mance	Reli-ability	Easy Creat.	Reus-ability
Separation	+	+	+	+	+/-		+/-	+
Abstraction	+	+	+	+	-		+	+
Compression	-	-	-	-	+		+/-	-
Uniform Composition	+		+				+	
Replication	-	-		-	+/-	+	-	-
Resource Sharing		+	+	+	+/-	-	+	+/-

Tabelle 3.1: Qualitätsattribute und Software-Architektur [KaBa94, 10]

Das Framework muss die Erstellung betrieblicher Anwendungssysteme entsprechend der hier vorgestellten Merkmale unterstützen[19]. Die in Tabelle 3.1 dargestellten Operationen (*Unit Operations*) (vgl. [KaBa94, 7-11]) zur Ableitung einer Software-Architektur, finden sich auch in der durch das Framework vorgegebenen Architektur für betriebliche Anwendungssysteme wieder, die auf dessen Basis erstellt werden[20].

Separation ist angelehnt an die *separation of concerns* (vgl. [Parn01a]) Einheiten mit unterschiedlicher Funktionalität werden in unterschiedliche Komponenten verpackt, die ihre Funktionalität jeweils isoliert über eine wohldefinierte Schnittstelle veröffentlichen. Änderungen der Systemumgebung betreffen damit die Komponente nicht und Änderungen der Komponente betreffen die Systemumgebung nicht, solange die Schnittstelle sich nicht ändert. Dies hat besonders positiven Einfluss auf die Skalierbarkeit und die Modifizierbarkeit eines Systems sowie die Integrationsfähigkeit und Wiederverwendbarkeit der einzelnen Komponenten.

Im objektorientierten Architekturmodell wird diese Trennung nach IOS, VOS und KOS vollzogen (siehe Abbildung 3.7).

Abstraktion wird durch die Erstellung einer virtuellen Maschine erreicht (vgl. auch [FeSi06, 320f]). Abstraktion wird vor allem auch bei der Schichtenbildung[21] eingesetzt. Höher liegende Schichten abstrahieren dabei von der technischen Implementierung der darunter liegenden, die ihre Funktionalität über eine Schnittstelle anbieten[22]. Bezogen auf die Qualitätsattribute von Software, wird vor allem die Portierbarkeit und die Erleichterung der Erstellung positiv beeinflusst.

Im objektorientierten Architekturmodell wird die Abstraktion durch die Schichtung in fachliche und technische Funktionalität sowie zugehörige Basismaschinen erreicht (siehe Abbildung 3.7).

Kompression stellt eine der Separation entgegengesetzte Operation dar. Es wird vor allem zu Gunsten von Performanz und Entwicklungsgeschwindigkeit die Trennung nach Funktionalität teilweise aufgehoben. Die Kompression findet im Software-Engineering kaum

[19] Beim Framework handelt es sich um ein Software-System, dass natürlich selbst auch nach den vorgestellten Qualitätskriterien entwickelt werden muss.

[20] Vgl. das objektorientierte Architekturmodell in Kapitel 3.3.4.

[21] Siehe z. B. OSI-7-Schichtenmodell der *International Standardization Organization* (ISO) (z. B. in [Tane01]).

[22] Vgl. auch das Konzept der Nutzer- und Basismaschinen in Kapitel 3.3.1.

noch Anwendung, da vor allem Konzepte wie Datenkapselung, Objektorientierung und das Client/Server-Paradigma die Operation der Separation unterstützen.

Im Rahmen des objektorientierten Architekturmodells findet die Kompression keine Beachtung.

Einheitliche Komposition bezeichnet das Zusammenfassen von zwei oder mehr Systemkomponenten zu einer größeren Komponente. Sie erleichtert die Integration von Komponenten sowie die Skalierbarkeit des Systems als Ganzes.

Im objektorientierten Architekturmodell können z. B. für die Realisierung der technischen Funktionalität für das IOS[23] zusammengefasste Komponenten gemäß dem Model-View-Controller Paradigma[24] eingesetzt werden. Die Framework-Komponente zur Generierung der Benutzungsoberfläche enthält die einzelnen Komponenten *Model* (Applikation), *View* (Präsentation) sowie *Controller* (der die anderen beiden verknüpft).

Replikation bedeutet allgemein das Duplizieren einer Komponente innerhalb der System-Architektur. Dieses Vorgehen dient vor allem einer erhöhten Zuverlässigkeit und zum Teil einer verbesserten Performanz, die jedoch große Nachteile bei anderen Qualitätsattributen mit sich bringt.

In Software-Systemen wird Replikation für eine bessere Performanz vor allem durch das Puffern (*caching*) von Daten eingesetzt. Die Realisierung der technischen Funktionalität für das KOS kann den Zugriff auf die persistente Datenbasis durch diese Zwischenpufferung beschleunigen[25].

Gemeinsame Nutzung von Ressourcen ist eine Operation die entweder Daten oder Dienste kapselt und sie unabhängigen Clients zur Verfügung stellt. Die jeweilige Ressource wird dann durch einen Ressourcen-Manager verwaltet. Dadurch wird die Kopplung der Komponenten untereinander verringert, mit positiven Auswirkungen auf die Integrationsfähigkeit, die Portierbarkeit und die Modifizierbarkeit des Systems.

Die Verwaltung der Schemata des objektorientierten Architekturmodells sollen auf der Ebene der technischen Funktionalität explizit von einem Ressourcen-Manager - ein so

[23] Siehe auch Kapitel 8.2.2.

[24] Siehe Kapitel 8.2.2.2. Ähnlich ist das Konzept der Presentation-Abstraction-Control, das in Kapitel 8.2.2.1 vorgestellt wird.

[25] Siehe Kapitel 9.6.

genanntes *Repository* - übernommen werden[26].

Die Software-Qualitätsmerkmale werden im Folgenden einzeln diskutiert und in den Kontext der Framework-Entwicklung gestellt.

Funktionalität

Das Programmsystems erfüllt die zu Grunde gelegte Spezifikation und umfasst alle spezifizierten Funktionen. Es handelt sich dabei um die Übereinstimmung von *Spezifikation* und *Programmtext*.

Konsequenz:

Die Modellierung und Spezifikation des Anwendungssystems orientiert sich am SOM-Ansatz (siehe Kapitel 5.3), der durch seine ganzheitliche Betrachtung unter Verwendung des objektorientierten Paradigmas eine besondere Durchgängigkeit aufweist [Ohle98, 89]. Das Framework muss dem Entwickler eine entsprechende Architektur zur Verfügung stellen, um die durchgängig modellierten und verfeinerten Spezifikationen direkt als Anwendungssystem implementieren zu können (Verringerung der semantischen Lücke zwischen fachlichem Entwurf und softwaretechnischer Realisierung).

Zuverlässigkeit

Das Programmsystem muss entsprechend *korrekt* und *verfügbar* sein. Die „[...] Zuverlässigkeit (reliability) eines technischen Systems ist die Wahrscheinlichkeit, daß dieses System während eines vorgegebenen Zeitintervalls unter den vorgegebenen Bedingungen die Funktion erfüllt, für die es vorgesehen ist.“ [Kope76, 12].

Konsequenz:

Der Einsatz eines Framework als Basismaschine für das zu entwickelnde Anwendungssystem erlaubt die Wiederverwendung getesteter Klassen. Neben der Wiederverwendung von Code wird durch die gemeinsame Nutzung einer Architektur auch die Wiederverwendung von Design durch den Einsatz eines Framework ermöglicht [JoFo88, 24]. Die gewählten Basismaschinen des Framework müssen als zuverlässig getestet sein und das Framework selbst muss stabil sein, damit auf dieser Grundlage zuverlässige Anwendungssysteme realisiert werden können.

[26] Siehe Kapitel 9.3

Benutzbarkeit

Das System soll adäquat, erlernbar[27] und robust sein. Robustheit wird in diesem Zusammenhang als eine gewisse Unempfindlichkeit gegenüber Bedienungsfehlern, falschen Eingabedaten und Hardware-Fehlern verstanden. Es handelt sich hierbei um ein Qualitätsmerkmal, dass sich auf die Software-Ergonomie des Anwendungssystems bezieht, auf die in Kapitel 8.2.3 näher eingegangen wird.

Konsequenz:

Die Adäquatheit des Anwendungssystems ist durch das Framework kaum beeinflussbar. Es kann nur die unter Korrektheit beschriebene Architektur zur Verfügung stellen, um ein adäquates System entsprechend der mit SOM spezifizierten Anforderungen zu ermöglichen. Für die Erlernbarkeit muss das Framework eine Komponente zur durchgängigen Dokumentation des Anwendungssystems zur Verfügung stellen. Intuitiv begreifbare Bedienelemente sind für die Benutzungsoberfläche als Standard-Implementierung anzubieten. Robustheit gegenüber Fehleingaben und Bedienungsfehlern sind direkt im Framework zu implementieren. Dazu gehört auch eine Validierungskomponente, die verhindert, dass Fehleingaben in die Anwendungsfunktionalität gelangen und verarbeitet werden (siehe auch Konsistenz als Integrationsziel in Kapitel 4). Die technische Robustheit ist Ergebnis der gewählten Basismaschinen.

Änderbarkeit

Das Programmsystem soll für die Lokalisierung von Fehlerursachen, die Durchführung von Fehlerkorrekturen sowie zur Veränderung oder Erweiterung der Programmfunktionen geeignet sein. Es finden sich hierfür auch die Bezeichnungen Lesbarkeit, Erweiterbarkeit und Testbarkeit [PoBl96, 12]. FAYAD & CLINE sehen Anpassbarkeit (*adaptability*) jedoch nicht als generisches Qualitätskriterium. Vielmehr muss es nicht nur explizit in die Software entwickelt werden, sondern auch an die optimale Stelle [FaCl96, 58]. Man kann zwischen Anpassbarkeit bei Änderungen auf höherer Ebene (*extensibility* und *flexibility*) und auf niedriger Ebene (*tunability* und *fixability*) unterscheiden.

Konsequenz:

Das Framework muss es dem Anwendungsentwickler erlauben, standardisierte Komponenten

[27] Für die bessere Erlernbarkeit wird direkt durch den Modellierer bzw. den Entwickler die Dokumentation mit angelegt, die im Anschluss daran vom Nutzer auf verschiedene Weise abgerufen werden kann (Kapitel 8.5).

und Objekte flexibel zu parametrisieren [FaCl96, 59]. Die Funktionalität der erweiterbaren Schnittstellen muss so gekapselt sein, dass eine Änderung sich nicht auf das gesamte Programmsystem auswirkt. Das wird durch Trennung von Schnittstelle und Implementierung erreicht. Zusätzlich müssen während der Programmausführung präzise Fehlermeldungen (*exceptions*) ausgegeben werden, die das schnelle Auffinden von Fehlerquellen ermöglichen. Durch eine geeignete Architektur müssen Anpassungen der betrieblichen Abläufe nachgelagert leicht in das Anwendungssystem übernommen werden können.

Effizienz

Die benötigten und verfügbaren Ressourcen sollen bestmöglich genutzt werden.

Konsequenz:

Hier kann das Framework nur selbst auf die effiziente Nutzung der verfügbaren Ressourcen ausgerichtet werden. Dem Anwendungsentwickler müssen die benötigten Ressourcen verfügbar gemacht werden, so dass ein Zusatzaufwand durch zu viel Eigenentwicklung verhindert werden kann.

Übertragbarkeit

Das Anwendungssystem soll auf verschiedenen Rechnern und Plattformen lauffähig sein (siehe auch [PoBl96, 13]).

Konsequenz:

Die zu wählenden Basismaschinen für die Realisierung des Framework müssen plattformunabhängig sein, so dass die Anwendungsfunktionalität portiert werden kann. Desweiteren muss die Benutzungsoberfläche austauschbar sein, so dass Clients, die auf unterschiedlichen Rechner und Plattformen laufen darauf zugreifen können (z. B. mobile Endgeräte, Web-Browser).

Das zu entwickelnde Framework und seine Komponenten sollen die Erreichung von Software-Qualitätszielen weitestgehend unterstützen. Die Überprüfung der erreichten Qualität erfolgt durch die **Software-Qualitätssicherung**[28]. Diese beinhaltet alle systematischen Aktivitäten, die den Beweis für die Einsatzfähigkeit des gesamten Software-Produkts bringen. Die Bewer-

[28] Auch *Software Quality Assurance* (siehe u. a. [Will85], [Hump90], [Jaco93]). Metriken zur Messung der Qualität finden sich in [ChKe91]. Die Beschreibung von Szenarien zum Test von Software-Qualität wird u. a. in [Bas+01] und [Star05] vorgestellt.

tung einer Architektur wird in [Star05] beschrieben.

3.5 Vorgehen bei der Anwendungssystementwicklung

Da es sich bei der Entwicklung von Software um eine komplexe Aufgabenstellung handelt, die bezüglich Zeit-, Kosten- und Qualitätsaspekten hohen Anforderungen unterliegt ist ein ingenieursmäßiges, standardisiertes **Vorgehen** notwendig. Dazu wird der Entwicklungsprozess in mehrere Stufen bzw. **Phasen** zerlegt, die sequentiell, iterativ oder in einer Ableitung dieser beiden Formen durchlaufen werden. Ziel der Projektdurchführung ist die *Führung von Mehrpersonen-Projekten* [Balz82] in Form eines **Software Engineering-Prozesses**. Dieser umfasst die gesamte Menge notwendiger Aktivitäten, um aus Nutzeranforderungen Software zu machen [Hump90, 248].

FERSTL & SINZ beschreiben ein phasenorientiertes Lösungsverfahren der Systementwicklungsaufgabe in allgemeiner Form [FeSi06, 460ff]. Es umfasst die Phasen **Anforderungsanalyse und -definition (A)**, **Software-Design (D)** sowie **Realisierung (R)**. Die Phasen werden in der Regel im Rahmen eines Projektes durchlaufen, so dass den vorgestellten Phasen eine Phase Projektplanung (P) vorangeht und ihnen die Phase der Abnahme und Einführung (E) nachfolgt. Die Phasen A, D und R werden als Übergänge zwischen den Beschreibungsebenen der Systementwicklung interpretiert. Die Phasen lassen sich auf das Software-Systemmodell beziehen [FeSi06, 462ff], wie in Abbildung 3.11 dargestellt.

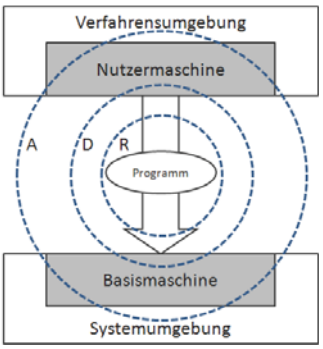

Abbildung 3.11: Phasen des Lösungsverfahrens der Systementwicklungsaufgabe im Software-Systemmodell [FeSi06, 463]

Die Phasen können wie folgt beschrieben werden [FeSi06, 462f]:

Phase A: Dient der Definition von Nutzermaschine und Verfahrensumgebung sowie einer groben Spezifikation der einzusetzenden Basismaschinen und Systemumgebung.

Phase D: Dient der Spezifikation der Software-Architektur des Produkts („Programmieren im Großen“). Durch sie wird die semantische Lücke zwischen Nutzer- und Basismaschine überbrückt.

Phase R: Dient der Implementierung der Programme des Anwendungssystems („Programmieren im Kleinen“). Die von der Software-Architektur spezifizierten einzelnen Bausteine werden in dieser Phase programmiert.

Die Phasen A, D und R beeinflussen den Entwurf und die Einsatzkriterien für ein Software-Framework. Für die Ausgestaltung der Phasen gibt es verschiedene Vorgehensmodelle. Ein **Vorgehensmodell**[29] beschreibt einen festgelegten organisatorischen Rahmen für die Software-Erstellung[30] [Fer+96b, 48f]:

> „Ein Vorgehensmodell strukturiert die Durchführung eines Projektes in Phasen und ggf. Teilphasen, spezifiziert die Reihenfolge- und Koordinationsbeziehungen sowie die Schnittstellen zwischen den Phasen (Phasenübergänge) und bestimmt die Granularität der in einem Phasenübergang zu übergebenden Leistungspakete.“

Da ein Framework zur Entwicklung betrieblicher Anwendungssysteme auch den Entwicklungsprozess unterstützen muss, werden im folgenden kurz hierfür relevante Vorgehensmodelle vorgestellt (siehe u. a. [Booc94], [Fric95], [Somm07], [Balz08]) und vor dem Hintergrund der Framework-Eignung betrachtet. Das Framework **moccabox** soll auch die Entwicklung der Benutzungsoberfläche unterstützen. Deshalb sind bei der Auswahl des Vorgehensmodells auch Kriterien der Entwicklung ergonomischer Benutzungsoberflächen zu beachten[31].

[29] Synonym auch als Prozessmodell [Balz08] oder *life cycle model* [Somm07] bezeichnet.
[30] Siehe auch [Boeh88, 61].
[31] Dieses Vorgehen wird auch als *Human-Centred Design Process* oder *User-Centred Design Process*, die auf dem ISO Standard 13407 basieren, bezeichnet [BeCu99].

3.5.1 Wasserfall-Modell

Das **Wasserfall-Modell** wird als erste durchdachte Vorgehensstrategie[32] in der Software-Entwicklung angesehen [Fric95, 45] und geht auf ROYCE zurück [Royc70].

Die zu durchlaufenden Phasen sind *Analyse*, *Definition*, *Entwurf*, *Implementierung*, *Abnahme und Test* sowie *Betrieb*. Das Modell wird sequentiell Phase um Phase im top-down-Verfahren durchlaufen. Es wird jede Aktivität in der richtigen Reihenfolge und vollständig durchgeführt. Dabei werden am Ende einer jeden Phase die Ergebnisse in einem *Review* erneut geprüft und dann in Form eines fertiggestellten Dokumentes an die folgende Phase weitergeleitet. Das Vorgehensmodell wird dementsprechend als *dokumenten-getrieben* bezeichnet [Balz08, 520]. Werden die Ergebnisse in der Folgephase als unzureichend oder falsch angesehen, kann die vorhergehende Phase erneut durchlaufen werden. Die Benutzerbeteiligung ist nur für die Phase der Definition vorgesehen. Entwurf und Implementierung erfolgen ohne Beteiligung des Auftraggebers [Balz08, 519-522].

Nachteilig wirkt sich aus, dass es nicht immer sinnvoll ist, alle Entwicklungsschritte sequentiell und vollständig in voller Breite durchzuführen [Boeh88, 63]. Vor allem bei langjährigen Projekten ist dieses Vorgehen zu idealtypisch. Es kommt hinzu, dass erst nach Fertigstellung des Systems geprüft werden kann, ob es den in der Definition festgelegten Anforderungen entspricht.

Die Entwicklung der Benutzungsoberfläche wird nicht explizit unterstützt, was sich im Hinblick auf das Erreichen qualitativ hochwertiger Software-Ergonomie (siehe Kapitel 8.2.3) als nachteilig erweist [Herd00, 20].

Weiterentwicklungen zu iterativen Vorgehensmodellen sind das Prototyping (siehe Kapitel 3.5.2) und das Spiralmodell (siehe Kapitel 3.5.3).

3.5.2 Prototyping

Das **Prototyping** ist eine Möglichkeit, während des Software-Entwicklungsprozesses für den Kunden ein greifbares Ergebnis vorzustellen[33]. Im Grunde ist das Prototyping ein Ansatz klas-

[32] BOEHM nennt als weiteres sehr frühes Vorgehen die *code & fix* Methode [Boeh88, 62], bei der man einfach Code schreibt und diesen anschließend um seine Fehler bereinigt. Dieses Vorgehen kann jedoch nicht wirklich als Strategie angesehen werden.

[33] Zum Thema Rapid Prototyping siehe auch [Mull90]

sischer Ingenieursdisziplinen. Den Impetus zur prototypischen Entwicklung in der Software-Erstellung beschreibt SOMMERVILLE folgendermaßen [Somm07, 443]:

> „Anhand von Software-Prototypen können die Benutzer erkennen, wie gut das System sie bei der Arbeit unterstützt. Sie gewinnen dadurch möglicherweise neue Ideen für Anforderungen und können Stärken und Schwächen in der Software aufspüren."

Im Gegensatz zum Wasserfall-Modell muss zu Beginn des Projektes keine vollständige Spezifikation der Anforderungen gegeben sein, die in vielen Fällen auch nicht zu leisten ist. Die Anforderungen werden beim Prototyping durch Rückgriffe auf frühere Phasen sukzessive verfeinert und ergänzt, wodurch das System iterativ erweitert wird [Herd00, 21]. Ergebnis ist ein Software-Prototyp, der ein ausführbares Modell mit wesentlichen Eigenschaften des Zielsystems darstellt [Pom+92, 2]. Dieses Modell dient als Grundlage für die Systemspezifikation und unterstützt die Kommunikation zwischen Kunden und Entwickler.
Es werden, wie in Abbildung 3.12 dargestellt, horizontale und vertikale Prototypen unterschieden [Balz08, 539].

- **Horizontale Prototypen** realisieren lediglich spezifische Ebenen des Systems (z. B. Benutzungsoberfläche), diese aber möglichst vollständig. Im Falle der Benutzungsoberfläche ist somit die gesamte Oberfläche zu sehen, die dahinter liegende Funktionalität fehlt jedoch (noch).

- **Vertikale Prototypen** realisieren ausgewählte Teile des zu entwickelnden Anwendungssystems vollständig über alle Ebenen. Es kann zum Beispiel die Funktionalität für das Erfassen eines neuen Kunden entwickelt werden von der Eingabe, über die Validierung und Verarbeitung bis zur Persistierung.

Je nach Beziehung des entworfenen Prototypen zum endgültigen Anwendungssystem kann dieser mit unterschiedlichen Zielen entworfen werden. Er kann einzig der Klärung von Problemen dienen. Der Prototyp kann aber auch Teil der Produktdefinition sein, ohne selbst ein Teil des Produktes zu werden (Wegwerf-Prototyp). Schließlich kann der Prototyp als *working prototype*

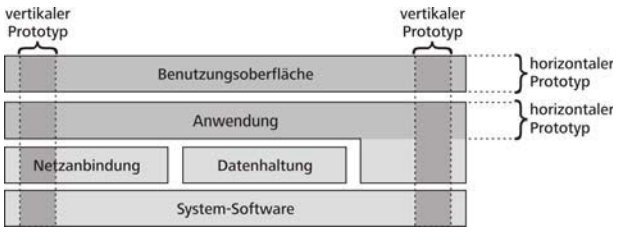

Abbildung 3.12: Horizontaler und vertikaler Prototyp [Balz08, 540]

durch inkrementelle Weiterentwicklung selbst zum marktfähigen Produkt werden. Dieses Vorgehen wird häufig bei der Entwicklung der Benutzungsoberfläche von Anwendungssystemen angewandt (vgl. auch [Herd00, 22f]).

Das Prototyping ermöglicht die Reduzierung des Entwicklungsrisikos durch die schnelle Entwicklung und Bewertung eines lauffähigen Modells und die daraus resultierende enge Einbindung des Auftraggebers. Wird der Prototyp aber nicht in das fertiggestellte System überführt, sondern verworfen, ist mit einem erhöhten Entwicklungsaufwand zu rechnen. Wird der Prototyp doch übernommen, besteht die Gefahr, dass geforderte Entwicklungs- und Qualitätsstandards nicht beachtet wurden.

Dem letztgenannten Nachteil kann durch den Einsatz von *Application Frameworks* (siehe hierzu Kapitel 6.7) für den Prozess des Prototyping begegnet werden [Pom+92, 4f]. Konventionelle Klassenbibliotheken eignen sich hingegen wegen ihres geringen Abstraktionsniveaus weniger, da der Entwickler dann einen Großteil der prototypischen Anwendung selbst realisieren müsste, anstatt auf Modulbausteine zurückgreifen zu können, die ein Framework bietet. Das in Kapitel 8 und Kapitel 9 vorgestellte Framework **moccabox** bietet die Möglichkeit des vertikalen Prototyping unter Berücksichtigung von Qualitätsstandards und Integrationszielen.

3.5.3 Spiralmodell

Das **Spiralmodell** [Boeh88] stellt eine Kombination des Wasserfall-Modells und des Prototyping dar. Es handelt sich um einen *risikogetriebenen* Ansatz [Boeh88, 61], denn das Erkennen und Beseitigen von Projekt- und Entwicklungsrisiken ist das zentrale Anliegen. Es werden insgesamt vier Schritte mehrfach zyklisch durchlaufen [Balz08, 556]. Bei jedem Durchlauf werden Ziele festgelegt, Alternativen erarbeitet und evaluiert, Risiken identifiziert und eliminiert, worauf anschließend die Entwicklung und Verifikation des Produktes der nächsten Generation

folgt [Fric95, 49]. Der Durchlauf wird durch die Planung der nächsten Phasen abgeschlossen, bevor die nächste iterative Phase durchlaufen wird. In den Teilphasen des Spiralmodells können jeweils verschiedene Methoden eingesetzt werden.
Das Spiralmodell ist ein sehr flexibles Konzept, mit dem sehr gut auf Risiken und Alternativen reagiert werden kann. Die Betrachtung von Alternativen fördert zudem die architektonische und software-technische Wiederverwendung. Nachteilig wirkt sich aus, dass die Steuerung des Entwicklungsprozesses großen Aufwand für das Projektmanagement bedeutet. Aus diesem Grund eignet sich das Spiralmodell kaum für kleine und mittlere Projekte.

3.5.4 Objektorientiertes Vorgehensmodell

Die Aufgabe *objektorientierte Anwendungssystementwicklung* kann in die Teilaufgaben *objektorientierte Analyse* (OOA), *objektorientierter Entwurf* (OOD) und *objektorientierte Realisierung* (OOP) zerlegt werden [Popp94, 12ff]. Setzt man die Teilaufgaben in eine Reihenfolgebeziehung, so ergibt sich das in Abbildung 3.13 dargestellte Phasenmodell [Sinz91].

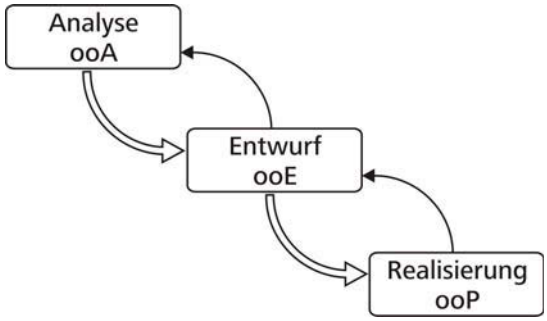

Abbildung 3.13: Phasenmodell der objektorientierten Anwendungssystementwicklung [Sinz91]

Das **objektorientierte Vorgehensmodell** unterstützt neben den Prinzipien der objektorientierten Methodik (vgl. hierzu Kapitel 6.3) einen inkrementellen Entwicklungsprozess (siehe u. a. [Booc94, 312-333], [Balz08, 526-529]), der in Abbildung 3.13 dargestellt ist. Das Konzept der Wiederverwendung kann als wesentlicher Vorteil des objektorientierten Software-Entwicklung angesehen werden.
Diese kann auf verschiedenen *Ebenen* und *Gebieten* sowie mit unterschiedlicher *Herkunft* erfolgen. **Ebenen der Wiederverwendung** sind Ebene der Modelle der objektorientierten Analyse (OOA), die Ebene der technischen Entwürfe (OOD) sowie die Ebene der implementierten Klassen und Klassenbibliotheken (OOP) erfolgen (siehe Abbildung 3.13). Als **Gebiete der Wieder-**

verwendung können der erneute Einsatz anwendungs- bzw. branchenspezifischer Klassen und Subsysteme gesehen werden oder aber die Anbindung einer Anwendung an die Umgebung bzw. System-Software durch Verwendung bereits bestehender Klassen. Bei der **Herkunft der wiederverwendeten Klassen** wird zwischen Eigenentwicklungen und eingekauften Klassenbibliotheken unterschieden [Balz08, 526].

Im Gegensatz zu den bisher vorgestellten Modellen, die allesamt ein *top-down-orientiertes* Vorgehen verfolgten, orientiert sich das objektorientierte Vorgehensmodell in seiner Vorgehensweise *bottom-up*.

Durch den Einsatz eines objektorientierten Frameworks wird bei diesem Vorgehensmodell die Wiederverwendung und damit die wirtschaftliche Entwicklung von Software unterstützt. Da ein Großteil anwendungsneutraler Funktionalität wiederverwendet werden kann, ist ein effizientes Prototyping - wie in Kapitel 3.5.2 vorgestellt - möglich.

SOM-Projektmodell

Ein konkretes objektorientiertes Vorgehensmodell ist das **SOM-Projektmodell**, das auf die in Abbildung 3.14 vorgestellte SOM-Methodik abgestimmt ist. Es umfasst die Phasen *Projektplanung*, *fachliche Modellierung*, *Software-Entwurf & Realisierung* sowie die anschließende Einführung. Objektorientierte Analyse- und Entwurfsmethoden geben kaum Anhaltspunkte für die framework-basierte Anwendungsentwicklung [Maie98]. Die durchgehende Modellierung und Entwicklung auf Basis des objektorientierten SOM-Ansatzes unter Verwendung des Frameworks **moccabox** kann diese Lücke schließen.

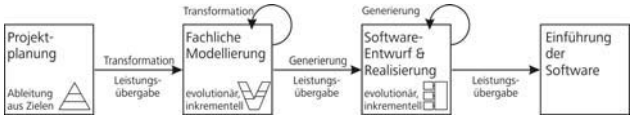

Abbildung 3.14: SOM-Projektmodell [Sinz00, 33]

Projektplanung Die Phase der **Projektplanung** dient der groben Erfassung von Anforderungen an das zu realisierende Anwendungssystem. Dem Software-Systemmodell [Fer+96b, 46f] entsprechend werden Randbedingungen für die Verfahrensumgebung, die Nutzermaschine, das Programmsystem, die Basismaschine sowie die Systemumgebung festgelegt und der Ressourcenbedarf für die Durchführung des Projektes geplant. Die durch

das System zu unterstützenden Geschäftsprozesse werden in dieser Phase identifiziert [Sinz00, 33].

Fachliche Modellierung Die Phase der **fachlichen Modellierung** dient der Anforderungsanalyse, durch die die Nutzermaschine ohne Implementierungsdetails vollständig aus rein fachlicher Sicht beschrieben wird. Ausgehend vom Objekt- und Zielsystem, das in der Phase Projektplanung spezifiziert wurde, werden dem V-Modell des SOM-Ansatzes [FeSi06, 188ff] (siehe auch Kapitel 5.3) folgend die Geschäftsprozesse in Form des Interaktionsschemas (IAS) und des Vorgangs-Ereignis-Schemas (VES) modelliert. Das zugehörige Anwendungssystem wird in Form des konzeptuellen Objektschemas (KOS) und des Vorgangsobjektschemas (VOS) abgeleitet. Eine Unterstützung der Integrationsziele (siehe Kapitel 4) durch entsprechende Komponenten des Frameworks kann bereits in dieser Phase erfolgen.

Software-Entwurf & Realisierung In der Phase **Software-Entwurf** wird das Programmsystem gestaltet[34]. In der anschließenden Phase der **Realisierung** erfolgt die Implementierung[35] der anwendungsspezifischen Klassen auf Basis der SOM-Fachspezifikationen IOS, VOS und KOS. Die Implementierung erfolgt in der vorliegenden Arbeit der in Kapitel 3.3.4 vorgestellten Software-Architektur. Bei der Realisierungsphase handelt es sich um einen inkrementellen, evolutionären Prozess. Dieser ist vor allem im Hinblick auf den Vorgang der Integration von zentraler Bedeutung, da das zu entwickelnde Framework diesen durch entsprechende Komponenten unterstützen soll. Das im Rahmen der Arbeit realisierte Framework **moccabox** wird in den Kapiteln 8 und 9 vorgestellt.

Einführung Die Phase der **Einführung** umfasst die Übernahme des Anwendungssystems in den produktiven Betrieb auf Kundenseite.

3.5.5 Agile Software-Entwicklung

Als Gegengewicht zu traditionellen Ansätzen der Software-Entwicklung mit sehr starrem Vorgehen und einem hohen Maß an manuell erstellten Fließtextdokumenten [StVö05, 13], finden in

[34] Der Vorgang des Entwerfens des Programmsystems wird *Programmieren im Großen* genannt (siehe u. a. [Balz00]).
[35] *Programmierung im Kleinen*

den letzten Jahren vor allem in der Praxis zunehmend **agile Ansätze** der Software-Entwicklung Anhänger. Sie manifestieren folgende Präferenzen[36] [StVö05, 85ff]:

Individuum & Interaktion vor Prozessen & Werkzeugen Ein Team soll seinen eigenen Entwicklungsprozess definieren können. Dabei hat die Interaktion zwischen den Team-Mitgliedern Vorrang vor formalen Prozessen.

Lauffähige Systeme vor verständlicher Dokumentation Nicht die gut aussehende Projektion des Systems in Form von Dokumenten ist höchstes Ziel, sondern die funktionierende Software.

Kundenkontakt vor Vertragsbestimmungen Nicht zu Beginn vertraglich festgelegte Funktionalität, die nach Fertigstellung des Systems abgenommen wird, ist Grundlage der Software-Entwicklung. Vielmehr soll der Kunde direkt in den Prozess der Entwicklung eingebunden werden, um somit ständig auf geänderte Anforderungen reagieren zu können.

Reaktion auf Änderung vor Planverfolgung In Anknüpfung an den vorangegangenen Punkt, müssen Anforderungen des Kunden flexibel übernommen werden, anstatt anfangs formal festgelegte Anforderungen zu verfolgen, die für den Kunden mittlerweile irrelevant sind.

Der wohl bekannteste agile Ansatz ist das **Extreme Programming** (XP) ([Beck00] und [BeFo01]). Es handelt sich dabei um ein leichtgewichtiges Vorgehensmodell für die Software-Entwicklung, das für kleine bis mittelgroße Teams geeignet ist [Rump01, 121]. Die vier Grundkonzepte des XP sind Kommunikation, Einfachheit, Feedback und Mut [Beck00, 29-34]. Zu den Besonderheiten gehört, dass stets zwei Programmierer an einem Rechner sitzen (*pair programming*[37]), auf Dokumentation größtenteils zu Gunsten von Quelltext-Dokumentation sowie Tests verzichtet wird und stets mindestens ein Kunde für die Entwickler verfügbar sein muss - idealerweise im selben Raum (vgl. [Beck00]). Das hier vorgestellte Framework **moccabox** wurde am Centrum für betriebliche Informationssysteme (Ce-bIS) auch für die Anwendungsentwicklung gemäß XP eingesetzt.

Agile Ansätze geben keine direkten Vorschriften und Hilfestellungen für den Prozessablauf der Software-Entwicklung. Ihr Einsatz ist daher allein noch nicht Garant für die erfolgreiche

[36] `http://agilemanifesto.org`

[37] Eine empirische Untersuchung zur Produktivität beim *pair programming* findet sich in [Can+05].

Erstellung qualitativ hochwertiger Software. Außerdem ist fraglich, ob sie auch auf größere Projekte anwendbar sind, die ein Vielfaches komplexer sind und verlässliche Rahmenbedingungen benötigen. Hier bietet sich die Kombination agiler Ansätze in Teilphasen des Vorgehensmodells an. Vor allem die Einbindung des Kunden in die Entwicklung und ein frühzeitig evaluierbares Ergebnis (z. B. in Form eines ersten Prototypen) ist von großer Wichtigkeit. Eine framework-basierte Entwicklungsumgebung und Generierung von Code können das agile Vorgehen auf Seiten der software-technischen Realisierung stark unterstützen [Rump01, 130f] und somit helfen, Entwicklungskosten zu sparen.

4 Integration betrieblicher Anwendungssysteme

Die Systementwicklung ist nicht isoliert zu betrachten, sondern ganzheitlich als Teilaufgabe der Gestaltungsaufgabe *Automatisierung betrieblicher Informationssysteme* (vgl. [Fers92, 4]). Deren Sachziel *Vorgangsautomatisierung* leitet sich aus der Aufgabenebene des betrieblichen Informationssystems ab. Mit diesem korrespondieren die Formalziele Korrektheit, Integration, Echtzeitverhalten sowie Flexibilität [Fers92, 11], die sich auf Merkmale des (zu entwickelnden) Anwendungssystems und damit auf das Aufgabenobjekt der eigentlichen Systementwicklungsaufgabe beziehen. Die Verfolgung des Formalziels Integration[1] im Entwicklungsprozess durch Software-Komponenten zu unterstützen, ist eine wichtige Anforderung an den Framework-Entwurf.

Unter **Integration** kann allgemein sowohl der Vorgang als auch das Ergebnis verstanden werden. Bei der Entwicklung betrieblicher Anwendungssysteme[2] ist die Integration als Ergebnis ein wichtiges Formalziel [Fers92, 3] und damit Qualitätskriterium. Im Rahmen der Aufgabenanalyse werden automatisierbare (Teil-)Aufgaben des Informationssystems abgegrenzt und zur kooperativen Erfüllung auf ein Anwendungssystem[3] abgebildet. Dabei handelt es sich in der Regel um ein verteiltes System, bestehend aus autonomen, zu integrierenden Teilsystemen.

Über die Ausprägung der Integration einer Anwendung, gibt der **Integrationsgrad** Auskunft. Um diesen festzulegen und zu verfolgen, werden **Integrationsziele** formuliert. Diese werden in

[1] FERSTL merkt an, dass eine gleichzeitige Verfolgung des Formalziels Korrektheit Voraussetzung ist [Fers92, 12].

[2] Es werden verschiedene Dimensionen der Integration unterschieden [Rose99]. FERSTL nennt hier die innerbetriebliche und zwischenbetriebliche Integration [Fers92, 5]. Die in der vorliegenden Arbeit behandelten Konzepte beziehen sich vorrangig auf die innerbetriebliche Integration. Für Konzepte und Lösungen zwischenbetrieblicher Integration sei auf die entsprechende Literatur verwiesen (siehe z. B. [Sch+02], [Sch+05]).

[3] Die Abbildung kann auch auf mehrere Anwendungssysteme erfolgen, die dann im Rahmen einer *Enterprise Application Integration (EAI)* wiederum integriert werden müssen. Die Service Oriented Architecture (SOA) ist hierfür eine immer häufiger gewählte Architektur des entsprechenden betrieblichen Informationssystems (siehe auch Kapitel 3.3.5). Diese Form der Integration ist nicht Gegenstand der Arbeit.

Kapitel 4.1 behandelt. Für das Erreichen der Ziele stehen verschiedene **Integrationskonzepte**[4] zur Verfügung, die in Kapitel 4.2 diskutiert werden.
Im zweiten Teil der Arbeit werden in Kapitel 5.6 die daraus entstehenden Anforderungen an die Komponenten des Frameworks **moccabox** abgeleitet, die eine ausreichende Unterstützung des Entwicklers bei der Verfolgung der Integrationsziele gewährleisten sollen. Denn anders als in [Robr05, 142ff] dargestellt, wird hier nicht davon ausgegangen, dass bereits der Einsatz selbst eines Frameworks einen hohen Integrationsgrad garantiert.

4.1 Integrationsziele

Integration ist eines mehrerer Formalziele bei der Automatisierung der Vorgangsdurchführung (siehe Kapitel 3.1), das wiederum in verschiedene Einzelziele (Integrationsziele) unterteilt werden kann. Diese betreffen Struktur- und Verhaltenseigenschaften von Anwendungssystemen [FeSi06, 226]. In Tabelle 4.1 sind die Einzelziele und die Merkmale, auf die sie Bezug nehmen, dargestellt.

Merkmalsgruppe		**Ziel: Einhaltung einer vorgegebenen Ausprägung des Merkmals ...**
Struktur	Redundanz	Datenredundanz
		Funktionsredundanz
	Verknüpfung	Kommunikationsstruktur
Verhalten	Konsistenz	Semantische Integrität
		Operationale Integrität
	Zielorientierung	Vorgangssteuerung
Aufgabenträgerunabhängigkeit		Unabhängigkeit vom Aufgabenträger

Tabelle 4.1: Merkmale integrierter Anwendungssysteme [FeSi06, 227]

Die Einzelziele können folgendermaßen charakterisiert werden (siehe [Fers92, 13f] und [FeSi06, 226ff]):

Redundanz Das Merkmal Redundanz spiegelt wider, inwiefern Systemfunktionen doppelt oder mehrfach vorhandenen Systemkomponenten zugeordnet sind und inwiefern Ausprägungen der Systemkomponenten entfernt werden könnten, ohne dass die Funktionsfähigkeit

[4] Eine ausführliche Betrachtung verschiedener Integrationskonzepte anhand eines umfangreichen Kriterienkataloges findet sich bei [MeHo92].

der Systemfunktion eine Beeinträchtigung erfahren würde. Je nachdem, ob es sich um redundante Datenobjekttypen bzw. Datenattribute oder um redundante Lösungsverfahren in Form von gleichen Aktionen handelt, wird von **Datenredundanz** respektive **Funktionsredundanz**. Nicht immer ist die Vermeidung von Redundanz oberstes Ziel, sondern vielmehr ihre Optimierung.

Verknüpfung Das Merkmal Verknüpfung bestimmt Art und Anzahl der Kommunikationskanäle zwischen den Systemkomponenten. In objektorientierten Anwendungssystemen existieren diese Kanäle zwischen Objekten. Ein eigenständiges Kommunikationssystem, das die Kommunikationskanäle kontrolliert, ist Gegenstand des Integrationsziels *Verknüpfung*. Aufgabe dieses Kommunikationssystems ist die Übertragung der Nachrichten zwischen den einzelnen Systemkomponenten und die Versorgung der Komponenten mit Informationen über die zu Grunde liegende Kommunikationsstruktur. Damit die Auswirkungen von Änderungen der Komponenten auf das Gesamtsystem kontrollierbar bleiben, sind die Kommunikationsbeziehungen zu minimieren.

Konsistenz Das Merkmal der Konsistenz bzw. Integrität beschreibt die zulässigen korrekten Zustände des Systems bezogen auf den modellierten Ausschnitt der realen Welt. Es werden **semantische Integrität** und **operationale Integrität** unterschieden. Die semantische Integrität beschreibt die gültigen Zustände und Zustandsübergänge der Aufgabenobjekte eines Anwendungssystems. Erlaubt ein System parallele Zustandsübergänge (Transaktionen) stellt die operationale Integrität Bedingungen für konsistente Systemzustände vor und nach Zustandsübergängen auf. Zusätzlich kann noch die Vermeidung von Änderungen durch nicht autorisierte Personen (Datenschutz) zum Ziel der Integrität gezählt werden [ScSt83, 275]. Bei semantischen Integritätsbedingungen kann zwischen statischen und dynamischen Integritätsbedingungen unterschieden werden. Je nachdem, ob sich die Bedingung auf genau einen Zustand des Systems oder mehrere bezieht [KeEi99, 133].

Zielorientierung Die Aufgabenzerlegung eines Anwendungssystems erfolgt nach dem *top-down-Ansatz*. Folglich liegt dem Anwendungssystem eine Gesamtaufgabe zu Grunde, bei deren sukzessiver Zerlegung die Aufgabenobjekte und Aufgabenziele zerlegt werden. Zur Koordination der erzeugten Teilaufgaben wird eine Vorgangssteuerung benötigt, die

eine Zielausrichtung aller Komponenten und Teilsysteme im Hinblick auf übergeordnete Ziele vornimmt.

Aufgabenträgerunabhängigkeit Aufgabenträger eines Anwendungssystems ist ein Rechnerverbund. Dieser kann Komponenten verschiedenen Typs und verschiedener Hersteller enthalten. Ziel der Unabhängigkeit vom Aufgabenträger ist die Portierbarkeit des Anwendungssystems zwischen verschiedenen Rechnersystemen und -generationen[5].

4.2 Integrationskonzepte

Es existieren verschiedene **Integrationskonzepte**, die zur Erreichung der vorgenannten Integrationsziele bei der Entwicklung betrieblicher Anwendungssysteme eingesetzt werden können. Sie stellen damit Lösungsverfahren zur Erreichung des Formalziels *Integration* dar. Der Zielerreichungsgrad ist bei den jeweiligen Konzepten unterschiedlich ausgeprägt. Während sich die Funktionsintegration hauptsächlich an der Integration von Aufgabenzielen bzw. Lösungsverfahren und deren Verknüpfung orientiert, stellt die Datenintegration die Integration der Aufgabenobjekte in den Mittelpunkt. Die Objektintegration bezieht sich auf beide vorgenannten Integrationsbereiche. Im Folgenden werden die genannten Konzepte sowie deren Grad der Zielerreichung einzeln vorgestellt[6].

4.2.1 Funktionsintegration

Es werden die aufgabenträgerorientierte und die datenflussorientierte Funktionsintegration (vgl. [FeSi06, 230-232]) unterschieden. Die **aufgabenträgerorientierte Funktionsintegration** ist auf teilautomatisierte Aufgaben ausgerichtet, bei denen der personelle Anteil einer Aufgabe von einer Person durchgeführt wird, die mit dem Anwendungssystem über ein Mensch-Computer-Interface (MCI) kommuniziert. Bei diesem Integrationskonzept können keine Aussagen über die Erreichung der Integrationsziele gemacht werden. Im Gegensatz dazu wird bei der **datenflussorientierten Funktionsintegration** zumindest das Integrationsziel *Kommunikationsstruktur* verfolgt, da das Anwendungssystem als Netz flussorientierter Aufgaben betrachtet wird. In der frühesten und einfachsten Form wurde die Kommunikation durch transportierba-

[5] Siehe hierzu auch das Qualitätsmerkmal *Übertragbarkeit* in Kapitel 3.4.
[6] Eine Repräsentation der Integrationskonzepte als Entwurfsmuster findet sich in [HoWo03].

re Datenträger erreicht, später wurden diese größtenteils durch Kommunikationsnetze ersetzt. Durch die multiple Speicherung der Anwendungsdaten in den jeweiligen Anwendungssystemen wird das Ziel der Vermeidung von Redundanz nicht nur nicht erreicht, sondern diesem sogar entgegengewirkt.

4.2.2 Datenintegration

Die **Datenintegration** (vgl. [FeSi06, 233-235]) stellt die Verfolgung der Ziele *Datenredundanz* und *Integrität* in den Vordergrund, die beim Konzept der datenflussorientierten Funktionsintegration vernachlässigt wurden. Der Datenintegration liegt das Paradigma des Zustandsraummodells zu Grunde. In einer erweiterten Fassung wird die Datenstruktur des Zustandsraumes vom konzeptuellen Datenschema eines Datenbanksystems repräsentiert, auf der eine Menge von Funktionen operieren. Jede Funktion nimmt eine so genannte externe Sicht (View) auf eine Teilmenge des konzeptuellen Datenschemas ein, wie in Abbildung 4.1 dargestellt[7]. Die Spezifikation eines datenintegrierten Anwendungssystems umfasst die Spezifikation des konzeptuellen Datenschemas und die Spezifikation entsprechender Funktionen einschließlich deren externer Sichten.

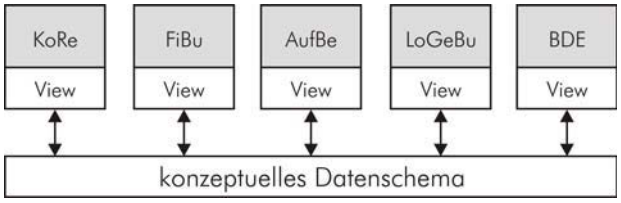

Abbildung 4.1: Beispiel eines datenintegrierten Anwendungssystems [FeSi06, 234]

Die Integration der Aufgaben wird über die Aufgabenobjekte realisiert. Dort, wo sich Views auf das konzeptuelle Datenschema überlappen enstehen zwischen den Aufgaben Kommunikationskanäle, über die Informationen ausgetauscht werden können.
Die Datenredundanz wird mit Methoden der Datenmodellierung kontrolliert. Dabei wird semantische Integrität durch die Formulierung semantischer Integritätsbedingungen überwacht. Das eingesetzte Datenbankverwaltungssystem (DBVS) kontrolliert die operationale Integrität durch seine Transaktionsverwaltung. Weitere Merkmale wie die *Kommunikationsstruktur* und die *Vorgangssteuerung* werden nicht verfolgt.

[7] Siehe hierzu das ANSI/SPARC Schichtenmodell [ANSI75].

4.2.3 Objektintegration

Das Konzept der **Objektintegration** wird im Folgenden anhand der Fachspezifikationen der SOM-Methodik erläutert (vgl. [FeSi06, 235-237]). Das Anwendungssystem wird in Form eines konzeptuellen Objektschemas (KOS) und eines Vorgangsobjektschemas (VOS) spezifiziert. Konzeptuelle und Vorgangsobjekte stellen Aufgabenobjekte bzw. Lösungsverfahren des Anwendungssystems dar und sind jeweils untereinander lose gekoppelt. Entsprechend sind die Vorgangsobjekte mit den konzeptuellen Objekten gekoppelt. Jeder Vorgangsobjekttyp entspricht einer Teilaufgabe, die aus der Zerlegung der Gesamtaufgabe hervorgegangen ist. Die Beziehungen der Vorgangsobjekttypen untereinander entsprechen der Zerlegungsstruktur der Gesamtaufgabe, wodurch das Merkmal der *Vorgangssteuerung* eine globale Zielerreichung unterstützt. Die Vorgangssteuerung erfolgt derart, dass die konzeptuellen Objekte die Aktionen eines Vorgangs realisieren, während die Aktionensteuerung vom Vorgangsobjekt übernommen wird. Die Objekte kommunizieren wie in Abbildung 4.2 dargestellt per Nachrichtenaustausch über ein globales Kommunikationssystem. Zur Beschreibung des Austausches von Diensten zwischen lose gekoppelten, autonomen Komponenten dient das Client/Server-Modell[8] [Raue96, 45]. Der Client fordert durch eine Nachricht beim Server einen Dienst an. Für die Komposition lose gekoppelter Komponenten ist wichtig, dass alle Server, die den gleichen Dienst anbieten (z. B. Durchführung eines Vorgangs), über eine standardisierte Schnittstelle verfügen. Zudem muss die Zusammensetzung der Komponenten so spezifiziert werden, dass sie zu einem sinnvollen, komponentenübergreifenden Verhalten führt (vgl. [BiRi89]).

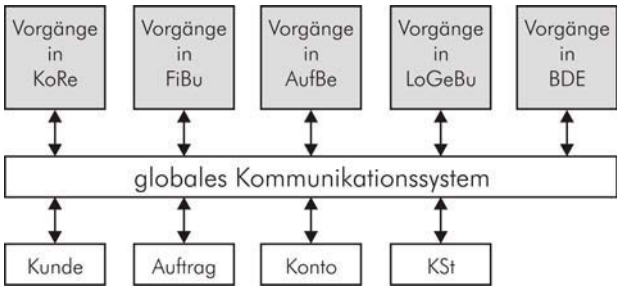

Abbildung 4.2: Beispiel eines objektintegrierten Anwendungssystems [FeSi06, 236]

Die Merkmale *Datenredundanz* sowie *Integrität* werden durch die Modellierung des KOS verfolgt, das in der Datensicht weitgehend einem konzeptuellen Datenschema entspricht. Die se-

[8] Zum Client-Server-Modell siehe u.a. [Orf+97].

mantische Integrität wird beim Konzept der Objektintegration durch die nachrichtenbasierte Kommunikation sichergestellt. Ein konzeptuelles Objekt kann bei einer Zustandsänderung nur überwachen, ob die lokale Integrität gewahrt ist. Erstreckt sich aber eine Integritätsbedingung über zwei konzeptuelle Objekte, wird dieses weitere benachrichtigt. Verletzt die Änderung des ersten die globale Integrität, so führt das zweite konzeptuelle Objekt eine Zustandsänderung aus, um diese wiederherzustellen. Alternativ wird die Zustandsänderung des ersten konzeptuellen Objektes wieder rückgängig gemacht.

Das Merkmal der *Funktionsredundanz* wird durch Generalisierung allgemeiner Lösungsverfahren kontrolliert, die durch Vererbung auf spezielle Lösungsverfahren genutzt werden. Die *Kommunikationsstruktur* im Anwendungssystem wird durch ein eigenes Kommunikationssystem kontrolliert, das für die Kommunikation der konzeptuellen Objekte und Vorgangsobjekte untereinander sowie der Vorgangsobjekte mit den konzeptuellen Objekten verwendet wird. Durch die Verwendung des Kommunikationssystems sind die Beziehungen der Objekte zueinander transparent und können überwacht werden. Bezüglich der geforderten *Unabhängigkeit vom Aufgabenträger* lassen sich standardisierte Plattformen für die Entwicklung eines objektorientierten und objektintegrierten Anwendungssystems verwenden, die diese ermöglichen.

Damit konnte gezeigt werden, dass das Konzept der Objektintegration die Unterstützung aller geforderten Integrationsziele ermöglicht. Ein objektorientiertes Software-Framework muss demzufolge Komponenten bieten, die die durchgängige Verfolgung der Objektintegration bei der Entwicklung betrieblicher Anwendungssysteme unterstützen. Die entsprechenden Unterstützungspotenziale durch ein geeignetes Framework werden in Kapitel 5.6 beschrieben.

