Softwarearchiteckturen

Dirk Bäumer

für die rahmenwerkbasierte Konstruktion großer Anwendungssysteme

Dirk Bäumer

Softwarearchitekturen für die rahmenwerkbasierte Konstruktion großer Anwendungssysteme

Vom Fachbereich Informatik der Universität Hamburg
zur Erlangung der Würde eines Doktors der Natur-
wissenschaften (Dr. rer. nat.) genehmigte Abhandlung.

Vorgelegt von Dirk Bäumer aus Zürich

Hauptgutachter:	Prof. Dr. Heinz Züllighoven, Universität Hamburg.
Zweitgutachter:	Prof. Dr. Gustav Pomberger, Universtät Linz.
Drittgutachter:	Prof. Dr. Arno Rolf, Universität Hamburg.

Danksagung

Viele Menschen haben – direkt oder indirekt – dazu beigetragen, daß diese Arbeit entstanden ist. Mein erster Dank gilt allen ehemaligen Kollegen und Kolleginnen der RWG GmbH in Stuttgart, mit denen ich bei der Entwicklung des GEBOS-Systems zusammengearbeitet habe. Für zahlreiche spannende und fruchtbare Diskussionen möchte ich mich besonders bei Judith Waibel, Angelika Wiedemann, Rolf Knoll, Reinhard Matz, Peter Rieth, Harry Riethmüller, Herwig Scheidel, Frank Schneider, Wolf Siberski und Volker Weimer bedanken.

Zu danken habe ich weiter dem Management der RWG, das es mir ermöglicht hat, meine im Rahmen des GEBOS-Projektes gesammelten Erfahrungen einer interessierten Fachöffentlichkeit zu präsentieren. Herrn Zentner, Herrn Wurster und Raimer Rudolph, die stets ein offenes Ohr für mein Anliegen gezeigt haben, danke ich für die gute Zusammenarbeit.

Meinem Betreuer, Herrn Professor Dr. Heinz Züllighoven danke ich für die ständige Bereitschaft, sich mit meiner Arbeit in konstruktiven Diskussionen auseinanderzusetzen. Auf diese Weise hat er unverzichtbar dazu beigetragen, meine softwaretechnischen Ideen in einen breiteren wissenschaftlichen Kontext einzubetten.

Bei Herrn Professor Dr. Gustav Pomberger bedanke ich mich sehr für die zahlreichen und wertvollen Anregungen zu meiner Arbeit. Dank auch an Frau Professor Dr. Christiane Floyd, die mir während meines Forschungsaufenthaltes am Arbeitsbereich Softwaretechnik der Universität Hamburg eine Arbeitsumgebung bereitgestellt hat, die anregende Diskussionen und wissenschaftlichen Austausch gefördert hat.

Bei Walter Bischofberger möchte ich mich für alles persönliche Mutmachen und die fachlichen Gespräche bedanken, die mir einen wesentlichen Anstoß für diese Arbeit gegeben haben. Mein besonderer Dank gilt auch Guido Gryczan, der mich jederzeit mit freundschaftlicher Zuwendung bei meinem Vorhaben unterstützt hat. Dirk Riehle danke ich für den vielfältigen fachlichen Austausch während der letzten Jahre. Viele der softwaretechnischen Konzepte, die ich hier beschreibe, sind in Diskussionen mit ihm herangereift.

Ein außergewöhnlicher und von Herzen kommender Dank gilt meiner Lebensgefährtin Martina. Denn ohne ihr ständiges Mutmachen – und zwar immer dann, wenn es notwendig war –, ihre stetige Bereitschaft, die fachlichen Konzepte mit mir zu diskutieren und ihre unendliche Geduld, die Arbeit immer wieder auf Rechtschreibfehler zu überprüfen, wäre diese Arbeit nicht das geworden, was sie heute ist.

Inhaltsverzeichnis

1 Zielsetzung der Arbeit

"Indeed, the essence of systems integration is finding ways to treat as subsystems today those things that were systems last week." ([SDK+95], S. 320)

Anwendungssysteme werden heute in weiten Bereichen der Wissenschaft und Wirtschaft eingesetzt. Das Spektrum der Aufgaben, die in diesen Bereichen mit Hilfe von Softwaresystemen unterstützt werden, wächst ständig. In der Regel entstehen dabei Softwarelandschaften, die häufig aus einer Vielzahl von Anwendungssystemen bestehen. So werden im Bankenbereich zunehmend Softwarelandschaften etabliert, die einen Mitarbeiter nicht mehr nur bei den üblichen Kontogeschäften, sondern auch bei der Beratung eines Kunden im Kredit-, Anlage- und Wertpapiergeschäft unterstützen.

Solche Softwarelandschaften sind meistens dadurch gekennzeichnet, daß die in ihnen zur Verfügung gestellten Anwendungssysteme nur eine geringe Integration untereinander aufweisen. Diese geringe Integration erschwert eine optimale Nutzung der verschiedenen Anwendungssysteme, da Informationen zwischen den Systemen schwer ausgetauscht und die Kooperation zwischen Mitarbeitern, die die Anwendungssysteme zur Erledigung ihrer Aufgaben verwenden, softwaretechnisch nur minimal unterstützt werden kann.

Um eine Softwarelandschaft von integrierten Anwendungssystemen zu erstellen, empfiehlt die Literatur (vgl. [Rum91] und [SM93]) die Konstruktion eines unternehmensweiten Modells, das mit dem Anspruch verbunden ist, das Unternehmen oder den entsprechenden Anwendungsbereich in nur einem Modell umfassend zu beschreiben. In der traditionellen Softwareentwicklung geschieht dies in Form eines Datenmodells, bei einer objektorientierten Vorgehensweise durch ein sogenanntes Business Object Model. Allerdings ist die Konstruktion eines solchen unternehmensweiten Modells mit den folgenden Problemen verbunden (vgl. [CD94]):

- Auf Grund der Komplexität solch großer Anwendungsbereiche ist die Konstruktion eines einheitlichen Modells äußerst schwierig.
- Für die Erstellung des Modells ist eine fachliche Aufarbeitung des gesamten Anwendungsbereichs notwendig, die nur durch eine Analyse der im Anwendungsbereich relevanten Aufgaben und Tätigkeiten erfolgen kann. Eine solche Analyse nimmt für komplexe Anwendungsbereiche, wie etwa das Bankgeschäft, einige Jahre in Anspruch.
- Das erstellte Modell entspricht bei seiner Fertigstellung (wenn es soweit kommt) meist nicht mehr der Arbeitsrealität, da sich diese im Laufe des Analyseprozesses verändert hat.

Zwar findet man in der Literatur (vgl. [CY91], [Jac92] und [Rum93]) Regeln, wie komplexe Anwendungsbereiche in kleinere Bereiche zu strukturieren sind, die Bereiche werden aber meistens unter rein technologischen Gesichtspunkten definiert. So wird beispielsweise häufig

eine Strukturierung in Teilbereiche entlang der Kriterien Präsentation, Kontrollfluß und Datenobjekt empfohlen. Diese Art der Aufteilung verhindert zum einem eine unabhängige Modellierung der so entstandenen Teilbereiche (vgl. [BBL+94] und [BBL+96]), zum anderen fördert sie nicht die Konstruktion von Softwarekomponenten, die sich einfach zu Anwendungssystemen kombinieren lassen. Beide Punkte sind allerdings notwendig, um integrierte Anwendungssysteme für komplexe Anwendungsbereiche erstellen zu können.

1.1 Die untersuchten Fragestellungen

Das Anliegen dieser Dissertation ist es daher, neue Strukturierungs- und Konstruktionsansätze zu beschreiben, die es ermöglichen, einen komplexen Anwendungsbereich so in Teilbereiche zu organisieren, daß diese getrennt voneinander konstruiert werden können. Das Erstellen eines unternehmensweiten Modells ist auf Grund der in dieser Arbeit präsentierten Vorgehensweise somit nicht mehr notwendig.

Auf diese Weise lassen sich komplexere Anwendungsbereiche als heute softwaretechnisch modellieren. Anwendungssysteme können zudem einfach in Form von Ausbaustufen, etwa durch den Einsatz von Komponententechnologien, entwickelt werden. Zusätzlich wird eine evolutionäre Vorgehensweise bei der Entwicklung integrierter Softwaresysteme für komplexe Anwendungsbereiche vereinfacht.

Neben neuen Strukturierungs- und Konstruktionsansätzen existieren auch andere Faktoren, die die Entwicklung von komplexen Softwarelandschaften vereinfachen. Hier wäre beispielsweise ein veränderter Entwicklungsprozeß zu nennen, bei dem Softwareentwicklung nicht als Produktionsprozeß unter Verwendung eines Phasenmodells, sondern als Lern- und Kommunikationsprozeß aller beteiligten Gruppen unter Einsatz einer evolutionären Vorgehensweise gesehen wird (vgl. [Flo87]). Genauso können sich Veränderungen an der Organisationsstruktur des Unternehmens, das die Anwendungssysteme entwickelt, positiv auswirken (vgl. [BBE95] und [KGZ+94]). Beide Faktoren werden allerdings im weiteren nicht untersucht. Die Dissertation konzentriert sich ausschließlich auf den softwaretechnischen Bereich.

Im Detail beschäftigt sich die Arbeit mit den folgenden Fragestellungen:

- Welche für die softwaretechnische Struktur des Anwendungssystems relevanten Beziehungen existieren zwischen dem Softwaresystem, der Technik, die für seine Konstruktion eingesetzt wird, und dem Anwendungsbereich, der die fachliche Ausgangsbasis für das Anwendungssystem bildet?
- Wie können komplexe Anwendungsbereiche so in kleinere Bereiche strukturiert werden, daß sich diese weitgehend getrennt voneinander analysieren, modellieren und implementieren lassen? Welche Abhängigkeit besteht dabei zu der bereits existierenden Organisationsstruktur des Anwendungsbereichs?
- Welche *Softwarearchitekturen* sind geeignet, um die Strukturen der eingesetzten Technik und des Anwendungsbereichs softwaretechnisch umzusetzen? Die Softwarearchitektur muß zudem die Konfektionierung von Anwendungssystemen auf der Basis von bereits entwickelten Komponenten berücksichtigen.

- Wie müssen Klassenhierarchien und Rahmenwerke (engl. *frameworks*) in Bibliotheken partitioniert werden, damit sie sich als Softwarekomponenten zu größeren Anwendungssystemen kombinieren lassen?
- Sind die vorhandenen objektorientierten Modellierungsmittel Objekt, Klasse, Vererbung und Komposition ausreichend, um die entwickelten Softwarearchitekturen implementierungstechnisch umzusetzen?

1.2 Einordnung der Arbeit

Mein Beitrag zur Diskussion über Softwarearchitekturen für große rahmenwerkbasierte Anwendungssysteme ist geprägt durch meine mehrjährige praktische Tätigkeit in industriellen Entwicklungsprojekten, die Mitarbeit im Arbeitskreis Prototyping der Fachgruppe 2.1.6 der Gesellschaft für Informatik sowie durch meinen Forschungsaufenthalt am Arbeitsbereich Softwaretechnik der Universität Hamburg.

Eine wesentliche Grundlage dieser Arbeit ist die an diesem Arbeitsbereich entwickelte WAM-Methode (vgl. [BZ90]). Die Abkürzung WAM steht dabei für Werkzeug, Automat und Material (vgl. [Gry96]). Die WAM-Methode ist speziell auf eine evolutionäre Entwicklung objektorientierter Anwendungssysteme ausgerichtet. Ihre besondere Eigenschaft liegt in der Bereitstellung der Entwurfsmetaphern Werkzeug, Automat und Material sowie der speziellen Dokumenttypen Szenario, Systemvision und Glossar zur Konstruktion objektorientierter Softwaresysteme (vgl. [GZ92] und [BGZ95]).

WAM wurde bei der Rechenzentrale der württembergischen Genossenschaften (RWG) als Ausgangspunkt für die Entwicklung der organisationsspezifischen GEBOS[1]-Methode verwendet. Sie bildet die methodische Grundlage für die Entwicklung der GEBOS-Softwarelandschaft, die aus einer Reihe von integrierten Anwendungssystemen für das Bankgeschäft besteht. Während meiner Tätigkeit bei der RWG habe ich sowohl die GEBOS-Methode als auch die Softwarearchitektur der Anwendungsfamilie aktiv geprägt. Ein Teil der dabei entstandenen Ergebnisse wurde bereits in einer Reihe von Publikationen, an denen ich als Autor beteiligt bin, veröffentlicht (vgl. [Bae93], [BBS+95], [BBL+96], [BGK+97], [BKG+96], [BR97] und [BRS+97]). Diese Publikationen bilden neben der WAM-Methode ebenfalls eine starke inhaltliche Grundlage für die vorliegende Arbeit. Ihr Hauptanliegen ist es, die WAM-Methode konzeptionell um Softwarearchitekturen und Entwurfsmuster für die Konstruktion großer rahmenwerkbasierter Anwendungssysteme zu erweitern.

Auf Grund meines Erfahrungshintergrundes beschäftige ich mich auf der Anwendungsseite mit integrierten Softwaresystemen, die einen Mitarbeiter bei der Erledigung seiner fachlichen Aufgaben in einer dienstleistungsorientierten Organisation softwaretechnisch unterstützen sollen. Bei diesen Softwaresystemen handelt es sich daher in der Regel um interaktive graphische Anwendungssysteme. Beispiele hierfür sind Anwendungsfamilien für Banken, Versicherungen und Krankenhäuser, aber auch für die professionelle Softwareentwicklung selbst.

[1] Das Akronym GEBOS steht dabei für: Genossenschaftliches Bürokommunikations- und Organisationssystem.

Die Diskussion über Softwarearchitekturen erfolgt vor dem Hintergrund der Arbeiten aus dem Umfeld von Mary Shaw und David Garlan (vgl. etwa [GAO95], [MKM+97], [SDK+95] und [SG96]).

Obwohl diese Arbeit in weiten Teilen auf Beispielen aus dem Bankenbereich beruht, soll hier keine domänenspezifische Softwarearchitektur (engl. Domain-Specific Software Architecture, DSSA) für das Bankgeschäft vorgestellt werden (vgl. [MG92], [Tra95a] und [Tra95b]). Die Entwicklung einer DSSA für eine bestimmte Domäne geht weit über die von mir vorgestellten Softwarearchitekturen hinaus. Sie umfaßt nach [Tra95b] neben einer Menge von Referenzanforderungen auch ein Domänenmodell (Szenarios, Datenglossar, Entity-Relationship- bzw. Klassen-Diagramme, Datenflussdiagramme, Zustandsübergangsdiagramme, Objektmodell) sowie eine Referenzarchitektur. Zudem legt eine domänenspezifische Softwarearchitektur auch Teile des Entwicklungsprozesses fest (vgl. [Tra95a]):

> "A DSSA is a process and infrastructure that supports the development of a Domain Model, Reference Requirements, and Reference Architecture for a family of applications within a particular problem domain. The expressed goal of a DSSA it to support the generation of applications within a particular domain (also known as a product-line)." ([Tra95a], S. 3)

Mein Verständnis von Entwurfsmustern ist geprägt durch das Werk „Design Patterns – Elements of Reusable Object-Oriented Software“ von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides (vgl. [GOF95]). Die von mir in dieser Arbeit vorgestellten Entwurfsmuster orientieren sich daher auch an dem in [GOF95] verwendeten Beschreibungsschema. Die Diskussion über Rahmenwerke erfolgt vor dem Hintergrund der Arbeiten von R. Johnson, B. Foote und V. F. Russo (vgl. [JF88], [Joh92], [JR91]).

Bei den zu untersuchenden Fragestellungen habe ich ausgeführt, daß die zu entwickelnde Softwarearchitektur eine komponentenbasierte Konstruktion der Anwendungsfamilie unterstützen muß. Obwohl der Begriff der komponentenbasierten bzw. komponentenorientierten Softwareentwicklung derzeit stark diskutiert wird (vgl. [NT95] und [Str97]), hat sich bisher noch kein einheitliches Verständnis darüber etabliert. Folgende Pole sind momentan erkennbar:

(a) *Entwicklung von generischen Softwarekomponenten*. Diese werden dann mittels einer Skripting-Sprache, die in der Regel als „Glue“ bezeichnet wird, zu Softwaresystemen kombiniert. Die Skripting-Sprache wird dabei zur Anpassung von inkompatiblen Schnittstellen, die zwischen einzelnen Komponenten existieren, und zur Implementierung der noch nicht vorhandenen fachlichen Logik verwendet.

(b) *Kompontenbasierter Ausbau eines Softwaresystems hin zu einer Anwendungsfamilie*. Ein Beispiel hierfür ist ein in mehreren Ausbaustufen entwickeltes Bankensystem für die Bereiche Anlagen-, Kredit- und Wertpapiergeschäft. Ziel einer derartigen Entwicklung ist es, bereits existierende Komponenten mit neu zu realisierenden Komponenten zu neuen Anwendungssystemen zu konfektionieren (vgl. [Pin95]). Da die Komponenten bereits sehr viel fachliche Funktionalität implementieren, müssen sie nicht nur technisch, sondern auch fachlich miteinander kombinierbar sein. Um dies

sicherzustellen, ist neben der Realisierung von Komponenten auch deren Einbettung in eine Softwarearchitektur notwendig, die sowohl die technische als aus auch die fachliche Kombinierbarkeit der Komponenten gewährleistet. Dieser Ansatz fokussiert somit auf die Wiederverwendung von Entwürfen und Architekturen bei der Realisierung von Komponenten. In [ND95] steht hierzu:

> "'Software reuse' is not something that can be achieved cheaply by arbitrarily introducing libraries or 'repositories' into an existing method. In fact, rather than focusing on software reuse, we must concentrate on reuse of design, of architecture and of expertise." ([ND95], S. 25)

Die vorliegende Arbeit beschäftigt sich mit einer komponentenbasierten Entwicklung von Anwendungsfamilien, wie sie unter (b) charakterisiert wurde.

Aus den zu untersuchenden Fragestellungen und der Einordnung der Arbeit ergibt sich ihr weiterer Aufbau. Dieser orientiert sich an der folgenden, prinzipiellen Vorgehensweise: die von mir zu untersuchenden Fragestellungen werden zuerst auf der Grundlage eigener praktischer Entwurfs- und Konstruktionserfahrungen diskutiert. Die hierbei entwickelten Thesen bzw. Lösungsvorschläge werden dann mit relevanten Ergebnissen aus den Wissenschaftsdisziplinen Semiotik und Organisationstheorie kombiniert, bevor ich sie abschließend dem Stand der Kunst in der Softwareentwicklung gegenüberstelle.

1.3 Weiterer Aufbau der Arbeit

Um die dargelegten Fragestellungen diskutieren und beantworten zu können, untersuche ich in Kapitel 2 zunächst die Konstruktion von Modellen im Softwareentwicklungsprozeß. Dies erscheint notwendig, da die Definition von Strukturen zur Organisation großer Softwaresysteme ein detailliertes Wissen über die im Softwareentwicklungsprozeß erstellten Modelle sowie über die Kontexte, in denen sie erstellt werden, voraussetzt. Im Vergleich zu anderen Autoren (vgl. [Jac92], [Rum91], [Som95]) stelle ich nicht so sehr die Tätigkeiten Analyse und Design selbst in den Vordergrund der Ausarbeitung, sondern betrachte vielmehr die dabei erstellten Modelle sowie ihre beeinflussenden Kontexte. In Anlehnung an [Lic93] verwende ich dabei den Begriff des Modells wie folgt:

Begriff 1.1 Modell

Ein Modell wird stets für einen speziellen Zweck erstellt. Es repräsentiert dabei entweder ein Abbild von etwas oder ein Vorbild für etwas.

Die äußeren Faktoren, die den Inhalt und die Struktur eines Modells beeinflussen, fasse ich in fachlichen Einheiten zusammen, die im weiteren als Kontexte bezeichnet werden. Ein Modell kann dabei ein beeinflussender Kontext für ein weiteres Modell sein.

Darauf aufbauend diskutiere ich dann in Kapitel 3, welche Beziehungen zwischen den Modellen und ihren Kontexten gegeben sein müssen, damit sich der strukturelle Aufbau eines Kontextes (der auch ein Modell sein kann) weitgehend bruchlos auf ein zu konstruierendes Modell übertragen läßt. Zur Beantwortung dieser Fragestellung werde ich in diesem Kapitel

Ergebnisse aus der Wissenschaftsdiziplin der Semiotik (vgl. [Eco76]) auf das in Kapitel 2 erarbeitete Prozeßmodell der objektorientierten Softwareentwicklung übertragen. Dies geschieht insbesondere unter den folgenden zwei Gesichtspunkten:

- der erstmaligen Konstruktion von Modellen innerhalb eines Softwareentwicklungsprozesses und
- der Rekonstruktion von Beziehungen zwischen bereits existierenden Modellen und den beeinflussenden Kontexten während der Analyse eines bereits konstruierten Softwaresystems.

Im Rahmen des Kapitels 3 werde ich zeigen, daß die Softwarearchitektur eines objektorientierten Anwendungssystems möglichst strukturgleich mit den Organisationsstrukturen im Anwendungsbereich gestaltet sein sollte. Da der Schwerpunkt dieser Arbeit auf Anwendungssystemen liegt, die ganze Organisationen oder Bereiche innerhalb einer Organisation softwaretechnisch unterstützen, beschäftigt sich das Kapitel 4 mit ausgewählten Aspekten der Organisationstheorie. Dabei thematisiere ich zum einen, wie sich Organisationsstrukturen auf Anwendungssoftware übertragen lassen. Zum anderen werden Möglichkeiten erörtert, den Anwendungsbereich in kleinere Teilbereiche zu zerlegen, so daß die entstandenen Teilbereiche weitgehend unabhängig voneinander analysiert, modelliert und implementiert werden können. Speziell darin unterscheidet sich mein Strukturierungsmodell konzeptionell von bestehenden objektorientierten Ansätzen, wie sie beispielsweise von Shlaer & Mellor in [SM93a] oder von Coad & Yourdon in [CY91] verfolgt werden.

Die mit diesem Ansatz erarbeitete Struktur wird anschließend auf das *fachliche Modell* übertragen. Dabei werde ich zeigen, daß sich fachlich integrierte Anwendungssysteme nur konstruieren lassen, wenn den im fachlichen Modell gebildeten Teilbereichen ein gemeinsames Teilmodell unterliegt, welches ich im weiteren als *Gegenstandsbereich* bezeichne. Um hierbei nicht mit denselben Problemen konfrontiert zu werden, die auch mit der Konstruktion eines unternehmensweiten Modells verbunden sind, diskutiere ich im weiteren Regeln und Konstruktionsprinzipien, insbesondere das Offen-Geschlossen-Prinzip (vgl. [Mey88]), die eine *optimale* Gestaltung des Gegenstandsbereichs erlauben.

Bevor die für das fachliche Modell erarbeitete Struktur auf das *softwaretechnische Modell* übertragen werden kann, erörtere ich zuerst auf der Grundlage des Kapitels 2 die Merkmale von Architekturstilen sowie die Eigenschaften von Rahmenwerken. In Anlehnung an Johnson et al. (vgl. [JF88] und [JR91]) wird der Begriff des Rahmenwerks wie folgt verwendet:

Begriff 1.2 Rahmenwerk (1. Fassung)

Ein Rahmenwerk stellt eine softwaretechnische Lösung für eine Reihe verwandter Probleme zur Verfügung.

Darauf aufbauend wird dann in Kapitel 5 ein rahmenwerkbasierter Architekturstil definiert, der die Basis für eine softwaretechnische Kombination von Komponenten zu Anwendungssystemen bildet. Dabei werde ich im Gegensatz zur gängigen Literatur (vgl. etwa [JF88] und [JR91]) herausarbeiten, daß die Rahmenwerkattribute *White-Box* und *Black-Box* keine

statische Klassifikation darstellen, sondern unterschiedliche Verwendungsformen eines Rahmenwerks bei der Konstruktion von Anwendungssystemen beschreiben.

Eine Diskussion der Merkmale von Schichtenarchitekturen, insbesondere ihre Bewertung im Zusammenhang mit dem objektorientierten Konzept der Vererbung und dem Offen-Geschlossen-Prinzip, sowie eine Evaluierung von bekannten Schichtenarchitekturen wird dann zusammen mit dem rahmenwerkbasierten Architekturstil die Grundlage für die Ausarbeitung eines objektorientierten Schichtenarchitekturstils bilden. Dieser gewährleistet einerseits die softwaretechnische Umsetzung der in Kapitel 4 erarbeiteten Struktur des fachlichen Modells, andererseits legt er auch Partitionierungsvorschriften für Klassenhierarchien und Rahmenwerke fest.

Da heutige Programmiersprachen keine adäquaten Modellierungselemente bereitstellen, um (a) Rahmenwerke über einzelne Schichten hinweg miteinander zu kombinieren und (b) den Gegenstandsbereich möglichst optimal zu gestalten, thematisiere ich in Kapitel 6 die beiden Konzepte Produkthändler und Rolle (vgl. [GSR95], [KØ96] und [Ree96]). Beide Konzepte werden dabei durch Entwurfsmuster im Stil von [GOF95] beschrieben. Den Abschluß dieses Kapitels bildet eine Konfigurationssprache, die es ermöglicht, Anwendungssysteme mit Hilfe von Produkthändlern dynamisch aus einer Menge von Rahmenwerken und Klassenbibliotheken zu konfektionieren.

Die in dieser Arbeit erzielten Ergebnisse werden in Kapitel 7 abschließend zusammengefaßt. Desweiteren zeige ich in diesem Kapitel Richtungen für zukünftige Forschungsarbeiten auf.

2 Modellbildung in der Softwareentwicklung

> "We create a myriad of mental models to help us understand phenomena of interest and to master them. A model is an artifact created for a purpose; it cannot be right or wrong, only more or less useful for its purpose." ([Ree96], S. 33)

Objektorientierte Softwareentwicklung fokussiert auf die Rekonstruktion der fachlichen Begriffe des Anwendungsbereichs. Ist mittels eines Abstimmungsprozesses, der zwischen den am Entwicklungsprozeß beteiligten Gruppen stattfindet, der Umfang des zu entwickelnden Softwaresystems festgelegt, so erstellt der Softwareentwickler ein auf den fachlichen Konzepten beruhendes Modell des Anwendungsbereichs. Im Sinne von [KGZ+94] und [RWG95] werden durch Analyse der relevanten Aufgaben und Tätigkeiten[2] des Anwendungsbereichs (vgl. auch [Jac92]) die vorhandenen und verwendeten Gegenstände und deren Umgangsformen ermittelt. An den Gegenständen wird dabei untersucht, welche Informationen an ihnen abgelesen und welche Veränderungen an ihnen vorgenommen werden können, ohne daß der Gegenstand zerstört oder in einen andersartigen Gegenstand transformiert wird (vgl. [KGZ+94]). Jeder betrachtete Gegenstand im Anwendungsbereich hat eine Identität, die ihn von anderen unterscheidbar macht. Durch den Prozeß der Abstraktion (vgl. [Ode92a]) werden Unterschiede zwischen einzelnen Gegenständen verdeckt und die Gemeinsamkeiten, die sie aufweisen, herausgearbeitet. Somit lassen sich eine Reihe von Gegenständen, die gleiche Umgangsformen besitzen, auf einen gemeinsamen Begriff bringen. Eco schreibt zum Prozeß der Begriffsbildung:

> "Begriffe […] sind die Resultate eines Abstraktionsprozesses (bei dem - das ist wichtig - nur einige relevante Elemente beibehalten wurden), der uns nicht die individuelle Essenz der benannten Dinge, sondern ihre nominale Essenz liefert." ([Eco76], S. 223)

Die so modellierten Begriffe werden in einem nächsten Schritt zueinander in Beziehung gesetzt. Die objektorientierte Modellierung stellt prinzipiell zwei Beziehungsarten zur Verfügung (vgl. [KGZ+94], [Ode92a] und [Ode92b]):

- *Generalisierung*: Gemeinsamkeiten, die über verschiedene Begriffe hinweg existieren, werden in einem allgemeineren Konzept zusammengefaßt, das sich wiederum durch einen Begriff repräsentieren läßt. Die Generalisierung wird als Oberbegriff bezeichnet. Die Begriffe, zu denen eine Generalisierung gebildet wurde, werden in Bezug auf diese Generalisierung als Unterbegriffe bezeichnet. Auf Gegenstände, die sich durch einen

[2] Die Aufgaben und Tätigkeiten werden im Sinne der Prozeßorientierung in Szenarios (vgl. Abschnitt 2.2), ihre antizipierte softwaretechnische Unterstützung in Systemvisionen (vgl. Abschnitt 2.3) beschrieben.

Unterbegriff charakterisieren lassen, trifft auch der Oberbegriff zu. Durch den Prozeß der Generalisierung (bzw. Spezialisierung) entsteht eine Begriffshierarchie.

- *Komposition*: Der Mechanismus, Gegenstände aus anderen Gegenständen zusammenzusetzen oder zu diesen in eine benutzende Beziehung zu setzen, wird als Komposition bezeichnet. Die Komposition als Beziehung zwischen Gegenständen wird bei der Bildung von Begriffen beibehalten, indem die Kompositionsbeziehung zum Bestandteil des konstruierten Begriffsmodells wird.[3]

Durch den Prozeß der Abstraktion, Generalisierung/Spezialisierung und der Komposition entsteht ein auf den fachlichen Gegenständen des Anwendungsbereichs beruhendes Begriffsmodell. Dieses Begriffsmodell läßt sich anschließend in ein Klassenmodell transformieren, das auf den Konzepten der verwendeten Programmiersprache beruht. Die Transformation des Begriffsmodells in das Klassenmodell erfolgt dabei weitgehend ohne Modellbruch, da die verwendeten Modellelemente des Begriffs- und Klassenmodells ähnlich sind (vgl. [KGZ+94], [Jac92] und [Ode92]). So entsprechen die Modellelemente Begriff, Generalisierung/Spezialisierung, Komposition und Begriffshierarchie des Begriffsmodells den Modellelementen Klasse, Vererbung, Aggregation/Assoziation und Klassenhierarchie im Klassenmodell. Gegenstände werden als Objekte, ihre Umgangsformen durch Operationen modelliert (vgl. Abb. 2.1).

Begriffsmodell	Klassenmodell
Gegenstand	Objekt
Umgangsform	Operation
Begriff	Klasse
Generalisierung, Spezialisierung	Vererbung
Komposition	Aggregation, Assoziation
Begriffshierarchie	Klassenhierarchie

Abb. 2.1: Verwendete Modellelemente des Begriffs- und Klassenmodells

Das durch die Transformation des Begriffsmodells konstruierte Klassenmodell beinhaltet allerdings nicht nur Klassen und Operationen, die bereits im Begriffsmodell modelliert worden sind, sondern auch solche, die durch die verwendete Technik motiviert sind. Diese Aussage soll beispielhaft anhand der Rekonstruktion des Begriffs „Girokonto“ genauer analysiert werden.

Die softwaretechnische Rekonstruktion eines Begriffs erfolgt mittels Analyse von Gegenständen und deren Umgangsformen im Kontext relevanter Aufgaben und Tätigkeiten. Bezogen auf das Beispiel des Girokontos muß also evaluiert werden, was bankfachlich mit einem Girokonto im Kontext des alltäglichen Kundengeschäftes getan werden kann. Mit einem Girokonto verbinden wir folgende typischen Umgangsformen: Geld einzahlen, Geld abheben, einen Dauerauftrag einrichten, den Dispo-Kredit des Girokontos in Anspruch

[3] Eine Unterscheidung zwischen Assoziation und Aggregation, wie sie technisch in vielen Programmiersprachen getroffen wird, wird fachlich nicht eingeführt.

nehmen oder Zinsen berechnen. Auf Grund der Aufzählung wäre ein Softwareentwickler mit hinreichender Bankerfahrung in der Lage, eine Klasse `GiroKonto` mit den Operationen `einzahlen`, `auszahlen`, `berechneZinsen`, usw. zu konstruieren.

Für die Verwendung der Klasse `GiroKonto` in einem Softwaresystem ist diese Schnittstelle nicht ausreichend. Heutige Anwendungssoftware wird sehr häufig als interaktive Software konstruiert. Die Benutzungsschnittstelle solcher Softwaresysteme beruht meist auf der Metapher des elektronischen Schreibtisches (engl. Desktop). Softwaregegenstände, die der Benutzer zur Erledigung seiner Arbeit benötigt, werden dabei analog zu den Gegenständen, die auf einem realen Schreibtisch anzutreffen sind, auf dem elektronischen Schreibtisch visualisiert. Das Macintosh-System der Firma Apple war das erste erfolgreiche Betriebssystem, das die Metapher des elektronischen Schreibtisches konsequent umgesetzt hat. Heute ist diese Metapher in fast allen graphikorientierten Benutzungsschnittstellen verankert.

In einem zur Unterstützung eines Bankberaters entwickelten Softwaresystem, das an der Benutzungsschnittstelle nach der Metapher des elektronischen Schreibtisches konzipiert wird, visualisiert der elektronische Schreibtisch neben den üblichen Gegenständen wie Notizblock, Uhr oder Terminkalender auch die bankfachlichen Gegenstände wie Girokonten, Formulare oder Kundenakten. Die Visualisierung eines Gegenstandes auf dem elektronischen Schreibtisch erfolgt durch eine entsprechende Ikone (oder Sinnbild). Um diese Präsentation als Ikone softwaretechnisch realisieren zu können, muß innerhalb des Softwaresystems eine Zuordnung von Objekt (Gegenstand) zu Ikone verwaltet werden. Dies erfolgt am einfachsten, indem die Klasse, deren Objekte als Ikonen auf dem Desktop visualisierbar sind, eine entsprechende Operation `gibIkone` implementiert. Die Klasse `GiroKonto` bietet also neben ihren anwendungsfachlich motivierten Operationen auch solche Operationen an, die durch die eingesetzte Technik des interaktiven graphischen Anwendungssystems und der Metapher des elektronischen Schreibtisches motiviert werden.