Teil II

Geschäftsprozessmodellgetriebene Anwendungsentwicklung

5 Geschäftsprozesse als Treiber der Anwendungsentwicklung

Der software-technischen Realisierung eines Anwendungssystems geht dessen fachliche Ableitung im Rahmen der Anforderungsanalyse und -definition voraus. Ergebnis dieser Phase ist ein fachliches Modell, das die zu realisierende Nutzermaschine spezifiziert. Der Komplexitätsabstand der Komponenten der gewählten Basismaschine bestimmt sich aus der Nähe zu den Fachkonzepten des verwendeten fachlichen Modells.

Die Abgrenzung des Anwendungssystems erfolgt auf Aufgabenebene des Informationssystems. Die fachliche Abgrenzung und die anschließende technische Realisierung erfolgen in der Praxis häufig nach funktionsorientierten Kriterien, also der Einteilung der Organisation in Funktionsbereiche (z. B. Beschaffungs- oder Produktionsbereich). Gleichzeitig nahm und nimmt der Druck auf Unternehmen zu, ihre Geschäftsprozesse anzupassen [FeSi06, 210], unabhängig von Funktionsbereichen. In Theorie und Praxis der organisatorischen Gestaltung von Unternehmen hat sich der Einsatz des Paradigma der Geschäftsprozessorientierung weitgehend durchgesetzt (vgl. [MaSc97, 4]). Das Informationssystem der Organisation besteht aus den wechselseitig voneinander abhängigen Teilsystemen Aufgabenstruktur, Aufbauorganisation und Anwendungssysteme. Für eine abgestimmte, evolutionäre Weiterentwicklung der drei Teilsysteme sind Strukturähnlichkeiten und -analogien zwischen ihnen Voraussetzung [Sinz99, 20]. Für die Systementwicklung bedeutet das, die Struktur von Anwendungssystemen korrespondierend zur Aufgabenstruktur der Organisation auszurichten. In der Arbeit wird von der Ausrichtung der Struktur an den Geschäftsprozessen der Organisation ausgegangen.

Dieses Kapitel führt in Kapitel 5.1 in das der Arbeit zu Grunde liegende Verständnis von Geschäftsprozessen und in Kapitel 5.2 allgemein in deren Modellierung ein. In Kapitel 5.3 werden die Fachkonzepte des Semantischen Objektmodells (SOM) vorgestellt. SOM ist eine spezielle geschäftsprozess- und objektorientierte Methodik für die Modellierung betrieblicher

Informationssysteme und Ableitung der Fachspezifikation korrespondierender Anwendungssysteme. Modelle sind bei der software-technischen Realisierung von Anwendungssystemen immer häufiger eine treibende Kraft. In Kapitel 5.4 wird mit der architekturzentrierten modellgetriebenen Software-Entwicklung (MDSD) ein Ansatz vorgestellt, durch den Modell und Systemplattform über (automatisierte) Transformationen miteinander verbunden werden können. Erkenntnisse daraus werden auf die geschäftsprozessbasierten SOM-Modelle übertragen. Bei der Abbildung der Nutzermaschine durch ein Programm sind bei der Entwicklung Integrationsziele (siehe Kapitel 4.1) zu beachten. Kapitel 5.6 leitet daraus Anforderungen an Komponenten einer geeigneten Basismaschine ab.

5.1 Geschäftsprozesse

Der Begriff des Geschäftsprozesses wird in der Literatur nicht einheitlich verwendet[1]. Die vorliegende Arbeit bezieht sich auf die in [FeSi95a, 214] gegebene Definition, der die Erstellung betrieblicher Leistung durch Geschäftsprozesse zu Grunde liegt. Danach erstellt ein **Geschäftsprozess** (engl. *business process*) eine oder mehrere betriebliche Leistungen[2], die er an die ihn beauftragenden Geschäftsprozesse übergibt. Er kann seinerseits weitere Geschäftsprozesse mit der Lieferung von Leistung beauftragen. Ein Geschäftsprozess setzt sich zusammen aus zueinander in Beziehung stehenden Aufgaben [HaCh94, 35f]. Die Koordination zwischen den Geschäftsprozessen erfolgt dabei nach dem **Client/Server-Prinzip**. Die Lieferung einer bestimmten Leistung wird durch einen Client-Geschäftsprozess beauftragt und durch einen korrespondierenden Server-Geschäftsprozess erbracht.

Abbildung 5.1: Geschäftsprozess [FeSi95a, 214]

Ein Geschäftsprozess umfasst also (a) die Erstellung und Lieferung von Leistung, (b) die Koordination der betrieblichen Objekte, die an der Erstellung und Lieferung der Leistung beteiligt sind sowie (c) eine Reihe von Vorgängen, die bei seiner Durchführung ausgeführt werden

[1] Einen Überblick geben u.a. [BeSc] und [Tinn95].
[2] Leistung schließt hier auch Produkte, Güter und Zahlungen mit ein [FeSi96a, 9].

[FeSi93, 9f].

5.2 Geschäftsprozessmodellierung

Die zweckorientierte Abbildung eines abgegrenzten Ausschnitts eines betrieblichen Systems (Diskurswelt)[3] und seiner relevanten Umgebung in ein formales oder semi-formales Modellsystem wird als **Modellierung betrieblicher Systeme**[4] bezeichnet [Fer+96b, 26]. Die Modellbildung in zwei Stufen ist in Abbildung 5.2 dargestellt. Der Komplexitätsabstand zwischen Diskurswelt und fachlichem Modell des Anwendungssystems wird entscheidend verringert durch das zweistufige Vorgehen. Für die Modellierung der Diskurswelt und anschließende Modellierung der fachlichen Anforderungen an das Anwendungssystem wird in der vorliegenden Arbeit ein geschäftsprozessorientiertes Vorgehen gewählt.

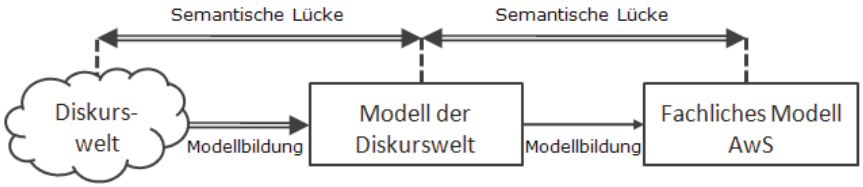

Abbildung 5.2: Zweistufiges Vorgehen bei der Anforderungsspezifikation des Anwendungssystems [Popp94, 4]

Das so abgeleitete fachliche Modell wird durch eine Nutzermaschine spezifiziert, die im Zuge der Systementwicklung auf eine geeignete Basismaschine abzubilden ist. Dieser Zusammenhang ist in Abbildung 5.3 dargestellt.

Während in den 1970er und 1980er Jahren bei der Modellierung betrieblicher Anwendungssysteme noch die Datenmodellierung (siehe [Chen76]) sowie die Funktionsmodellierung (siehe [DeMa79], [YoCo79]) vorherrschten, wurden seit Beginn der 1990er Jahre vermehrt objektorientierte Ansätze (siehe [CoYo91], [ShMe92], [Jac+92], [Rum+91], [Booc94]) sowohl für die fachliche als auch für die software-technische Modellierung eingesetzt[5]. RUMBAUGH sieht einen der größten Nutzen der objektorientierten Modellierung in der modellgestützten Abbil-

[3] Die Modellierung eines ganzen Unternehmens wird auch als *enterprise modeling* bezeichnet (vgl. u. a. [Rumb93]).

[4] Zum Thema *Modellbildung* siehe [Hamm99] und auch [Schm01].

[5] Eine Diskussion der verschiedenen Ansätze findet sich u. a. in [Jaco93], [Popp94], [Sinz96], [Stei97].

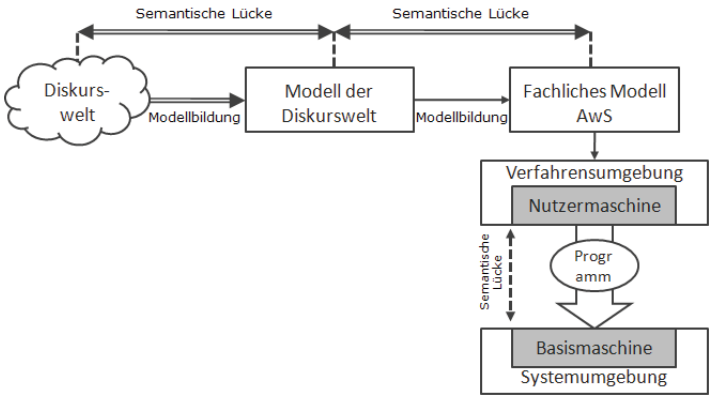

Abbildung 5.3: Zweistufiges Vorgehen bei der Anforderungsspezifikation des Anwendungssystems und anschließende software-technische Abbildung (nach [Popp94, 4])

dung der gesamten Unternehmung[6] (*enterprise modeling*), durch die eine komplexe soziale Organisation (Unternehmung) mittels Modellbildung verstanden werden kann [Rumb93, 18]. Aus einer Kombination verschiedener objektorientierter Ansätze (OMT, OOSE, Booch) ging schließlich die *Unified Modeling Language* (UML) hervor, die gegenwärtig in der Version 2.0 vorliegt [UML04] und eine Notationssprache für objektorientierte Software-Systeme darstellt. Im Allgemeinen sind diese Ansätze auf die Modellierung software-technischer Anforderungen und Konzepte des betrieblichen Anwendungssystems beschränkt.

Geschäftsprozessorientierte Modellierungsansätze stellen den Übergang von einer statischen und strukturorientierten hin zu einer dynamischen und verhaltensorientierten Betrachtung der Unternehmung dar [FeSi93]. Die Unternehmung ist ein System aus miteinander verbundenen Geschäftsprozessen [PiFr95, 14]. Die **Geschäftsprozessmodellierung** bildet die realen Geschäftsprozesse der Diskurswelt auf ein entsprechendes Modellsystem ab[7]. Durch die anschließende Abbildung des Geschäftsprozessmodells auf die fachliche Spezifikation des Anwendungssystems werden die Geschäftsprozesse zum Bindeglied zwischen Unternehmensstrategie und Systementwicklung [Gada01, 30]. Eine essentielle Anforderung für spätere Anpassungen an das geänderte Geschäftsumfeld, wie eingangs bereits erwähnt. Prominente Vertreter der geschäftsprozessorientierten Modellierung[8] sind **ARIS** [Sche98] und **SOM** [FeSi06]. In den

[6] Das Unterfangen einer software-technischen „Gesamtkarte“ des Unternehmens wird in der Literatur sehr skeptisch betrachtet (vgl. u. a. [CoDa94]).

[7] Die Abgrenzung und Spezifikation von Geschäftsprozessen selbst ist eine komplexe und viel disktutierte Aufgabe. Für eine Diskussion wird u. a. auf [MaSc97] verwiesen.

[8] Ein Vergleich verschiedener geschäftsprozessorientierter Modellierungsansätze findet sich u.a. bei [Raue96], [Sinz96], [FeSi06].

letzten Jahren sind auch Modelle basierend auf UML weiterentwickelt worden[9]. Aus der objektorientierten Software-Entwicklung kommend, sollen diese eine Brücke schlagen zwischen der software-technischen und der betrieblichen Sicht auf ein Anwendungssystem [LoWe05, 300]. Eine solche Methodik wird in [Oes+03] und [Oest05] beschrieben. Es handelt sich dabei aber um ein *bottom-up*-Vorgehen, das software-technische Anforderungen und Konzepte zu Grunde legt. Das UML-basierte Vorgehen bietet zwar den Vorteil, dass die UML formal fundiert ist und eine einheitliche Beschreibungssprache durchgehend vom Geschäftsprozessmodell zum System-Design-Modell genutzt wird [LoWe05, 300]. Doch auch wenn die in [Oes+03] vorgestellte objektorientierte Geschäftsprozessmodellierung einige Ähnlichkeiten zum Vorgehen spezifischer geschäftsprozessorientierter Modellierungsmethoden aufweist (siehe [Oest05, 28]), fehlen der UML die Konzepte zur direkten Ableitung der Anforderungen aus Struktur und Verhalten des betrieblichen Systems, wie sie die im Folgenden vorgestellte SOM-Methodik aufweist.

5.3 Die SOM-Methodik

Die **SOM-Methodik**[10] (Semantisches Objektmodell) ist ein umfassender geschäftsprozess- und objektorientierter Ansatz zur Modellierung betrieblicher Systeme und zur fachlichen Spezifikation von Anwendungssystemen. Seine Fachkonzepte dienen als Ausgangspunkt für den Entwurf der Architektur des zu entwickelnden Frameworks. Ein betriebliches System wird in SOM als offenes, zielgerichtetes und sozio-technisches System betrachtet [FeSi97, 3] (vgl. auch [Stae90, 581]). SOM weist durch seine ganzheitliche Betrachtung eine besondere Durchgängigkeit auf [Ohle98, 89] und stellt den Zusammenhang zwischen Unternehmenszielen, Geschäftsprozessen und Anwendungssystem her [Stei97, 170]. Zudem ist die Methodik durchgehend objektorientiert und bietet daher sehr gute Voraussetzungen für die Ableitung eines objektorientierten

[9] Eine formale Spezifikation von Geschäftsprozessen mittels Diagrammen mit Fokus auf den Bereich der Produktion findet sich in [Krei04].

[10] Für eine ausführliche Betrachtung der SOM-Methodik sei hier auf die entsprechende Literatur von FERSTL & SINZ verwiesen, z. B. [FeSi94b], [FeSi94a], [FeSi95b], [FeSi96b], [FeSi06]. Eine Vergleichende Einordnung zu anderen Modellierungsansätzen findet sich u. a. in [Ohle98, 66-89] und [Crüs98, 10-36].

Software-Systems[11]. Sie stellt die fachliche Grundlage der Arbeit dar und wird im folgenden vorgestellt[12].

Unternehmensarchitektur

Das Architekturmodell des SOM-Ansatzes ist die in Abbildung 5.4 dargestellte **SOM-Unternehmensarchitektur**, deren fünf Modellebenen mit dem Ziel der Entwicklung betrieblicher Anwendungssysteme durchlaufen werden. Die Phasen des Lösungsverfahrens der Systementwicklungsaufgabe korrespondieren mit den Modellebenen: Phase Anforderungsanalyse und -definition für Ebene 1 bis 3, Phase Software-Design für Ebene 4 und Phase Realisierung für Ebene 5.

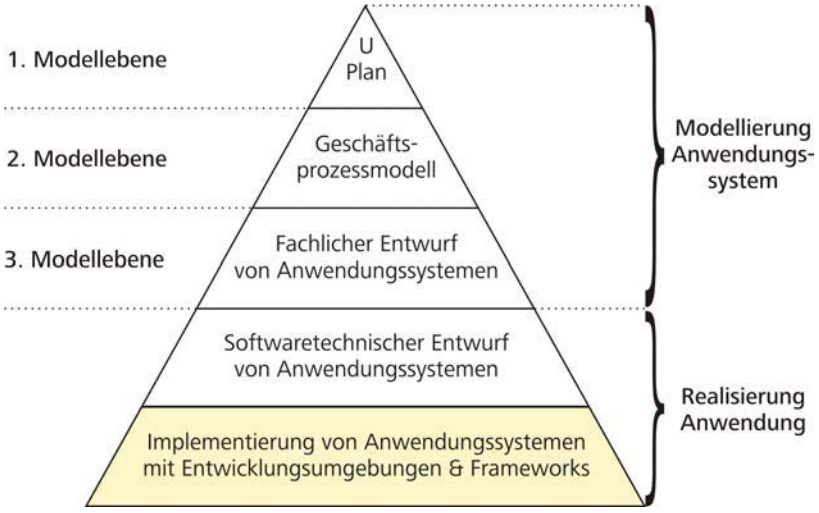

Abbildung 5.4: Unternehmensarchitektur der SOM-Methodik (nach [Mali97, 6])

1. **Unternehmensplan** (Modellebene 1)

   Der Unternehmensplan ist ein Modell der Außensicht eines betrieblichen Systems und das Ergebnis der strategischen Unternehmensplanung einer Organisation. Es liegt die Metapher einer Sicht auf die globale Unternehmensaufgabe zu Grunde. Der Unternehmensplan

[11] Im Gegensatz zu ARIS [Sche98], bei dem es sich nicht um einen durchgehend objektorientierten Ansatz handelt. ARIS kann lediglich als kombinierter Ansatz bezeichnet werden, da Daten und Operationen explizit getrennt betrachtet werden [Crüs98, 15ff]. Er stützt sich dabei auf die Einteilung objektorientierter Analyse- und Entwurfsansätze in [MoPu92].

[12] Eine Erweiterung der SOM-Methodik um eine UML-basierte Notation, die über ein Meta-Modell die Konzepte beider Ansätze verknüpft, ist Gegenstand der Forschung, aber nicht Teil der vorliegenden Arbeit. Ziele und Architektur einer werkzeug-basierten Entwicklung mit SOM werden in [Fer+94] beschrieben. Die Beschreibung eines CASE-Tools findet sich bei [Mali97].

legt die Sach- und Formalziele der Unternehmung, Strategien zu deren Umsetzung sowie die Wertschöpfungsketten der Diskurswelt fest [FeSi06, 186]. Erstellung und Modellierung des Unternehmensplans werden dieser Arbeit nicht näher betrachtet (vgl. hierzu beispielsweise [StSc00]).

2. **Geschäftsprozessmodelle** (Modellebene 2)

   Das Geschäftsprozessmodell spezifiziert die Innensicht des betrieblichen Systems und beschreibt die Lösungsverfahren für die Realisierung des Unternehmensplans [FeSi06, 186]. Geschäftsprozesse, die ihre Leistung direkt an die Umwelt abgeben werden als Hauptprozesse bezeichnet. Sie tragen unmittelbar zur Erfüllung des Sachziels des Unternehmens bei. Als Service-Prozess werden Geschäftsprozesse bezeichnet, die ihre Leistung gegenüber Hauptprozessen und weiteren Service-Prozessen erbringen. Die Geschäftsprozesse sind miteinander durch Leistungsbeziehungen nach dem Client/Server-Prinzip verbunden. Durch die SOM-Methodik wird die Modellierung von Struktur- und Verhaltensmerkmalen der Geschäftsprozesse sowie eine mehrstufige Verfeinerung der Modelle unterstützt [Mali97, 6].

3. **Fachlicher Entwurf von Anwendungssystemen** (Modellebene 3)

   Die zur Durchführung der auf Modellebene 2 modellierten Geschäftsprozesse benötigten Ressourcen werden auf der dritten Modellebene spezifiziert. Neben Anwendungssystemen handelt es sich dabei um Personal sowie Maschinen und Anlagen[13] [FeSi06, 187]. Die fachliche Spezifikation betrieblicher Anwendungssysteme steht im Vordergrund der Betrachtung[14]. Die Anwendungssysteme werden als objektorientierte und objektintegrierte verteilte Systeme spezifiziert (siehe unten) und dienen (in Kooperation mit personellen Aufgabenträgern) der Durchführung (teil-) automatisierter informationsverarbeitender Aufgaben betrieblicher Objekte[15] [FeSi06, 187].

4. **Software-technische Spezifikation von Anwendungssystemen**

   Ausgehend vom fachlichen Entwurf wird die software-technische Spezifikation des Anwendungssystems unter Beachtung eines Software-Architekturmodells abgeleitet. Auf

[13] Die Aufgabenträger Personal sowie Maschinen und Anlagen werden im Rahmen der Arbeit nicht betrachtet.

[14] Die rechnergestützte Ableitung der Rohstruktur der fachlichen Spezifikation von Anwendungssystemen aus dem Geschäftsprozessmodell wird in [Mali97] beschrieben. Siehe auch Kapitel 5.4.

[15] Methodische Grundlage betrieblicher Objekte ist das Konzept der betrieblichen Aufgabe [FeSi06, 192].

dem SOM-Ansatz baut das objektorientierte Software-Architekturmodell (siehe auch Kapitel 3.3.4) auf [Ambe93]. Das Ergebnis dieser Phase ist eine anwendungsspezifische bzw. fachliche Software-Komponente, die ihrerseits wiederum anwendungsneutrale bzw. technische Software-Komponenten nutzt [Mali97, 7]. Diese werden auf der nächsten Ebene ebenfalls modelliert.

5. **Realisierung von Anwendungssystemen mit Software-Entwicklungsumgebungen**
   Die für die Realisierung der fachlichen und technischen Software-Komponenten eingesetzten Basismaschinen (z.B. Software-Frameworks) werden auf dieser Ebene modelliert. Die Basismaschinen-Modelle werden mit dem software-technischen Modell des Anwendungssystems kombiniert [Mali97, 7]. Das hier zu entwickelnde Framework stellt eine solche Basismaschine dar und wird im dritten Teil der Arbeit vorgestellt.

Für Geschäftsprozessmodelle und Anwendungssystem-Spezifikationen sind in SOM **Metamodelle** sowie zugehörige **Beziehungs-Metamodelle** verfügbar[16] [Sinz97, 11]. Metamodelle definieren die Konstruktionsregeln einer Modellebene, Beziehungs-Metamodelle die paarweise Beziehung zwischen Modellebenen [Sinz97, 4] und ermöglichen dadurch die Integration der jeweiligen Modelle[17]. Anhand des zugehörigen Metamodells kann die Konsistenz eines Modells überprüft werden. Zudem verwendet SOM **Strukturmuster** [Fer+96b, 11] (siehe auch [Gam+95]). Im Bereich der Geschäftsprozessmodelle stehen elementare[18] und domänenspezifische Strukturmuster[19] sowie hybride Lenkungs-Strukturmuster zur Verfügung [Sinz97, 11]. Bei der Spezifikation des Anwendungssystems wird auf Strukturmuster des objektorientierten Software-Entwurfs zurückgegriffen (siehe Kapitel 6.6).

Vorgehensmodell

Die Modellbildung in SOM erfolgt anhand des korrespondierenden Vorgehensmodells **V-Modell**, das in Abbildung 5.5 dargestellt ist.

Der Ablauf der Modellierung[20] erfolgt entlang der drei Ebenen des Vorgehensmodells, die mit

[16] Zu den Metamodellen und Ableitungsregeln vgl. u. a. [FeSi06].

[17] Es können zwischen Modell verfeinernde Beziehungen, ergänzende Dokumentation und Konstruktionsbeziehungen unterschieden werden [Hof+96, 7].

[18] Bspw. für das Regelungs- und Verhandlungsprinzip.

[19] Referenzmodelle wie in [Fer+98] und [Rüff99] beschrieben.

[20] Ein durchgängiges Beispiel zur SOM-Methodik findet sich in [FeSi06, 190-221].

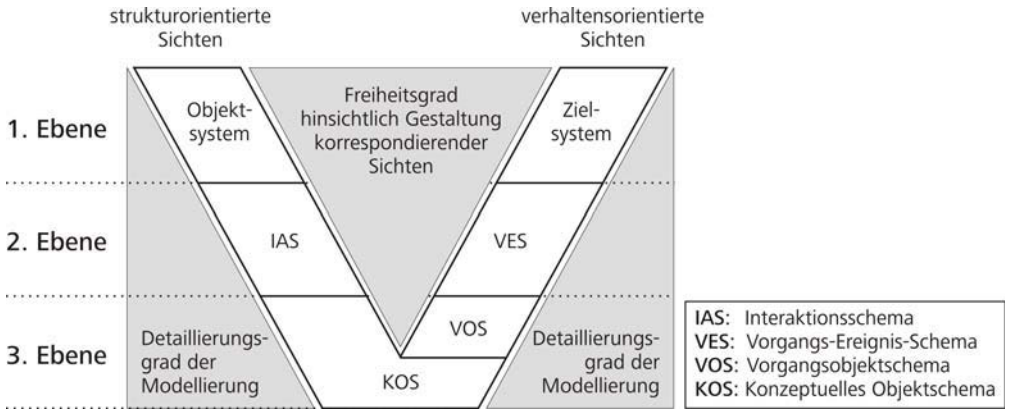

Abbildung 5.5: Vorgehensmodell der SOM-Methodik ([FeSi06, 188])

den Modellebenen der Unternehmensarchitektur korrespondieren (siehe Abbildung 5.4). Auf jeder der Ebenen wird eine struktur- und eine verhaltensorientierte Sicht[21] modelliert. Die Ergebnisse der Modellierung sind mit dem Ergebnis der korrespondierenden Sicht derselben Ebene sowie dem derselben Sicht benachbarter Ebenen abzustimmen [FeSi06, 188].

1. Auf der ersten Modellebene der Unternehmensarchitektur wird das betriebliche System durch Angabe eines Unternehmensplans aus Außensicht beschrieben. Im Vorgehensmodell spezifiziert ein entsprechendes **Objektsystem**, das aus strukturorientierter Sicht Diskurswelt und Umwelt sowie ihre zugehörigen Leistungsbeziehungen enthält. Das Verhalten wird auf dieser Ebene durch das **Zielsystem** beschrieben, das Sach- und Formalziele definiert sowie die Strategien und Rahmenbedingungen für deren Umsetzung. Die Modellierung auf dieser Ebene erfolgt informal und ist vom Verständnis des Modellierers abhängig [Mali97, 16].

2. Die zweite Modellebene der Unternehmensarchitektur beschreibt mit dem Geschäftsprozessmodell die Innensicht des betrieblichen Systems. Im Vorgehensmodell wird auf der zweiten Ebene die Struktur des Geschäftsprozessmodells durch das **Interaktionsschema (IAS)** repräsentiert, dessen Verhalten durch das **Vorgangs-Ereignis-Schema (VES)** (ausführlich in [FeSi06]). Aus dem informal beschriebenen Modell der ersten Ebene wird auf dieser Ebene eine semi-formale Darstellung abgeleitet [Mali97, 17], die dem SOM-Metamodell für Geschäftsprozessmodelle genügt (vgl. [Sinz97] und [FeSi06, 203]).

[21] Zu Struktur- und Verhaltenssicht auf ein Modellsystem siehe auch [Hof+96, 7f].

3. Die Ressourcen zur Durchführung der Geschäftsprozesse werden auf der dritten Modellebene beschrieben. Dazu zählen Personal, Maschinen und Anlagen[22] sowie Anwendungssysteme. Zur Automatisierung von Ausschnitten des Geschäftsprozessmodells werden Anwendungssysteme auf dritter Ebene des Vorgehensmodells aus fachlicher Sicht anhand der Entwurfsmetaphern konzeptuelle Objekte, Vorgangsobjekte und Interface Objekte spezifiziert[23]. Die strukturorientierten Komponenten werden durch ein **konzeptuelles Objektschema (KOS)** repräsentiert, welches **konzeptuelle Objekttypen (KOT)** enthält. Das Verhalten und Zusammenwirken der KOT wird durch **Vorgangsobjekttypen (VOT)** beschrieben, die in einem so genannten **Vorgangsobjektschema (VOS)** zusammengefasst sind (siehe Kapitel 5.3.2). Die Benutzungsschnittstelle für die Kommunikation mit personellen wie maschinellen Aufgabenträgern wird durch **Interface Objekttypen (IOT)** in einem **Interface Objektschema (IOS)** abgebildet.

Die Freiheitsgrade hinsichtlich der Gestaltung der korrespondierenden Sichten einer Modellebene nehmen von oben nach unten ab und werden durch die Abstände zwischen den beiden Schenkeln des V-Modells symbolisiert (siehe Abbildung 5.5) [FeSi06, 188]. Das Prinzip der Objektorientierung sieht die Kapselung von Struktur- und Verhalten eines Objektes vor (siehe Kapitel 6.3). Daher wächst mit zunehmendem Detaillierungsgrad der Modellierung von Struktur- und Verhaltenssicht die Notwendigkeit der Abstimmung zwischen beiden, um eine objektorientierte Spezifikation von Anwendungssystemen ableiten zu können [Mali97, 16].

5.3.1 Architektur des Informationssystems

Die Informationssystem-Architektur des SOM-Konzepts unterscheidet drei Modellebenen eines verteilten Informationssystems [Sinz97, 16f]:

1. Ein betriebliches System wird auf der ersten Modellebene als verteiltes System von kooperierenden Geschäftsprozessen spezifiziert, welches sowohl den Lenkungsanteil als auch des Leistungsanteil der Geschäftsprozesse abbildet. Diese Modellebene ist damit Bindeglied zur Leistungserstellung des betrieblichen Systems.

[22] Wie bereits erwähnt, werden Personal sowie Maschinen und Anlagen hier nicht weiter betrachtet.

[23] Eine ähnliche Kategorisierung in *entity objects*, *interface objects* und *control objects* wird in [Jaco93] vorgeschlagen. Als weitere mögliche Entwurfsmetapher ist zum Beispiel die WAM-Methode zu nennen, die eine Kategorisierung des Anwendungsbereichs nach Werkzeug, Automat und Material verwendet (vgl. [Züll98]).

2. Modellebene zwei spezifiziert verteilte Anwendungssysteme als maschinelle Aufgabenträger für die automatisierten Aufgaben der Geschäftsprozesse in Form einer Fachkonzept- und einer Software-Konzept-Spezifikation. Bei der Verknüpfung der ersten und dieser Ebene werden einzelnen Komponenten des Geschäftsprozessmodells korrespondierende Komponenten der Anwendungssysteme zugeordnet. Die Anpassbarkeit der Informationssystem-Architektur wird verbessert, indem isomorphe Beziehungen der Strukturen beider Ebenen geschaffen werden.

3. Auf unterster Ebene werden anwendungsneutrale Plattformen als virtuelle Maschinen für die Realisierung der verteilten Anwendungssysteme spezifiziert.

5.3.2 Fachliche Spezifikation von Anwendungssystemen

Die Aufgaben von Geschäftsprozessen werden von Aufgabenträgern durchgeführt und können nach ihrer Automatisierbarkeit bzw. ihrem Automatisierungsgrad klassifiziert werden. Betriebliche Anwendungssysteme stellen dabei maschinelle Aufgabenträger für Aufgaben des Informationssystem-Anteils[24] dar, die als automatisierbares und automatisiertes Teilsystem abgegrenzt werden [FeSi06, 208ff].
Die **fachliche Spezifikation eines Anwendungssystems**[25] wird in der SOM-Methodik durch das konzeptuelle Objektschema (KOS) und das darauf operierende Vorgangsobjektschema (VOS) gebildet (siehe unten). Sie ist Ausgangspunkt für den systemtechnischen Entwurf sowie für die Realisierung des Anwendungssystems. Das Anwendungssystem wird als objektorientiertes und objektintegriertes verteiltes System verstanden [FeSi06, 211f].

- **Objektorientiertes System**:
  Das Anwendungssystem wird durch ein konzeptuelles Objektschema (KOS) und ein Vorgangsobjektschema (VOS) in durchgängig objektorientierter Form spezifiziert (siehe auch das folgende Kapitel 6.3).

- **Verteiltes System**:
  Das Anwendungssystem ist ein integriertes System, bestehend aus mehreren autonomen

[24] Der Basissystem-Anteil bleibt im Folgenden unberücksichtigt (vgl. [FeSi06, 208]).
[25] Das Metamodell für die Spezifikation von Anwendungssystemen findet sich u. a. in [FeSi06, 221].

Komponenten, die in der Verfolgung gemeinsamer Ziele kooperieren. Keine der Komponenten besitzt die globale Kontrolle über das System (siehe [FeSi94b]).

- **Objektintegriertes System**:
  Im Gegensatz zur engen Kopplung der Datenintegration, sind die Teil-Anwendungssysteme eines objektorientierten Anwendungssystems autonom und untereinander lose gekoppelt[26]. Die Konsistenz des Anwendungssystems wird dadurch gewahrt, dass zwischen den Teil-Anwendungssystemen Nachrichten anhand detaillierter Kommunikationsprotokolle ausgetauscht werden (siehe hierzu auch Kapitel 4.2.3).

Die **Abgrenzung von Anwendungssystemen**, also die „[...] Festlegung seiner Aufgabenmenge und seiner Aufgabenträger, der Gestaltung der Kommunikation mit seiner Außenwelt und ggf. der Modellierung seiner Außenwelt“ [Fers92, 5], erfolgt in der SOM-Methodik prozessorientiert [FeSi06, 210]. Im Gegensatz dazu fanden und finden in der Praxis häufig funktionsorientierte Kriterien Anwendung zur Abgrenzung von Anwendungssystemen. In einer Zeit, in der der Druck auf Unternehmen, ihre Geschäftsprozesse anzupassen, stetig zunimmt, erweisen sich jedoch nach Funktionsbereichen (z. B. Beschaffungsbereich oder Produktionsbereich) abgegrenzte Anwendungssysteme häufig als Barrieren für die nachgelagerte software-technische Umsetzung dieser Anpassungen [FeSi96a][27].

Automatisierung von Geschäftsprozessen durch Anwendungssysteme

Die Systementwicklung ist ganzheitlich als Teilaufgabe der Gestaltungsaufgabe *Automatisierung betrieblicher Informationssysteme* (vgl. [Fers92, 4]) mit dem Sachziel *Vorgangsautomatisierung* zu sehen. Der Begriff **Automatisierung** beschreibt die Zuordnung von Aufgaben zu Aufgabenträgern [FeSi84, 19]. Die Beschreibung des **Automatisierungsgrades** eines Unternehmens hängt von der gewählten Zerlegungsstruktur seiner Aufgaben ab [FeSi06, 51]. Es wird zwischen Automatisierbarkeit und Automatisierung von Aufgaben und Transaktionen unterschieden [FeSi06, 209f]. Vor der eigentlichen Automatisierung muss die entsprechende Automatisierbarkeit überprüft werden. Die Zerlegung erfolgt ausgehend von einer globalen Aufgabe,

[26] Das Kernprinzip der losen Kopplung beschreiben HOHPE & WOOLF als die Reduktion von Annahmen, die zwei Parteien (z. B. Komponenten, Applikationen, Dienste, Programme, Nutzer) übereinander treffen müssen, wenn sie Informationen austauschen [HoWo03, 10].

[27] Siehe hierzu auch Kapitel 7.3. Das IBM SanFrancisco Framework war an den Funktionsbereichen und nicht an den Prozessen der Unternehmung orientiert [Zeid99, 8].

bei der es sich zumeist um eine teilautomatisierbare Entscheidungsaufgabe[28] handelt. Diese Aufgabe wird solange sukzessive in Teilaufgaben zerlegt, bis jede Teilaufgabe entweder als automatisierbar oder nicht-automatisierbar gilt. Diesen werden die entsprechenden Aufgabenträger zugeordnet.

Das Konzept der Automatisierung von Aufgaben und Transaktionen dient in SOM ausgehend vom Interaktionsschema (IAS) und Vorgangs-Ereignis-Schema (VES) zur vollständigen Beschreibung der Beziehung zwischen einem hinreichend detaillierten Geschäftsprozessmodell (Aufgabenebene) und zugehörigen Anwendungssystemen (Aufgabenträgerebene) [FeSi06, 211ff].

Die **Automatisierung von Aufgaben** bezieht sich auf die Automatisierung der Aufgabendurchführung, die auch als Vorgangsautomatisierung bezeichnet wird. Das Sachziel *Vorgangsautomatisierung* umfasst die Automatisierung der Aktionen, die Automatisierung der Vorgangssteuerung sowie die Automatisierung der Vorgangsauslösung [Fers92, 9]. Es können automatisierte, teilautomatisierte und nicht automatisierte Aufgaben [FeSi84, 19f] des Informationssystem-Teils von Geschäftsprozessen unterschieden werden. Während erstere vollständig von einem Anwendungssystem durchgeführt werden, nutzen personelle Aufgabenträger zur Durchführung teilautomatisierter Aufgaben kooperativ Anwendungssysteme. Wird eine Aufgabe hingegen vollständig von einem personellen Aufgabenträger erbracht, so gilt sie als nicht-automatisiert.

Analog können bei der **Automatisierung von Transaktionen**[29] automatisierte und nicht-automatisierte Transaktionen unterschieden werden. Ersteren liegt ein Kommunikationssystem Mensch-Computer (MCK) bzw. Computer-Computer (CCK) zu Grunde. Nicht-automatisierte Transaktionen werden durch ein Mensch-Mensch-Kommunikationssysteme (MMK) durchgeführt [FeSi06, 208].

Die Darstellung von potentieller Automatisierbarkeit und tatsächlicher Automatisierung kann geeigneterweise im Interaktionsschema (IAS) des Geschäftsprozessmodells erfolgen (vgl. [FeSi06, 208f]).

[28] Es werden Transformationsaufgaben (mit und ohne Speicher) sowie Entscheidungsaufgaben unterschieden [FeSi84]. Dabei entsprechen die Strukturen der Entscheidungsaufgaben denen der Transformationsaufgabe, jedoch kommen als Input-Informationen noch Zielgrößen hinzu [FeSi84, 7ff].

[29] Ebenso gelten die Aussagen zur Automatisierbarkeit von Aufgaben auch für Transaktionen.

Konzeptuelles Objektschema (KOS)

Das **konzeptuelle Objektschema** [FeSi06, 211-217]) eines betrieblichen Anwendungssystems stellt eine objektorientierte Erweiterung und Modifikation eines konzeptuellen Datenschemas im **Semantischen Entity Relationship Model**[30] (SERM) dar. Es besteht aus einer Menge untereinander in Beziehung stehender **konzeptueller Objekttypen** (KOT). Die Beschreibung eines konzeptuellen Objekttypen umfasst einen Namen, eine Menge von Attributen, Nachrichtendefinitionen und Operatoren. Während die Attributausprägungen den Zustand des konzeptuellen Objekts (Instanz) beschreiben, legen die Nachrichtendefinitionen die Arten interpretierbarer Nachrichten fest. Die Operatoren (Methoden) sind auf den Attributen des Objekttypen definiert und dienen der Behandlung eingehender Nachrichten.
Objektorientierte Ansätze realisieren Beziehungen zwischen Objekten mit Hilfe so genannter Objekt-Identifikatoren. Dennoch wird beim KOS die quasi-hierarchische Struktur des SERM beibehalten [FeSi06, 214]. Dies wird in Abbildung 5.6 am Beispiel einer M:N-Beziehung[31] verdeutlicht. In der objektorientierten Darstellung c) wird kein Beziehungsobjekttyp repräsentiert (vgl. [Oest98, 52]).

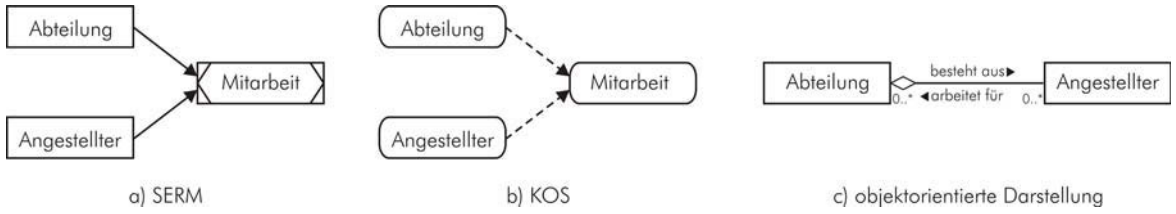

Abbildung 5.6: Relationale und objektorientierte Darstellung einer M:N-Beziehung

Durch die quasi-hierarchische Struktur des KOS werden bidirektionale Beziehungen ausgeschlossen und Existenzabhängigkeiten zwischen KOTs explizit dargestellt. Mit der Folge einer besseren Lesbarkeit des konzeptuellen Datenschemas. Wird das konzeptuelle Objektschema auf ein relationales Datenschema abgebildet, wird der *Impedance Mismatch* (siehe Kapitel 9.6) verringert. Als weiterer Vorteil kann der geringere Speicherverbrauch und verbesserte Performance im Vergleich zur Abbildung bidirektionaler Beziehungen zwischen Objekttypen

[30] Das SERM wurde von SINZ entwickelt und stellt eine „stark verbesserten Nachfolger des Entity Relationship-Modells“ [Wint98, 22] von CHEN dar [Chen76]. Für eine ausführliche Beschreibung des SERM wird auf [Sinz87] und [Sinz92] verwiesen.

[31] Eine Abteilung kann beliebig viele Mitarbeiter haben und ein Angestellter kann in beliebig vielen Abteilungen arbeiten. Es kann auch Abteilungen ohne Mitarbeiter geben sowie Angestellte, die in keiner Abteilung arbeiten.

angeführt werden [WaMa04, 51f].

Grundlage für das KOS ist das Geschäftsprozessmodell der zweiten Modellebene der SOM-Unternehmensarchitektur. Der für die **Ableitung des KOS** relevante Ausschnitt bestimmt sich anhand einer oder mehrerer betrieblicher Objekte, für die ein betriebliches Anwendungssystem spezifiziert werden soll. Das KOS wird initial unter Beachtung der Existenzabhängigkeiten zwischen den einzelnen KOT aus Interaktionsschema und Vorgangs-Ereignis-Schema abgeleitet. Ausgehend von diesem initialen KOS erfolgt die weitere Spezifikation zyklisch in mehreren Schritten. Dabei werden diejenigen KOT entfernt, die nicht-automatisierte Aufgaben und Transaktionen repräsentieren, die Kardinalitäten der Beziehungen zwischen den KOT werden ermittelt und den KOT werden Attribute sowie Nachrichtendefinitionen und Operatoren zugeordnet. Abschließend werden KOT mit sich deckenden Attributen und/oder Operatoren zusammengefasst, um Daten- und Funktionsredundanz zu vermeiden [FeSi06, 215f] (vgl. Kapitel 5.6).

Vorgangsobjektschema

Das **Vorgangsobjektschema (VOS)** (siehe [FeSi06, 218-221]) spezifiziert das Verhalten des zu konstruierenden Anwendungssystems zur zielgerichteten Durchführung der betrieblichen Aufgabe [Sinz95, 20]. Es besteht aus einer Menge von untereinander in *interacts_with*-Beziehung stehenden Vorgangsobjekttypen (VOT), die jeweils das Zusammenwirken von KOT bei der Durchführung einer betrieblichen Aufgabe beschreiben. Die Beschreibung eines VOT umfasst einen Namen, Attribute, Nachrichtendefinitionen sowie Operatoren. Durch die Attribute wird das Aufgabenobjekt der vom VOT durchzuführenden Aufgabe beschrieben. Es umfasst alle KOT, deren Attribute den Zustandsraum der Aufgabe bilden. Nachrichtendefinitionen beschreiben alle Ereignisse, die ein VOT empfangen sowie senden kann und verknüpft diese mit Operatoren. Diese repräsentieren das Lösungsverfahren auf dem Aufgabenobjekt für die diesem VOT zugeordnete Aufgabe [FeSi06, 218].

Die **Spezifikation des Vorgangsobjektschemas** wird ausgehend von einem Ausschnitt des Vorgangs-Ereignis-Schemas (VES) eines Geschäftsprozessmodell abgeleitet. Dabei wird jede Aufgabe des VES in einen Vorgangsobjekttyp abgebildet und jede Ereignisbeziehung[32] zwischen den Aufgaben in eine zwei VOT verbindende *interacts_with*-Beziehung. Im Anschluss

[32] Einschließlich der Transaktionen, die ebenfalls Ereignisse sind.

werden alle nicht automatisierbaren VOT entfernt. Den verbleibenden VOT werden ihre Attribute in Form eines Teilgraphen des korrespondierenden KOS zugeordnet, wobei die Teilgraphen verschiedener VOT nicht zwingend disjunkt sind. Die Nachrichtendefinitionen werden korrespondierend zu den Ereignisbeziehungen bestimmt und die Operatoren werden spezifiziert. Abschließend werden zur Wahrung der semantischen Integrität des Anwendungssystems diejenigen VOT zusammengefasst, deren Aufgaben stets gemeinsam durchzuführen sind [FeSi06, 219] (vgl. Kapitel 5.6).

Interface Objektschema

Das Interface Objektschema (IOS) besteht aus einer Menge von Interface- Objekttypen (IOT), die der fachlichen Spezifikation der Kommunikation mit der Umgebung des Anwendungssystems dienen. Das Schema für Interface Klassen spezifiziert Schnittstellen für die Mensch-Computer-Kommunikation (MCK) sowie die Computer-Computer-Kommunikation (CCK). Mensch-Computer-Schnittstellen werden verwendet, um teilautomatisierte Aufgaben zu implementieren [FeSi94b, 15]. Bei teilautomatisierten Aufgaben wird in der Regel die Vorgangssteuerung und/oder die Vorgangsauslösung von einem Menschen ausgeführt.

5.4 Modellgetriebene Software-Entwicklung

Die Systementwicklung stellt eine Aufgabe dar, für deren Lösungsverfahren ein großes Automatisierungspotenzial besteht. Die **modellgetriebene Software-Entwicklung** (MDSD[33]) zielt auf die automatische Ableitung und Generierung von Programmquellcode aus domänenspezifischen Modellen betrieblicher Anwendungssysteme ab [Völt05, 1]. Sie wird im Gegensatz zu *modellbasierter* Nutzung des Modells im Entwicklungsprozess gesehen, die vor allem durch den Einsatz der *Unified Modeling Language* (UML) eine gewisse Tradition hat. Das Modell dient dort lediglich als Dokumentation des Software-Systems, die Verbindung zur Software-Implementierung ist rein gedanklich. Durch die automatisierte Umsetzung, ist das Modell bei der modellgetriebenen[34] Software-Entwicklung hingegen mit Code gleichzusetzen [StVö05,

[33] Englisch *Model Driven Software Development*.

[34] Das Adjektiv „getrieben“ soll im Gegensatz zu „basiert“ deutlich machen, dass den Modellen bei diesem Paradigma eine treibende und damit zentrale Position zukommt, die den gleichen Stellenwert wie der Quellcode selbst besitzt [StVö05, 4].

3]. Voraussetzung für einen solchen Schritt ist zum einen eine Modellierungsmethodik mit entsprechend definierten Ableitungsregeln, wie sie mit SOM vorliegt. Zum anderen eine geeignete Software-Architektur für die software-technische Abbildung des Anwendungsmodells. Ziel der Arbeit ist es, mit der Implementierung einer geeigneten Architektur durch ein Software-Framework diese Voraussetzung zu erfüllen.

Zur Formalisierung der fachlichen Modelle ist eine **domänenspezifische Modellierungssprache** (DSL[35]) notwendig. Sie enthält Artefakte der jeweiligen Domäne und kann als Referenzmodell der Anwendung verstanden werden. Die Bestimmung und Ableitung eines solchen Referenzmodells ist nicht Teil dieser Arbeit. Zur Entwicklung von Referenzmodellen betrieblicher Anwendungssysteme und domänenspezifische Ausprägungen siehe u. a. [Ohle98] und [Rüff99]. STAHL & VÖLTER beschreiben drei Bedingungen, die notwendig sind, um das Konzept der domänenspezifischen Modelle erfolgreich umsetzen zu können [StVö05, 4]:

- Domänenspezifische Sprachen zur Formulierung der Modelle.
- Sprachen, um die Transformationen von Modell zu Code auszudrücken.
- Entsprechende Compiler, Generatoren oder Transformatoren, um aus den Modellen Programme zu erzeugen.

MDA

Die **modellgetriebene Architektur** (MDA[36]) ist eine Standardisierungsbestrebung der OMG[37], die dem Paradigma der MDSD folgt und eine Spezialisierung darstellt. Ihr liegt kein geschäftsprozessorientiertes Modell zu Grunde, sondern eine Spezifikation des Anwendungssystems entsprechend der Notifikation von UML 2.0[38]. Daher ist nicht das konkrete Vorgehen mittels MDA[39] Gegenstand der Arbeit. Eine ausführliche Beschreibung findet sich u. a. in [Fran03], [MDA03] und [StVö05, 373-396], ein Beispiel für den Einsatz in der Praxis wird in [CoOs04b] beschrieben.

[35] Englisch *Domain-Specific Language*.
[36] Englisch *Model Driven Architecture*.
[37] Die *Object Management Group* ist ein Non-Profit-Konsortium mit dem Ziel, Standards und Spezifikationen für die Computer-Industrie zu schaffen. Mehr Informationen finden sich unter `http://www.omg.org`.
[38] Zu MDA und UML 2.0 siehe u. a. [StBo03].
[39] Vgl. auch `http://www.omg.org/mda/`.

Transformation

Die modellgetriebene Entwicklung wird in mehreren Schritten durchgeführt. Das Grundprinzip ist in Abbildung 5.7 dargestellt. Zunächst wird in einer formalen Modellierungssprache die fachliche Spezifikation des Anwendungssystems in einem **plattformunabhängigen Modell** (PIM[40]) definiert und somit die Unabhängigkeit von der Zielplattform der späteren Implementierung gewahrt. Mittels Modell-Transformation (vgl. hierzu u. a. [MaBr03] und [Mars05]) wird die plattformunabhängige fachliche Spezifikation in ein **plattformspezifisches Modell** (PSM[41]) überführt, welches die spezifischen Konzepte der Zielplattform enthält. Durch Transformation eines oder mehrerer PSM wird die Implementierung für die konkrete Zielplattform erzeugt [StVö05, 17ff].

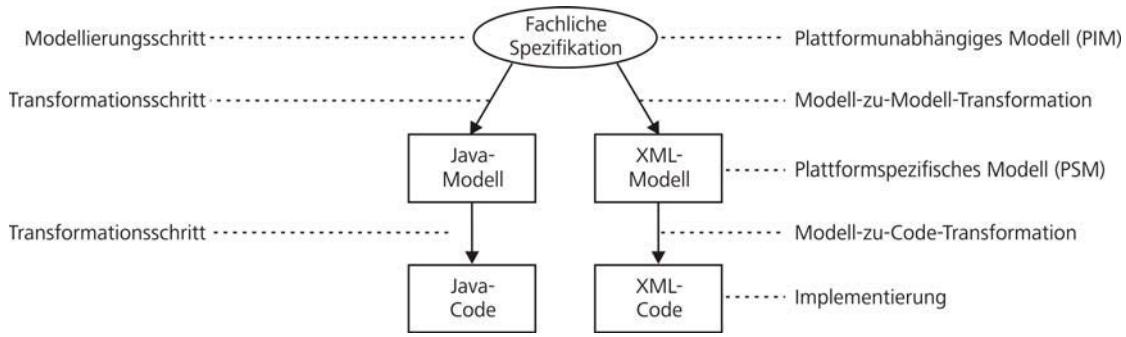

Abbildung 5.7: Grundprinzip des MDSD (nach [StVö05, 18])

Sowohl beim PIM als auch beim PSM handelt es sich um Konzepte, die sich relativ zur Plattform unabhängig oder spezifisch verhalten. Ein PSM mit Persistenz-Konzepten ist noch unabhängig von der konkret eingesetzten Technologie der Persistierung, also aus der Sicht der Persistenz-Plattform ein PIM. Durch eine weitere Transformation auf ein bestimmtes DBMS wird daraus das konkrete PSM erzeugt. Wie in Abbildung 5.8 dargestellt, können bis zur letztgültigen Implementierung durchaus in mehreren Stufen jeweils konkretere PSM durch Transformation erzeugt werden.

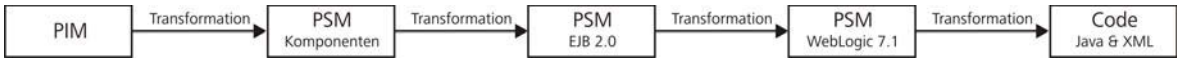

Abbildung 5.8: Transformationen des MDSD [StVö05, 18]

[40] Englisch *Platform Independent Model*.
[41] Englisch *Platform Specific Model*.

Software-Architektur

Die Software-Architektur nimmt im Rahmen der MDSD eine besondere Rolle ein, da die Konstrukte eines Quell-Metamodells (fachliche Spezifikation des Anwendungssystems) auf die entsprechenden Konstrukte eines Ziel-Metamodells (Software-Architektur) abgebildet werden müssen [Völt05, 7]. Das entspricht einer wohldefinierten Architektur der Zielplattform zur Entwicklung betrieblicher Anwendungssysteme (vgl. Kapitel 3.3), die den Entwickler bei der Realisierung der Anwendung unterstützt. Die Plattform stellt eines der Schlüsselkonzepte der modellgetriebenen Software-Entwicklung dar [StVö05, 21] und wird durch das hier entwickelte Framework realisiert (siehe Kapitel 8). Die **architekturzentrierte modellgetriebene Software-Entwicklung** (vgl. [StVö05, 24-33]) rückt methodische Aspekte in den Vordergrund. Voraussetzung für sie ist eine formalisierte Software-Architektur wie sie von Frameworks geboten wird [John97, 10]. Als Eingabe wird ein fachliches Modell der Anwendung benötigt, aus dem der Infrastrukturcode des Anwendungssystems mittels Schablonen (*Templates*) erzeugt wird [StVö05, 26]. Durch architekturzentriertes Design wird auf die Entwicklung von PSM auch zu Gunsten einer Vereinfachung der Entwicklung verzichtet, was im klaren Gegensatz zur MDA der OMG steht. Bei der Einführung von modellgetriebener Software-Entwicklung ist der Fokus auf architekturzentrierte MDSD ratsam [Völt05, 11].