Aus Diskussionen mit Bankfachleuten erfährt der Softwareentwickler weiterhin, daß die Berechnung der Zinsen auf einem Girokonto nach unterschiedlichen Verfahren durchgeführt werden kann. Er entscheidet sich, das Strategie-Entwurfsmuster (engl. design pattern) für die Zinsberechnung einzusetzen. Zur Anwendbarkeit des Strategie-Entwurfsmusters formulieren Gamma et al.:

> "[...] you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms." ([GOF95], S. 316)

Bei einer Modellierung, die nach dem Strategie-Entwurfsmuster realisiert ist, werden die einzelnen Zinsberechnungsverfahren in jeweils eigenen Klassen, die Strategieklassen genannt werden, gekapselt. Für die Implementierung der Klasse `GiroKonto` ergeben sich damit folgende Alternativen. Die Operation `berechneZinsen` kann mit einem zusätzlichen Parameter versehen werden, um das gewünschte Zinsberechnungsstrategie-Objekt an die Operation zu übergeben. Alternativ kann eine Exemplarvariable in der Klasse `GiroKonto` zur Aufnahme

der Strategie vereinbart werden. Zusätzlich definieren wir eine entsprechende Operation, mit der man einem Girokonto die entsprechende Zinsberechnungsstrategie setzen kann. Neben den bereits aufgeführten Einflußgrößen Anwendungsbereich (Bankgeschäft) und eingesetzte Technik (interaktive graphische Anwendungen) beeinflußt ein dritter Kontext die Konstruktion des Begriffs- und Klassenmodells: die durch die Verwendung einer objektorientierten Entwicklungsmethode zur Verfügung gestellten Entwurfsmuster und Softwarearchitekturen. Daraus ergibt sich abschließend folgendes Ergebnis:

Ergebnis 2.1

Die Entwicklung eines Softwaresystems wird durch verschiedene Kontexte geprägt, die die Bedeutung der softwaretechnisch rekonstruierten Begriffe beeinflussen. Es lassen sich folgende Kontexte identifizieren: (1) der Anwendungsbereich, (2) die verwendete Technik sowie (3) die Entwurfsmuster und die Softwarearchitekturen, die auf Grund der eingesetzten Entwicklungsmethode verwendbar sind.

In den nachfolgenden Kapiteln ist dieses Ergebnis weiter auszuarbeiten. Insbesondere soll die Struktur der einzelnen Kontexte detaillierter untersucht werden, um zu zeigen, wie sie das Begriffs- und Klassenmodell beeinflussen. Ebenso muß der Zusammenhang, der zwischen den einzelnen Kontexten und den erstellten Modellen existiert, genauer analysiert werden.

Neben der Tatsache, daß die verschiedenen Kontexte die Konstruktion eines Softwaresystems beeinflussen, ist es wichtig, daß ein Softwareentwickler bei der Interpretation eines bestehenden Begriffs bzw. einer bestehenden Klasse die unterschiedlichen Kontexte berücksichtigt. Denn je nach Situation wird ein anderer die Begriffe bzw. Klassen prägender Kontext betrachtet. So ist es aus Sicht der Konstruktion zum Beispiel wichtig, welche Entwurfsmuster der Softwareentwickler zur Implementierung der Klasse eingesetzt hat.

Das präsentierte Beispiel des Girokontos zeigt eine weitere Eigenschaft: Die Transformation des aus dem Anwendungsbereich erstellten Begriffsmodells in das Klassenmodell kann ohne Modellbruch durchgeführt werden, da die verwendeten Modellelemente, und dadurch der strukturelle Aufbau des Begriffs- und Klassenmodells, weitgehend homolog sind. Die Strukturähnlichkeit beschränkt sich allerdings auf die Modellelemente Gegenstand - Objekt, Umgangsform - Operation, Begriff - Klasse, Generalisierung/Spezialisierung - Vererbung, Komposition - Aggregation/Assoziation und Begriffshierarchie - Klassenhierarchie. In Softwaresystemen repräsentieren diese Modellelemente Mikroelemente.

Große Softwaresysteme mit mehreren tausend Klassen sind praktisch nicht mehr weiterentwickelbar, wenn die Zusammenfassung der Mikroelemente zu größeren Strukturen konzeptionell nicht ausreichend durchdacht wird. Die Konstruktion großer Softwaresysteme ließe sich stark vereinfachen, wenn im Anwendungsbereich Makrostrukturen identifiziert werden könnten, die sich ohne Modellbruch bis ins Klassenmodell übernehmen lassen. Dann beruhen die Makrostrukturen des Softwaresystems, vergleichbar der Klassen- und Begriffsanalogie, auf den entsprechenden Makrostrukturen des Anwendungsbereichs.

Das Beispiel des Girokontos hat gezeigt, daß die erstellten Klassen außer von fachlichen auch von softwaretechnischen Aspekten geprägt werden. Eine Diskussion über Mikro- und Makrostrukturen im Anwendungsbereich, Begriffsmodell und Klassenmodell muß also auch vorhandene Mikro- und Makrostrukturen, die durch die eingesetzte Technik gegeben sind, berücksichtigen, da auch hier mit der analogen Argumentation eine weitgehende Übereinstimmung der Modellelemente gegeben sein sollte. Die von der Technik zur Verfügung gestellte Möglichkeit, eine Reihe von Klassen in Bibliotheken zusammenzufassen, sollte zum Beispiel von der Struktur zu den identifizierten fachlichen Einheiten im Anwendungsbereich passen. Diese Überlegungen führen zu der folgenden These:

> Die Entwicklung von großen Softwaresystemen könnte vereinfacht werden, wenn bei der Erstellung des Begriffs- und Klassenmodells im Anwendungsbereich existierende Makrostrukturen auf die zu erstellenden Modelle übertragbar wären. Diese fachlichen Strukturen müssen zudem homolog zu den Strukturen der verwendeten Technik sein.

Um die in Ergebnis 2.1 und der These dargelegten Aussagen belegen zu können, ist es notwendig, sich mit den folgenden Themengebieten detaillierter zu beschäftigen:

- mit einem Prozeßmodell für die objektorientierte Softwareentwicklung, auf dessen Basis erläutert wird, welche Modelle von Softwareentwicklern erstellt und verwendet werden, wie sich diese wechselseitig beeinflussen und wie das Begriffs- und Klassenmodell darin einzuordnen sind. Insbesondere ist der Einfluß vorhandener Kontexte auf das Klassenmodell zu untersuchen.
- mit der Semiotik, da sie sich als wissenschaftliche Disziplin mit der Frage der (kontextuellen) Bedeutung von Begriffen beschäftigt. Sie ist definiert als:

  > "Die Wissenschaft vom Ausdruck und der Bedeutungslehre." ([Dud90a], S. 709)

  Eco begründet zudem in [Eco76] seine Argumentation über die Korrelation von Ausdruck (Bezeichner) und Inhalt (Sinn) auf einer Theorie der Codes. Er zeigt, daß diese Korrelation um so einfacher etabliert werden kann, je strukturähnlicher die Systeme sind, die Ausdruck und Inhalt beschreiben. Wenn sich also die im Softwareentwicklungsprozeß erstellten Modelle als Systeme auffassen lassen, deren Elemente durch Regeln (Codes) miteinander verbunden sind, so ist unter diesem Gesichtspunkt auch die Disziplin der Semiotik für die weitere Arbeit von Interesse.

Ich möchte daher in den nachfolgenden Abschnitten die notwendigen Grundlagen aus dem Bereich objektorientierter Softwareentwicklung und in Kapitel 3 die der Semiotik einführen.

2.1 Ein vereinfachtes Modell des Softwareentwicklungsprozesses

Intention dieses Abschnitts ist es, ein vereinfachtes Modell des objektorientierten Softwareentwicklungsprozesses einzuführen, um daran die im letzten Abschnitt gestellte Frage nach der Bedeutung der erstellten Modelle und der beeinflussenden Kontexte diskutieren zu können.

Cook & Daniels (vgl. [CD94], S. 11), Jacobson (vgl. [Jac92], S. 15) und Odell ([Ode92], S. 45) beschreiben die Softwareentwicklung als Erstellungsprozeß einer Reihe von Modellen, die durch die Tätigkeiten Analyse und Design ineinander transformiert werden. Jacobson schreibt hierzu:

> "Thus it can be observed that system development is a gradual transformation of a sequence of models. The first model describes the customer's requirements and the last step is the fully tested program." ([Jac92], S. 14)

Die entsprechenden Modelle und die Tätigkeiten, die zur Erstellung der Modelle geführt haben, stehen laut Jacobson in folgender Beziehung:

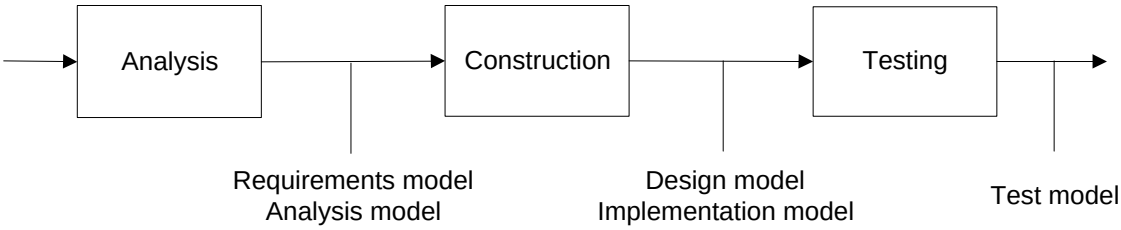


Abb. 2.2: Der Softwareentwicklungsprozeß nach Jacobson (aus [Jac92], S. 115)

Die Darstellung von Jacobson hebt die Tätigkeiten Analyse, Design sowie Test und nicht die entstandenen Modelle hervor. Um die oben dargelegten Fragen diskutieren zu können, ist es notwendig, den Fokus auf die entstandenen Modelle und die Kontexte, die die Konstruktion der Modelle beeinflussen, zu legen. Ein entsprechendes Modell des objektorientierten Softwareentwicklungsprozesses ist in Abb. 2.3 dargestellt. Die Kernaussagen, die durch das Prozeßmodell vermittelt werden, beruhen wie die Begrifflichkeit weitgehend auf der Literatur [Boo91], [CD94], [Jac92], [KGZ+94], [Ode92], [Rum91], [Som95] und [WWW93].

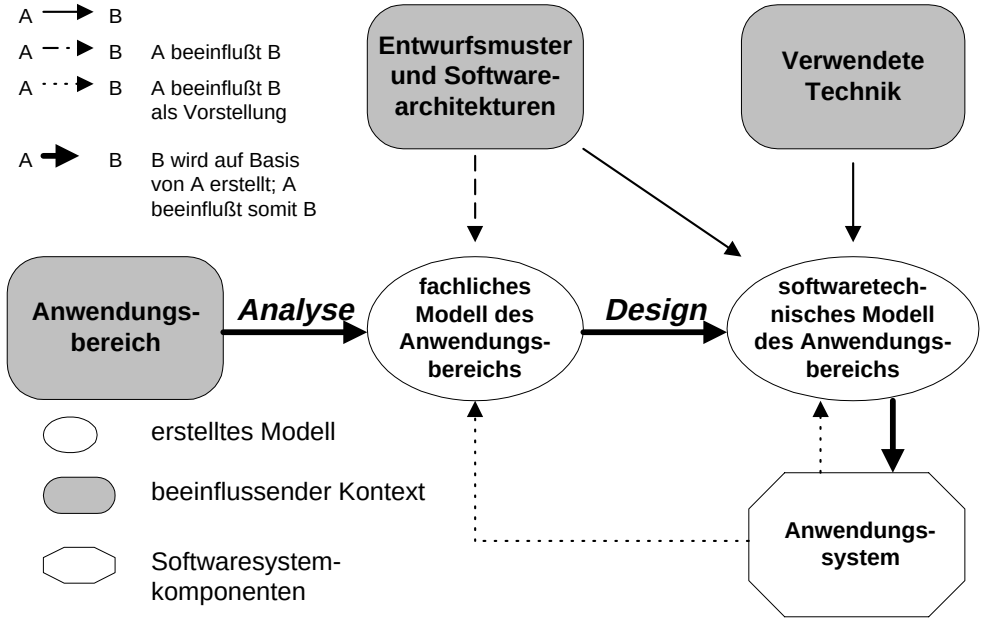


Abb. 2.3: Ein vereinfachtes Modell des Softwareentwicklungsprozesses

Da der Fokus der Diskussion auf dem konstruierten Begriffs- bzw. Klassenmodell liegt, habe ich gegenüber der gängigen Literatur eine Vereinfachung vorgenommen. Es fehlen in dem Prozeßmodell aus Abb. 2.3 die Bereiche Anforderungsermittlung, Systemdefinition, Validation, Systemeinführung und Wartung.

2.2 Der Anwendungsbereich

Bevor mit der Entwicklung eines Softwaresystems begonnen werden kann, muß der inhaltlich betroffene Anwendungsbereich festgelegt werden. Er bildet den Kontext, in dem das zukünftige System eingesetzt wird. Beispiele für Anwendungsbereiche sind das Kreditgeschäft einer Bank, die Patientenaufnahme in einem Krankenhaus oder die Verwaltung von Büchern in einer Bibliothek.

Der Anwendungsbereich bildet in zweifacher Hinsicht den Ausgangspunkt für die Erstellung eines Anwendungssystems. Um sich ein Verständnis des zu modellierenden Anwendungsbereiches zu erarbeiten, beschreiben die Entwickler in der Fachsprache der Anwendung die aktuellen Arbeitsaufgaben und Situationen, die ihnen Anwender und Benutzer berichten. Dies sind im Sinne einer Ist-Analyse (vgl. [KGZ+94] und [FKR+97]) typische Arbeitsabläufe, in denen die Gegenstände in ihrem fachlichen Verwendungszusammenhang dargestellt werden. Im Kontext der WAM-Methode (vgl. [Gry96], [KGZ+94] und [RWG95]), die zur Konstruktion des GEBOS-Systems eingesetzt wurde, werden diese Ist-Situationen in Form von Szenarios festgehalten. Der Anwendungsbereich ist weiterhin Ausgangspunkt für die Konstruktion des fachlichen Modells, das ein Softwareentwickler aufbauend auf dem Wissen, das er bei der Beschreibung der Ist-Situation von dem Anwendungsbereich gewonnen hat, erstellt (vgl. Abschnitt 2.3). In der Anfangsphase eines Projektes bezieht sich das fachliche Modell meist auf Arbeitssituationen, die in Szenarios beschrieben sind. Das nachfolgende Szenario beschreibt beispielsweise den Zusammenhang zwischen einer Sicherheit und dem Sicherungsobjekt, das als Sicherheit verwendet wird:

> Sicherheiten und Sicherungsobjekte stehen in Beziehung zueinander. Eine Sicherheit bezieht sich immer auf mindestens ein Sicherungsobjekt. So bezieht sich die Übereignung eines KFZ als Sicherheit immer auf ein Sicherungsobjekt KFZ. Sicherheiten und Sicherungsobjekte stehen in der Regel in einer n : 1 Beziehung zueinander. So kann es zu einem Haus mehrere Grundpfandrechte geben, oder ein Bürge kann mehrere Bürgschaften übernehmen.

Der Anwendungsbereich umfaßt nicht nur die Ausschnitte, die in dem fachlichen Modell abgebildet werden, sondern auch die Bereiche, die notwendig sind, um innerhalb der Ist-Analyse ein Bild von der momentanen Arbeitssituation zu gewinnen. So ist beispielsweise bei der Konstruktion eines Kreditvergabesystems auch das Wertpapiergeschäft von Bedeutung, da Wertpapiere als Sicherheiten für einen Kredit herangezogen werden können. Der Anwendungsbereich läßt sich demnach nicht auf das Kreditgeschäft reduzieren, sondern umfaßt weitere Bereiche des Bankgeschäfts, die nicht Gegenstand des zu konstruierenden Anwendungssystems sind. Den Begriff des Anwendungsbereichs verwende ich im weiteren wie folgt:

Begriff 2.1 Anwendungsbereich

Der Anwendungsbereich ist Ausgangsbasis sowohl für die Ist-Analyse als auch für die Erstellung des fachlichen Modells. Es kann sich bei ihm um eine Organisation, einen Bereich innerhalb einer Organisation oder einen Arbeitsplatz handeln. Der Anwendungsbereich ist weit genug gefaßt, um die für die Konstruktion von Modellen relevanten fachlichen Zusammenhänge während der Ist-Analyse zu verstehen. Durch einen Anwendungsbereich ist typischerweise auch eine entsprechende Anwendungsfachsprache festgelegt.

In der Literatur wird dieser Bereich als „problem domain“, „real-world“ oder „reality“ (vgl. [CD94], [Rum91] und [Ode92]) bezeichnet.

Die Größe des zu betrachtenden Anwendungsbereichs ist je nach Problemstellung sehr unterschiedlich. Dabei kann es sich um die Bücherverwaltung in einer Bibliothek, aber auch um die Gesamtheit der Geschäftsfelder eines Unternehmens (beispielsweise das gesamte Geschäftsfeld einer Bank) handeln. Die Modellierung ganzer Unternehmen wird in der Literatur als *Enterprise-Modellierung* bezeichnet. Rumbaugh versteht den Begriff denn auch wie folgt:

> "One of the major uses of object-oriented modeling is enterprise modeling, the process of understanding a complex social organization by constructing models." ([Rum93], S. 18)

Über den Erfolg, solche komplexen Anwendungsbereiche zu modellieren, schreiben Cook & Daniels allerdings:

> "You may be aware of the millions of dollars wasted on attempts to build huge software ‘maps’ (actually, corporate data models) of all aspects of a large company’s operations. Part of the problem here is the sheer scale of the undertaking; the map is out of date before it’s finished. But there’s another problem. These huge models often lack purpose. We’ve said that models are versatile but you can only take that so far; […]. If you try to solve too many problems with a single model it becomes unmanageably large and cumbersome." ([CD94], S. 5)

Obwohl Cook & Daniels bezweifeln, daß sich eine Organisation in einem umfassenden Modell abbilden läßt, existiert bei vielen Unternehmen die Notwendigkeit, unternehmensweit integrierte Softwaresysteme zu konstruieren, mit deren Hilfe Arbeitsprozesse spartenübergreifend unterstützt werden können. Im Bankenbereich besteht etwa der Anspruch, Softwaresysteme zu entwickeln, die den Bankmitarbeiter vom Schaltergeschäft bis hin zu Kundenberatungsgesprächen im Kredit-, Anlage- und Wertpapierbereich optimal unterstützen.

Ein Ergebnis dieser Arbeit wird darin bestehen zu zeigen, wie sich solche unternehmensweit integrierten Softwaresysteme konstruieren lassen, ohne solche komplexen Anwendungsbereiche in einem einheitlichen Modell beschreiben zu müssen. Auf diese Problemstellung werde ich detailliert in Kapitel 4 eingehen.

2.3 Das fachliche Modell des Anwendungsbereichs

Das fachliche Modell ist ein auf den Konzepten, Strukturen und Gegenständen des Anwendungsbereichs beruhendes Modell. In Methodenbüchern, die sich mit objektorientierter Softwareentwicklung beschäftigen, wird dieses Modell als „OO-Model of reality“, „analysis model“ oder „essential model“ bezeichnet (vgl. [CD94], [Jac92] und [Ode92]). Der Konstruktionsprozeß des fachlichen Modells wird Analyse genannt.

2.3.1 Begriffsmodell

Um einen Anwendungsbereich zu verstehen und ein sinnvolles Anwendungssystem entwerfen zu können, ist es wesentlich, die vorhandenen Arbeitszusammenhänge innerhalb des Anwendungsbereichs zu betrachten. Im Kontext der WAM-Methode werden ausgehend von den Aufgaben, die im Anwendungsbereich durch das Softwaresystem unterstützt oder automatisiert werden sollen, die entsprechenden Gegenstände und deren Umgangsformen identifiziert. Der Begriff Umgangsform, der im Bereich der Anwendungsanalyse eher ungewöhnlich ist, wird in [Gry96] wie folgt festgelegt:

Begriff 2.2 Umgangsform

"Eine Umgangsform ist eine charakteristische Handlung (an oder mit) einem Gegenstand." ([Gry96], S. 91)

Diese Interpretation deckt sich mit dem Verständnis von Verantwortlichkeiten, die Wirfs-Brock, Wilkerson und Wiener in [WWW93] geben. Verantwortlichkeiten beschreiben bei Wirfs-Brock et al. die Dienstleistungen, die eine Klasse auf Grund ihrer Verträge mit anderen Klassen erbringen muß. Sie halten fest:

"Verantwortlichkeiten bedeuten, dem Zweck eines Objektes und dessen Platz im System einen Sinn zu geben." ([WWW93], S. 63)

Ausgehend von den so identifizierten Gegenständen und deren Umgangsformen lassen sich durch den Prozeß der Abstraktion die hinter den Gegenständen stehenden relevanten Begriffe in der Benutzerfachsprache festhalten. Die so gebildeten Begriffe werden in einem nächsten Schritt in Beziehung zueinander gesetzt. Im Bereich der objektorientierten Modellierung stehen die Beziehungstypen *Generalisierung* und *Komposition* zur Verfügung. Das so entstandene Modell wird als Begriffsmodell bezeichnet. Es ist wie folgt definiert:

Begriff 2.3 Begriffsmodell

Ein Begriffsmodell ist ein auf den fachlichen Gegenständen des Anwendungsbereichs beruhendes Begriffssystem, in dem die Begriffe in Generalisierungs- und Kompositionsbeziehungen zueinander gesetzt und durch Umgangsformen beschrieben werden.

Die nachfolgende Abbildung zeigt einen Ausschnitt aus dem Begriffsmodell des Kreditvergabesystems, die Sicherheitenhierarchie. Eine Sicherheit wird von einer Bank für die Vergabe

eines Kredites verlangt, um bei einer Zahlungsunfähigkeit des Kreditnehmers offene Forderungen abdecken zu können.

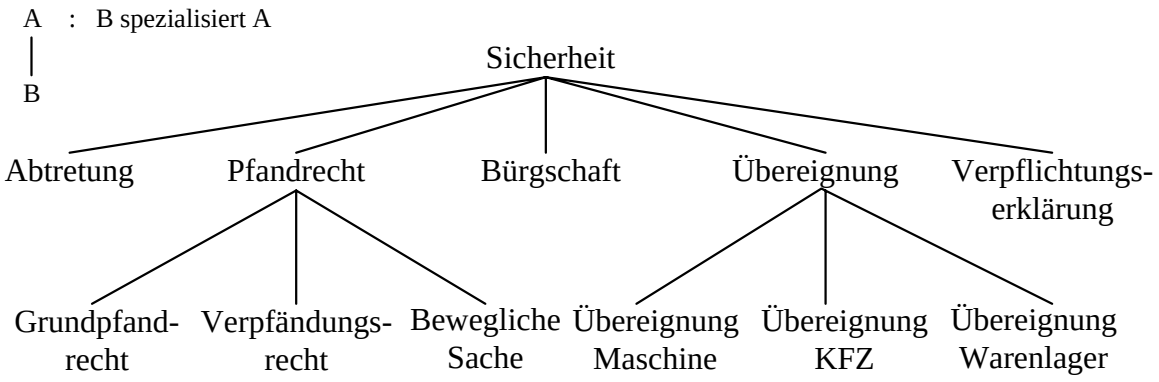


Abb. 2.4: Ein Ausschnitt aus dem Begriffsmodell der Sicherheiten

2.3.2 Leitbild und Entwurfsmetaphern[4]

Die Konstruktion von Begriffsmodellen fällt einem Softwareentwickler leichter, wenn er eine grundlegende Idee davon hat, welche Gegenstände des Anwendungsbereiches wie fachlich beschrieben werden sollen. Leitbilder (vgl. [Rol92] und [Gry96]) ermöglichen es dem Softwareentwickler, eine orientierende Vorstellung von einer Situation zu gewinnen. Diese Vorstellung bietet dann Anleitung und Gestaltungsperspektiven für den Entwurf des Anwendungssystems. Das für die Konstruktion des GEBOS-Systems verwendete Leitbild ist der Arbeitsplatz für qualifizierte menschliche Tätigkeit (vgl. [Gry96] und [BBS+94]). Das kann der Schreibtisch eines Bankmitarbeiters oder die Werkbank eines Schreiners sein.

Passend zum gewählten Leitbild werden Metaphern eingesetzt, die es ermöglichen, eine konkret faßliche Vorstellung von der Gestalt und den Anwendungsmöglichkeiten eines Softwaresystems zu gewinnen. Gryczan definiert den Begriff Entwurfsmetapher denn auch wie folgt:

> "Eine Entwurfsmetapher ist eine bildhafte Vorstellung, die ein Leitbild fachlich und konstruktiv konkretisiert." ([Gry96], S. 100)

Passend zum gewählten Leitbild wurde für das GEBOS-System die Entwurfsmetapher Werkzeug, Automat und Material gewählt, die im weiteren kurz als WAM-Metapher bezeichnet wird. Dies läßt sich fachlich motivieren: Werkzeuge, Automaten und Materialien gehören zu den grundlegenden Kategorien menschlicher Arbeit. Menschen sind gewohnt, nicht nur im handwerklich produktiven Bereich, sondern auch bei der täglichen Büroarbeit Arbeitsmittel als Werkzeuge und Automaten und Arbeitsgegenstände als Materialien zu sehen (vgl. [KGZ+94] und [Gry96]).

[4] Der nachfolgende Abschnitt beruht weitgehend auf [BB+95], an dem ich als Koautor beteiligt bin.

Im Bankgeschäft lassen sich Gegenstände recht selbstverständlich in die drei Gruppen einteilen. So ist zum Beispiel ein Formular Arbeitsgegenstand, also Material, eine Schreibmaschine und ein Kopierer Arbeitsmittel, wobei die Schreibmaschine Werkzeug-, der Kopierer Automatencharakter aufweist.

Als Materialien werden somit diejenigen Dinge betrachtet, die bearbeitet werden und schließlich in das Arbeitsergebnis eingehen. Bearbeiten bedeutet zum einen, das Material zu verändern, aber auch seinen Zustand und den Arbeitsfortschritt zu sondieren. Beispiele für Materialien an einem Bankarbeitsplatz sind Sicherheiten, Behälter, Mappen, Zinsen oder Renditen (vgl. Abb. 2.4). Diese Aufzählung enthält sowohl handgreifliche als auch ideelle Gegenstände. Bei der Modellierung von Materialien kommt es also weniger auf die physische Existenz eines Gegenstands an, der Softwareentwickler orientiert sich vielmehr daran, ob mit einem Gegenstand der Anwendung ein Begriff und ein fachlicher Umgang verbunden sind. In Anlehnung an [Gry96] verwende ich den Begriff Material wie folgt:

Begriff 2.4 Material

"Ein Material ist im Rahmen einer Aufgabenerledigung ein Arbeitsgegenstand. Materialien lassen sich in bestimmter Weise bearbeiten, d.h. verändern oder sondieren." ([Gry96], S. 103)

Werkzeuge sind diejenigen Dinge, mit denen sich Materialien interaktiv verändern und sondieren lassen. Werkzeuge vermitteln zwischen dem Benutzer eines Werkzeugs und dem zu bearbeitenden Material und sind nicht Teil des Arbeitsergebnisses. Werkzeuge werden mit der Zeit so selbstverständlich in der Handhabung, daß Benutzer sie beim Arbeiten nicht mehr bewußt wahrnehmen. Beispiele für Softwarewerkzeuge an einem Bankarbeitsplatz sind Auflister, um Behälterinhalte darzustellen und zu verändern, Formulareditoren, um damit Formulare zu bearbeiten, aber auch ein so spezielles Werkzeug wie ein Chartdarsteller (vgl. Abb. 2.5), mit dessen Hilfe Kursverläufe von Wertpapieren analysiert werden. Der Begriff Werkzeug wird wie folgt verwendet:

Begriff 2.5 Werkzeug

"Ein Werkzeug ist ein Arbeitsmittel, mit dem in einer bestimmten Arbeitssituation ein Arbeitsgegenstand, das Material, bearbeitet wird. Ein Werkzeug verändert ein Material oder sondiert seinen Zustand. Mit einem Werkzeug wird sowohl eine fachliche Funktionalität (es ist zu etwas gut) als auch eine bestimmte Art der Handhabung (es muß richtig gehandhabt werden) verbunden." ([Gry96], S. 106)

Automaten sind, ebenso wie Werkzeuge, in einem Arbeitszusammenhang Arbeitsmittel. Sie unterscheiden sich von Werkzeugen dadurch, daß sie zuvor festgelegte Resultate produzieren, ohne daß der Benutzer eines Automaten interaktiv auf die Erstellung des Resultats einwirkt. Er stellt lediglich zu Beginn der Bearbeitung den Automaten geeignet ein. Ein Beispiel für einen anwendungsfachlich motivierten Automaten in einem Banksoftwaresystem ist etwa ein Kreditprotokollautomat, der für einen Kredit automatisch das für die Genehmigung des Kredites benötigte Kreditprotokoll erstellt. Ein technisch motivierter Automat ist dagegen ein Datenbankautomat, mit dessen Hilfe Objekte in externen Speichermedien, wie relationalen

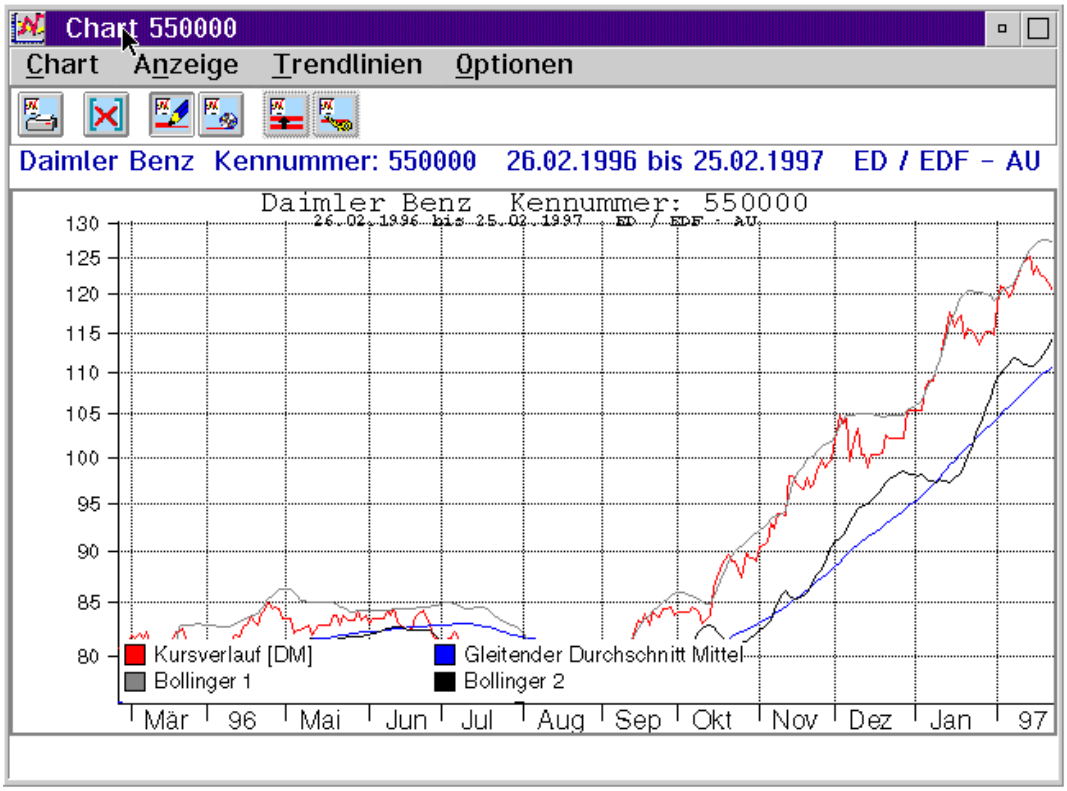


Abb. 2.5: Das Werkzeug Chartdarsteller

Datenbanken oder elektronischen Archiven, verwaltet werden. Den Begriff des Automaten verwende ich wie folgt:

Begriff 2.6 Automat

"Ein Automat ist die Vergegenständlichung einer formalisierten Routine, die über längere Zeiträume ohne äußere menschliche Eingriffe ablaufen kann. Ein Automat realisiert eine formalisierte Routine, die weitgehend ohne Kontextinformationen oder interaktive Steuerung auskommt. Einmal eingestellt, läuft die formalisierte Routine ab und produziert vorab festgelegte Resultate." ([Gry96], S. 123)

Die Entwurfsmetapher Werkzeug, Automat und Material ermöglicht es, den Zusammenhang zwischen den bisher am Arbeitsplatz vorhandenen Gegenständen und neuen Komponenten eines Anwendungssystems herzustellen.

Wird die geführte Diskussion auf die Konstruktion des Begriffsmodells übertragen, so bedeutet dies, daß bereits bei der Bildung der Begriffe eine Kategorisierung nach Werkzeugen, Automaten und Materialien erfolgt. Dies erhöht die Verständlichkeit der fachlichen Modelle.