5.5 Geschäftsprozessmodellgetriebene Software-Entwicklung mit SOM

Der Realisierung des Anwendungssystems entspricht der Festlegung und Implementierung von Aufgabenträgern und Lösungsverfahren automatisierbarer betrieblicher Aufgaben. Dem geht die Modellierung der Aufgabenebene voraus. Für die Modellierung der Aufgabenebene von Informationssystemen stellt die Objektorientierung eine der wichtigsten Metaphern dar [FeSi06, 129].

Das Semantische Objektmodell ist eine durchgängig geschäftsprozess- sowie objektorientierte Modellierungsmethodik. Auf den Ebenen des Geschäftsprozessmodells sowie der Anwendungsspezifikation sind korrespondierende Metamodelle verfügbar sowie Beziehungs-Metamodelle, um den Bezug der Geschäftsprozessmodelle zu den Modellen der fachlichen Spezi-

fikation des Anwendungssystems herzustellen [Sinz97, 11]. Das *top-down*-Vorgehen einerseits und die durchgängige Ableitbarkeit der Modelle andererseits ermöglichen die automatisierte Generierung entsprechender Fachspezifikationen mit der SOM-Methodik.

Die Ableitung und Konstruktionsregeln von Modellen in SOM sind nicht Gegenstand der Arbeit, die sich mit der software-technischen Architektur des Zielsystems der Ableitung beschäftigt. Es soll aber gezeigt werden, dass die Modelle der Ebenen *Geschäftsprozessmodelle* und *Anwendungsspezifikation* der SOM-Methodik als Grundlage für die architekturzentrierte Entwicklung betrieblicher Anwendungssysteme geeignet sind. MALISCHEWSKI beschreibt in [Mali97] sowohl die Ableitung der Grobstruktur der fachlichen Spezifikation von Anwendungssystemen aus dem Geschäftsprozessmodell als auch die software-technische Spezifikation von Anwendungssystemen. MALISCHEWSKIS Arbeit umfasst außerdem die Beschreibung eines vom Autor entwickelten *CASE-Tools*[42], mit dem die Generierung initialen Quellcodes für verschiedene Basismaschinen (z. B. C++, Smalltalk) möglich ist.

Ableitung der Grobstruktur der fachlichen Spezifikation von Anwendungssystemen

Die fachliche Spezifikation des Anwendungssystems (siehe [Mali97, 37-72]) bezieht sich auf Modellkonstrukte der Geschäftsprozessmodellierung. Dieser Bezug wird in SOM durch ein Beziehungs-Metamodell sowie zugehörige Transformationsregeln festgelegt. Zudem ist dieser Bezug für den Einklang des Anwendungssystems mit den Geschäftsprozessen[43] der Unternehmung und dessen Wiederverwendbarkeit notwendig [Raue96, 4].

Das Anwendungssystem kann so verstanden werden, dass es personelle Aufgabenträger bei der Durchführung betrieblicher Aufgaben im Rahmen von Geschäftsprozessen unterstützen soll. Ausgehend von einem Interaktionsschema (IAS) und korrespondierendem Vorgangs-Ereignis-Schema (VES) werden Teilschemata für ein betriebliches Objekt definiert, die Grundlage der folgenden Ableitung sind. Die Ableitung erfolgt anhand eines Ziel-Metamodells [Mali97, 40] gemäß der SOM-Unternehmensarchitektur. Es wird zunächst die Struktur des objektinternen Speichers eines betrieblichen Objektes in Form eines konzeptuellen Objekttypen abgeleitet. Anschließend erfolgt die Ableitung der Vorgangsobjekttypen als Träger der Lösungsverfahren

[42] *CASE* steht für *Computer-Aided Software Engineering*.

[43] Auf die Konstruktion entsprechender Geschäftsprozessmodelle wird hier nicht weiter eingegangen. Vgl. hierzu [Raue96].

der zu automatisierenden Aufgaben dieses betrieblichen Objekts. Die Kommunikation der zum Teil verteilten Anwendungssysteme und vor allem die Kommunikation mit dem Benutzer wird in Form von Interface-Objekttypen modelliert.

Software-technische Spezifikation von Anwendungssystemen

Ziel der software-technischen Spezifikation von Anwendungssystemen (siehe [Mali97, 73-82]) ist die Erstellung eines möglichst basismaschinenunabhängigen Modells des Anwendungssystems. Die Realisierung des Anwendungssystems mit Hilfe von Basismaschinen (z. B. des im Rahmen der Arbeit erstellten Frameworks) erfolgt auf Grundlage dieses Modells.
Die Ableitung erfolgt anhand eines Metamodells [Mali97, 74], das den Rahmen bildet zur Erweiterung fachlicher Anwendungssystem-Spezifikationen bezogen auf die Realisierung von Anwendungssystemen. Es erweitert das Metamodell zur Ableitung der fachlichen Spezifikation um Typisierungen von Attributen, Parametern und Rückgabewerten, Nicht-Objekttypen sowie Fehlerbehandlungen.
Nach erfolgter Ableitung der software-technischen Spezifikation und Generierung des initialen Quellcodes hilft ein korrespondierendes Application Framework den Realisierungs-, Integrations- und Testaufwand zu begrenzen [Mali97, 197]. Die Grundlagen dafür und die Realisierung eines solchen als Zielplattform werden in den folgenden Kapiteln vorgestellt. Die Entwicklung eines geeigneten *CASE-Tools* für die Transformation der Modelle und die Generierung des basismaschinenspezifischen Quellcodes würden den Rahmen der Arbeit sprengen. Es wird daher nur die Zielplattform betrachtet, die korrespondierenden Modelle der Anwendung werden als gegeben angenommen.

Vorteile geschäftsprozessmodellgetriebener Anwendungsentwicklung

Der modellgetriebene Software-Entwicklungsprozess bietet einige Vorteile gegenüber klassischem Vorgehen, bei dem der Entwickler wesentlich mehr Routinearbeit zu verrichten hat. Die Systementwicklungsaufgabe kann durch die Automatisierung hinsichtlich Umfang und Komplexität erheblich verringert werden. Die geschäftsprozessmodellgetriebene Entwicklung verknüpft diese Vorteile der automatischen Transformation von Modellen und der anschließenden Code-Generierung mit den Vorteilen der Anforderungsableitung aus dem betrieblichen

Informationssystem (*top-down*-Vorgehen).

Neben der höheren Produktivität kann auch von einer verbesserten Qualität des entwickelten Anwendungssystems ausgegangen werden, da fehlerträchtige Routinearbeit von Rechnern ausgeführt wird. Bezüglich der Flexibilität sind vor allem Strukturähnlichkeiten und -analogien zwischen den Teilsystemen des Informationssystems ausschlaggebend. Da das Anwendungssystem direkt aus den zu Grunde liegenden Geschäftsprozessen der betrieblichen Informationssystems abgeleitet wird, können Änderungen an den Geschäftsprozessen direkt im Anwendungssystem nachvollzogen werden [FeSi06, 210]. Zusätzlich erfolgt die Ableitung der initialen Grobstruktur des Software-Systems automatisiert aus Modellen, so dass notwendige Anpassungen in aller Regel nur auf Ebene des Geschäftsprozessmodells eingepflegt werden müssen, die dann automatisch durch Transformation im Software-System nachgezogen wird. Der Abgleich unterstützt die Konsistenz des Systems, die automatische Erzeugung die Produktivität.

Die höhere Produktivität kann die Notwendigkeit von Off-Shoring, also der Auslagerung von Entwicklungsprozessen ins kostengünstigere Ausland, vermeiden helfen. Der Vorteil der Nicht-Auslagerung der Entwicklung, die durch die geschäftsprozessmodellgetriebene Anwendungsentwicklung ermöglicht wird, ist der Verbleib des Know-How im Haus. Zudem kann die so genannte Time-to-Market verkürzt werden. Ein Vorteil, der durch klassisches Outsourcing nicht erreicht werden kann [StVö05, 350ff].

5.6 Unterstützung der Integration

Wie in Kapitel 4.2.3 beschrieben, handelt es sich bei der SOM-Methodik um einen objektintegrierten Modellierungsansatz. Ausgehend von den Fachspezifikationen Vorgangsobjektschema (VOS) und konzeptuelles Objektschema (KOS) der Ebene 3 des SOM-Vorgehensmodells (siehe Kapitel 5.3) ergeben sich **Unterstützungspotenziale** für die Verfolgung von Integrationszielen durch Komponenten des Frameworks. Die durchgehende Unterstützung der Integrationszielverfolgung von der fachlichen Modellierung bis zur technischen Implementierung ist von großer Bedeutung für die effiziente Entwicklung qualitativ hochwertiger Anwendungssysteme (siehe hierzu auch [Rose99]). Die Unterstützungspotenziale werden im Folgenden einzeln vorgestellt.

5.6.1 Unterstützung strukturorientierter Integrationsziele

Kontrolle der Redundanz

Es werden Funktions- und Datenredundanz unterschieden [Fers92, 12]. Bei der Verknüpfung der Modellebenen *Geschäftsprozessmodell* und *Spezifikation von Anwendungssystemen* werden den einzelnen Komponenten des Geschäftsprozessmodells korrespondierende Komponenten des Anwendungssystems zugeordnet. Sind Funktions- und Datenredundanzen auf Ebene der Anwendungssysteme vorhanden, so werden diese ebenso aufgedeckt, wie eventuelle Automatisierungslücken. Bei der Abstimmung der beiden Modellebenen können diese korrigiert werden [Sinz97, 16f].

Der SOM-Ansatz ist damit geeignet, schon auf Ebene der Modellierung und Ableitung der fachlichen Spezifikation des Anwendungssystems die Verfolgung der Integrationsziele Funktions- und Datenredundanz zu unterstützen.

Kontrolle der Kommunikationsstruktur

Ein in SOM modelliertes und damit objektorientiertes Anwendungssystem besteht aus lose gekoppelten Objekttypen (IOT, VOT, KOT), die miteinander in Beziehung stehen [FeSi06, 211f]. Das Integrationsmerkmal Verknüpfung bezieht sich auf die Kontrolle der zu Grunde liegenden **Kommunikationsstruktur** [Fers92, 13]. In Abbildung 5.9 ist ein Ausschnitt der fachlichen Objekte eines Anwendungssystems in SOM-Notation und die zugehörige Kommunikationsstruktur dargestellt.

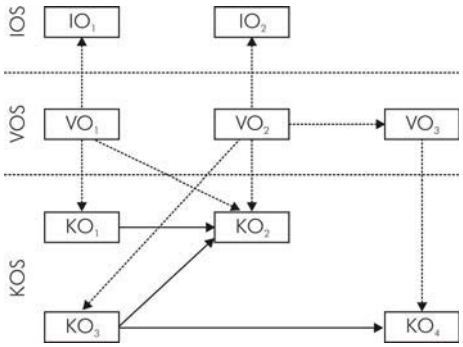

Abbildung 5.9: Beispiel Kommunikationsstruktur der fachlichen Objekte eines Anwendungssystems

Das Framework soll eine Komponente bereitstellen, mit der die Verwaltung und **Kontrolle der**

Systembeziehungen (vgl. auch [Grif98, 116]) zur Entwicklungs- und Laufzeit unabhängig vom Quelltext ermöglicht wird. Zur Entwicklungszeit ist damit an zentraler Stelle erfasst, welcher VOT auf welchen KOT zugreift. Wird beispielsweise ein KOT gelöscht, so sind die Auswirkungen auf die in Beziehung stehenden VOT sofort sichtbar[44] und der Entwickler kann gezielt die entsprechenden Vorgangsobjekte anpassen. Zur Laufzeit kann das Framework auf Basis der erfassten Beziehungen sicherstellen, dass nur berechtigte Vorgangsobjekte auf die jeweiligen konzeptuellen Objekte zugreifen. Wird eine Beziehung versehentlich nicht erfasst, so wird zur Laufzeit beim unberechtigten Zugriff eine entsprechende Meldung generiert. Die in der Komponente des Frameworks erfassten Beziehungen müssen folglich konsistent gehalten werden, um die Lauffähigkeit des Anwendungssystems sicherzustellen. Sie bilden gleichzeitig die Interaktionen der Fachspezifikationen VOS und KOS der dritten Ebene der SOM-Methodik ab. Die Komponente als Teil des Frameworks verhindert, dass der Entwickler bei der Implementierung des Anwendungssystems die Kontrollkomponente umgeht (siehe Kapitel 8.3.4).

5.6.2 Unterstützung verhaltensorientierter Integrationsziele

Hinsichtlich der Unterstützung verhaltensorientierter Eigenschaften von Anwendungssystemen werden die Merkmale **Konsistenz** im Sinne von Widerspruchsfreiheit und Vollständigkeit [ScSt83, 275] sowie **Zielverfolgung** unterschieden. Das Framework soll durch Komponenten die Verfolgung der **verhaltensorientierten Integrationsziele** *Integrität* und *Vorgangssteuerung* unterstützen.

Kontrolle der Integrität

Bezogen auf die Lenkungsaufgabe des Management eines Betriebes stellt das erfasste Modellsystem des betrieblichen Informationssystems eine Hilfsregelstrecke dar, anhand derer die eigentliche Regelstrecke - das betriebliche System - gelenkt wird [Sinz97, 11]. Das betriebliche Anwendungssystem als automatisiertes Teilsystem des betrieblichen Informationssystems [Ambe93, 11] muss daher stets ein konsistentes, sprich widerspruchsfreies und vollständiges, Abbild des entsprechenden Ausschnitts der betrieblichen Diskurswelt repräsentieren.

[44] PIETSCH weist darauf hin, dass die Durchsuchung des Quelltextes nach Einbindung des entsprechenden KOT keine verlässliche Alternative darstellt [Piet04, 35]. Grund dafür ist die Late-Binding-Fähigkeit moderner objektorientierter Programmiersprachen (siehe hierzu [Oest98, 60f]).

Um **semantische Integrität** zu gewährleisten, werden bereits bei der Ableitung der fachlichen Spezifikation in SOM diejenigen Vorgangsobjekttypen zusammengefasst, deren korrespondierende Aufgaben stets gemeinsam durchzuführen sind [FeSi06, 219]. Zudem müssen auf software-technischer Seite semantische Integritätsbedingungen formuliert und durch das Framework überprüft werden. Die Überprüfung der semantischen Integritätsbedingungen kann durch das Programmsystem (in diesem Fall das Framework) oder durch das Datenbankverwaltungssystem (DBVS) erfolgen. Die Überprüfung durch eine Komponente des Frameworks bietet mehrere Vorteile [Neum99, 135]:

- Semantische Integritätsbedingungen werden häufig schon als Teil der Bearbeitungslogik berücksichtigt.
- Die Überprüfung in der Anwendung erfolgt effizient nur auf die tatsächlichen Erfordernisse.
- Die Überprüfung kann in Teilen schon während der Dateneingabe erfolgen (bspw. Gültigkeit von Attributwerten).
- Spezifische Fehlermeldungen können vom System erzeugt und Fehler dediziert behandelt werden.
- Die Implementierung der Integritätsbedingungen ist nicht mehr abhängig vom verwendeten DBVS, mit dem Ergebnis einer leichteren Portierbarkeit der Anwendung (siehe auch Integrationsziel *Aufgabenträgerunabhängigkeit* in Kapitel 4.1).

Dem stehen verschiedene Nachteile gegenüber [Neum99, 135]. So obliegt die Überprüfung dem Anwendungsentwickler, wodurch diese Gefahr läuft vernachlässigt zu werden. Zudem findet die Überprüfung nicht mehr an zentraler Stelle (DBVS) statt, mit negativen Auswirkungen auf die Übersichtlichkeit. Eine entsprechende Komponente des Frameworks kann die geschilderten Nachteile abmildern bzw. ausschalten und bietet gleichzeitig die vorgenannten Vorteile. Eine entsprechende Komponente wird somit für das Framework gefordert. Die zu unterstützenden statischen Integritätsbedingungen sind Einschränkung von Datentypen, Voreinstellung von Werten, Angabe von Schlüsseln sowie referentielle Integritätsbedingungen [Neum99, 55ff]. Hinzu kommen Bedingungen über mehrere Relationen sowie transitionale Integritätsbedingungen.

Bei letzteren handelt es sich um Wenn-Dann-Bedingungen, die von relationalen DBVS nicht unterstützt werden [Neum99, 66]. Die geforderte Komponente muss jedoch in der Lage sein, diese zu überprüfen.

Um die **operationale Integrität** des Systems sicherzustellen, dürfen parallele Zugriffe auf die Datenbasis keine Inkonsistenz des Systems verursachen. Auf der technischen Ebene schützt das DBVS mit dem **Transaktionskonzept**[45] vor Inkonsistenzen durch parallelen Zugriff [KeEi99, 245f]. Auf Ebene des Programmsystems werden **Synchronisationsverfahren** eingesetzt, um unkontrollierte Nebenwirkungen bei gleichzeitigem Zugriff auf ein Objekt zu verhindern [ScSt83, 294]. Die Komponente des Frameworks, die den Zugriff auf die Datenbasis verwaltet, muss geeignete Verfahren implementieren, um Transaktionsschutz und operationale Integrität bei parallelen Zugriffen zu gewährleisten. Dabei muss dem Anwender die Möglichkeit gegeben werden, über die Isolationsstufe[46] der Transaktion selbst zu entscheiden, da mit zunehmender Isolationsstufe zwar die Möglichkeit inkonsistenter Zustände abnimmt, gleichzeitig die Performance der Programmausführung sich verschlechtert [Piet04, 42]. PIETSCH fordert für eine Komponente zur Sicherung der operationalen Integrität auch die Vermeidung der **Lost Update**-Problematik [Piet04, 43]. Diese tritt dann auf, wenn nach dem Auslesen und Ändern eines Datensatzes, dieser von einer zweiten Stelle in der Datenbasis geändert wurde, bevor die ausgelesene und geänderte Instanz zurückgeschrieben wird. Wird der Datensatz ohne Beachtung der Änderungen in der Datenbasis gespeichert, werden diese überschrieben und gehen verloren (*lost update*).

Als ein weiterer Aspekt des Integrationsziels *Integrität* muss der **Datenschutz** gesehen werden. Die Konsistenz des Zustands des Anwendungssystems ist nur dann gegeben, wenn keine Zustandsänderung von unautorisierten Aufgabenträgern durchgeführt wird. Die Überwachung der systeminternen Zugriffe kann durch die vorher beschriebene Kontrolle der Kommunikationsstruktur gewährleistet werden. Eine weitere Komponente muss vor allem den Zugriff personeller Aufgabenträger authentisieren und autorisieren (vgl. [Summ97, 340f]). Diese wird in Kapitel 8.3.5 vorgestellt. Die von SUMMERS verlangte Netzwerksicherheit muss mittels der für die Netzwerkkommunikation gewählten Basismaschinen realisiert werden.

[45] Das Transaktionskonzept besagt, dass die Eigenschaften einer Transaktion als Folge von Operationen auf der Datenbasis dem ACID-Prinzip entsprechend müssen [FeSi06, 385f]. ACID steht für *Atomicity*, *Consistency*, *Isolation* und *Durability* (vgl. hierzu [HäRe83]).

[46] Es werden die Isolationsstufen *Read Uncommitted*, *Read Committed*, *Repeatable Read* sowie *Serializable* unterschieden [Fowl03, 74].

Die Komponente zur Kontrolle der Integrität wird in Kapitel 8.3.3 beschrieben.

Vorgangssteuerung

Die **Vorgangssteuerung** ist Ergebnis der Zerlegung der Gesamtaufgabe in Teilaufgaben. Jeder Vorgang stellt die Aufgabendurchführung einer Teilaufgabe dar. Diese Teilaufgaben müssen im Hinblick auf die Gesamtaufgabe zielorientiert koordiniert werden, da ein Anwendungssystem aus Außensicht eine Gesamtaufgabe unterstützt [FeSi06, 229]. Eine Steuerung auf höherer Ebene übernimmt die Koordination der Vorgänge. Im Falle einer nicht-automatisierten Steuerung wird sie von einem personellen Aufgabenträger übernommen, der die Vorgänge des Anwendungssystems über die IOTs steuert, ohne direkten Zugriff auf die KOTs zu haben.
Neben der Koordination der Aufgabendurchführung überprüft die Komponente der Vorgangssteuerung vorher die Durchführbarkeit und bricht diese im negativen Fall ab. In Abhängigkeit vom Systemzustand kann ein Vorgang durchführbar sein oder nicht (z. B. zeitabhängig, datumsabhängig oder aber ein Vorgang darf im Sinne der Performanz nur eine bestimmte Zeitspanne beanspruchen). Die bereits beschriebene Autorisierung des Zugriffs ist ebenfalls Teil der Komponente der Vorgangssteuerung [Piet04, 45]. Bei der personellen Vorgangssteuerung muss die Komponente in der Lage sein, dem Nutzer nur durchführbare Vorgänge zur Ausführung anzubieten.
Die Komponente der Vorgangssteuerung wird in Kapitel 8.3.2 vorgestellt.

5.6.3 Unterstützung der Aufgabenträgerunabhängigkeit des Anwendungssystems

Als letztes Integrationsmerkmal wird die **Aufgabenträgerunabhängigkeit** von Anwendungssystemen hinsichtlich der Unterstützungspotenziale durch Komponenten des Frameworks vorgestellt. Die geforderte Unabhängigkeit kann durch die Verwendung abstrakter und standardisierter Basismaschinen erreicht werden. Die Trennung des Anwendungssystem in Schichten erleichtert zudem die Standardisierung und Portierbarkeit [FeSi06, 306] (vgl. auch Kapitel 3.3.2 und Kapitel 3.3.3).

Aufgabenträgerunabhängigkeit der Kommunikation

Anwendungssysteme kommunizieren als Teile eines betrieblichen Informationssystems mit personellen Aufgabenträgern und weiteren Anwendungssystemen. Die Interaktion mit personellen Aufgabenträgern (Mensch-Computer-Kommunikation) wird mittels einer graphischen Benutzungsoberfläche realisiert. Das Framework muss die Realisierung der Kommunikationsschicht ermöglichen, die den transparenten Austausch der entsprechenden Oberfläche erlaubt. Dafür wird ein eigenes GUI-Framework realisiert, das die plattformunabhängige Beschreibung der Interaktionsobjekte ermöglicht (siehe Kapitel 8.2).

Aufgabenträgerunabhängigkeit der Anwendungsfunktionalität

Damit das Programmsystem, das die Anwendungsfunktionalität realisiert, auf verschiedenen Plattformen lauffähig ist, muss das Framework die Implementierung der Anwendung auf einer Basismaschine anbieten, die von der entsprechenden Plattform abstrahiert. Die Anwendungsfunktionalität soll unabhängig von der technischen Realisierung beschrieben werden können, so dass ein Wechsel der darunter liegenden Plattform transparent erfolgen kann. Die für die Realisierung des Frameworks unter diesem Aspekt eingesetzten Basismaschinen werden in Kapitel 9.1 vorgestellt.

Aufgabenträgerunabhängigkeit der Datenverwaltung

Die Datenverwaltung ist für die persistente Verwaltung der konzeptuellen Objekte zuständig. Dies wird mit Hilfe von Datenbankverwaltungssystemen (DBVS) realisiert. Aus Sicht der Anwendung muss die Persistierung und das zu Grunde liegende Paradigma der Persistierung (Relationenmodell, Objektmodell etc.) verborgen sein. Das Framework muss eine Komponente zur Verwaltung der konzeptuellen Datenbasis anbieten, die von der Realisierung und der technischen Datenbasis abstrahiert (siehe Kapitel 8.4). Die Anwendung arbeitet lediglich auf den konzeptuellen Objekten. Wird beispielsweise von einem relationalen auf ein objektorientiertes DBVS gewechselt, muss die Anwendung davon unberührt bleiben. Gleiches gilt für den Austausch eines relationalen DBVS gegen ein anderes.

6 Konzepte der Software-Entwicklung

Im vorangegangene Kapitel wurde mit der SOM-Methodik ein Lösungsverfahren der Systementwicklungsaufgabe vorgestellt, um im Rahmen der Anforderungsanalyse und -definition die zu entwickelnde Nutzermaschine zu definieren. Ausgehend von den dort abgeleiteten Anforderungen an eine geeignete Basismaschine für die software-technische Realisierung der Nutzermaschine, befasst sich dieses Kapitel mit den software-technischen Konzepten, die bei der Gestaltung der Basismaschine zu beachten sind.
Zunächst wird in Kapitel 6.1 der Lösungsraum zwischen den Polen Standard- und Individualsoftware aufgespannt. Kapitel 6.2 widmet sich dem Konzept der Wiederverwendung im Software-Entwicklungsprozess im Allgemeinen und durch Software-Frameworks im Speziellen. Das objektorientierte Paradigma findet in fast allen Phasen der Software-Entwicklung Anwendung. Es wird in Kapitel 6.3 vorgestellt. Ein komplementäres Paradigma stellt die Aspektorientierung dar, die in Kapitel 6.4 dargestellt wird. Eng verknüpft mit dem Framework-Gedanken sind die Komponentenorientierung und die Verwendung von Entwurfsmustern, die in den Kapiteln 6.5 respektive 6.6 beschrieben werden. Abschließend wird in Kapitel 6.7 kurz in das Thema Frameworks eingeführt. Dieses wird im nachfolgenden Kapitel 7 vertieft.

6.1 Standard- und Individualsoftware

Bei der Entwicklung betrieblicher Anwendungssysteme können zwei Pole der Software-Entwicklung unterschieden werden. Auf der einen Seite maßgeschneiderte Individualsoftware, auf der anderen Seite vorgefertigte Standardsoftware[1] [Kirc96, 13f]. Beide bieten spezifische Vor- und Nachteile, die im folgenden Abschnitt vorgestellt werden. Anschließend wird diskutiert, wie Vorteile beider Ansätze kombiniert werden können. Entsprechende Methoden werden wei-

[1] MERTENS bezeichnet diese als Uniform [Mert96, 57] im Gegensatz zum Maßanzug.

ter unten in diesem Kapitel vorgestellt.

6.1.1 Standardsoftware

Von **Standardsoftware** spricht man, wenn diese auf Vorrat gefertigt wird, basierend auf prognostizierten Anforderungen für einen anonymen Markt [Ambe99, 12]. Es kann zwischen funktionsneutraler (z. B. Textverarbeitung, Tabellenkalkulation, Datenbankverwaltung) und funktionsbezogener Standardsoftware (z. B. Finanzbuchhaltung, Lohn & Gehalt) sowie funktionsübergreifender, integrierter Branchensoftware unterschieden werden[2]. Diese zeichnet sich im Idealfall durch die von FRANK beschriebenen Bestimmungsmerkmale aus [Fran80, 15ff] (siehe auch [Kirc96, 14f]):

- Übernahme einer definierten Funktion, einer Problemlösung.
- Generelle Einsatzfähigkeit - auch universelle Software.
- Klare Fixierung und Minimierung der organisatorisch-systemtechnischen Anpassung in zeitlichem und mengenmäßigem Aufwand.
- Eindeutige Festpreisformulierung und -garantie für die komplette, problemlose Integration der Software in das EDV-System des Anwenders mit Erfüllung aller notwendigen Zusatzleistungen (hierunter werden Installation, Dokumentation, Wartung, Schulung etc. verstanden).

Bei Standardsoftware handelt es sich somit um Software, die nur durch Parametrisierung[3] (*customizing*) für den Einsatz beim Kunden angepasst wird (vgl. [Fran80, 29ff] und [Ambe99, 12]). Man spricht dabei von der *Einführung von Standardsoftware* [Kirc96]. Kriterien zur Beurteilung und Auswahl von Standardsoftware-Paketen finden sich in [Fran80, 29ff].

Stärken Die Stärken der Standardsoftware liegen vor allem bei den Faktoren Zeit, Kosten- (sicherheit) und Qualität [Ohle98, 31f]. Dies wird durch einen geringeren Anpassungs-

[2] Die in der vorliegenden Arbeit dargestellten Lösungsverfahren sind zur Erstellung funktionsbezogener sowie funktionsübergreifender, auf Abgrenzung nach Geschäftsprozessen basierenden Anwendungssysteme geeignet.

[3] Bekanntestes Beispiel für betriebliche Standardsoftware ist das ERP-System *R/3* von SAP, das in einem aufwendigen Anpassungs- und Einführungsprozess von meist mehreren Monaten für den Kunden *customized* wird (vgl. u. a. [Kirc96], [Maa+05]).

aufwand im Vergleich zur Neuentwicklung sowie der Wiederverwendung eines bestehenden Systems erreicht. Letzteres reduziert die Fehleranfälligkeit und erhöht dadurch die Software-Qualität[4]. In der Folge sind Kosten und Aufwand für den Einsatz in aller Regel präzise abschätzbar. Durch den mehrfachen Einsatz können die Stückkosten deutlich gesenkt werden.

Schwächen Der Nutzen von Standardsoftware hängt entscheidend von der Flexibilität zur Variantenbildung, der Integrationsfähigkeit, der Portierbarkeit sowie den unterstützten Sprachen ab [Fer+96b, 41]. Je größer die Flexibilität und damit das Einsatzgebiet der Software-Lösung sein soll, desto umfangreicher muss der angebotene Funktionsumfang sein. Das hat einen komplexen Anpassungsprozess durch Parametrisierung zur Folge (vgl. [Mert96]). KIRCHMER beschreibt zudem, dass Standardsoftware in den meisten Fällen funktionsorientiert strukturiert ist [Kirc96, 20], mit der Folge einer erschwerten Anpassung an Änderungen der Geschäftsprozesse des Unternehmens [FeSi06, 210].

Standardsoftware wird vor allem eingesetzt, wenn Inhalt und Durchführung betrieblicher Aufgaben z. B. durch rechtliche Vorgaben oder betriebswirtschaftliche Modelle weitgehend festgelegt sind [Fer+96b, 41].

6.1.2 Individualsoftware

Unter **Individualsoftware** werden hingegen Anwendungen verstanden, die exklusiv für einen Auftraggeber und einen bestimmten Einsatzzweck entwickelt werden. Dabei handelt es sich um Auftragsfertigung [Ambe99, 12f]. Dies kann wiederum auf Basis von Wiederverwendung bereits entwickelter Systeme geschehen. Es wird zwischen Eigenentwicklung durch das Unternehmen und Fremdentwicklung durch Software-Hersteller unterschieden [Ohle98, 31]. Für die Realisierung individueller Lösungen müssen ausreichend Software-Entwicklungs-Kapazitäten sowie entsprechendes Know-How verfügbar sein [Fer+96b, 42]. Aus Kostengründen kommt es jedoch immer häufiger zu *Offshoring*, bei dem die Entwicklung in Länder mit kostengünstigeren Entwicklungsdienstleistungen wie Indien oder China vergeben wird.

[4] Diese bezieht sich auf nicht-funktionale Anforderungen. Siehe Kapitel 3.4.

Vor dem Hintergrund der Stärken von Standardsoftware bezüglich der Faktoren Zeit, Kosten-(sicherheit) und Qualität (s.o.), wird die Entwicklung von Individualsoftware nur unter bestimmten Bedingungen vorgezogen [Ambe99, 13]:

- Das zu lösende Problem ist zu speziell, so dass keine absehbare Nachfrage mehrerer Anwender besteht oder erkennbar ist.
- Standardsoftware-Produkt ist überhaupt nicht verfügbar. Zum Teil soll ein solches bewusst nicht eingesetzt werden und/oder der Anpassungsaufwand ist zu groß.
- Aus Sicherheitsgründen soll die Kontrolle über die Software vollständig sein.

Bei der Beurteilung, ob sich die Entwicklung einer individuellen Software-Lösung rechnet, sind deren Stärken und Schwächen zu beachten.

Stärken Da Individualsoftware eine Maßanfertigung für ein bestimmtes Einsatzgebiet darstellt, ist bei gegebener Qualität der Lösung die Unterstützung der betrieblichen Aufgabe optimal. Außerdem kann der Kunde größeren Einfluss auf das Ergebnis nehmen, was die Akzeptanz deutlich stärken kann.

Schwächen Der Aufwand für eine individuelle Lösung ist oftmals sehr hoch und kann im Allgemeinen nur sehr schwer geschätzt werden. Der Aufwand verursacht hohe sowie schlecht abschätzbare Kosten und die Qualitätssicherung gestaltet sich als sehr schwierig, da die Fehler häufiger erst beim Einsatz auftreten.

6.1.3 Kombination von Individual- und Standardsoftware

Ist die Programmierung nicht auf Wiederverwendung ausgelegt, so entstehen bei der Entwicklung Unikate. Der Einsatz eines Frameworks und der generativen Software-Architektur kann aber eine Software-Systemfamilie von gleichartigen Anwendungen ermöglichen [StVö05, 26]. Der Übergang zwischen Standard- und Individualsoftware ist fließend. So kann für die Herstellung von Individualsoftware auf Standardsoftware aufgebaut werden (z. B. bei der Verwendung höherer Systemplattformen und Basismaschinen oder auch durch den Einsatz von Standardsoftware-Komponenten). Für das in der vorliegenden Arbeit zu lösende Problem wird

versucht, die Vorteile von Individual- und Standardsoftware zu kombinieren, ohne gleichzeitig die jeweiligen Nachteile in Kauf nehmen zu müssen. Die Möglichkeit hierzu bietet Wiederverwendung, sowohl von Design-Entscheidungen (vgl. [Gam+95]) als auch von Programmcode. Frameworks bieten für diese Art der Entwicklung ideale Voraussetzungen. Eine detaillierte Beschreibung von Frameworks und deren Einsatz findet sich in Kapitel 6.7.
SCHRYEN beschreibt, dass durch Wiederverwendung, vor allem bereits vorhandener Komponenten, die Grenzen zwischen Standard- und Individualsoftware zum Teil aufgehoben werden können [Schr01], was die Kombination der Vorteile beider Ansätze ermöglicht.

6.2 Wiederverwendung

In allen Bereichen stellt die **Wiederverwendung** ein Vorgehen zur Problemlösung dar. Auch in der Software-Entwicklung spielt sie seit langem eine wichtige Rolle[5]. Unter Wiederverwendung von Software wird das erneute Anwenden von Artefakten und Wissen bei der Entwicklung eines neuen Software-Systems verstanden, mit dem Ziel der Reduktion von Aufwand für Erstellung und Pflege des neuen Systems [BiPe89]. Die verschiedenen Aspekte der Wiederverwendung sind in Tabelle 6.1 dargestellt.

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, Bibliothek von Funktionen oder Routinen, Klassenbibliothek, Framework, Software-Komponente, Standardsoftware, Muster

Tabelle 6.1: Aspekte der Wiederverwendung (vgl. [Same97, 22])

Bei der software-technischen Wiederverwendung sind vor allem zwei Differenzierungskriterien von großer Bedeutung. Zum einen der **Typ** des Wiederverwendeten[6], zum anderen der

[5] BROOKS geht sogar soweit, dass er in der Wiederverwendung und im Massenmarkt von Software eine potenzielle Art von „Wunderwaffe“ (*silver bullet*) [Broo87]. Zum Thema Wiederverwendung siehe auch [Mil+95].
[6] Es kann ein Vorgehen oder ein Objekt wiederverwendet werden.

Modus der Wiederverwendung[7] [Schr01, 8]. In ihrer frühesten Form fand Wiederverwendung im Bereich der Software-Entwicklung fast ausschließlich in der Implementierungsphase durch Wiederverwendung von Code statt. Mit Beginn der 1980er Jahre wurde sie auch auf weitere Phasen des Software-Entwicklungsprozesses wie Analyse, Entwurf und Wartung ausgeweitet, vor allem durch die systematische Wiederverwendung von Domänenwissen, Architekturen und Entwurfsmuster [Schr01, 9].

Bei der Entwicklung betrieblicher Anwendungssysteme kann im Rahmen jeder der durchzuführenden Teilaufgaben Wiederverwendung stattfinden [Szyp02, 151ff]. Es wird Wiederverwendung bei der Spezifikation, beim Entwurf sowie bei der Implementierung von Anwendungssystemen unterschieden (vgl. u. a. [NATO94, 3ff], [Hamm99]). Studien zeigen, dass jeweils nur ein kleiner Teil des Quelltextes einer Anwendung wirklich anwendungsspezifisch ist (vgl. hierzu u. a. [Jone84], [Trac88], [Jing03]). Der anwendungsneutrale Teil kann also zwischen verschiedenen Anwendungen wiederverwendet werden[8].

Vorteile der Wiederverwendung

Die Vorteile der Wiederverwendung sind mannigfaltig[9]. Neben verbesserter Produktivität und niedrigeren Wartungskosten, kann auch die Interoperabilität verbessert und die Einarbeitungskosten gesenkt werden. Zudem ist eine Unterstützung des Entwicklungsprozesses in Form des *Rapid Prototyping* (vgl. Kapitel 3.5.2) möglich. [NATO94, 3-2]

Dimensionen der Wiederverwendung

Wiederverwendung hat verschiedene Dimensionen, die die Wiederverwendbarkeit der einzelnen Elemente beeinflussen [NATO94, 3-4]:

- **Kompositorische vs. generative Ansätze**
  Kompositorische Ansätze unterstützen die *bottom-up*-Entwicklung von Software-Systemen auf der Grundlage von Bibliotheken und Komponenten niedrigerer Ebenen. Generative Ansätze hingegen sind spezifisch für Anwendungsdomänen, mit dem Ziel, ein neues

[7] Es kann z. B. geplante und spontane Wiederverwendung unterschieden werden.
[8] Zur Trennung von anwendungsspezifischem und -neutralem Teil einer Anwendung, siehe auch das objektorientierte Architekturmodell von SOM in Kapitel 3.3.4.
[9] Beispiele erfolgreicher Wiederverwendung in der Software-Industrie finden sich in [NATO94].

System direkt aus der entsprechenden Spezifikation seiner Parameter automatisch erzeugen zu können.

- **Wiederverwendung in kleinem vs. in großem Umfang**
  Der Umfang der wiederverwendeten Elemente kann von einem einzelnen Algorithmus bis hin zu ganzen Subsystemen reichen. Der Nutzen der Wiederverwendung im großen Umfang steht der geringeren Wiederverwendbarkeit entgegen.

- **Unveränderte vs. modifizierte Wiederverwendung**
  Elemente können entweder wiederverwendet werden *as is* oder sie bedürfen der Anpassung durch Modifikation. Modifizierbarkeit ist ein wichtiges Kriterium der Software-Wiederverwendung, da sie die Wiederverwendbarkeit erhöht.

- **Generik vs. Performanz**
  Der häufig auftretende *trade-off* zwischen generischer Anwendbarkeit bzw. Flexibilität und Performanz kann zum Teil durch Richtlinien für den Nutzer - also Entwickler - vermindert werden.

Hindernisse der Wiederverwendung

In der Praxis gibt es mehrere Hindernisse für Wiederverwendung, die technischer, betriebswirtschaftlicher oder kulturell-psychologischer Art sind [Reif01, 454]. Ein typisches Beispiel ist das so genannte „not-invented-here“-Syndrom. Demzufolge trauen viele Entwickler nur selbstentwickelten Komponenten, keinen fremden[10].
Ein Anwendungssystem kann nur dann sinnvoll wiederverwendet werden[11], wenn es sich im Einklang mit den Geschäftsprozessen des betrieblichen Informationssystems der Unternehmung befindet [Raue96, 4]. Durch die Verwendung der SOM-Methodik wird diese Konsistenz besonders unterstützt. Die nachfolgenden Kapitel stellen grundlegende software-technische Konzepte der Unterstützung von Wiederverwendung vor[12].

[10] Ausführlicher zu den Hindernissen der Wiederverwendung und wie man mit ihnen umgeht siehe u. a. [Reif01]. Etwaige Hindernisse werden in der vorliegenden Arbeit nicht betrachtet
[11] Zur Konstruktion von Geschäftsprozessmodellen mit dem Ziel der verbesserten Wiederverwendbarkeit siehe u. a. [Raue96].
[12] Für fachliche Referenzmodelle mit SOM siehe u. a. [Rüff99].

6.3 Objektorientierung und objektorientierte Software-Technik

Objektorientierte Ansätze nehmen mittlerweile nicht nur einen festen Platz in der Programmierung ein, sondern auch bei Rechner- und Systemarchitekturen, Software-Architekturen, Datenbanksystemen, grafischen Nutzerschnittstellen, im Software-Design sowie bei Modellierungsansätzen [Booc94, 54ff], [FeSi06, 184].
Im Folgenden wird das *objektorientierte Paradigma* kurz charakterisiert. Die software-technische Realisierung von objektorientierten Anwendungssystemen mittels so genannter *Frameworks* (Rahmenwerke) wird in Kapitel 7 ausführlich beschrieben.

6.3.1 Objektorientierter Ansatz

Der objektorientierte Ansatz ist nicht beschränkt auf die Programmierung. Er findet sich auch in den oben erwähnten Bereichen Rechner- und Systemarchitekturen, Software-Architekturen, Datenbanksysteme, grafische Nutzerschnittstellen, Software-Design sowie Modellierungsansätze (siehe u. a. [Kil+93, 5-12], [Booc94, 54ff], [FeSi06, 184]). Allgemein beruht dieser Ansatz auf folgenden Prinzipien [Booc94]:

Abstraktion : Objekte modellieren Entitäten oder Sachverhalte der realen Welt. Mit dem Ziel der Komplexitätsreduktion abstrahieren sie von Eigenschaften dieser Entitäten und Sachverhalte, die für das zu konstruierende System unwichtig sind.

Kapselung (information hiding): Der Begriff des *Information Hiding* wurde von PARNAS geprägt [Parn01a]. Er bezeichnet das Verbergen der Implementierung bzw. der Innensicht von Objekten. Ein Zugriff auf die Funktionalität des Objektes erfolgt lediglich über die öffentlich zugängliche Außensicht. Da die Realisierung des Verhaltens verborgen bleibt, kapseln die Objekte ihre interne Struktur und ihr Verhalten.

Modularisierung : Das zu konstruierende System wird anhand von Zerlegungskriterien in Teilsysteme zerlegt. Diese Kriterien können sich an Zielen und Aufgaben von Objekten orientieren.

Hierarchiebildung : Optimierung von Funktions- und Datenredundanzen sind Integrationsziele. Ihrer Kontrolle und der Reduktion von Komplexität dienen Aggregations- und Generalisierungsbeziehungen.

Der Arbeit liegt mit dem SOM-Ansatz (vgl. Kapitel 5.3) ein geschäftsprozess- und objektorientierter Modellierungsansatz zu Grunde. Die zu entwickelnde Software-Architektur (vgl. Kapitel 8) und die zur Realisierung verwendete Programmiersprache *Java* (vgl. Kapitel 9.1.1) folgen ebenfalls dem objektorientierten Ansatz.

6.3.2 Objektorientierte Programmierung (OOP)

BOOCH definiert **objektorientierte Programmierung** folgendermaßen [Booc94, 57]:

> „Objektorientierte Programmierung ist eine Implementierungsmethode, bei der Programme als kooperierende Ansammlungen von Objekten angeordnet sind. Jedes dieser Objekte stellt eine Instanz einer Klasse dar, und alle Klassen sind Elemente einer Klassenhierarchie, die durch Vererbungsbeziehungen gekennzeichnet sind."

Die Konzepte der objektorientierten Programmierung sind [Jaco93, 85]:

- Datenkapselung in Objekten
- Konzept von Instanz und Klassen
- Vererbung
- Polymorphismus

6.3.3 Objektorientiertes Design (OOD)

Für **objektorientiertes Design** findet sich in der Literatur folgende Definition [Booc94, 58]:

> „Objektorientiertes Design ist eine Designmethode, die den Prozeß der objektorientierten Zerlegung beinhaltet sowie eine Notation für die Beschreibung der logischen und physikalischen wie auch statischen und dynamischen Modelle des betrachteten Systems."

6.3.4 Objektorientierte Analyse (OOA)

Objektorientierte Analyse kommt in den frühen Phasen des Software-Entwicklungsprozesses zum Einsatz und legt das größte Gewicht auf die Erzeugung von Modellen der realen Welt. Bei der Analyse handelt es sich um den Konstruktionsprozess des fachlichen Modells. Sie wird bei BOOCH folgendermaßen definiert [Booc94, 59]:

> „Die objektorientierte Analyse ist eine Analysemethode, die die Anforderungen aus der Perspektive der Klassen und Objekte, die sich im Vokabular des Problembereichs finden, betrachtet."

6.4 Aspektorientierung

Ein Software-System stellt die Realisierung einer Reihe von *Concerns* dar. Unter einem *Concern* wird hier eine spezifische Anforderung verstanden, die beachtet werden muss, um dem globalen Ziel des Systems zu genügen [Ladd03, 7]. Es werden zwei Arten von Belangen unterschieden: *Core Concerns* einerseits und *Crosscutting Concerns* andererseits.

Core Concerns können unter anderem mit objektorientierten Ansätzen sauber in Module gepackt werden. *Crosscutting Concerns*[13] verhalten sich orthogonal zu den *Core Concerns*, da sie in mehreren Modulen benötigt werden und in der Regel über diese verteilt sind[14]. Systeme manifestieren sich in mehr als einer Dimension, wodurch Komponenten und Aspekte unterschieden werden. Die Begriff **Aspekt** wurde von KICZALES ET AL. geprägt. Er bezeichnet Eigenschaften eines Systems, die nicht notwendigerweise eingereiht werden können mit dessen funktionalen Komponenten, sondern diese vielmehr quer schneiden [Kic+97]. Die **Aspektorientierung** beschäftigt sich mit mehrdimensionalen Lösungen für das Problem der verstreuten Aspekt-Implementierungen.

Separation of Concerns [Dijk76] bezeichnet das Vorgehen, jeweils nur den Teil der Software zu identifizieren, zu kapseln und zu manipulieren, der für ein bestimmtes Konzept, ein bestimmtes

[13] *Crosscutting Concerns* sind nicht zwingend gleich zu setzen mit nicht-funktionalen Anforderungen. Für eine ausführliche Diskussion der Unterschiede und Gemeinsamkeiten siehe [Bono04, 18-21] sowie die dort zitierte Literatur.

[14] Die Tatsache, ob ein *Concern* als quer schneidend (*crosscutting*) zu bezeichnen ist, steht in Relation zu einer bestimmten Dekomposition [Elr+01]. Bei [OsTa01] wird dies auch als *Tyrannei der dominanten Dekomposition* bezeichnet.

Ziel oder einen bestimmten Zweck von Relevanz ist [OsTa01]. Eine saubere Trennung der *Concerns* (Modularisierung) ist für die Kernfunktionalität möglich. Sobald aber Querschnittsfunktionalität hinzu kommt, wird diese über das System verstreut (*scattered*) und bei der Implementierung innerhalb der *Core Concerns* mit anderen Funktionen „verknotet“ (*tangled*). Diese Nachteile der eindimensionalen Lösung [Ladd03, 10ff] sind in Abbildung 6.1 dargestellt. So ist die Prüfung der Autorisierung a) über mehrere Module verteilt (*scattering*) und b) implementiert das dargestellte Modul nicht nur die Geschäftslogik, sondern auch diverse Module mit Querschnittsfunktionalität (*tangling*). Mit den Auswirkungen schlechter Verfolgbarkeit, geringerer Produktivität, weniger Code-Wiederverwendung sowie schwierigerer Weiterentwicklung [Ladd03, 18f].

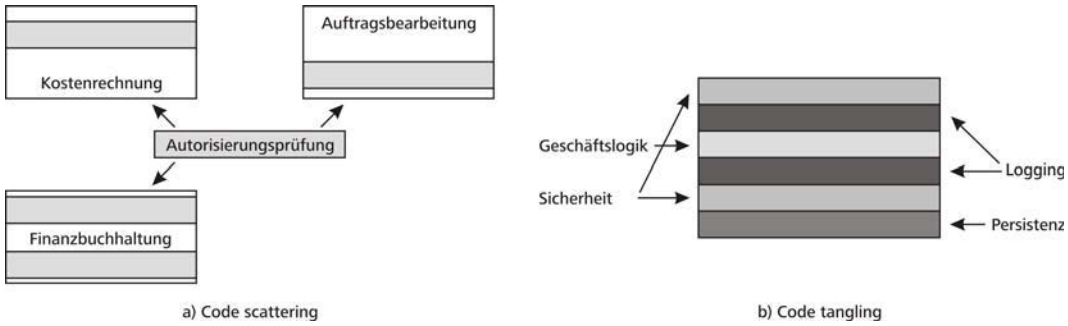

Abbildung 6.1: Code scattering und tangling bei crosscutting Concerns [Ladd03, 16f]

Aspektorientierung begegnet diesen nachteiligen Folgen mit einem mehrdimensionalen Ansatz. Vor allem die **aspektorientierte Programmierung** mit Sprachen wie AspectJ (vgl. [Ladd03]) baut auf objektorientierten Sprachen (Java) und Konzepten auf. Es werden neue Konzepte eingeführt, um auch quer schneidende Belange modularisieren zu können [Ladd03, 19], die Kernfunktionalität wird weiterhin mit der gewählten Methodik z. B. objektorientiert implementiert. Die aspektorientierte Entwicklung erfolgt dabei in drei Schritten [Ladd03, 21f]:

1. **aspektbezogene Abgrenzung** (*decomposition*)
   Im ersten Schritt werden *Core Concerns* (z. B. Geschäftslogik) sowie *Crosscutting Concerns* (z. B. Logging, Persistenz) identifiziert und von einander getrennt.

2. **Implementierung des *Concern***
   Jeder *Concern* wird mit konventionellen (objektorientierten) Methoden unabhängig implementiert (z. B. Modul Geschäftslogik, Modul Logging, Modul Persistenz).

3. **aspektbezogene Integration** (*recomposition*)

   Die Regeln der Integration (Aspekte) werden aufgestellt und das System diesen entsprechend auf der Implementierungsebene zusammengeführt. Beim Einsatz **aspektorientierter Programmierung** (AOP) geschieht das, indem die Aspekte zum Zeitpunkt des Kompilierens durch einen so genannten *weaver* mit den *Core Concerns* verwoben werden (*weaving*)[15].

Um Aspekte zu implementieren können a) Bibliotheken eingesetzt, b) domänenspezifische Aspektsprachen verwendet oder c) Spracherweiterung für die Unterstützung von Aspekten entwickelt werden [Czar98, 225]. Aspekte können statisch[16] oder dynamisch[17] mit dem Code der Komponenten verwoben werden [Con+00, 4], wobei die statische Methode klare Vorteile hinsichtlich der Performanz bietet.

FAYAD ET AL. schlagen einen framework-basierten Ansatz vor, bei dem eine Art Aspekt-Moderator (`AspectModerator`) vor Aufruf einer Klasse überprüft, welche Aspekte in diesem Zusammenhang auszuführen sind und entsprechend durch ihn ausgeführt werden [Con+00]. Dabei kennen sich Aspekt und Komponente zur Entwicklungszeit nicht, erst zur Laufzeit erfahren sie - wenn nötig - voneinander [Con+00, 7].

Wird das Anwendungssystem aus der modellgetriebenen Software-Entwicklung abgeleitet, so können schon auf dieser Ebene die Aspekte beachtet werden (siehe [StVö05, 160ff] und [Völt05, 8]).

6.5 Komponentenorientierung

Ein besonderes Bestreben bei der Entwicklung betrieblicher Anwendungssysteme ist das Zusammenfügen wiederverwendbarer Software-Komponenten. MCILROY fordert schon früh einen Massenmarkt für Software-Komponenten, um so der Software-Krise begegnen zu können [McIl68]. Durch diese Art der Wiederverwendung können Vorteile von Standard- und Individualsoftware verbunden werden [Turo99, 132]. TUROWSKI definiert eine **Komponente** wie folgt [Turo99, 134]:

[15] Siehe z. B. Einsatz von AspectJ [Ladd03].

[16] Aspekt-Code und Komponenten-Code werden noch zur Entwicklungszeit verwoben. Entweder zu 'vermischtem' Quellcode (vor dem Kompilieren) oder 'vermischtem' ausführbarem Code (beim Kompilieren).

[17] Der Code von Aspekten und Komponenten wird erst zur Laufzeit miteinander verwoben.

> „Eine *Komponente* ist ein wiederverwendbarer, abgeschlossener und vermarktbarer Softwarebaustein, der Dienste über eine wohldefinierte Schnittstelle zur Verfügung stellt und in zur Zeit der Entwicklung unvorhersehbaren Kombinationen mit anderen Komponenten einsetzbar ist."