Die Verwendung von Entwurfsmetaphern wird auch von anderen Autoren empfohlen. So schlägt Madsen in [Mad94] die Metapher „TV-Broadcasting“ vor, Taligent in [Cot95] die Entwurfsmetapher „People, Places, and Things“. Jacobson (vgl. [Jac92]) empfiehlt bei der Analyse eines Anwendungsbereichs eine Kategorisierung der Gegenstände nach „entity objects“, die die Informationen eines Systems modellieren, deren Lebensdauer hoch ist,

„interface objects“, die das Verhalten und die dargestellten Informationen der Benutzungsschnittstelle modellieren, und „control objects“, die die Funktionalität des Systems modellieren, die nicht bereits an einem „entity object“ oder einem „interface object“ angesiedelt ist. Die Autoren führen als Gründe für die Notwendigkeit einer solchen Kategorisierung sowohl die dadurch beim Softwareentwickler erzeugte bildhafte Vorstellung des Softwaresystems als auch eine bessere Konstruier- und Wartbarkeit an.

2.3.3 Systemvisionen und Glossare

Neben dem Begriffsmodell werden weitere Informationen des fachlichen Modells in Entwicklungsdokumenten, die in Prosatext in der Sprache der Anwendung verfaßt werden, festgehalten. Im Kontext der WAM-Methode geschieht dies in Systemvisionen (vgl. [Gry96] und [KGZ+94]). Gryczan schreibt zur Bedeutung von Systemvisionen:

> "Eine Systemvision beschreibt als kurzer Prosatext in der Sprache der Anwendung Arbeitssituationen, Aufgaben und Handlungsfolgen unter Verwendung von Werkzeugen und Materialien des zukünftigen Anwendungssystems. Systemvisionen können als eine Art in die Zukunft projizierte Szenarios betrachtet werden (vgl. [CR90]), d.h. sie beschreiben eine zukünftige Arbeitssituation, so wie die Entwickler sie sich vorstellen. Diese Systemvisionen gehen auf unterschiedlichen Detaillierungsebenen den jeweiligen Prototypen voraus." ([Gry96], S. 114)

Der folgende Auszug aus einer Systemvision, die aus dem Kreditvergabebereich des GEBOS-Systems stammt, soll dies verdeutlichen:

> […]. Ein elektronischer Zugriffsschutz kann im Bereich der elektronischen Kundenakte durch den Berater oder Sachbearbeiter vorgenommen werden, um die Kundenakte oder ein bestimmtes darin enthaltenes Register vor ungewollter oder unberechtigter Einsichtnahme Dritter zu schützen. Der Zugriffsschutz kann nur eingegeben werden, wenn sich die Kundenakte auf dem elektronischen Schreibtisch des Beraters/Sachbearbeiters befindet. Bei der Vergabe eines Zugriffsschutzes können bestimmte Anwender vom Zugriffsschutz ausgeschlossen werden. Dazu ist die USER-ID derjenigen Anwender anzugeben. Automatisch ausgenommen ist der Administrator, der auch jederzeit den Zugriffsschutz verändern kann. […].

In Ergänzung zu den Szenarios und Systemvisionen werden Glossare entwickelt. Während Szenarios, als Bestandteil der Ist-Analyse, und Systemvisionen, als Elemente des fachlichen Modells, dynamische Aspekte des Anwendungsbereiches bzw. des fachlichen Modells beschreiben, dokumentieren Glossare die Bedeutung und den Zusammenhang von Begriffen. Sie fokussieren somit auf die statischen Aspekte des Anwendungsbereichs. Der folgende Auszug zeigt beispielhaft den Eintrag des Begriffs »Sicherheit« in einem Glossar:

> **Sicherheit**: Eine Sicherheit wird von einer Bank für die Vergabe eines Kredits verlangt. Falls der Kredit nicht zurückgezahlt werden kann, veräußert die Bank die Sicherheit, um offene Forderungen abzudecken. Allgemein dienen Sicherheiten zur Verminderung des Kreditrisikos.

Der Begriff des fachlichen Modells läßt sich nun abschließend wie folgt definieren:

Begriff 2.7 Das fachliche Modell

Das fachliche Modell ist ein auf den Konzepten, Strukturen und Gegenständen des Anwendungsbereichs beruhendes Modell. Es fokussiert ausschließlich auf die anwendungsorientierten Begebenheiten und umfaßt das Begriffsmodell sowie weitere, in Prosatext verfaßte Entwicklungsdokumente, wie Systemvisionen oder ein Glossar. Die Gestaltungsperspektiven des fachlichen Modells werden durch entsprechende Leitbilder und Entwurfsmetaphern geprägt. Das fachliche Modell zeigt somit den Ausschnitt des Anwendungsbereiches, der durch das Anwendungssystem softwaretechnisch unterstützt werden soll.

Technische Überlegungen sind nicht Bestandteil des fachlichen Modells. So sollten die Datenbank, die gewählte Programmiersprache oder die eingesetzte Rechnerstruktur, zum Beispiel eine Client-Server Architektur, keinen Einfluß auf das Modell haben. Jacobson betont allerdings, daß das fachliche Modell bereits unter dem Gesichtspunkt konstruiert wird, daß es technisch realisierbar ist. So schreibt er in [Jac92]:

> "It will, of course, be impossible to develop the models entirely without consideration of their realization; the models and their architecture must be built so that everything in them can be realized." ([Jac92], S. 149)

Ein Beispiel für die im Zitat angesprochene notwendige Realisierbarkeit stellt die Entwurfsmetapher Werkzeug, Automat und Material dar. Eine Kategorisierung der Begriffe des fachlichen Modells in diese drei Typen ergibt nur dann einen Sinn, wenn beispielsweise die Realisierung eines Werkzeugs als interaktive graphische Komponente im softwaretechnischen Modell gegeben ist. Diese Tatsache ist in der Abb. 2.3 durch den gestrichelten Pfeil vom Kontext Softwarearchitektur und Entwurfsmuster zum fachlichen Modell dargestellt. Die vorausgegangenen Überlegungen sollen in einem Ergebnis festgehalten werden:

Ergebnis 2.2

Das erstellte fachliche Modell und dessen interne Struktur sollten rein fachlich motiviert sein. Das Modell ist allerdings mit Blick auf seine potentielle Realisierbarkeit zu erstellen.

Da das konstruierte fachliche Modell die konzeptuellen Eigenschaften des Anwendungsbereichs widerspiegeln soll, sind auch die für die Beschreibung des fachlichen Modells verwendeten Modellelemente Begriff, Umgangsform, Generalisierung, Komposition und Begriffshierarchie sowie die gewählten Bezeichner der rekonstruierten Begriffe und deren Umgangsformen von ihrem Bezeichner her an die Fachsprache des Anwendungsbereichs angelehnt. So existiert sowohl im fachlichen Modell eines zu konstruierenden Softwaresystems der Begriff Girokonto als auch in der Fachsprache des Anwendungsbereichs. Da in dieser Arbeit die Transformation von Modellen eine wichtige Rolle spielt, ist es notwendig, daß der Leser weiß, in welchem Kontext ein Bezeichner verwendet wird. Geht nicht aus dem umgebenden Text

eindeutig hervor, ob ein Bezeichner sich auf den Anwendungsbereich oder das fachliche Modell bezieht, so wird der Bezeichner wie nachfolgend beschrieben annotiert:

- *tiefgestelltes A*: der Bezeichner bezieht sich auf den Anwendungsbereich. Beispiel: Girokonto_A.
- *tiefgestelltes F*: der Bezeichner bezieht sich auf das fachliche Modell. Beispiel: Girokonto_F.

2.4 Das softwaretechnische Modell des Anwendungsbereichs und das Anwendungssystem

Das durch die Analyse entstandene fachliche Modell wird in ein softwaretechnisches Modell überführt, das um die für die Konstruktion des Systems notwendigen technischen Charakteristika, die zum Beispiel durch die eingesetzte Programmiersprache oder die verwendete Datenbank bestimmt werden (vgl. Kapitel 2.5), ergänzt ist. Das softwaretechnische Modell repräsentiert somit die softwaretechnische Vergegenständlichung des fachlichen Modells. In der Literatur wird das softwaretechnische Modell als „model of design and implementation“ oder „model of software“ bezeichnet (vgl. [Jac92] und [Ode92]).

Beschrieben wird das softwaretechnische Modell auf der Basis der Modellelemente, die das Objektmodell der gewählten Programmiersprache zur Verfügung stellt. Das Objektmodell der Sprache C++ bietet beispielsweise die folgenden Modellelemente an: Klasse, Objekt, Exemplarvariable, Operation, Klassenvariable, Klassenoperation, Mehrfachvererbung, Aggregation und Assoziation. Die Sprache Smalltalk hingegen verfügt nur über Einfachvererbung, besitzt keine Aggregation, außer bei der Verwendung der Klassen Integer, Character und Symbol. Sie weist allerdings zusätzlich das Modellelement Exemplarvariable einer Klasse auf.

Konkretisiert wird das softwaretechnische Modell durch die folgenden zwei Komponenten (vgl. [Jac92]):

1. das *Designmodell*, in dem mittels geeigneter Diagrammarten, wie zum Beispiel Klassen-, Objekt-, Zustandsübergangs- oder Interaktionsdiagramme (vgl. [Gam92] und [Jac92]), die Zusammenhänge zwischen den einzelnen Klassen sowie ihre fachlichen und technischen Eigenschaften repräsentiert werden. Es ist nicht notwendigerweise für alle Entitäten des fachlichen Modells ein Designmodell zu erstellen. Läßt sich die Struktur der fachlichen Modelle direkt in Programmkode übersetzen, so wird in der Praxis häufig auf die Konstruktion von Designmodellen verzichtet.
2. das *Implementierungsmodell*, das die Umsetzung der fachlichen Modelle und der Designmodelle in Programmkode darstellt. Dieses wird durch geeignete Werkzeuge (Compiler und Linker) in ausführbare Anwendungssoftware transformiert.

Die Abb. 2.6 präsentiert das Klassenmodell der Sicherheitenhierarchie. Es beruht auf dem Begriffsmodell der Abb. 2.4. Das Klassenmodell zeigt dabei eine hohe strukturelle Übereinstimmung mit dem Begriffsmodell. Lediglich der Begriff Sicherheit_F ist durch zwei Klassen, die abstrakte Schnittstellenklasse `Sicherheit` und die Implementationsklasse `SicherheitImpl`, realisiert worden. Die öffentliche Schnittstelle ist in einer eigenen

Schnittstellenklasse festgelegt, da die in GEBOS eingesetzte Programmiersprache C++ die Definition einer separaten öffentlichen Exportschnittstelle für eine Klasse nicht unterstützt (vgl. [CE95] und [Lak96]). In C++ werden sowohl die öffentlichen als auch die privaten Schnittstellen in derselben Datei, Header-Datei genannt, beschrieben. Dies hat zur Folge, daß Änderungen an privaten Methoden einer Klasse A auch zur Recompilation von Klassen führen, die diese Klasse A nur über ihre öffentliche Schnittstelle benutzen. Ursache hierfür ist die Abhängigkeit, die zwischen den benutzenden Klassen und der allgemeinen Header-Datei der benutzten Klasse A besteht. Um diese unnötigen Abhängigkeiten zu umgehen, wird in C++ die öffentliche Schnittstelle einer wichtigen Abstraktion in eine eigene Schnittstellenklasse ausgelagert. Das Designmodell der Sicherheitenhierarchie ist demnach durch die technischen Gegebenheiten der eingesetzten Programmiersprache geprägt.

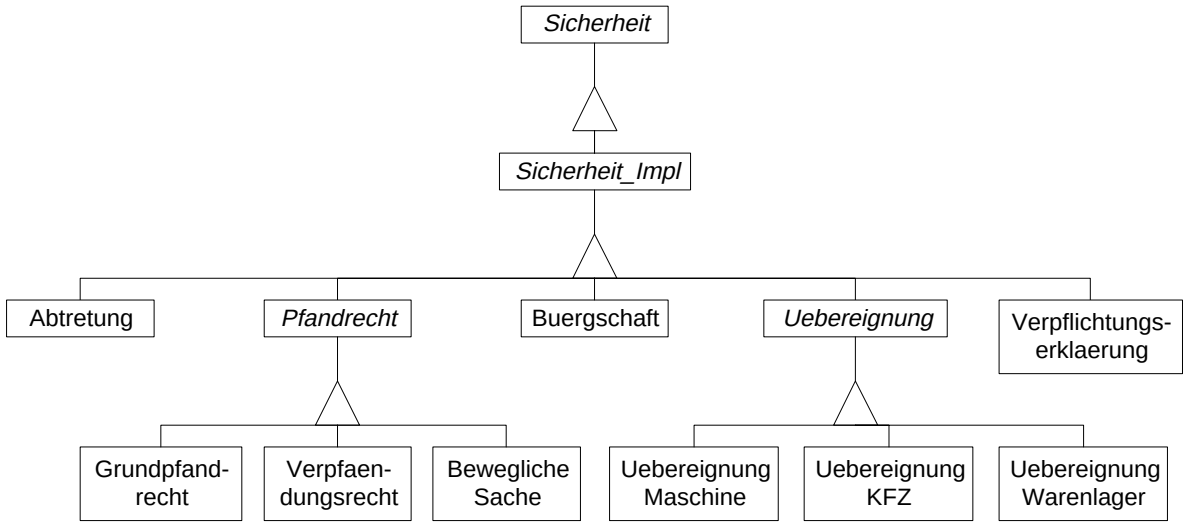


Abb. 2.6: Ein Ausschnitt aus dem Klassenmodell der Sicherheitenhierarchie

Innerhalb von Projekten, die als Ziel die Konstruktion von unternehmensweit integrierten Anwendungssystemen haben, besteht die Notwendigkeit, Teile des softwaretechnischen Modells in Form von wiederverwendbaren Komponenten anderen Teilprojekten zur Verfügung zu stellen. Wird das Anwendungssystem unter Verwendung von objektorientierten Konzepten konstruiert, so stellen Klassenbibliotheken und Rahmenwerke die wiederverwendbaren Komponenten des softwaretechnischen Modells dar. Die Struktur, nach der das softwaretechnische Modell in wiederverwendbare Komponenten eingeteilt wird, sollte auf Makrostrukturen des Anwendungsbereichs beruhen. Klassenbibliotheken und Rahmenwerke sind dabei im Sinne von konzeptionellen Einheiten zu sehen. Eine Umsetzung dieser Einheiten in strukturgleiche vorübersetzte Einheiten (d.h. statische oder dynamische Bibliotheken) wird dadurch nicht impliziert. Eine detaillierte Diskussion über die Begriffe Klassenbibliothek und Rahmenwerk werde ich in Abschnitt 5.2 führen. Abschnitt 5.5 präsentiert dann die Umsetzung von Klassenbibliotheken und Rahmenwerken als statische und dynamische Bibliotheken.

Begriff 2.8 Das softwaretechnische Modell

Das softwaretechnische Modell nimmt die Einbettung des fachlichen Modells in eine technische Umgebung vor. Es wird durch das Design- und Implementierungsmodell konkretisiert. Wiederverwendbare Komponenten des softwaretechnischen Modells werden in Form von Klassenbibliotheken oder Rahmenwerken zur Verfügung gestellt.

Ausgehend vom softwaretechnischen Modell und eventuell zur Verfügung stehenden Bibliotheken und Rahmenwerken wird mit Hilfe von Werkzeugen (Compiler, Linker, etc.) ein ablauffähiges Anwendungssystem erzeugt. Dieses vergegenständlicht das fachliche und softwaretechnische Modell.

Die Vorstellung, die ein Softwareentwickler vom Anwendungssystem als Endprodukt des Softwareentwicklungsprozesses hat, beeinflußt bereits zu Beginn der Konstruktion das fachliche und softwaretechnische Modell. Denn so wie ein Autobauer bereits das fertige Auto im Kopf hat, wenn er ein Modell des zu bauenden Autos erstellt, hat der Softwareentwickler eine Vorstellung des zu konstruierenden Anwendungssystems, wenn er beginnt, das fachliche Modell für einen Anwendungsbereich zu entwerfen. Diese Tatsache ist in Abb. 2.3 durch jeweils eine gepunktete Linie vom Anwendungssystem zum fachlichen und softwaretechnischen Modell dargestellt. Abschließend ergibt sich folgende Begriffsdefinition:

Begriff 2.9 Anwendungssystem

Das Anwendungssystem als Endprodukt des Softwareentwicklungsprozesses vergegenständlicht das fachliche und softwaretechnische Modell. Es wird mittels entsprechender Werkzeuge aus dem softwaretechnischen Modell erstellt. Es beeinflußt als Vorstellung, die ein Entwickler über diesen bestimmten Anwendungstyp hat, bereits zu Beginn des Entwicklungsprozesses das fachliche und softwaretechnische Modell.

2.5 Die verwendete Technik

Das aus dem Anwendungsbereich abgeleitete fachliche Modell sollte weitgehend ohne Berücksichtigung technischer Faktoren erstellt werden. Diese beeinflussen erst die Konstruktion des softwaretechnischen Modells. So führt zum Beispiel die Verwendung der Programmiersprache C++ dazu, daß im softwaretechnischen Modell die Freigabe von nicht mehr benötigten Objekten zu berücksichtigen ist. Wird eine Programmiersprache eingesetzt, die über einen Garbage Collector verfügt, so ist dies nicht notwendig. Soll das Softwaresystem weiterhin an eine relationale Datenbank angeschlossen werden, so sind in jeder Klasse, deren Objekte persistent in der Datenbank gespeichert werden, entsprechende Operationen vorzusehen. Diese transformieren Objekte dieser Klasse in eine passende relationale Darstellung.

Technik[5] (vgl. [Bro86], S. 680)[6], als die Menge der nutzenorientierten, künstlichen Gebilde (Artefakte oder technische Sachsysteme), kommt bei der Entwicklung eines Anwendungssystems unterschiedlich zum Einsatz. Im Rahmen meiner Arbeit sollen grundsätzlich drei Arten unterscheiden werden:

1. Technik, die unabhängig von den gestellten fachlichen Anforderungen zur Entwicklung eines Anwendungssystems benötigt wird, bezeichne ich als *benötigte Technik*. Beispiele hierfür sind die Programmiersprache mit ihrem Objektmodell, Werkzeuge zur Montage von statischen und dynamischen Bibliotheken sowie Werkzeuge, die ausführbare Anwendungssysteme erzeugen. Die benötigte Technik prägt somit die Struktur des softwaretechnischen Modells. Ein Beispiel für diese Form der Prägung zeigt die vorangegangene Diskussion über Programmiersprachen mit und ohne Garbage Collector.
2. Technik, die eingesetzt wird, um das Anwendungssystem so zu gestalten, daß es den gestellten fachlichen Anforderungen bestmöglich genügt, wird als *eingesetzte Technik* bezeichnet. So führt beispielsweise die fachliche Anforderung, daß alle Bankmitarbeiter ein gemeinsames Archiv benutzen, in dem Kundenakten elektronisch gespeichert werden, zum Einsatz einer Datenbank und entsprechenden Client- und Serverrechnern. Zum Bereich der eingesetzten Technik zählen beispielsweise auch Gestaltungsrichtlinien für eine graphische Benutzungsoberfläche.
3. Die Konstruktion eines Anwendungssystems wird mittels *unterstützender Technik* erleichtert. Diese prägt aber weder das fachliche noch das softwaretechnische Modell. Beispiele hierfür sind Textverarbeitungs- oder Emailsysteme zur Kommunikation im Entwicklungsprozeß.

Eingesetzte und benötigte Technik wirken sich, im Gegensatz zur unterstützenden Technik, auf den Inhalt des fachlichen und softwaretechnischen Modells aus. Sie werden unter dem Begriff *verwendete Technik* zusammengefaßt. Der Bereich der unterstützenden Technik wird im weiteren nicht mehr betrachtet.

Für die Konstruktion eines softwaretechnischen Modells ist es erforderlich, daß der Softwareentwickler über ein Modell von der verwendeten Technik verfügt. Dieses Modell repräsentiert sein subjektives Wissen darüber, wie die Technik zu verwenden ist. Wird in einem Projekt beispielsweise eine relationale Datenbank benötigt, so entwirft der Softwareentwickler die objekttransformierenden Operationen auf der Basis seines mehr oder minder ausgereiften Wissens über das Relationenmodell. Dieses kann allgemeingültig sein oder spezielle Eigenschaften der eingesetzten Datenbank berücksichtigen. Zur Implementierung der entsprechenden Operationen verwendet der Softwareentwickler Bibliotheken, die ihm Schnittstellenoperationen zum Datenbanksystem anbieten, oder Embedded SQL-Anweisungen. Die

[5] Im Bereich der Informatik müßte man genau genommen von Informationstechnik sprechen. Da in dieser Arbeit nur technische Faktoren von Interesse sind, die in Bezug zur Informatik stehen, verwende ich den Begriff Technik ohne das Präfix Information.

[6] Im Brockhaus wird diese Definition noch um das Adjektiv materiell ergänzt. Softwaresysteme, die allerdings keinen materiellen Charakter haben, sind aber auch als Technik zu betrachten.

Verwendung einer Technik bei der Konstruktion des softwaretechnischen Modells geschieht demnach auf den folgenden zwei Ebenen:

- Bei der Entwicklung des Designmodells verwendet der Softwareentwickler seine verschiedenen Technik-Modelle. Technik, im Sinne von verwendeter Technik, wird bei dieser Aktivität also nicht direkt eingesetzt. Die Technik-Modelle beeinflussen unmittelbar die Struktur des softwaretechnischen Modells (im Beispiel der Datenbank hat die Verwendung der Technik zu zusätzlichen Operationen an den Klassen geführt).
- Bei der Erstellung des Implementierungsmodells wird Technik direkt über Schnittstellen-Bibliotheken oder Rahmenwerke verwendet. Die Menge aller Schnittstellen zur Technik wird als *Systembasis* bezeichnet. Darunter fallen auch Schnittstellen zum Betriebssystem. Um ein Softwaresystem in unterschiedlichen Hardware- und Softwareumgebungen einsetzen zu können, wird die Systembasis in der Regel durch eine Portabilitätsschicht gekapselt (vgl. [Gam92]). Der Begriff der Systembasis soll explizit in einer Begriffsdefinition festgehalten werden:

  Begriff 2.10 Systembasis

  Die Menge aller Schnittstellen zur verwendeten Technik wird als Systembasis bezeichnet.

In der Regel eignen sich Softwareentwickler die verschiedenen Technik-Modelle im Rahmen ihrer Aus- und Weiterbildung an. Technik-Modelle fließen bei der Erstellung von Designmodellen meist nur implizit ein, da Softwareentwickler sie häufig unbewußt verwenden.

Zum Bereich Technik zählt natürlich auch der in einem Projekt etablierte Entwicklungsprozeß. Dieser beeinflußt bereits die grundsätzliche Herangehensweise innerhalb eines Projektes. So würden in einem auf struktureller Analyse beruhenden Entwicklungsprozeß andere Modelle erstellt werden, und es würden andere Kontexte existieren, die diese Modelle beeinflussen, als bei einer objektorientierten Herangehensweise. Da dieser Arbeit durchgängig die Annahme zugrunde liegt, daß für die Entwicklung eines Anwendungssystems eine objektorientierte Methode verwendet wird, soll der vorhandene Einfluß des Prozesses auf die zu erstellenden Modelle und die beeinflussenden Kontexte nicht betrachtet werden.

2.6 Entwurfsmuster und Softwarearchitekturen

Die Bedeutung von Entwurfsmustern und Softwarearchitekturen für die objektorientierte Anwendungsentwicklung wird von vielen Autoren betont (vgl. [GOF95] und [SG96]). Zu Entwurfsmustern (engl. design pattern) schreibt Gamma:

> "Ein Ansatz für die Unterstützung des Entwurfs mit Design-Mustern drängt sich bei einem objektorientierten Vorgehen deshalb auf, weil die Ausdrucksmittel objektorientierter Programmiersprachen Möglichkeiten anbieten, Design-Strukturen als entsprechende Klassenstrukturen festzuhalten. […]. Design-Muster ermöglichen die Wiederverwendung existierender Design-Strukturen und Design-Erfahrungen. Sie sind deshalb neben Frameworks eine weitere Möglichkeit zur Wiederverwendung von Design."

Bezug nehmend auf die Definition von Gamma et al. verwende ich den Begriff Entwurfsmuster im weiteren wie folgt:

Begriff 2.11 Entwurfsmuster

Ein Entwurfsmuster ist eine Beschreibung, die die Zusammenarbeit von Klassen und Objekten im softwaretechnischen Modell wiedergibt. Es ist benannt und soll ein bestimmtes, kontextbezogenes Entwurfsproblem lösen. Als Bausteine bei der Beschreibung eines Entwurfsmusters stehen die Modellelemente des Objektmodells der verwendeten Programmiersprache zur Verfügung.

Entwurfsmuster als Lösungen bestimmter Entwurfsprobleme sind somit ein Mittel, um Erfahrungen innerhalb der objektorientierten Softwareentwicklung festzuhalten und anderen Softwareentwicklern zu kommunizieren.

In Abschnitt 2.3 wurde bereits darauf hingewiesen, daß Entwurfsmetaphern sowohl eine fachliche als auch eine softwaretechnische Interpretation besitzen. Aus diesem Grund muß der Entwurfsmetapher Werkzeug, Automat und Material, die eine grundlegende Kategorisierung des fachlichen Begriffsmodells vorgibt, eine Realisierung im softwaretechnischen Modell gegenüberstehen. Die Konstruktion wird durch passende Entwurfsmuster beschrieben. Für die Konstruktion eines Werkzeugs wird im WAM-Kontext beispielsweise das Entwurfsmuster „separation of powers“ (vgl. [RZ95]) eingesetzt.

Shaw & Garlan betonen, daß der Entwurf und die Spezifikation einer Gesamtarchitektur für das Gelingen eines Software-Projektes eine wesentlich wichtigere Aufgabe darstellt, als die Wahl der richtigen Datenstrukturen und Algorithmen. Diese Feststellung ist vor allem vor dem Hintergrund zu sehen, daß die Größe und Komplexität der zu entwickelnden Softwaresysteme ständig zunimmt. Zudem sind die Autoren der Meinung, daß auch die Pflege eines Softwaresystems durch eine geeignete und stabile Softwarearchitektur positiv beeinflußt wird. Bei der Pflege eines Softwaresystems wird sehr viel Zeit darauf verwendet, bestehenden Kode, insbesondere den Zusammenhang zwischen komplexeren Komponenten, zu verstehen. Dieser Aufwand kann durch eine das System beschreibende Softwarearchitektur drastisch verringert werden. Zum Begriff Softwarearchitektur schreiben Shaw & Garlan:

> "Software architecture is emerging as an important discipline for engineers of software. But it did not simply appear, well-formed and clearly articulated, as a new concern for software systems. Rather, it has emerged over time as a natural evolution of design abstractions, as engineers have searched for better ways to understand their software and new ways to build larger, more complex software systems. […]. Abstractly, *software architecture* involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among those components. Such a system may in turn be used as a (composite) element in a larger system design." ([SG96], S. 1)

Der Forschungszweig Softwarearchitektur wird in der Informatik derzeit sehr weit gefaßt. Als Ergebnis zweier Workshops über Softwarearchitekturen (vgl. [SG96], S. 15) haben sich die folgenden vier Forschungsrichtungen herauskristallisiert:

- *architectural description languages*: Dieser Bereich beschäftigt sich mit Sprachen und Notationen, die zur Beschreibung von Softwarearchitekturen verwendet werden. Wie bei den Entwurfsmustern soll auch hier durch eine einheitliche Notation die Verständlichkeit und Kommunizierbarkeit von Softwarearchitekturen erhöht werden.
- *codification of architectural expertise*: Die Arbeiten in dieser Kategorie beschäftigen sich mit der Dokumentation und Katalogisierung bekannter Softwarearchitekturen.
- *frameworks for specific domains*: Rahmenwerke gewinnen zunehmend an Bedeutung, wenn es darum geht, Softwarearchitekturen in Form von Komponenten zur Verfügung zu stellen. Dieser Bereich beschäftigt sich daher mit der Architektur von Rahmenwerken, die für spezielle Anwendungsbereiche, wie etwa Robotersteuerungen oder das Bankgeschäft, entwickelt werden.
- *formal underpinnings for architecture*: Diese Kategorie hat zum Ziel, die formalen Grundlagen einer Softwarearchitektur zu erarbeiten. Dazu zählen beispielsweise formale Sprachen, mit denen sich das Verhalten von Architekturkonzepten beschreiben läßt.

In Bezug auf die von Shaw & Garlan eingeführte Kategorisierung von Softwarearchitekturen ist meine Arbeit den Bereichen „codification of architectural expertise“ und „frameworks for specific domains“ zuzuordnen.

In den nachfolgenden Abschnitten soll eine Definition des Begriffs Softwarearchitektur erarbeitet werden. Die Begriffsdefinition wird durch zwei Aspekte geprägt: (a) Softwarearchitekturen werden vor dem Hintergrund der objektorientierten Softwareentwicklung betrachtet (b) es wird untersucht, wie Softwarearchitekturen als Kontext das softwaretechnische Modell beeinflussen (vgl. Abb. 2.3).

Softwarearchitekturen beschreiben das grundlegende Muster, nach denen Softwaresysteme strukturiert und organisiert sind. Shaw & Garlan halten dazu fest:

> "One of the hallmarks of architectural design is the use of idiomatic patterns of system organization." ([SG96], S. 19)

Shaw & Garlan haben in [SG96] für die Beschreibung einer Softwarearchitektur die Modellelemente Komponente (engl. component) und Verbindungsstück (engl. connector) definiert. Mit ihnen soll der strukturelle Aufbau eines Softwaresystems in Form von benannten Mustern dargestellt werden. Die Autoren geben folgende Beispiele für Softwarearchitekturen an: Client-Server, Pipes und Filters oder eine Schichtenarchitektur. Im weiteren sollen die Eigenschaften von Komponenten und Verbindungsstücken in Anlehnung an Shaw & Garlan beschrieben und im Kontext dieser Arbeit interpretiert werden.

Die Eigenschaften einer *Komponente* lassen sich wie folgt charakterisieren:

- Eine Komponente repräsentiert eine gekapselte (wieder)verwendbare Einheit eines Softwaresystems. Ihre Dienste stellt sie über eine Schnittstelle zur Verfügung. Eine Komponente beschreibt keine Implementierungsdetails, wie etwa Algorithmen oder den

inneren Aufbau einer Klasse. Im objektorientierten Paradigma sind Komponenten nicht mit Klassen gleichzusetzen, die wesentlich feingranularer sind. Komponenten werden daher immer durch Strukturen kooperierender Klassen realisiert. Solche Klassenstrukturen lassen sich jedoch im Sinne des Facade-Musters (vgl. [GOF95], S. 185) über eine ausgezeichnete Klasse verwenden. Diese dient dann als Schnittstelle zur Klassenstruktur. Komponenten stellen gekapselte (wieder)verwendbare Einheiten dar, die über eine bekannte Schnittstelle verwendet werden. In diesem Sinne lassen sich Rahmenwerke als Komponenten auffassen.

- Eine Komponente erfüllt einen spezifischen softwaretechnischen Zweck. So stellt beispielsweise ein Rahmenwerk ein wiederverwendbares Design als abstrakte Lösung für eine Familie verwandter Problemstellungen bereit. Ein Filter soll als Komponente einer Pipes und Filters Architektur einen Strom von Eingabedaten lesen, bearbeiten und als Strom von Ausgabedaten schreiben.
- Anwendungssysteme sollten gemäß einer Softwarearchitektur realisiert werden. Innerhalb eines solchen Anwendungssystems haben Komponenten neben ihrer technischen auch immer eine spezifische fachliche Interpretation. Jede Komponente ist für die Erbringung eines fachlichen Services zuständig. So wird mit dem Konto-Rahmenwerk das fachliche Anliegen verfolgt, Umgangsformen mit Konten im Bankgeschäft bereitzustellen. Der sort-Filter des Unix-Systems wird fachlich dann eingesetzt, wenn ein Eingabestrom sortiert in einen Ausgabestrom transformiert werden soll. Eine Komponente des softwaretechnischen Modells muß daher einen Bezug zu fachlichen Einheiten des Anwendungsbereichs aufweisen.
- Eine Komponente beschreibt keine konkreten, auf ein spezielles Anwendungssystem bezogenen Charakteristika, wie etwa die Beziehung zwischen zwei Elementen, noch irgendwelche Implementierungsdetails.
- Komponenten sind benannt.

Den Begriff Komponente[7] definiere ich bezogen auf objektorientierte Softwarearchitekturen wie folgt:

Begriff 2.12 Komponente

Eine Komponente ist eine benannte, (wieder)verwendbare Einheit eines Softwaresystems. Sie stellt ihre Dienste über eine Schnittstelle zur Verfügung. Eine Komponente ist grobkörniger als eine Klasse und muß daher durch eine Struktur kooperierender Klassen realisiert werden. Implementierte Komponenten erfüllen nicht nur einen bestimmten softwaretechnischen Zweck, sondern reflektieren auch

[7] Nierstrasz und Dami definieren den Begriff der Komponente im Kontext komponentenorientierter Softwareentwicklung wie folgt: "In short, we say that a component is a ‘static abstraction with plugs’. By ‘static’, we mean that a software component is a long-lived entity that can be stored in a software base, independently of the applications in which it has been used. By ‘abstraction’, we mean that a component puts a more or less opaque boundary around the software it encapsulates. ‘With plugs’ means that there are well-defined ways to interact and communicate with the component (parameters, ports, messages, etc.)." ([ND95], S. 5). Diese Definition zeigt eine inhaltliche Übereinstimmung mit der Definition von Komponente und Verbindungsstück im Kontext von Softwarearchitekturen.

immer eine fachliche Einheit des Anwendungsbereichs. Im Kontext von Softwarearchitekturen stellt ein Rahmenwerk ein typisches Beispiel für eine objektorientierte Komponente dar.