Sind Komponenten Bestandteil betrieblicher Anwendungssysteme, wird der Begriff noch enger als **Fachkomponente** gefasst. Eine Fachkomponente implementiert eine bestimmte Menge von Diensten einer betrieblichen Anwendungsdomäne [Turo99, 134]. Werden die Komponenten zu einem Anwendungssystem zusammengefasst, so ist deren Integration durch die Verfolgung von Integrationszielen von großer Bedeutung (vgl. Kapitel 4). Eine entsprechende Implementierungsplattform stellt das im Rahmen dieser Arbeit entwickelte Framework dar. Es handelt sich dabei um ein Architektur-Framework (vgl. [Whit02, 110–115], [Evan04, 74]), das als Komponentenplattform dient, auf der Komponenten sowohl neu entwickelt als auch bestehende wiederverwendet und zu Anwendungssystemen montiert werden können.
Auch wenn die technischen Voraussetzungen für die Entwicklung und Montage von standardisierten Fachkomponenten in den letzten Jahren deutlich besser geworden sind, so wird wohl dennoch kein „Massenmarkt" (s.o.) dafür entstehen. Grund dafür ist die zu hohe Änderungsgeschwindigkeit bzw. die notwendige Variantenvielfalt, die eine ganzheitliche Standardisierung der betrieblichen Anwendungsdomäne durch Fachkomponenten verhindern [Turo99, 135]. Dies führt zur Definition von Teil- bzw. Kernstandards, die durch standardkonforme Erweiterbarkeit angepasst werden können. Dieser Flexibilität kommt eine hohe Bedeutung zu. Für die Beschreibung der eigentlichen Nutzdaten durch Meta-Informationen eignet sich in diesem Zusammenhang besonders die *Extensible Markup Language* (XML)[18].

6.6 Entwurfsmuster

Um bei der Lösung eines Problems von bestehenden Erfahrungen und erfolgreich getroffenen Entwurfsentscheidungen profitieren zu können, verwendet man in vielen Bereichen Muster, die die Lösung des Problems in einem bestimmten Kontext beschreiben[19]. In Abhängigkeit von

[18] Für eine detailliertere Beschreibung von XML siehe Kapitel 9.1.2.
[19] Eines der bekanntesten Beispiele ist die Mustersprache für das Bauwesen des Architekten Christopher Alexander, die aus insgesamt 253 Mustern besteht [Ale+78].

der Granularität der Muster schlägt BUSCHMANN bezogen auf die Entwicklung von Software-Systemen folgende Kategorisierung der Muster vor [Bus+00, 12ff].

- **Architekturmuster**
  Ein Architekturmuster wie *Presentation-Abstraction-Control* (vgl. auch Kapitel 8.2.2.1) [Bus+00, 145ff] spiegelt das generelle Prinzip der Strukturierung eines Software-Systems wider.

- **Entwurfsmuster**
  Ein Entwurfsmuster dient zur Beschreibung eines Schemas, das zur Verfeinerung von Komponenten des Software-Systems oder deren Beziehungen dient. Das Entwurfsmuster *Observer* [Gam+95, 293ff] wird u. a. im Kontext des Architekturmusters *Model-View-Controller* (vgl. auch Kapitel 8.2.2.2) [Bus+00, 124ff] verwendet.

- **Idiom**
  Idiome sind in der Regel programmiersprachenspezifisch und beschreiben eine wiederverwendbare Lösung bei Implementierungsproblemen in einem bestimmten Kontext. Diese weisen die niedrigste Abstraktionsstufe auf.

Im Rahmen der software-technischen Spezifikation von Anwendungssystemen beschreiben **Entwurfsmuster** (*design patterns*) wiederverwendbare Lösungen für Entwurfsprobleme, die in einem bestimmten Kontext auftreten[20]. Sie ermöglichen die Wiederverwendung von Entwurfserfahrung. Sie werden bei GAMMA ET AL. orthogonal klassifiziert nach Zweck (was ein Entwurfsmuster tut) und Reichweite (bezieht sich ein Entwurfsmuster primär auf Klassen oder Objekte) [Gam+95, 10]. Obwohl Entwurfsmuster nicht auf objektorientierte Ansätze beschränkt sind, besteht eine enge Verbindung zwischen beiden Ansätzen durch objektorientierte Konzepte wie Vererbung oder abstrakte Klassen. Ein Entwurfsmuster wird durch vier Elemente charakterisiert [Gam+95, 3]:

- **Name:** Gibt den Namen des Musters an und identifiziert das Entwurfsproblem.

- **Problem:** Beschreibung, wann das Entwurfsmuster eingesetzt werden kann. Es werden das Problem selbst und der Kontext der Anwendung sowie eventuell Bedingungen für den Einsatz spezifiziert.

[20] Zu Entwurfsmustern vgl. vor allem [Gamm92] und [Gam+95]

- **Lösung:** Spezifikation der Elemente des Entwurfs, ihre Beziehungen untereinander sowie die Zusammenarbeit der einzelnen Elemente einschließlich ihrer jeweiligen Verantwortung. Es wird aber keine konkrete Implementierung beschrieben.
- **Konsequenzen:** Beschreibung der Konsequenzen und Zielkonflikte des Einsatzes, die für die Beurteilung der konkreten Verwendung des Entwurfsmusters von Bedeutung sind.

Will man ein bestimmtes Problem mit Entwurfsmustern lösen, muss man herausfinden, ob es ein entsprechendes Muster gibt, es dann finden und für den eigenen Kontext anwenden. Architekturmuster für Anwendungssysteme finden sich in [Fowl03].

6.7 Frameworks

Ein **Framework**[21] besteht aus einer Menge von Klassen und stellt einen abstrakten Lösungsentwurf für eine Familie verwandter Probleme[22] dar [JoFo88, 22]. Von Entwurfsmustern unterscheiden sich Frameworks vor allem hinsichtlich der Abstraktionsstufe der konkreten technischen Realisierung [Gam+95, 28]:

1. *Frameworks sind weniger abstrakt als Design Patterns.* Während Frameworks als Code vorliegen können, werden bei Entwurfsmustern höchstens Beispiele als Code realisiert.
2. *Frameworks sind größere architektonische Elemente als Design Patterns.* Ein typisches Framework umfasst mehrere Design Patterns. Das umgekehrte Verhältnis ist nicht möglich.
3. *Frameworks sind spezialisierter als Design Patterns.* Frameworks haben immer eine bestimmte Anwendungsdomäne.

Frameworks[23] dienen der Erfüllung einer bestimmten Aufgabe und umfassen eine Sammlung verschiedener, individueller Komponenten, die ein definiertes Kooperationsverhalten besitzen [Pree97, 7]. Einige dieser Komponenten sind austauschbar und dienen der möglichen Verfeinerung des Frameworks. Sie werden *hot spots* genannt. Bei Frameworks handelt es sich

[21] Zu Frameworks vgl. u. a. [John92], [John97], [Pree97].
[22] Zum Beispiel die Erstellung eines Dialogfenster-Systems oder die Entwicklung eines internet-basierten interaktiven betrieblichen Anwendungssystems.
[23] Frameworks werden in [Grif98, 114] als „Applikationsbaukästen“ bezeichnet.

um eine objektorientierte Wiederverwendungstechnik [John97, 10], die über die reine Code-Wiederverwendung hinausgeht. Es handelt sich, anders als bei Klassenbibliotheken von Bausteinklassen[24], nicht um eine Sammlung voneinander unabhängiger Klassen, die bei jeder Anwendungsentwicklung neu miteinander verknüpft werden müssen.

Dient ein Framework als Grundlage für eine vollständige Anwendung (*application*), so wird es als *Application Framework* bezeichnet. Der Einsatz von Frameworks ermöglicht dadurch bei der Anwendungsentwicklung durch Wiederverwendung[25] des Entwurfs[26] (*design reuse*) die gemeinsame Nutzung von Architektur [Gam+95, 27], die als eine der wichtigsten Arten der Wiederverwendung im Software-Engineering gilt [JoFo88, 24]. Die Wiederverwendung findet in zweifacher Hinsicht statt. Zum einen wird die Software-Architektur und die damit verbundenen Entwurfsentscheidungen für die Entwicklung des Anwendungssystems wiederverwendet. Zum anderen kann der Programm-Code selbst wiederverwendet werden[27], da das Framework abstrakte Klassen beinhaltet, die vom Entwickler erweitert, angepasst und wiederverwendet werden können (siehe Abbildung 6.2).

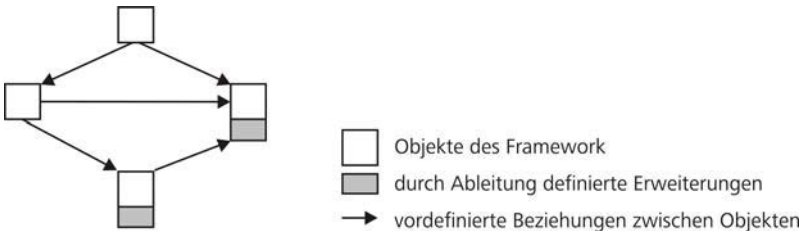

Abbildung 6.2: Wiederverwendung eines Frameworks [Gamm92, 12]

Das Komponentenparadigma geht davon aus, dass vollständige, in sich abgeschlossene und bereits vorgefertigte funktionale Einheiten (Komponenten) nur noch zusammengefügt werden müssen [Grif98, 115]. Im Unterschied dazu stellt ein Framework eine Art Schablone dar, die der zu entwickelnden Software eine Grundstruktur und Organisation vorgibt. Erst die Verfeinerung führt zu anwendbaren Endprodukten (vgl. auch Kapitel 7.1.). Ein Framework kann eine

[24] Bausteinklassen können ohne weitere Ableitung unabhängig von anderen Klassen wiederverwendet werden (vgl. [Gamm92, 11]. Beispiele hierfür sind Listen oder Hash-Tabellen, die der Verwaltung von Datenstrukturen dienen.

[25] Zur Wiederverwendung von Frameworks vgl. auch [Pree97].

[26] Die Art und Weise der Zerlegung des Systems in Komponenten ist der wichtigste Teil des Frameworks (vgl. [Deut89]). Die Aufteilung der Funktionalität auf Komponenten und die Wiederverwendung der internen Schnittstellen wird als zum Teil als wichtiger erachtet als die eigentliche Implementierung (vgl. [John97]).

[27] Die Reduktion des Aufwands der Quellcode-Programmierung für die Anpassung eines ausgereiften Framework im Vergleich zur korrespondierenden Neuentwicklung einer Anwendung wird in der Literatur zum Teil mit bis zu 80% angegeben (vgl. [Wei+89]).

ergänzende Infrastruktur für Komponenten darstellen. Dadurch entsteht ein anwendungsbezogenes Komponentenmodell [Grif98, 116].

Ursprünglich stammen Software-Frameworks aus dem Umfeld der grafischen Benutzungsoberflächen[28], werden aber seit Mitte der 1990er Jahre zunehmend auch im Umfeld betrieblicher Anwendungsentwicklung eingesetzt. Die Entwicklung betrieblicher Anwendungssysteme mit Frameworks stellt den Lösungsansatz der vorliegenden Arbeit für einen effizienteren Entwicklungsprozess dar. Dem Themenkomplex Framework ist deshalb ein eigenes Kapitel gewidmet (siehe Kapitel 7).

[28] Vgl. [John97, 12f]. Bekanntestes Beispiel ist das GUI-Framework von *Smalltalk-80*, das auf dem Entwurfsmuster *Model-View-Controller* (*MVC*) basiert (vgl. [KrPo88]).

7 Frameworks in der Systementwicklung

Ein **Framework** definiert und implementiert anwendungsneutrale Software-Komponenten sowie deren Kooperationsverhalten [Pree97, 7]. Im folgenden soll dargelegt werden, wie der Einsatz von Frameworks in den Phasen Software-Design und Realisierung sowohl den Umfang als auch die Komplexität der Systementwicklungsaufgabe verringern kann.

Frameworks liegt das Konzept der Wiederverwendung auf Entwurfs- sowie Code-Ebene zu Grunde [John97, 10f]. Ein Framework stellt eine Basismaschine dar, die für die Entwicklung entsprechende Flexibilität bietet (vgl. [FeSi06, 304]), in dem Sinne, dass verschiedene Anwendungen unter Verwendung desselben Framework auf Grund infrastruktureller, architektonischer oder fachlicher Gemeinsamkeiten realisiert werden können. Diese bezeichnet man als **Software-Systemfamilie** [Parn01b] (z. B. ein E-Business-System), innerhalb derer es verschiedene **Produktlinien** geben kann (vgl. [Bett05]).

In Abbildung 7.1 ist dargestellt, dass die Komponenten des Frameworks, das als Basismaschine dient, den Aufruf der jeweiligen Programmkomponenten auslösen.

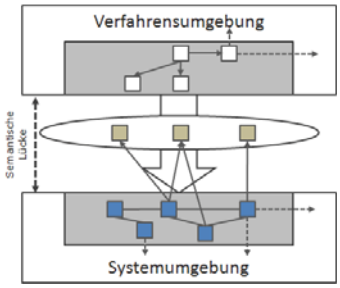

Abbildung 7.1: Programmerstellung mit Framework-Einsatz (nach [Fer+96b, 46])

Dieser Mechanismus - auch *Inversion of Control* genannt - ist eine der wichtigsten Eigenschaften von Frameworks, die es vor allem vom Einsatz von Klassenbibliotheken oder auch Komponenten unterscheidet. Er wird von JOHNSON wie folgt beschrieben [JoFo88, 25]:

„One important characteristic of a framework is that the methods defined by the

> user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application."

Je nach Anwendungsgebiet können verschiedene Arten von Frameworks unterschieden werden [FaSc97, 34]:

- *System Infrastructure Frameworks* dienen hauptsächlich dem internen Gebrauch (z. B. Betriebssysteme).
- *Middleware Integration Frameworks* wie *Object Request Broker (ORB) Frameworks* dienen der Integration verteilter Anwendungen.
- *Enterprise Application Frameworks* adressieren weitverbreitete Anwendungsdomänen mit Fokus auf betriebliche Geschäftsaktivitäten.

Der Lösungsansatz der Arbeit ist die Konzeption und Realisierung eines *Enterprise Application Frameworks*, das der Entwicklung betrieblicher Anwendungssysteme dient. Diese Art von Framework wird auch auch als *Business Framework* bezeichnet. SCHMITZER unterscheidet diesbezüglich drei Ansätze [Schm00, 11f]:

- Das Business Framework zur Konfiguration betrieblicher Funktionen und Prozesse auf Basis fertiger Anwendungskomponenten und Standard-Software, wie sie die *SAP AG* anbietet.
- Das Business Framework enthält als Rahmen und Werkzeug zur Entwicklung von Fachkomponenten Basiswissen des Anwendungsfeldes inklusive Anwendungslogik. Als Beispiel kann das *SanFrancisco Framework* der *IBM* gesehen werden [Car+00].
- Anbieter von *Component Integration Frameworks* konzentrieren sich auf technische Middleware-Standards für die Entwicklung von Bausteinen. Anbieter sind bspw. *Sun*[1]

[1] `http://www.sun.com/aboutsun/media/presskits/sunone`

mit dem *J2EE Framework* und *Microsoft*[2] mit seinem *.NET Framework* [Gras03]. Solche technischen Frameworks fokussieren zwar die Entwicklung betrieblicher Anwendungssysteme, doch geht die technische Abstraktion dabei nicht weit genug, um die semantische Lücke zu fachlichen Konzepten entscheidend reduzieren zu können.

Das hier entwickelte Framework soll als Rahmen und Werkzeug zur Entwicklung von Fachkomponenten dienen. Ganz generell verringert ein solches Framework durch Wiederverwendung seiner Komponenten den Umfang der Systementwicklungsaufgabe:

- Die Software-Architektur des Anwendungssystems muss nicht im Rahmen des Entwicklungsprozesses entworfen werden, da sie vom Framework definiert und implementiert wird.
- Das Kooperationsverhalten der Komponenten muss nicht durch das Programm abgebildet werden, sondern wird vom Framework implementiert. Dadurch wird eine Trennung der Geschäftslogik, wie sie anwendungsspezifisch umgesetzt wird, von der Komplexität des Software-Systems erreicht.
- Der Umfang der Programmerstellung sinkt, da anwendungsneutrale Funktionalität nicht während der Aufgabendurchführung entwickelt werden muss, sondern vom Framework zur Verfügung gestellt wird.

Für den Einsatz als Lösungsverfahren der Systementwicklungsaufgabe ist zudem die zu Grunde liegende fachliche Modellierung zu beachten. Das im Rahmen der Anforderungsanalyse und -definition abgeleitete fachliche Modell des Anwendungssystems (vgl. [FeSi06, 460ff]) bestimmt die Komplexität der zu realisierenden Nutzermaschine. Für den Entwurf eines Frameworks zur vollständigen Abbildung der Nutzermaschine bedeutet das, dass sich dieser an der Art und am Umfang der Komponenten der Nutzermaschine orientieren muss.
Zunächst wird der Framework-Entwicklungsprozess in Kapitel 7.1 vorgestellt. Der ökonomische Nutzen des Framework-Einsatzes ist Gegenstand von 7.2. Anschließend werden in Kapitel 7.3 und Kapitel 7.4 zwei fachlich motivierte Framework-Ansätze vorgestellt und auf ihre Eignung für die Entwicklung von Anwendungssystemen betrachtet, die gemäß SOM aus den

[2] `http://www.microsoft.com/net`

Geschäftsprozessen der Organisation abgeleitet wurden. Die Motivation eines eigenen Framework-Entwurfs wird in Kapitel 7.5 erläutert. Entwurf und Realisierung des Frameworks **moccabox** sind Gegenstand des dritten Teils der Arbeit.

7.1 Der Framework-Entwicklungsprozess

Im Gegensatz zu Klassenbibliotheken definiert ein Framework nicht nur die einzelnen Komponenten, sondern auch wie sie interagieren (siehe Abbildung 7.2). Ein Framework besteht aus festgelegten Teilen *frozen spots* und erweiterbaren Einschüben *hot spots*. Je nachdem, ob die Einschübe durch Überschreiben vorhandener Methoden oder Implementieren spezifizierter Schnittstellen realisiert werden, spricht man von *Gerüsten* (*White-Box-Frameworks*) bzw. *Baukästen* (*Black-Box-Frameworks*) [Schm00, 9]. Der Lebenszyklus eines Frameworks entspricht meist einer Evolution, so wird ein Framework zunächst als *white-box* implementiert und erst mit zunehmender Nutzung und Weiterentwicklung zum *black-box* Framework (vgl. [Pree97]).

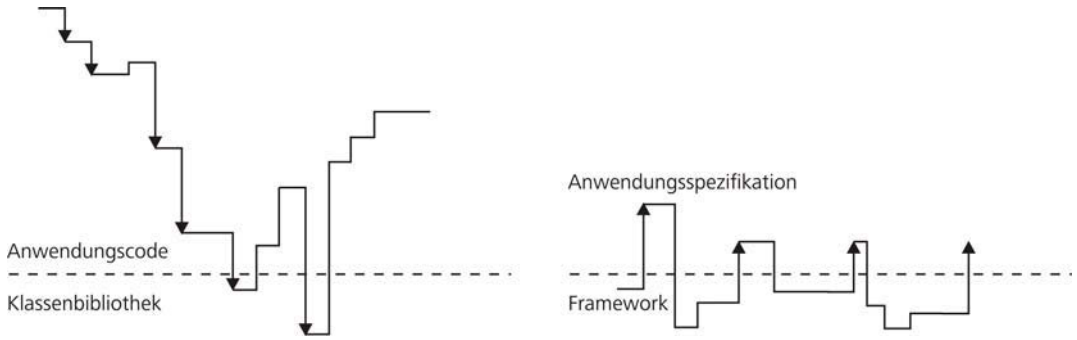

Abbildung 7.2: Aufrufbeziehungen Klassenbibliothek und Framework (nach [Zeid99, 5])

Bei einem Framework handelt es sich um eine generische Applikation, mit dem Zweck der Erzeugung verschiedener Anwendungen einer bestimmten Anwendungsdomäne (z. B. internetbasierte betriebliche Anwendungssysteme) [Schm97, 48]. Aufgrund der dafür notwendigen Flexibilität und Variabilität stellt der Entwurf eines Frameworks im Vergleich zum Entwurf einer konkreten Applikation eine entsprechend komplexere Aufgabe dar (vgl. [Nie+92, 164], [Car+00, 1]). Der Entwicklungsprozess kann sich daher wesentlich von dem konventioneller

Software-Systeme unterscheiden[3]. Die Entwicklung findet in der Regel nicht streng sequentiell statt, sondern wird *bottom up* in mehreren iterativen Schritten durchlaufen (vgl. u. a. [WiJo90], [FaSc97], [Schm97]). Zudem handelt es sich um einen evolutorischen Prozess, der nicht beendet ist, sobald das Framework zur Entwicklung von Applikationen eingesetzt wird. Gesammelte Erfahrungen und neue Anforderungen aus der Anwendung fließen in den weiteren Entwicklungsprozess des Frameworks ein (vgl. u. a. [RoJo97a]). Der Framework-Entwicklungsprozess *bottom-up* kann in Übereinstimmung mit [Pree97], [RoJo97b] wie in Abbildung 7.3 dargestellt werden.

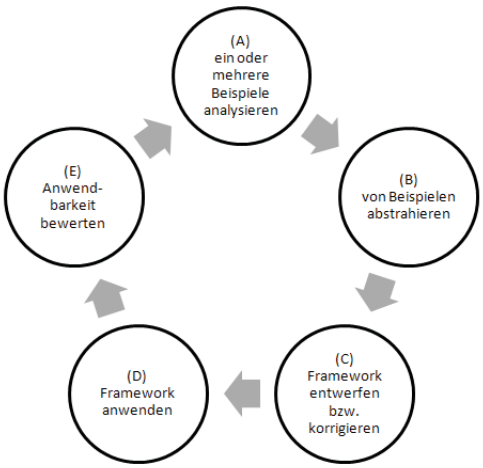

Abbildung 7.3: Framework-Entwicklungsprozess (nach [RoJo97b])

Die **Variabilität** eines Frameworks wird durch seine variablen Anteile - so genannte *hot spots* - bestimmt. Bei der Entwicklung von Frameworks wird zunächst von konkreten Anwendungsfällen abstrahiert [Nie+92, 164]. Frameworks können nach White-Box und Black-Box Frameworks unterschieden werden (vgl. [Schm96]).

7.1.1 White-Box Frameworks

White-Box-Frameworks zeichnen sich im objektorientierten Kontext dadurch aus, dass sie aus einer Menge von **abstrakten Klassen** bestehen, die unvollständig spezifiziert sind[4] (vgl. [Pree97, 19f]). Eine abstrakte Klasse besitzt mindestens eine nicht implementierte Methode

[3] ROBERTS & JOHNSON stellen in [John97] eine Mustersprache (*pattern language*) vor, die der Entwicklung von Frameworks dient. Sie enthält Entwurfsmuster von der ersten Ableitung des Frameworks aus drei Beispiel-Anwendungen über den White-Box-Entwurf bis hin zum visuellen Baukasten.

[4] Zu abstrakten Klassen vgl. [Pree97, 26f]. Das Auffinden abstrakter Klassen beschreiben [JoFo88].

[John97, 11]. Solche Methoden werden als **abstrakte Methoden** bezeichnet und sind bei der Erweiterung einer abstrakten Klasse zu einer konkreten Klasse vom Entwickler entsprechend zu implementieren. Damit das Framework eingesetzt werden kann, müssen vom Entwickler die abstrakten Klassen erweitert und deren abstrakte Methoden überschrieben werden. Auf diese Weise wird das Framework für den entsprechenden Einsatz parametrisiert. Für die Anpassung des Frameworks benötigt der Entwickler mithin eine gewisse Kenntnis[5] des Entwurfs und der Implementierung des White-Box-Framework. Wann die Methoden aufgerufen und wie eventuelle Rückgabewerte verarbeitet werden bleibt in der Kontrolle des Frameworks.
White-Box-Frameworks bauen sehr stark auf dem Prinzip der Vererbung auf, mit entsprechenden Einschränkungen bei der Verwendung [RoJo97a]:

- sehr enge Kopplung der Komponenten, die durch Vererbung erweitert werden
- der Entwickler kann Änderungen in der erweiterten Komponente vornehmen, die vom Framework-Entwickler nicht vorgesehen waren
- Erweiterung erfolgt durch Programmierung neuer Klassen
- Vererbung ist statisch und kann zur Laufzeit nicht leicht geändert werden

Dennoch bietet der Einsatz von White-Box-Frameworks vor allem den Vorteil großer Flexibilität. Vor allem bei der Neuentwicklung eines Frameworks, wenn die Anwendungsdomäne und notwendige Funktionalität noch nicht abschließend bekannt sind. Beim Systementwurf kommen Entwurfsmuster wie *Template Method* und *Factory* zum Einsatz (vgl. [Gam+95]), um den Umfang des wiederverwendbaren Codes zu erhöhen.

7.1.2 Black-Box-Frameworks

Anders als bei den eben beschriebenen White-Box-Frameworks basiert das Anpassungskonzept von **Black-Box-Frameworks** (vgl. [Pree97, 20f]) nicht auf unvollständig spezifizierten Klassen. Vielmehr zeichnen sie sich durch eine Menge fertiger Komponenten aus. Die Anpassung des Frameworks erfolgt nicht über eine vervollständigende Programmierung der Komponenten

[5] Gleichbedeutend mit Verständnis.

oder Klassen, sondern durch die sinnvolle und zielgerichtete Komposition (*plugging*) der Komponenten einer Komponenten-Bibliothek [JoFo88, 25]. Die innere Struktur des Frameworks und seiner Komponenten bleibt dem Entwickler folglich verborgen (*black box*).
Das Prinzip der polymorphen Komposition [RoJo97a], das Black-Box-Frameworks zu Grunde liegt, bietet gegenüber dem Prinzip der Vererbung gewisse Vorteile (vgl. [RoJo97a]), mit der Einschränkung, dass der Entwickler sehr genau wissen muss, was sich durch die Komposition ändert:

- Komposition unterstützt auf sehr mächtige Weise Wiederverwendung
- es müssen keine neuen Klassen geschrieben werden
- Komposition kann zur Laufzeit geändert werden

Zwar sind Black-Box-Komponenten leichter zu verstehen, da die internen Zusammenhänge verborgen bleiben. Doch sie sind weniger flexibel als die White-Box-Erweiterung durch Vererbung. Daher wird die polymorphe Komposition auf Basis einer Komponenten-Bibliothek vor allem dort zum Einsatz kommen, wo die Anwendungsdomäne sehr gut bekannt ist.
Die wenigsten Frameworks sind reine White-Box-Frameworks oder Black-Box-Frameworks, sondern enthalten sowohl abstrakte Klassen wie auch fertige Elemente der Komposition. Die Entwicklung eines Frameworks erfolgt zumeist als White-Box-Framework durch die Ableitung aus einer bestimmten Applikation. Anwendungsneutrale Funktionalität wird als Code wiederverwendet. Der anwendungsspezifische Teil wird durch abstrakte Klassen repräsentiert und bei der Erweiterung zur Applikation vom Entwickler programmiert. Die Identifikation und Spezifikation der entsprechenden *hot spots* ist die entscheidende Aufgabe bei der Anwendungsentwicklung. PREE schlägt hierfür die Verwendung so genannter *hot-spot*-Karten vor [Pree97, 64ff]. Mit zunehmendem Reifegrad durch Wiederverwendung und iterative Erweiterung des Frameworks wird aus der White-Box-Schnittstelle eine Black-Box-Schnittstelle (vgl. [JoFo88]).
Es bleibt festzuhalten, dass das Vorgehen *bottom up* zum einen den Nachteil hat, sehr aufwendig zu sein. Zum anderen werden bei der Ableitung aus Referenz-Anwendungen lediglich software-technische Übereinstimmungen festgestellt und abstrahiert. Die Anwendungsdomäne des hier zu entwickelnden Frameworks ist die Entwicklung betrieblicher Anwendungssysteme. Die dabei zu realisierenden Fachkonzepte leiten sich aus einem fachlichen Modell des

Anwendungssystems ab, das hier entsprechend der SOM-Methodik spezifiziert wird. Dadurch ist ein initialer Entwicklungsprozess des Frameworks *top down* möglich, wie in Abbildung 7.4 dargestellt. Der Entwurf muss in weiteren Iterationen verfeinert und verbessert werden, orientiert sich aber stets an den Konzepten des fachlichen Modells.

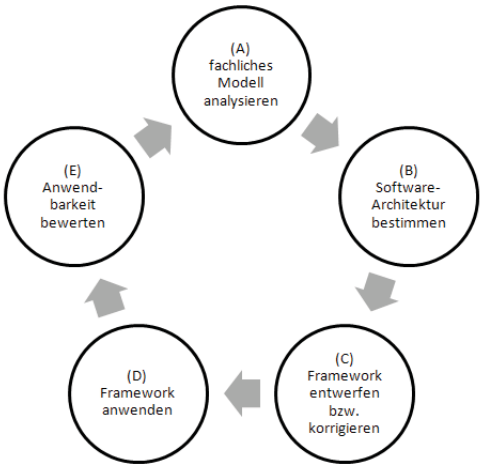

Abbildung 7.4: Framework-Entwicklungsprozess nach fachlichem Modell

7.2 Wirtschaftliche Aspekte des Framework-Einsatzes

Die Framework-Entwicklung ist teuer [RoJo97a]. Deshalb sollte sie nur dann vorgenommen werden, wenn absehbar ist, dass eine Vielzahl von Applikationen aus derselben Problemdomäne damit realisiert werden können. Der primäre Nutzen des Einsatzes von objektorientierten Applikations-Frameworks leitet sich aus der dem Nutzer gebotenen Modularität, Wiederverwendbarkeit, Erweiterbarkeit und *Inversion of Control* ab [FaSc97, 32].

Frameworks und Wiederverwendung

Der Grad der Wiederverwendung bei Frameworks ist enorm hoch, da nicht nur Klassenbibliotheken definiert werden, sondern deren Zusammenwirken durch das Framework bestimmt wird (vgl. [John92], [John97], [Pree97]). Dieses kann bei jeder Anwendung erweitert und verfeinert werden. Bei Frameworks liegt idealerweise eine Wiederverwendung sowohl des Entwurfs als auch der Implementierung vor [Schr01, 30]. Das bekannteste Business-Framework ist das von der *IBM* entwickelte *San Francisco Framework* [Car$^+$00], bei dem laut IBM bereits 40% der

zu entwickelnden Anwendung zur Verfügung gestellt wird. Eine der wichtigsten Eigenschaften von Frameworks ist, dass bei einer Entwicklung von Anwendungen auf Basis eines Frameworks nicht nur den Quellcode wiederverwendet, sondern auch den vordefinierten Architekturentwurf [Pree97, 8]. Nach Art der Anpassung werden White- und Black-Box-Frameworks unterschieden [Pree97, 19ff] (siehe oben). Die Klassen eines Frameworks, die den Entwurf seiner Struktur bestimmen, sind in der Regel abstrakt [John97, 13]. Konkrete Unterklassen werden zumeist in Form einer Komponentenbibliothek mitgeliefert. Die Komponentenbibliothek ist nicht zwingend Bestandteil des Frameworks. Sie kann mit zunehmendem Reifegrad wachsen.

Frameworks und Komponenten

Mit Blick auf die steigende Komplexität geforderter Anwendungssysteme verlangte MCILROY bereits in den 1960er Jahren nach einem Markt für Software-Komponenten [McIl68]. Frameworks und Komponenten (siehe auch Kapitel 6.5) ergänzen sich sehr gut, da Frameworks zu Komponenten (*black box*) einen wiederverwendbaren Kontext liefern (siehe auch [Grif98]). Der Trend zur Komponentenbildung und Dienstnutzung hält an [Alk+03]. Dabei sollen neue Software-Systeme durch das Zusammenfügen existierender und bewährter Software-Bauteile (Komponenten) erstellt werden [Fran99]. Komponenten werden als eine höhere Abstraktionsstufe zur Objektorientierung gesehen [Same97, 68] (siehe auch [Jaco93], [Grif98], [Szyp02]). SAMETINGER definiert eine Komponente wie folgt [Same97, 68]:

> „Reusable software components are self-contained, clearly identifiable artifacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status."

Durch den Einsatz von Komponententechnologie können Vorteile der Standardsoftware mit denen individuell erstellter Systeme kombiniert werden können [Szyp02, 2ff]. Gleichzeitig können Potenziale hinsichtlich Kostenreduktion und Produktivitätssteigerung realisiert werden, die hauptsächlich aus der ermöglichten Wiederverwendung entstehen [Alk+03, 230ff].

Automatisierte Entwicklung mit Frameworks

STAHL und VÖLTER weisen darauf hin, dass Frameworks eine ideale Hilfe für die Implementierung von Anwendungen darstellen, die mittels modellgetriebener Software-Entwicklung

realisiert werden [StVö05, 133]. Ein Framework kann als Zielplattform für die Abbildung eines plattformspezifischen Modells durch (automatische) Transformation dienen (siehe Kapitel 5.4). Voraussetzung dafür ist, dass die Abbildung des zu Grunde liegenden fachlichen Modells vollständig ist. Dies ermöglicht eine stärkere Automatisierung der Durchführung der Systementwicklungsaufgabe.

Frameworks im Software-Entwicklungsprozess

In vielen Software-Projekten liegt der Schwerpunkt auf der technischen Infrastruktur und technologisch getriebenen Entscheidungen [Eic+04]. Gleichzeitig steigt mit wachsender Größe und Komplexität von Anwendungssystemen die Wichtigkeit des Entwurfs und der Spezifikation der Struktur des Gesamtsystems im Vergleich zur Auswahl von Algorithmen und reinen Datenstrukturen [GaSh93], [ShGa96].

Ein geeignetes Framework kann im Software-Entwicklungsprojekt den software-technischen Lösungsraum begrenzen und zu treffende Entscheidungen beschleunigen. Hinzu kommt bei einem fachlich abgeleiteten Framework die Verlagerung der Entscheidungen auf die fachliche Beschreibungsebene, weg von technologie-getriebenen Kriterien.

Sofern ein Framework für die Umsetzung der zu Grunde liegenden Fragestellung geeignet ist, kann durch dessen Einsatz bei der Entwicklung betrieblicher Anwendungssysteme ein hoher Grad an Code- und Entwurfswiederverwendung erreicht werden. Der Entwickler muss dadurch ausschließlich die *hot spots* implementieren, da die Struktur und der Kontrollfluss der Anwendung bereits festgelegt sind. Somit sind sowohl die Software-Architektur als auch die Implementierung der Kontrollkomponenten vorhanden.

In der Konsequenz sind kürzere Entwicklungszeiten und geringere Kosten der Entwicklung erreichbar. Ebenso steigt durch die Wiederverwendung getesteter Klassen die Qualität gegenüber einer kompletten Neuentwicklung aller Programmteile. Der Aufwand wird überschaubarer, wodurch sich das Projektrisiko verringert.

7.3 IBM: *SanFrancisco™Framework*

Frameworks sind häufig rein software-technisch motiviert. Sie dienen der Entwicklung von Software in Domänen wie Entwicklung von Benutzungsoberflächen, Persistenz etc. Der Domäne Entwicklung betrieblicher Anwendungssysteme dient das von IBM entwickelte *SanFrancisco™Framework*. Es wird im Folgenden vorgestellt.

7.3.1 Einsatzbereich

In der zweiten Hälfte der 1990er Jahre wurden verteilte Systeme und Objektorientierung von vielen Anwendungsentwicklern (*Independent Software Vendors ISV*) als zukunftsweisende Technologien angesehen um betriebliche Anwendungssysteme zu entwickeln. Jedoch mangelte es an entsprechender personeller Kompetenz auf diesem Gebiet. Vor diesem Hintergrund beabsichtigte die *IBM* mit dem *SanFrancisco™Projekt*[6], den Entwicklern zu helfen, kommerzielle betriebliche Anwendungssysteme auf Basis dieser Technologien zu realisieren ([Bohr97], [Bohr98], [Boh+98]), basierend auf getesteten und somit robusten Komponenten [Schm00, 71]. Beispiele für die Anwendungsentwicklung mit dem *SanFrancisco™Framework* finden sich bei [ScLi99] und [Schm00].

BOHRER nennt die drei Hauptziele für die Produktentwicklung [Bohr98, 157ff]:

- **Ziel 1**

  Software-Anbietern, die wenig Erfahrung mit Objektorientierung haben, soll ein leichter Einstieg geschaffen werden. Erreicht werden soll dies durch Wiederverwendung[7], Beispiele grafischer Oberflächen, evolutionären Einstieg in die Technologie, leicht erweiterbare Komponenten, flexible Persistenzmechanismen und integrierte Werkzeuge für Entwickler.

- **Ziel 2**

  Applikationen sollen entwickelt werden können, die den Unternehmen einen Wettbewerbsvorteil verschaffen. Möglich soll dies werden durch objektorientiertes Design, Un-

[6] In der Literatur finden sich die beiden Schreibweisen 'San Francisco' (siehe u. a. [Pet+97], [Bohr98], [Henn98]) und 'SanFrancisco' (siehe u. a. [ScLi99], [Car+00], [O00]). Hier wird durchgehend die Schreibweise 'San-Francisco' verwendet, auch um von der gleichnamigen Stadt zu unterscheiden.

[7] Bis zu 40% einer Anwendung sollen bereits durch das Framework in generischer Form vorliegen

terstützung der Client-Server-Verteilung, Trennung von grafischer Oberfläche und Geschäftsobjekten (Einsatz von *MVC*), Trennung von *business objects*, *commands* und *selections* sowie die Möglichkeit weitere Komponenten selbst zu entwickeln.

- **Ziel 3**
  Entwickeln einer offenen Lösung, die eine gegenseitige Abstimmung von Kosten, Performanz und Fähigkeiten erlaubt.

7.3.2 Architektur

Die Architektur des *SanFrancisco™Framework* ist ausgehend von der fachlichen Ebene (*top down*) entworfen worden [Henn98]. Es besteht aus den drei Ebenen *Core Business Processes*, *Common Business Objects* und *Foundation Layer* (siehe Abbildung 7.5). Das Framework ist komplett in Java realisiert, um Unabhängigkeit von der Plattform und dem Betriebssystem zu erreichen [Schm00, 71].

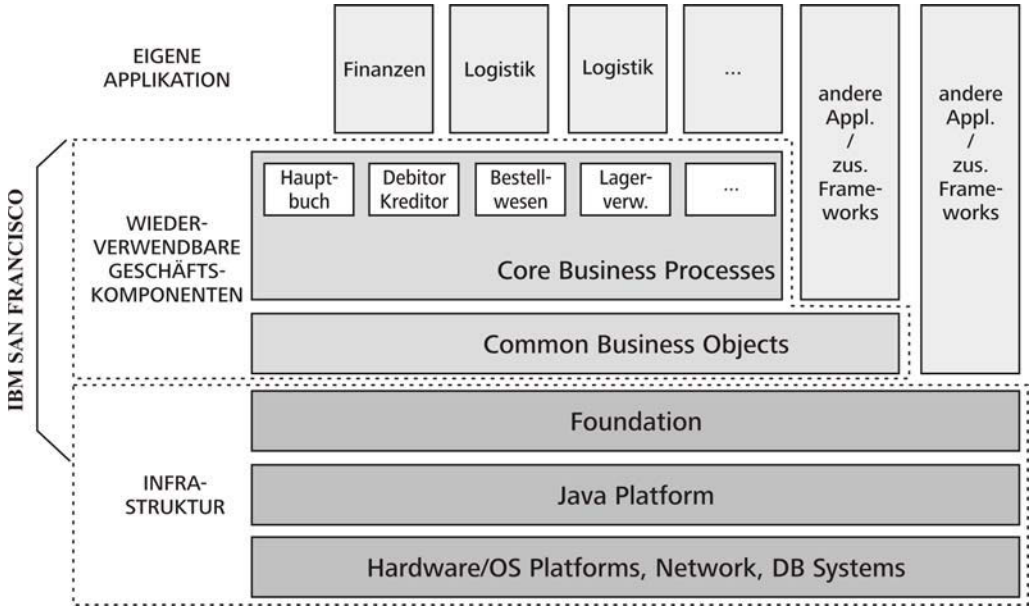

Abbildung 7.5: Architektur des IBM *SanFrancisco™Framework* (nach [Henn98, 37])

Core Business Processes

Auf oberster Ebene der Wiederverwendung finden sich die so genannten Kerngeschäftsprozesse (*Core Business Processes*) Hauptbuch, Debitoren- und Kreditorenkonten sowie Lager- und Be-

stellverwaltung [Bohr98, 158ff]. Der *Core Business Process Layer* liefert Software-Bausteine für die Anwendungsentwicklung in einer spezifischen betrieblichen Domäne, bei flexibel gestaltbarer Ablauflogik [Schm00, 73]. Die beinhalteten Fragmente sind aus der Ableitung konkreter Anwendungen der jeweiligen Domäne entstanden.
Bei der Entwicklung einer Applikation auf Basis des *SanFrancisco™Framework* können drei Szenarien unterschieden werden [Schm00, 73]:

- Alle Funktionen sind schon in der Domänenvorgabe enthalten.
- Die Anforderungen differieren in der Abwicklung bestimmter Algorithmen. In diesem Fall werden an die Hot-Spots entsprechende Änderungen angefügt, Divergenzen werden durch neu implementierte Verfahren angeglichen.
- Der angebotene Funktionsumfang entspricht in keiner Weise den Anforderungen, so dass die Anwendungsentwicklung unter Einbeziehung der im Folgenden beschriebenen Schichten *Common Business Objects* und *Foundation* stattfinden muss.

Common Business Objects

Funktionen, die von der Mehrzahl der betrieblichen Anwendungen benötigt werden, sind im *Common Business Objects Layer* in allgemeinen Geschäftsobjekten implementiert [Schm00, 72]. Diese unterstützen die Wiederverwendung, Verständlichkeit, Wartbarkeit und Interoperabilität der Kerngeschäftsprozesse [Bohr98, 159]. Die *Common Business Tasks* dieser Schicht können über mehrere *Core Business Processes* wiederverwendet werden.
Es werden drei Gruppen von *Common Business Objects* unterschieden:

- *General Business Objects*
- *Generalized Mechanisms*
- *Financial Business Objects*

Foundation Layer

Der *Foundation Layer* stellt die technische Basisfunktionalität für das Framework bereit und kapselt gleichzeitig die technologischen Aspekte [Schm00, 72]. Die Funktionalität umfasst

unter anderem Dienste für Synchronisation, Persistenz, Datenbankzugriff, Objektverteilung und weitere. Um diese nutzen zu können, müssen die vom Entwickler implementierten Objekte von den bereitgestellten Klassen erben. Für persistente Objekte ist das der Typ *Entity*, für abhängige Teilobjekte gibt es den leichtgewichtigeren Typen *Dependent*. Beide werden jeweils über die *Base Factory* bereitgestellt. Implementierte *Entities* sind über Interfaces hinsichtlich Struktur und Verhalten erweiterbar zu *Dynamic Entities* und *Extensible Items* [Pich99, 53f]. Dadurch sind die Objekte dynamisch zur Laufzeit anpassbar, ohne dass alle Attribute zum Zeitpunkt der Generierung bekannt sein müssen.
BOHRER vergleicht die erbrachte Funktionalität dieser Schicht mit der von *CORBA* der *OMG*, jedoch ohne speziell *CORBA-konform* sein zu wollen [Bohr98, 159].

Design Patterns

Bei der Entwicklung des *SanFrancisco™Framework* wurden in hohem Maße *Design Patterns* eingesetzt [Car+00]. Neben den von GAMMA ET AL. in [Gam+95] beschriebenen Patterns wurden weitere speziell für das Framework entwickelt. Eine Liste dieser Patterns findet sich in [Bohr98]. Der Anwendungsentwickler ist angehalten bei der Realisierung einer Applikation auf diese Patterns zurückzugreifen.

7.3.3 Kritische Würdigung

Das *SanFrancisco™Framework* von IBM war ein großer Schritt in der Software-Entwicklung, da es den Entwicklungsprozess strukturierter machte und im Bereich der Wiederverwendung mit bis zu 40% Code eine sehr hohe Produktivität versprach.
Die Methodik für die Entwicklung mit dem *SanFrancisco™Framework* basiert auf einer Kombination der Methodiken von JACOBSON [Jaco93] und BOOCH [Booc94], die erweitert wurden, um wiederverwendbare Geschäfts-Frameworks entwickeln zu können an Stelle spezieller Anwendungen [Bohr98, 159f].
Problematisch ist allerdings, dass die Architektur an der funktionalen Organisation ausgerichtet ist. Diese hat nichts mit den Begriffen der prozessorientierten Organisation oder mit Geschäftsprozessen zu tun [Zeid99, 8]. Auf der Basis dieses Frameworks entwickelte Anwendungssysteme folgen damit auch einer Abgrenzung nach Funktionsbereichen.

FERSTL & SINZ weisen darauf hin, dass die Abgrenzung von Anwendungssystemen nach Funktionsbereichen sich zunehmend als problematisch erweist [FeSi06, 210]. Das liegt an der notwendigen Abstimmung der wechselseitig voneinander abhängigen Teilsysteme Aufgabenstruktur, Aufbauorganisation und Anwendungssysteme des betrieblichen Informationssystems [Sinz99, 20]. Die Aufgabenstruktur wird zunehmend an den Geschäftsprozessen der Organisation ausgerichtet. Gleichzeitig stehen Unternehmen unter dem Druck, ihre Geschäftsprozesse permanent anzupassen. Bei der Umsetzung dieser Anpassungen erweisen sich entsprechend nach funktionalen Kriterien abgegrenzte Anwendungssysteme häufig als Barriere [FeSi96a]. Das *SanFrancisco™Framework* ist damit nicht für die Abbildung geschäftsprozessorientierter Modelle geeignet.

Weitere Nachteile finden sich in der Literatur (u. a. [Pet+97], [EmEl98], [Wolf98], [Zeid99]), die hier nur kurz erwähnt werden. Das *SanFrancisco™Framework* ist lediglich eine neue Form einer Software-Entwicklungsumgebung und rein aus der Software-Entwicklungsperspektive entwickelt worden [Zeid99]. Es wird kein korrespondierender Modellierungsansatz zur Verfügung gestellt [EmEl98, 4]. Große Anzahl an zu implementierenden Klassen innerhalb der Anwendungsentwicklung. Nachteilig wirkt sich zudem aus, dass nur eine rudimentäre grafische Benutzungsschnittstelle zu Testzwecken verfügbar ist [Pet+97, 81] und die Lernkurve sehr hoch ist [Wolf98, 117]. Aufgrund der aufwendigen Infrastruktur des Frameworks ist der Einsatz für kleine bis mittlere Anwendungen nicht sinnvoll.

Das *SanFrancisco™Framework* ist inzwischen Teil des IBM-WebSphere- Produktprogramms geworden.

7.4 OMG: Business Object Facility

Das Konzept der **Business Objects** wurde Mitte der 1990er Jahre von der *Object Management Group* (OMG) eingeführt, mit dem Ziel, den Verlust an semantischer Information zwischen fachlichem Entwurf und software-technischer Implementierung zu minimieren. Ein Business Object kann sowohl eine Person als auch einen Geschäftsprozess oder ein Konzept repräsentieren [OMG96, 19] (siehe auch [Fer+97, 14]).

Um die Entwicklung komplexer Anwendungssysteme unter Verwendung sowohl allgemeiner wie auch unternehmensspezifischer Business Objects möglichst einfach zu gestalten, sollte ein

Framework auf Basis der *Common Object Request Broker Architecture* (CORBA) spezifiziert werden. In [OMG96] werden Common Business Objects sehr weit gefasst definiert als Abbildung solcher Geschäftssemantik, die sich als gemeinsam und allgemein über die meisten Unternehmen hinweg erweist (z.B. in den Bereichen Fertigung, Finanzen oder Personalverwaltung). Oder wie es die OMG beschreibt [OMG96, 21]:

> 'Common business objects are those objects that represent things and concepts so common to every business that some of the semantics of those things and concepts can be standardized."

Korrespondierend wird unter der **Business Object Facility** (BOF) eine Infrastruktur verstanden, die benötigt wird, um Business Objects zu definieren und zu unterstützen, die als kooperierende Anwendungskomponenten in einem verteilten Objektumfeld operieren [OMG96, 18]. Die Architektur der Business Object Facility wie in Abbildung 7.6 dargestellt ähnelt der des *SanFrancisco™Framework* sehr (siehe Abbildung 7.5).

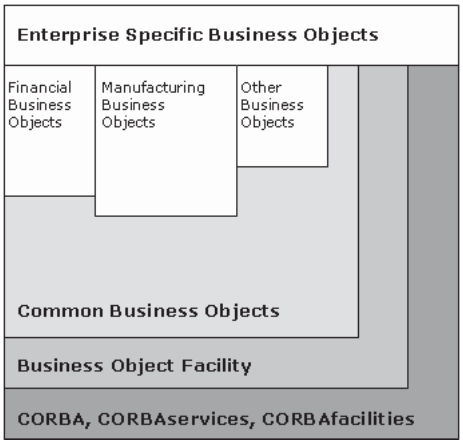

Abbildung 7.6: Business Object Facility und Common Business Objects [OMG96, 20]

Auf oberster Ebene befinden sich Business Objects, die unternehmensspezifische Funktionalität abbilden (Enterprise Specific Business Objects). Diese werden unter Einsatz der darunter liegenden Schichten, wie vorgefertigter und allgemeiner Business Objects, realisiert. Von der Business Object Facility werden neben einer geeigneten technischen Infrastruktur auch Mechanismen zur abstrakten und anwendungsnahen Beschreibung von Business Objects zur Verfügung gestellt.

Geschäftsprozesse werden bei diesem Ansatz als eigene Business Objects betrachtet und dadurch in einer eigenen Komponente realisiert. So werden zwar alle relevanten Objekte, die für die Durchführung des Geschäftsprozesses notwendig sind, gemeinsam mit diesem im Business Object zusammengefasst. Eine Trennung zu den übrigen Komponenten findet nicht statt (vgl. [Zeid99]). Hinzu kommt, dass keine Beziehung festgelegt ist zwischen fachlichen Geschäftsprozessmodellen und software-technischen Komponenten.
Die Business Objects bilden auch, ähnlich wie im *SanFrancisco™Framework*, hier nicht das Verständnis von Geschäftsprozessen ab, sondern orientieren sich zunächst nur an Organisationseinheiten bzw. betrieblichen Funktionen (siehe Financial Business Objects, Manufacturing Business Objects etc.)
Der Ansatz der OMG zur Verknüpfung von fachlicher Ebene und systemtechnischer Ebene der Business Objects ist die *Business Application Architecture*. So finden sich auf fachlicher Ebene ebenfalls das Business Object als Baustein wieder. Dieses besteht aus Name, Attributen, Verhalten und Beziehungen zu anderen Business Objects sowie Business Rules, zur Steuerung des Verhaltens. Jedem fachlichen Business Object wird ein solches auf systemtechnischer Ebene zugeordnet [OMG96, 19]. Eine Ableitungsvorschrift für fachliche Business Objects existiert ebensowenig wie klare Abkrenzungskriterien [Fer+97, 15].
Für die hier geforderte durchgängige fachliche Ableitung des Software-Entwurfs ist der beschriebene Ansatz der OMG nicht tauglich.

7.5 Zusammenfassung

Die Arbeit betrachtet den Einsatz eines Frameworks im Rahmen eines modellgetriebenen Entwicklungsprozesses auf Basis der Modellierungsmethodik SOM. Aus Sicht des fachlichen Modells sind Frameworks aus software-technischen Domänen (z. B. Benutzungsoberflächen, Persistenz) und *Component Integration Frameworks* [Schm00, 11] (z. B. J2EE von Sun, .NET von Microsoft) zu unspezifisch. Frameworks wie *SanFrancisco*, haben ein von SOM abweichendes Verständnis von Geschäftsprozessen. Ein geeignetes Framework ist im Rahmen der Arbeit zu entwerfen und zu implementieren. Mit Bezug zu den Fachkonzepten der SOM-Methodik ist dabei nicht technische Funktionalität Ausgangspunkt des vorliegenden Framework-Entwurfs, sondern die fachliche Ableitung aus Anforderungen betrieblicher Informationssysteme.

Das schrittweise, modellbasierte Vorgehen der Spezifikation von Anwendungssystemen erfolgt in SOM *top down*, ausgehend vom Unternehmensplan. In der SOM-Unternehmensarchitektur lässt sich dieser Zusammenhang wie folgt abbilden (siehe Abbildung 7.7).

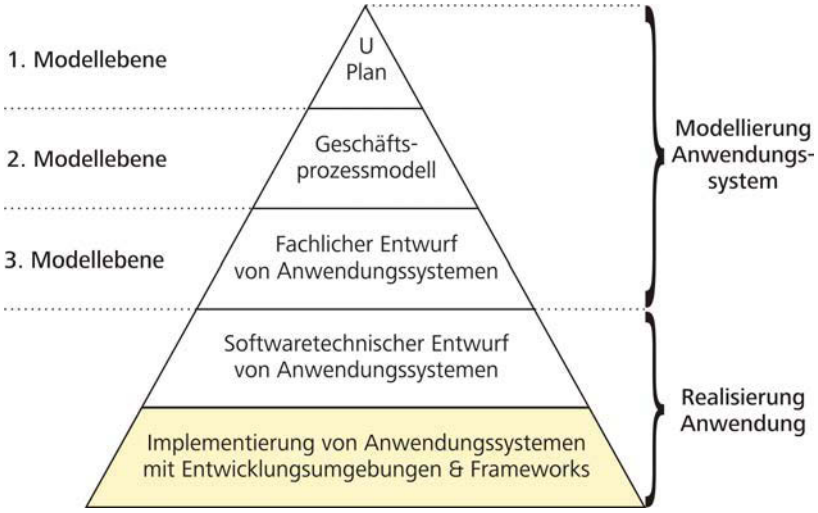

Abbildung 7.7: Unternehmensarchitektur der SOM-Methodik (nach [Mali97, 6])

Der fachliche Entwurf von Anwendungssystemen erfolgt auf der 3. Modellebene. Die Anwendungssystem-Spezifikation umfasst ein konzeptuelles Objektschema (KOS), das aus konzeptuellen Objekttypen und ihren Beziehungen besteht, sowie ein Vorgangsobjektschema (VOS), das mittels Vorgangsobjekttypen das Zusammenwirken konzeptueller Objekttypen bei der Durchführung betrieblicher Aufgaben beschreibt [FeSi06, 189]. Hinzu kommt das Interface Objektschema (IOS). Es umfasst die Schnittstellen-Eigenschaften des Anwendungssystems, die aus den betrieblichen Vorgängen ableitbar sind [Ambe93, 35]. Im Rahmen der software-technischen Spezifikation von Anwendungssystemen wird das fachliche Modell im Hinblick auf dessen Realisierung auf das objektorientierte Software-Architekturmodell abgebildet. Das Ergebnis ist die anwendungsspezifische Software-Komponente, sie spezifiziert die zu realisierende Nutzermaschine.

Das Framework bietet anwendungsneutrale Software-Komponenten an, die sich an der fachlichen Abgrenzung der Komponenten der Nutzermaschine orientieren, die Abbildung 7.8 zeigt. Der Einsatz des Frameworks lässt sich wie in Abbildung 7.9 darstellen. Das zu realisierende Anwendungssystem wird in Form der Modelle IOS, VOS und KOS spezifiziert. Das Framework stellt für die jeweiligen Konzepte anwendungsneutrale Funktionalität bereit. Die Realisierung des Anwendungssystems erfolgt durch Abbildung der anwendungsspezifischen Funktionalität

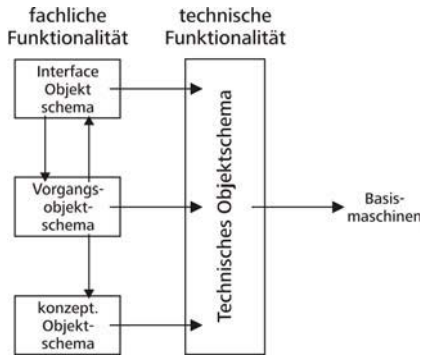

Abbildung 7.8: Das ooAM mit Schnittstellen [Ambe93, 37])

auf den Komponenten des Frameworks.

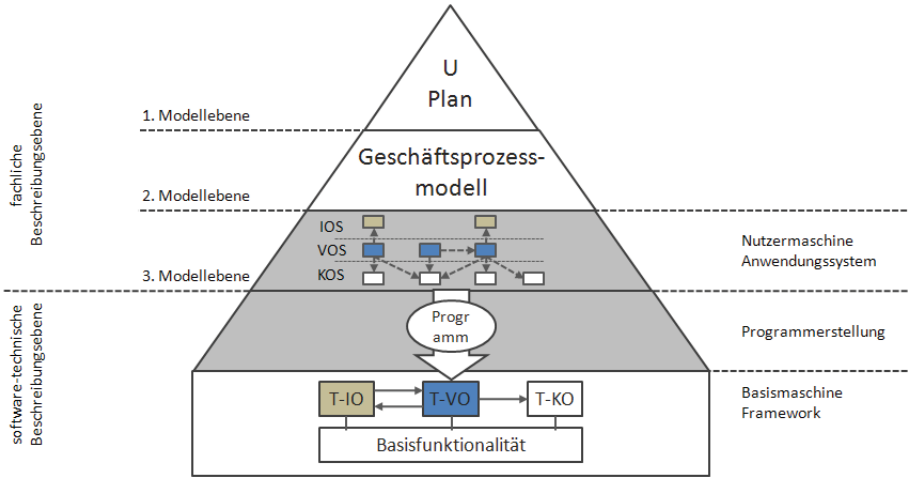

Abbildung 7.9: Framework-Einsatz in der Unternehmensarchitektur der SOM-Methodik (nach [Mali97, 6])

Dadurch kann die Komplexität der Systementwicklungsaufgabe bei der Realisierung verringert werden. Die erreichte Vereinfachung kann anhand des Software-Systemmodells verdeutlicht werden:

- Die Programmerstellung ist vereinfacht, da die zu überbrückende semantische Lücke klein ist, bedingt durch den geringen Komplexitätsabstand zwischen den Komponenten des Frameworks und den Konzepten des Anwendungsmodells.
- Die Abbildung des fachlichen Modells auf die Komponenten des Frameworks wird vollständig für alle definierten Fachkonzepte unterstützt, einschließlich des definierten Kooperationsverhaltens (siehe oben). Die Fachkonzepte der SOM-Methodik spezifizieren interaktive Anwendungssysteme. Mit den Komponenten des Frameworks lassen sich die

entsprechenden Oberflächen zur Mensch-Computer-Kommunikation (MCK) ohne weitere Programmierung ableiten.

- Die Fachkonzepte der SOM-Methodik unterstützen die fachliche Einhaltung von Integrationszielen [Ambe99, 13] (siehe auch [Rose99]). Das Framework hält software-technische Komponenten bereit, um die Verfolgung der Integrationsziele zur Entwicklungs- und Laufzeit zu unterstützen.

- Das fachliche Modell des Anwendungssystems besitzt Strukturähnlichkeiten und -analogien zur Ebene der Geschäftsprozesse des Unternehmens. Diese Analogien werden von der Basismaschine Framework abgebildet. Anpassungen der Geschäftsprozesse bleiben durch korrespondierende Anpassungen auf Ebene des Anwendungssystems nachvollziehbar.

- Die Programmerstellung lässt sich bei entsprechend geringer semantischer Lücke und gegebener Vollständigkeit der Abbildbarkeit automatisieren (nicht Teil der Arbeit). Im Sinne der modellgetriebenen Software-Entwicklung stellt das Framework eine geeignete Zielplattform dar.

Der Entwurf und die Realisierung des Frameworks **moccabox** wird im abschließenden Teil III der Arbeit vorgestellt.

Teil III

Das Framework moccabox

8 Software-Architektur des Frameworks moccabox

Aufbauend auf den vorangegangenen Kapiteln wird im folgenden eine Basismaschine entworfen, die eine vollständige Abbildung einer gemäß der SOM-Methodik spezifizierten Nutzermaschine ermöglicht, mit dem Ziel dadurch sowohl Umfang als auch Komplexität der Systementwicklungsaufgabe für interaktive Anwendungssysteme zu verringern. Als Basismaschine wird ein Framework entworfen, das aus Außensicht wiederum als Nutzermaschine Funktionalität anbietet. Im folgenden Kapitel 9 wird auf eine mögliche Realisierung des Frameworks unter Verwendung verfügbarer Basismaschinen eingegangen.

Interaktive Anwendungssysteme sind häufig komplex und müssen daher strukturiert werden, um eine bessere Wartbarkeit zu erreichen. Die Struktur solcher Systeme wird durch ihre Architektur festgelegt, die das System in Komponenten mit festgelegten Rollen und Schnittstellen zwischen den Komponenten unterteilt. [HuCa95, 1]

Das vorliegende Kapitel dient der Darstellung und Beschreibung der Architektur des Frameworks **moccabox**. Die **moccabox** stellt ein Architektur-Framework (*Architectural Framework*) für die Entwicklung betrieblicher Anwendungssysteme dar. Es integriert eine Vielzahl technischer Aspekte, die eine Implementierung der jeweiligen Schichten auf eine ganz bestimmte Weise bedingen [Evan04, 74]. Es wird erreicht, dass sich der Entwickler auf die Modellierung der grundlegenden Geschäftsprobleme - hier die Fachspezifikationen VOS und KOS der anwendungsspezifischen Funktionalität - konzentrieren kann, mit dem Ergebnis höherer Produktivität und Qualität [Evan04, 75].

Der Aufbau des Frameworks **moccabox** folgt dem in Kapitel 3.3.4 vorgestellten objektorientierten Software-Architekturmodell (ooAM) von AMBERG [Ambe93] und realisiert die technische Funktionalität unter Verwendung entsprechender Basisfunktionalität. Es unterstützt damit die *Separation of Concerns* und bietet gleichzeitig die Vorteile des aspektorientierten Para-

digmas [Fay+00, 45]. Aspekte werden als Eigenschaften eines Systems betrachtet, die nicht notwendigerweise mit den funktionalen Komponenten des Systems übereinstimmen [Con+00]. Neben dem objektorientierten Software-Architekturmodell basiert die Architektur des Frameworks auf dem Konzept **verteiltes Software-System**[1], das sich durch folgende Struktur- und Verhaltensmerkmale auszeichnet [Sinz97, 15], [FeSi06, 211]:

- Aus Außensicht stellt das verteilte System ein integriertes System dar, das gemeinsame Ziele verfolgt.
- Zur Erfüllung dieser Ziele kooperieren mehrere autonome Komponenten, aus denen das System besteht.
- Keine der Komponenten besitzt die globale Kontrolle über das Gesamtsystem.
- Für den Nutzer ist die Verteilung auf Grund der Verteilungstransparenz nicht sichtbar.
- Die Komponenten des Systems sind lose gekoppelt und interagieren durch den Austausch von Nachrichten.

Ein Framework stellt aber nicht nur Funktionalität im Sinne einer Klassenbibliothek zur Verfügung (vgl. hierzu Kapitel 6.7). Es übernimmt auch die Ablaufsteuerung der Anwendung, entsprechend seiner Teilkomponenten. Deshalb wird zwischen den Schichten mit technischer Funktionalität für *Schnittstellen-* und *Vorgangsfunktionalität* sowie *konzeptueller Funktionalität* unterschieden. Diese Unterteilung ist in Abbildung 8.1 dargestellt. Entsprechend dem Konzept der *Inversion of Control* (siehe u.a. [JoFo88, 25] und [FaSc97, 34]) realisiert das Framework anwendungsunspezifisch die Schnittstellen der Komponenten IOS, VOS und KOS untereinander.

Bezogen auf die Entwicklung betrieblicher Anwendungssysteme mit einem **Application Framework**[2] lassen sich die (Teil-)Funktionalitäten entsprechend Abbildung 8.1 einordnen. Die fachliche Funktionalität repräsentiert das zu entwickelnde Anwendungssystem. Diese wird auf Basis der technischen Funktionalität realisiert, welche vom Framework zur Verfügung gestellt wird. Es enthält Konzepte der fachlichen Domäne (betriebliche Anwendungssysteme) und verringert so den Abstand und die Komplexität beim Übergang von fachlicher Spezifikation zur

[1] Vgl. [Ensl78] zitiert in [Sinz97, 15].
[2] Bei GAMMA werden diese auch als *Framework im Großen* bezeichnet [Gamm92, 107].

technischen Implementierung. Es handelt sich daher um eine reichhaltige, domänenspezifische Plattform [StVö05, 137]. Das ermöglicht eine architekturzentrierte MDSD (vgl. Kapitel 5.4) und erhöht das Abstraktionsniveau der Implementierung. Das Framework stellt seine Funktionalität abstrakt zur Verfügung [John97, 11]. Unter Nutzung bestimmter Basistechnologien wie Persistenzmechanismen, GUI-Frameworks etc. lassen sich verschiedene konkrete Implementierungen realisieren, ohne dass die fachliche oder technische Funktionalität angepasst werden müssen. Eine konkrete Realisierung der **moccabox** sowie die eingesetzten Basismaschinen werden in Kapitel 9 vorgestellt.

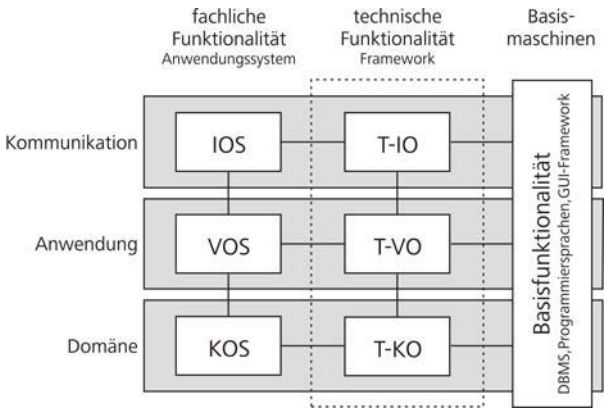

Abbildung 8.1: Architektur des Frameworks moccabox

Bei AMBERG werden die Schemata der fachlichen Funktionalität IOS, VOS und KOS direkt den Teilkomponenten Kommunikation, Anwendungsfunktionen und Datenverwaltung des ADK-Strukturmodells zugeordnet [Ambe93, 68f]. In der vorliegenden Arbeit wird die Entsprechung anders verwendet (siehe Abbildung 8.1). VOS und KOS repräsentieren gemeinsam die Anwendungsfunktionen des zu realisierenden Anwendungssystems. Vielmehr entspricht das VOS dem *Application Layer* der in Abbildung 3.6 dargestellten Schichtenarchitektur, das KOS entsprechend der Domäne. Die Funktionalität der Datenverwaltung findet sich als technische Funktionalität im Framework wieder, die unter Verwendung geeigneter Basismaschinen realisiert wird. Im Folgenden wird die Unterteilung in die drei Schichten Kommunikation, Anwendungsfunktionen und Domäne verwendet, die Schicht der Infrastruktur wird als Basismaschine interpretiert.

Desweiteren folgt die Architektur des Frameworks **moccabox** den Architekturmustern Nutzer- und Basismaschine, ADK-Strukturmodell und dem Client/Server-Modell [FeSi06, 187].

Abbildung 8.2 zeigt die Komponenten der **moccabox** entsprechend dem Verständnis des SOM-

Architekturmodells, wie es in Abbildung 8.1 vorgestellt wurde.

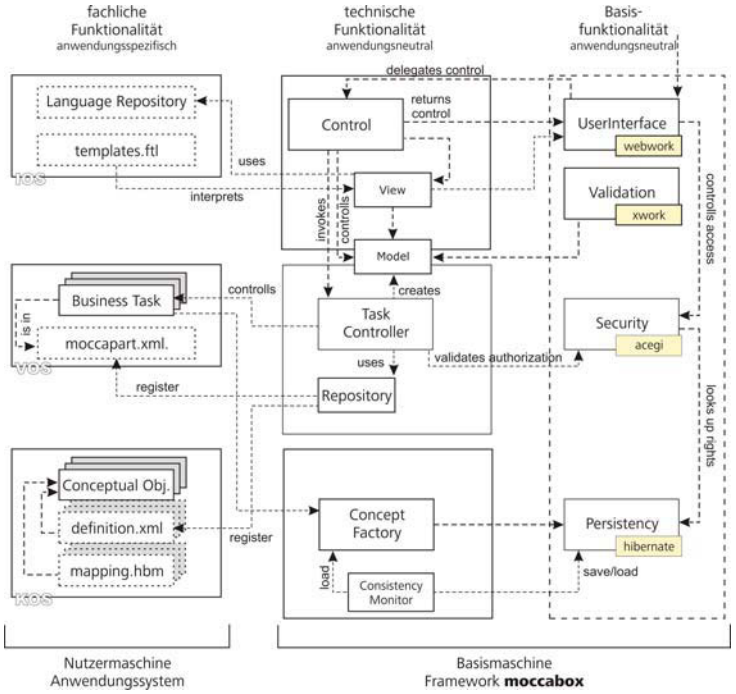

Abbildung 8.2: Architektur der **moccabox** nach dem ooAM

8.1 Das Repository

Das **Repository** der **moccabox** ist ein Katalog zur Verwaltung der Metadaten, die die spezifische Funktionalität des zu realisierenden Anwendungssystems enthält. Ein Repository ist allgemein ein System, das Metadaten enthält über „Struktur und Aspekte des Verhaltens von Systemen, die Daten der Umwelt (als Abbild eines Ausschnitts der Realität) manipulieren“ [Habe93, 17]. Es beinhaltet Informationen über Objekte der Software-Produktion wie Programme, Datenfelder, Masken, Listen sowie deren Beschreibungen und Beziehungen. Zudem verwaltet ein Repository diese Informationen und stellt sie dem System zur Verfügung ([Habe93, 15ff]).

Die Daten, die das Repository der **moccabox** enthält, sind anwendungsspezifisch und repräsentieren die fachliche Funktionalität (siehe Kapitel 3.3.4) des Anwendungssystems. Der Inhalt entspricht den Fachspezifikationen der 3. Ebene des SOM-Ansatzes: Interface-Objektschema (IOS), Vorgangsobjektschema (VOS) sowie konzeptuelles Objektschema (KOS). Diese anwendungsspezifischen Definitionen werden im Repository auf entsprechende Objektbeschreibun-

gen abgebildet. Die jeweiligen Komponenten des Frameworks greifen auf das Repository zu, um entsprechende Informationen abzurufen[3]. Die Interaktion zwischen den jeweiligen Objekttypen ist durch das Framework festgelegt. Somit bleibt die Kontrolle beim Framework, die Funktionalität des Anwendungssystems wird aber durch die auszuführenden Objekte beschrieben. Das entspricht der *Inversion of Control*. Es wird beispielsweise ein auszuführender Vorgang des Anwendungssystems im Repository definiert, der Aufruf erfolgt hingegen durch das Framework (siehe Abbildung 7.2).

Die Literatur unterscheidet zwischen passiven und aktiven Repositories [Habe93]. Ein **passives Repository** steht nur zur Entwicklungszeit (*build-time*) zur Verfügung. Es ist somit nicht Bestandteil des eigentlichen Anwendungssystems. Ein **aktives Repository** steht im Gegensatz dazu auch zur Laufzeit des Systems (*run-time*) als Komponente zur Verfügung und dient weiteren Komponenten des Systems Informationen während der Ausführung des Programms als Informationsquelle. Das Repository der **moccabox** übernimmt sowohl passive wie auch aktive Aufgaben.

Passive Aufgaben

Alle bereits beschriebenen Objekttypen, deren Struktur und Beziehungen werden zur Entwicklungszeit im Repository erfasst. Es dient dem Entwickler folglich als Informationsquelle über bereits implementierte Datenobjekte und Funktionen. Auf dieser Basis kann es helfen, Daten- und Funktionsredundanz zu vermeiden, da bereits erfasste Objekte dargestellt werden. Ein Editor, der schon zur Entwicklungszeit die Daten auf Redundanz überprüft und doppeltes Erfassen überprüft kann den Entwickler bei der Realisierung eines Anwendungssystem zusätzlich unterstützen[4]. Die Unterstützung ist gegenwärtig rein passiv, da der Nutzer des Frameworks selbst die Redundanz auf Basis der Informationen optimieren muss, die ihm das Repository zur Verfügung stellt. Da die fachliche Funktionalität des Anwendungssystems während der Entwicklungszeit formal erfasst wurde, bietet es sich an, einen Teil der physischen Implementierung des Anwendungssystems daraus abzuleiten [Tane01, 69]. Ein aktuelles Repository erleichtert zudem die Wartbarkeit des Systems. Da es die Struktur des Anwendungssystems

[3] Vgl. auch das Prinzip des *Resource Sharing* als Operation zur Erreichung von Software-Qualität in [KaBa94].

[4] Ein solcher Editor ist eine Komponente einer framework-basierten Entwicklungsumgebung. Dies kann zum Beispiel ein Plug-In der Entwicklungsumgebung Eclipse sein. Ein solcher Editor ist nicht Teil der vorliegenden Arbeit bzw. der gegenwärtigen Version der **moccabox** (siehe Ausblick in Kapitel 10.2).

enthält, sind die miteinander in Beziehung stehenden Systemelemente direkt ersichtlich und die von einer Änderung betroffenen Teile sind leicht erkenntlich [Habe93, 12].

Aktive Aufgaben

Das Framework bezieht seine Informationen über das Anwendungssystem zur Laufzeit direkt aus dem Repository. So verwendet die Kontrollkomponente (siehe Kapitel 8.3.2) die Informationen über die verfügbaren Vorgangsobjekte und die Integritätskomponente (siehe Kapitel 8.3.3) nutzt die erfassten Integritäts- und Zugriffsbedingungen für konzeptuelle sowie Vorgangsobjekttypen. Die erfassten Daten des Repository sind direkte Grundlage der Ausführung des Anwendungssystems. Der Zustand des Repository muss damit konsistent und aktuell sein, da das Anwendungssystem nur dann lauffähig ist, wenn alle Objekttypen und ihre Beziehungen korrekt erfasst sind. So verhindert beispielsweise die Kontrollkomponente einen Zugriff auf einen Objekttyp als unzulässig, sofern er nicht im Repository spezifiziert ist. Die Pflege der Daten bedeutet keinen zusätzlichen Aufwand und wird demzufolge nicht vernachlässigt.
Im Folgenden werden die Komponenten beschrieben, die auf den Metadaten des Repository aufbauen. Diese befinden sich auf der jeweiligen funktionalen Ebene *IO-Basisschicht* (Kapitel 8.2), *VO-Basisschicht* (Kapitel 8.3) respektive *KO-Basisschicht* (Kapitel 8.4).

8.2 Technische Funktionalität für Interface Objekte

Die Schnittstellenfunktionalität eines Anwendungssystems wird gemäß dem ooAM als Ergebnis der Interface-Objektmodellierung durch ein anwendungsspezifisches Interface-Objektschema (IOS) spezifiziert [Ambe93, 119-129]. Mit dem Sachziel, „jeden Vorgang mit der für seine Durchführung notwendigen Schnittstellen-Funktionalität zu versorgen“ [Ambe93, 120]. Jede Kommunikation eines AwS mit seiner Verfahrensumgebung erfolgt über das IOS, wodurch diese Schicht für die Qualität und Akzeptanz des gesamten Anwendungssystems von besonderem Interesse ist [Ambe93, 119].
AMBERG beschreibt die folgenden Aufgaben des IOS für die weiteren Komponenten des ooAM [Ambe93, 120]:

- Durchführung einer Mensch-Computer-Kommunikation während einer teilautomatisier-

ten Vorgangsbearbeitung.

- Durchführung einer Computer-Computer-Kommunikation mit der sonstigen Verfahrensumgebung während einer Vorgangsbearbeitung.
- Verwaltung von Ereignissen. Die auslösenden Ereignisse werden zur Aktivierung von Vorgängen an Vorgangs-Objekttypen weitergereicht.

Aufgabe der *IO-Basisschicht* des Frameworks **moccabox** ist die Abbildung aller nicht anwendungsspezifischen Schnittstellenfunktionalität, die der IOS-Komponente als Basismaschine zur Verfügung steht. Der funktionale Kern der Anwendung muss unabhängig von der Benutzungsschnittstelle gehalten werden. Die Benutzungsschnittstelle muss häufig geändert bzw. angepasst werden, der funktionale Kern der Anwendung bleibt hingegen weitgehend stabil [Bus+00, 122]. Es wird auch hier zwischen der Schnittstelle für die Kommunikation mit Personen (Mensch-Computer-Schnittstelle) und weiteren AwS (Computer-Computer-Schnittstelle) unterschieden [Ambe93, 119]. Für die MCK steht auf Seiten des AwS ein so genanntes **User-Interface Management System** (UIMS)[5] [FeSi06, 403-408] zur Verfügung. Wichtig bei der Realisierung der UIMS-Basismaschine im Framework ist vor allem die Austauschbarkeit des **Graphical User Interface** (GUI) ohne Beeinträchtigung der anderen Schichten des Anwendungssystems (vgl. [Bus+00, 122]).

Die Benutzungsoberfläche umfasst „[...]alle Einheiten, Formen und Techniken, durch die Benutzer mit dem Computersystem kommunizieren“ [Wand93, 4] und stellt eine Teilmenge der Benutzungsschnittstelle[6] dar. Der Benutzungsoberfläche kommt eine sehr wichtige Rolle zu, denn an der Schnittstelle zum personellen Aufgabenträger muss Komplexität verborgen und die „Illusion von Einfachheit“ erzeugt werden [Booc94, 20]. Um die effiziente Entwicklung der Benutzungsoberfläche interaktiver Anwendungssysteme auf Basis des Frameworks **moccabox** zu unterstützen, muss die *IO-Basisschicht* folgende Anforderungen erfüllen:

- Aufbau entsprechend einer objektorientierten Architektur

[5] Zu UIMS siehe u.a. auch [BaCo91, 191-220]

[6] Die Definition der Benutzungsschnittstelle fasst WANDMACHER weiter. Sie umfasst auch notwendiges werkzeugunabhängiges Aufgabenwissen und werkzeugspezifisches Wissen des Benutzers [Wand93, 2]. Die Benutzungsoberfläche repräsentiert lediglich den für den Anwender sichtbaren Teil der zum Anwendungssystem gehörenden Benutzungsschnittstelle [Wand93, 4].

- Trennung von Layout, Inhalt und Logik
- Unterstützung verschiedener Ausgabeformate
- Unterstützung software-ergonomischer Prinzipien
- Unterstützung von Mehrsprachigkeit

Der Aufwand für die Entwicklung der Benutzungsoberfläche für die Mensch-Computer-Kommunikation liegt nach empirischen Studien von MYERS & ROSSON bei bis zu 50% [MyRo92, 195]. Durch automatische Generierung der Oberfläche zur Laufzeit entsprechend der im IOS modellierten oder abgeleiteten Interface Objekte kann dieser Aufwand erheblich reduziert werden. Im Rahmen der Entwicklung des Frameworks **moccabox** wird der Realisierung eines entsprechenden GUI-Framework große Aufmerksamkeit gewidmet.
Die folgenden Kapitel beschreiben den Aufbau der *IO-Basisschicht* entsprechend der vorgestellten Anforderungen.

8.2.1 Ereignisbehandlung der Kommunikations-Schnittstelle

Die *IO-Basisschicht* muss zum einen die Schnittstelle zur Applikation selbst schaffen zum anderen ist sie für die Verarbeitung der eingehenden Anfragen zuständig.
AMBERG unterscheidet drei Fälle, durch die ein Ereignis im Anwendungssystem ausgelöst werden kann [Ambe93, 123]. Dies kann a) ein Nutzer über die Mensch-Computer-Schnittstelle des Anwendungssystems tun, b) die sonstige Verfahrensumgebung über die Computer-Computer-Schnittstelle oder c) durch ein Ereignis infolge des Nach-Ereignisses [FeSi06, 193] eines Vorgangs ausgelöst werden. Bei a) und b) spricht man von externen Ereignissen, bei c) von internen.
Nach (Teil-)Durchführung des Vorgangs liefert die Kommunikationsschnittstelle das Ergebnis an den Absender der Anfrage zurück.
Im ooAM obliegt dem IOS die Ereignis-Bearbeitung [Ambe93, 123]. Diese umfasst die folgenden Aufgaben:

- Externe Ereignisse in Empfang nehmen.
- Ereignisse puffern, sofern mehrere auslösende Ereignisse vorliegen müssen, um eine Vorgangsbearbeitung durchzuführen.

- Vorgänge im VOS aktivieren, indem auslösende Ereignisse an den jeweils zuständigen Vorgangs-Objekttyp weitergeleitet werden[7].
- Ereignisse der Verfahrensumgebung geeignet bekannt zu machen.

Die Schnittstelle zwischen IOS und technischem Objektschema, welches vom Framework zu realisieren ist, beschreibt AMBERG folgendermaßen [Ambe93, 127]:

> „Die Bearbeitung von Schnittstellen-Funktionalität erfolgt unter Zuhilfenahme des TOS. Diese stellt typischerweise eine anwendungsneutrale Funktionalität zur Durchführung der Kommunikation mit den Nutzern oder der sonstigen Verfahrensumgebung bereit. Typischerweise in Form von objektorientierten UIMS und Kommunikationsdiensten."

Das Framework **moccabox** realisiert dafür eine umfassende Basismaschine. Empfang, Verarbeitung und Rückmeldung der Anfrage werden entsprechend einem objektorientierten UIMS realisiert. Im folgenden werden zwei entsprechende Ansätze vorgestellt: das MVC-Pattern (vgl. [KrPo88], [Gamm92, 141f], [Gam+95, 4ff]) und das PAC-Modell [Cout87]. Bei beiden handelt es sich um objektorientierte Architekturmodelle für die Realisierung von *Graphical User Interfaces* (GUI) [HuCa95, 1], die dem Ansatz der Interobjekt-Modellierung zur Abbildung der Funktionalität folgen. Dieser teilt die Funktionalität Präsentation, Dialogkontrolle und Applikationsschnittstelle auf drei kooperierende Objekte auf[8] [Prei99, 280].

8.2.2 Architekturmodelle für UIMS

Es werden verschiedene Architekturmodelle für User-Interface Management Systeme unterschieden [Götz95]:

- **Gerätemodelle** stellen Ein- und Ausgabegeräte in den Vordergrund.
- **Schichtenmodelle** unterteilen interaktive Systeme in Anlehnung an die Übersetzung von Programmiersprachen in Schichten.

[7] Diese Aufgabe teilt sich die *IO-Basisschicht* mit der *VO-Basisschicht*. Die *IO-Basisschicht* ordnet der eingehenden Anfrage das entsprechende Ereignis zu und gibt diese Information an das *VO-Basisschicht* weiter, welches dann die Durchführung der Vorgänge steuert.

[8] Im Gegensatz dazu wird nach der Intraobjekt-Modellierung die gesamte Funktionalität in einem Objekt abgebildet (siehe [Prei99, 280]).

- **Objektorientierte Modelle** betrachten die Nutzerschnittstelle als System kooperierender Interaktionsobjekte.

Die *IO-Basisschicht* wird auf Basis des objektorientierten Architekturmodells realisiert, die eine Weiterentwicklung der Architektur des Seeheim-Modells - dem wohl bekanntesten Schichtenmodell - darstellt [9]. Als objektorientierte Architekturmodelle werden das PAC-Pattern (siehe 8.2.2.1) und das MVC-Pattern (siehe Kapitel 8.2.2.2) vorgestellt. Gerätemodelle werden nicht weiter betrachtet. Die hier vorgestellten Architekturmodelle zerlegen die Benutzungsschnittstelle in eine Menge einheitlicher Abstraktionen[10], deren Module anschließend miteinander verknüpft werden [KaBa94, 9].

8.2.2.1 Das PAC-Modell

Die objektorientierte Unterteilung des UIMS kann nach dem **PAC-Modell** [Cout87] (siehe auch [Bus$^+$00, 145-169]) erfolgen, das die folgenden Aufgaben verfolgt [Cout87, 127f][11].

1. Definition eines konsistenten Frameworks zur Konstruktion eines *User Interface* für die Mensch-Computer-Kommunikation, das auf jeder Abstraktionsebene eingesetzt werden kann. Daraus ergibt sich, dass die ausgetauschten Einheiten (units) zwischen dem abstrakten Teil (Applikation) und dem Controller (UIMS) Konzepte der Anwendung sind.

2. Saubere Trennung von funktionellen Begriffen und Vereinbarungen der Darstellung. Die *Control*-Komponente schließt die Lücke zwischen abstrakter und realer Welt. Die Rolle der *Control*-Komponente kann über die Wahrung der Konsistenz zwischen beiden Welten hinausgehen. So kann sie auch die Verwaltung lokaler Kontextinformationen umfassen.

3. Aufgreifen des objektorientierten Paradigmas und Verwendung seiner Vorteile mit den Begriffen des interaktiven Objekts.

Für größere Anwendungssysteme kann die Architektur in mehrere hierarchisch kooperierende Agenten mit unterschiedlichen Aufgaben unterteilt werden, die jeweils aus den drei Komponen-

[9] Für eine ausführliche Darstellung des Seeheim-Modells siehe u. a. [Gree85].
[10] Vgl. auch das Prinzip der Einheitlichen Komposition als Operation zur Erreichung von Software-Qualität in [KaBa94].
[11] Zum PAC-Modell siehe auch [BaCo91]. Eine Weiterentwicklung für Groupware ist PAC* [Cout97].

ten *Presentation*, *Abstraction* und *Control* bestehen (vgl. [Bus+00, 145]). Abbildung 8.3 zeigt die Präsentation des numerischen Wertes einer Abstraktion als Diagramm.

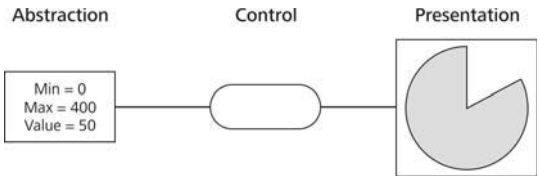

Abbildung 8.3: Das PAC-Modell [Cout87, 126]

Presentation

Die Komponente **Presentation** definiert die konkrete Syntax einer Anwendung, d.h. sie definiert das Input- und Output-Verhalten des Anwendungssystems aus Sicht des Nutzers [Cout87, 126]. Sie spezifiziert die MCK-Schnittstelle des Anwendungssystems [FeSi06, 403].

Abstraction

Die Komponente **Abstraction** repräsentiert die Semantik auf Seiten der Anwendung, die mit der Syntax der Präsentation korrespondiert [Cout87, 126]. Sie implementiert die von der Applikation ausführbaren Funktionen und definiert die MCK aus Sicht des Anwendungssystems [FeSi06, 403] unabhängig vom Ausgabemedium [BaCo91, 170]. Die Komponente Abstraction stellt die Schnittstelle zwischen Anwendungssystem und MCK-Teil dar.

Control

Der **Controller** übernimmt das Mapping der abstrakten Einheiten - implementiert durch die Abstraction - und ihrer Präsentation für den Nutzer [Cout87, 126]. Durch sie werden die Objekte und Operatoren der Präsentationsebene mit denen der Abstraktionsebene verknüpft [FeSi06, 403]. Der Controller überwacht das Verhalten der beiden Perspektiven [BaCo91, 170].

8.2.2.2 Model-View-Controller

Alternativ zum PAC-Modell zur Strukturierung von UIMS existiert das MVC-Modell. Für die Nutzerschnittstelle der Programmierumgebung Smalltalk™wurde eine spezielle Strategie

für Informationsrepräsentation, Anzeige und Kontrolle entwickelt, mit der zwei Ziele verfolgt wurden [KrPo88, 26]:

> „(1) to create the special set of system components needed to support a highly interactive software development process: and (2) to provide a general set of system components that make it possible for programmers to create portable interactive graphical applications easily."

Das **MVC-Modell** unterteilt die Schnittstelle der Nutzerkommunikation in drei Komponenten [HuCa95, 3]: *Model*, *View* und *Controller* (siehe u. a. [KrPo88], [Gam+95, 4ff], [Bus+00, 124-144]). In Abbildung 8.4 ist diese Triade von Klassen [Gam+95, 4] inklusive ihrer Zustände und Nachrichten dargestellt.

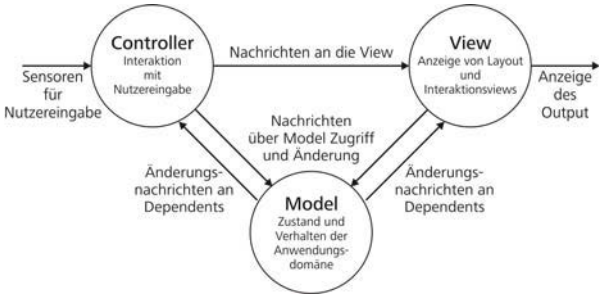

Abbildung 8.4: Model-View-Controller Zustand und Nachrichten [KrPo88, 27]

Controller

Der **Controller** definiert die Art, in der die Nutzerschnittstelle auf die Eingabe des Nutzers reagiert [Gam+95, 4]. Er nimmt Anfragen an die Schnittstelle (Nutzerinteraktion oder Aufruf durch anderes AwS) entgegen. Abhängig von der auslösenden Aktion, wird ein Ereignis zur Durchführung eines Vorgangs erzeugt oder lediglich das Model bzw. die View modifiziert.

Model

Das **Model** stellt das Applikationsobjekt dar [Gam+95, 4]. Es enthält die Daten des Dialogs und spiegelt die domänenspezifische Information des Anwendungssystems wider [KrPo88, 27].
Im Framework **moccabox** ist das Model die zentrale Komponente, die *IO-Basisschicht* und *VO-Basisschicht* miteinander verbindet. Entsprechend dem Verständnis des MVC-Pattern ist es

die Repräsentation des Zustand des Aufgabenobjekts eines Vorgangs sowie darauf verfügbarer Operationen, die anschließend dem Anwender mittels einer korrespondierenden View (siehe unten) dargestellt wird.
Das Model enthält nicht nur Daten über den Zustand des Systems, sondern auch Angaben über diese Daten, so genannte Metadaten. Für die Schicht der Interface Objekte wird von den zu Grunde liegenden Aufgabenobjekten abstrahiert, da nach dem SOM-Architekturmodell die Interface Objekte keinen Zugriff auf die konzeptuellen Objekte und ihre Methoden haben.

View

Die **View** ist die Bildschirmrepräsentation des Zustandes des Model [Gam+95, 4]. Wann immer sich der Zustand des Model ändert wird dies den abhängigen Views mitgeteilt, woraufhin sich diese selbst aktualisieren.
Im MVC-Modell können Views auch geschachtelt sein. Eine View (z. B. eine Seite) enthält dann verschiedene SubViews (z. B. Menü, Navigationsleiste, Objektrepräsentation etc.). In diesem Fall spricht man von **CompositeView** [Gam+95, 5].
COLSMAN & OSTHUS merken an, dass komplexe Anforderungen von dialogbasierten Anwendungen kaum durch UML abgebildet werden können und dadurch die Modellierung der Benutzeroberfläche nicht zufrieden stellend durchgeführt werden kann [CoOs04a, 27]. Zu den Anforderungen gehören:

- Mehrsprachigkeit
- Tabellen
- Registerblätter
- dynamischer Seitenaufbau

Anforderungen an den dynamischen Seitenaufbau sind [CoOs04a, 27]:

- Masken in Abhängigkeit vom Datenbestand gestalten
- Maskenbereiche abhängig vom Programmzustand (Workflow) ein- oder ausblenden
- Eingabefelder entsprechend der Berechtigungen des Benutzers zur Bearbeitung freigeben

Voraussetzung für die Umsetzung dieser Anforderungen ist die strikte Trennung in Datenbestand (*Model*) und beliebige Sichten (*Views*) darauf, da bei einer unstrukturierten Sammlung von Attributen die Übersicht schnell verloren geht.

- Die *View* definiert eine Sicht auf die Daten. Es kann mehrere Sichten auf die gleichen Daten geben. Views fragen Daten vom zugehörigen Model ab und interpretieren diese grafisch. Views können wiederum *SubViews* enthalten oder Teil von *SuperViews* sein.
- Der *Controller* nimmt Benutzerinteraktionen (*Events*) entgegen und verändert das *Model* oder die *View*.

Der zustandslose *Controller* wird als Dienst verwaltet und vom `RequestHandler` aufgerufen. Das *Model* liegt als zustandsbehaftetes Datenobjekt in der *Session* des Nutzers. Bei einer reinen Änderung der *View* wird das *Model* nicht verändert. Ansonsten wird es aus der Anwendungsschicht, in der die Geschäftslogik abläuft, erzeugt und mit Fachdaten der Anwendung gefüllt.
Steuerelementattribute hingegen werden in der View verwaltet. Im Sinne des MVC-Pattern handelt es sich dabei um zwei Strukturelemente:

- Steuerelement-Attribute, wie z.B. das aktuell ausgewählte Registerblatt oder die aktuell zur Sortierung einer Tabelle ausgewählte Spalte, liegen in einem *ModelNodeDescription*-Datenobjekt ebenfalls in der Session.
- Die *ViewInterpretation* liest den Zustand von *Model*- und *View*-Daten und erzeugt aus Layout und Daten mit Hilfe einer Templating-Engine Ausgabeformate wie HTML oder PDF.

Smalltalk kennt vier verschiedene View-Idiome [KrPo88, 26]:

- Textparagraphen (paragraphs of text)
- Textlisten (Menues) (lists of text)
- Auswahlbuttons (choice buttons)
- Grafische Formulare (graphical forms)

Auf diesen können drei grundlegende Nutzer-Interaktions-Paradigmen ausgeführt werden [KrPo88, 26]:

- Blättern (browsing)
- Inspizieren (inspecting)
- Editieren (editing)

Es werden Komponenten für Nutzerschnittstellen entworfen, die diese Idiome und Paradigmen einmalig behandeln und von der kompletten Programmumgebung genutzt werden können. Dadurch müssen einzelne Dialoge nicht mehr geschrieben werden. Es genügt, diese zu beschreiben, da die Repräsentation des Model durch Interpretation zu Grunde liegender Information erzeugt wird.

8.2.2.3 Vergleich MVC-Modell und PAC-Modell

Beide Ansätze, sowohl das MVC-Modell als auch das PAC-Modell, basieren auf der Trennung von Applikation und Schnittstelle. Sie stellen jeweils objektorientierte Modelle eines UIMS dar. Die Ansätze unterscheiden sich jedoch darin, wie die Synchronisation der einzelnen Teilnehmer hergestellt wird und welches Objekt verantwortlich für Ein- und Ausgabe ist [HuCa95, 25]. Ein ausführliches Beispiel zur Realisierung eines *Interactor* in beiden Modellen findet sich bei HUSSEY & CARRINGTON [HuCa95].

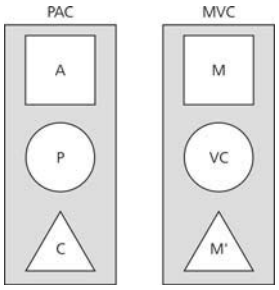

Abbildung 8.5: PAC-Modell und korrespondierende MVC Komponenten [BaCo91, 172]

Aufteilung der Funktionalität

Hauptunterschied zwischen den Modellen von PAC und MVC ist die Art und Weise der Aufteilung verschiedener Funktionalitäten [BaCo91, 170]. Ein *Model* in MVC entspricht der *Ab-*

straction in PAC: beide repräsentieren medienunabhängige Funktionalität. In MVC ist die *View* verantwortlich für den Output und *Control* wiederum behandelt den Input [KrPo88, 27]. Damit entsprechen MVC-View und MVC-Controller zusammen der *Presentation* in PAC [BaCo91, 171]. Die Komponente *Control* in PAC, die zwischen medienunabhängiger Funktionalität und medienabhängiger Perspektive vermittelt, hat kein direktes Pendant im MVC-Modell. BASS & COUTAZ beschreiben sie als eine spezielle Model-Klasse [BaCo91, 171]. Ein Vergleich der Komponenten der beiden Konzepte ist in Abbildung 8.5 dargestellt.

Synchronisation der Teilnehmer

Die Synchronisation der Teilnehmer sichert im PAC-Modell die Komponente Control, indem sie die lokale Semantik (Abstraction) mit der lokalen Syntax (Presentation) verbindet [Cout87, 128]. Anders im MVC-Modell. Dort sind die drei Objekte Model, View und Controller selbst dafür verantwortlich ihre Konsistenz durch Nachrichtenaustausch gemäß dem Observer-Pattern [Gam$^+$95, 293ff] sicher zu stellen. In MVC ist damit die Kontrolle auf drei in Verbindung stehende Objekte verteilt, während er bei PAC explizit zentralisiert ist [Cout87, 128].

Ein- und Ausgabeverhalten

Zwar unterscheidet PAC Funktionen und Präsentation. Doch kapselt es diese in einem Objekt: der Presentation [Cout87, 128]. In diesem sind Eingabe- und Ausgabeverhalten kombiniert. Das MVC-Modell hingegen teilt Eingabe und Ausgabe auf in Controller und View [KrPo88, 26ff]. COUTAZ merkt an, dass es teilweise unmöglich ist, die Trennung von Ein- und Ausgabe zu erreichen, da häufig Eingabeaktionen direkt mit zugehörigen Rückmeldungen (feedback) der Ausgabe verbunden sind [Cout87, 128]. Die Kapselung von Input und Output in der Komponente Presentation beurteilt sie somit als hilfreich. Die Trennung der Eingabe von der Ausgabe erhöht hingegen die Flexibilität und Wiederverwendbarkeit. Die Erscheinung (View) und das Verhalten (Control) können auf verschiedene Arten miteinander kombiniert werden [HuCa95, 26].

Die Realisierung des GUI-Frameworks der **moccabox** ist prinzipiell entsprechend beider Modelle möglich. Das MVC-Modell bietet gegenüber dem PAC-Modell für die vorliegenden Anforderungen verschiedene Vorteile. Das MVC-Modell ist einfacher aufgebaut als das PAC-

Modell. Im Ergebnis bedeutet das weniger komplexe Spezifikationen und einfachere Implementierungen, denn durch das so genannte *model sharing* in MVC ist die unterschiedliche Interpretation ein und derselben Abstraktion möglich. PAC hingegen erlaubt kein *model sharing*. Dadurch können in PAC duplizierte Zustandsinformationen auftreten, die in MVC vermieden werden. [HuCa95, 26]
Die Umsetzung der Schnittstellenfunktionalität in **moccabox** folgt dem MVC-Paradigma. Vor allem die Flexibilität und Möglichkeiten der Wiederverwendbarkeit sprechen dafür, es dem PAC-Modell vorzuziehen.

8.2.3 Software-Ergonomie

Bei der kooperativen Bearbeitung von Aufgaben interagiert der personelle Aufgabenträger als Benutzer mit dem Anwendungssystem über die verfügbare Benutzungsschnittstelle[12]. Bei der Realisierung der zugehörigen Benutzungsoberfläche sind deshalb spezielle ergonomische Aspekte zu beachten. Diese liegen allesamt im Aufgabenbereich des Anwendungsentwicklers und können nicht direkt durch das Framework vorgegeben werden. Die *IO-Basisschicht* muss dennoch die Entwicklung der Oberfläche entsprechend der Anforderungen an diese unterstützen.
Die Disziplin der **Software-Ergonomie** befasst sich demzufolge mit der „[...] Anpassung der Arbeitsbedingungen bei der Mensch-Computer-Interaktion an die sensumotorischen und kognitiven Fähigkeiten und Prozesse des Menschen.“ [Wand93, 1].
Im Folgenden wird ein kurzer Überblick über Software-Ergonomie und Grundlagen der Gestaltung von Benutzungsoberflächen gegeben. Für eine ausführliche Behandlung des Themas wird auf die entsprechende Literatur verwiesen (siehe u. a. [Wand93], [Star96], [Hofm98], [Shne02]). Zu den rechtlichen Grundlagen der Software-Ergonomie siehe u. a. [Eich93]. Kriterien für ergonomische Arbeitsplätze finden sich u. a. in [Blas02].
Die Entwicklung ergonomischer Anwendungssysteme, die sowohl auf Methoden des klassischen Software Engineering als auch auf software-ergonomischen Anforderungen beruht, wird als **Usability Engineering** bezeichnet. Durch die Bereitstellung von Methoden, Modellen und Prinzipien des benutzerzentrierten Entwicklungsprozesses wird auf eine ergonomische Produkt-

[12] IBM führte Ende der 1980er Jahre im Zuge der System-Anwendungsarchitektur (SAA) den *Common User Access* (CUA) ein, der Regeln für die einheitliche und ergonomische Gestaltung von Benutzungsoberflächen enthält (vgl. [Schm90]).

qualität abgezielt [Gimn91, 115]. Benutzbarkeit und Gebrauchstauglichkeit (siehe auch Kapitel 3.4) stellen zentrale ergonomische Qualitätsmerkmale dar [Herd00, 32][13].

Das Framework **moccabox** muss somit die Entwicklung entsprechender Prototypen und schnelle Rückkopplung der Eignung der Benutzungsoberfläche mit der Gruppe der Anwender unterstützen. Die Generierung der Oberfläche darf deshalb keinen großen zusätzlichen Aufwand für die Anwendungsentwicklung bedeuten. Zudem muss der Entwurf und die Umsetzung des Layout und Strukturierung der Oberfläche unabhängig von der Anwendung erfolgen können.

8.2.4 Komponenten

Wie in Kapitel 8.2.2.3 beschrieben liegt der *IO-Basisschicht* des Frameworks **moccabox** das MVC-Pattern zu Grunde. Zudem muss die strikte Trennung in Layout, Inhalt und Logik [KeKi01] beachtet werden. Alle drei sind anwendungsspezifisch. Das Layout und Inhalt werden vom IOS der Anwendung spezifiziert, die Logik entsprechend vom VOS. Der konkret darzustellende Inhalt ist das Ergebnis einer Aufgabendurchführung im VOS.

Das Framework bestimmt das anwendungsneutrale Zusammenspiel der jeweiligen Objekte. Es bietet eine Schnittstelle zum IOS der Anwendung und zur *VO-Basisschicht* des Frameworks.

Die Architektur der *IO-Basisschicht* ist in Abbildung 8.6 abgebildet.

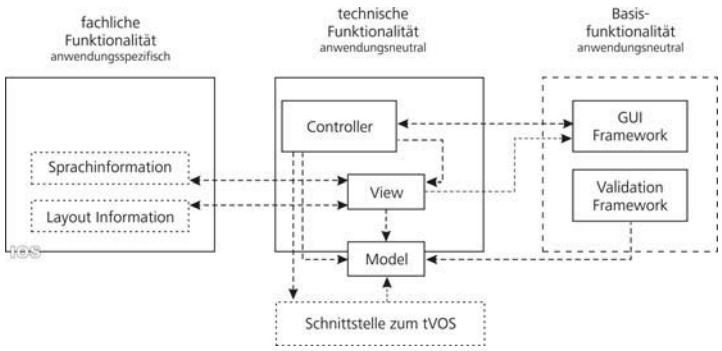

Abbildung 8.6: Komponenten der technischen IO-Schicht und ihre Beziehungen

Controller

Entsprechend der im ooAM definierten Aufgaben des IOS [Ambe93, 123f] nimmt die Komponente **Controller** externe Ereignisse in Empfang und gibt auslösende Ereignisse an die *VO-*

[13] Die genaue Definition der Begriffe ist in DIN-Norm 66272 respektive DIN-Norm 9241-11 festgelegt.

Basisschicht weiter, damit diese die angeforderte Aufgabe ausführen kann. Im Anschluss an die Durchführung kommuniziert der Controller entsprechend der View, dass sich das Model geändert hat.

Model

Die Komponente **Model** repräsentiert den Inhalt, der dargestellt werden soll, und beinhaltet zusätzlich Metadaten über den Inhalt. Metadaten sind definiert als Daten, die andere Daten und Prozesse definieren und beschreiben [ISO03, 10]. Das Model ist hierarchisch aufgebaut und kann Sub-Models enthalten, die in jeweiligen Sub-Views dargestellt werden. Es handelt sich um eine Art Model-Baum, dessen Navigation durch ein Framework im Kleinen [Gamm92, 107-109] realisiert wird.
Beispiel: Die Eingabe von Kundendaten durch einen personellen Anwender erfolgt über ein entsprechendes Formular, das die verfügbaren Kundendaten repräsentiert. Das Model abstrahiert vom zu Grunde liegenden konzeptuellen Objekt `Client` und repräsentiert lediglich die zu befüllenden Attribute. Desweiteren ist das den `Client` repräsentierende Sub-Model abhängig von einem Kontext, in dem es stattfindet, und es kann weitere Sub-Models - so z. B. Kundenadresse - enthalten.

ViewGenerator

Die Komponente **ViewGenerator** kombiniert Inhalt, Angaben zur Repräsentation und Layout-Templates, um eine Darstellung des Inhalts zu erzeugen. Diese Darstellung wird dem Anwender als Ergebnis seiner an den Controller gesendeten Anfrage gesendet. Zur Darstellung des Inhalts bezieht sie sich auf das entsprechende LocaleRepository, um der Anforderung der Mehrsprachigkeit gerecht zu werden.
Beispiel: Der Anwender möchte eine Instanz des konzeptuellen Objektes `Client` ansehen. Das Model enthält alle Attribute und Ausprägungen des Objektes sowie Informationen über verfügbare Methoden (z. B. *Kunde bearbeiten* und *Kunde löschen*). Die Darstellung erfolgt entsprechend einem durch Templates vorgegebenen Layout und unter Verwendung der Bezeichner des LocaleRepository.

LocaleRepository

Die Komponente **LocaleRepository** enthält die Bezeichnungen für die anwendungsspezifischen Interface Objekte, konzeptuellen Objekte und Vorgangsobjekte sowie für deren Attribute in der jeweiligen Sprache und Domäne. Es enthält zudem verschiedene Formate (*patterns*) für die Eingabe von Datum und Zahlen. Metadaten über die entsprechenden Objekte und Aufgaben, die sie repräsentieren, können im LocaleRepository gespeichert werden (u. a. Dokumentation). Beispiel: Das konzeptuelle Objekt `Client` kann je nach eingesetzter Domäne als 'Kunde' oder auch als 'Mandant' bezeichnet werden. Im Englischen könnte die Bezeichnung dementsprechend 'Customer' bzw. 'Client' lauten. Da die Semantik innerhalb der Anwendung die gleiche ist, wird die Umschreibung lediglich für die Mensch-Computer-Kommunikation benötigt.