Die Charakteristika eines *Verbindungsstücks*, als weiteres Modellelement von Softwarearchitekturen, lassen sich wie folgt beschreiben:

- Verbindungsstücke beschreiben, wie einzelne Komponenten einer Softwarearchitektur auf Schnittstellenebene untereinander verknüpft werden und wie sie miteinander interagieren. Beispiele für Verbindungsstücke sind: ein Prozeduraufruf, ein Operationsaufruf an einem Objekt, das Versenden von Ereignissen oder Pipes. In der objektorientierten Softwareentwicklung werden zur Kopplung von Rahmenwerken neben einfachen Operationsaufrufen auch komplexere Verbindungsstücke eingesetzt. Shaw et al. schreiben hierzu:

  > "Connectors do not in general correspond individually to compilation units; they manifest themselves as table entries, buffers, instructions to a linker, dynamic data structures, sequences of system calls embedded in code, initialization parameters, servers that support multiple independent connections, and the like." ([SDK+95], S. 317)

  Komplexe Verbindungsstücke zur Kopplung von Rahmenwerken setzen ein fundiertes Verständnis über die Funktionsweise der zu koppelnden Rahmenwerke voraus. Entwurfsmuster beschreiben das Zusammenspiel von Klassen und Objekten bei der Lösung eines kontextbezogenen Problems. Unter diesem Gesichtspunkt können sie als Gebrauchsanweisung für die Verwendung eines Rahmenwerks dienen (vgl. Begriff 1.2). Johnson schreibt zur Rolle eines Entwurfsmuster im Kontext von Rahmenwerken:

  > "The main purpose of a set of patterns is to show how to use a framework, not to show how it works, but patterns can also describe a great deal of the theory of its design." ([Joh92], S. 63)

  Verbindungsstücke legen fest, wie eine Komponente mit einer anderen Komponente interagiert, wie sie sie verwendet. Komplexe Verbindungsstücke lassen sich daher in Form eines Entwurfsmusters beschreiben. Konkret erfolgt die Kopplung der verschiedenen Komponenten über ausgewählte Klassen der zu koppelnden Komponenten, deren Verantwortlichkeiten und Zusammenarbeit durch das Entwurfsmuster festgelegt werden. Es definiert somit das Protokoll des Verbindungsstücks. Beispielsweise könnte neben einem Konto-Rahmenwerk ein davon losgelöstes Verzinser-Rahmenwerk entwickelt werden, das die bankfachlichen Zinsberechnungsstrategien zur Verfügung stellt (vgl. Abb. 2.7). Die Kopplung der beiden Rahmenwerke ließe sich dann durch das Strategie-Muster beschreiben.

- Ein Verbindungsstück ist, ebenso wie eine Komponente, benannt.

Der Begriff Verbindungsstück wird bezogen auf objektorientierte Softwarearchitekturen wie folgt definiert:

Begriff 2.13 Verbindungsstück

Ein Verbindungsstück beschreibt, wie zwei oder mehr Komponenten miteinander gekoppelt werden und wie sie interagieren. Verbindungsstücke erfüllen somit primär einen softwaretechnischen Zweck. Im Kontext objektorientierter Softwareentwicklung lassen sich Entwurfsmuster zur Beschreibung komplexerer Verbindungsstücke verwenden.

Unter Verwendung der definierten Begriffe Komponente und Verbindungsstück wird der Begriff Softwarearchitektur in Anlehnung an [SG96] wie folgt gefaßt:

Begriff 2.14 Softwarearchitektur

Eine Softwarearchitektur organisiert und strukturiert das softwaretechnische Modell als eine Menge interagierender Komponenten. Verbindungsstücke beschreiben dabei die Kopplung und das Zusammenspiel dieser Komponenten untereinander.

Die Abb. 2.7 zeigt die Komponenten Konto- und Verzinser-Rahmenwerk sowie das Verbindungsstück Strategie-Muster (vgl. [GOF95], S. 316).

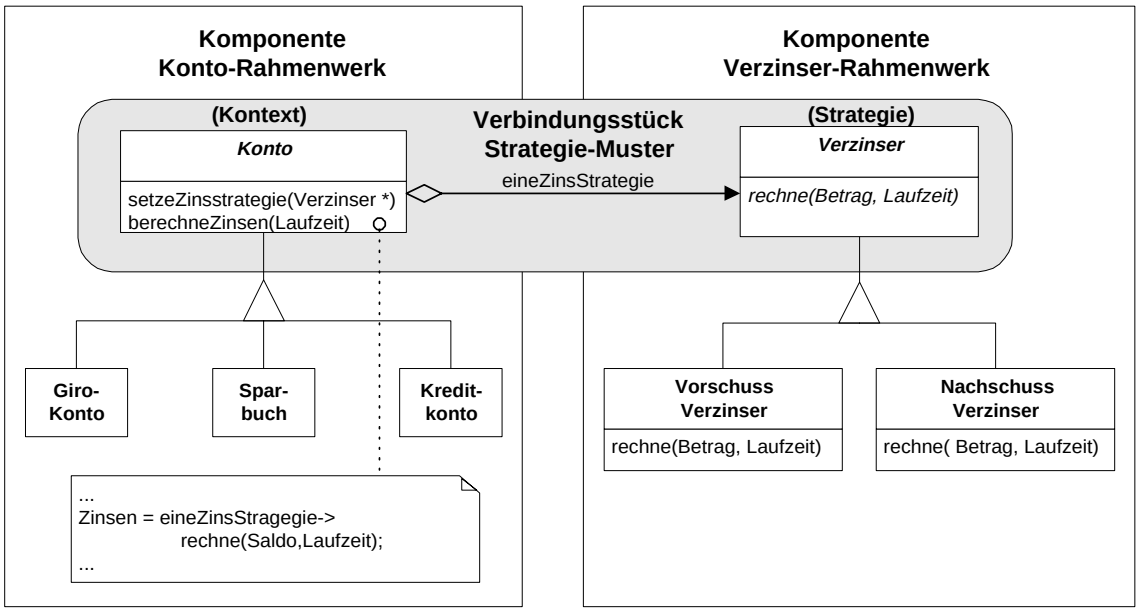


Abb. 2.7: Rahmenwerke als Komponenten und Entwurfsmuster als Verbindungsstücke

2.7 Zusammenfassung und Ausblick

Das in diesem Kapitel vorgestellte Prozeßmodell der objektorientierten Softwareentwicklung identifiziert diejenigen Kontexte, die das fachliche und softwaretechnische Modell beeinflussen. Zusammenfassend ergibt sich das folgende Bild:

- Der *Anwendungsbereich*, der einen späteren Einsatzkontext des zu konstruierenden Softwaresystems bildet, ist die fachliche Ausgangsbasis für die Konstruktion
- *des fachlichen Modells*. Dieses wird erstellt, indem der Softwareentwickler die im Anwendungsbereich relevanten Aufgaben und Tätigkeiten analysiert und dabei die dort

vorhandenen Gegenstände und deren Umgangsformen identifiziert. Durch den Prozeß der Abstraktion werden ähnliche Gegenstände auf den Begriff gebracht. Diese werden in Kompositions- und/oder Generalisierungsbeziehungen zueinander gesetzt, wodurch ein Begriffsmodell entsteht. Die Gestaltungsmöglichkeiten des fachlichen Modells werden durch entsprechende

- *Leitbilder und Entwurfsmetaphern* geprägt. Beispiele dieser Arbeit beziehen sich dabei immer auf das Leitbild vom Arbeitsplatz für qualifizierte menschliche Tätigkeit, das durch die Entwurfsmetapher Werkzeug, Automat und Material fachlich und softwaretechnisch konkretisiert wird. Die softwaretechnische Konkretisierung erfolgt durch entsprechende
- *Entwurfsmuster*. Entwurfsmuster beschreiben die Anordnung und die Zusammenarbeit von Klassen und Objekten, um ein kontextbezogenes Entwurfsproblem zu lösen.
- *Softwarearchitekturen* organisieren und strukturieren das softwaretechnische Modell als eine Menge interagierender Komponenten, die in einem objektorientierten Softwaresystem als Klassenbibliotheken und Rahmenwerke zur Verfügung gestellt werden. Verbindungsstücke beschreiben die Kopplung und das Zusammenspiel dieser Komponenten. Softwarearchitekturen prägen das
- *softwaretechnische Modell*. Dieses wird weiterhin beeinflußt durch den Inhalt und die Strukturen des fachlichen Modells, das verwendete Leitbild, die eingesetzten Entwurfsmetaphern sowie die benutzten Entwurfsmuster. Es wird durch das Design- und Implementierungsmodell konkretisiert. Die verwendete Technik umfaßt die zur Konstruktion eines Anwendungssystems benötigte Technik sowie die eingesetzte Technik, um das Anwendungssystem so zu gestalten, daß es den gestellten fachlichen Anforderungen bestmöglich genügen kann. Die Menge aller Schnittstellen zur verwendeten Technik wird zur
- *Systembasis* zusammengefaßt. Sie beeinflußt das Implementierungsmodell. Die verwendete Technik wird für die Konstruktion des softwaretechnischen Modells weiterhin in Form eines
- *Modells der verwendeten Technik* benutzt. Dieses Modell repräsentiert das subjektive Wissen des Softwareentwicklers darüber, wie die Technik zur Konstruktion des softwaretechnischen Modells zu verwenden ist. Zudem bestimmt das Modell der verwendeten Technik die Softwarearchitekturen und Entwurfsmuster, die sinnvoll bei der Erstellung eines Softwaresystems eingesetzt werden können. Das softwaretechnische Modell wird unter Verwendung geeigneter Werkzeuge zu
- wiederverwendbaren *Bibliotheken* und *Anwendungssystemen* montiert. Die Systemkomponenten stützen sich dabei auf die Systembasis ab, die die Schnittstelle zur verwendeten Technik repräsentiert. Das Wissen, das ein Entwickler über den Typ des zu entwickelnden Anwendungssystems hat, beeinflußt die Erstellung des fachlichen und softwaretechnischen Modells.

Die drei Bereiche Leitbilder und Entwurfsmetaphern, Entwurfsmuster und Softwarearchitekturen sowie Modell der verwendeten Technik beinhalten die technischen Kenntnisse, Fähigkeiten und Möglichkeiten, die bei der Konstruktion eines Anwendungssystems von Interesse

sind. Laut Brockhaus[8] ist die Gesamtheit dieser drei Bereiche als Technologie zu bezeichnen. Den Begriff Technologie verwende ich im weiteren wie folgt:

Begriff 2.15 Technologie

Der Begriff der Technologie umfaßt die technischen Kenntnisse, Fähigkeiten und Möglichkeiten, die die Struktur und den Inhalt des fachlichen und softwaretechnischen Modells prägen. Im Detail sind dies: Leitbilder und Entwurfsmetaphern, Softwarearchitekturen und Entwurfsmuster sowie das Modell der verwendeten Technik.

Der Anwendungsbereich, die eingesetzte Technologie und die verwendete Technik sind somit die Kontexte, die das fachliche und softwaretechnische Modell prägen. Dieser Zusammenhang ist in Abb. 2.8 präsentiert, welche eine Konkretisierung der Abb. 2.3 darstellt.

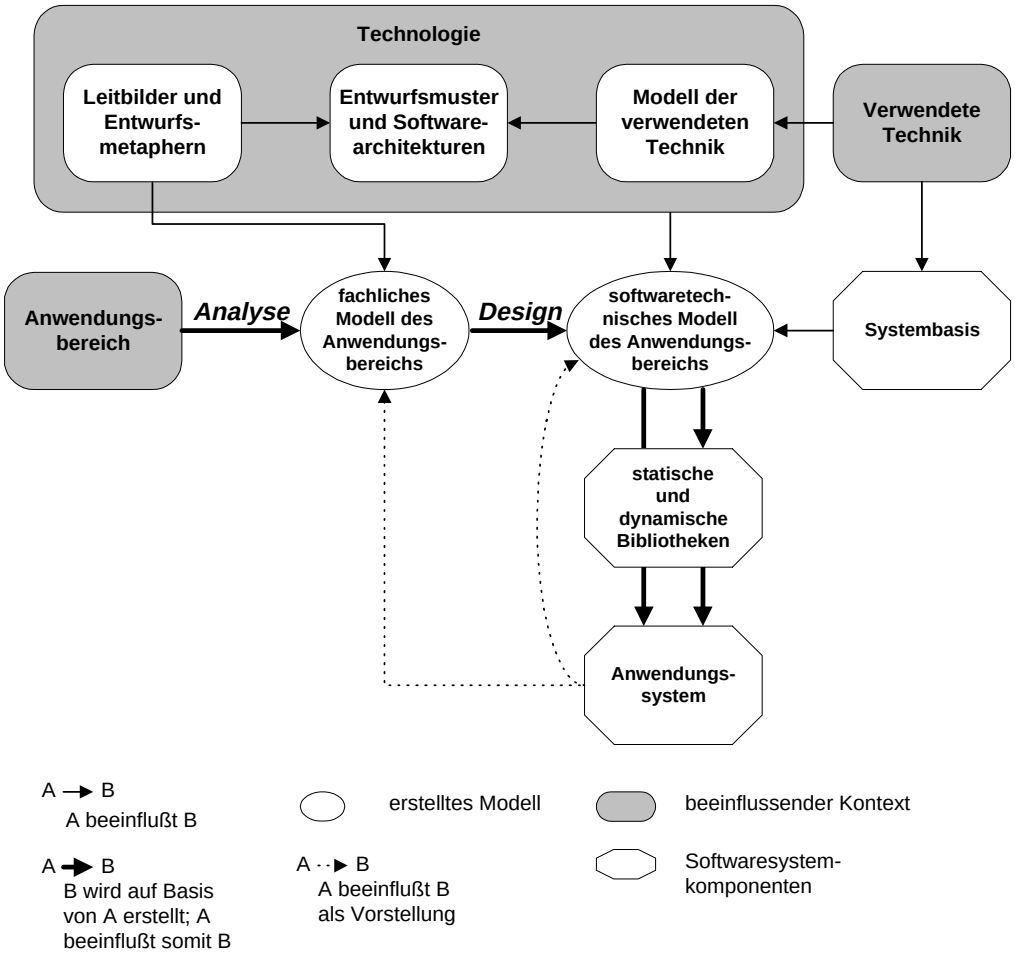


Abb. 2.8: Kontexte, Modelle und Softwaresystemkomponenten im objektorientierten Entwicklungsprozeß

[8] Im Brockhaus ist der Begriff Technologie wie folgt definiert: "Heute wird mit Technologie daneben auch die Gesamtheit der technischen Kenntnisse, Fähigkeiten und Möglichkeiten, das technische Wissen eines Gebietes bezeichnet." ([Bro86], S. 680)

Als Vorgabe für ein systematisches Vorgehen bei der Softwareentwicklung beschreibt die eingesetzte Methode, wie der Softwareentwicklungsprozeß durchgeführt und organisatorisch eingebettet werden soll (vgl. [Ree96]). Insbesondere schreibt die Methode vor, welche Techniken und Modellelemente zur Konstruktion der fachlichen und softwaretechnischen Modelle zu verwenden sind. Die in Abb. 2.8 dargestellten Zusammenhänge lassen sich daher auch nur vor dem Hintergrund argumentieren und verstehen, daß zur Konstruktion des Anwendungssystems eine objektorientierte Methode eingesetzt wird. Insbesondere prägt die gewählte Methode die Art des Technologiekontextes, dem sie selbst zugeordnet ist. Methode bzw. Technologiekontext besitzen daher Metacharakter.

Die strukturerhaltende Transformation des fachlichen Modells in das softwaretechnische Modell wird durch den Einsatz objektorientierter Entwicklungsmethoden unterstützt. So existiert bei geeigneter Modellierung eine Analogie zwischen den Konzepten des Anwendungsbereichs zu den Begriffen im fachlichen Modell und schließlich zu den Klassen des softwaretechnischen Modells. Dies ist in Abb. 2.9 in tabellarischer Form dargestellt. Die Abbildung repräsentiert eine Detaillierung der Abb. 2.1.

Anwendungsbereich	fachliches Modell	softwaretechnisches Modell
Begriff		Klasse
Gegenstand	—	Objekt
Umgangsform		Operation
Generalisierung und Spezialisierung		Vererbung
Komposition und Beziehung		Aggregation, Assoziation
Begriffshierarchie		Klassenhierarchie
Beispiele		
Girokonto	Girokonto	Klasse `GiroKonto`
einzahlen	einzahlen	Operation `einzahlen(Betrag)`
Schreibmaschine	Formulareditor	Klassen: `IAKEditor` `FKEditor`

Abb. 2.9: Die Modellelemente des Anwendungsbereichs, des fachlichen und softwaretechnischen Modells sowie konkrete Beispiele für Begriff und Umgangsform

Die durch die Strukturähnlichkeit der Modellelemente des Anwendungsbereichs, des fachlichen und softwaretechnischen Modells ermöglichte bruchlose Modellierung wird von vielen Autoren (vgl. [Jac92], [KGZ+94], [Mey88], [Ree96], [Rum91] und [WWW93]) als einer der entscheidenden Vorteile der Objektorientierung gegenüber herkömmlichen Analyse- und Designmethoden gesehen. Jacobson schreibt denn auch:

> "The analysis model has provided us with an ideal structure which we shall try to keep as long as possible. This is due to the fact that object-oriented analysis bases the system structure on the human way of looking at reality, namely the objects, classification and hierarchical understanding used when people understand their surroundings. Thus the result from object-oriented analysis is easier to understand and thus also easier to maintain than the result from a function/data analysis.
>
> Another advantage of object-oriented analysis is that items with low modification probability are naturally identified and it is possible to isolate at an early stage items with a high probability of modification." ([Jac92], S. 80)

Im weiteren sollen die von Cook & Daniels angesprochenen Probleme, die sich bei der Abbildung von komplexen Anwendungsbereichen in ein gemeinsames Modell ergeben, mittels der in diesem Kapitel erarbeiteten Grundlagen angegangen werden. Der von mir verfolgte Ansatz sei hier als Ausblick bereits grob skizziert:

- Komplexe Anwendungsbereiche sind so in kleinere Einheiten zu zerlegen, daß das fachliche und softwaretechnische Modell dieser Einheiten unabhängig voneinander konstruiert werden kann. Die Teilung sollte an den im Anwendungsbereich existierenden fachlichen Strukturen orientiert sein. Denn gemäß der Argumentation von Jacobson (siehe oben), müßte diese Teilung leichter zu verstehen und besser zu warten sein. Die kleineren Einheiten sind mittels einer rahmenwerkbasierten Softwarearchitektur zu realisieren.
- Angesichts des Zieles, unternehmensweit integrierte Anwendungssysteme zu entwickeln, stellen Leitbilder und Entwurfsmetaphern sicher, daß die entstandenen Komponenten auf der Ebene der Systemhandhabung miteinander kombiniert werden können.
- Einheitlich verwendete Softwarearchitekturen und Entwurfsmuster sowie ein gemeinsames Modell der verwendeten Technik garantieren die softwaretechnische Integrier- und Kombinierbarkeit der Komponenten.
- Die fachliche Integrier- und Kombinierbarkeit der einzelnen Komponenten muß durch eine geeignete Wahl der Makrostruktur gewährleistet sein. Eine Definition der Makrostruktur muß somit diese Anforderungen berücksichtigen.

3 Semiotik und der Softwareentwicklungsprozeß

> "Ein Zeichen ist alles, was sich als signifizierender Vertreter für etwas anderes auffassen läßt. Dieses andere muß nicht unbedingt existieren oder in dem Augenblick, in dem ein Zeichen für es steht, irgendwo vorhanden sein. *Also ist die Semiotik im Grunde die Disziplin, die alles untersucht, was man zum Lügen verwenden kann.*" ([Eco76], S. 26)

Aus der vorangegangenen Diskussion wird deutlich, daß zwischen den erstellten Modellen und ihren beeinflussenden Kontexten eine Menge von komplexen Beziehungen existieren. Als Softwareentwickler müssen wir in der Lage sein, diese Beziehungen zu etablieren, zu kommunizieren und weiterzuentwickeln. Da sich große Softwaresysteme heute praktisch nur noch evolutionär entwickeln lassen (vgl. Abschnitt 2.2 und Kapitel 4), muß ein Wechsel zwischen den verschiedenen Modellen und Kontexten möglichst einfach vollzogen werden können.

Die Semiotik beschäftigt sich als Wissenschaftsdisziplin mit den Beziehungen, die zwischen sogenannten *Ausdrucks-* und *Inhaltsystemen* bestehen, und erklärt, wie diese Beziehungen aufgebaut werden. Die Semiotik zeigt auch, daß sich die Korrelationen um so einfacher etablieren lassen, je strukturähnlicher die Ausdrucks- und Inhaltssysteme sind.

Ich werde argumentieren, daß die im Softwareentwicklungsprozeß erstellten Modelle und ihre beeinflussenden Kontexte als solche Systeme aufgefaßt werden können und sich daher Ergebnisse aus der Semiotik auch auf die Softwareentwicklung anwenden lassen. Die Ergebnisse stützen die in den nachfolgenden Kapiteln ausgearbeitete Idee, anwendungsfachliche Makrostrukturen als Grundlage für Softwarearchitekturen stärker zu berücksichtigen.

Das Kapitel hat den folgenden Aufbau: Abschnitt 3.1 gibt eine Einführung in die Semiotik. Dabei werden die für diese Arbeit relevanten Begriffe Signifikationssystem, Code[9], S-Code, sowie Inhalts- und Ausdruckssystem eingeführt. In Abschnitt 3.2 zeige ich, daß im objektorientierten Softwareentwicklungsprozeß Ketten von Inhalts- und Ausdruckssystemen konstruiert werden. Auf dieser Basis lassen sich in Abschnitt 3.3 Ergebnisse aus der Semiotik auf die Softwareentwicklung übertragen. Sie betreffen vor allem die Bedeutung von Strukturen bei der Konstruktion von Anwendungssystemen.

Die Anwendung von Ergebnissen aus einer anderen Wissenschaftsdisziplin ist immer mit der Entscheidung verbunden, wie detailliert eine Einführung in ihre Grundlagen gestaltet sein muß. Da Erkenntnisse aus der Semiotik bislang in der Informatik noch keinen weit verbreiteten Eingang gefunden haben, habe ich mich entschlossen, die Einführung etwas

---

[9] Die Schreibweise Code werde ich verwenden, wenn ich von einem Code im Sinne der Semiotik rede. Die Schreibweise Kode wird eingesetzt, wenn damit Programmkode im Sinne der Informatik gemeint ist.

umfänglicher zu gestalten. Das vorgestellte Gedankengebäude beinhaltet daher mehr Ergebnisse, als für das reine Verständnis einzelner Ideen notwendig wäre. Diese Vorgehensweise soll es dem Leser erleichtern, die semiotischen Zusammenhänge und ihre Übertragung auf den objektorientierten Softwareentwicklungsprozeß besser nachvollziehen zu können. Die Einführung beruht dabei auf den drei Werken [Eco72], [Eco73] und [Eco76] von Umberto Eco. Diese Einschränkung ist möglich, da die Werke von Umberto Eco anerkannte Grundlagenwerke in der Semiotik darstellen. So empfiehlt Peter Børg Anderson in [And90] für Arbeiten im Bereich Semiotik und Softwareentwicklung die Werke europäischer Semiotiker, insbesondere die von Umberto Eco. Ulrich Piepenburg stützt sich in seiner Dissertationsschrift »Semantikerstellung in der Softwareentwicklung« (vgl. [Pie94]) ebenfalls auf Eco als den Theoretiker ab, der eine Theorie der Codes vorgelegt hat.

3.1 Grundlagen der Semiotik

Die Semiotik ist als die Lehre von den Zeichen definiert. Nach [Eco76] lassen sich folgende Wissenschaftsdisziplinen innerhalb der Semiotik unterscheiden:

- *Semiotik der Signifikation*: Im Sinne von Eco liegt eine Signifikation dann vor, "wenn – auf Basis einer zugrundeliegenden Regel – etwas der Wahrnehmung eines Empfängers Dargebotenes für etwas anderes steht." ([Eco76], S. 28). So signalisiert beispielsweise ein Totenkopf auf einer Flasche, daß in der Flasche eine giftige Substanz enthalten ist. Eco erklärt Signifikation mit seiner »Theorie der Codes« (vgl. [Eco76], S. 76 ff.).
- *Semiotik der Kommunikation*: Im Sinne von Eco liegt ein Kommunikationsprozeß dann vor, "wenn die durch ein Signifikationssystem bereitgestellten Möglichkeiten benutzt werden, um physische Ausdrücke für viele praktische Zwecke zu erzeugen." ([Eco76], S. 23). Technisch läßt sich Kommunikation wie folgt definieren: von einem Sender wird eine Nachricht (Zeichen) in Form von Signalen gesendet, die ein Empfänger in Form einer Botschaft entgegennimmt. Der Empfänger kann die Nachricht allerdings nur dann als Botschaft verstehen, wenn zwischen Sender und Empfänger ein Code existiert, mit dem der Nachricht eine Bedeutung zugeordnet werden kann. Dieser Code wird durch ein zugrundeliegendes Signifikationssystem bereitgestellt. Kommunikationsprozesse untersucht Eco vor dem Hintergrund seiner »Theorie der Zeichenerzeugung« (vgl. [Eco76], S. 203 ff.).

Erkenntnisse aus der Signifikations- und Kommunikationssemiotik sind für die Softwareentwicklung zumindest in zweierlei Hinsicht relevant:

a) Bei der Konstruktion des fachlichen und softwaretechnischen Modells werden Zeichen in Form von Begriffen, Umgangsformen, Klassen, Klassendiagrammen, usw. erzeugt. Dies geschieht im Rahmen eines Kommunikationsprozesses, der gemeinsam von den am Softwareentwicklungsprozeß beteiligten Gruppen getragen wird (vgl. [Flo94]). Auf diesen Bereich lassen sich Erkenntnisse aus der Theorie der Zeichenerzeugung anwenden.

b) In Kapitel 2 wurde aufgezeigt, welche Korrelationen einerseits zwischen softwaretechnischem Modell und fachlichem Modell bzw. verwendeter Technologie und andererseits zwischen fachlichem Modell und Anwendungsbereich bzw. verwendeter Technologie bestehen. Unter semiotischen Gesichtspunkten sind diese Korrelationen mittels eines Signifikationssystems beschreibbar. Eine Signifikation liegt etwa dann vor, wenn ein

Begriff des fachlichen Modells für eine fachliche Konvention im Anwendungsbereich steht.

Gegenstand dieser Arbeit sind die Korrelationen zwischen den erstellten Modellen und den Kontexten, die diese Modelle beeinflussen. Wie einzelne Modelle und Kontexte konstruiert bzw. erzeugt werden, soll in dieser Arbeit nicht problematisiert werden. Auf die Semiotik der Kommunikation wird deshalb im weiteren nicht mehr eingegangen.

Im oben aufgeführten Punkt (b) wurde dargelegt, daß unter semiotischen Gesichtspunkten eine Korrelation zwischen Modellen und ihren beeinflussenden Kontexten durch ein Signifikationssystem beschrieben werden könnte. Um also Ergebnisse aus der Semiotik der Signifikation auf den Softwareentwicklungsprozeß übertragen zu können, ist zu klären, ob diesem ein Signifikationssystem unterliegt. Zum Begriff des Signifikationssystems schreibt Eco:

> "Ein Signifikationssystem (und damit ein Code) liegt dann vor, wenn eine sozial konventionalisierte Möglichkeit zur Erzeugung von Zeichen-Funktionen gegeben ist, gleichgültig ob die Träger solcher Funktionen diskrete Einheiten sind, die man Zeichen nennt, oder größere Teile des Diskurses, wenn nur die Korrelation durch soziale Konvention vorher festgesetzt wurde." ([Eco76], S. 23)

Ausgehend von der dargelegten Argumentationskette und dem Zitat von Eco ergibt sich für die einführenden Abschnitte der nachfolgende Aufbau: zuerst wird der Begriff des Zeichens erläutert; darauf aufbauend wird ein elementares Kommunikationsmodell vorgestellt, mit dessen Hilfe sich die Begriffe Code und Zeichenfunktion erklären lassen. Am Ende jedes Abschnitts wird in Form von Fragen ein Ausblick gegeben, wie sich die Erkenntnisse der Semiotik auf die für diese Arbeit relevanten Aspekte der Softwareentwicklung übertragen lassen.

3.1.1 Das Zeichen und das semiotische Dreieck

Untersucht man Zeichen im Kontext von Signifikation, so wird ein Zeichen als etwas aufgefaßt, daß aus drei Komponenten besteht: dem Signifikanten, dem Signifikat und dem Referenten (vgl. Abb. 3.1). Die Bedeutung der drei Begriffe läßt sich folgendermaßen charakterisieren (vgl. [Eco73]):

- Der *Signifikant* ist der Bezeichner oder der Ausdruck eines Zeichens, zum Beispiel der Bezeichner Computer, wenn wir von einem Computer reden. Oft verbindet man mit dem Signifikanten auch das eigentliche Zeichen als physische Entität. Eco schlägt vor, den Signifikanten eines Zeichens in Schrägstriche einzuschließen, um ihn von den anderen beiden Einheiten des Zeichens syntaktisch zu unterscheiden. Für das Beispiel Computer stellt sich der Signifikant syntaktisch folgendermaßen dar: /Computer/.
- Das *Signifikat* meint das, was von einem Zeichen ausgesagt wird und keine physische Entität darstellt. Das Signifikat wird oft auch als Sinn oder Inhalt bezeichnet. Im Falle des Zeichens Computer kann ein zugehöriges Signifikat durch die Analyse von Computern als Gegenstand in entsprechenden Arbeitszusammenhängen des Anwendungsbereichs definiert werden. Es soll bereits hier darauf hingewiesen werden, daß diese Sichtweise der Bedeutungszuweisung problematisch ist, da es für die zu modellierenden

Begriffe im Anwendungsbereich nicht immer einen physischen Gegenstand geben muß. Eco schlägt auch für das Signifikat eine besondere Syntax vor: «Computer».

- Der *Referent* umfaßt die Gegenstände, auf die sich das Zeichen bezieht. Am Beispiel des Computers sind dies alle Computer, die es in der Welt je gab, gibt oder geben wird. Das von Eco vorgeschlagene syntaktische Zeichen für den Referenten ist wie folgt: //Computer//.

Diese Dreiteilung wird in der Semiotik durch das semiotische Dreieck repräsentiert. In Klammern führe ich alternative Bezeichner auf, die von anderen Autoren verwendet werden.

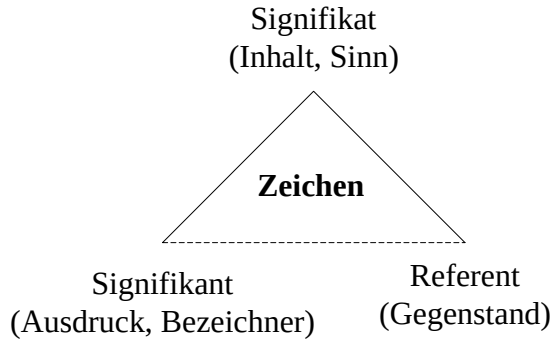


Abb. 3.1: Das semiotische Dreieck (aus [Eco73], S. 28)

Der Referent im semiotischen Dreieck repräsentiert die Gegenstände, auf die sich ein Zeichen bezieht. Am Beispiel des Zeichens Girokonto zeigt sich aber, daß ein handgreiflicher Gegenstand //Girokonto// im Anwendungsbereich nicht zwangsläufig existieren muß. Eco schreibt hierzu:

> "Man braucht nicht sehr scharfsinnig sein, um zu erkennen, wie unbestimmt dieser Begriff des Referenten ist, aber auch, um gleichzeitig zu sehen, daß er vorläufig auf die bequemste Weise einer Erfahrung Rechnung zu tragen erlaubt, die wir jeden Tag machen: daß wir, wenn wir mit Zeichen arbeiten, zumeist der Ansicht sind, damit Dinge zu bezeichnen." ([Eco73], S. 29)

Der Verlust des Referenten als physisch greifbares Ding verhindert allerdings nicht, daß zu einem Signifikanten ein Signifikat korreliert werden kann. So existiert bei Bankfachleuten auch ohne die handgreifliche Existenz eines Girokontos eine hinreichende Klarheit über seine Bedeutung. Der Verlust des Referenten als physisch greifbares Ding wird in Abb. 3.1 dadurch gekennzeichnet, daß der Referent mit dem Signifikanten nur über eine gestrichelte Linie verbunden ist. Wenn in dieser Arbeit der Begriff des Gegenstandes im Kontext „Analyse eines Anwendungsbereichs“ verwendet wird, so sind damit sowohl physisch greifbare Dinge, wie etwa ein Formular oder ein Geldausgabeautomat, als auch Entitäten, die ideellen Charakter aufweisen, wie beispielsweise ein Girokonto oder ein Zinssatz, gemeint.

Das semiotische Dreieck vermittelt in dieser Form noch eine weitere Annahme: die Mutmaßung, daß mit einem Signifikanten nur ein Signifikat verbunden sein kann. Durch einfache Beispiele kann diese Fehlinterpretation sofort widerlegt werden: mit dem Signifikanten

/Totenkopf/ wird auf einer Fahne «Pirat» und auf einer Flasche «Gift» denotiert. Mit «Pirat» wird weiter der Sinn «Flucht» oder «Angriff» mit «Gift» der Sinn «Krankheit» oder «Tod» verbunden. Die Klasse `GiroKonto` denotiert ebenfalls unterschiedliche Inhalte in den Kontexten fachliches Modell und verwendete Technologie (vgl. das Beispiel des Girokontos in Kapitel 2). Innerhalb der Semiotik wird dieses Problem durch ein semantisches Modell gelöst, das berücksichtigt, daß zu einem Signifikanten mehrere Signifikate korreliert werden können. In Anlehnung an Eco wird der Begriff des Zeichens nun wie folgt gefaßt:

Begriff 3.1 Zeichen

Die Korrelation eines Signifikanten mit einem Signifikat wird als Zeichen betrachtet. Ein Signifikant kann sich auf mehrere Signifikate beziehen, und ein Signifikat kann mit mehreren Signifikanten korreliert werden. Der Referent eines Zeichens muß nicht notwendigerweise existent sein.