Schnittstelle zur IOS der Anwendung

Die *IO-Basisschicht* ist direkt dem IOS der Anwendung zugeordnet. Sie dient ihr als Basismaschine für anwendungsneutrale Funktionalität. Gemäß dem Prinzip der *Inversion of Control* [FaSc97, 32] bestimmt die *IO-Basisschicht* den Ablauf der Verarbeitung des Aufrufs und greift auf die anwendungsspezifischen Informationen des IOS zu.

Schnittstelle zur *VO-Basisschicht*

Da die Schnittstellenfunktionalität die externen Ereignisse aufnimmt und diese als auslösende Ereignisse an die entsprechenden Vorgangsobjekte weiter gibt, hat die *IO-Basisschicht* eine Schnittstelle zur *VO-Basisschicht*. Der Controller der IO-Schicht kommuniziert mit dem Controller der VO-Schicht. Beide tauschen das medienneutrale Objekt Model aus.

8.3 Technische Funktionalität für Vorgangsobjekte

Ergebnis der Vorgangs-Objektmodellierung ist die umfassende und objektorientierte Spezifikation der Vorgangs-Funktionalität eines Anwendungssystems in Form des Vorgangsobjektschemas (VOS). Dieses wird im ooAM auf die VOS-Komponente [Ambe93, 71-94] abgebildet. Das Framework stellt der VOS-Komponente entsprechende Basisfunktionalität zur Verfügung,

um die Abbildung anwendungsspezifischer Funktionalität auf korrespondierende anwendungsneutrale Komponenten zu erleichtern. Die *VO-Basisschicht* realisiert zudem die Schnittstellen der VOS-Komponente zu den anderen Komponenten des ooAM. Dadurch kann während der Systementwicklung ein Großteil der Programmierung vermieden werden.

8.3.1 Das Vorgangsobjekt

Das **Vorgangsobjekt** stellt mit seinen vorgangsspezifischen Eigenschaften die Vorgangs-Funktionalität des Anwendungssystems bereit [Ambe93, 35]. Die Aufgabe des Vorgangsobjektes *Führe Vorgang aus* lässt sich in fünf Teilaufgaben untergliedern:

1. **Vorereignis liegt vor:** Das Vorgangsobjekt wird ausgelöst
2. **Aufgabenobjekt laden:** Das Vorgangsobjekt erstellt eine lokale Kopie des Aufgabenobjekts, das Grundlage seiner Aufgabendurchführung ist.
3. **Aufgabenobjekt manipulieren:** Das Vorgangsobjekt plant und durchläuft die Aktionen seiner Aufgabendurchführung auf der lokalen Kopie seines Aufgabenobjekts und eventuell unter Delegation an weitere Vorgangsobjekte.
4. **Aufgabenobjekt speichern:** Das Vorgangsobjekt veranlasst das Zurückschreiben des Aufgabenobjekts, so dass die Datenbasis in einen konsistenten Zustand überführt wird.
5. **Nachereignis auslösen**: Das Vorgangsobjekt teilt nach Durchführung aller Aktionen seinen Zustand mit und gibt die Kontrolle zurück an die Kontrollkomponente.

Je nachdem, ob das Vorgangsobjekt Änderungen an der Datenbasis ausführt oder nicht, sind die Schritte 3. und 4. nicht zwingend zu durchlaufen. Dies kann zum Beispiel der Fall sein, wenn die Aufgabe lediglich im Auslesen und Verfügbarmachen von Informationen besteht. Oder zur weiteren Bearbeitung eine Eingabe des personellen Aufgabenträgers notwendig ist, die eine Unterbrechung der Bearbeitung erfordert.

8.3.2 Komponente zur Vorgangssteuerung

Die **Vorgangssteuerung** wird vom Controller der *IO-Basisschicht* über das Vorliegen eines eingehenden Ereignisses benachrichtigt und beauftragt, den entsprechenden Vorgang zur Durchführung

der korrespondierenden (Teil-)Aufgabe auszuführen.
Die verfügbaren Vorgangsobjekte (siehe Kapitel 8.3.1) sind im zentralen Repository (siehe Kapitel 8.1) hinterlegt und werden dort angefordert. Für ein eingehendes Ereignis muss für die erfolgreiche Durchführung mindestens ein Vorgangsobjekt vorliegen. Alle Vorgangsobjekte werden in einen Speicher (*Stack*) gelegt und der Reihe nach abgearbeitet. Wird für das eingegangene Ereignis kein korrespondierendes Vorgangsobjekt gefunden, bricht die Durchführung ab und eine Fehlermeldung wird zurückgeliefert.
Liegt die Instanz des für die Nachricht verantwortlichen Vorgangsobjektes vor, so wird dieses mit der Durchführung beauftragt (*invoke*).
Nachdem das Vorgangsobjekt ausgeführt wurde und seine (erfolgreiche) Ausführung kommuniziert hat, erhält die Kontrollkomponente die Kontrolle zurück. Liegen noch weitere Vorgangsobjekte im Speicher, werden diese ebenfalls ausgeführt (*invoked*). Nach Bearbeitung aller Vorgänge, muss die Kontrollkomponente anhand der Informationen des Repository entscheiden, welcher Vorgang auf das erhaltene Nachereignis zu folgen hat und welches Ergebnis der *IO-Basisschicht* übergeben ist. Anschließend wird die Kontrolle an den Controller des *IO-Basisschicht* zurückgegeben.

8.3.3 Komponente für die Sicherung der Integrität

Dem Anwendungssystem liegt eine Gesamtaufgabe zu Grunde, die im Aufgabenobjekt ein Modell der Diskurswelt enthält [FeSi06, 227]. Das Integrationsmerkmal der Konsistenz nimmt Bezug auf dieses Aufgabenobjekt. Die Konsistenz des Systems wird durch die Verfolgung des Integrationsziels Integrität sicher gestellt. In diesem Kapitel wird die Komponente vorgestellt, die innerhalb der **moccabox** für die Einhaltung der in Kapitel 4.1 beschriebenen **semantischen** und **operationalen Integrität** zuständig ist. Um die Integrität des Systemzustands testen zu können, werden **Integritätsbedingungen** formuliert, die mit der hier vorgestellten Komponente überprüft werden.
Das Anwendungssystem ist zudem nur dann konsistent, wenn die für sich genommen valide Zustandsänderung (a) von einem autorisierten Nutzer beauftragt und (b) das ausführende Vorgangsobjekt zum Zugriff auf das Aufgabenobjekt berechtigt ist. Die zuständigen Komponenten werden für (a) in Kapitel 8.3.5 und für (b) in Kapitel 8.3.4 vorgestellt.

Es werden statische und transitionale semantische Integritätsbedingungen unterschieden. Erstere geben gültige Anfangs- und Endzustände des Systems an. Letztere beschreiben die gültigen Zustandsübergänge. Demzufolge kann ein Zustandsübergang auch dann die Integrität und folglich die Konsistenz des Systems verletzen, wenn zwar Anfangs- und Endzustand gültig sind, der Übergang aber nicht zulässig ist.

Semantische Integritätsbedingungen

Da die konzeptuellen Objekte den Zustand der konsistent zu haltenden Hilfsregelstrecke repräsentieren, werden die **semantischen Integritätsbedingungen** auf diese angewendet. Eine Integritätsbedingung wird durch eine entsprechende **Condition** repräsentiert, die im Bezug auf die zu Grunde liegenden konzeptuellen Objekte auf ihre Gültigkeit hin überprüft werden kann. Die jeweiligen Bedingungen (*conditions*) werden im **Repository** hinterlegt und falls nötig dort auch angepasst. Sie werden durch Prüfklassen auf die jeweiligen KOT angewandt.
Statische Integritätsbedingungen spezifizieren Zustände, die nach der Durchführung von Aktionen am Ende eines Vorganges für die Instanzen der konzeptuellen Objekttypen erfüllt sein müssen und werden deshalb im Repository dem jeweiligen KO zugeordnet. Sobald sich der Zustand eines KOT ändert, werden alle zugeordneten Integritätsbedingungen überprüft. Sobald eine Bedingung verletzt wird, werden keine Änderungen zurückgeschrieben, sondern die Ausführung des Vorgangs wird abgebrochen. Die Komponente unterstützt die folgenden Typen Integritätsbedingungen:

Einschränkung von Datentypen: Jedes Attribut eines KOT wird durch seinen Typ (z. B. Zahl, Datum, Text) auf bestimmte Ausprägungen eingeschränkt. Innerhalb dieses impliziten Wertebereiches können die Attributausprägungen durch weitere Angaben zum Wertebereich (engl. *codomain*) eingeschränkt werden. Dabei kann es sich um generische Einschränkungen handeln (z. B. Gültigkeit von E-Mail-Adressen oder Datumsformat) oder um anwendungsspezifische Angaben, die Teil der Domänenabgrenzung sind (z. B. Mindestalter eines Mitarbeiters oder maximales Gehalt). Der Wertebereich wird zusammen mit der Definition des KOT erfasst.

Voreinstellung von Werten: Für neu instantiierte konzeptuelle Objekte können initiale Standardwerte angegeben werden, die dem entsprechenden Attribut als Ausprägung gegeben

werden.

Angabe von Schlüsseln: Ein KO muss über einen Schlüssel für den Nutzerzugriff identifizierbar sein. Dies kann ein fachlicher Schlüssel sein (z.B. Vorname/Nachname des Mitarbeiters). Zusätzlich kann für jede Objektinstanz ein so genannter Surrogatschlüssel erzeugt werden, der die eindeutige Identifizierung ermöglicht. Das Objektmodell nutzt als Referenz für Objekte in Beziehungen den systemspezifischen Objektidentifikator (vgl. [Ambe93, 101]). Der hier verwendete Ansatz verfolgt die Nutzung des generierten Surrogatschlüssels auch als Objektidentifikator in Beziehungen zwischen den KO. Die Komponenten sind hierauf ausgerichtet.

Referentieller Integritätsbedingungen: Das konzeptuelle Objektschema bildet die referentiellen Abhängigkeiten der KOT untereinander ab. Damit diese nicht verletzt werden müssen unzulässige Änderungen entweder verhindert werden oder mit einer entsprechenden Kompensationsstrategie ausgeglichen werden (z. B. Löschweitergaben). Desweiteren erlaubt die Komponente die Angabe von Kardinalitäten für die Beziehung der konzeptuellen Objekte untereinander (z. B. eine Abteilung hat mindestens einen, aber maximal vier Mitarbeiter), die bei der Passivierung oder dem Löschen von Objekten überprüft werden. Ein Beispiel dieser Integritätsprüfung findet sich bei [Piet04, 59f].

Bedingungen über mehrere Relationen: Weitere komplexe Bedingungen, die sich eventuell über mehrere Instanzen von KOT erstrecken werden in eigenen Prüfklassen abgebildet.

Transitionale Integritätsbedingungen: Dabei handelt es sich um Bedingungen, bei denen die Zulässigkeit eines Systemzustands von der Ausprägung eines anderen Wertes abhängt. Diese Bedingungen können bei der Definition des konzeptuellen Objekttyps angegeben und später überprüft werden.

Zur Überprüfung der Validität werden Validierungsregeln angegeben, die von entsprechenden Klassen (Validatoren) getestet werden.

Operationale Integritätsbedingungen

Bei der Sicherstellung der **operationalen Integrität** wird auf die implementierte Transaktionsverwaltung vorhandener DBVS zurückgegriffen. Lange Transaktionen und Lost Update werden

in zwei Transaktionen aufgeteilt [Piet04, 43]. Durch optimistisches Verfahren werden mögliche Konflikte und ungültige Änderungen verhindert.

Weitere Integritätsbedingungen betreffen beispielsweise die Autorisierung des zugreifenden VOT auf die zu ändernden KOT. In SOM werden diese Bedingungen durch *interacts-with*-Beziehungen zwischen VOT und KOT dargestellt. Der Zustand eines Systems gilt auch dann als inkonsistent, wenn ein nicht autorisierter Nutzer eine ansonsten gültige Zustandsänderung im System durchführt.

8.3.4 Integritätsmonitor

Für die Entwicklung eines betrieblichen Anwendungssystems ist es notwendig, den automatisierbaren Teilausschnitt des betrieblichen Informationssystems in einem Modell abzubilden. Dieses Modell wird als **Hilfsregelstrecke** verwendet (siehe Abbildung 8.7). Hilfsregelstrecken bilden die Struktur und das Verhalten der Regelstrecke nach. Die Struktur wird durch eine Menge von Attributen beschrieben. Das Verhalten wird realisiert durch geeignete Operatoren auf diesen Attributen. [FeSi06, 29f]

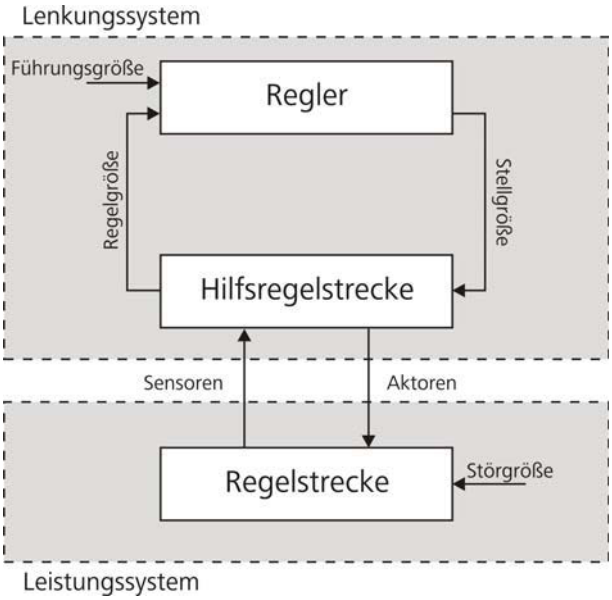

Abbildung 8.7: Betrieblicher Regelkreis mit Hilfsregelstrecke (in Anlehnung an [FeSi06, 77].

Um stets einen konsistenten Zustand der Hilfsregelstrecke zu erreichen, ist der permanente Abgleich mit der zu Grunde liegenden Regelstrecke notwendig. Die für das reale System

geltenden Regeln bezüglich Konsistenz, erlaubter Zustände und Kopplung sowie impliziter Integritätsbedingungen müssen im Anwendungssystem realisiert werden können.
In SOM wird der Regler durch Vorgangsobjekte abgebildet, die Hilfsregelstrecke durch entsprechende konzeptuelle Objekte mit ihren Attributen und Operatoren.
Keines der Vorgangsobjekte hat eine globale Sicht auf das System, sondern stets nur eine lokale. Deshalb ist es notwendig, eine Instanz einzuführen, die die korrekte Zielverfolgung auf globaler Ebene überwacht. Diese Aufgabe soll durch das Repository (siehe Kapitel 9.3) wahrgenommen werden. Aufgaben des Meta-Systems:

- Verhinderung inkonsistenter Zustände (Veto)
- Kopplung von Anwendungssystem und Modulen
- Überwachung impliziter Integritätsbedingungen
- Wiederherstellung von Konsistenz

Die Konsistenz einer Datenbasis kann sich sowohl auf ihren Inhalt (statische Konsistenz) als auch auf die Inhaltsänderung (dynamische Konsistenz) beziehen [Wint98, 24].
Neben der „passiven“ Zurückweisung unzulässiger Datenmanipulationen können dynamische Konsistenzbedingungen auch zur „aktiven“ Herstellung eines konsistenten Zustandes genutzt werden [Wint98, 25].

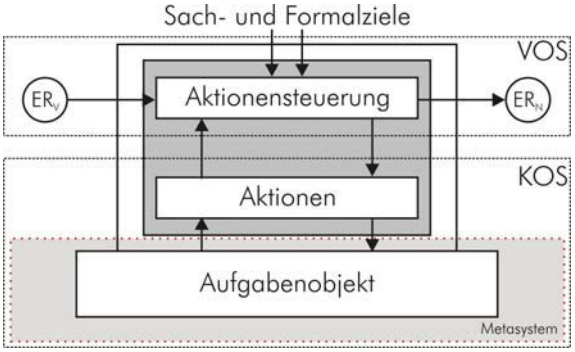

Abbildung 8.8: Aufgabe und Metasystem (in Anlehnung an [FeSi06, 98].

In einem Repository muss hinterlegt werden, welchen Vorgangsobjekten der Zugriff auf welche Aktionen eines konzeptuellen Objektes erlaubt ist. Zusätzlich kann erfasst werden, welche weiterführenden Aktionen zur Koppelung durchgeführt werden müssen bzw. welche Konsistenzüberprüfungen stattfinden. Quelltext 9.3 zeigt den Ausschnitt eines XML-Dokumentes, in

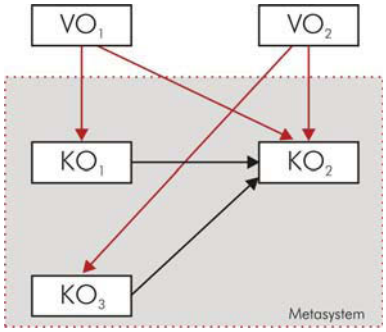

Abbildung 8.9: Zugriff von VO auf KO und Metasystem.

dem diese Informationen erfasst sind. Der Integritätsmonitor wird im Framework als Aspekt realisiert.

Eine Inkonsistenz des Zustands eines KO kann entweder durch ungültige Attributwerte entstehen oder wenn der Aufgabenträger nicht berechtigt ist, die Zustandsänderung auszuführen.

Die Informationen werden dem Entwickler zur Entwicklungszeit verfügbar gemacht und zur Laufzeit vom System überprüft.

8.3.5 Die Sicherheitskomponente

Personelle Aufgabenträger führen teilautomatisierte Aufgaben in Kooperation mit dem betrieblichen Anwendungssystem durch. Jedem Vorgang ist ein Recht zugeordnet, das dem Besitzer den Zugriff darauf gestattet. Die Autorisierung der Vorgangsdurchführung übernimmt die **Sicherheitskomponente**.

Rollenbasiertes Rechtemodell

Jeder Nutzer besitzt eine Teilmenge der gesamten Rechte für alle Vorgänge. LAMPSON schlägt für die Zuordnung von Subjekt (Nutzer) zu Recht eine Zugriffs-Matrix vor [Lamp74]. Im Ergebnis hieße diese Zuordnung aber einen großen Verwaltungsaufwand in Kauf nehmen, auf Grund der entstehenden Komplexität (Komplexität $O(|Subjekte| \times |Objekte|)$). Hier wird die **Role Based Access Control** (RBAC)[14] des NIST[15] eingesetzt [Goo+02, 306]. Bei der rollenbasierten Zugriffskontrolle werden Nutzer Rollen zugeordnet entsprechend ihrer Stellung innerhalb der Organisation. Den Rollen werden entsprechende Rechte gestattet. Daraus ergibt

[14] `http://csrc.nist.gov/rbac/`
[15] National Institute of Standards and Technology (`http://csrc.nist.gov`)

sich eine geringere Komplexität: $O(|Subjekte| \times |Rollen|) + O(|Rollen| \times |Objekttypen|)$. Für die Administration muss - außer bei der Initialisierung des Systems - fast ausschließlich die Rollenzuordnung der Nutzer verwaltet werden.
Die RBAC sieht keine Rechtezuordnung für die Instanz eines Objekttyps zu einer Rolle vor (z. B. Mitarbeiter dürfen Projektdaten nur ändern wenn sie Mitarbeiter des Projektes ist). Diesen Mangel behebt die Sicherheitskomponente, indem für die bestimmten Rechte auch Einschränkungen bezüglich der Instanzen hinterlegt werden können.

Authentisierung und Autorisierung

Ausgehend von den zugestandenen Rechten muss bei jedem Zugriff auf ein Vorgangsobjekt überprüft werden, ob der Nutzer dazu berechtigt ist. Die Sicherheitskomponente übernimmt zwei Aufgaben. Sie **authentisiert** den Nutzer und **autorisiert** seine Zugriffe im System.
Für die **Authentisierung** des Nutzers im System bieten sich verschiedene Verfahren an. Die **moccabox** bietet mit der Sicherheitskomponente eine Infrastruktur, die eine Wiederverwendung der Nutzerverwaltung ermöglicht. Diese kann selbstverständlich auch durch eine andere Komponente oder das zu entwickelnde Anwendungssystem selbst realisiert werden. Eine konkrete Implementierung inklusive verwendeter Basismaschinen wird in Kapitel 9 vorgestellt. Die Anmeldung des Nutzers am System erfolgt über die Eingabe von Benutzerkennung und Passwort. Versucht ein nicht authentisierter Nutzer auf das System zuzugreifen, wird die Ausführung der Anfrage verweigert und der Nutzer wird aufgefordert, sich zu authentisieren.
Vor jeder Durchführung eines Vorgangs erfolgt die **Autorisierung** des authentisierten Nutzers. Es wird überprüft, ob der Nutzer das Recht hat, den angeforderten Vorgang auszuführen. Sind an das Recht zur Ausführung eines Vorgangs (z. B. Projektdaten ändern) Bedingungen geknüpft (z. B. muss Mitarbeiter im Projekt sein), so werden diese zusätzlich anhand der Instanz des Aufgabenobjekts überprüft. Sind mit dem Recht keine Bedingungen verbunden, so ist der Nutzer in jedem Fall berechtigt, den Vorgang auszuführen. Erst nach positiver Überprüfung der Berechtigung wird der eigentliche Vorgang ausgeführt. Andernfalls wird die Ausführung wie im Falle nicht vorhandener Authentisierung abgebrochen.
Die Sicherheitskomponente enthält eine Schnittstelle, die von der *IO-Basisschicht* genutzt wird, um für einen authentisierten Nutzer die Teilmenge der erlaubten Operationen auf einem an-

geforderten Objekt zu bestimmen. Da ein Interface Objekt dem Nutzer die Operatoren auf den repräsentierten Vorgangsobjekten sowie konzeptuellen Objekten anbietet, wird dadurch verhindert, dass ein Nutzer Funktionen angeboten bekommt, die er nicht nutzen darf und die bei der anschließenden Autorisierung zum Abbruch des Prozesses führen würden.

8.4 Technische Funktionalität für konzeptuelle Objekte

Die konzeptuelle Funktionalität eines Anwendungssystems wird gemäß dem ooAM als Ergebnis der konzeptuellen Objektmodellierung durch ein anwendungsspezifisches konzeptuelles Objektschema (KOS) spezifiziert [Ambe93, 95-118]. Analog zu den bereits beschriebenen Schichten *IO-Basisschicht* und *VO-Basisschicht* realisiert die *KO-Basisschicht* des Frameworks alle anwendungsneutrale Funktionalität für das konzeptuelle Objektschema. Zur Funktionalität zur Sicherung der Integrität siehe auch Kapitel 8.3.3.

8.4.1 Persistenzkomponente

Ziel der Anwendungsentwicklung ist unter anderem die Unabhängigkeit des Systems vom Aufgabenträger, also der grundlegenden Basistechnologie (siehe hierzu auch Kapitel 3.4 und Kapitel 4.1). Unter **Persistenz** versteht man allgemein das Schreiben von Daten auf ein beständiges Medium bzw. das Lesen davon [Star05, 161]. Die Persistierung der konzeptuellen Objekte muss aus Sicht der Anwendung bzw. des Frameworks unabhängig vom gewählten Speicherformat (z. B. relationale Datenbank, objektorientierte Datenbank, Dateisystem) und dem konkreten System (z. B. Oracle, Firebird) erfolgen. Das Framework stellt den Zugriff auf persistierte Objekte und deren Zurückschreiben in die Datenbasis bereit.

8.5 Dokumentation des Anwendungssystems

Obwohl die **Dokumentation** nur ein sekundäres Ergebnis des Entwicklungsprozesses ist [Booc94, 351], kommt ihr doch eine wichtige Rolle zu [Alk+03, 228]:

> „Die Dokumentation soll in diesem Kontext idealerweise durchgängig, nachvollziehbar und konsistent über alle Phasen der Softwareerstellung hinweg sein."

Die Komponenten des Frameworks **moccabox** sollen eine durchgängige und komfortable Dokumentation des Systems ermöglichen, so dass sie direkt während des Entwicklungsprozesses entsteht und zudem dem Anwender als Direkthilfe angeboten werden kann. Dabei soll die Dokumentation nicht als Handbuch entstehen, sondern Teil des Anwendungssystems sein. Die Gestaltung von Dokumentation und ihre Aufbereitung für den Anwender wird hier nur insofern betrachtet, als sie für die Entwicklung des Frameworks **moccabox** von Belang sind. Im Modellierungsprozess müssen Dokumentation und Kommentierung der entsprechenden Objekte Teil des Modellierungyswerkzeuges sein, um diese im Anschluss bei der Implementierung in das Anwendungssystem übernehmen zu können. Der Umfang der Systementwicklungsaufgabe kann dadurch entsprechend verringert werden.

9 Software-technische Realisierung des Frameworks moccabox

Das Framework selbst kann aus Außensicht als Nutzermaschine gemäß dem Modell von Nutzer- und Basismaschine interpretiert werden. Der software-technische Entwurf des Frameworks **moccabox** wurde unter Verwendung verschiedener Basismaschinen realisiert. Die geringen technischen Produkthalbwertzeiten von Basismaschinen im Software-Bereich im Allgemeinen erzeugen einen hohen Anpassungsdruck auf den software-technischen Entwurf [Schm01, 261]. Da Konzepte also stabiler sind als Technologien [StVö05, 22], muss die hier beschriebene Realisierung als nur eine mögliche Lösung des im vorangegangenen Kapitel beschriebenen fachlichen Entwurfs gesehen werden. Abhängigkeiten zu konkreten Basistechnologien (z. B. GUI-Framework, Persistenz-Framework, Sicherheit) wurden dadurch vermieden (vgl. [Star05, 133ff]), dass jeweils ein `Interface` implementiert wurde, das für die jeweilige Basistechnologie als Standardimplementierung realisiert wurde (vgl. Abbildung 9.1).

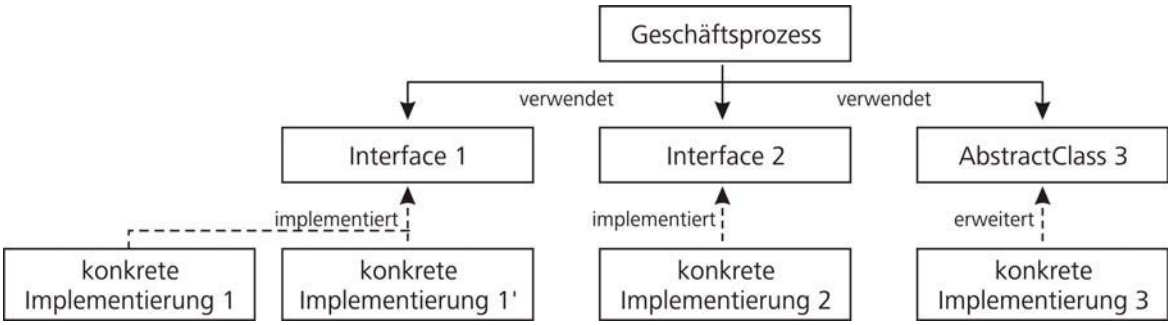

Abbildung 9.1: Unabhängigkeit durch Interfaces [Star05, 134]

In den folgenden Kapiteln werden einzelne System-Komponenten unter Verwendung der in Kapitel 9.1 beschriebenen Basismaschinen dargestellt.

Quellcode des Frameworks

Der Arbeit liegt ein elektronischer Anhang bei, der sowohl den Quellcode des Frameworks **moccabox**[1] als auch die korrespondierende IO-Implementierung[2] für web-basierte Anwendungssysteme umfasst. Die Dokumentation[3] der einzelnen Klassen ist ebenfalls Teil des elektronischen Anhangs und kann im Browser navigiert werden.
Der Quellcode ist, wie in Abbildung 9.2 dargestellt, entsprechend der Projektstruktur einschließlich der zugehörigen *Eclipse*-Projektdateien[4] abgelegt. In den folgenden Kapiteln wird von den vorgestellten Komponenten auf das jeweilige Quellcode-Paket der Implementierung verwiesen. Sofern nicht anders angegeben, bezieht sich der Hinweis auch auf die strukturgleiche Dokumentation.

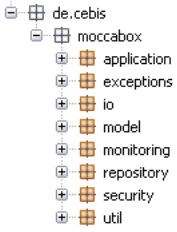

Abbildung 9.2: Das Paket de.cebis.moccabox im Anhang.

Die im Folgenden verwendeten qualifizierten Namen verweisen auf die im Anhang befindlichen Klassen im jeweiligen Paket. So befindet sich die im Text referenzierte Klasse `de.cebis.moccabox.repository.impl.RepositoryImpl` als Datei `RepositoryImpl.java` im Ordner *anhang/quellcode moccabox/src/main/java/de/cebis/moccabox/repository/impl* im elektronischen Anhang. Die Dokumentation der Klasse selbst sowie ihre hierarchischen Beziehungen zu Interfaces und anderen Klassen lässt sich über *anhang/dokumentation moccabox/index.html* einsehen und nachvollziehen, wie in Abbildung 9.3 dargestellt.

[1] Ordner *anhang/quellcode moccabox*.
[2] Ordner *anhang/quellcode io implementierung*.
[3] Datei *anhang/dokumentation moccabox/index.html*.
[4] *Eclipse* ist ein Programmierwerkzeug zur Entwicklung von Software. Mehr Informationen zu *Eclipse* finden sich unter `http://www.eclipse.org/`.

Abbildung 9.3: Dokumentation der Klasse RepositoryImpl und ihrer Beziehungen im Anhang.

9.1 Auswahl der Basismaschinen

Zunächst werden die grundlegenden, querschnittlich verwendeten Basismaschinen Java und XML vorgestellt. Weitere eingesetzte Basismaschinen und Frameworks werden in den jeweiligen Kapiteln der sie einsetzenden Komponenten vorgestellt.

9.1.1 Java

Java™ (siehe auch [Flan05]) von Sun Microsystems Inc. hat sich in den letzten Jahren zu einer der wichtigsten objektorientierten Programmiersprachen und Plattform für Anwendungssysteme entwickelt [Reed02, 27-48]. Java-Programme nutzen zur Laufzeit als Basismaschine eine virtuelle Maschine [FeSi06, 320], die als Java-Plattform bezeichnet wird. Der Java-Quellcode wird vom Java-Compiler nicht in Maschinencode übersetzt, sondern in maschinenneutralen Byte-Code, der anschließend von einer Java-Laufzeitumgebung (virtuelle Maschine) interpretiert wird [DePe02, 21]. Java-Programme sind somit ohne neu kompiliert zu werden auf jedem Rechnersystem lauffähig, das über eine Implementierung der abstrakten Maschine verfügt. Dies wird als „write once, run anywhere“ Prinzip bezeichnet und erfüllt damit die Anforderung der Plattformunabhängigkeit (vgl. Kapitel 4.1) des Programmcodes. Für den Entwickler stellt sich die Java Plattform als abstrakte Maschine mit virtuellen Betriebsmitteln (vgl. [FeSi06, 320f]) dar [Schm01, 288].

9.1.2 Extensible Markup Language (XML)

Die **Extensible Markup Language** (XML) ist eine Meta-Auszeichnungssprache, die es erlaubt, eigene Auszeichnungssprachen zu definieren[5] (u. a. [HaMe04]). Sie stellt eine Teilmenge der als ISO 8879 standardisierten **Standard Generalized Markup Language** (SGML) dar [Turo99, 136].

Die in diesem Kapitel vorgestellten XML-Dokumente und XML-Fragmente zur Parametrisierung der Anwendung stellen Konzepte des Ziel-Metamodells im Sinne der modellgetriebenen Software-Entwicklung [Völt05, 7] dar. Entsprechende Änderungen am Modell müssen dann nur in diesen (idealerweise durch Generatoren automatisch) nachvollzogen werden und nicht im Programmcode auf den verschiedenen Ebenen (GUI, Anwendungsfunktionalität, Persistenz-Ebene) [Völt05, 11].

9.2 Standardisierte & erweiterbare Anwendungsschicht

Ziel ist die Entwicklung einer Java-Schnittstelle, über die die Funktionalität der Vorgangsobjekttypen aus dritter Ebene SOM aufgerufen wird. Es werden generische Klassen abgeleitet, die zum Zwecke der Wiederverwendung einen Großteil standardisierbarer VO-Aktionen abbilden. Die Wiederverwendung dient einerseits der Reduktion von Redundanz. Andererseits wird eine höhere Qualität der Software erreicht, da die Vorgangsobjekte jeweils nur beschrieben und anschließend interpretiert werden.

Ein solcher Task ist der `SingleConceptCollectTask`[6], der Funktionalität zum Auslesen eines bestimmten konzeptuellen Objektes anbietet. Die Konfiguration des Task findet innerhalb der verwendenden moccaparts statt (siehe Kapitel 9.3.1). Weitere Tasks[7] sind `CreateTask`, `DeleteTask`, `RetrieveTask` sowie `UpdateTask`, deren Anwendung und Konfiguration in der Dokumentation des Anhangs beschrieben wird.

[5] Informationen finden sich im Internet unter `http://www.w3c.org/xml`.
[6] `de.cebis.moccabox.application.standard.SingleConceptCollectTask`
[7] Alle im Paket `de.cebis.moccabox.application.impl.hibernate`.

9.3 Repository

Das Herzstück des Lösungsverfahrens bildet ein zentraler Katalog, in dem möglichst umfassende **Metadaten**[8] über ein AwS nach dem Software-Architekturmodell formal abgebildet werden. Dieser Metadatenkatalog wird im Weiteren auch **Repository**[9] genannt und im Sinne der Definition von HABERMANN & LEYMANN als ein System verstanden, das Metadaten über „Struktur und Aspekte des Verhaltens von Systemen, die Daten der Umwelt (als Abbild eines Ausschnitts der Realität) manipulieren“ [Habe93, 17] enthält. Das Repository beinhaltet „Informationen über Objekte der Softwareproduktion (z. B. Programme, Datenfelder, Masken, Listen), deren Beschreibungen und Beziehungen“, verwaltet diese Informationen und stellt sie bereit [Habe93, 15ff]. Das verwendete Repository ist im Sinne von HABERMANN & LEYMANN als aktives Repository realisiert, das heißt es steht zur Laufzeit des Systems als Komponente zur Verfügung und versorgt weitere Laufzeitkomponenten mit Informationen. Der Katalog wird damit zum Bestandteil des ausgelieferten Programmsystems. Folgende Definitionen bilden den Grundstock des Repository (zusammengehörige Elemente werden jeweils in moccaparts zusammengefasst - siehe Kapitel 9.3.1) und werden jeweils durch XML-basierte Ressourcen beschrieben[10]:

- VOT und ihr Workflow
- KOT und Metadaten über sie
- Strukturierung und Definition von View-Elementen zur Generierung der IOT
- Validatoren
- Pakete mit domänen-, landes- und sprachspezifischen Informationen

Die Definition der oben beschriebenen Elemente kann entweder durch die XML-basierte Parametrisierung generischer Java-Klassen erfolgen oder durch spezielle Java-Klassen. Auch die Referenzierung eines externen Service, der die Parametrisierung übernimmt, ist denkbar, jedoch gegenwärtig nicht realisiert.

[8] Metadaten sind Daten über Daten [ISO03, 10].
[9] Siehe `de.cebis.moccabox.repository`.
[10] Vgl. dazu Flexibilität durch Ressourcen in [Pree97, 14f].

Das Paket de.cebis.moccabox.repository

Im Anhang befinden sich die entsprechenden Klassen im Paket `de.cebis.moccabox.repository`, wie in Abbildung 9.4 dargestellt. Es umfasst neben der Definition der Interfaces auch eine entsprechende Implementierung sowohl des `IRepository` als `RepositoryImpl` als auch des `IRepositoryManager` als `RepositoryManagerImpl`. Die einzelnen Elemente zum Einlesen, Abbilden und Verarbeiten von moccapart-Definitionen befinden sich im Paket `de.cebis.moccabox.repository.elements`. Die Beschreibung aller Elemente befindet sich im Ordner *anhang/dokumentation moccabox/de/cebis/moccabox/repository*, dessen Inhalt auch direkt im Browser über *anhang/dokumentation moccabox/index.html* navigiert werden kann.

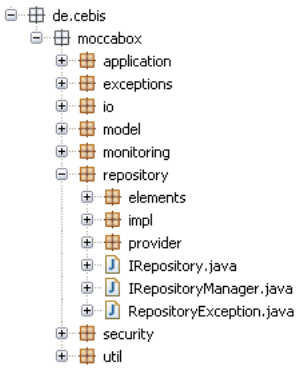

Abbildung 9.4: Das Paket de.cebis.moccabox.repository im Anhang.

9.3.1 moccaparts

In das oben beschriebene Repository können verschiedene Module - so genannte moccaparts - injiziert werden. Diese moccaparts bilden die automatisierten Aufgaben eines oder mehrerer betrieblicher Objekte der 2. Ebene des SOM Ansatzes ab. Sie umfassen die KOT, die darauf operierenden VOT sowie die Beschreibung der KOT und ihrer Operatoren, die zur Generierung entsprechender IOT verwendet werden.

Quelltext 9.1: Definition eines moccapart

```
1    <bean id="MOCCAPART.CALENDAR" class="de.cebis.moccabox.repository
         .provider.MoccaPartDefinitionBean">
```

```
2       <property name="providerFactoryClass"><value>de.cebis.
            moccabox.repository.provider.xmlbeans.
            XMLMoccaPartProviderFactory</value></property>
3       <property name="definitionSource"><value>de/cebis/moccapart/
            calendar/moccapart_calendar.xml</value></property>
4       <property name="conceptDefinitionSources">
5           <list>
6               <value>de/cebis/moccapart/calendar/domain/def/
                    CalendarAssignmentDefinition.xml</value>
7               <value>de/cebis/moccapart/calendar/domain/def/
                    CalendarComponentContainerDefinition.xml</value>
8               <value>de/cebis/moccapart/calendar/domain/def/
                    CalendarTimeUnitDefinition.xml</value>
9               <value>de/cebis/moccapart/calendar/domain/def/
                    CalendarDefinition.xml</value>
10              <value>de/cebis/moccapart/calendar/domain/def/
                    CalendarEventDefinition.xml</value>
11          </list>
12      </property>
13    </bean>
```

Die moccaparts werden registriert und beim Start der Anwendung in die moccabox injiziert.

Quelltext 9.2: Integration mehrerer moccaparts

```
1 <bean id="MOCCABOX" class="de.cebis.moccabox.MoccaBoxApplication"
      init-method="init">
2   <property name="moccaBoxRepository"><ref bean="REPOSITORY"/></
        property>
3   <property name="languageRepository"><ref bean="LIR"/></property>
4   <property name="authorizationService"><ref bean="AUTHORIZATION.
        SERVICE"/></property>
```

```
  <property name="moccaPartProviderDefinitions">
    <list>
      <ref bean="MOCCAPART.PM" />
      <ref bean="MOCCAPART.CALENDAR" />
    </list>
  </property>
</bean>
```

9.3.2 Metasystem

Die Komponente **Metasystem** stellt die semantische Integrität des Anwendungssystems sicher und kontrolliert dessen Systembeziehungen (siehe Kapitel 8.3.4). Die Konfiguration wird in der hier vorgestellten Implementierung über die jeweilige `moccapart.xml` vorgenommen. Listing 9.3 zeigt den Ausschnitt einer Beschreibung des Konzeptzugriffs in XML, der innerhalb des Tag `<monitor>` definiert wird.

Quelltext 9.3: Konfiguration Metasystem

```
<monitor>
  <conceptaccess class="de.cebis.shop.domain.Client">
    <access type="LOAD">
      <granted>*</granted>
      <notify>de.cebis.moccabox.application.ConceptRetrieveLog</
          notify>
    </access>
    <access type="CREATE">
      <granted>client_create</granted>
      <granted>company_create</granted>
      <notify>de.cebis.shop.application.tasks.CalendarCreateTask</
          notify>
    </access>
```

```
12    <access type="UPDATE">
13      <granted>client_update</granted>
14    </access>
15    <access type="DELETE">
16      <granted>client_delete</granted>
17      <notify>de.cebis.shop.application.veto.ClientInvoiceVeto</
            notify>
18      <notify>de.cebis.shop.application.tasks.
            DeleteClientCalendarTask</notify>
19    </access>
20  </conceptaccess>
21 </monitor>
```

Im jeweiligen `moccapart` werden alle Konzepte erfasst, die für die jeweilige Komponente durch das Metasystem zu überwachen sind.

Das Paket de.cebis.moccabox.monitoring

Im Anhang befinden sich die entsprechenden Klassen im Paket `de.cebis.moccabox.monitoring`, wie in Abbildung 9.5 dargestellt. Es umfasst neben der Definition der Interfaces auch eine entsprechende Implementierung des `IMetaSystem` als `MetaSystem`. Die Implementierung bezieht sich auf die für die Persistierung verwendete Basismaschine Hibernate. Die Beschreibung aller Elemente befindet sich im Ordner *anhang/dokumentation moccabox/de/cebis/moccabox/monitoring*, dessen Inhalt auch direkt im Browser über *anhang/dokumentation moccabox/index.html* navigiert werden kann.

Konfiguration des Metasystems

Durch `<conceptaccess>` wird das konzeptuelle Objekt definiert, für das die folgenden Zugriffsbedingungen festgelegt werden. Es werden der lesende Zugriff `LOAD` und die schreibenden Zugriffe `CREATE`, `UPDATE` und `DELETE` unterschieden. Der Zugriff kann bestimmten Vorgangsobjekten (repräsentiert durch das korrespondierende Kommando der Aufgabe) ge-

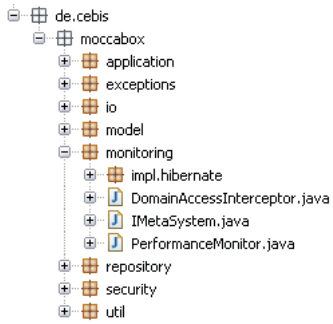

Abbildung 9.5: Das Paket de.cebis.moccabox.monitoring im Anhang.

stattet werden oder per *Wildcard* () allen. Diese Einträge konfigurieren das Metasystem für die Überwachung der Kommunikationsstruktur. Hierdurch wird eine nicht zulässige Änderung der Datenbasis verhindert. Ist ein Vorgangsobjekt nicht für den lesenden oder schreibenden Zugriff auf einen KOT eingetragen, so wird zur Laufzeit die gewünschte Aktion verweigert. Es müssen alle in Abbildung 5.9 durch Pfeile repräsentierten Beziehungen abgebildet werden, damit das Anwendungssystem lauffähig ist. Gleichzeitig ist damit die Konsistenz zwischen Modellierung und Implementierung gegeben. Für eine direkte Ableitung der Konfiguration aus den SOM-Fachspezifikationen müssen die Beziehungen noch mit der Art des Zugriffs spezifiziert werden. Desweiteren können für jede Zugriffsaktion Objekte bestimmt werden, die benachrichtigt werden, um die semantische Integrität des Systems sicherzustellen. Diese werden mit `<notify>` gekennzeichnet.

Ablauf des Metasystems

Das Metasystem wird in **moccabox** durch das Interface `IMetaSystem`[11] repräsentiert. Es kapselt die lesenden und schreibenden Zugriffe auf die Konzepte in einzelnen Methoden. Die Klasse `MetaSystem`[12] implementiert das Interface und stellt die Standardimplementierung für den Zugriff auf die Datenbasis mit der Basismaschine *Hibernate* bereit. Beim Initialisieren des Systems werden - wie in Kapitel 9.3.1 beschrieben - die spezifizierten `moccaparts` injiziert. Die Konfigurationen, die in den jeweiligen `moccapart_x.xml`-Dateien beschrieben sind, werden beim `MetaSystem` registriert. Das Metasystem wird den ausführenden `IBusiness-Task`-Objekten vom Interface `ITaskContext` zur Verfügung gestellt. Es stellt beim Zugriff

[11] `de.cebis.moccabox.monitoring.IMetaSystem`
[12] `de.cebis.moccabox.monitoring.impl.hibernate.MetaSystem`

auf ein KO durch ein VO fest, ob dieser Zugriff gestattet ist und verhindert diesen im negativen Fall.

Bevor das Metasystem die Änderungen an die Datenbasis weiter gibt, werden die zu benachrichtigenden *Listener* benachrichtigt. Diese implementieren das Interface `IMoccaPersistenceListener`[13], das von `EventListener` aus dem Paket `java.util` abgeleitet ist. Beim Aufruf der Methode `call` wird ein Objekt vom Typ `MoccaPersistenceEvent`[14] übergeben, das als Attribute die aufrufende Instanz von `MetaSystem` und das Konzept enthält. Der *Listener* führt anschließend die Aktionen aus, die zur Sicher- oder Wiederherstellung der Konsistenz notwendig sind. Es handelt sich dann um *vetoing Listeners* respektive *updating Listeners*. Ein Veto greift nicht aktiv auf die Datenbasis zu, sondern verhindert nur inkonsistente Systemzustände, indem eine `VetoException`[15] geworfen wird, sobald eine geplante Änderung einen nicht validen Zustand zur Folge hätte. Dabei kennt der *Listener* nur einen ganz bestimmten Ausschnitt der konzeptuellen Datenbasis.

Beispiel:

Der Vorgang `client_delete` lädt die zu löschende Instanz von Typ `Client`. Das Metasystem erlaubt den Zugriff, da alle Vorgänge lesenden Zugriff auf `Client` haben. Nach einer Rückfrage an den Nutzer beauftragt der Vorgang das Löschen der Objektinstanz. Da dem Vorgang `client_delete` dieser Zugriff erlaubt ist, führt die Klasse `MetaSystem` die Löschaktion durch. Es ist ersichtlich, dass lediglich dieser Vorgang das Recht hat, ein Kundenobjekt zu löschen. Soll es auch einem anderen Vorgang gestattet werden, so muss er in die Konfiguration eingetragen werden[16]. Bevor die Objektinstanz gelöscht wird, werden die registrierten *Listener* benachrichtigt. Der `DeleteClientCalendarTask` stellt sicher, dass auch die Kalender des Kunden gelöscht werden. Sind noch Rechnungen des Kunden offen, so verhindert `ClientInvoiceVeto` durch eine `VetoException` das Löschen. Der `TaskManager` führt in diesem Fall ein *roll-back* durch.

[13] `de.cebis.moccabox.application.context.IMoccaPersistenceListener`
[14] `de.cebis.moccabox.application.context.MoccaPersistenceEvent`
[15] `de.cebis.moccabox.application.exceptions.VetoException`
[16] Dieser Eintrag kann auch in einer weiteren `moccapart`-Konfigurationsdatei hinterlegt sein. Das ist nachteilig für die Übersichtlichkeit, aber von Vorteil für die Wiederverwendung. Ein Editor für die Konfiguration des Frameworks - der nicht Bestandteil der vorliegenden Arbeit ist - kann diese Übersichtlichkeit wiederherstellen.

9.4 GUI-Framework für Web-Applikationen

Die Bedeutung des Internet für elektronische Geschäftsbeziehungen (*E-Business*) und den elektronischen Handel (*E-Commerce*) sind kaum zu überschätzen. Da es sich bei Frameworks um eine *enabling technology* handelt, muss ein Framework für die Entwicklung betriebliche Anwendungssysteme diese Potenziale vollends unterstützen [Fay+00, 45]. Im Folgenden wird die Umsetzung einer web-basierten Kommunikationsschnittstelle vorgestellt. Das GUI-Framework dient der Kommunikation mit personellen Aufgabenträgern und ist Teil der Kommunikationsschicht entsprechend dem ADK-Strukturmodell (siehe Kapitel 3.3.2). Weitere Oberflächen wie *Java Swing*[17] oder solche für mobile Endgeräte können für Anwendungen ebenfalls eingesetzt werden[18], sind in der vorliegenden Arbeit aber nicht Gegenstand der Betrachtung. Dem Framework liegt die Trennung von Layout, Inhalt und Logik zu Grunde [KeKi01].
Da Web-Anwendungen zustandslos sind, stellen sie besondere Anforderungen an die Architektur der Präsentationsschicht (siehe auch [CoOs04a, 27]). KOJARSKI & LORENZ beschreiben die Probleme, die bei der Verwendung herkömmlicher statischer und dynamischer Web-Seiten auftreten. Sie bezeichnen diese als *intra-crosscutting* und *inter-crosscutting* [KoLo03].
Für die Mensch-Computer-Kommunikation können dabei folgende Interaktionsstile unterschieden werden (vgl. [Shne02, 96f]):

- direkte Manipulation
- Auswahlmenüs
- Eingabefelder
- Befehlssprache
- natürliche Sprache

Für grafische Oberflächen, die mit dem Framework **moccabox** entwickelt werden, sind die ersten drei Interaktionsstile von Bedeutung und müssen somit vom Web-Framework unterstützt werden.

[17] Die API und Komponenten von *Java Swing* finden sich unter `http://java.sun.com/j2se/1.4.2/docs/api/javax/swing/package-summary.html`.
[18] Vgl. Austauschbarkeit des UIMS bei [KaBa94, 13ff].

Direkte Manipulation

Direkte Manipulation realisiert Aufgaben durch direktes Zeigehandlungen auf grafische Darstellungen von Objekten[19] und Konzepten [Prei99, 122f]. Die Grundprinzipien der direkten Manipulation sind [Shne02, 249]:

- Fortlaufende Darstellung der interessierenden Objekte und Aktionen mit bedeutungsvollen visuellen Metaphern (z. B. *Icons*).
- Physische Aktionen oder Bedienung von Schaltflächen an Stelle komplexer Syntax.
- Schnelle, inkrementelle, umkehrbare Operationen, deren Effekte auf dem Zielobjekt unmittelbar sichtbar werden.

Eine ausführliche Betrachtung der Vor- und Nachteile direkter Manipulation finden sich bei STARY [Star96].

Auswahlmenüs

Bei der Verwendung von Auswahlmenüs kann der Benutzer aus einer vorgegebenen Zahl von Aktionen auswählen, die durch eine vollständige Aufgabenanalyse ermittelt wurde. Grundlagen zum Design von Auswahlmenüs finden sich bei SHNEIDERMAN [Shne02, 96ff].

Eingabefelder

Für die Eingabe und Verarbeitung von Daten werden Eingabefelder verwendet. Die einzelnen Felder werden zu diesem Zweck zu Formularen zusammengefasst, die der Nutzer mit Informationen befüllt [Shne02, 98].

Das Paket de.cebis.moccabox.io

Im Anhang befinden sich die entsprechenden Klassen im Paket `de.cebis.moccabox.io`, wie in Abbildung 9.6 dargestellt. Es umfasst die Definition der Interfaces der Kommunikationsschicht, die zur Kapselung und Verarbeitung von Vorgangsaufrufen durch Anwender

[19] Diese objektorientierte Art der grafischen Benutzungsoberfläche wird auch *Object-Oriented User Interface (OOUI)* genannt [Orf+97, 87ff].

notwendig sind[20]. Zudem enthält das Paket die notwendigen Elemente zur Beschreibung der Sichten (*Views*)[21]. Die Internationalisierung, die in Kapitel 9.8 beschrieben wird, findet sich im Paket `de.cebis.moccabox.io.i18n`. Der Quellcode der Implementierung der Interfaces für das als Basismaschine genutzte Web-Framework *WebWork* liegt im Anhang in Ordner *anhang/quellcode io implementierung/src/main*. Die Beschreibung aller Elemente befindet sich im Ordner *anhang/dokumentation moccabox/de/cebis/moccabox/io*, dessen Inhalt auch direkt im Browser über *anhang/dokumentation moccabox/index.html* navigiert werden kann.

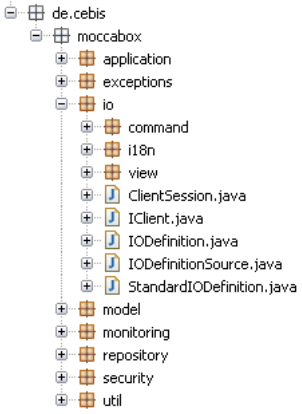

Abbildung 9.6: Das Paket de.cebis.moccabox.io im Anhang.

9.4.1 Architektur des Web-Framework

Die Architektur der Präsentationsschicht (IOS)[22] folgt dem *MVC-Pattern*. Als Basismaschine dient das Web-Framework *WebWork*[23] von *OpenSymphony*[24]. Dabei handelt es sich um ein mächtiges web-basiertes *MVC-Framework*, welches wiederum auf der *API* von *XWork*[25] basiert. *XWork* ist ein Framework für Kommandomuster (*command-patterns*) desselben Herstellers. Weitere Web-Frameworks sind u. a. *Ruby on Rails* [O05b], *Java Server Faces* von Sun [O05c] (JFC) oder *Apache Struts* [O05a].

[20] Siehe auch `de.cebis.moccabox.io.command`.
[21] Siehe `de.cebis.moccabox.io.view`.
[22] Siehe Packages `de.cebis.moccabox.io` sowie die Implementierung `de.cebis.moccabox.kiosk`.
[23] `http://www.opensymphony.com/webwork`
[24] `http://www.opensymphony.com`
[25] `http://www.opensymphony.com/xwork/`

Controller

Der *Controller* wird durch die Klasse `ActionAdaptor`[26] realisiert, indem sie die Superklasse `ActionSupport`[27] aus dem *XWork-Framework* erweitert.
Der `ActionAdaptor` stellt die Controller-Komponente im Web-Framework dar. Als Controller stellt er die Schnittstelle zwischen der zu erzeugenden *View* und der Applikations-Schicht, die durch das *Model* repräsentiert wird, dar. Er ist dafür zuständig zu entscheiden, ob die Applikations-Schicht aufgerufen werden muss, um das *Model* entsprechend einer Aufgabendurchführung neu zu bestücken, oder ob nur dem *Model* ein *Update* mitgeteilt werden muss, um die *View* anzupassen.

Model

Das *Model* wird durch die Baumstruktur der Klasse `ModelNode`[28] realisiert[29]. Sie spezifiziert damit die Schnittstelle zwischen Anwendungsprogramm und dem Anwendungsteil Mensch-Computer-Kommunikation (MCK). Der `ModelNode` definiert die MCK aus Sicht des Anwendungssystems.
Ein ModelNode repräsentiert das Aufgabenobjekt eines Vorgangs sowie die darauf verfügbaren Operationen, die als `TaskCommand`[30] hinterlegt werden. Ein `ModelNode` enthält gegebenenfalls weitere `ModelNode` Objekte als Kinder.
Identifiziert wird jeder Knoten des *Model*-Baums anhand eines eindeutigen 'key', über den er direkt adressiert werden kann. Zum Auslesen eines beliebigen Nachkommens innerhalb des Baums wird die OGNL (Object Graph Navigation Language) verwendet. Jeder Knoten kennt seine eigene OGNL-Adresse sowie seinen Vater.
Metadaten über den jeweiligen Knoten sind in der `ModelNodeDescription` enthalten, die zur Erzeugung einer entsprechenden *View* interpretiert wird. Auf Blattebene enthält ein `ModelNode` einen Wert, den er repräsentiert. Besitzt ein `ModelNode` Kinder, so ist sein Wert null.

[26] `de.cebis.moccabox.kiosk.ww.actions.ActionAdaptor`
[27] `com.opensymphony.xwork.ActionSupport`
[28] `de.cebis.moccabox.model.ModelNode`
[29] Vgl. auch das *Composite*-Muster in [Gam+95].
[30] `de.cebis.moccabox.io.command.TaskCommand`

View

Die *View* wird für das übergebene *Model* erzeugt. Dafür wird eine *Templating Engine* eingesetzt, die entsprechende Vorlagen (*Templates*) enthält, um die Daten des Models darzustellen. Es existieren Vorlagen für die Erstellung der Ausgabeformate HTML sowie PDF. Vorlagen werden von **moccabox** für die anwendungsneutrale Interpretation von *Models* zur Verfügung gestellt, können bei der Anwendungsentwicklung aber angepasst oder ersetzt werden. Als *Templating Engine* wird *Freemarker*[31] eingesetzt.

Das Paket de.cebis.moccabox.model

Im Anhang befinden sich die entsprechenden Klassen zur Definition des Models im Paket `de.cebis.moccabox.model`, wie in Abbildung 9.7 dargestellt. Es umfasst neben der zentralen Klasse `ModelNode` alle Interfaces und Klassen, die zur Berechnung und Verarbeitung des `ModelNode` im Zusammenspiel mit der VO- sowie IO-Schicht notwendig sind. Die Beschreibung aller Elemente befindet sich im Ordner *anhang/dokumentation moccabox/de/cebis/moccabox/model*, dessen Inhalt auch direkt im Browser über *anhang/dokumentation moccabox/index.html* navigiert werden kann.

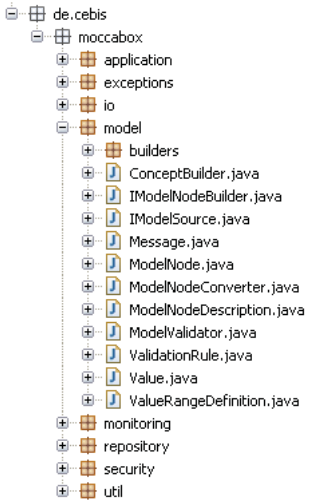

Abbildung 9.7: Das Paket de.cebis.moccabox.model im Anhang.

[31] `http://freemarker.sourceforge.net`

9.4.2 Verhalten des Web-Framework

Der Nutzer setzt einen Request über eine verfügbare *View* ab. Der eingehende `ITaskRequest`[32] wird abgefangen vom `FilterDispatcher`[33] als *Controller*, der über den `ActionAdaptor` mit der Anwendungsschicht (VOS) verbunden ist (siehe Abbildung 9.8). Das im `ITaskRequest` enthaltene `Model` - auf dem die Änderungen geschrieben werden - wird an das entsprechende VO zur Bearbeitung weitergeleitet. Sobald das VO durchgeführt wurde, wird das zurückgelieferte `Model` durch die spezifizierte *View* interpretiert. Der `TaskController` liefert den *Response* an den *Client* zurück.

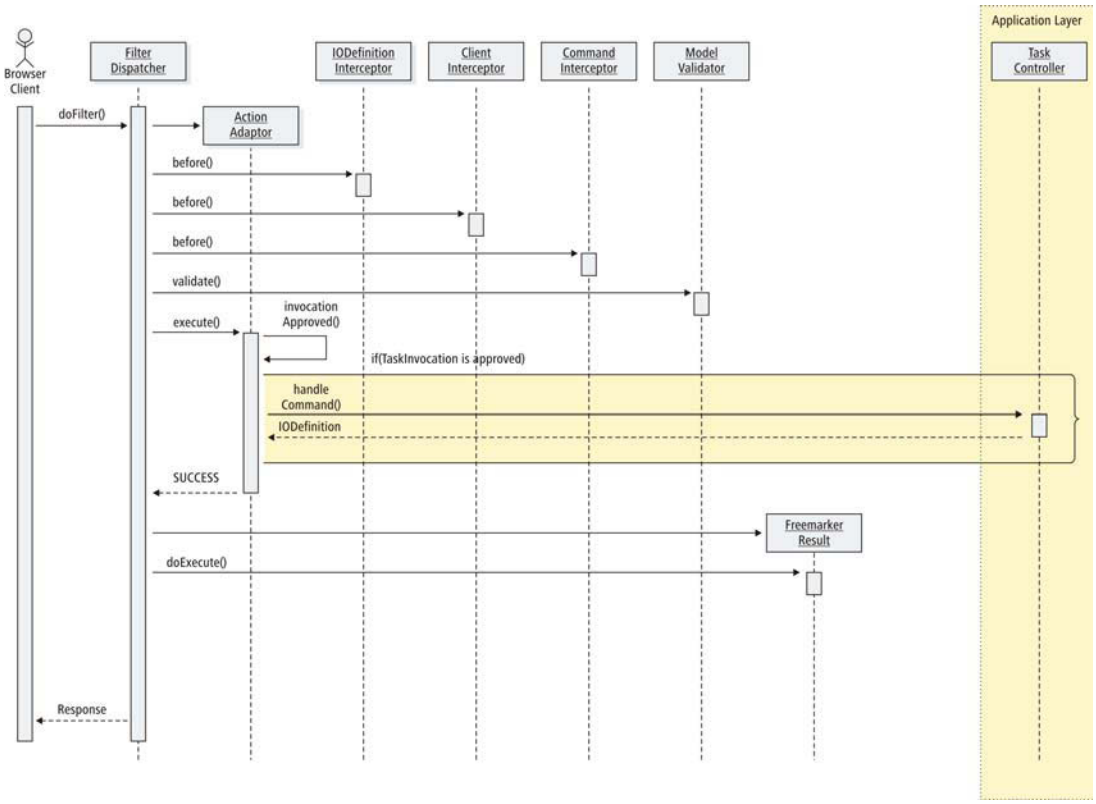

Abbildung 9.8: Sequenzdiagramm des Web-Framework

Handelt es sich lediglich um ein Update der View auf dem unveränderten Model, ist keine Interaktion mit der Anwendungsschicht notwendig (z.B. Sortieren von Tabellenspalten). Der Controller veranlasst das Update der View und gibt diese als Response zurück. Dadurch wird eine höhere Performanz erreicht.

[32] `de.cebis.moccabox.application.request.ITaskRequest`
[33] `com.opensymphony.webwork.dispatcher.FilterDispatcher`

Mit XWork und *Freemarker*[34] als Basismaschinen wird anhand von `ConceptDefinitions` das IO erzeugt, durch das die Nutzer-Interaktion ermöglicht wird. Dabei ist die Verwendung der Templates als Views so flexibel, dass durch Erweiterung der Standardinterpretation eine sehr individuelle Oberfläche generiert werden kann. Dadurch wird ein hoher Grad an Wiederverwendung erreicht, ohne Verlust an Flexibilität. Views können wiederum ineinander geschachtelt werden.

Quelltext 9.4: Ausschnitt eines concept.xml zur Definition des KO CalendarEvent

```
1  <?xml version="1.0" encoding="UTF-8"?>\\
2  <cd:conceptdefinition name="calendarevent"
3  concept="de.cebis.moccapart.calendar.domain.CalendarEvent">
4
5      <attribute name="calendar">
6          <type>text</type>
7          <required>true</required>
8          <editable>true</editable>
9          <hidden>false</hidden>
10         <viewref id="singlechoice">
11             <configuration xsi:type="vr:attributeConfiguration"
                   configurator="de.cebis.moccabox.kiosk.view.attribute.
                   config.xmlbeans.AttributeViewConfigurator">
12                 <size>15</size>
13                 <label>true</label>
14             </configuration>
15         </viewref>
16     </attribute>
17
18     <commandbuilder class="de.cebis.moccabox.application.standard.
           StandardTaskCommandBuilder">
```

[34] `http://freemarker.sourceforge.net`

```
19          <command name="calendarevent_create" type="class">
20              <param name="calendarevent.relatedConcept" sourcename="
                    this" provider="de.cebis.moccabox.application.request
                    .paramprovider.ConceptParamProvider"/>
21          </command>
22      </commandbuilder>
23
24  </cd:conceptdefinition>
```

9.5 Anwendungsfunktionalität

Basisfunktionalität für die Durchführung der Anwendung wird vom Framework zur Verfügung gestellt[35]. Das Interface `BusinessTask` hat genau eine öffentliche Methode `perform`, die mit dem Interface `ITaskRequest` als Parameter aufgerufen wird. Der `TaskRequest` enthält alle Informationen der Client-Anfrage, die für die Ausführung des VOT wichtig sind. Nach dem Aufruf liefert der `BusinessTask` ein `BusinessTaskResult` zurück, in dem die angeforderten Informationen enthalten sind.

Ein VO kann durch die Implementierung des Interface `BusinessTask` realisiert werden oder durch die Konfiguration eines entsprechenden Standard-Task. Der Vorteil der Konfiguration besteht in der Wiederverwendbarkeit vorhandener Funktionalität. Zum Teil müssen für Teilmodule (moccaparts) keinerlei neuen Tasks geschrieben werden, was den Entwicklungsaufwand drastisch reduziert.

Quelltext 9.5: Ausschnitt eines moccapart.xml zur Definition eines VO

```
1       <businesstask command="department_retrieve">
2           <taskhandler class="de.cebis.moccabox.application.impl.
                hibernate.RetrieveTask">
3               <configuration xsi:type="ths:standardConfiguration"
                    configurator="de.cebis.moccabox.application.standard.
```

[35] Siehe `de.cebis.moccabox.application`.

```
                XmlStandardBusinessTaskConfigurator">
4               <concept type="concept" name="departments" class="de.
                    cebis.pm.ko.Department">
5                   <query>
6                       <statement>from Department as d where d.
                            client = :client</statement>
7                       <param name="client" sourcename="client.this"
                             provider="de.cebis.moccabox.application.
                            request.paramprovider.
                            TaskRequestParamProvider"/>
8                   </query>
9               </concept>
10          </configuration>
11      </taskhandler>
12      <taskstatus type="finished_with_success" viewref="
            standardpage">
13          <viewelement name="departments" attributes="id,client.
                name,name,telephone,lastEditDate" conceptdefinition="
                department" commands="department_create,
                department_show,department_update,department_delete"
                viewref="table"/>
14      </taskstatus>
15  </businesstask>
```

Es besteht die Möglichkeit, konfigurierbare Standard Tasks um so genannte `Extensions` zu erweitern. Dies dient dazu, `BusinessTasks` so flexibel wie möglich zu nutzen und die Wiederverwendung von Funktionalität zu steigern. Beispiel für eine derartige `Extension` ist ein Adress-Kollektor. Diese `Extensions` werden über die Konfiguration des `BusinessTask` injiziert.

Eine zusätzliche Erweiterung muss hinsichtlich des Workflow erreicht werden. Es muss erreicht

werden, dass komplette Wizards und Ablaufroutinen von VOT abgebildet werden können. Dies betrifft auch die mögliche Persistierung von Zuständen von VOT sowie zugehöriger Transaktionen. An dieser Stelle muss noch Forschungsarbeit geleistet werden, bevor eine Umsetzung in das Framework möglich ist.

Das Paket de.cebis.moccabox.application

Im Anhang befinden sich die entsprechenden Klassen im Paket `de.cebis.moccabox.application`, wie in Abbildung 9.9 dargestellt. Es umfasst die Definition der Interfaces sowohl des `ITaskController` im Oberverzeichnis als auch des `IBusinessTask` im Unterverzeichnis `de.cebis.moccabox.application.task`. Die zugehörigen Implementierungen finden sich im Unterverzeichnis `de.cebis.moccabox.application.standard`. Die Implementierung der Standard-Tasks des `IBusinessTask` liegen im Unter-Paket `de.cebis.moccabox.application.impl.hibernate`, wie bereits in Kapitel 9.2 beschrieben. Die Implementierung bezieht sich auf die für die Persistierung verwendete Basismaschine Hibernate. Die Beschreibung aller Elemente befindet sich im Ordner *anhang/dokumentation moccabox/de/cebis/moccabox/application*, dessen Inhalt auch direkt im Browser über *anhang/dokumentation moccabox/index.html* navigiert werden kann.

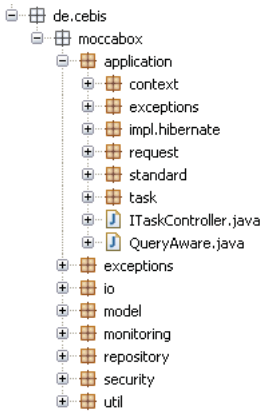

Abbildung 9.9: Das Paket de.cebis.moccabox.application im Anhang.

9.6 Persistenzmechanismus

Ein Formalziel der Anwendungsentwicklung ist die Unabhängigkeit des Systems vom Aufgabenträger, also der grundlegenden Basistechnologie (siehe hierzu auch Kapitel 3.4 und Kapitel 4.1). Die Persistierung der konzeptuellen Objekte muss aus Sicht der Anwendung bzw. des Frameworks unabhängig vom gewählten Speicherformat (z. B. relationale Datenbank, objektorientierte Datenbank, Dateisystem)[36] und dem konkreten System (z. B. Oracle, Firebird) erfolgen. Das Framework stellt den Zugriff auf persistierte Objekte und deren Zurückschreiben in die Datenbasis bereit.

Aus Sicht der Anwendung wird rein objektorientiert auf die entsprechenden Entitäten zugegriffen. Für die Persistierung stehen objektorientierter Datenbanken[37] und relationale Datenbanken zur Auswahl. In der Praxis werden zumeist relationale DBVS verwendet. Die Standardisierung objektorientierter Datenbanksysteme ist noch nicht ähnlich weit fortgeschritten wie im Bereich relationaler DBMS und daher ist die Implementierung des Standards bezüglich Performanz, Kompatibilität, Verbreitung und Verfügbarkeit noch nicht befriedigend [Schm01, 271]. Hinzu kommt, dass objektorientierte DBMS vergleichsweise teuer sind und in bestehende relationale DBMS meist hohe Investitionen - auch in Form von Wissen - geflossen sind [Oest98, 198]. Für die vorliegende Implementierung wird ein relationales Datenbankverwaltungssystem[38] auf Basis der *Structured Query Language* (SQL) eingesetzt[39]. Die entstehende semantische Lücke wird **Impedance Mismatch** genannt und muss durch eine geeignete Zugriffsschicht überbrückt werden[40]. Das ist Aufgabe eines so genannten **Object-Relational-Mapper** oder kurz **O/R-Mapper**[41].

Die für die Implementierung der **moccabox** gewählte Persistenz-API ist **Hibernate**[42]. Ein frei verfügbarer *O/R-Mapper*, der von einer großen Entwickler-Gemeinschaft unterstützt wird und

[36] Ein Vergleich verschiedener DBMS findet sich bei [Voss00].

[37] Für einen Überblick über objektorientierte Datenbanken vgl. u.a. [Heue97].

[38] Zum Relationenmodell siehe [Codd70].

[39] In der gegenwärtigen Implementierung handelt es sich dabei um MySQL bzw. Firebird, da diese Datenbanksysteme frei verfügbar sind und keine weiteren Lizenzkosten entstehen. Der Wechsel auf ein professionelles System wie Oracle ist aber dank der Schicht des O/R-Mapping problemlos möglich.

[40] Für eine ausführliche Betrachtung des Impedance Mismatch sei an dieser Stelle unter anderem auf [Ambl03, 106ff] und [ZiBe00, 253-283] verwiesen.

[41] Es wird auf eine verfügbare API zurückgegriffen. Deshalb ist der Design und Beschreibung eines solchen Mappers nicht Gegenstand der Arbeit. Dafür sei auf die entsprechende Literatur verwiesen (u. a. [Kell97], [Ambl00]).

[42] `http://www.hibernate.org`

als eigenständige Komponente verfügbar ist. Er ist seit mehreren Jahren auf dem Markt und in der gegenwärtigen Version 3.0 ausreichend mächtig, um alle Anforderungen hinsichtlich der relationentransparenten Speicherung zu erfüllen. Seit Mitte 2005 steht auch die Persistenz-API der *Enterprise Java Beans 3.0* oder kurz *EJB3* von *SunONE* zur Verfügung, die in weiten Teilen ihrer Spezifikation der API von Hibernate gleicht. Das abstrakte Framework wurde so gestaltet, dass es keinerlei Präferenz für die Implementierung einer bestimmten Persistenz-API enthält. Eine konkrete Implementierung auf Basis von *EJB3* ist durchaus anzustreben, zieht aber weitere Design-Entscheidung auch bezüglich des Einsatzes von *J2EE* nach sich (vgl. [Star05, 178]) und ist nicht Gegenstand der Arbeit.

Entwicklungsphase

Zur **Entwicklungszeit** werden die konzeptuellen Objekttypen des Anwendungssystems in so genannten *Mapping-Dateien* in XML beschrieben. Sie enthalten alle Attribute der Domänenobjekte und ihre Beziehungen untereinander. Die Ableitung der zugehörigen Java-Klassen übernimmt die API von Hibernate während des Kompilierens. Es werden die Klasse `BaseClient`[43] und die davon abgeleitete Klasse `Client`[44] angelegt. Änderungen, die Einfluss auf die Persistierung der Objekte nehmen, werden lediglich in den *Mapping-Dateien* durchgeführt und beim erneuten Kompilieren auf die Klasse `BaseClient` übertragen. Die Klasse `Client` bleibt davon unberührt. Änderungen, die an `Client` vorgenommen wurden (z. B. Operationen, die unabhängig von der Persistierung sind) werden nicht überschrieben. Das Verfahren ist unidirektional, d.h. Änderungen in der Klasse `BaseClient` haben keinen Einfluss auf die korrespondierende *Mapping-Datei* und dürfen daher nicht vorgenommen werden.
Gleichzeitig bildet Hibernate beim Kompilieren die Objekte auf das gewählte relationale DBMS ab. Das geschieht transparent für die Nutzermaschine Framework. Auch der jeweilige SQL-Dialekt der gewählten relationalen Plattform bleibt vor der Anwendung verborgen[45], wodurch der Forderung nach Aufgabenträgerunabhängigkeit genügt wird. Das Datenbank-Schema wird automatisch beim Initalisieren der Anwendung (Server-Start) aktualisiert, sofern das erforder-

[43] `de.cebis.pm.ko.base.BaseClient`
[44] `de.cebis.pm.ko.Client`
[45] Die SQL ist ein Standard zur Manipulation und Abfrage relational verwalteter Datenbestände. Dennoch verwendet fast jeder Hersteller einen eigenen SQL-Dialekt, der nur eine Teilmenge der mächtigen Sprache implementiert (siehe u. a. [Türk03]).

lich ist.

Quelltext 9.6: Ausschnitt einer Mapping-Datei Client.hbm.xml zur Definition eines KO

```
1  <?xml version="1.0"?>
2  <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
       DTD//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.
       dtd" >
3  <hibernate-mapping package="de.cebis.pm.ko">
4      <class name="Client" table="T_CLIENT">
5      <meta attribute="implements">de.cebis.moccabox.domain.
           PersistentConcept</meta>
6      <cache usage="read-write" />
7      <id name="id" column="a_id" type="java.lang.Long">
8          <generator class="native" />
9      </id>
10     <version name="version" column="a_Version" type="timestamp" />
11     <property name="name" column="a_Name" type="java.lang.String"
           length="255" />
12     </class>
13 </hibernate-mapping>
```

Laufzeitphase

Die Persistenz-API bietet zur Laufzeit Methoden zum Auslesen (`load`), Persistieren (`persist`), Aktualisieren (`update`) und Löschen (`delete`) von Objekten der Datenbasis an. Für das Auslesen stehen zwei verschiedene Methoden zur Verfügung. Das `Criteria`-Interface[46] und das `Query`-Interface[47]. Letzteres ermöglicht das Formulieren und Absenden von Abfragen in der *Hibernate Query Language* (HQL), eine Art objektorientiertes Pendant zu SQL.

[46] `net.sf.hibernate.Criteria`
[47] `net.sf.hibernate.Query`

9.7 Zugriffskontrolle

Generell muss sichergestellt werden, dass nur berechtigte Aufgabenträger einen Vorgang ausführen können.

Das Paket de.cebis.moccabox.security

Im Anhang befinden sich die Klassen, die die Zugriffskontrolle realisieren, im Paket `de.cebis.moccabox.security`, wie in Abbildung 9.10 dargestellt. Es umfasst neben der Definition der Interfaces auch eine entsprechende Implementierung des `IAuthorizationService` als `AuthorizationServiceImpl`. Die Implementierung bezieht sich auf die für die Persistierung verwendete Basismaschine Hibernate. Die Beschreibung aller Elemente befindet sich im Ordner *anhang/dokumentation moccabox/de/cebis/moccabox/security*, dessen Inhalt auch direkt im Browser über *anhang/dokumentation moccabox/index.html* navigiert werden kann.

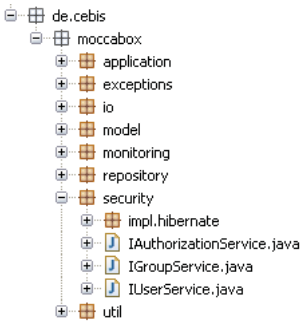

Abbildung 9.10: Das Paket de.cebis.moccabox.security im Anhang.

9.7.1 Zugriffsberechtigung für Vorgangsdurchführung

Es wird überprüft, ob der Aufgabenträger berechtigt ist, den Vorgang auszuführen. Handelt es sich um einen personellen Aufgabenträger, so wird mittels der Nutzer- und Rechteverwaltung überprüft, ob sich der Nutzer in einer Rolle befindet, der das Recht an dem auszuführenden Vorgang zugestanden wurde (siehe Kapitel 9.7.2).
Für Vorgänge wird anhand des Metasystems (siehe Kapitel 9.3.2) überprüft, ob der `BusinessTask` berechtigt ist, auf die entsprechenden konzeptuellen Objekte (`IPersistentConcept`) lesend oder schreibend zuzugreifen.

9.7.2 Nutzer- und Rechteverwaltung

Die Komponente[48] der Nutzer- und Rechteverwaltung bietet die Authentisierung und Autorisierung der entsprechenden Anwender, um eine konsistenten Zugriff auf die VOT zu gewährleisten. Das Recht für den Zugriff kann auf Typebene (Recht ein bestimmtes VO aufzurufen) oder auf Instanzebene (Recht ein VO für einen bestimmten Kontext aufzurufen) zugestanden werden. Die Rechteverwaltung muss beide Möglichkeiten anbieten.

Authentisierung mittels acegi-Security. Beliebige Authentisierungsprovider nutzbar (CAS, eigene).

Autorisierung durch Interceptor-Mechanismus, der transparent vor Aufruf eines VO überprüft, ob der Benutzer hierzu berechtigt ist.

In bisherigen Anwendungen war lediglich die Rechteverwaltung auf Typebene möglich. Mit der Basismaschine von acegi ist die Rechteverwaltung entsprechend realisiert und funktionsfähig. Zudem erfolgt die Synchronisation neuer Rechte automatisch beim Neustart, eine manuelle Anpassung wie bisher entfällt. Dadurch wird die Entwicklung entscheidend beschleunigt.
Die Rollen- und Rechteverwaltung als Teil der Anwendung ist als moccapart implementiert und kann direkt über die bekannte Oberfläche genutzt werden.
Die Rechteüberprüfung muss auf Instanzebene ausgeweitet werden. Ebenso die bedienbare Einrichtung der entsprechenden Rechte. Es muss z. B. möglich sein, über Restriktionen einem Nutzer Zugriff auf das Löschen von Bestellungen zu geben, aber eben nur auf Bestellungen, die er selbst geschrieben hat. Dieses Zugeständnis an Rechten muss noch implementiert werden.
Es wurde bereits erwähnt, dass die Durchführbarkeit eines Vorganges an Rechte gebunden sein kann, über die der Benutzer des Systems verfügen muss, um auf einen Vorgang und damit eine Ressource zugreifen zu dürfen. Für die Zuordnung von Rechten, die Benutzer an Ressourcen besitzen können, schlägt LAMPSON in [Lamp74] die Verwendung einer Zugriffs-Matrix vor. In ihr wird für jedes Subjekt spezifiziert, welche Rechte es an einer Ressource des Systems besitzt. Die direkte Zuordnung von Rechten an Ressourcen für Benutzer bringt aber einen sehr hohen Verwaltungsaufwand mit sich (Komplexität $O(|Subjekte| \times |Objekte|)$. Dieser wird

[48] Siehe `de.cebis.moccabox.security`.

durch das rollenbasierte Zugriffsmodell (engl. Role Based Access Model, **RBAC**[49]) des NIST[50] vermieden [Goo+02, 306]. Es nimmt Bezug auf die Tatsache, dass personelle AT als Bestandteil in einer Organisation eine Funktion haben, in der sie eine oder mehrere Rollen einnehmen. Die Rechte an Ressourcen werden nicht mehr direkt an Subjekte, sondern indirekt über Rollen an sie geknüpft. Die Komplexität reduziert sich damit zu $O(|Subjekte| \times |Rollen|) + O(|Rollen| \times |Objekttypen|)$. Da die Zuordnung von Rollen an Objekttypen in einer Organisation tendenziell eher statisch ist, beschränkt sich die Verwaltung auf die Zuordnung der Rollen zu Subjekten. Allerdings kann das RBAC nur eine Zuordnung von Objekttypen an Rollen ermöglichen. Ob ein Benutzer ein Recht für eine Instanz eines Objekttyps besitzt, kann nicht erfasst werden [51]. Diese Einschränkung des Modells kann durch die Formulierung einer notwendigen Beziehung, die zwischen der KO-Instanz und dem Benutzer bestehen muss, aufgehoben werden.

9.8 Internationalisierung

Die Mensch-Computer-Kommunikation muss in Sprache, Zahlenformaten etc. an den jeweiligen Benutzer angepasst sein. Vor allem in Zeiten der Globalisierung ist es unabdingbar, dass die Benutzungsoberfläche eines Anwendungssystem eine lokale Differenzierung auch zur Laufzeit ermöglicht. Die **Internationalisierung**[52] bezeichnet die Architektur von Software, für die nur einen Satz Quell- und Binärcode produziert wird, um alle Märkte zu bedienen, in der das fertige Software-Produkt eingesetzt wird [CzDe01, 7]. Sie umfasst u. a. Sprache, Maßeinheiten und Zahlenformate. Die **Lokalisierung**[53] ist die Anpassung der Anwendung an bestimmte lokale Vorgaben [CzDe01, 9] und umfasst die Übersetzung externer Text-Dateien sowie von Bildern. Sie ist damit Aufgabe des Anwendungsentwicklers. Es folgt, dass das Framework **moccabox** für die Generierung der Oberfläche nicht nur Layout, Inhalt und Logik getrennt halten muss [Star05, 250], sondern den Inhalt auch noch unterscheiden nach Information und Beschreibung

[49] Das RBAC-Modell wurde im Februar 2004 zu einem ANSI-Standard erklärt.
[50] National Institute of Standards and Technology, siehe `http://csrc.nist.gov/rbac/`.
[51] „A significant shortcoming of RBAC is the inability to distinguish which instances of a resource an individual role holder can access.“ [Goo+02, 309].
[52] In der Software-Industrie auch bekannt als I18N - *Internationalization* beginnt mit I und ende mit N, dazwischen sind 18 Buchstaben [CzDe01, 6f].
[53] Auch bekannt als L10N für *Localization*.

der Information. Die jeweilige Beschreibung wird gesondert gehalten und bei der Auswahl durch den Nutzer allen internationalisierten Komponenten zur Verfügung gestellt[54] [Star05, 251].

Die *IO-Basisschicht* stellt eine Komponente[55] zur Verfügung, in der - entsprechend der Domäne und Sprache - Beschreibungen für die Attribute hinterlegt werden können, die zum Zeitpunkt der Generierung der View die Präsentation des Inhaltes mit der jeweils vom Nutzer gewählten Sprache versehen.

Neben Maskenübersetzungen und Zahlenformaten spielen auch mehrsprachige Anwendungsdaten eine große Rolle (z. B. Anrede, Mengen- und Farbbezeichnungen bei Artikeln) [Star05, 253]. Sind diese Daten in der Datenbasis selbst gehalten, ist das Problem Teil der Anwendungsentwicklung und wird nicht vom Framework generisch zur Verfügung gestellt. Andernfalls können die Bezeichnungen per Platzhalter in der framework-basierten *Locale*-Information hinterlegt werden.

Die **moccabox** stellt den Komponenten, die eine Lokalisierung von Attributen und Werten benötigen, zwei Schnittstellen zur Verfügung. Einerseits das `ILocaleInformationRepository`[56] und andererseits das `ILocaleAwareInformationRepository`[57], das ersteres erweitert. Das zweite unterscheidet sich dadurch vom ersten, dass es als Attribut ein Objekt enthält, das `ILocaleCallback`[58] implementiert. Es muss für Aufrufe das `Locale` also nicht mit übergeben werden. Konkrete Implementierungen des jeweiligen Schnittstellen sind `LocaleInformationRepository`[59] und `LocaleAwareInformationRepository`[60].

Die verfügbaren Lokalisierungen werden mit dem entsprechenden Landeskürzel[61] hinterlegt, um das `LocaleAwareInformationRepository` zu instantiieren (siehe Quelltext 9.7).

Quelltext 9.7: Konfiguration zur Lokalisierung moccabox.xml

[54] Die länderspezifische Beschreibung wird häufig als *Locale* bezeichnet.
[55] Siehe Package `de.cebis.moccabox.io.i18n`.
[56] `de.cebis.moccabox.io.i18n.ILocaleInformationRepository`
[57] `de.cebis.moccabox.io.i18n.ILocaleAwareInformationRepository`
[58] `de.cebis.moccabox.io.i18n.ILocaleCallback`
[59] `de.cebis.moccabox.io.i18n.LocaleInformationRepository`
[60] `de.cebis.moccabox.io.i18n.LocaleAwareInformationRepository`
[61] Es werden die entsprechend dem Standard ISO-3166 mit zwei Buchstaben kodierten Landeskürzel verwendet [CzDe01, Anhang A]. Vgl. auch `http://www.unicode.org/unicode/onlinedat/countries.html`.

```
<bean id="LIR" class="de.cebis.moccabox.io.i18n.
    LocaleAwareInformationRepository" init-method="init">
  <property name="messageSource"><ref bean="MESSAGE.SOURCE"/></
      property>
  <property name="supportedLocales">
    <list>
      <value>DE_de</value>
      <value>EN_en</value>
    </list>
  </property>
</bean>
```

Die zugehörigen Informationen zur Lokalisierung sind in einer entsprechenden Textdatei hinterlegt, die beim Instantiieren der Anwendung zusammen mit dem `moccapart_x.xml` eingelesen wird. Für `moccapart_system.xml` müssen die Dateien `system_de.properties` und `system_en.properties` hinterlegt werden. Ein Ausschnitt ist in Quellcode 9.8 dargestellt.

Quelltext 9.8: Lokalisierungsdatei system_de.properties

```
pattern.required.list=java.util.Date
pattern.list.java.util.Date=pattern.date.format1,pattern.date.format2
pattern.date.format1=d.M.yy pattern.date.format2=d/M/yy
pattern.date.display=dd.MM.yyyy

error.key.required=Es handelt sich um ein Pflichtattribut!

command.user_create.name=#\%de.cebis.moccabox.security.domain.User.
    Plural.Name\%# #\%general.command.create\%#

de.cebis.moccabox.security.domain.User.Singular.Name=Benutzer
de.cebis.moccabox.security.domain.User.Plural.Name=Benutzer
```

```
12 de.cebis.moccabox.security.domain.User.attribute.login=Benutzername
13 de.cebis.moccabox.security.domain.User.attribute.password=Passwort
14 de.cebis.moccabox.security.domain.User.attribute.id=Id
15 de.cebis.moccabox.security.domain.User.attribute.accountLocked=Zugang
16 de.cebis.moccabox.security.domain.User.attribute.accountEnabled=aktiv
17
18 general.command.create=anlegen
```

Die Komponenten, die Attributbezeichnungen oder Fehlermeldungen abhängig von der Lokalisierung benötigen, können das Repository einsetzen. Ebenso für die Konvertierung von Text in Zahlen- oder Datumsformat, das von einem lokalisierten Konvertierungsmuster abhängt[62]. Die Verwendung im Framework und in der Anwendung ist beispielhaft in Quelltext 9.9 wiedergegeben.

Quelltext 9.9: Verwendung der Lokalisierung im Quellcode

```
1 String ERROR_KEY_REQUIRED = "error.key.required";
2 LocaleAwareInformationRepository  _lir =
3     (LocaleAwareInformationRepository)LocaleInformationRepository.
          getInstance();
4 String strError = _lir.text(ERROR_KEY_REQUIRED);
5
6 String strValue ="10.05.05"; Class type =
7 Class.getClassForName("java.util.Date"); Date dValue =
8 (Date)_lir.convert(strValue, type);
```

Die Konvertierungsmuster werden sowohl für die Überprüfung der Nutzereingabe wie für die Formatierung der Ausgabe beachtet.

Durch den Einsatz von HTML kann auf einen Layout-Manager größtenteils verzichtet werden [Star05, 251], da die Maskenelemente automatisch in der Größe angepasst werden.

[62] Die deutsche Datumsangabe wird meist nach dem Muster 'dd.mm.yyyy' angegeben, während im angelsächsischen Raum 'mm-dd-yyyy' üblich ist.

9.9 Dokumentation

Die Dokumentation der Anwendung erfolgt durch Beschreibung der entwickelten Vorgangsobjekte.

Dokumentation auf Entwicklungsebene

Auf Entwicklungsebene findet dies durch Kommentierung des erstellten Programm-Codes statt, wie in Quelltext 9.10 am Beispiel des Interface `IBusinessTask` abgebildet.

Quelltext 9.10: Dokumentation des Interface IBusinessTask (Ausschnitt)

```
1 /**
2  * <p>Ein <code>BusinessTask</code> repr\"{a}sentiert ein
3  * Vorgangsobjekt (VO) ... </p>
4  *
5  * @author weichelt - 17.01.2006
6  */
7 public interface IBusinessTask { ... }
```

Die Dokumentation der Code-Basis kann dann mit Hilfe des Programms *javadoc* als HTML-Manual erzeugt werden.

Dokumentation auf Anwendungsebene

Der Entwickler kann zudem direkt aus der Beschreibung der Vorgangsobjekte die Dokumentation für den Anwender übernehmen, sofern diese bei der Modellierung verfasst wurde. Andernfalls kann sie auch vom Entwickler oder einem Dritten verfasst werden, da sie gemeinsam mit den Lokalisierungsdaten in einer eigenen Datei gehalten wird[63]. Zur Verdeutlichung wird ein Beispiel aus dem Anwendungssystem ..::biene::.. vorgestellt, das auf Basis des Frameworks **moccabox** als Kursverwaltungssystem für eine Business-Sprachschule entwickelt wurde[64]. In der Lokalisierungsdatei muss sich der in Quelltext 9.11 dargestellte Eintrag finden.

[63] Siehe Kapitel 9.8.
[64] Siehe auch Kapitel 10.1.

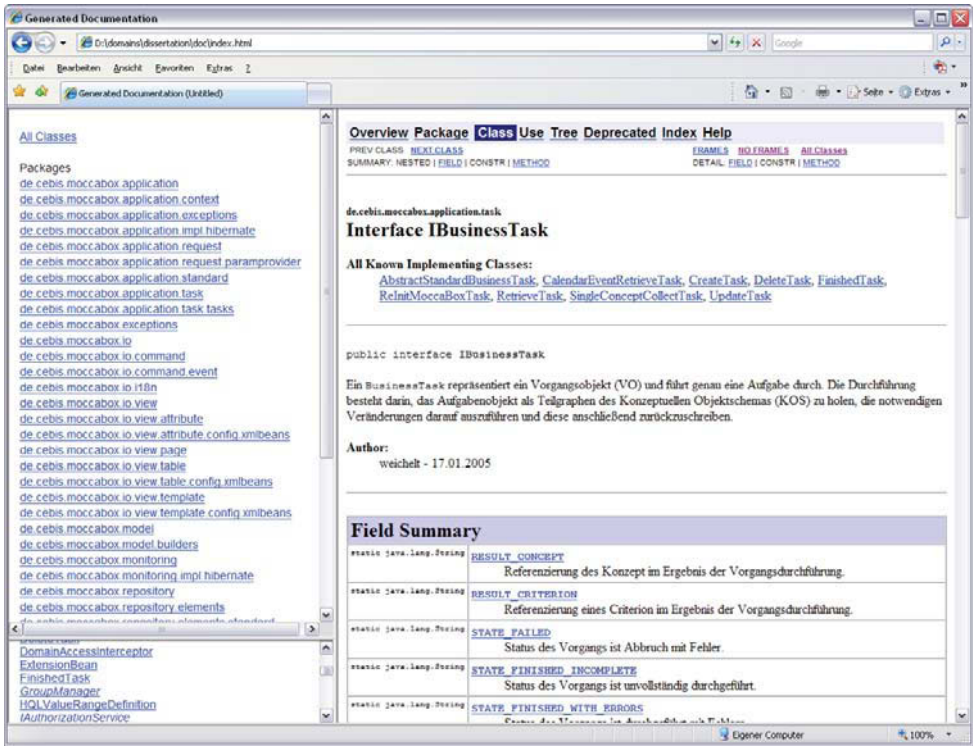

Abbildung 9.11: Generierte HTML-Dokumentation von Java-Klassen

Quelltext 9.11: Konfigurationsdatei nations_lang_de.properties (Ausschnitt)

```
1 de.cebis.nations.vo.search.ServiceEnforcementSearch.Help=Die Suche
2 nach Phasen erm\"{o}glicht es, ...
```

Dieser wird dann für den Anwender wie in Abbildung 9.12 gezeigt interpretiert. Die Hilfe kann über das Symbol der Glühbirne am rechten oberen Bildschirmrand zur Laufzeit ein- und ausgeblendet werden.

Der Aufwand für die Dokumentation der Anwendung ist damit deutlich verringert, da er entweder direkt aus der Modellierung übernommen werden kann oder aber ohne Notwendigkeit von Programmierkenntnissen als Notation angelegt werden kann.

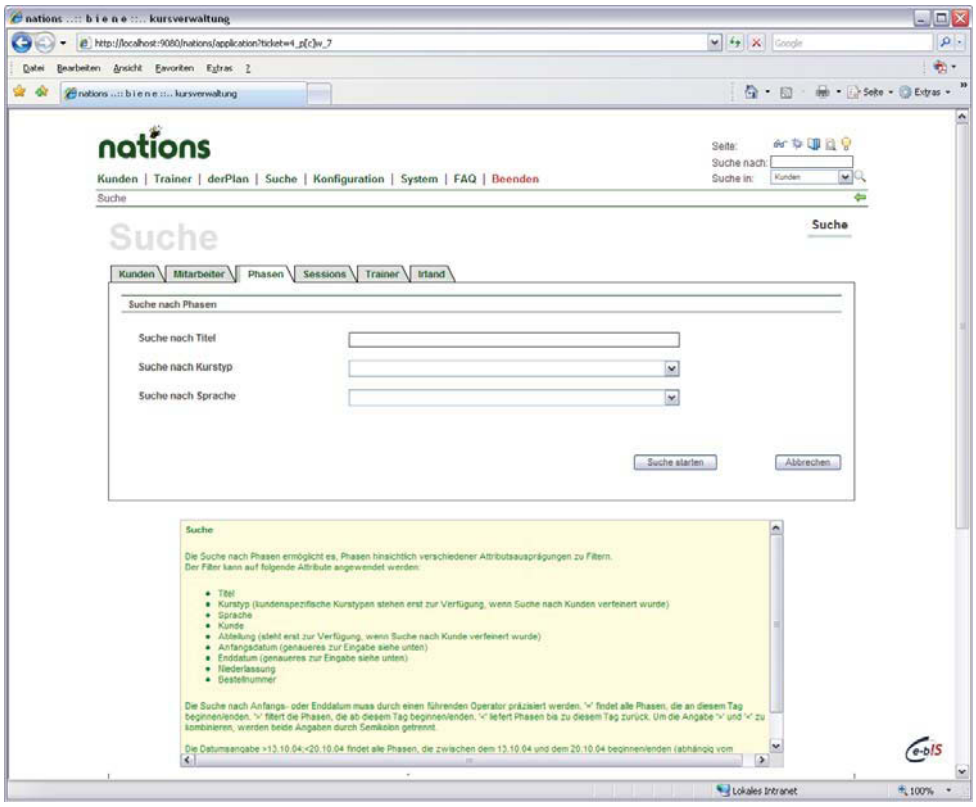

Abbildung 9.12: Darstellung der Dokumentation im Anwendungssystem

10 Zusammenfassung, Bewertung und Ausblick

10.1 Zusammenfassung und Bewertung der Ergebnisse der Arbeit

Ziel der Arbeit war es, die Entwicklung betrieblicher Anwendungssysteme durch eine geeignete Basismaschine beim Übergang von fachlicher zu software-technischer Beschreibungsebene und der anschließenden Realisierung zu unterstützen. Als methodischer Rahmen wurde der SOM-Ansatz gewählt (u. a. [FeSi06, 184-221]).

Ausgehend von den Fachkonzepten der SOM-Methodik zur fachlichen Spezifikation eines Anwendungssystems wurde ein Software-Framework entworfen und implementiert, das basierend auf der Software-Architektur des objektorientierten Architekturmodells [Ambe93] eine geringe semantische Lücke zur Spezifikation der Nutzermaschine aufweist.

Durch die Wiederverwendbarkeit des Frameworks sowohl auf Ebene des Software-Entwurfs als auch der Implementierung [Johnn97, 10f] wird der Umfang der Systementwicklungsaufgabe verringert. Da das Framework den Kontrollfluss der Anwendung übernimmt (*Inversion of Control*) [JoFo88, 88] ist es möglich die allgemeine Komplexität des Software-Systems von der spezifischen Geschäftslogik der Anwendung zu trennen.

Diese allgemeinen Aspekte des Framework-Einsatzes werden durch die hier entwickelte Lösung **moccabox** noch erweitert. Das Framework **moccabox** wurde speziell für die Abbildung eines fachlichen Anwendungsmodells gemäß SOM konzipiert. Dadurch konnte der Entwicklungsprozess des Frameworks (siehe Kapitel 7.5) *top down* initiiert werden, entgegen des in der Literatur häufig beschriebenen Vorgehens *bottom up* (u. a. [Pree97], [RoJo97b]). Im Ergebnis ist nicht technische Funktionalität Ausgangspunkt des vorliegenden Framework-Entwurfs ge-

wesen, sondern die fachliche Ableitung aus Anforderungen betrieblicher Informationssysteme. Der geringe Komplexitätsabstand zwischen der Spezifikation der Nutzermaschine und den Komponenten der Basismaschine trägt dazu bei, die Mächtigkeit der SOM-Methodik auch bei der Realisierung zu erhalten:

- **Verfolgung von Integrationszielen**

  SOM leitet das fachliche Modell des Anwendungssystems aus den Geschäftsprozessen der Organisation ab. SOM unterstützt dabei die Verfolgung definierter Integrationsziele (siehe 5.6). Zur Verfolgung der Integrationsziele während der Realisierung bietet das Framework entsprechende Komponenten an.

- **Vollständige Abbildung**

  SOM spezifiziert interaktive Anwendungssysteme, die neben der Anwendungsfunktionalität[1] auch die Kommunikation mit personellen Aufgabenträgern zur Durchführung teilautomatisierter Aufgaben umfasst. Das Framework bietet neben der Basisfunktionalität für Anwendung und Datenverwaltung auch ein umfangreiches GUI-Framework zur Ableitung der Benutzungsoberfläche.

- **Abgrenzung von Anwendungssystemen**

  SOM grenzt Anwendungssysteme entsprechend der Geschäftsprozesse der Organisation ab. Dies ist im Sinne der notwendigen Abstimmung der wechselseitig voneinander abhängigen Teilsysteme Aufgabenstruktur, Aufbauorganisation und Anwendungssysteme des betrieblichen Informationssystems [Sinz99, 20]. Anwendungssysteme, die mit dem Framework **moccabox** realisiert werden, folgen dieser Abgrenzung. Spätere Anpassungen sind entsprechend leichter nachvollziehbar [FeSi06, 210].

- **Modellgetriebene Entwicklung**

  SOM bietet über alle Modellebenen definierte Ableitungsregeln, die eine modellgetriebene Ableitung und Spezifikation eines fachlichen Modells des Anwendungssystems erlauben. Um diese Durchgängigkeit auf die Realisierung ausweiten zu können, ist eine entsprechende software-technische Zielplattform notwendig. Das hier entwickelte Framework stellt eine solche Zielplattform dar.

[1] Siehe ADK-Strukturmodell [FeSi06, 305-307].

Das Framework stellt eine flexible Basismaschine in dem Sinn dar, dass beliebig viele Nutzermaschinen anhand zugehöriger Programme realisiert werden können [FeSi06, 304]. Aus Sicht des fachlichen Modells gesehen, sollte jedes Anwendungssystem, das mit SOM fachlich spezifiziert werden kann, auch mit dem Framework **moccabox** umgesetzt werden können. Im Rahmen der Arbeit des Centrums für betriebliche Informationssysteme (Ce-bIS) wurde das Framework für die Realisierung verschiedener Anwendungen eingesetzt, die zum Großteil auch heute noch im Einsatz sind. Folgende Übersicht stellt einige davon kurz vor:

- **Leistungsverwaltungs- und Abrechnungssystem für Beratungen**

  Das Anwendungssystem KiSy wurde für eine Anwaltskanzlei, basierend auf der Analyse ihrer Geschäftsprozesse, entwickelt. Dabei stand vor allem der Zugriff nur durch autorisierte Anwender (siehe Kapitel 8.3.5), die intuitive Bedienbarkeit sowie umfangreiche Auswertungsmöglichkeiten im Vordergrund. Die Entwicklung nahm ungefähr 4 Personenmonate in Anspruch. Das System ist seit 2002 im Einsatz.

- **Weiterbildungsportal**

  Das Portal des Campus wissenschaftliche Weiterbildung Bayern[2] (cwwb) wurde zur Verwaltung der Weiterbildungsangebote aller bayerischen Hochschulen entwickelt. Im Vordergrund stand die Kopplung mit dem Katalogsystem *FlexNow* des Wissenschaftlichen Instituts für Hochschulsoftware der Universität Bamberg[3] (ihb). Grundlage war die fachliche Spezifikation der Anwendung in SOM. An der software-technischen Entwicklung waren kooperativ bis zu 8 Personen über einen Zeitraum von mehreren Monaten beteiligt. Das System ist seit 2004 im Einsatz.

- **Kursverwaltungssystem**

  Das Anwendungssystem ..::biene::.. wurde für eine Sprachschule zur Organisation der individuellen Business-Sprachkurse für über 500 Kunden mit etwa 400 Trainern an 3 deutschen und einem irischen Standort entwickelt. Ausgangspunkt war auch hier die Analyse der zu Grunde liegenden Geschäftsprozesse sowie Abbildung der Domäne in SOM. Im Vordergrund stand die Standortunabhängigkeit der Anwendung, die Konsistenz der Ressourcenplanung (siehe Kapitel 8.3.3), Mehrsprachigkeit (siehe Kapitel 8.2.4) sowie

[2] http://cwwb.de/
[3] http://www.ihb.uni-bamberg.de/

die Ausgabe in verschiedenen Formaten (HTML, PDF). Die Entwicklung umfasste etwa 5 Personenmonate. Die Anwendung ist seit 2004 im Einsatz.

- **Evaluationssystem für Online-Befragungen**
  Die Flexibilität des Frameworks zeigt sich beim Evaluationssystem Kovius, das kurzfristig für ein universitäres Forschungsprojekt zur Befragung von etwa 500 Kommunen entwickelt wurde. Es wurde eine generische Umfrage-Komponente für **moccabox** entwickelt, die vom Auftraggeber der Umfrage selbst definiert werden kann. Die Komponente wurde in etwa 2 Personenmonaten realisiert und in verschiedenen weiteren Projekten eingesetzt[4]. Die Umfragen fanden in den Jahren 2004 bis 2005 statt.

Damit wurde in dieser Arbeit ein Framework entwickelt, das nicht nur theoretisch die Systementwicklungsaufgabe in der Phase der Realisierung unterstützen kann, sondern dass auch erfolgreich bei der Umsetzung mehrerer praxisrelevanter Anwendungen zum Einsatz kam.

10.2 Ausblick

Der Framework-Entwicklungsprozess ist ein iterativ-evolutorischer Prozess, der nicht beendet ist, sobald das Framework zur Entwicklung von Applikationen eingesetzt wird[5]. Gesammelte Erfahrungen und neue Anforderungen aus der Anwendung fließen in den weiteren Entwicklungsprozess des Frameworks ein[6]. Daher ist die Entwicklung der **moccabox** auch nicht als abgeschlossen zu betrachten. Vor allem die ständige Evolution der bei der Entwicklung eingesetzten Basismaschinen kann eine Anpassung in Zukunft sinnvoll machen.
Eine der großen Herausforderungen bei der framework-basierten Entwicklung von Anwendungssystemen ist sicherlich die Koordination und die notwendigerweise enge Abstimmung zwischen Framework-Entwickler und Applikations-Entwickler [Maie98]. Bei den verschiedenen Projekten, die mit der gegenwärtigen Version der **moccabox** und ihren Vorgängern realisiert wurden, war in den meisten Fällen mindestens ein Framework-Entwickler im Projekt-Team der Anwendungsentwicklung vertreten. Dennoch stellt die Weiterentwicklung und Erweiterung

[4] Unter anderem für die Virtuelle Hochschule Bayern (vhb).
[5] Vgl. Kapitel 7.1.
[6] Vgl. u. a. [RoJo97a].

der Funktionalität besondere Ansprüche an die Flexibilität und Lernbereitschaft des Anwendungsentwicklers.
Für den Einsatz im Entwicklungsprozess bestehen weitere Unterstützungspotenziale, die den in der vorliegenden Arbeit gesetzten Rahmen überschreiten. Für die weitere Forschung und Entwicklung sind die im Folgenden aufgeführten Punkte durchaus von Interesse.

Werkzeug-Unterstützung des Entwicklers

Die Entwicklung mit dem Framework erfordert noch den Zugriff auf einige der Konfigurationsdateien im XML-Format. Hier wäre eine Werkzeug-Unterstützung durch *CASE-Tools* hilfreich, da sie den Entwickler durch Visualisierung, Validierung und Vereinfachung unterstützen können.