Das Signifikat eines Signifikanten kann durch eine Definition erklärt werden, so wie wir sie in Begriffsglossaren, Wörterbüchern oder Enzyklopädien vorfinden. Alternativ kann auch eine erklärende Graphik gezeichnet werden. Beide Lösungen haben eines gemeinsam: anstelle des zu erklärenden Signifikanten wird ein anderes Zeichen dargeboten. Dieses andere Zeichen wird in der Semiotik Interpretant genannt. Der Begriff des Interpretanten wird wie folgt vereinbart:

Begriff 3.2 Interpretant

Ein Interpretant ist ein Zeichen, das ein anderes Zeichen in einem gegebenen Kontext erklärt.

Da Interpretanten selbst wieder Zeichen sind, werden sie wieder durch Interpretanten erklärt, die wiederum Zeichen sind und so weiter. Irgendwann herrscht beim Empfänger eines Zeichens eine hinreichende Klarheit über dessen Sinn, und er kann mittels einer geeigneten Reaktion auf das Zeichen reagieren. Der Prozeß, bei dem Interpretanten durch Interpretanten erklärt werden, wird als unbegrenzte Semiose bezeichnet.

3.1.2 Ein elementares Kommunikationsmodell und der Begriff des Codes

Signifikation bedeutet, daß etwas der Wahrnehmung eines Empfängers Dargebotenes für etwas anderes steht. Dieses Darbieten kann technisch als Kommunikation gesehen werden, bei der ein Sender eine Nachricht (Zeichen) in Form von Signalen sendet, die ein Empfänger mit Hilfe eines Codes in eine Botschaft übersetzt. Das Phänomen des Codes steht also in engem Zusammenhang mit einem Kommunikationsprozeß. Es erscheint somit notwendig, diesen Zusammenhang anhand eines Beispiels näher zu untersuchen. Das Beispiel ist eine komprimierte Fassung des Beispiels aus [Eco76], S. 58 ff., das ursprünglich auf einem Modell von DeMauro [DeM71] beruht, welches in der Semiotik als eine der klarsten und nützlichsten Einführungen in die Problematik der Codierung gilt. Das Beispiel geht dabei von einer Kommunikation zwischen zwei Maschinen aus. Eco begründet die Wahl des Beispiels damit, daß man die elementaren Strukturen der Kommunikation dort aufsuchen muß, wo Kommunikation minimal stattfindet, nämlich zwischen zwei Maschinen. Dies erleichtert die

Konstruktion eines elementaren Kommunikationsmodells. Das Beispiel stellt sich folgendermaßen dar:

Ein Ingenieur im Tal möchte wissen, welcher Pegelstand momentan in einem Staubecken am Berg herrscht. Er definiert sich dafür die vier relevanten Pegelstände «Gefahrpegel», «Alarmpegel», «Sicherheitspegel» und «zu niedriger Pegelstand». Um diese Informationen über den Pegelstand ins Tal zu übertragen, installiert der Ingenieur im Staubecken einen Schwimmer, der ein Sendegerät in Gang setzt. Dieses ist fähig, ein elektrisches Signal zu erzeugen, das durch einen Kanal an ein Empfangsgerät übermittelt wird. Das Empfangsgerät transformiert die Signale in eine Reihe von Elementen, die für den Empfänger eine Botschaft darstellen. Der Empfänger, auch ein Apparat, kann nun auf einen gegebenen Pegelstand, zum Beispiel «Gefahrpegel», geeignet reagieren, zum Beispiel mit «Senken des Pegelstandes». Weitere zu den Pegelständen passende Reaktionen sind: «Alarmzustand», «Ruhezustand» und «Anheben des Pegelstandes». Die Situation der Kommunikation läßt sich graphisch folgendermaßen darstellen[10]:

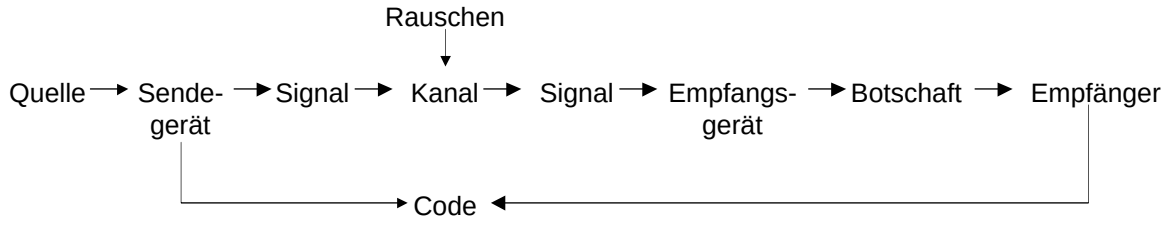


Abb. 3.2: Ein Kommunikationsmodell (aus [Eco76], S. 58)

Um die entsprechenden Informationen über den Pegelstand zu übermitteln, führt der Ingenieur einen Code ein, der aus insgesamt vier verschiedenen Signalen A, B, C und D besteht, wobei immer zwei Signale zu einer Botschaft kombiniert werden, um die Übertragung sicherer zu gestalten[11], also AA, AB, AC, usw. Daraus ergeben sich theoretisch 16 mögliche Botschaften. Wenn das zeitliche Aufeinanderfolgen der Signale keine Relevanz für die zu übermittelnde Botschaft hat, verfügt der Ingenieur über folgende sechs Botschaften: AB, BC, CD, AD, AC und BD. Von diesen verwendet er die vier Botschaften AB, BC, CD und AD, um die vier Pegelstände am Staubecken zu übermitteln und zu den Reaktionen im Tal in Beziehung zu setzen. Die Botschaften AC und BD bleiben ungenutzt. Zur Übermittlung einer Information werden also zwei Signale benutzt, für «Gefahrpegel» beispielsweise die Signale A und B. Damit der Ingenieur auf Empfängerseite besser kontrollieren kann, wie der Apparat arbeitet, entscheidet er, die Signale A, B, C und D durch vier Glühbirnen, die unterschiedliche Farben haben, zu visualisieren. Daraus ergibt sich abschließend die Darstellung in Abb. 3.3.

[10] Diese Form der Darstellung eines Kommunikationsvorgangs ist vielen Informatikern aus der Disziplin der Codierungstheorie bekannt. Die Codierungstheorie beschäftigt sich allerdings mit der optimalen Verschlüsselung und Übertragung von Daten und nicht mit der beim Empfänger einer Botschaft stattfindenden Korrelation von Daten und ihrem Sinn. Die Thematik der Codierungstheorie muß daher im weiteren nicht vertieft werden.

[11] In [Eco76] werden zuerst einfachere Codes eingeführt, die dann erweitert werden, um den Code sicherer und störungsfreier zu machen. Auf diese Ausführung wurde hier verzichtet.

(a) Glühbirnen		(b) Pegelstände oder Informationen über den Pegelstand		(c) Reaktion des Empfängers
AB	==	Gefahrpegel	==	Senken des Pegelstandes
BC	==	Alarmpegel	==	Alarmzustand
CD	==	Sicherheitspegel	==	Ruhezustand
AD	==	zu niedriger Pegelstand	==	Anheben des Pegelstandes

Abb. 3.3: Das Staubeckenmodell (aus [Eco76], S. 61)

Die in der Abb. 3.3 aufgeführte Tabelle führt eine Reihe von Codes ein. Die Semiotik definiert nun, was als Code bezeichnet werden kann und welche Kategorien einzelner Codes in dem Beispiel auftreten. Eco unterscheidet hierzu die folgenden vier Phänomene (vgl. [Eco76], S. 62), die umgangssprachlich mit dem Begriff Code belegt sind:

(a) eine Reihe von Signalen, die mithin als *syntaktisches System* bezeichnet werden. Im Beispiel des Staubeckenmodells sind dies die Signale A, B, C und D. Unter Einsatz von Kombinationsregeln lassen sich daraus Ketten dieser Signale generieren, etwa die Botschaften AB, BC, CD und AD. Eine Korrelation der Botschaften mit möglichen Informationen oder entsprechenden Reaktionen muß nicht gegeben sein. Die Signale könnten genauso gut auch beliebige andere Informationen übermitteln oder beliebige andere Reaktionen hervorrufen. Das System kann sogar existieren, ohne daß irgendeine Information mitgeteilt oder irgendeine Reaktion ausgelöst werden soll. Die Signale sind somit an keine Signifikations- oder Kommunikationsabsicht gebunden.

(b) eine Reihe von Informationen über den am Staubecken herrschenden Pegelstand. Im Beispiel sind dies die Informationen: «Gefahrpegel», «Alarmpegel», «Sicherheitspegel» und «zu niedriger Pegelstand». Diese Informationen lassen sich durch eine Kette von Signalen (siehe (a)) transportieren. Sie sind allerdings unabhängig davon und lassen sich daher auch durch andere Signale, etwa von Morsezeichen, übermitteln. Die Informationen können zu einer Reihe möglicher Kommunikationsinhalte werden. Dieser Komplex von Inhalten wird als *semantisches System* bezeichnet.

(c) eine Reihe von möglichen Verhaltensreaktionen des Empfängers. Im Beispiel sind dies die Reaktionen: «Senken des Pegelstandes», «Alarmzustand», «Ruhezustand» und «Anheben des Pegelstandes». Diese Reaktionen sind unabhängig von den Signalen A, B, C und D, da sie auch von einem anderen (a) System ausgelöst werden könnten. Genauso gut ließe sich das System (a) verwenden, um andere Reaktionen beim Empfänger zu bewirken. Die Verhaltensreaktionen sind ebenfalls eigenständig in Bezug auf die Informationen, die durch das (b) System definiert sind.

(d) eine Regel, die Elemente des (a) Systems mit Elementen des (b) oder (c) Systems verbindet. Diese Regel wird in dem Beispiel des Staubeckenmodells (vgl. Abb. 3.3) durch die horizontale Zuordnung der Elemente der (a), (b) und (c) Systeme visualisiert (beispielsweise AB == «Gefahrpegel» == «Senken des Pegelstandes»). Insgesamt lassen sich für die drei Systeme folgende Kombinationen, die eine Regel darstellen, angeben:

> (I) eine Regel, die syntaktische Signale des Systems (a) mit bestimmten Informationen über Pegelstände des semantischen Systems (b) korreliert, (II) eine Regel, die obige Korrelation (I) mit einer bestimmten Empfängerreaktion verbindet und (III) eine Regel, die Signale des Systems (a) auch dann mit Reaktionen des Systems (c) verbindet, wenn keine Information über den Pegelstand übermittelt werden soll.

Eco macht in diesem Zusammenhang deutlich, daß nur die komplexe Form einer Regel, wie sie in (d) beschrieben ist, korrekterweise als Code bezeichnet werden darf, da sie eine kodierte Zuordnung von Elementen zweier Systeme beschreibt. Er unterscheidet deshalb Systeme vom Typ (a), (b) und (c) von Systemen des Typs (d), indem er die unterschiedlichen Begriffe S-Code und Code einführt. Eco führt dazu aus:

> "Ich werde deshalb ein System von Elementen in der Art der unter *(a)*, *(b)* und *(c)* beschriebenen als *S-Code* (oder Code als System) bezeichnen; während ich eine Regel, die die Elemente eines S-Codes mit denen eines oder mehrerer anderer S-Codes so wie bei (d) beschrieben korreliert, einfach als *Code* bezeichne." ([Eco76], S. 64)

Da sowohl Codes als auch S-Codes wichtige Grundelemente eines Signifikations- und Kommunikationsprozesses darstellen, sollen sie als Begriffe auf Grundlage der Ausführungen über die vier unterschiedlichen Code-Phänomene wie folgt gefaßt werden:

> **Begriff 3.3 S-Code**
>
> Ein S-Code ist ein System, das aus einer endlichen Zahl von Elementen besteht, die oppositionell strukturiert sind. Er kann daher auch als Struktur aufgefaßt werden. Interne Kombinationsregeln ermöglichen es, sowohl endliche als auch unendliche Ketten der Elemente zu generieren. S-Codes können auch unabhängig von jeglicher Signifikations- oder Kommunikationsabsicht existieren.

> **Begriff 3.4 Code** (1. Fassung)
>
> Ein Code ist eine Regel, die die Elemente eines S-Codes mit denen eines oder mehrerer anderer S-Codes korreliert.

Beispiele für S-Codes sind etwa der phonologische Code oder die Zeichen ›kurz‹ und ›lang‹ des Morsealphabets. Das Morsealphabet oder der kulturelle Code, der es einer Person in Mitteleuropa ermöglicht, den Begriff Technik richtig zu interpretieren, sind zwei Beispiele für Codes.

S-Codes können ohne Codes existieren. Ihre Aufgabe ist es, Systemen eine Struktur zu geben, um eine Korrelation mit anderen Systemen zu ermöglichen, die ebenfalls durch einen S-Code beschrieben sind. S-Codes stellen somit eine notwendige, aber keine hinreichende Bedingung für die Existenz von Codes dar. Allerdings weist Eco ausdrücklich darauf hin, daß die Struktur der S-Codes die Etablierung eines Codes erleichtern oder erschweren kann. Er schreibt hierzu:

> "Die strukturelle Anordnung eines Systems hat eine wichtige praktische Funktion und zeigt gewisse Eigenschaften. Sie macht eine Situation verstehbar und mit anderen Situationen vergleichbar und bereitet damit eine mögliche Codierungs-Korrelation vor." ([Eco76], S. 66 f.)

Falls den Beziehungen zwischen Anwendungsbereich, fachlichem und softwaretechnischem Modell sowie verwendeter Technologie ein Signifikationssystem unterliegt, stellt dieses Zitat eine wichtige Aussage dar. Denn dann würde der strukturelle Aufbau dieser Modelle bzw. Kontexte die Konstruktion eines Codes vereinfachen. Das Zitat soll daher in einem Ergebnis zusammengefaßt werden:

Ergebnis 3.1

Die Etablierung eines Codes wird durch einen geeigneten strukturellen Aufbau der zu korrelierenden Systeme unterstützt.

Betrachtet man die durch einen Code miteinander korrelierten S-Codes als ein übermittelndes und übermitteltes System, so können die Elemente des übermittelnden Systems als Ausdruck des übermittelten Systems und die Elemente der übermittelten Systems als Inhalt des übermittelnden Systems aufgefaßt werden. Im Beispiel des Staubeckenmodells ist die Botschaft AB Ausdruck des Inhalts «Gefahrpegel» (vgl. Abb. 3.4).

Ausdruck	**Inhalt**
Element eines übermittelnden Systems	Element eines übermittelten Systems
AB	Gefahrpegel

Abb. 3.4: Ausdruck und Inhalt

Die beiden Begriffe Inhalt und Ausdruck werden daher wie folgt definiert:

Begriff 3.5 Ausdruck und Inhalt

Ein Element eines übermittelnden Systems wird als Ausdruck, ein Element eines übermittelten System wird als Inhalt bezeichnet.

Betrachtet man die durch einen Code verbundenen S-Codes als Systeme, so etabliert ein Code eine Korrelation eines Ausdruckssystems mit einem Inhaltssystem. Im Beispiel des Staubeckenmodells bilden die Botschaften AB, BC, CD und AD ein Ausdruckssystem, das durch einen Code mit dem Inhaltssystem «Gefahrpegel», «Alarmpegel», «Sicherheitspegel» und «zu niedriger Pegelstand» verbunden ist (vgl. Abb. 3.5).

Der Code betrachtet die beiden Systeme dabei unter rein formalen Aspekten, d.h. er berücksichtigt nicht die Bedeutung einzelner Elemente, sondern nur die Beziehung zwischen zwei Elementen aus dem Ausdrucks- bzw. Inhaltssystem.

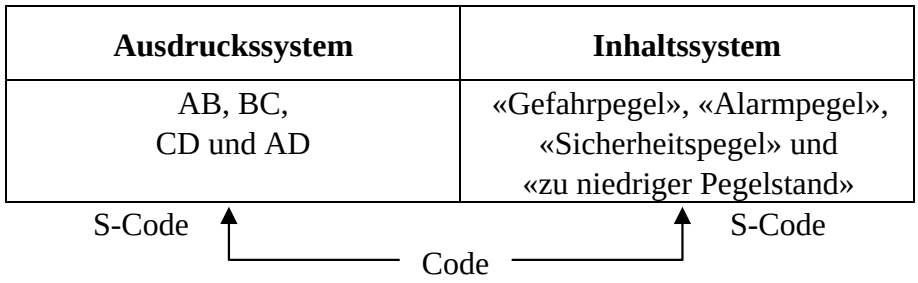


Abb. 3.5: Ausdruckssystem und Inhaltssystem

Unter Verwendung der Begriffe Ausdruck und Inhalt bzw. Ausdruckssystem und Inhaltssystem läßt sich nun eine zweite Fassung des Begriffs Code definieren:

Begriff 3.6 Code (2. Fassung)

Ein Code etabliert die Korrelation eines Ausdruckssystems mit einem Inhaltssystem.

Ein Code verbindet ein Ausdruckssystem mit einem Inhaltssystem. Einzelne Elemente dieser Systeme werden dabei durch eine Funktion verbunden, die auf Basis des Codes, der die Systeme korreliert, eingerichtet wird. Eine solche Funktion wird in der Semiotik als Zeichenfunktion bezeichnet. Der Begriff Zeichenfunktion ist dabei wie folgt definiert:

Begriff 3.7 Zeichenfunktion

Eine Zeichenfunktion korreliert ein Element eines Ausdruckssystems mit einem Element eines Inhaltssystems. Ein Element kann in eine weitere Korrelation eintreten, wodurch eine neue Zeichenfunktion entsteht.

Ich möchte jetzt schon einmal in Form einer Frage einen Ausblick auf Abschnitt 3.2 geben, in dem die hier aus der Semiotik vorgestellten Erkenntnisse in Zusammenhang mit der Softwareentwicklung gebracht werden:

- Können die Beziehungen, die zwischen den Kontexten und den erstellten Modellen bzw. zwischen den erstellten Modellen selbst existieren (vgl. Abb. 2.8), als Codes aufgefaßt werden? In diesem Fall wären Modelle und Kontexte als Ausdrucks- bzw. Inhaltssysteme interpretierbar, etwa das softwaretechnische Modell als Ausdruckssystem des Inhaltssystems fachliches Modell.

Auf Grund der bis jetzt definierten Begriffe könnte nun überprüft werden, ob dem Softwareentwicklungsprozeß ein Signifikationssystem zugrunde liegt. Es sollen allerdings zuvor die beiden Begriffe denotative und konnotative Semiotik und die zur ihrer Darstellung verwendete Notation eingeführt werden. Denn insbesonders die Notation eignet sich sehr gut, um die später auf den Softwareentwicklungsprozeß übertragenen Ergebnisse aus der Semiotik graphisch zu präsentieren.

3.1.3 Denotative und konnotative Semiotiken

Reagiert ein Ingenieur auf die Übermittlung der Information «Niedrigwasser» mit «Wasserzufluß», so erfolgt seine Reaktion auf Grund der Tatsache, daß «Niedrigwasser»

durch das Signal AD (visualisiert durch Aufleuchten der Glühbirnen) signifiziert wurde. Der zum Inhalt «Wasserzufluß» korrelierte Ausdruck ist demnach nicht elementar. Intern kann dieser nicht-elementare Ausdruck erklärt werden als Inhalt «Niedrigwasser», der selbst wieder mit einem Ausdruck, dem Signal AD, in Beziehung steht. Der Ausdruck AD denotiert «Niedrigwasser» und konnotiert zugleich «Wasserzufluß». Die wiederholte Signifikation führt zu einer Überlagerung der im Staubeckenmodell vorhandenen Codes und läßt sich wie in Abb. 3.6 darstellen:

Ausdruck				Inhalt
Ausdruck		Inhalt		
AB	==	Gefahrpegel	==	Wasserablassen
BC	==	Alarmpegel	==	Alarmzustand
CD	==	Sicherheitspegel	==	Ruhezustand
AD	==	Niedrigwasser	==	Wasserzufluß

Abb. 3.6: Die Überlagerung von Codes am Beispiel des Staubeckenmodells (aus [Eco76], S. 84)

Nach Hjelmslev wird eine derartige Überlagerung von Codes als *konnotative Semiotik* (vgl. [Eco76], S. 84) bezeichnet. Eine konnotative Semiotik wird grundsätzlich durch folgende Form dargestellt:

Ausdruck		Inhalt
Ausdruck	Inhalt	

Abb. 3.7: Konnotative Semiotik nach Hjelmslev (aus [Eco76], S. 84)

Daß die Reaktion des Empfängers semiotisch als konnotativ betrachtet wird, ist nicht fest vorgegeben, sondern wurde durch den Ingenieur bewußt so konstruiert. Er hätte genauso gut zwei denotative Semiotiken etablieren können. Das Signal AD wäre dann durch einen ersten Code mit «Niedrigwasser» und durch einen zweiten direkt mit «Wasserzufluß» korreliert. Abb. 3.8 stellt das Staubeckenmodell als zwei denotativ korrelierte Codes dar.

Inhalt		Ausdruck		Inhalt
Gefahrpegel	==	AB	==	Wasserablassen
Alarmpegel	==	BC	==	Alarmzustand
Sicherheitspegel	==	CD	==	Ruhezustand
Niedrigwasser	==	AD	==	Wasserzufluß

Abb. 3.8: Das Staubeckenmodell als System zweier denotativer Codes

Auf Grund der Ausführungen lassen sich nun die Begriffe denotative und konnotative Semiotik folgendermaßen fassen:

Begriff 3.8 Denotative Semiotik

Eine denotative Semiotik liegt vor, wenn die korrelierten Ausdrucks- und Inhaltssysteme S-Codes sind.

Begriff 3.9 Konnotative Semiotik

Eine konnotative Semiotik liegt vor, wenn das mit dem Inhaltssystem korrelierte Ausdruckssystem selbst wieder eine denotative oder konnotative Semiotik darstellt.

Das Beispiel des Staubeckenmodells mit seinen drei S-Codes zeigt, daß die Korrelation der S-Codes als zwei denotative Semiotiken oder eine konnotative Semiotik nicht per se vorgegeben ist, sondern von dem Ingenieur, der das Modell konstruiert hat, so in Form von Codierungskonventionen festgelegt worden ist. Diese Tatsache soll abschließend in einem Ergebnis festgehalten werden:

Ergebnis 3.2

Ob ein Signifikant ein Signifikat denotiert oder konnotiert, wird durch die Codierungskonvention festgelegt, die mittels der Zeichenfunktionen für den Code etabliert wird.

Analog zum Vorgehen im letzten Abschnitt soll auch hier ein Ausblick auf den Zusammenhang mit der Softwareentwicklung in Form einer Frage gegeben werden:

- Wenn wir die im Softwareentwicklungsprozeß erstellten Modelle als Inhalts- und Ausdruckssysteme auffassen, werden sie dann denotativ oder konnotativ korreliert?

3.2 Semiotik und der Softwareentwicklungsprozeß

Bevor Aussagen einer Semiotik der Signifikation auf den Softwareentwicklungsprozeß übertragen werden, ist die Frage zu klären, ob den Beziehungen Anwendungsbereich –fachliches Modell, fachliches – softwaretechnisches Modell, fachliches Modell – Technologie und softwaretechnisches Modell – Technologie ein Signifikationssystem zugrunde liegt. Um diese Frage zu klären, wird ein Ausschnitt aus der in der Einleitung zitierten Definition des Begriffs Signifikationssystem betrachtet:

> "Ein Signifikationssystem liegt dann vor, wenn eine sozial konventionalisierte Möglichkeit zur Erzeugung von Zeichen-Funktionen gegeben ist." ([Eco76], S. 23)

Eine Zeichenfunktion etabliert eine Korrelation zwischen Ausdrucks- und Inhaltsebenen, wobei die Ausdrucksebene das übermittelnde System und die Inhaltsebene das übermittelte System repräsentieren. Betrachtet man die Beziehungen zwischen den bei der Entwicklung eines Softwaresystems erstellten Modellen und ihren beeinflussenden Kontexten, so können in Bezug auf Ausdrucks- und Inhaltssysteme nachfolgende Aussagen getroffen werden. Der

beschriebene Modellbildungsprozeß ist dabei aus Anwendungssicht zu sehen, d.h. vom Anwendungsbereich hin zum Anwendungssystem:

- das *softwaretechnische Modell* ist Ausdruckssystem des *fachlichen Modells*;
- das *softwaretechnische Modell* ist auch Ausdruckssystem der *verwendeten Technologie*;
- das *fachliche Modell* ist ein Ausdruckssystem der Teile des *Anwendungsbereichs*, die durch das Anwendungssystem unterstützt werden sollen;
- das *fachliche Modell* ist auch ein Ausdruckssystem des verwendeten *Leitbildes* und der *Entwurfsmetaphern*.

Die sozial konventionalisierte Möglichkeit zur Erzeugung von Zeichenfunktionen ist beispielsweise durch die folgenden Faktoren gegeben: (a) eine gemeinsame Sprache, (b) gemeinsames Arbeiten und Kommunizieren in einem Projekt, (c) ein gemeinsames Verständnis der am Entwicklungsprozeß beteiligten Gruppen über den Anwendungsbereich oder (d) das beim Softwareentwickler vorhandene Wissen über die zu verwendende Technologie. Dies schließt insbesondere die eingesetzte Entwicklungsmethode mit ein. Die Aussagen führen zu folgendem Ergebnis:

Ergebnis 3.3

Die Beziehungen, die während der Konstruktion eines Anwendungssystems zwischen den erstellten Modellen, dem Anwendungsbereich und der verwendeten Technologie etabliert werden (vgl. Abb. 2.8), beruhen auf einem Signifikationssystem. Dabei ist das softwaretechnische Modell Ausdruck der Inhalte »fachliches Modell« und »verwendete Technologie«. Das fachliche Modell ist Ausdruck des Anwendungsbereichs sowie des verwendeten Leitbildes und der entsprechenden Entwurfsmetaphern.

3.2.1 Konnotative und denotative Semiotiken in der Softwareentwicklung

Insgesamt lassen sich im Softwareentwicklungsprozeß drei relevante Semiotiken identifizieren. Ob die erstellten Modelle und die beeinflussenden Kontexte dabei in Form einer denotativen oder konnotativen Semiotik in Beziehung zueinander gesetzt werden, wird durch die Art der Codeetablierung entschieden. Denn der Unterschied zwischen Denotation und Konnotation beruht lediglich auf einer Codierungskonvention (vgl. Ergebnis 3.2). Es muß daher für jede Semiotik argumentiert werden, ob sie denotativen oder konnotativen Charakter hat. Für konnotative Semiotiken wurde die Darstellung von Hjelmslev (vgl. Abb. 3.7) gespiegelt, da in dieser Arbeit die Darstellung des Softwareentwicklungsprozesses vom Anwendungsbereich hin zum softwaretechnischen Modell von links nach rechts, und nicht von rechts nach links beschrieben wird. Die Spiegelung hat zum Ergebnis, daß zuerst der Inhalt und dann der für diesen Inhalt stehende Ausdruck dargestellt wird.

Die Abbildungen 3.9, 3.10 und 3.11 präsentieren die für den Softwareentwicklungsprozeß relevanten Kontexte und Modelle in Form von denotativen und konnotativen Semiotiken. Die grau hinterlegte Zeile stellt die Ausdrucks- und Inhaltssysteme bezogen auf das Prozeßmodell

dar (vgl. Abb. 2.8). Die darunterliegende Zeile zeigt an Beispielen, welche Elemente der Ausdruckssysteme mit Elementen der Inhaltssysteme korreliert werden.

Die Beziehungen, die zwischen dem Anwendungsbereich, dem fachlichen und dem softwaretechnischen Modell bestehen, werden in Form einer konnotativen Semiotik beschrieben. Ich habe mich auf Grund des folgenden Arguments, das aus Sicht eines Softwareentwicklers zu sehen ist, für eine konnotative Semiotik entschieden: als fachlicher Inhalt einer Klasse wird die Bedeutung des entsprechenden Begriffs im fachlichen Modell gesehen, nicht der anwendungsfachliche Inhalt, der zu einer Klasse im Anwendungsbereich existiert. Des weiteren ist sowohl das fachliche als auch das softwaretechnische Modell als Ausdruck des Inhalts Anwendungssystem zu sehen. Die Klasse denotiert somit einen Begriff im fachlichen Modell und konnotiert einen anwendungsfachlichen Inhalt im Anwendungsbereich.

Inhalt	**Ausdruck**	
	Inhalt	**Ausdruck**
ein Anwendungs-bereich	*ein fachliches Modell*	*ein software-technisches Modell*
Girokonto	Begriff Girokonto	Klasse `GiroKonto`

Abb. 3.9: Anwendungsbereich, fachliches und softwaretechnisches Modell als konnotative Semiotik

Betrachtet man nicht den Anwendungsbereich als Ausgangspunkt (Inhalt) für das softwaretechnische Modell, sondern die verwendete Technologie, so ergibt sich das in Abb. 3.10 dargestellte Bild. Die eingesetzte Technologie ist dabei Inhalt des softwaretechnischen Modells. Da nur jeweils ein Ausdrucks- und ein Inhaltssystem korreliert werden, handelt es sich um eine denotative Semiotik.

Inhalt	**Ausdruck**
verwendete Technologie	*ein softwaretechnisches Modell*
Ikone einer graphischen Benutzungsschnittstelle	Operation `gibIkone` an der Klasse `GiroKonto`

Abb. 3.10: Verwendete Technologie und softwaretechnisches Modell als denotative Semiotik

Die verwendete Technologie wirkt sich in Form der gewählten Leitbilder und Entwurfsmetaphern auch auf das fachliche und softwaretechnische Modell aus. Analog zu der Argumentation, die bei der konnotativen Semiotik »Anwendungsbereich, fachliches und softwaretechnische Modell« geführt wurde, ergibt sich hier eine weitere konnotative Semiotik, die in Abb. 3.11 dargestellt ist.

Inhalt	**Ausdruck**	
	Inhalt	**Ausdruck**
gewähltes Leitbild Entwurfsmetaphern	*ein fachliches Modell*	*ein softwaretechnisches Modell*
das Element Werkzeug der WAM-Metapher	das Werkzeug Chartdarsteller	die Klassen: `IAKChartdarsteller` `FKChartdarsteller`

Abb. 3.11: Leitbilder und Entwurfsmetaphern, fachliches Modell sowie softwaretechnisches Modell als konnotative Semiotik

Die dargestellten Schemata präsentieren den Softwareentwicklungsprozeß teilweise in idealisierter Form. Sie weisen die folgenden Defizite auf, die allerdings keine Auswirkungen auf die Codes und die Strukturen der beeinflussenden Kontexte sowie der erstellten Modelle im Softwareentwicklungsprozeß haben:

- Erfahrungen zeigen, daß innerhalb der Softwareentwicklung bei der Konstruktion des softwaretechnischen Modells sehr oft direkt auf Informationen des Anwendungsbereichs zurückgegriffen wird, ohne sie vorher im fachlichen Modell zu modellieren. Dies bedeutet, daß das softwaretechnische Modell sich als geschachtelte Ausdrucksebene der Inhaltsebene fachliches Modell auch direkt auf die Inhaltsebene Anwendungsbereich bezieht. Dieser Tatbestand müßte als denotative Semiotik dargestellt werden.
- Das softwaretechnische Modell wird durch das Design- und Implementierungsmodell konkretisiert. An die Stelle des softwaretechnischen Modells hätte eine weitere konnotative Semiotik mit dem Designmodell als Inhaltsebene und dem Implementierungsmodell als Ausdrucksebene treten können.
- Die Inhaltssysteme Anwendungsbereich und Technologie werden auch noch mit anderen als den in den Abbildungen aufgeführten Ausdruckssystemen korreliert, d.h. es existieren weitere Zeichenfunktionen, die Elemente des Ausdruckssystems mit Elementen eines anderen Inhaltssystems verbinden. Beispielsweise ist ein relationales Datenbanksystem Ausdruck des Inhalts «Modell einer relationalen Datenbank». Dieses Modell der relationalen Datenbank ist Bestandteil des Technologiekontextes.

Die Überlegungen zu konnotativen und denotativen Semiotiken im Softwareentwicklungsprozeß sollen in einem Ergebnis festgehalten werden:

Ergebnis 3.4

Die während des Softwareentwicklungsprozesses konstruierten Modelle mit ihren beeinflussenden Kontexten können in Form von denotativen und konnotativen Semiotiken dargestellt werden. Dabei werden die Modelle und Kontexte gemäß Ergebnis 3.3 als Ausdrucks- und Inhaltssysteme interpretiert.

3.2.2 Etablierte Codes im Softwareentwicklungsprozeß

Die drei Darstellungen (vgl. Abb. 3.9, 3.10 und 3.11), die den Softwareentwicklungsprozeß als denotative bzw. konnotative Semiotik auffassen, enthalten insgesamt vier Codes, die jeweils ein Inhaltssystem mit einem Ausdruckssystem korrelieren (vgl. Abb. 3.12).