Automatisierung der Abbildung

Für die durchgehend modellgetriebene Entwicklung von Anwendungssystemen auf der Basis der SOM-Methodik stellt das Framework als mögliche Implementierungsplattform einen ersten Schritt dar. Eine stärkere Automatisierung der Durchführung der Systementwicklungsaufgabe kann durch die Entwicklung von Transformatoren und Generatoren im Sinne der modellgetriebenen Software-Entwicklung [StVö05] erreicht werden. Vorarbeiten finden sich bei [Mali97].

Vorfertigung von Komponenten

Für SOM wurden in verschiedenen Projekten domänenspezifische Referenzmodelle entwickelt (u. a. [Fer+96a], [Fer+98], [Rüff99]). Diese Modelle können mit dem Ziel einer höheren Vorfertigung als Software-Komponenten auf das Framework abgebildet werden und zur Wiederverwendung verfügbar gemacht werden. Unterstützung könnte auch das Framework selbst liefern, indem das Auffinden und Wiederverwenden von Vorgängen, die bereits als Teilmodul umgesetzt wurden durch das Repository erleichtert wird (siehe [Mali97, 197]).

Einbinden in SOA-Umgebung

Primäre Aufgabe des Informationsmanagements ist die Gestaltung und Lenkung des Informationssystems bezüglich der Aspekte Automatisierung und Integration. Das betriebliche Informations-

system umfasst dabei mehrere Anwendungssysteme, die ihm als maschinelle Aufgabenträger zur (teil-)automatisierten Durchführung seiner Aufgaben dienen. Für die Koordination bzw. Orchestrierung der verschiedenen Anwendungssysteme wird im betrieblichen Kontext zunehmend die Architektur des Informationssystems entsprechend der Service Oriented Architecture (SOA) strukturiert (siehe Kapitel 3.3.5). Dabei werden ein oder mehrere Anwendungssysteme zu einem Service zusammengefasst. Es ist zu untersuchen, welche Schnittstellen das Framework **moccabox** für die Unterstützung solcher Informationssystemarchitekturen in Zukunft anbieten muss.

Literaturverzeichnis

[Ale+78] ALEXANDER, C.; ISHIKAWA, S.; SILVERSTEIN, M.: *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, 1978

[Alk+03] ALKASSAR, A.; BROY, M.; GEHRING, F.; GARSCHHAMMER, M.; HEGERING, H.-G.; KEIL, P.; KELTER, H.; LÖWER, U.; PANKOW, M.; PICOT, A.; SADEGHI, A.-R.; SCHIFFERS, M.: *Kommunikations- und Informationstechnik 2010+3: Neue Trends und Entwicklungen in Technologie, Anwendungen und Sicherheit*. 2003

[Ambe93] AMBERG, M.: *Konzeption eines Software-Architekturmodells für die objektorientierte Entwicklung betrieblicher Anwendungssysteme*, Universität Bamberg, Dissertation, 1993

[Ambe99] AMBERG, M.: *Prozeßorientierte betriebliche Informationssysteme*. Berlin, Heidelberg u.a.: Springer Verlag, 1999

[Ambl00] AMBLER, S.: *Mapping Objects to Relational Databases - What You Need to Know and Why*. http://www-128.ibm.com/developerworks/webservices/library/ws-mapping-to-rdb/ [Abruf 12.10.2005], 2000

[Ambl03] AMBLER, S. W.: *Agile Database Techniques*. Chichester, New York u.a.: John Wiley & Sons, 2003

[ANSI75] ANSI/X3/SPARC: *Interim Report ANSI/X3/SPARC Study Group on Data Base Management Systems*. In: Bulletin of ACM-SIGMOD, Volume 7 (1975)

[BaCo91] BASS, L. J.; COUTAZ, J.: *Developing Software for the User Interface*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1991

[Balz82] BALZERT, H.: *Die Entwicklung von Software-Systemen*. Mannheim: B.I.-Wissenschaftsverlag, 1982

[Balz00] BALZERT, H.: *Lehrbuch der Software-Technik - Software-Entwicklung*. Lehrbücher der Informatik, 2. Aufl., Heidelberg AND Berlin: Spektrum Akademischer Verlag, 2000

[Balz08] BALZERT, H.: *Lehrbuch der Softwaretechnik: Softwaremanagement*. Lehrbücher der Informatik, 2. Aufl., Heidelberg: Spektrum Akademischer Verlag, 2008

[Bas+01] BASS, L.; CLEMENTS, P.; KAZMAN, R.: *Software Architecture in Practice*. 8. Druck, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001

[Beck00] BECK, K.: *Extreme Programming Explained: Embrace Change*. Boston: Addison-Wesley, 2000

[BeCu99] BEVAN, N.; CURSON, I.: *Planning and Implementing User-Centred Design*. In: *CHI '99: CHI '99 extended abstracts on Human factors in computing systems*. New York, NY, USA: ACM Press, 1999, S. 137–138

[BeFo01] BECK, K.; FOWLER, M.: *Planning Extreme Programming*. Boston: Addison-Wesley, 2001

[BeSc] BEA, F. X.; SCHNAITMANN, H.: *Begriff und Struktur betriebswirtschaftlicher Prozesse*. In: Wirtschaftswissenschaftliches Studium, 6

[Bett05] BETTIN, J.: *Grundlagen des Product Line Engineering*. In: STAHL, T. (Hrsg.); VÖLTER, M. (Hrsg.): *Modellgetriebene Softwareentwicklung*. Heidelberg: Dpunkt Verlag, 2005, S. 243–256

[Bie+05] BIEBERSTEIN, N.; BOSE, S.; WALKER, L.; LYNCH, A.: *Impact of Service-Oriented Architecture on Enterprise Systems, Organizational Structures, and Individuals*. In: IBM Systems Journal, 44 (2005) 4, S. 691–708. – ISSN 0018–8670

[BiPe89] BIGGERSTAFF, T.; PERLIS, A.: *Software Reusability*. Reading, Massachusetts: ACM Press & Addison Wesley, 1989

[BiRi89] BIGGERSTAFF, T. J.; RICHTER, C.: *Reusability Framework, Assessment, and Directions*. (1989), S. 1–17

[Blas02] BLASCHEK, G.: *Mensch-Maschine-Kommunikation*. In: RECHENBERG, P. (Hrsg.); POMBERGER, G. (Hrsg.): *Informatik-Handbuch*, 2002, S. 839–855

[Boeh88] BOEHM, B.: *A Spiral Model of Software Development and Enhancement*. In: SIGSOFT Software Engineering Notes, 11 (1988) 4, S. 14–24

[Boeh08] BOEHM, B.: *Das Software-Engineering im 20. und 21. Jahrhundert*. In: Objekt-Spektrum, (2008) Heft 6, S. 16–25

[Boh+98] BOHRER, K.; JOHNSON, V.; NILSSON, A.; RUBIN, B.: *Business process components for distributed object applications*. In: Communications of the ACM, 41 (1998) 6, S. 43–48. – ISSN 0001–0782

[Bohr97] BOHRER, K.: *Middleware isolates business logic.* In: Object Magazine, 7 (1997) 9, S. 41–46. – ISSN 1055–3641

[Bohr98] BOHRER, K.: *Architecture of the San Francisco Frameworks.* In: IBM Systems Journal, 37 (1998) 2, S. 156–169

[Bono04] BONOMO-KAPPELER, I.: *Aspektorientierte Software-Entwicklung - unter besonderer Berücksichtigung der begrifflichen Zusammenhänge und der Einbettung in den Entwicklungsprozess*, Universität Zürich, Diplomarbeit, 2004

[Booc94] BOOCH, G.: *Object-Oriented Analysis and Design with Applications*. 2^{nd} Edition, Redwood City: Benjamin Cummings, 1994

[BoSc04] BOES, A.; SCHWEMMLE, M.: *Herausforderung Offshoring - Internationalisierung und Auslagerung von IT-Dienstleistungen*. Düsseldorf: Edition der Hans Böckler Stiftung, 2004

[Broo87] BROOKS, F. P.: *No silver bullet: Essence and Accidents of Software Engineering*. In: Computer, 20 (1987) 4, S. 10–19

[Bus+00] BUSCHMANN, F.; MEUNIER, R.; ROHNERT, H.; SOMMERLAD, P.; STAL, M.: *Pattern-orientierte Software-Architektur*. 1. korr. Nachdruck, München: Addison-Wesley, 2000

[Can+05] CANFORA, G.; CIMITILE, A.; VISAGGIO, C. A.: *An Empirical Study on the Productivity of the Pair Programming*. In: BAUMEISTER, H. (Hrsg.); MARCHESI, M. (Hrsg.); HOLCOMBE, M. (Hrsg.): *Extreme Programming and Agile Processes in Software Engineering, XP 2005*. Heidelberg: Springer-Verlag, 2005, S. 92–99

[Car+00] CAREY, J.; BRENT, C.; GRASER, T.: *SanFrancisco Design Patterns. Blueprints for Business Software*. Reading, Massachusetts: Addison-Wesley, 2000

[Che+05] CHERBAKOV, L.; GALAMBOS, G.; HARISHANKAR, R.; KALYANA, S.; RACKHAM, G.: *Impact of Service Orientation at the Business Level*. In: IBM Systems Journal, 44 (2005) 4, S. 653–668. – ISSN 0018–8670

[Chen76] CHEN, P. P.-S.: *The Entity-Relationship Model - Toward a Unified View of Data*. In: ACM Transactions on Database Systems, 1 (1976) 1, S. 9–36

[ChKe91] CHIDAMBER, S. R.; KEMERER, C. F.: *Towards a metrics suite for object oriented design*. In: *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM Press, 1991, S. 197–211

[CoDa94] COOK, S.; DANIELS, J.: *Designing Object Systems. Object-Oriented modeling with Syntropy.* Englewood Cliffs, New Jersey: Prentice Hall, 1994

[Codd70] CODD, E. F.: *A Relational Model of Data for Large Shared Data Banks*. In: Communications of the ACM, Volume 13 (1970) Issue 6, S. 377–387

[Con+00] CONSTANTINIDES, C. A.; BADER, A.; ELRAD, T. H.; NETINANT, P.; FAYAD, M. E.: *Designing an Aspect-Oriented Framework in an Object-Oriented Environment*. In: ACM Computing Survey, 32 (2000) 1es, S. 41

[CoOs04a] COLSMAN, W.; OSTHUS, T.: *Einsatz eines Web-Frameworks in einer MDA-Entwicklungsumgebung*. In: ObjektSpektrum, (2004) Heft 4, S. 27–30

[CoOs04b] COLSMAN, W.; OSWALD, M.: *Vom ER-Modell zur MDA: Ein Weg zur praktischen Anwendung der Model Driven Architecture in größeren Projekten*. In: ObjektSpektrum, (2004) Heft 1, S. 2–4

[Cout87] COUTAZ, J.: *The construction of user interfaces and the object paradigm*. In: *European conference on object-oriented programming on ECOOP '87*. London, UK: Springer-Verlag, 1987, S. 121–130

[Cout97] COUTAZ, J.: *PAC-ing the Architecture of Your User Interface*. In: *Proceedings of the 4th Eurographics Workshop on Design, Specication and Verication of Interactive Systems (DSV-IS'97)*, Springer, Heidelberg, New York, 1997, S. 15–32

[CoYo91] COAD, P.; YOURDON, E.: *Object-Oriented Analysis*. 2nd Edition, Upper Saddle River, NJ, USA: Yourdon Press, 1991

[Crüs98] CRÜSEMANN, J.: *Untersuchung und Bewertung von Methoden zur Unternehmens- und Geschäftsprozeßmodellierung am Beispiel von EP/KID*, Universität Hamburg, Diplomarbeit, 1998

[Czar98] CZARNECKI, K.: *Generative Programming: Prinicples and Techniques of Software engineering Based on Automated Configuration and Fragment-Based Component Models*, Technische Universität Ilmenau, Dissertation, 1998

[CzDe01] CZARNECKI, D.; DEITSCH, A.: *Java Internationalization*. Sebastopol: O'Reilly and Associates, 2001

[DeMa79] DEMARCO, T.: *Structured Analysis and System Specification*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1979

[DePe02] DENNINGER, S.; PETERS, I.: *Enterprise JavaBeans 2.0*. 2. Auflage, München: Addison-Wesley, 2002

[Deut89] DEUTSCH, L. P.: *Design Reuse and Frameworks in the Smalltalk-80 System*. (1989), S. 57–71

[Dijk68] DIJKSTRA, E. W.: *The structure of the THE-multiprogramming system*. In: Communications of the ACM, 11 (1968) 5, S. 341–346

[Dijk72] DIJKSTRA, E. W.: *The Humble Programmer*. In: Communications of the ACM, 15 (1972) 10, S. 859–866

[Dijk76] DIJKSTRA, E. W.: *A Discipline of Programming*. Engelwood Cliffs, New Jersey: Prentice-Hall, 1976

[Eic+04] EICH, S.; WAGNER, E.; MENGES, D.: *Geschäftslogik-zentrierte Architekturen in J2EE*. In: ObjektSpektrum, (2004) Heft 4, S. 51–58

[Eich93] EICHENER, V.: *Software-ergonomische Normung im Rahmen des europäischen Arbeitsschutzes*. In: KONRADT, U. (Hrsg.); DRISIS, L. (Hrsg.): *Software-Ergonomie in der Gruppenarbeit* Band Band 5. Neue Informationstechnologien und flexible Arbeitssysteme, Opladen: Leske und Budrich, 1993, S. 101–120

[Elr+01] ELRAD, T.; FILMAN, R. E.; BADER, A.: *Aspect-Oriented Programming: Introduction*. In: Communications of the ACM, 44 (2001) 10, S. 29–32

[EmEl98] EMMERICH, W.; ELLMER, E.: *Business objects: The next step in component technology ?* In: *CBISE 98*, 1998

[Ensl78] ENSLOW, P. H.: *What is a "distributed" data processing system?* In: j-COMPUTER, 11 (1978) 1, S. 13–21

[Erl04] ERL, T.: *Service-Oriented Architecture*. Upper Saddle River, New Jersey: Prentice Hall PTR, 2004

[Evan04] EVANS, E.: *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Reading, Massachusetts: Addison-Wesley, 2004

[FaCl96] FAYAD, M.; CLINE, M. P.: *Aspects of software adaptability*. In: Commununications of the ACM, 39 (1996) 10, S. 58–59

[FaSc97] FAYAD, M.; SCHMIDT, D. C.: *Object-Oriented Application Frameworks*. In: Communications of the ACM, 40 (1997) 10, S. 32–38

[Fay+00] FAYAD, M. E.; HAMU, D. S.; BRUGALI, D.: *Enterprise frameworks characteristics, criteria, and challenges*. In: Communications of the ACM, 43 (2000) 10, S. 39–46

[Fer+94] FERSTL, O. K.; SINZ, E. J.; AMBERG, M.; HAGEMANN, U.; MALISCHEWSKI, C.: *Tool-Based Business Process Modeling Using the SOM Approach*. In: *Bamberger Beiträge zur Wirtschaftsinformatik Nr. 19*. Universität Bamberg, 1994

[Fer+96a] FERSTL, O. K.; HAMMEL, C.; KELLER, G.; POPP, A.; SCHLITT, M.; SINZ, E. J.; WOLF, S.; ZENCKE, P.: *Wiederverwendbare und erweiterbare Geschäftsporzeß- und Anwendungssystem-Architekturen*. In: *Statusseminar des BMBF zur Softwaretechnologie*. Berlin: BMBF, 1996, S. 3–21

[Fer+96b] FERSTL, O. K.; SINZ, E. J.; AMBERG, M.: *Stichwörter zum Fachgebiet Wirtschaftsinformatik*. In: *Bamberger Beiträge zur Wirtschaftsinformatik Nr. 36*. Universität Bamberg, 1996

[Fer+97] FERSTL, O. K.; SINZ, E. J.; HAMMEL, C.; SCHLITT, M.; WOLF, S.: *Applications Objects - fachliche Bausteine für die Entwicklung komponentenbasierter Anwendungssysteme*. In: *Bamberger Beiträge zur Wirtschaftsinformatik Nr. 42*. Universität Bamberg, 1997

[Fer+98] FERSTL, O. K.; SINZ, E. J.; HAMMEL, C.; SCHLITT, M.; WOLF, S.; POPP, K.; KEHLENBECK, R.; PFISTER, A.; KNIEP, H.; NIELSEN, N.; SEITZ, A.: *WEGA - Wiederverwendbare und erweiterbare Geschäftsprozeß- und Anwendungssystemarchitekturen*. Walldorf: Abschlussbericht, 1998

[Fers79] FERSTL, O. K.: *Konstruktion und Analyse von Simulationsmodellen*. Königstein/Taunus: Hain, 1979

[Fers92] FERSTL, O. K.: *Integrationskonzepte betrieblicher Anwendungssysteme*. In: *Fachberichte Informatik*. Universität Koblenz-Landau, 1992

[FeSi84] FERSTL, O. K.; SINZ, E. J.: *Software-Konzepte der Wirtschaftsinformatik*. Berlin: de Gruyter, 1984

[FeSi93] FERSTL, O. K.; SINZ, E. J.: *Geschäftsprozessmodellierung*. In: Wirtschaftsinformatik, 35 (1993) 6, S. 589–592

[FeSi94a] FERSTL, O. K.; SINZ, E. J.: *Der Ansatz des Semantischen Objektmodells (SOM) zur Modellierung von Geschäftsprozessen*. In: *Bamberger Beiträge zur Wirtschaftsinformatik Nr. 21*. Universität Bamberg, 1994

[FeSi94b] FERSTL, O. K.; SINZ, E. J.: *From Business Process Modeling to the Specification of Distributed Business Application Systems - An Object-Oriented Approach*. In: *Bamberger Beiträge zur Wirtschaftsinformatik Nr. 20*. Universität Bamberg, 1994

[FeSi95a] FERSTL, O. K.; SINZ, E. J.: *Der Ansatz des Semantischen Objektmodells (SOM) zur Modellierung von Geschäftsprozessen*. In: Wirtschaftsinformatik, 37 (1995) 3, S. 209–220

[FeSi95b] FERSTL, O. K.; SINZ, E. J.: *Re-Engineering von Geschäftsprozessen auf der Grundlage des SOM-Ansatzes*. In: *Bamberger Beiträge zur Wirtschaftsinformatik Nr. 26*. Universität Bamberg, 1995

[FeSi96a] FERSTL, O. K.; SINZ, E. J.: *Flexible Organizations Through Object-oriented and Transaction-oriented Information Systems*. In: *Bamberger Beiträge zur Wirtschaftsinformatik Nr. 37*. Universität Bamberg, 1996

[FeSi96b] FERSTL, O. K.; SINZ, E. J.: *Geschäftsprozessmodellierung im Rahmen des Semantischen Objektmodells*. In: VOSSEN, G. (Hrsg.); BECKER, J. (Hrsg.): *Geschäftsprozeßmodellierung und Workflow-Management*. Bonn et al.: Thomson, 1996, S. 47–62

[FeSi97] FERSTL, O. K.; SINZ, E. J.: *Modeling of Business Systems Using the Semantic Object Model (SOM) - A Methodological Framework*. In: *Bamberger Beiträge zur Wirtschaftsinformatik Nr. 43*. Universität Bamberg, 1997

[FeSi06] FERSTL, O. K.; SINZ, E. J.: *Grundlagen der Wirtschaftsinformatik*. 5. Auflage, München, Wien: Oldenbourg Verlag, 2006

[Flan05] FLANAGAN, D.: *Java in a Nutshell: A Desktop Quick Reference*. 5th Edition, Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2005

[Fowl03] FOWLER, M.: *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003

[Fran80] FRANK, J.: *Standard-Software: Kriterien und Methoden*. Rolf Müller GmbH, 1980

[Fran99] FRANK, U.: *Component Ware - Software-technische Konzepte und Perpektiven für die Gestaltung betrieblicher Informationssysteme*. In: Information Management & Consulting, (1999), S. 11–18

[Fran03] FRANKEL, D.: *Model Driven Architecture: Applying MDA to Enterprise Computing*. Indianapolis: Wiley, 2003

[Fric95] FRICK, A.: *Der Software-Entwicklungsprozeß. Ganzheitliche Sicht*. München, Wien: Hanser Verlag, 1995

[Gada01] GADATSCH, A.: *Management von Geschäftsprozessen. Methoden und Werkzeuge für die IT-Praxis ; eine Einführung für Studenten und Praktiker*. Braunschweig u.a.: Vieweg, 2001

[Gam+95] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J.: *Design Patterns: Elements of reusable object-oriented software*. Reading, Massachusetts: Addison-Wesley, 1995

[Gamm92] GAMMA, E.: *Objektorientierte Softwareentwicklung am Beispiel von ET++*. Berlin: Springer, 1992

[GaSh93] GARLAN, D.; SHAW, M.: *An Introduction to Software Architecture*. In: AMBRIOLA, V. (Hrsg.); TORTORA, G. (Hrsg.): *Advances in Software Engineering and Knowledge Engineering*. Singapore: World Scientific Publishing Company, 1993, S. 1–39

[Gimn91] GIMNICH, R.: *Usability Engineering and User Interface Management*. In: DUCE, D. A. (Hrsg.); GOMES, M. R. (Hrsg.); HOPGOOD, F. R. A. (Hrsg.); LEE, J. R. (Hrsg.): *UIMS: Proceedings of the Workshop on User Interface Management Systems and Environments on User Interface Management and Design*. New York, NY, USA: Springer-Verlag New York, Inc., 1991, S. 113–122

[Goo+02] GOODWIN, R.; GOH, S. F.; WU, F. Y.: *Instance-level access control for business-to-business electronic commerce*. In: IBM Systems Journal, Volume 41 (2002) Number 2, S. 303–317

[Gras03] GRASL, O.: *J2EE und .NET: kein Vergleich?* In: HMD - Praxis Wirtschaftsinformatik, 230 (2003)

[Gree85] GREEN, M.: *Report on Dialogue Specification Tools*. In: PFAFF, G. (Hrsg.): *User Interface Management Systems*. Berlin: Springer-Verlag, 1985

[Grif98] GRIFFEL, F.: *Componentware: Konzepte und Techniken eines Softwareparadigmas*. Heidelberg: dpunkt-Verlag, 1998

[Grou04] GROUP, G.: *The Gartner Glossary of Information Technology Acronyms and Terms*. Gartner Group, Forschungsbericht, 2004

[Götz95] GÖTZE, R. (Hrsg.): *Dialogmodellierung für multimediale Benutzerschnittstellen*. Leipzig: Teubner, 1995

[Habe93] HABERMANN, H.-J.: *Repository*. München, Wien: Oldenbourg Verlag, 1993

[HaCh94] HAMMER, M.; CHAMPY, J.: *Reengineering the Corporation. A Manifesto for Business Revolution.* London: Brealey, 1994

[HaMe04] HAROLD, E. R.; MEANS, W. S.: *XML in a Nutshell*. 3. Auflage, Sebastopol, CA: O'Reilly Verlag, 2004

[Hamm99] HAMMEL, C.: *Generische Spezifikation betrieblicher Anwendungssysteme*. Aachen: Shaker Verlag, 1999

[Henn98] HENN, J.: *IBM San Francisco - Object-oriented infrastructure and reusable business components for distributed, multi-platform business applications - implemented entirely in Java.* In: Software - Concepts and Tools, 19 (1998) 1, S. 37–48

[Herd00] HERDA, N.: *Ein Ansatz zur formalen Repräsentation und automatisierten Generierung graphischer Benutzungsoberflächen*. Aachen, Universität Bamberg, Dissertation, 2000

[Heue97] HEUER, A.: *Objektorientierte Datenbanken*. 2. Auflage, Bonn [u.a.]: Addison-Wesley-Longman, 1997

[Hof+96] HOFMANN, C.; HORN, E.; KELLER, W.; RENZEL, K.; SCHMIDT, M.: *The Field of Software Architecture*. Technische Univerität München, Forschungsbericht TUM-I9641, 1996

[Hofm98] HOFMANN, F.: *Grafische Benutzungsoberflächen - Generierung aus OOA-Modellen*. Heidelberg: Spektrum Akademischer Verlag, 1998

[HoWo03] HOHPE, G.; WOOLF, B.: *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003

[HäRe83] HÄRDER, T.; REUTER, A.: *Principles of Transaction-Oriented Database Recovery*. In: ACM Computing Surveys, 15 (1983) 4, S. 287–317

[HuCa95] HUSSEY, A.; CARRINGTON, D.: *Comparing two User-Interface Architectures: MVC and PAC*, 1995

[Hump90] HUMPHREY, W. S.: *Managing the Software Process*. Reprinted with Corrections, Reading, Massachusetts: Addison-Wesley, 1990

[ISO01] ISO: *ISO/IEC 9126-1: Software Engineering - Product Quality - Part 1: Quality Model*, 2001

[ISO03] ISO: *ISO/IEC 11179-1: Information technology - Metadata Registries (MDR) - Part 1: Framework (Final Committee Draft)*. http://metadata-stds.org/11179-1/ [Zugriff am 01.10.2005], 2003

[Jac+92] JACOBSON, I.; CHRISTERSON, M.; JONSSON, P.; ÖVERGAARD, G.: *Object-Oriented Software-Engineering. A Use Case Driven Approach.* Workingham, England: Addison-Wesley, 1992

[Jaco93] JACOBSON, I.: *Object-Oriented Software Engineering: A Use Case Driven Approach*. überarb. Auflage, Reading, Massachusetts: Addison-Wesley, 1993

[Jing03] JINGYUE, L.: *A Survey on Reuse: Research Fields and Challenges*. http://www.idi.ntnu.no/grupper/su/courses/dif8901/Essay2003/essay2003-jingyue.pdf [Abruf 01.12.2005], 2003

[JoFo88] JOHNSON, R. E.; FOOTE, B.: *Designing Reusable Classes*. In: Journal of Object-Oriented Programming, 1 (1988) 2, S. 22–35

[John92] JOHNSON, R. E.: *Documenting Frameworks Using Patterns*. In: Proceedings OOPSLA 92, (1992), S. 63–76

[John97] JOHNSON, R. E.: *Components, Frameworks, Patterns (extended abstract)*. In: *ACM SIGS OFT Software engineering notes*. 1997, S. 10–17

[Jone84] JONES, T. C.: *Reusability in Programming: A Survey of the State of the Art.* In: IEEE Transactions on Software Engineering, 10 (1984) 5, S. 488–494

[KaBa94] KAZMAN, R.; BASS, L.: *Toward Deriving Software Architectures from Quality Attributes*, 1994

[KeEi99] KEMPER, A.; EICKLER, A.: *Datenbanksysteme*. München, Wien: Oldenbourg Verlag, 1999

[KeKi01] KERER, C.; KIRDA, E.: *Layout, Content and Logic Separation in Web Engineering*. In: MURUGESAN, S. (Hrsg.); DESPHANDE, Y. (Hrsg.): *Web Engineering. Managing Diversity and Complexity of Web Application Development*. Berlin, Heidelberg: Springer, 2001 LNCS, S. 135–147

[Kell97] KELLER, W.: *Mapping Objects to Tables - A Pattern Language*. In: BUSHMAN, F. (Hrsg.); RIEHLE, D. (Hrsg.): *Proceeding Of European Conference on Pattern Languages of Programming Conference (EuroPLOP)97; Irsee, Germany, 1997*, 1997

[Kic+97] KICZALES, G.; LAMPING, J.; MENHDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.-M.; IRWIN, J.: *Aspect-Oriented Programming*. In: AKŞIT, M. (Hrsg.); MATSUOKA, S. (Hrsg.): *Proceedings European Conference on Object-Oriented Programming* Band 1241. Berlin, Heidelberg, and New York: Springer-Verlag, 1997, S. 220–242

[Kil+93] KILBERTH, K.; GRYCZAN, G.; ZÜLLIGHOVEN, H.: *Objektorientierte Anwendungsentwicklung. Konzepte, Strategien, Erfahrungen.* Wiesbaden: Vieweg, 1993

[Kirc96] KIRCHMER, M.: *Geschäftsprozeßorientierte Einführung von Standardsoftware: Vorgehen zur Realisierung strategischer Ziele*. Wiesbaden: Gabler, 1996

[KoLo03] KOJARSKI, S.; LORENZ, D. H.: *Domain driven web development with WebJinn*. In: *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. Wiesbaden: ACM Press, 2003, S. 53–65

[Kope76] KOPETZ, H.: *Software-Zuverlässigkeit*. Hanser Verlag, 1976

[Kra+04] KRAFZIG, D.; BANKE, K.; SLAMA, D.: *Enterprise SOA : Service-Oriented Architecture Best Practices (The Coad Series)*. Prentice Hall PTR, November 2004. – ISBN 0131465759

[Krei04] KREISCHE, D.: *Geschäftsprozessmodellierung mit der Unified Modelling Language (UML)*, Universität Erlangen-Nürnberg, Dissertation, 2004

[Krüg98] KRÜGER, W.: *Management permanenten Wandels*. In: H. GLASER, A. W. (Hrsg.): *Organisation im Wandel der Märkte*. Wiesbaden: Gabler, 1998, S. 227–249

[KrPo88] KRASNER, G. E.; POPE, S. T.: *A cookbook for using the model-view controller user interface paradigm in Smalltalk-80*. In: Journal for Object-Oriented Programming, (1988) 1(3), S. 26–49

[Ladd03] LADDAD, R.: *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003

[Lamp74] LAMPSON, B. W.: *Protection*. In: ACM SIGOPS Operating Systems Review, Volume 8 (1974) Issue 1, S. 18–24

[LoWe05] LORENZ, W.; WEILKIENS, T.: *Objektorientierte Geschäftsprozessmodellierung*. In: STAHL, T. (Hrsg.); VÖLTER, M. (Hrsg.): *Modellgetriebene Softwareentwicklung*. Heidelberg: Dpunkt Verlag, 2005, S. 299–311

[Lude96] LUDEWIG, J.: *Von der Software-Zivilisation zur Software-Kultur: Die Vision einer verläßlichen Software-Umgebung.* In: *GI Jahrestagung*, 1996, S. 255–266

[LuTy03] LUBLINSKY, B.; TYOMKIN, D.: *Dissecting Service-Oriented Architectures*. In: Business Integration Journal, October (2003)

[Maa+05] MAASSEN, A.; SCHOENEN, M.; WERR, I.: *Grundkurs SAP R/3*. 3., durchges. und verb. Aufl., Wiesbaden: Vieweg, 2005

[MaBr03] MARSCHALL, F.; BRAUN, P.: *Model Transformations for the MDA with BOTL.* In: *Proceedings of the Workshop on Model Driven Architecture: Foundations and Applications, CTIT Technical Report TR-CTIT-03-27.* University of Twente, 2003

[Maie98] MAIER, P.: *Kritische Erfolgsfaktoren objektorientierter Frameworks*, Oktober 1998

[Mali97] MALISCHEWSKI, C.: *Generierung von Spezifikationen betrieblicher Anwendungssysteme auf der Basis von Geschäftsprozeßmodellen*. Aachen, Universität Bamberg, Dissertation, 1997

[MaRe00] MAIER, M. W.; RECHTIN, E.: *The Art of Systems Architecting*. 2nd edition, CRC Press LLC, 2000

[Mars05] MARSCHALL, F.: *Modelltransformationen als Mittel der modellbasierten Entwicklung von Software-Systemen*, Technische Universität München, Dissertation, 2005

[MaSc97] MAURER, G.; SCHWICKERT, A. C.: *Kritische Anmerkungen zur Prozeßorientierung*. In: WI, (1997)

[McCl08] MCCLURE, R. M.: *Garmisch 1968 und die Folgen*. In: ObjektSpektrum, (2008) Heft 6, S. 12–14

[McIl68] MCILROY, M.: *Mass Produced Software Components*. In: BUXTON, J. (Hrsg.); NAUR, P. (Hrsg.); RANDELL, B. (Hrsg.): *Software Engineering Concepts and Techniques, NATO Conference on Software Engineering*. 1968, S. 88–98

[MDA03] MDA: *MDA Guide Version 1.0.1*. http://www.omg.org/docs/omg/03-06-01.pdf, 2003

[MeHo92] MERTENS, P.; HOLZNER, J.: *WI - State Of The Art. Eine Gegenüberstellung von Integrationsansätzen der Wirtschaftsinformatik.* In: Wirtschaftsinformatik, 34 (1992) 1, S. 5–25

[Mert96] MERTENS, P.: *Individual- und Standardsoftware: tertium datur?* In: *GI Jahrestagung*, 1996, S. 55–81

[Mil+95] MILI, H.; MILI, F.; MILI, A.: *Reusing Software: Issues and Research Directions.* In: IEEE Trans. Software Eng., 21 (1995) 6, S. 528–562

[MoPu92] MONARCHI, D. E.; PUHR, G. I.: *A research typology for object-oriented analysis and design*. In: Commununications of the ACM, 35 (1992) 9, S. 35–47

[Mull90] MULLIN, M.: *Rapid Prototyping for Object-Oriented Systems*. Addison-Wesley, 1990

[MyRo92] MYERS, B. A.; ROSSON, M. B.: *Survey on User Interface Programming*. In: *Proceedings of the Conference on Human Factors in Computing Systems*. Monterey, CA, USA, 3–7 May 1992, S. 195–202

[NaRa68] NAUR, P. (Hrsg.); RANDELL, B. (Hrsg.): *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. 1968

[NATO94] NATO: *Standard for the Development of Reusable Software Components. Volume 1*, 1994

[Neum99] NEUMANN, K.: *Integritätsbedingungen in relationalen Datenbanken*. Egelsbach, Frankfurt u.a.: Verlag Dr. Hänsel-Hohenhausen, 1999

[Nie+92] NIERSTRASZ, O.; GIBBS, S.; TSICHRITZIS, D.: *Component-Oriented Software Development*. In: Commununications of the ACM, 35 (1992) 9, S. 160–165

[Noac01] NOACK, J. (Hrsg.): *Techniken der objekt-orientieren Softwareentwicklung*. Berlin, Heidelberg: Springer Verlag, 2001

[O00] O, O.: *SanFrancisco - Concepts and Facilities*, 2000

[O05a] O, O.: *The Apache Struts Framework*, 2005

[O05b] O, O.: *Ruby on Rails*, 2005

[O05c] O, O.: *Sun: JavaServer Faces*, 2005

[Oes+03] OESTEREICH, B.; WEISS, C.; SCHRÖDER, C.; WEILKIENS, T.; LENHARD, A.: *Objektorientierte Geschäftsprozessmodellierung mit der UML.* Heidelberg: Dpunkt Verlag, 2003

[Oest98] OESTEREICH, B.: *Objektorientierte Softwareentwicklung*. München, Wien: Oldenbourg Verlag, 1998

[Oest05] OESTEREICH, B.: *Objektorientierte Geschäftsprozessmodellierung und modellgetriebene Softwareentwicklung*. In: HMD - Praxis Wirtschaftsinformatik, 241 (2005), S. 27–33

[Ohle98] OHLENDORF, T.: *Architektur betrieblicher Referenzmodellsysteme: Konzept und Spezifikation zur Gestaltung wiederverwendbarer Norm-Software-Bausteine für die Entwicklung betrieblicher Anwendungssysteme*. Aachen, Universität Hildesheim, Dissertation, 1998

[OMG96] OMG: *Common Facilities RFP-4, Common Business Objects and Business Object Facility*, 1996

[Orf+97] ORFALI, R.; HARKEY, D.; EDWARDS, J.: *Abenteuer Client/Server: The Essential Client/Server Survival Guide*. Bonn: Addison Wesley, 1997

[OsTa01] OSSHER, H.; TARR, P.: *Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software*. In: Communications of the ACM, 44 (2001) 10, S. 43–50

[Pall01] PALLOS, M. S.: *Service-Oriented Architecture: A Primer*. In: EAI Journal, December (2001)

[Parn01a] PARNAS, D. L.: *On the Criteria to be Used in Decomposing Systems into Modules*. (2001), S. 145–155

[Parn01b] PARNAS, D. L.: *On the Design and Development of Program Families*. (2001), S. 193–213

[Pet+97] PETER, J.; VOLLMER, M.; STRIPF, W.: *IBM San Francisco - Anwendungsentwicklung mit Java-Geschäftsprozeß-Komponenten.* In: HMD - Praxis Wirtschaftsinformatik, 197 (1997), S. 76–90

[Pete00] PETERS, T.: *Kreatives Chaos*. München: Heyne, 2000

[Pich99] PICH, N.: *Das semantische Repository der IBM San Francisco Application Business Components for Java*. In: TUROWSKI, K. (Hrsg.): *Bericht zum 1. Workshop Komponentenorientierte betriebliche Anwendungssysteme (WKBA 1), 30.03.1999 in Magdeburg).* Band 1999, Rundbrief der GI-Fachgruppe 5.10 Informationssystem-Architekturen, 1999, S. 51–58

[Piet04] PIETSCH, C.: *Konzeption und Realisierung von Komponenten einer Software-Entwicklungs- und Laufzeitumgebung für die Umsetzung der Integrationsziele eines objektorientierten Software-Architekturmodells*, Universität Bamberg, Diplomarbeit, 2004

[PiFr95] PICOT, A.; FRANCK, E.: *Prozeßorganisation. Eine Bewertung der neuen Ansätze aus Sicht der Organisationslehre*. (1995), S. 13–38

[PoBl96] POMBERGER, G.; BLASCHEK, G.: *Software-Engineering: prototyping und objektorientierte Software-Entwicklung*. München;Wien: Hanser Verlag, 1996

[Pom+92] POMBERGER, G.; PREE, W.; STRITZINGER, A.: *Methoden und Werkzeuge für das Prototyping und ihre Integration.* In: Informatik, Forschung und Entwicklung, 7 (1992) 2, S. 49–61

[Popp94] POPP, K. M.: *Spezifikation der fachlichen Klassen-Beziehungs-Struktur objektorientierter Anwendungssysteme auf Grundlage von Modellen der betrieblichen Diskurswelt*, Universität Bamberg, Dissertation, 1994

[Pree97] PREE, W.: *Komponentenbasierte Softwareentwicklung mit Frameworks*. Heidelberg: dpunkt.Verlag, 1997

[Prei99] PREIM, W.: *Entwicklung interaktiver Anwendungssysteme*. Berlin: Springer-Verlag, 1999

[Rand79] RANDELL, B.: *Software engineering in 1968*. In: *ICSE '79: Proceedings of the 4th international conference on Software engineering*. Piscataway, NJ, USA: IEEE Press, 1979. – ISBN none, S. 1–10

[Raue96] RAUE, H.: *Wiederverwendbare betriebliche Anwendungssysteme: Grundlagen und Methoden ihrer objektorientierten Entwicklung*. Wiesbaden, Universität Bamberg, Dissertation, 1996

[Reed02] REED, P. R.: *Developing Applications with Java and UML.* Boston: Pearson, 2002

[Reif01] REIFER, D. J.: *Implementing a Practical Reuse Program for Software Components*. (2001), S. 453–466

[Rüff99] RÜFFER, T.: *Referenzgeschäftsprozeßmodellierung eines Lebensversicherungsunternehmens*. In: SINZ, E. J. (Hrsg.): *Modellierung betrieblicher Informationssysteme. PROCEEDINGS er MobIS-Fachtagung 1999*. Gesellschaft für Informatik, 1999, S. 86–107

[Robr05] ROBRA, C.: *Ein komponentenbasiertes Software-Architekturmodell zur Entwicklung betrieblicher Anwendungssysteme*, Universität Bamberg, Dissertation, 2005

[RoJo97a] ROBERTS, D.; JOHNSON, R. E.: *Evolve Frameworks into Domain-Specific Languages*. In: *Pattern Languages of Program Design 3*. Addison Wesley, 1997

[RoJo97b] ROBERTS, D.; JOHNSON, R. E.: *Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks*. In: *Pattern Languages of Program Design 3*. Addison Wesley, 1997

[Rose67] ROSE, M.: *Computers, Managers and Society*. Harmondsworth, 1967

[Rose99] ROSEMANN, M.: *Gegenstand und Aufgaben des Integrationsmanagements*. In: SCHEER, A.-W. (Hrsg.); ROSEMANN, M. (Hrsg.); SCHÜTTE, R. (Hrsg.): *Integrationsmanagement. Arbeitsbericht Nr. 65.* Münster: Institut für Wirtschaftsinformatik der Westfälischen Wilhelms-Universität Münster, 1999, S. 5–18

[Royc70] ROYCE, W. W.: *Managing the development of large software systems: concepts and techniques*. In: *ICSE '87: Proceedings of the 9th international conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987 (Nachdruck von 1970). – ISBN 0–89791–216–0, S. 328–338

[Rum+91] RUMBAUGH, J.; BLAHA, M.; PREMERLANI, W.; EDDY, F.; LORENSEN, W.: *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991

[Rumb93] RUMBAUGH, J.: *Objects in the Twilight Zone*. In: The Journal of Object-Oriented Programming, (1993) 6(3), S. 18–24

[Rump01] RUMPE, B.: *Extreme Programming - Back to Basics?* In: ENGELS, G. (Hrsg.); OBERWEIS, A. (Hrsg.); ZÜNDORF, A. (Hrsg.): *Modellierung 2001, Workshop der Gesellschaft für Informatik e.V.(GI) 28.-30.3.2001, Bad Lippspringe*, GI-Edition, Lecture Notes in Informatics, 2001, S. 121–132

[Same97] SAMETINGER, J.: *Software Engineering with Reusable Components*. New York, NY, USA: Springer-Verlag New York, Inc., 1997

[Sch+02] SCHISSLER, M.; MANTEL, S.; FERSTL, O. K.; SINZ, E. J.: *Kopplungsarchitekturen zur überbetrieblichen Integration von Anwendungssystemen und ihre Realisierung mit SAP R/3*. In: Wirtschaftsinformatik, (2002), S. 459–468

[Sch+05] SCHISSLER, M.; MANTEL, S.; ECKERT, S.; FERSTL, O. K.; SINZ, E. J.: *Entwicklungsmethodiken zur Integration von Anwendungssystemen in überbetrieblichen Geschäftsprozessen - ein Überblick über ausgewählte Ansätze.* In: FERSTL, O. K. (Hrsg.); SINZ, E. J. (Hrsg.); ECKERT, S. (Hrsg.); ISSELHORST, T. (Hrsg.): *Wirtschaftsinformatik 2005*, Physica-Verlag, 2005. – ISBN 3–7908–1574–8, S. 1463–1482

[Sche88] SCHEER, A.-W.: *Wirtschaftsinformatik - Informationssysteme im Industriebetrieb*. 2. Auflage, Berlin: Springer, 1988

[Sche98] SCHEER, A.-W.: *ARIS - Modellierungsmethoden, Metamodelle, Anwendungen*. 3., völlig neubearb. und erw. Aufl., Berlin: Springer-Verlag, 1998

[Schm90] SCHMID, P.: *SAA - die IBM-System-Anwendungsarchitektur: Grundlagen, Konzepte, Trends*. Vaterstetten: IWT Verlag, 1990

[Schm96] SCHMID, H. A.: *Design Patterns for Constructing the Hot Spots of a Manufacturing Framework.* In: JOOP, 9 (1996) 3, S. 25–37

[Schm97] SCHMID, H. A.: *Systematic Framework Design by Generalization*. In: Communications of the ACM, 40 (1997) 10, S. 48–51

[Schm00] SCHMITZER, B.: *Beiträge zur Verwendung der Framework-Technologie bei der Entwicklung und Einführung von Systemen der betrieblichen Informationsverarbeitung*. Berlin, Universität Erlangen-Nürnberg, Dissertation, 2000

[Schm01] SCHMITZ, K.: *Virtualisierung von wirtschaftswissenschaftlichen Lehr- und Lernsituationen. Konzeption eines Application Framework.* Wiesbaden, Universität Bamberg, Dissertation, 2001

[Schr01] SCHRYEN, G.: *Komponentenorientierte Softwareentwicklung in Softwareunternehmen: Konzeption eines Vorgehensmodells zur Einführung und Etablierung*. Wiesbaden, RWTH Aachen, Dissertation, 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.): *Bericht zum 1. Workshop Komponentenorientierte betriebliche Anwendungssysteme (WKBA 1), 30.03.1999 in Magdeburg).* Band 1999, Rundbrief der GI-Fachgruppe 5.10 Informationssystem-Architekturen, 1999, S. 75–88

[ScNa96] SCHULTE, R. W.; NATIS, Y. V.: *"Service-Oriented" Architectures*. Gartner Group, Forschungsbericht Part I, 1996

[ScSt83] SCHLAGETER, G.; STUCKY, W.: *Datenbanksysteme, Konzepte und Modelle*. Teubner Verlag, 1983

[ShGa96] SHAW, M.; GARLAN, D.: *Software Architecture: Perspectives on an Emerging Discipline*. New Jersey: Prentice-Hall, 1996

[ShMe92] SHLAER, S.; MELLOR, S.: *Object Lifecycles: Modeling the World in States*. Englewood Cliffs, New Jersey: Prentice Hall, 1992

[Shne02] SHNEIDERMAN, B.: *User Interface Design*. Bonn: mitp-Verlag, 2002

[Sinz87] SINZ, E. J.: *Datenmodellierung betrieblicher Probleme und ihre Unterstützung durch ein wissensbasiertes Entwicklungssystem*. Universität Regensburg: Habilitationsschrift, 1987

[Sinz91] SINZ, E. J.: *Objektorientierte Analyse (ooA).* In: Wirtschaftsinformatik, 33 (1991) 5, S. 455–457

[Sinz92] SINZ, E. J.: *Datenmodellierung im Strukturierten Entity-Relationship-Modell (SERM)*. In: *Bamberger Beiträge zur Wirtschaftsinformatik Nr. 10*. Universität Bamberg, 1992

[Sinz95] SINZ, E. J.: *Ansätze zur fachlichen Modellierung betrieblicher Informationssysteme. Entwicklung, aktueller Stand und Trends*. In: *Bamberger Beiträge zur Wirtschaftsinformatik Nr. 35*. Universität Bamberg, 1995

[Sinz96] SINZ, E. J.: *Ansätze zur fachlichen Modellierung betrieblicher Informationssysteme. Entwicklung, aktueller Stand und Trends.* In: HEILMANN, H. (Hrsg.); HEINRICH, L. J. (Hrsg.); ROITHMAYR, F. (Hrsg.): *Information Engineering. Wirtschaftsinformatik im Schnittpunkt von Wirtschafts-, Sozial- und Ingenieurwissenschaften*. München, Wien: Vahlen, 1996, S. 123–143

[Sinz97] SINZ, E. J.: *Architektur betrieblicher Informationssysteme*. In: *Bamberger Beiträge zur Wirtschaftsinformatik Nr. 40*. Universität Bamberg, 1997

[Sinz99] SINZ, E. J.: *Konstruktion von Informationssystemen*. In: *Bamberger Beiträge zur Wirtschaftsinformatik Nr. 53*. Universität Bamberg, 1999

[Sinz00] SINZ, E. J.: *Entwicklung betrieblicher Anwendungssysteme II*. Skript, 2000

[Somm07] SOMMERVILLE, I.: *Software Engineering*. 8. Auflage, München: Pearson Studium, 2007

[Stae90] STAEHLE, W.: *Management. Eine verhaltenswissenschaftliche Perspektive*. München: Vahlen, 1990

[Star96] STARY, C.: *Interaktive Systeme - Softwareentwicklung und Softwareergonomie*. Braunschweig: Vieweg-Verlag, 1996

[Star05] STARKE, G.; VERLAG, H. (Hrsg.): *Effektive Software-Architekturen - Ein Leitfaden*. 2. Auflage, München; Wien, 2005

[StBo03] STURM, T.; BOGER, M.: *Softwareentwicklung auf Basis der Model Driven Architecture.* In: HMD - Praxis Wirtschaftsinform., 231 (2003)

[Stei97] STEIN, W.: *Objektorientierte Analysemethoden*. 2., völlig neu bearb. Aufl., Heidelberg: Spektrum Akademischer Verlag, 1997

[StSc00] STEINMANN, H.; SCHREYÖGG, G.: *Management. Grundlagen der Unternehmensführung. Konzepte - Funktionen - Fallstudien*. 4. Aufl., Wiesbaden: Gabler, 2000

[StVö05] STAHL, T.; VÖLTER, M.: *Modellgetriebene Softwareentwicklung*. Heidelberg: Dpunkt Verlag, 2005

[Summ97] SUMMERS, R. C.: *Secure Computing*. New York, NY: McGraw-Hill Verlag, 1997

[Szyp02] SZYPERSKI, C.: *Component Software: Beyond Object-Oriented Programming*. Second Edition, Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002

[Tane01] TANENBAUM, A. S.: *Modern Operating Systems*. 2nd Edition, Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001

[Tinn95] TINNILÄ, M.: *A Strategic Perspective to Business Process Redesign*. In: Business Process Re-engineering & Management, 1 (1995) 1, S. 44–59

[Trac88] TRACZ, W.: *Reusability Comes of Age*. (1988), S. 89–91

[Türk03] TÜRKER, C.: *SQL:1999 & SQL:2003*. Heidelberg: dpunkt Verlag, 2003

[Turo99] TUROWSKI, K.: *Architekturkonzept zur Realisierung flexibel erweiterbarer Fachkomponenten*. In: SINZ, E. J. (Hrsg.): *Modellierung betrieblicher Informationssysteme. PROCEEDINGS der MobIS-Fachtagung 1999*. Gesellschaft für Informatik, 1999, S. 132–152

[UML04] UML: *Unified Modeling Language: Superstructure*. http://www.omg.org/docs/ptc/04-10-02.pdf, 2004

[Völt05] VÖLTER, M.: *Modellgetriebene Softwareentwicklung*. http://www.voelter.de/data/articles/MDSD.pdf [Abruf 02.12.2005], 2005

[Voss00] VOSSEN, G.: *Datenmodelle, Datenbanksprachen und Datenbankmanagementsysteme*. 4. Auflage, München: Oldenbourg, 2000

[WaMa04] WANNER, G.; MAUTE, O.: *Persistenzoperationen mit Mustern optimieren*. In: Java Spektrum, Ausgabe 1 (2004), Februar/März Nr. 49, S. 48–52

[Wand93] WANDMACHER, J.: *Software-Ergonomie*. Berlin: de Gruyter-Verlag, 1993

[Wei+89] WEINAND, A.; GAMMA, E.; MARTY, R.: *Design and Implementation of ET++, a Seamless Object-Oriented Application Framework.* In: Structured Programming, 10 (1989) 2, S. 63–87

[Whit02] WHITEHEAD, K.: *Component-Based Development - Principles and Planning for Business Systems*. Addison-Wesley, 2002

[WiJo90] WIRFS-BROCK, R.; JOHNSON, R. E.: *Surveying Current Research in Object-Oriented Design.* In: Communications of the ACM, 33 (1990) 9, S. 104–124

[Will85] WILLMER, H.: *Systematische Software-Qualitätssicherung anhand von Qualitäts- und Produktmodellen*. Informatik-Fachberichte, Springer, 1985

[Wint98] WINTER, R.: *Informationsableitung in betrieblichen Anwendungssystemen*. Braunschweig/Wiesbaden, Johann Wolfgang Goethe Universität Frankfurt a.M., Dissertation, 1998

[Wolf98] WOLFF, E.: *San Francisco: Framework für Geschäftsanwendungen*. In: iX, (1998) 7, S. 116–121

[YoCo79] YOURDON, E.; CONSTANTINE, L. L.: *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1979

[Zach05] ZACHARIAS, R.: *Serviceorientierung: Der OO-König ist tot, es lebe der SOA-König!* In: ObjektSpektrum, (2005) Heft 2, S. 43–52

[Zeid99] ZEIDLER, F.: *Organisationsmodellierung auf der Basis eines Enterprise Application Frameworks.* In: Rundbrief der GI-Fachgruppe 5.10 Informationssystem-Architekturen, 1999 (1999) 1

[ZiBe00] ZIMMERMANN, J.; BENEKEN, G.: *Verteilte Komponenten und Datenbankanbindung*. Addison-Wesley, 2000

[Züll98] ZÜLLIGHOVEN, H.; ZÜLLIGHOVEN, H. (Hrsg.): *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz.* Heidelberg: dpunkt.verlag, 1998

Index

Danksagung

Ich danke meinem Doktorvater, Herrn Prof. Dr. Otto K. Ferstl, für die stets konstruktive Kritik und Betreuung dieser Arbeit, Herrn Dr. Klaus Schmitz für die vielen wertvollen Gespräche und Anregungen während der Zeit am Centrum für betriebliche Informationssysteme sowie Herrn Thomas Reeg für seinen fachlichen Rat und seine freundschaftliche Begleitung. Für seine Bestärkung und die eingehende Lektüre des Textes danke ich Dr. Matthias Weichelt. Ohne das Verständnis und die Unterstützung meiner Frau Dr. Lea Maria Wilkens hätte ich die Arbeit so nicht vollenden können. Widmen möchte ich sie meinen Eltern, für ihren Rückhalt und ihre Ermutigung.