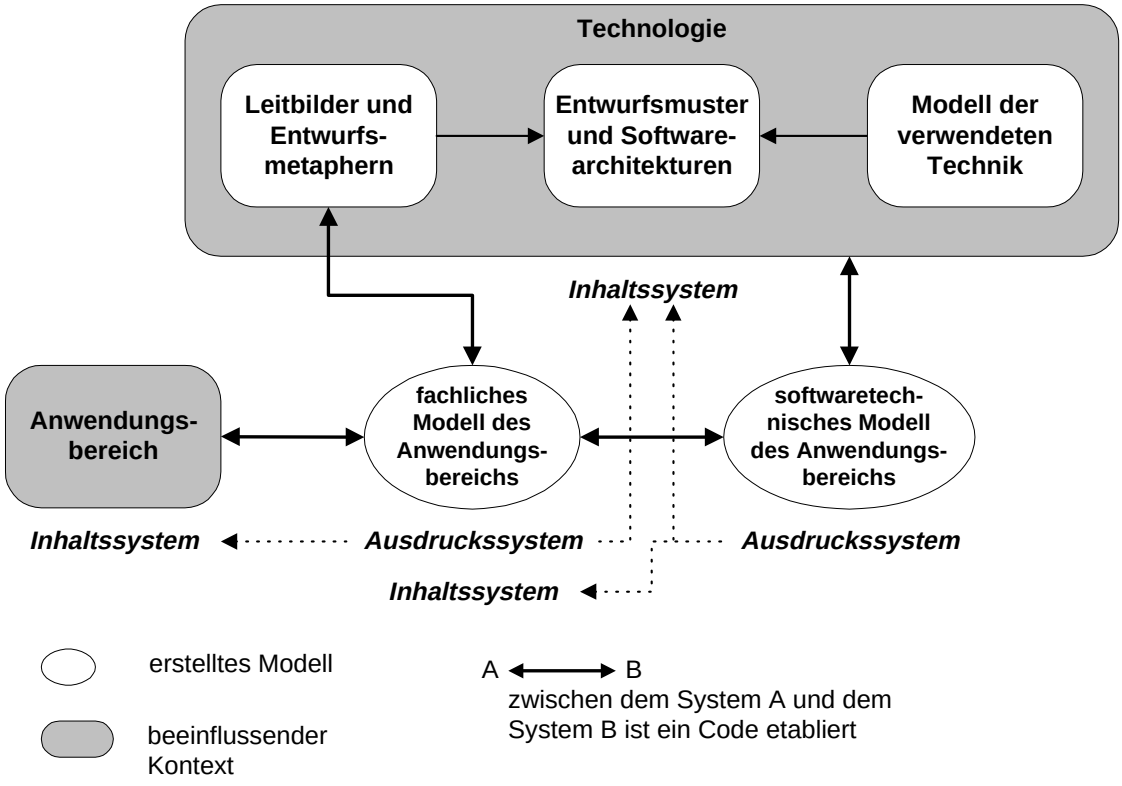


Abb. 3.12: Die etablierten Codes im Softwareentwicklungsprozeß

Die etablierten Codes sehen dabei wie folgt aus:

(I) ein Code, der die Elemente des Ausdruckssystems »softwaretechnisches Modell« mit Elementen des Inhaltssystems »fachliches Modell« verknüpft (vgl. Abb. 3.9). Die Korrelation zwischen «Girokonto$_F$» und der Klasse /`GiroKonto/` ist ein Beispiel hierfür.

(II) ein Code, der die Elemente des Ausdruckssystems »softwaretechnisches Modell« mit Elementen des Inhaltssystems »verwendete Technologie« verknüpft (vgl. Abb. 3.10). Ein Beispiel hierfür ist die Korrelation der Klasse /`GiroKonto/` mit dem «Strategie-Entwurfsmuster».

(III) ein Code, der Elemente des Inhaltssystems »Anwendungsbereich« mit Elementen des Ausdruckssystems »fachliches Modell« korreliert, das wiederum als Inhaltssystem durch den Code (I) mit dem »softwaretechnischen Modell« verbunden ist (vgl. Abb. 3.9). Es wird demnach eine konnotative Semiotik etabliert. Wenn in der weiteren Arbeit die Tatsache, daß das Ausdruckssystem des Inhaltssystems »Anwendungsbereich« wiederum eine Semiotik darstellt, nicht von Bedeutung ist, werde ich der Einfachheit halber von einem Code zwischen »Anwendungsbereich« und »fachlichem Modell«

reden. Ein Beispiel für diesen Code wäre die Korrelation von /Girokonto$_F$/ mit der entsprechenden anwendungsfachlichen Konvention «Girokonto$_A$».

(IV) ein weiterer Code, der die Elemente des Inhaltssystems »Leitbilder/Entwurfsmetaphern« mit Elementen des Ausdruckssystems »fachliches Modell« verknüpft, das wiederum mit dem softwaretechnischen Modell verbunden ist (vgl. Abb. 3.11). Die Korrelation von /Girokonto$_F$/ zu «Material» ist ein Beispiel hierfür. Analog zum Code (III) werde ich, wenn es möglich ist, der Einfachheit halber auch von einem Code zwischen »Leitbilder/Entwurfsmetaphern« und »fachlichem Modell« reden.

Die vier Codes werden bei der Konstruktion eines Softwaresystems durch einen entsprechenden Kommunikations- und Lernprozeß, der zwischen den am Softwareentwicklungsprozeß beteiligten Gruppen stattfindet, etabliert (vgl. [Flo94]). Dieser Prozeß der Codeetablierung ist somit Gegenstand einer Semiotik der Kommunikation. Es lassen sich demnach auch Aussagen aus diesem Zweig der Semiotik auf den Softwareentwicklungsprozeß übertragen. Da diese Thematik zu Beginn des Kapitels explizit ausgeschlossen wurde, wird sie hier jedoch nicht weiter vertieft.

Der Anwendungsbereich und die verwendete Technologie stellen keine elementaren Inhaltssysteme dar, sondern sind selbst wieder komplexe Gebilde, die aus einer Reihe von Ausdrucks- und Inhaltssystemen bestehen. Denn die im Softwareentwicklungsprozeß verwendeten Begriffe aus dem Anwendungsbereich oder der verwendeten Technologie werden erklärt durch entsprechende Interpretanten, die wiederum Bestandteil des Anwendungsbereichs oder der verwendeten Technologie sind.

Dies soll an einem Beispiel aus dem Anwendungsbereich erläutert werden (vgl. Abb. 3.13): /Girokonto$_F$/ des fachlichen Modells wird erklärt durch die anwendungsfachliche Einheit «Girokonto$_A$» im Anwendungsbereich. Diese wird, falls noch keine hinreichende Klarheit über /Girokonto$_A$/ im Anwendungsbereich besteht, durch eine weitere anwendungsfachliche Einheit, beispielsweise «Kontokorrent-Konto$_A$», erklärt. Das Inhaltssystem »Anwendungsbereich« ist somit zum Ausdruckssystem für ein weiteres Inhaltssystem geworden.

<table>
<tr><td></td><td></td><td>Ausdruck</td><td rowspan="2">Inhalt</td></tr>
<tr><td></td><td>Ausdruck</td><td rowspan="2">Inhalt</td></tr>
<tr><td>Ausdruck</td><td>Inhalt</td></tr>
<tr><td>fachliches Modell</td><td colspan="3">Anwendungs-bereich</td></tr>
<tr><td></td><td>/Girokonto$_A$/</td><td>/Kontokorrent-Konto$_A$/
«Kontokorrent-Konto$_A$»</td><td>…</td></tr>
<tr><td>/Girokonto$_F$/</td><td colspan="3">«Girokonto$_A$»</td></tr>
</table>

Abb. 3.13: Ausdruck und Inhalt im Anwendungsbereich

Das softwaretechnische Modell ist Ausdruckssystem sowohl für die verwendete Technologie als auch für das fachliche Modell. Betrachtet man das Ausdruckssystem als Menge von Signifikanten, so wird ein Teil dieser Signifikanten mit Signifikaten des Inhaltssystems »fachliches Modell« und ein anderer Teil mit Signifikaten des Inhaltssystems »verwendete Technologie« korreliert. Eine Reihe von Signifikanten ist mit beiden Inhaltssystemen verbunden (vgl. Abb. 3.14).

Inhalt	**Ausdruck**	**Inhalt**
fachliches Modell	softwaretechnisches Modell	verwendete Technologie
Begriff Girokonto	Klasse `GiroKonto`	Desktop-Metapher
Umgangsform einzahlen	Operation `einzahlen`	
	Operation `gibIkone`	Ikonen

Abb. 3.14: Die Inhaltssysteme des softwaretechnischen Modells

Analog werden die Signifikanten des fachlichen Modells sowohl mit dem Inhaltssystem »Anwendungsbereich« als auch mit dem Inhaltssystem »Leitbilder und Entwurfsmetaphern« korreliert. Die bisher ausgearbeiteten Überlegungen zur Thematik der Codes im Zusammenhang mit den erstellten Modellen und beeinflussenden Kontexten im Softwareentwicklungsprozeß sollen in einem Ergebnis festgehalten werden:

Ergebnis 3.5

Zwischen den erstellten Modellen und den verwendeten Kontexten werden insgesamt vier relevante Codes etabliert: zwischen Anwendungsbereich und fachlichem Modell, zwischen fachlichem Modell und softwaretechnischem Modell, zwischen fachlichem Modell und Leitbildern/Entwurfsmetaphern sowie zwischen verwendeter Technologie und softwaretechnischem Modell. Elemente des fachlichen und des softwaretechnischen Modells treten als Signifikanten in mehrere Korrelationen ein.

Es wäre denkbar gewesen, das ausführbare System als Ausdruck des Inhalts »softwaretechnisches Modell« aufzufassen und somit in die Abbildungen 3.9, 3.10 und 3.11 zu integrieren. Eine derartige Integration ist allerdings nicht sinnvoll, da zwischen dem softwaretechnischen Modell und dem ausführbaren System während des Softwareentwicklungsprozesses kein Code etabliert wird. Dies hat die folgenden Gründe: (a) der Compiler, der das softwaretechnische Modell in Objektkode übersetzt, verfügt nicht über einen Code, da er keine Zeichenfunktion versteht[12]. Der Code, der zwischen den Elementen der Programmiersprache und entsprechenden Elementen der Maschinensprache etabliert ist, wurde durch den

[12] Eco zeigt in [Eco76], S. 69, daß Apparate prinzipiell nicht über einen Code im semiotischen Sinn verfügen können.

Softwareentwickler, der den Compiler entwickelt hat, konstruiert; (b) die strukturelle Zuordnung der Klassen zu Klassenbibliotheken, Rahmenwerken oder Anwendungssystemen ist bereits im softwaretechnischen Modell beschrieben. Die Umsetzung in statische oder dynamische Bibliotheken (vgl. Abschnitt 5.5) erfolgt auch hier maschinell durch Werkzeuge, etwa durch das Programm `make`.

Es existieren, ähnlich dem Beispiel des Compilers, weitere Codes im Softwareentwicklungsprozeß, die nicht während der Konstruktion eines Softwaresystems etabliert, sondern durch entsprechende Werkzeuge oder Methoden zur Verfügung gestellt werden. So sind die Modellelemente des fachlichen Modells, etwa Begriff und Umgangsform, über einen Code mit den Modellelementen des softwaretechnischen Modells, wie Klasse und Operation, verbunden. Dieser Code wird durch die verwendete Entwicklungsmethode vorgegeben. Werden die Modellelemente des Begriffsmodells sowie die Modellelemente des Objektmodells als System betrachtet, so stellen diese Systeme einen S-Code dar, wobei das erste Inhalt des zweiten und das zweite Ausdruck des ersten ist. In Abb. 3.15 sind der Code, das Ausdrucks- sowie das Inhaltssystems einer objektorientierten Methode und eines C++-Compilers dargestellt.

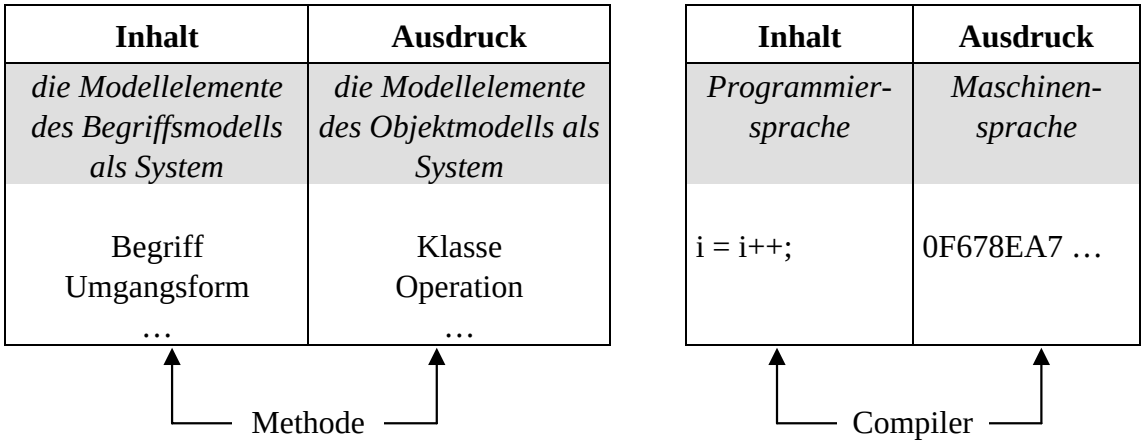


Abb. 3.15: Codes, Ausdrucks- und Inhaltssystem von Teilen einer objektorientierten Methode und eines C++-Compilers

Die Etablierung eines Codes zur Korrelation eines Ausdrucks- und eines Inhaltssystems kann durch den homologen Aufbau der Systeme vereinfacht werden (vgl. Ergebnis 3.1). Es muß demnach Ziel sein, die Struktur des fachlichen und softwaretechnischen Modells so zu gestalten, daß die Etablierung von Codes erleichtert wird, um so den Softwareentwicklungsprozeß zu vereinfachen. Im nachfolgenden Abschnitt wird daher das Konzept der Struktur genauer untersucht.

3.3 Der Begriff Struktur

In den letzen Abschnitten wurde sehr häufig argumentiert, daß die Struktur eines Systems ein wichtiges Merkmal darstellt. Dies gilt vor allem unter dem Gesichtspunkt, eine Situation verstehbar zu „gestalten“ und um einen Code zwischen einem Ausdrucks- und einem Inhaltssystem zu etablieren. Um darzulegen, wie die Struktur des Ausdrucks- und Inhaltssystems die Festlegung eines Codes vereinfachen kann, soll zuerst anhand der Aussagen über das

Staubeckenmodell und an Beispielen aus dem Bereich der Informatik aufgezeigt werden, welche Eigenschaften mit dem Begriff der Struktur verbunden sind. Die folgenden charakteristischen Merkmale lassen sich identifizieren:

(I) Struktur beschreibt den inneren Aufbau, das Gefüge eines Systems. Die Struktur bezieht sich damit auf die Ordnung, die in einem System herrscht. Im Staubeckenmodell trifft dieser Sachverhalt auf alle drei Systeme zu. In der Informatik weisen beispielsweise Datenstrukturen, die als eine Aufbau- oder Anordnungsvorschrift zu verstehen sind, diesen Aspekt auf. In heutigen Programmiersprachen werden Datenstrukturen gekapselt, d.h. der interne Aufbau der Datenstuktur ist vor dem Benutzer einer Datenstruktur verborgen. Abstrakte Datentypen (vgl. [AHU83]) in der imperativen Programmierung oder Klassen in der objektorientierten Programmierung (vgl. [Lis88], [Weg87] und [Mey88]) repräsentieren Sprachkonstrukte, mit denen sich Datenstrukturen kapseln lassen.

(II) In Programmiersprachen ist mit der Definition einer Klasse oder eines abstrakten Datentyps häufig die Deklaration eines Typs verbunden. Der Typ kann dabei als Modell oder Muster gesehen werden, das das Verhalten und die interne Struktur von Objekten dieses Typs beschreibt. Struktur wird in diesem Sinne als Operationsvorschrift verwendet, um Dingen, etwa dem fachlichen und softwaretechnischen Modell oder Objekten in einem objektorientierten Anwendungssystem, die gleiche Struktur zu geben; man standardisiert die Struktur von Dingen unter einem einheitlichen Gesichtspunkt. Unter diesem Aspekt verwendet ein Softwareentwickler auch Entwurfsmuster, Softwarearchitekturen und Entwurfsmetaphern, wenn er mit ihrer Hilfe ein Softwaresystem entwirft und strukturiert. Im Staubeckenmodell ist eine symmetrische Struktur für das (b) und (c) System verwendet worden. Dies hat die beiden Systeme einheitlich organisiert, was wiederum die Etablierung eines Codes zwischen den Systemen erleichtert hat.

(III) Strukturen werden aber auch verwendet, um in Systemen uns vertraute Eigenschaften zu identifizieren. Dazu vergleichen wir entweder die Strukturen eines Systems mit uns bekannten Strukturen anderer Systeme oder wir versuchen, uns bekannte Strukturen in dem zu analysierenden System wiederzufinden. Bei beiden Vorgehensweisen korrelieren wir die Struktur des zu untersuchenden Systems mit uns geläufigen Strukturen. Diese Korrelation fällt um so leichter, je übereinstimmender die beiden Strukturen sind. Beispielsweise identifiziert ein Softwareentwickler, wenn er einen Kodereview in einem Softwaresystem durchführt, gewisse Entwurfsmuster, deren Bedeutung er entweder aus eigener Erfahrung oder durch Studium eines Entwurfsmusterkatalogs (vgl. [GOF95]) kennt. Dieser Prozeß ermöglicht es ihm, ein Softwaresystem leichter zu verstehen.

Die Eigenschaft (I) existiert dabei unabhängig von der unter (II) und (III) beschriebenen Verwendung einer Struktur.

In der bisherigen Diskussion über Struktur blieb die Frage unberücksichtigt, ob Struktur im Gegenstand existiert oder erst durch eine Methode oder einen bestimmten Blickwinkel darin gesehen wird. Beispielsweise könnte man sich die Frage stellen, ob der Anwendungsbereich wirklich objektorientierte Strukturen aufweist, oder ob wir als Softwareentwickler nur dazu neigen, objektorientierte Strukturen darin zu suchen oder hinein zu projizieren, weil wir den Anwendungsbereich unter dem Blickwinkel einer objektorientierten Methode analysieren. Die Unterscheidung zwischen Gegenstand und Methode in Bezug auf Struktur wird in dieser

Arbeit nicht weiter ausdifferenziert, da die unter (I) bis (III) aufgeführten Charakteristika einer Struktur davon losgelöst sind.

Eine Struktur, wie sie unter (II) und (III) beschrieben ist, existiert dabei entweder nur als reine Erfahrung im „Kopf“ eines Softwareentwicklers, oder es wurde durch einen entsprechenden Abstraktionsprozeß ihre nominale Essenz herausgearbeitet. Die Struktur ist somit auf den Begriff gebracht worden. Sie erfüllt daher auch den Zweck, verschiedene Dinge auf eine homogene Art und Weise benennen zu können. Entwurfsmuster, Softwarearchitekturen und Entwurfsmetaphern sind typische Beispiele für auf den Begriff gebrachte Strukturen. Auf der Grundlage der geführten Diskussion wird der Begriff der Struktur nun wie folgt gefaßt:

Begriff 3.10 Struktur

Eine Struktur beschreibt den inneren Aufbau, die Gliederung eines Systems. Sie kann verwendet werden, um (a) unterschiedliche Systeme unter einem einheitlichen Gesichtspunkt zu standardisieren, indem die Struktur auf die Systeme übertragen wird und um (b) uns vertraute Eigenschaften in Systemen zu identifizieren. In beiden Verwendungszusammenhängen kann eine Struktur als Modell aufgefaßt werden, das von gewissen Details des beschriebenen Systems abstrahiert und so Komplexität reduziert.

Die Granularität der Strukturen, die in einem Softwaresystem identifiziert werden können, ist dabei sehr unterschiedlich. Sie reicht von einer Struktur, die als Entwurfsmuster das Zusammenspiel zweier Klassen und ihrer Objekte beschreibt, bis zu einer Struktur, die als Client-Server Architektur den prinzipiellen Aufbau des Softwaresystems darstellt. Für den Kontext dieser Arbeit sollen daher die beiden Begriffe Mikro- und Makrostruktur eingeführt werden:

Begriff 3.11 Mikrostruktur

Eine Mikrostruktur ist eine Struktur, die unter Verwendung der Modellelemente des Objektmodells der verwendeten Programmiersprache beschrieben wird. Beispiele hierfür sind Klassendiagramme, Interaktionsdiagramme und Entwurfsmuster.

Begriff 3.12 Makrostruktur

Eine Makrostruktur ist eine Struktur, die mit Hilfe der Modellelemente einer Softwarearchitektur dargestellt wird.

Die Etablierung eines Codes zwischen einem Ausdrucks- und einem Inhaltssystem wird durch den strukturellen Aufbau der Systeme vereinfacht (vgl. Ergebnis 3.1). Nachfolgend soll nun die Diskussion über homologe Strukturen wieder auf den objektorientierten Softwareentwicklungsprozeß übertragen werden. Besteht der Anspruch, daß sich die Codes zwischen den Ausdrucks- und Inhaltssystemen des Softwareentwicklungsprozesses (vgl. Ergebnis 3.4 und 3.5) einfach etablieren lassen, so können für die Struktur des fachlichen und softwaretechnischen Modells folgende Aussagen formuliert werden:

- Das fachliche Modell sollte homolog zum Anwendungsbereich sein. Da es zudem über einen Code mit dem Inhaltssystem »Leitbilder und Entwurfsmetaphern« verbunden ist,

sollte auch zwischen dem Ausdruckssystem »fachliches Modell« und dem Inhaltssystem »Leitbild und Entwurfsmethaphern« eine strukturelle Übereinstimmung existieren.

- Das softwaretechnische Modell sollte in den Teilen, die sich auf das Inhaltssystem »verwendete Technologie« beziehen, homolog zu diesem sein. Die Teile, die mit dem Inhaltssystem »fachliches Modell« korreliert sind, sollten homolog zum fachlichen Modell sein. Stehen Teile des softwaretechnischen Modells sowohl zum Inhaltssystem »verwendete Technologie« als auch zum Inhaltssystem »fachliches Modell« in Bezug, so sollten auch die Teile des softwaretechnischen Modells homolog zu beiden Inhaltssystemen sein.

Das fachliche Modell verbindet, da es sowohl Inhaltssystem für Teile des softwaretechnischen Modells als auch Ausdruckssystem des Anwendungsbereiches ist, den Anwendungsbereich mit Teilen des softwaretechnischen Modells. Diese Teile stimmen in ihrer Struktur demnach mit den Strukturen des Anwendungsbereiches überein.

Übereinstimmende Strukturen zwischen den beeinflussenden Kontexten und den erstellten Modellen erleichtern nicht nur die erstmalige Etablierung eines Codes bei der Konstruktion eines Anwendungssystems, sondern auch die Reetablierung eines Codes, wenn sich ein neuer Mitarbeiter in ein bestehendes Anwendungssystem einarbeiten muß.

Die Forderung nach homologen Strukturen kann bei der Konstruktion eines Anwendungssystems nicht immer erfüllt werden. Es entstehen dann zwischen den erstellten Modellen und den beeinflussenden Kontexten Strukturbrüche, deren typische Formen nachfolgend diskutiert werden:

(I) *Strukturbrüche, die zwischen Anwendungsbereich und fachlichen bzw. softwaretechnischen Modell existieren*. Diese Form des Strukturbruches entsteht, wenn die Struktur, die im Anwendungsbereich existiert, nicht zur Struktur paßt, mit der die verwendete Technologie das fachliche und softwaretechnische Modell organisiert. Ein typisches Beispiel hierfür ist die Modellierung eines Anwendungssystems zur Unterstützung qualifizierter menschlicher Tätigkeiten unter Verwendung von Methoden, die eine strukturierte Analyse und einen strukturierten Entwurf propagieren. Denn bei dieser Art der Modellierung sind die Konzepte des Anwendungsbereiches nur unzureichend mit den während des Entwicklungsprozesses erstellten Daten- und Funktionsmodellen verbunden (vgl. [BGZ95]).

(II) *Strukturbrüche, die zwischen unterschiedlichen Technologien, die zur Konstruktion des Softwaresystems verwendet werden, existieren*. Einzeln betrachtet können verschiedene Technologien unterschiedliche Strukturen im softwaretechnischen Modell induzieren. Sollen die verschiedenen Technologien dann in einem gemeinsamen softwaretechnischen Modell zusammengeführt werden, entsteht ein Strukturbruch. Ein typisches Beispiel hierfür ist der Anschluß einer relationalen Datenbank an ein Anwendungssystem, das unter Verwendung einer objektorientierten Methode entwickelt wird.

(III) *Strukturbrüche, die auf Grund von technischen Randbedingungen entstehen*. Diese werden im Entwicklungsprozeß bewußt in Kauf genommen, da ansonsten die Entwicklung eines aufgabenangemessenen Anwendungssystems praktisch nicht möglich ist. So wird bei einer Datenbank das logische Datenmodell, das an den Konzepten des

Anwendungsbereichs orientiert ist, typischerweise vom physikalischen Datenmodell getrennt. Dies ermöglicht es, das physikalische Modell zwecks Optimierung anders als das logische Datenmodell zu strukturieren und somit ein Anwendungssystem zu entwickeln, das dem von den Anwendern geforderten Antwortverhalten entspricht.

(IV) *Strukturbrüche, die dadurch entstehen, daß in einem einzigen Anwendungssystem unterschiedliche Aufgaben unterstützt werden sollen, die mit den zu modellierenden Gegenständen jeweils einen prinzipiell anderen Umgang verbinden.* Ein Beispiel hierfür ist die Integration von beratungs- und controllingorientierten Aufgaben in einem Anwendungssystem. Bei beratungsorientierten Aufgaben steht der Kunde im Vordergrund. Das Anwendungssystem muß einen Mitarbeiter des Unternehmens daher bestmöglich unterstützen, dem Kunden geeignete Produkte zu verkaufen. Bei controllingorientierten Aufgaben liegt der Fokus auf einer Menge von Kunden oder Produkten, deren Daten für statistische Aussagen, zum Beispiel über die Rentabilität eines bestimmten Produktes, verdichtet werden.

Strukturbrüche der Art (I) treten im Zusammenhang mit einer objektorientierten Methode und den im Kontext dieser Arbeit diskutierten Anwendungsbereichen nicht auf. Denn die Modellierungsmittel für das fachliche und softwaretechnische Modell können die Struktur des Anwendungsbereichs homolog abbilden. Diese Strukturbrüche werden daher nicht weiter beleuchtet.

Mit den diskutierten Strukturbrüchen der Form (II) und (III) wird bei der Konstruktion eines Anwendungssystems i.d.R. folgendermaßen verfahren: ein beeinflussender Kontext dominiert die Struktur des softwaretechnischen Modells. Die anderen Strukturen werden dann durch entsprechende Transformationsalgorithmen im softwaretechnischen Modell auf die dominierende Struktur abgebildet.

Strukturbrüche der Art (IV) entstehen, wenn zur Unterstützung unterschiedlicher Aufgaben verschiedene Technologien, die sich in ihren Konzepten voneinander unterscheiden (z.B.: Objektorientierung/OLAP[13]), benötigt werden. Es ist heute gängige Praxis, die Anwendungssysteme so voneinander zu trennen (vgl. [BKK+96]), daß lediglich eine Kommunikationsschnittstelle zum Austausch von Informationen vorhanden ist. Dies führt aber zu einem geringen Integrationsgrad der Anwendungssysteme. Da der Fokus dieser Arbeit auf der Beschreibung einer Gesamtarchitektur für objektorientierte Anwendungssysteme liegt, werde ich auch diese Form des Strukturbruches nicht weiter untersuchen. Ich möchte aber darauf hinweisen, daß in diesem Bereich weiterer Forschungsbedarf besteht. Die Überlegungen zu homologen Strukturen sollen in einem Ergebnis zusammengefaßt werden:

Ergebnis 3.6

Die Etablierung eines Codes wird durch die strukturelle Übereinstimmung von Ausdrucks- und Inhaltssystemen erleichtert. Deshalb sollten der Anwendungsbereich, das fachliche und softwaretechnische Modell sowie die verwendete Technologie homolog strukturiert sein.

[13] OLAP bedeutet Online Analytical Processing (vgl. [Inm96]).

Eine Softwarearchitektur kann als Operationsvorschrift aufgefaßt werden, die die strukturelle Organisation des softwaretechnischen Modells beschreibt. Damit diese Struktur sowohl homolog zur Struktur der verwendeten Technologie als auch (über die Struktur des fachlichen Modells) homolog zur Struktur des Anwendungsbereiches ist, darf eine Softwarearchitektur nicht nur auf der Basis von Technologie definiert werden, sondern muß auch die Struktur und die Gegebenheiten des Anwendungsbereichs berücksichtigen. Da die Struktur des Anwendungsbereichs homolog auf das fachliche Modell zu übertragen ist, kann eine Softwarearchitektur somit auch zur strukturellen Organisation des fachlichen Modells eingesetzt werden. Im Softwareentwicklungsprozeß wird demnach ein weiterer Code etabliert (vgl. Abschnitt 3.2.2). Dieser korreliert die Elemente des Inhaltssystems »Softwarearchitektur« mit den Elementen des Ausdruckssystems »fachliches Modell«. Diese Aussage führt zu dem folgenden Ergebnis:

Ergebnis 3.7

Eine Softwarearchitektur sollte immer in direktem Bezug zur Technologie und zu einer Klasse von strukturell übereinstimmenden Anwendungsbereichen stehen. Sie kann daher als Operationsvorschrift für die strukturelle Organisation des fachlichen und softwaretechnischen Modells verstanden werden.

Von vielen Autoren wird die bruchlose Modellierung vom Anwendungsbereich hin zum Anwendungssystem bei der Verwendung einer objektorientierten Methode hervorgehoben. Diese bruchlose Modellierung läßt sich nun mit Hilfe der in diesem Kapitel eingeführten Konzepte von Code, Ausdrucks- und Inhaltssystem erklären und somit weiter fundieren. Eine objektorientierte Methode stellt für die Modellelemente des fachlichen und softwaretechnischen Modells je ein System zur Verfügung. Dieses besteht für das fachliche Modell aus den Elementen Begriff, Umgangsform, Generalisierung/Spezialisierung, Komposition/Beziehung und Begriffshierarchie und für das softwaretechnische Modell aus den Elementen Klasse, Objekt, Operation, Vererbung, Aggregation/Assoziation und Klassenhierarchie. Die Elemente dieser beiden Systeme sind über eine Code miteinander korreliert. Zudem ist das System für die Modellelemente des fachlichen Modells in seiner Struktur übereinstimmend mit den Modellelementen, die den Anwendungsbereich beschreiben (vgl. Abb. 2.9 und Abb. 3.15). Diese Eigenschaften ermöglichen es, ein konkretes fachliches und softwaretechnisches Modell homolog zum Anwendungsbereich zu gestalten, was die Etablierung von Codes zwischen dem Anwendungsbereich als Inhaltssystem und dem fachlichen und softwaretechnischen Modell als Ausdruckssystemen erleichtert. Diese Überlegungen sollen abschließend in einem Ergebnis festgehalten werden:

Ergebnis 3.8

Die bruchlose Modellierung setzt voraus, daß eine einfache Etablierung von Codes zwischen Modellen bzw. zwischen Kontexten und Modellen im Softwareentwicklungsprozeß möglich ist.

3.4 Zusammenfassung und Ausblick

Ich habe in diesem Kapitel gezeigt, daß Aussagen aus der Wissenschaftsdisziplin der Semiotik Erkenntnisse aus der Softwareentwicklung weiter fundieren können. Dazu wurden zunächst in

Abschnitt 3.1 die Grundlagen der Semiotik anhand eines Staubeckenmodells eingeführt. An diesem Modell wurden die zentralen Begriffe Code und S-Code vorgestellt. Ein Code wurde definiert als eine Regel, die die Elemente eines Systems (S-Code) mit denen eines oder mehrerer anderer Systeme korreliert. Betrachtet man dabei das eine als ein übermittelndes System und das andere als ein übermitteltes System, so etabliert ein Code die Korrelation eines Ausdruckssystems mit einem Inhaltssystem. Die Etablierung eines solchen Codes kann dabei durch einen geeigneten strukturellen Aufbau der beiden miteinander zu korrelierenden Systeme vereinfacht werden. Daran anschließend habe ich den Unterschied zwischen einer denotativen und einer konnotativen Semiotik vorgestellt.

Aufbauend auf diesen Grundlagen wurde gezeigt, daß dem in Kapitel 2 dargestellten objektorientierten Softwareentwicklungsprozeß ein Signifikationssystem zugrunde liegt. Dies ermöglicht es, den Prozeß als konnotative und denotative Semiotiken zu beschreiben. In Bezug auf die während des Softwareentwicklungsprozesses etablierten Codes zwischen einem Ausdrucks- und einem Inhaltssystem konnten folgende Ergebnisse erarbeitet werden (vgl. Abschnitt 3.2.2 und 3.3):

- das Ausdruckssystem »softwaretechnisches Modell« wird durch einen Code mit dem Inhaltssystem »verwendete Technologie« korreliert;
- das Ausdruckssystem »softwaretechnisches Modell« wird durch einen weiteren Code mit dem Inhaltssystem »fachliches Modell« verbunden;
- das Ausdruckssystem »fachliches Modell« wird durch einen Code mit dem Inhaltssystem »Anwendungsbereich« korreliert;
- das Ausdruckssystem »fachliches Modell« wird auch durch einen Code mit dem Inhaltssystem »Leitbild und Entwurfsmetaphern« korreliert;
- das Ausdruckssystem »fachliches Modell« wird zudem durch einen Code mit dem Inhaltssystem »Softwarearchitektur« verbunden.

Eine Integration des ausführbaren Systems als Ausdruckssystem des Inhaltssystems »softwaretechnisches Modell« in die in Abb. 3.9, 3.10 und 3.11 dargestellten Semiotiken mußte verworfen werden. Ein Compiler und ein Binder, die das softwaretechnische Modell zu einem ausführbaren System montieren, etablieren keinen Code zwischen dem softwaretechnischen Modell und dem ausführbaren System. Der geltende Code wurde durch den Softwareentwickler, der den Compiler bzw. den Binder konstruiert hat, eingeführt.

In Abschnitt 3.3 wurde dann der Begriff Struktur vor dem Hintergrund eingeführt, daß die Struktur eines Systems für die Etablierung eines Codes eine wichtige Eigenschaft darstellt. Die Merkmale einer Struktur wurden dann auch wie folgt angegeben: eine Struktur beschreibt den inneren Aufbau, das Gefüge eines Systems. Sie kann verwendet werden, um unterschiedliche Systeme unter einem einheitlichen Gesichtspunkt zu standardisieren oder um in Systemen uns vertraute Eigenschaften zu identifizieren.

Um nun die Codes während des Softwareentwicklungsprozesses leichter etablieren zu können, sollten die erstellten Modelle und ihre beeinflussenden Kontexte homolog sein. Da Softwarearchitekturen die Strukturen der erstellten Modelle beschreiben, müssen sie demnach sowohl

die Struktur des Anwendungsbereiches als auch die Struktur der verwendeten Technologie berücksichtigen. Eine Softwarearchitektur ist folglich auf der Basis von Technologie und auf der Basis einer Klasse strukturell übereinstimmender Anwendungsbereiche zu definieren. So lassen sich sowohl die im Anwendungsbereich vorhandenen Mikrostrukturen als auch die Makrostrukturen bruchlos zur verwendeten Technologie in das fachliche und softwaretechnische Modell übertragen.

Im nächsten Kapitel wird daher zunächst auf der Grundlage von Ergebnissen der Organisationstheorie eine Makrostruktur für den Anwendungsbereich und das fachliche Modell definiert. Diese Makrostruktur ist mit dem Anspruch verbunden, sowohl den Anwendungsbereich als auch das fachliche Modell in kleinere modellierbare Einheiten zu zerlegen. Diese Makrostruktur wird dann in Kapitel 5 durch einen rahmenwerkbasierten Architekturstil und einen objektorientierten Schichtenarchitekturstil komplettiert. Beide Architekturstile sollen in ihrer Kombination die Konfektionierung von komplexen Anwendungssystemen auf der Basis einer Menge von wiederverwendbaren Rahmenwerken ermöglichen.

4 Softwarearchitektur und Unternehmensorganisation

"Wenn es dem Forscher zustatten kommt zu glauben, daß er strukturale Konstanten entdeckt, die allen Sprachen gemeinsam sind (und, so können wir hinzufügen, allen Erscheinungen) umso besser für ihn, wenn diese Annahme ihm bei der Forschung hilft." ([Eco72], S. 362)

Die bereits von Belady & Lehman in [BL79] beschriebene Tendenz, für immer komplexere Anwendungsbereiche Softwaresysteme zu modellieren, erfordert den Einsatz immer neuer Techniken, um insbesondere die Entropie der Softwaresysteme zu begrenzen. Ansonsten ist mit einer stetigen Degeneration der Architektur von Softwaresystemen zu rechnen. Neben neuen Techniken, wie etwa der Objektorientierung oder der Verwendung von Rahmenwerken, wird zunehmend eine Strukturierung des Anwendungsbereichs in kleinere modellierbare Einheiten notwendig. Dies gilt insbesondere vor dem Hintergrund, daß die Konstruktion eines unternehmensweiten Modells, sei es nun in Form eines Datenmodells oder als Business Object Model, nicht realisierbar ist (vgl. Abschnitt 2.2). Entwicklungsmethoden empfehlen daher, komplexe Anwendungsbereiche in kleinere Einheiten zu strukturieren. Eine solche Einheit werde ich im weiteren als Modellierungskontext bezeichnen.

Begriff 4.1 Modellierungskontext

Ein Modellierungskontext repräsentiert auf Makroebene einen Teil des Anwendungsbereichs, der weitgehend unabhängig vom Rest des Anwendungsbereichs analysiert, modelliert und implementiert werden kann.

Shlaer & Mellor führen zu der diskutierten Problematik aus:

> "What can we do if we wish to build a project too large to be built by a single, small team? Clearly, we must partition the project into pieces small enough to be worked on by a single team to ensure that the pieces can be made to fit back together again." ([SM93a], S. 8)

Ein häufig in diesem Zusammenhang diskutierter Begriff ist der des Subsystems. Mittels Subsystemen sollen große Anwendungssysteme in kleinere Einheiten zerlegt werden. Viele objektorientierte Methoden empfehlen, Subsysteme auf der Basis eines bereits erstellten fachlichen oder softwaretechnischen Modells zu bilden. Das Hauptargument hierfür lautet folgendermaßen: ein Softwareentwickler kann einen komplexen Anwendungsbereich leicht in Subsysteme zerlegen, wenn die strukturellen Beziehungen (Benutzt-Beziehung und Vererbung) zwischen den Klassen bereits erarbeitet worden sind. Um das Klassengeflecht zu zerteilen, wendet er die beiden Modularisierungsprinzipien maximierte Kohäsion und minimierte Kopplung an. Einige Methodiker weisen nun allerdings darauf hin, daß diese Vorgehensweise nicht die oben genannte Problematik zu lösen vermag. Denn die Bildung von Subsystemen

setzt ein fachliches Modell voraus, das überhaupt erst erstellt werden kann, nachdem einmal der gesamte Anwendungsbereich so in kleinere Bereiche zerlegt worden ist, daß diese die Eigenschaften eines Modellierungskontextes aufweisen. Coad & Yourdon, bei denen ein Subsystem als „Subject“ bezeichnet wird, führen hierzu aus:

> "For projects with about 35 Class-&-Objects, the Subjects can be introduced later in the analysis effort – after the Class-&-Objects are well understood – to guide the various readers through the model. But larger projects need Subjects right away, to partition the problem domain into problem sub-domains, establish work packages, and move ahead. Here, we recommend that a team of senior analysts do a rapid first-pass identification of Class-&-Objects and Structures, and then identify an initial set of Subjects; such a rapid first pass is sometimes called a ‘blitz’." ([CY91], S. 111)

Mit der im Zitat beschriebenen Blitz-Analyse ist aber ebenso die Aufarbeitung von Aufgaben im Anwendungsbereich, die Identifizierung von Gegenständen und ein entsprechender Abstraktionsprozeß, der die Gegenstände auf den Begriff bringt, verbunden. Die Blitz-Analyse ist, wenn sie sauber durchgeführt wird, auch für große Systeme bereits sehr aufwendig. Sie führt demnach zu ähnlichen Problemen wie eine vollständige Konstruktion des fachlichen Modells.

Viele Autoren von Methodenbüchern empfehlen daher eine Zerlegung in drei oder vier Bereiche, die sich meistens wie folgt darstellen (vgl. [CD94] und [CY91a]):

- *Benutzungsschnittstelle*: In diesem Bereich werden alle Klassen zusammengefaßt, die die Benutzungsschnittstelle des Anwendungssystems realisieren.
- *fachliche Funktionalität*: Klassen dieses Bereichs modellieren die fachliche Kernfunktionalität des Softwaresystems.
- *Datenobjekte*: Dieser Bereich modelliert die Daten des Anwendungssystems. Er entsteht sehr häufig dann, wenn das Softwaresystem an eine Datenbank angeschlossen wird.
- *Infrastruktur*: Klassen dieses Bereichs stellen anwendungsneutrale Lösungen bereit. Beispiele hierfür sind Container-Klassen oder Schnittstellen zu anderen Anwendungssystemen.

Cook & Daniels empfehlen, die Modellbildung für große Softwaresysteme getrennt nach den Bereichen Benutzungsschnittstelle, fachliche Funktionalität und Infrastruktur durchzuführen:

> "Broadly speaking, domains are of three types: concept domains, interaction domains, and infrastructure domains. Concept domains model the phenomena in the problem being solved. Interaction domains model the software-implemented mechanisms by which the concept domain objects are kept up-to-date with the external environment and vice-versa. Infrastructure domains provide general-purpose abstractions which provide application-independent services to other domains." ([CD94], S. 292)

Die Einteilung des Anwendungssystems in die genannten drei bzw. vier Bereiche weist zwei Nachteile auf. Erstens ist sie nicht an den fachlichen Strukturen des Anwendungsbereichs orientiert. Dies erschwert somit die Etablierung von Codes im Softwareentwicklungsprozeß (vgl. Ergebnis 3.6). Zweitens fördert sie nicht die notwendige getrennte Modellierbarkeit der Bereiche. So wirken sich i.d.R. Änderungen an der fachlichen Funktionalität sowohl auf die Benutzungsschnittstelle als auch auf die Datenobjekte aus. Diese Problematik werde ich noch ausführlich in Abschnitt 5.4.3 diskutieren.

In der Literatur wird als Vorteil einer derartigen Dreiteilung stets die Möglichkeit aufgeführt, einen der Bereiche Benutzungsschnittstelle, Datenobjekte oder Infrastruktur komplett durch eine andere Implementierung zu ersetzen. So beschreiben Shlaer und Mellor in [SM93a] diesen Vorteil am Beispiel eines von ihnen entwickelten Subsystems für die Benutzungsschnittstelle folgendermaßen:

> "In the product manufacturing plant example, we have characterized the user interface as iconic, with Screens, Icons, Scroll Bars, and the like. We could easily substitute for this domain another user interface domain that uses only a command-line-interface. […]. The entire [Anm. des Autors: graphical] user interface domain has been substituted with another." ([SM93a], S. 22)

Derartige Änderungen an einem Softwaresystem kommen allerdings in der Praxis relativ selten vor. So entstehen laut Meyer (vgl. [Mey88]) nur 6,2% der Wartungskosten eines Softwaresystems durch Änderungen an der Hardware, worunter auch der Wechsel von einer graphischen zu einer textorientierten Benutzungsschnittstelle fällt. Hingegen verursachen neue Benutzeranforderungen 41,8% der Wartungskosten. Der oben genannte Vorteil ist demnach in der Praxis von untergeordneter Bedeutung.

Die mit der vorgestellten Dreiteilung verbundene Trennung der Zuständigkeiten in die Bereiche Benutzungsschnittstelle und fachliche Funktionalität soll hier nicht kritisiert werden. Sie stellt eines der Grundprinzipien des Model-View-Controller Konzeptes dar, dessen Vorteile in der Literatur ausführlich dargelegt sind (vgl. [Cun85] und [KP88]). Wie eine detaillierte Diskussion in Abschnitt 5.4.3 jedoch zeigen wird, ist die Unterteilung eines Anwendungssystems in die genannten Bereiche nicht geeignet, voneinander unabhängige Modellierungskontexte zu etablieren.

Die bisher diskutierte Problematik der Zerlegung eines Anwendungsbereichs ist demnach mit einem inhärenten Problem verbunden, das Aksit & Bergmans wie folgt beschreiben:

> "The dilemma here is that if the software engineer does not identify subsystems before starting with object identification, then the project probably becomes unmanageable. On the other hand, if the software engineer identifies subsystems prior to object-identification, then the defined subsystems boundaries may not be optimal." ([AB92], S. 346)

Der mit der Objektorientierung verbundene Anspruch, den Anwendungsbereich vom fachlichen Modell bis hin zum softwaretechnischen Modell bruchlos zu modellieren, wird von

vielen Autoren als der entscheidende Vorteil gegenüber herkömmlichen Analyse- und Designmethoden gesehen. Die bruchlose Modellierung wird ermöglicht, da sich die Struktur des Anwendungsbereichs einfach auf das fachliche und softwaretechnische Modell übertragen läßt (vgl. Abschnitt 3.3). Das Grundprinzip, die Struktur des fachlichen und softwaretechnischen Modells auf der Basis der Struktur des Anwendungsbereichs zu gestalten, könnte auch zur Lösung des von Aksit & Bergmans beschriebenen Dilemmas angewendet werden. Es müßten allerdings die folgenden Punkte erfüllt werden:

(1) Für eine bestimmte Klasse von Anwendungsbereichen müssen sich unternehmensorganisatorische Makrostrukturen identifizieren lassen, die es dem Informatiker erlauben, den Anwendungsbereich in Modellierungskontexte zu strukturieren. Dies ermöglicht eine Zerlegung des Anwendungsbereichs zu Beginn des Entwicklungsprozesses.

(2) Die identifizierten Makrostrukturen müssen sich bruchlos auf das fachliche und softwaretechnische Modell übertragen lassen. Dies führt zu homologen Strukturen, die die Etablierung von Codes im Entwicklungsprozeß vereinfachen (vgl. Abschnitt 3.3).

(3) Die Prinzipien, nach denen ein Unternehmen aufbauorganisatorisch strukturiert ist, sollten zu Unternehmenseinheiten führen, die weitgehend autonom voneinander agieren können. Die aufbauorganisatorischen Prinzipien ließen sich dann mit bewährten technischen Modularisierungsprinzipien (minimierte Kopplung, maximierte Kohäsion) zusammenführen. Denn nur so entsteht im fachlichen und softwaretechnischen Modell auch eine unter softwaretechnischen Gesichtspunkten geeignete Struktur. Wie die Diskussion in Abschnitt 4.1.2 zeigen wird, gilt dies sowohl für herkömmliche, strukturorientierte Prinzipien als auch für solche, die der Prozeßorientierung (z.B. dem Business Process Reengineering) folgen.

Neben diesen drei Kriterien sollte die erarbeitete Makrostruktur auch die nachfolgenden Anforderungen erfüllen (vgl. Abschnitt 2.7):

(4) Die entstandenen Modellierungskontexte sollen sich mittels einer Softwarearchitektur, die speziell die Eigenschaften der Objektorientierung berücksichtigt, realisieren lassen.

(5) Um unternehmensweit integrierte Anwendungssysteme entwickeln zu können, muß die fachliche und softwaretechnische Kombinierbarkeit und Integrierbarkeit der Komponenten über die einzelnen Modellierungskontexte hinweg gegeben sein. Denn nur so ist es möglich, spezielle Anwendungssysteme aus Komponenten unterschiedlicher Modellierungskontexte zu konfektionieren.

(6) Eine evolutionäre Vorgehensweise muß bei der Entwicklung der einzelnen Anwendungssysteme unterstützt werden.

In Kapitel 2 wurde der Begriff des Anwendungsbereichs definiert als eine Organisation oder ein Bereich einer Organisation, von dem ein fachliches und softwaretechnisches Modell erstellt wird (vgl. Begriff 2.1). Anwendungsfachliche Makrostrukturen sind demnach in einer Organisation oder in Bereichen der Organisation zu suchen. In Abschnitt 4.1 diskutiere ich daher Prinzipien der Einheitenbildung in Unternehmen und beschreibe die dabei entstehenden Makrostrukturen. Der Abschnitt 4.2 stellt der unternehmensorganisatorischen Struktur eine rein technologisch motivierte Struktur gegenüber. Dies ist notwendig, da sowohl der Anwendungsbereich als auch die verwendete Technologie die Struktur des softwaretechnischen

Modells prägen muß. In Abschnitt 4.3 führe ich dann die unternehmensorganisatorisch und die technologisch motivierte Struktur in einem gemeinsamen Modell zusammen. Die daraus resultierende Makrostruktur wird anschließend in Abschnitt 4.4 auf das fachliche Modell übertragen.

4.1 Makrostrukturen eines Unternehmens

Softwarearchitekturen beschreiben die Makrostrukturen eines Softwaresystems. Wenn diese auf der Basis von anwendungsfachlichen Makrostrukturen etabliert werden sollen, dann ist der Fokus bei der Untersuchung von Strukturen im Anwendungsbereich auf die Unternehmensstruktur zu legen. Sie stellt ebenfalls eine Makrostruktur dar und weist eine ausreichend hohe Geltungsdauer (i.d.R. mehr als 10 Jahre) auf (vgl. [Sta90]). Im Mittelpunkt dieses Kapitels steht die Organisationsstruktur als Ergebnis, nicht der Prozeß, der zur Gestaltung der Organisationsstruktur geführt hat. Denn auch Softwarearchitekturen beschreiben primär die statischen Aspekte eines Softwaresystems, nicht dessen dynamisches Verhalten zur Laufzeit.

Im folgenden werden Ergebnisse aus der Organisationstheorie vorgestellt, um sie für einen später zu definierenden Architekturstil nutzbar zu machen. Abschnitt 4.1.1 untersucht als erstes verschiedene Prinzipien, nach denen sich Einheiten in einem Unternehmen bilden lassen. Darauf aufbauend werden in 4.1.2 verschiedene Organisationsmodelle diskutiert. Als offene soziotechnische Systeme stehen Unternehmen in starker Verbindung zu ihrer ökonomischen und sozialen Umwelt, die ebenfalls die Struktur eines Unternehmens beeinflußt. Abschnitt 4.1.3 setzt sich daher mit verschiedenen Umweltfaktoren auseinander. Die vorgestellte Ausarbeitung beruht weitgehend auf der Literatur [Ble91], [Eng95], [Fre92], [Gai83], [Sta90] und [Ste95].

4.1.1 Prinzipien der Einheitenbildung in Unternehmen

In der Literatur (vgl. [Ble91], [Eng95] und [Fre92]) werden drei verschiedene Prinzipien der Einheitenbildung unterschieden: das Verrichtungs-, das Objekt- und das Prozeßprinzip. Diese sind wie folgt charakterisiert:

- *Verrichtungsprinzip*: Beim Verrichtungsprinzip werden Aufgabenkomplexe entlang der gleichartigen Durchführung von Aufgaben gebildet. An einer so gebildeten Stelle werden Verrichtungen an verschiedenen Arbeitsobjekten (Stühle) ähnlichen Typs (z.B. Stuhl mit Lehne und ohne Lehne) durchgeführt. Dies entspricht dem Leitbild der industriellen Fertigung. Wird dieses Leitbild etwa auf die Erstellung eines Stuhles angewandt, so bearbeitet ein Arbeiter immer das Rohmaterial (für verschiedene Stühle), ein anderer schleift die gefertigten Stuhlteile, ein dritter baut den Stuhl zusammen und ein vierter lackiert den Stuhl.
- *Objektprinzip*: Beim Objektprinzip werden Aufgabenkomplexe durch die Zusammenfassung von Verrichtungen, die ein Arbeiter an gleichen Arbeitsobjekten vornimmt, gebildet. Daraus resultiert, daß an einer Stelle unterschiedliche Verrichtungen an einem Arbeitsobjekt durchgeführt werden. Arbeitsobjekte müssen dabei nicht physisch vorhanden sein. Es können auch ideelle Arbeitsobjekte, wie etwa ein Konto, das keinen Referenten aufweist (vgl. Begriff 3.1), als solche aufgefaßt werden. Beim Objektprinzip

bearbeitet demnach ein Mitarbeiter in einer Schreinerei einen Stuhl vom Rohmaterial bis hin zur Lackierung. Der Stuhl wird also aus einer Hand gefertigt. Diese Art der Zusammenfassung von Aufgaben folgt dem Leitbild der handwerklichen Fertigung.

- *Prozeßprinzip*: Dieses durch neue Ansätze im Bereich der Management- und Organisationstheorie insbesondere durch den Ansatz des Business Process Reengineering (vgl. [Eng95]) in den Vordergrund gerückte Prinzip propagiert die Bildung von Aufgabenkomplexen entlang von Prozessen. Ein Prozeß ist dabei definiert als "eine spezifisch zeitlich und räumlich angeordnete Menge von Aktivitäten, die zueinander in einer Leistungsbeziehung stehen" ([Eng95], S. 44). Das Business Process Reengineering (kurz BPR) empfiehlt sogar die ganzheitliche und radikale Neugestaltung von Unternehmensprozessen in Abhängigkeit von Umweltanforderungen, was zu einer veränderten Aufbau- und Ablauforganisation führt. Im Gegensatz zum Verrichtungs- und Objektprinzip basiert das Prozeßprinzip nicht auf vorgegebenen, statischen Strukturierungsmerkmalen. Eine Organisationsform, die auf dem Prozeßprinzip beruht, ist daher häufig unternehmensspezifisch. Zudem führt eine Prozeßorganisation, die primär an den Verrichtungen bzw. Objekten orientiert ist, zu vergleichbaren Ergebnissen wie die eigentliche Anwendung des Verrichtungs- oder Objektprinzips. Diese Eigenschaft des Prozeßprinzips werde ich nochmals in Abschnitt 4.3.1 aufgreifen. Eine typische Organisation, wie sie für das Verrichtungs- und Objektprinzip am Beispiel des Stuhls vorgestellt wurde, läßt sich daher für das Prozeßprinzip nicht angegeben. Das nachfolgende Beispiel stellt somit nur eine mögliche Zerlegung der Fertigung eines Stuhls in „Miniprozesse“ dar: in einem ersten Miniprozeß könnte das Rohmaterial bestellt und bearbeitet, in einem zweiten die Rückenlehne sowie der Korpus geschliffen und zusammengebaut und in einem dritten Miniprozeß die Rückenlehne und der Korpus lackiert, die Sitzfläche gepolstert und der Stuhl fertiggestellt werden.

4.1.2 Organisationsmodelle

Auf der Basis der unterschiedlichen Prinzipien, die zur Einheitenbildung in Unternehmen eingesetzt werden können, lassen sich in der Organisationstheorie unterschiedliche Modelle diskutieren. Prinzipielles Ziel der Unternehmensorganisation ist dabei die "Differenzierung des Systems Unternehmung in arbeitsteilige Subsysteme und deren Integration zu einem zielorientierten handelnden Ganzen" ([Fre92], S. 210).

Die verschiedenen Organisationsmodelle lassen sich in ein- und mehrdimensionale Modelle unterteilen. Eindimensionale Modelle berücksichtigen bei der Strukturbildung nur ein Prinzip der Einheitenbildung, während mehrdimensionale Modelle Unternehmen auf der Basis mehrerer Prinzipien organisieren. Die eindimensionalen Modelle untergliedern sich demnach in verrichtungs-, objekt- und prozeßorientierte Organisationsstrukturen. Diese sind wie folgt charakterisiert (vgl. [Eng95] und [Sta90]):

- *Funktionalorganisation (Verrichtungsprinzip)*: Bei der Funktionalorganisation werden die Unternehmenseinheiten nach dem Verrichtungsprinzip gebildet. Typischerweise werden dabei Abteilungen nach den betrieblichen Funktionen, wie etwa Beschaffung, Produktion oder Vertrieb organisiert. Die Strukturierung innerhalb einer so gebildeten Einheit (Abteilung) erfolgt dann wiederum nach einem der drei oben beschriebenen Prinzipien. Abb. 4.1 präsentiert eine nach dem Verrichtungsprinzip gestaltete

Organisation, deren zweite Ebene nach dem Objektprinzip (entlang von Produkten) organisiert ist.

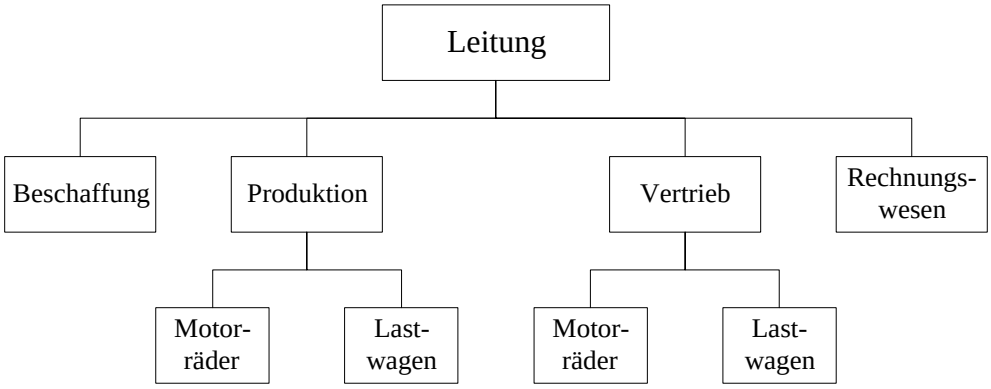


Abb. 4.1: Eine verrichtungsorientierte Organisation (vgl. [Sta90])

- *Geschäftsbereichs- oder Spartenorganisation (Objektprinzip)*: Bei diesem Modell wird die Struktur nach dem Objektprinzip gebildet. Da Arbeitsobjekte als Strukturierungsmerkmal zu feingranular sind, erfolgt die Gruppierung entlang von Produkten bzw. einheitlichen Produktgruppen (vergleichbar mit den Arbeitsobjekten), Regionen oder Kunden bzw. Kundengruppen. Wird die Strukturierung entlang von Produkten vorgenommen, so spricht man von einer Produktorganisation, erfolgt sie entlang von Regionen von einer Regionalorganisation. Gleichgültig, welches Kriterium zur Bereichsbildung herangezogen wird, Ziel ist es immer, Abteilungen zu bilden, die erstens nur geringe Interdependenzen zu anderen Abteilungen besitzen und zweitens einen hohen inneren Zusammenhalt (Gruppenkohäsion) aufweisen. Denn so kann gewährleistet werden, daß sie ihr Abteilungsziel weitgehend autonom erreichen können (vgl. [Eng95] und [Sta90]).

  Bei Produktorganisationen werden Arbeitsobjekte entlang von Produkten bzw. Produktgruppen zusammengefaßt. Die darunterliegende Ebene wird wiederum nach einem der drei beschriebenen Prinzipien organisiert (vgl. Abb. 4.2).

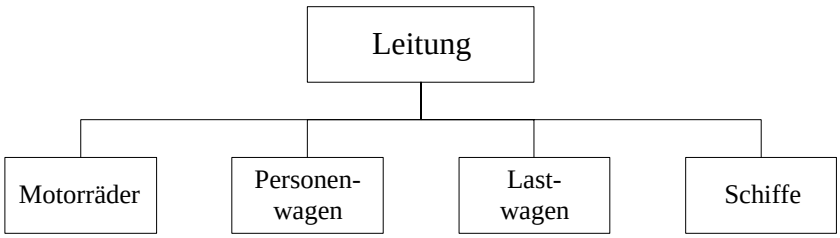


Abb. 4.2: Eine produktorientierte Organisation

- *Prozeßorganisation*: Bei der Prozeßorganisation werden die Unternehmenseinheiten nach dem Prozeßprinzip gebildet. Dadurch entstehen Abteilungen, in deren Mittelpunkt einzelne Unternehmensprozesse stehen. Bea & Hass schreiben zur Grundidee der Prozeßorganisation:

> "Die Grundidee der Prozeßorganisation besteht darin, daß Prozesse Gegenstand der Strukturierung von Unternehmen sein sollen. Es werden somit organisatorische Einheiten mit Prozeßverantwortung geschaffen." ([BH97], S. 402)

Einzelne Abteilungen sollten dabei so gestaltet sein, daß zwischen ihnen auf der Ebene der Prozesse nur geringe wechselseitige Abhängigkeiten existieren (vgl. [Eng95]).

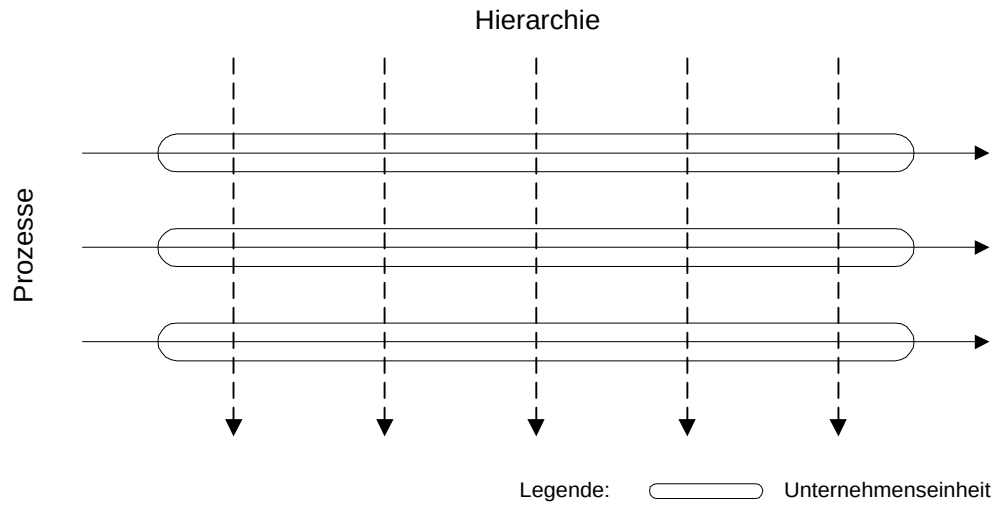


Abb. 4.3: Eine Prozeßorganisation ([BH97], S. 402)

Mit den vorgestellten eindimensionalen Organisationsmodellen ist auch immer eine Einfachunterstellung von Mitarbeitern verbunden, d.h. sie sind immer genau einer Abteilung zugeordnet. Die eindimensionalen Organisationsmodelle etablieren demnach auch immer eine Linienorganisation im Unternehmen.

Eindimensionale Organisationsmodelle sind darauf ausgerichtet, eindimensionale Strukturen zu unterstützen. Reale Probleme zeichnen sich allerdings durch Mehrdimensionalität aus. Dies erfordert mehrdimensionale Organisationsmodelle, die sich in Matrix- und Tensororganisationen untergliedern. Sie sind wie folgt charakterisiert:

- *Matrixorganisation*: Bei einer Matrixorganisation werden zwei der vorgestellten Gliederungsprinzipien Verrichtung, Objekt (Produkt, Region oder Kunde) und Prozeß gleichzeitig angewendet (vgl. Abb. 4.4). Sehr häufig anzutreffen sind die Überlagerung der Gliederungsprinzipien Verrichtung und Produkt (vgl. [Fre92]), Prozeß und Objekt sowie Objekt und Verrichtung (vgl. [Eng95]).

  Matrixorganisationen werden auch häufig bei der Durchführung von Projekten bzw. bei der Einführung von Produkten am Markt eingesetzt. Man spricht dann von Projekt- bzw. Produktmanagement.

- *Tensororganisation*: Für weltweit tätige Großunternehmen ist es wichtig, neben Produkt- und Verrichtungsmerkmalen auch länderspezifische Gesichtspunkte zu berücksichtigen. Dies bedeutet, daß die Matrixorganisation um eine weitere Dimension »Region« erweitert werden muß.

In der Organisationstheorie werden in jüngster Zeit vermehrt Team- bzw. Netzwerkstrukturen diskutiert. Sie sind im Gegensatz zu den klassischen Organisationsmodellen mit einem geringen Strukturierungsgrad verbunden. Sie werden daher vorwiegend zur Strukturierung der Arbeitsorganisation eingesetzt. Engelmann schreibt hierzu:

> "Insofern aber häufig neben dem netzwerkartigen Leistungs- und Kommunikationszusammenhang der Arbeitsgruppen keine wesentlich tiefergehende Strukturierung beschrieben wird, gehen Netzwerkmodelle hinsichtlich ihres Aussagegehaltes über arbeitsorganisatorische Gruppenkonzepte kaum hinaus. Es erscheint jedoch bei weitem unzureichend, die strukturelle Komponente von Organisationen auf den Aspekt der Arbeitsgruppe zu reduzieren." ([Eng95], S. 88)

Da in dieser Arbeit die Unternehmens- und nicht die Arbeitsorganisation im Vordergrund steht, wird auf eine weitere Untersuchung der Team- bzw. Netzwerkmodelle verzichtet.

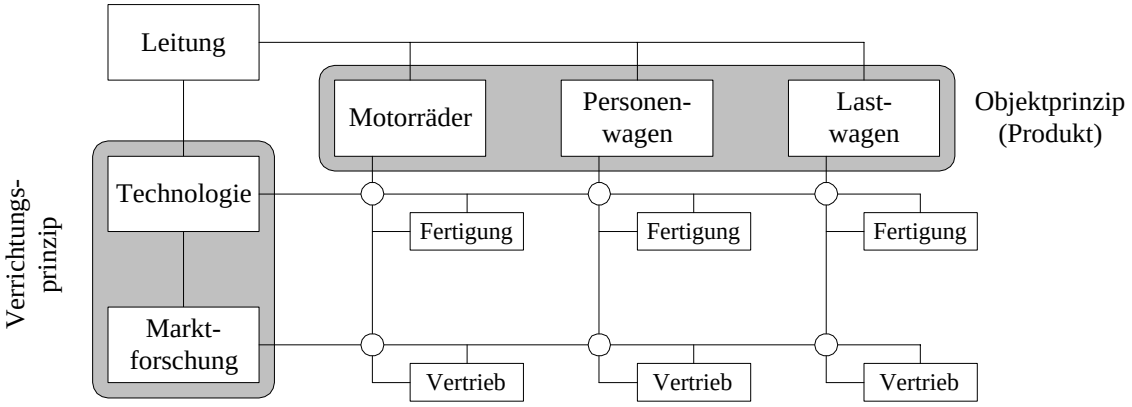


Abb. 4.4: Eine Matrix-Organisation (vgl. [Ste95], S. 187)

4.1.3 Die Umweltsituation

In der Organisationstheorie werden Unternehmen als Systeme betrachtet, für die Austauschprozesse mit der natürlichen und sozialen Umwelt überlebensnotwendig sind. In der Organisationstheorie werden deshalb Unternehmen auch als offene soziotechnische Systeme bezeichnet (vgl. [Sta90], S. 581). Demzufolge müssen Unternehmen bei der Gestaltung ihrer Organisationsstruktur die vorhandene Umweltsituation berücksichtigen. In diesem Abschnitt werden daher Umweltsituationen und sie beeinflussende Faktoren charakterisiert.

Emery & Trist (vgl. [ET65]) haben eine entsprechende Umwelttypologie erarbeitet, die die in Abb. 4.5 beschriebenen vier Umweltsituationen umfaßt. Die turbulente Umwelt (turbulent fields) wird von Emery & Triest als die zukünftig vorherrschende Umweltsituation angesehen. Sie begründen dies im wesentlichen mit dem stetigen Anwachsen von Unternehmen, den zunehmenden Interdependenzen zwischen ökonomischen und anderen Bereichen der Gesellschaft, wie etwa Gesetzgebung, Kultur und Politik, und den ständig neuen Impulsen, die aus Entwicklung und Forschung (z.B. Internet) auf die Unternehmen einwirken.

Umweltsituation	Merkmale	organisatorische Anpassung
I. placid, randomized environment	statisch, stabil, kaum Wandel, Vor- und Nachteile gleichmäßig verteilt, vollkommene Konkurrenz	keine Unterscheidung zwischen Taktik und Strategie notwendig, Organisationen können als kleine, unabhängige, anpassungsfähige Einheiten überleben
II. placid, clustered environment	statisch, Vor- und Nachteile sind ungleichmäßig verteilt, unvollkommene Konkurrenz	Strategie wird von Taktik unterschieden, Organisationen wachsen, werden hierarchisch gegliedert, zentrale Kontrolle und Koordination
III. disturbed-reactive environment	dynamisch, oligopolitisch, es bestehen mehrere gleichartige Organisationen mit gleichen Zielen, die sich untereinander bekämpfen	Strategie - Kampagne - Taktik, Dezentralisierung von Entscheidungen und Kontrolle, Notwendigkeit von Absprachen mit Konkurrenten
IV. turbulent fields	dynamisch, hohe Komplexität und Unsicherheit hinsichtlich zukünftiger Entwicklungen, hohe Interdependenz zwischen einzelnen Subumwelten induziert neue, eigengesetzliche Entwicklungen	Entwicklung von gemeinsamen Werten und Normen als informeller Kontrollmechanismus, Theory Y[14], wachsende Bedeutung von Forschung, Entwicklung und Planung, Matrix-Organisation

Abb. 4.5: Organisatorische Anpassungen an unterschiedliche Umweltsituationen (von Emery & Triest aus [Sta90], S. 435)

Die Umweltfaktoren werden in der Literatur in die ein Unternehmen unmittelbar beeinflussenden Märkte Beschaffungsmarkt, Arbeitsmarkt, Geld- und Kapitalmarkt und Absatzmarkt (Kunden) unterteilt. Neben diesen speziellen Umweltfaktoren existieren weitere generelle Faktoren, die in kulturelle, sozialpsychologische, rechtliche und politische, wirtschaftliche sowie technologische Faktoren unterschieden werden (vgl. Abb. 4.6).

Die in den letzten drei Abschnitten vorgestellten Ergebnisse aus der Organisationstheorie sollen nun in Bezug zu dieser Arbeit gesetzt werden. Es muß daher gezeigt werden, welche Makrostrukturen eines Unternehmens als Ausgangsbasis für die Softwarearchitektur von objektorientierten Anwendungssystemen geeignet sind.

[14] Die Theorie Y wurde von McGregor (vgl. [McG60]) als ein Pol einer dualistischen Sicht von Menschenbildern geprägt. Das Menschenbild der Theorie Y definiert McGregor wie folgt: "Der Mensch hat keine angeborene Abneigung gegen Arbeit, im Gegenteil, Arbeit kann eine wichtige Quelle der Zufriedenheit sein" ([Sta90], S. 173]). Nach der Theorie X hat "der Mensch […] eine angeborene Abscheu vor der Arbeit und versucht, sie so weit wie möglich zu vermeiden" ([Sta90], S. 173]).

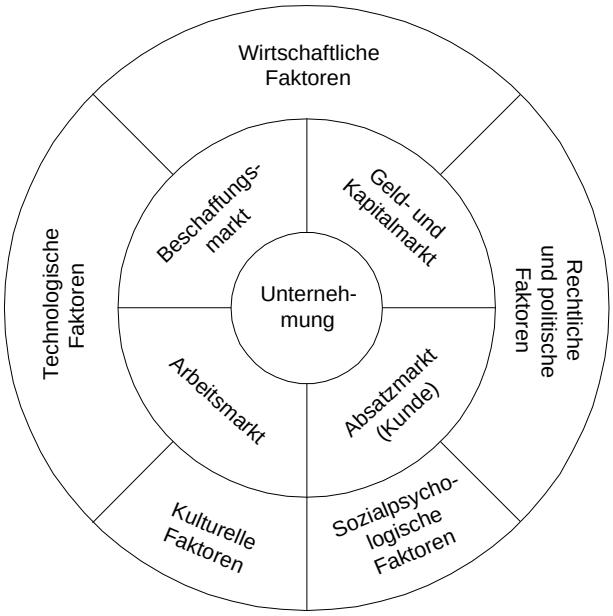


Abb. 4.6: Umweltsysteme der Unternehmung (aus [Sta90], S. 582)

Da die turbulente Umwelt als zukünftig vorherrschende Umweltsituation angenommen wird, müssen Anwendungssysteme so konstruiert werden, daß sich diese bei Veränderung der direkten Umweltfaktoren, insbesondere beim Absatz- und Beschaffungsmarkt, entsprechend einfach an die neue Situation anpassen lassen. Diese Tatsache muß ebenfalls von einer Softwarearchitektur berücksichtigt werden.

In Kapitel 2 wurde herausgearbeitet, daß die Struktur des softwaretechnischen Modells von der Struktur des Anwendungsbereichs und der verwendeten Technologie geprägt wird. Es ist daher notwendig, neben einer auf der Organisation des Anwendungsbereichs beruhenden Struktur auch eine technologisch motivierte Struktur des softwaretechnischen Modells als Grundlage für Softwarearchitekturen zu betrachten. Eine solche Struktur wird im nächsten Abschnitt diskutiert.

4.2 Eine technologisch motivierte Makrostruktur

Objektorientierte Softwareentwicklung fokussiert auf die Rekonstruktion der Begriffe des Anwendungsbereichs. Durch die Analyse der relevanten Aufgaben und Tätigkeiten des Anwendungsbereichs rekonstruiert ein Softwareentwickler die vorhandenen und verwendeten Gegenstände und ihre Umgangsformen. Mittels Abstraktion werden dann ähnliche Gegenstände auf den Begriff gebracht, die in einem nächsten Schritt in Kompositions- und/oder Generalisierungsbeziehungen zueinander gesetzt werden. Dieses Begriffsgeflecht ist Ausgangspunkt für das softwaretechnische Modell, das um die für die Konstruktion notwendigen technischen Charakteristika ergänzt ist. Müßte ein Softwareentwickler das so entstandene Klassengeflecht in kleinere, voneinander unabhängige Subsysteme zerlegen, so würde er eine Partitionierung entlang der softwaretechnischen Modularisierungsprinzipien maximierte

Kohäsion und minimierte Kopplung vornehmen. Die beiden Kriterien sind mit Blick auf die Objektorientierung wie folgt definiert worden (vgl. [YC79] und [PB93]):

- *Maximierte Kohäsion*: Maximierung der Bindungsstärke innerhalb einer Klasse.
- *Minimierte Kopplung*: Minimierung der Bindung zwischen Klassen. Dies ist vor allem für Klassen, die in unterschiedlichen Bereichen angesiedelt sind, von besonderer Bedeutung.

Die Anwendung dieser Kriterien führt zu Subsystemen, die weitgehend in sich geschlossen sind. Dies soll an einem vereinfachten Beispiel aus dem Bankgeschäft verdeutlicht werden. Betrachtet werden die folgenden Aufgaben: Erfassen der Sicherheit KFZ-Übereignung, Auszahlung eines Kredits, Wertpapierdepot einrichten und eine Order für den Aktienkauf erfassen. In Bezug auf die Gegenstände, die bei der Erledigung der Aufgaben eine Rolle spielen, lassen sich die folgenden Aussagen formulieren:

- *Erfassen der Sicherheit KFZ-Übereignung*: diese Aufgabe betrifft die Gegenstände KFZ-Übereignung, das entsprechende Kreditprodukt (z.B. eine Baufinanzierung), das dem Kunden offeriert wurde, den Kunden als Kreditnehmer und den Nominalbetrag der Sicherheiten, die der Kreditnehmer an die Bank übereignet hat.
- *Auszahlung eines Kredits (Baufinanzierung)*: die Auszahlung eines Kredits betrifft die Gegenstände Baufinanzierung, Kreditbetrag, Kreditkonto und den Kunden als Kreditnehmer.
- *Einrichten eines Wertpapierdepots*: hier sind die Gegenstände Depot, Risikoerklärung und der Kunde als Wertpapierinhaber betroffen.
- *Erfassen einer Order für den Aktienkauf*: diese Aufgabe betrifft die Gegenstände Depot, Order, Aktie, Aktienkurs, Ankaufbetrag, das Girokonto, von dem der Kauf bezahlt wird, und den Kunden als Wertpapierinhaber.

Die nachfolgende Tabelle zeigt die Gruppierung der als Klassen modellierten Gegenstände zu Subsystemen. Ihr Aufbau ist wie folgt: die Gegenstände sind, gruppiert nach den oben beschriebenen vier Aufgaben, entlang von Zeilen und Spalten angeordnet. Das Symbol + bedeutet, daß der Gegenstand in der Zeile von dem Gegenstand in der Spalte abhängig ist. Quadrant I stellt Abhängigkeiten zwischen den beiden Aufgaben KFZ-Übereignung und Kreditauszahlung dar, Quadrant II zwischen den Aufgaben KFZ-Übereignung und Kreditauszahlung einerseits sowie den Aufgaben Einrichten eines Depots und Aktienkauf andererseits. Quadrant III ist die Umkehrung des Quadranten II, Quadrant IV verdeutlicht die Abhängigkeiten zwischen den beiden Aufgaben Einrichten eines Depots und Aktienverkauf. In der Tabelle sind nur direkte Abhängigkeiten (Assoziation und Aggregation) zwischen den Gegenständen vermerkt. Aus Platzgründen wird für die Gegenstände eine Abkürzung, die in der Legende angegeben ist, eingeführt. Die mit einer Ellipse gekennzeichneten Einträge sollen beispielhaft erläutert werden: der Kreditnehmer (KN) hängt von den Sicherheiten (SH) und die Order (OR) von den Aktien (AE) ab.

		Kreditgeschäft							Wertpapiergeschäft							
		KF	SH	NB	BF	KN	KB	KK	RE	DP	WI	OR	AE	AK	AB	GK
Kredit	KF		+	+												
	SH			+												
	NB															
	BF		+													
	KN	+	(+)		+			+								
	KB															
	KK				+		I								II	
Wertpa.	RE															
	DP												+			
	WI								+	+						+
	OR												(+)		+	
	AE													+		
	AK															
	AB															
	GK						III								IV	

KF: KFZ-Übereignung *SH*: Sicherheit *NB*: Nominalbetrag
BF: Baufinanzierung *KN*: Kreditnehmer *KB*: Kreditbetrag
KK: Kreditkonto *RE*: Risikoerklärung *DP*: Depot
WI: Wertpapierinhaber *OR*: Order *AE*: Aktie
AK: Aktienkurs *AB*: Ankaufbetrag *GK*: Girokonto

Abb. 4.7: Gruppierung von Gegenständen

Die Gruppierung der Gegenstände nach den Prinzipien Kohäsion und Kopplung hat zu den beiden autonomen Bereichen Kredit- und Wertpapiergeschäft geführt. Es wurde allerdings in der Tabelle nicht berücksichtigt, daß zwischen Begriffen, die über den Gegenständen gebildet werden, weitere Beziehungen, zum Beispiel in Form einer gemeinsamen Generalisierung, bestehen können. So stellt der Begriff Konto eine Generalisierung der Begriffe Giro- und Kreditkonto dar. Eine Diskussion dieser Problematik wird in Abschnitt 4.4 geführt.

Eine Zusammenfassung der vier Aufgaben Erfassen der Sicherheit KFZ-Übereignung, Auszahlung eines Kredits, Einrichten eines Wertpapierdepots und Erfassen einer Order für den Aktienkauf zu größeren Einheiten nach dem Modell der Geschäftsbereichs- oder Spartenorganisation (vgl. Abschnitt 4.1.2) führt ebenfalls zu den beiden Bereichen Kredit- und Wertpapiergeschäft. Ursache hierfür ist die für eine Bank typische Ausrichtung von Aufgaben an Bankprodukten (vgl. Abschnitt 4.3.1). Dieser Sachverhalt ist in Abb. 4.7 daran erkennbar, daß Beziehungen zwischen Gegenständen nur zwischen den Aufgaben Erfassen der Sicherheit KFZ-Übereignung und Auszahlung eines Kredits sowie Einrichten eines Wertpapierdepots und Erfassen einer Order für den Aktienkauf existieren. Diese Paare bilden jeweils die unabhängigen Organisationseinheiten Kredit- und Wertpapiergeschäft.

4.3 Synthese von unternehmensbezogenen und technologischen Strukturen

Bisher wurden die in einem Unternehmen etablierten Makrostrukturen und die rein technologisch motivierten Strukturen im softwaretechnischen Modell, die nach den Prinzipien der Kohäsion und Kopplung gebildet wurden, getrennt voneinander betrachtet. Um Aussagen darüber treffen zu können, welche Beziehungen zwischen den anwendungsfachlichen und den technologischen Strukturen existieren, werden diese nachfolgend einander gegenübergestellt. Weiterhin ist zu diskutieren, wie der geforderten Anpassungsfähigkeit von Anwendungssystemen an veränderte Umweltsituationen entsprochen werden kann (vgl. Abschnitt 4.1.3).

4.3.1 Der Produktbereich

Der technologisch motivierte Ansatz der Subsystembildung hat in dem dargelegten Beispiel zu den beiden Bereichen Kredit- und Wertpapiergeschäft geführt. Derartige Bereiche finden sich auch in der Aufbauorganisation von Banken wieder, wenn diese nach dem Modell der Produktorganisation gestaltet sind. Die beiden Vorgehensweisen, die zur Bildung der Makrostrukturen geführt haben, sind dabei wie folgt gekennzeichnet:

- Das softwaretechnische Modell wurde auf der Basis der Gegenstände des Anwendungsbereichs gebildet, die im Sinne der Organisationstheorie Arbeitsobjekte darstellen. Die technologisch motivierte Makrostruktur entstand entlang der Modularisierungsprinzipien maximierte Kohäsion und minimierte Kopplung, was zu Bereichen geführt hat, die nur geringe wechselseitige Abhängigkeiten aufweisen.
- Bei einem Unternehmen, das nach dem Produktprinzip organisiert ist, werden Arbeitsobjekte und die damit verbundenen Verrichtungen so zu Bereichen zusammengefaßt, daß diese weitgehend autonom sind. Demzufolge bestehen zwischen den entstandenen Bereichen nur geringe oder keine Interdependenzen, was dazu führt, daß Arbeitsobjekte eines Bereichs nur geringe oder keine Beziehungen zu Arbeitsobjekten in anderen Bereichen aufweisen. Zudem werden die Unternehmenseinheiten so gebildet, daß sie einen hohen inneren Zusammenhalt aufweisen (vgl. Abschnitt 4.1.2).

Die Übereinstimmung der technologischen und organisatorischen Makrostrukturen basiert also auf der Verwendung vergleichbarer Strukturierungsprinzipien (vgl. Abb. 4.8).

Wenn die überwiegende Anzahl von Prozessen produktbezogen verlaufen, dann weisen nach Engelmann Organisationsformen, die nach dem Prozeßprinzip gestaltet sind, eine hohe strukturelle Übereinstimmung mit Produktorganisationen auf. Er betont sogar, daß beim Business Process Reegineering die Objektgliederung wieder verstärkt in den Vordergrund tritt. Engelmann schreibt zum Verhältnis von Produkt- und Prozeßorganisation:

> "Bei der Produktorganisation werden für die Produkte bzw. Produktgruppen mit den Geschäftsbereichen eigene Organisationseinheiten gebildet. Produktbezogene Prozesse verlaufen dann ausschließlich innerhalb der abgegrenzten Geschäftsbereiche, nicht zwischen diesen. Auf der ersten Gliederungsebene sind Produktorganisationen somit gewissermaßen prozeßorientiert strukturiert." ([Eng95], S. 55)

Anwendungsbereich	softwaretechnisches Modell
Mikrostrukturen	
Gegenstand	Objekt
Umgangsform	Operation
Begriff	Klasse
Generalisierung, Spezialisierung	Vererbung
Komposition	Aggregation, Assoziation
Begriffshierarchie	Klassenhierarchie
Makrostrukturen	
Produktbereich	Subsystem
geringe Interdependenz, innerer Zusammenhalt	minimierte Kopplung, maximierte Kohäsion

Abb. 4.8: Strukturierungsprinzipien im Anwendungsbereich und im softwaretechnischen Modell

Die vorausgegangenen Überlegungen sollen unter dem Gesichtspunkt einer Synthese von unternehmensbezogenen und technologisch motivierten Makrostrukturen in einem Ergebnis festgehalten werden:

Ergebnis 4.1

Die Organisationsstruktur eines Anwendungsbereichs, die nach dem Produktprinzip gestaltet ist, kann als Grundlage für die Makrostruktur des fachlichen und softwaretechnischen Modells verwendet werden. Ist die Organisationsstruktur nach dem Prozeßprinzip gebildet worden, so kann sie nur dann als Grundlage verwendet werden, wenn die überwiegende Anzahl der Prozesse produktbezogen verlaufen. Produktorientierte und an Produkten ausgerichtete prozeßorientierte Organisationsformen unterteilen den Anwendungsbereich in Unternehmenseinheiten, die softwaretechnisch als Modellierungskontexte betrachtet werden können. Die Organisationsstruktur kann dabei ein- oder mehrdimensional ausgelegt sein. Bei mehrdimensionalen Strukturen wird lediglich die Produkt- bzw. Prozeßdimension berücksichtigt.

Ausgehend von diesem Ergebnis wird der Begriff Produktbereich wie folgt definiert:

Begriff 4.2 Produktbereich

Modellierungskontexte (vgl. Begriff 4.1), die produktorientierte oder an Produkten ausgerichtete prozeßorientierte Organisationseinheiten repräsentieren, werden im weiteren als Produktbereiche bezeichnet.

4.3.2 Der Einsatzkontext

Die Strukturierung des Anwendungsbereichs in verschiedene Produktbereiche ist unabhängig von den konkreten Anwendungssystemen zu sehen, die in einem Unternehmen zur Unterstützung ihrer Mitarbeiter eingesetzt werden. Denn die Arbeitszusammenhänge, in denen Anwendungssysteme verwendet werden, sind sehr unterschiedlich. Sie werden sowohl durch

das Geschäftsfeld, in dem ein Unternehmen tätig ist, als auch durch Umweltfaktoren geprägt (vgl. Abschnitt 4.1.3). Im Bereich der Umweltfaktoren sind vor allem der Beschaffungsmarkt und der Absatzmarkt (Kunde) zu nennen. Insbesondere im Bereich der Dienstleistungsunternehmen zeichnet sich verstärkte Kundenorientierung ab.

Neben Anwendungssystemen, die auf einen speziellen Produktbereich zugeschnitten sind, werden zunehmend Systeme benötigt, die eine Unterstützung über mehrere Produktbereiche hinweg ermöglichen. Hinzu kommen Anwendungssysteme, mit denen ein Kunde direkt, d.h. ohne die Betreuung durch einen Mitarbeiter des Unternehmens, die Dienstleistung eines Unternehmens in Anspruch nehmen kann. In diesem Kontext wird der Kunde zum Benutzer des Anwendungssystems. Durch das Internet gewinnt diese Form der Benutzung von Anwendungssystemen für Unternehmen immer mehr an Bedeutung. Im Bankenbereich sind die folgenden Einsatzkontexte für Anwendungssysteme erkennbar:

- Beraterarbeitsplätze, die sich in zwei weitere Typen gliedern:

  1. *Arbeitsplätze für Kundenberater*: an diesen Arbeitsplätzen steht die kundenorientierte Unterstützung im Kredit-, Anlage- und Wertpapiergeschäft im Vordergrund. Ziel ist es, den Kunden umfassend, aber nicht notwendigerweise tief spezialisiert zu beraten. So werden an einem solchen Arbeitsplatz beispielsweise keine Derivatgeschäfte[15] abgewickelt.
  2. *Arbeitsplätze für bankfachliche Spezialisten*: diese Arbeitsplätze bieten eine spezialisierte Unterstützung für einzelne Aspekte des Kredit-, Anlage- und Wertpapiergeschäfts. An ihnen werden spezielle Dienstleistungen für spezielle Kundengruppen erbracht.

  In beiden Bereichen steht die qualifizierte und eigenverantwortliche Wahrnehmung von Aufgaben durch den Bankmitarbeiter im Vordergrund. Ein entsprechendes Anwendungssystem muß den Bankmitarbeiter daher bestmöglich bei der Beratung eines Kunden unterstützen.

- *Arbeitsplätze im Schalterbereich bzw. spezialisierte Arbeitsplätze für das Massengeschäft*: an diesen Arbeitsplätzen steht die standardisierte und rasche Erfüllung von Dienstleistungen durch Mitarbeiter der Bank im Vordergrund. Beispiele hierfür sind der Auslandszahlungsverkehr oder die Überwachung von Überziehungslisten.
- *Homebanking und Selbstbedienungsterminals*: hier kommt es vor allem darauf an, daß Dienstleistungen von Laien im Bankgeschäft wahrgenommen werden können. Diese sind unter Umständen gleichzeitig Computerlaien.

Für jeden dieser Einsatzkontexte sind jeweils unterschiedliche Anwendungssysteme notwendig. Ein Softwaresystem an einem Kundenberaterarbeitsplatz muß einen guten Querschnitt aus den Produktbereichen Schalter-, Kredit-, Anlage- und Wertpapiergeschäft unterstützen. Demgegenüber kommen in einem speziell für den Wertpapierverkauf gestalteten System nur geringe Anteile der anderen Produktbereiche vor. Dafür ist es um zusätzliche Funktionalität aus dem Wertpapierbereich erweitert. Zwischen Einsatzkontext und Produktbereich existiert

[15] Derivatgeschäfte sind Termingeschäfte auf Aktien, Anleihen, Indizes oder Währungen.

somit eine n : m Beziehung. Die softwaretechnischen Komponenten, die für die verschiedenen Produktbereiche konstruiert werden, müssen sich demnach einfach zu entsprechenden Anwendungssystemen kombinieren lassen. Dieser Sachverhalt wird in der nachfolgenden Tabelle (vgl. Abb. 4.9) dargestellt. Die Produktbereiche werden dabei in Zeilen, die Einsatzkontexte in Spalten angeordnet. Die in der Tabelle verwendeten Symbole -, o, + und ++ tragen die folgende Bedeutung:

- -: der Produktbereich ist nicht mit dem Einsatzkontext verknüpft.
- *o*: der Produktbereich ist gering mit dem Einsatzkontext verknüpft.
- +: der Produktbereich ist mit dem Einsatzkontext verknüpft.
- ++: der Produktbereich ist stark mit dem Einsatzkontext verknüpft.

Einsatzkontext / Produktbereich	Berater-arbeitsplatz	Wertpapier-arbeitsplatz	Selbst-bedienung	...
Schalter	o	o	o	
Kredit	+	-	-	
Anlage	+	-	o	
Wertpapier	+	++	-	

Abb. 4.9: Zusammenhang zwischen Produktbereich und Einsatzkontext

Einsatzkontexte müssen, da sie sich nicht genau einem Produktbereich zuordnen lassen, losgelöst von der Struktur der Produktbereiche betrachtet werden. Die Entkopplung von Einsatzkontext und Produktbereich schafft eine gute Voraussetzung, flexibler auf die sich kontinuierlich verändernde Umweltsituation reagieren zu können. Neue Formen der Kundenbetreuung schaffen neue Einsatzkontexte. Sie haben jedoch keine Auswirkungen auf bereits bestehende Anwendungssysteme, die in anderen Einsatzkontexten verwendet werden. Solange die Produktbereiche stabil bleiben, lassen sich neue Arbeitsplatz- und Kundensysteme durch Wiederverwendung von Softwarekomponenten der vorhandenen Produktbereiche konfektionieren. Der Vertrieb neuer Produkte (z.B. Expansion im Versicherungsgeschäft) führt zu neuen Produktbereichen. Softwarekomponenten, die für diese neuen Produktbereiche entwickelt werden, können zunächst an spezialisierten Arbeitsplatzsystemen erprobt werden. Langfristig ist jedoch zu überprüfen, inwieweit die neuen Produkte auch in bereits bestehenden Einsatzkontexten vertrieben werden sollen. Dann kann eine Erweiterung bestehender Arbeitsplatzsysteme nicht vermieden werden. Dies ist jedoch softwaretechnisch unproblematisch. Die Loslösung der Produktbereiche von spezifischen Einsatzkontexten ist nur sinnvoll, wenn sich die für die Produktbereiche entwickelten Softwarekomponenten technisch und fachlich leicht zu Anwendungssystemen integrieren lassen. Aus diesem Grund sollten sich auch Softwarekomponenten aus neuen Produktbereichen einfach in bestehende Anwendungssysteme hineinkonfigurieren lassen. Den Begriff des Einsatzkontextes fasse ich abschließend wie folgt:

Begriff 4.3 Einsatzkontext

Ein Einsatzkontext definiert den konkreten Arbeitszusammenhang, in dem ein Anwendungssystem in einem Unternehmen eingesetzt wird. Dieser Arbeitszusammenhang umfaßt dabei die Menge der Aufgaben, die mit Hilfe des Anwendungssystems in dem Einsatzkontext erledigt werden. Diese kann sich aus Aufgaben unterschiedlicher Produktbereiche zusammensetzen. Die Loslösung des Einsatzkontextes vom Produktbereich ermöglicht eine flexiblere Reaktionsfähigkeit auf die Umweltfaktoren Beschaffungs- und Absatzmarkt.

4.4 Die Struktur im fachlichen Modell

Die Struktur der Produktbereiche und der Einsatzkontexte, wie wir sie in einer Organisation vorfinden, soll nun auf das fachliche Modell übertragen werden. Daher ergibt sich als erstes die folgende Problematik: mit den generellen Signifikanten /Produktbereich/, /Prozeßbereich/, /Modellierungskontext/ oder /Einsatzkontext/ sowie mit speziellen Signifikanten, wie etwa /Kreditgeschäft/ oder /Beraterarbeitsplatz/, kann sowohl eine organisatorische Einheit in einem Unternehmen als auch ein Bereich im fachlichen Modell bezeichnet werden. Wenn aus dem Kontext nicht klar hervorgeht, welche der zwei Interpretationen mit dem Bezeichner verbunden ist, werde ich eine Annotation, wie sie in Abschnitt 2.3 bereits eingeführt wurde, vornehmen. Mit $_A$ annotierte Bezeichner stehen für Bereiche aus dem Anwendungsbereich, mit $_F$ annotierte Bezeichner für Bereiche aus dem fachlichen Modell. So ist beispielsweise mit Kreditgeschäft$_A$ der Produktbereich Kredit im Anwendungsbereich gemeint.

Wird die Struktur der Produktbereiche auf das fachliche Modell übertragen, so wird dieses in getrennte Bereiche zerlegt. Das in Abb. 4.7 dargestellte Beispiel, in dem ein kleiner Ausschnitt aus dem Bankgeschäft in die beiden Produktbereiche Kredit- und Wertpapiergeschäft eingeteilt wurde, verleitet zu der folgenden Annahme: der Anwendungsbereich läßt sich so in disjunkte Produktbereiche zerlegen, daß Gegenstände immer genau einem Produktbereich zugeordnet werden können. Eine solche Struktur ist unter dem Gesichtspunkt der Softwareentwicklung, insbesondere der Aufteilung der Aufgaben auf einzelne Projekte, günstig. Dies wird von Shlaer & Mellor in [SM93] auch explizit empfohlen. Sie schreiben dazu:

> ”A domain is a separate real, hypothetical, or abstract world inhabited by a distinct set of objects that behave according to rules and policies characteristic of the domain. [...]. Each domain forms a separate and cohesive whole. In terms of the conceptual entities, or objects, required to build the system. [...]. *A conceptual entity is defined in exactly one domain.*“ ([SM93], S. 16)

Erfahrungen aus dem GEBOS-Kontext zeigen allerdings, daß eine solche disjunkte Aufteilung nicht verwirklicht werden kann. So existieren bereits in dem obigen Beispiel Überschneidungen zwischen den Bereichen Kredit- und Wertpapiergeschäft, die in der Tabelle nicht sofort erkennbar sind (vgl. Abb. 4.7). Die unterschiedlichen Arten von Überschneidungsmöglichkeiten sollen nachfolgend charakterisiert werden. Die Diskussion wird dabei am Beispiel des fachlichen Modells geführt:

(a) *gemeinsam benutzte Begriffe*: dies sind Begriffe, die in unterschiedlichen Produktbereichen gemeinsam verwendet werden. Die Gegenstände Nominalbetrag, Kreditbetrag und Ankaufbetrag lassen sich im fachlichen Modell durch den generalisierten Begriff Betrag modellieren. Dieser ist dann Bestandteil aller Produktbereiche.

(b) *gemeinsame Generalisierung*: zu Begriffen aus unterschiedlichen Produktbereichen existiert eine gemeinsame Generalisierung. Ein Beispiel hierfür ist der Begriff Konto als Generalisierung der Begriffe Kreditkonto und Girokonto.

(c) *fachliche Identität*: Gegenstände, die anscheinend getrennt voneinander in unterschiedlichen Produktbereichen existieren, können trotzdem eine gemeinsame fachliche Identität aufweisen. So ist der Kreditnehmer Hans Müller mit dem Wertpapierinhaber Hans Müller identisch. Hans Müller wird aber durch zwei verschiedene Begriffe repräsentiert.

Integrierte Anwendungssysteme eines Anwendungsbereichs müssen auf einer gemeinsamen technischen und fachlichen Basis konstruiert werden. Die fachliche Basis stellt dabei die fachliche Integration von Anwendungssystemen sicher. Dadurch wird die gemeinsame Verwendung von Gegenständen und deren fachliche Interpretation über Anwendungssysteme hinweg ermöglicht. Zudem gewährleistet eine fachliche Basis die softwaretechnische Unterstützung von Kooperation, da kooperatives Handeln nur auf der Grundlage eines gemeinsamen Verständnisses über die verwendeten Gegenstände möglich ist (vgl. [Gry96]). Fehlt diese gemeinsame fachliche Basis, so können Anwendungssysteme zwar über technische Schnittstellen miteinander verbunden werden, diese ermöglichen es aber nur, Daten zwischen den Anwendungssystemen auszutauschen. Um ein Datum (Signifikant) mit seinem Sinn (Signifikat) zu verbinden, ist die Etablierung eines gemeinsamen Codes über beide Anwendungssysteme hinweg erforderlich. Dieser ist dann notwendigerweise in beiden Systemen abzubilden, was zwangsläufig zu Mehrfachmodellierungen in den Anwendungssystemen führt.

Dies soll wieder an einem Beispiel aus dem Bankgeschäft illustriert werden: In fast allen Anwendungssystemen ist das Kundengrundbild wichtiger Bestandteil. Im Wertpapierbereich enthält dieses Grundbild die zu dem Kunden gehörenden Depots und die darin enthaltenen Aktienbestände sowie allgemeine Informationen zum Kunden, wie seine Adresse, seine Kundennummer und sein Gesamtengagement. Im Kreditbereich sind die genauen Informationen zu einem Depot uninteressant. Hier stellen die Kreditkonten und die vom Kreditnehmer an die Bank übereigneten Sicherheiten wichtige Gegenstände dar. Beiden Kundenbilder sind aber Informationen wie der Kundenname, seine Kundennummer und das Gesamtengagement aller Konten gemeinsam. Fehlt die gemeinsame fachliche Basis, so ist diese Informationen in beiden Anwendungssystemen zu modellieren.

4.4.1 Der Gegenstandsbereich

Die bisher geführte Diskussion hat gezeigt, daß die Konstruktion von fachlich integrierten Anwendungssystemen eine gemeinsame fachliche Basis voraussetzt. Ein eigener Modellierungsbereich vergegenständlicht die gemeinsame fachliche Basis. Er wird als Gegenstandsbereich bezeichnet und ist wie folgt definiert:

Begriff 4.4 Gegenstandsbereich

Der Gegenstandsbereich umfaßt die fachlichen Kernkonzepte, auf denen alle Produktbereiche basieren. Er bildet einen übergreifenden Modellierungsbereich und charakterisiert demnach das Geschäft des Unternehmens.

Der Gegenstandsbereich bildet die fachliche Basis für die unterschiedlichen Produktbereiche. Er repräsentiert daher ein für alle Produktbereiche gemeinsames Modell. Das bedeutet jedoch nicht, daß jedes Konzept im Gegenstandsbereich zwangsläufig auch in allen existierenden Produktbereichen verwendet wird. Ein Kernkonzept kann auch nur für einen Teil der Produktbereiche fachlich relevant sein. So ist beispielsweise das Konzept der Sicherheit für viele Geschäftsbereiche einer Bank derart wichtig, daß es Bestandteil des Gegenstandsbereichs sein sollte, auch wenn es für den Geschäftsbereich Schalter oder Anlagegeschäft nicht von Bedeutung ist.

Um nicht mit denselben Problemen konfrontiert zu sein, die mit der Entwicklung eines unternehmensweiten Modells verbunden sind (das sollte durch diese Arbeit ja gerade vermieden werden), müssen Eigenschaften, Vorgehensweisen und Konstruktionsprinzipien beschrieben werden, um den Gegenstandsbereich möglichst optimal gestalten zu können. Mit »optimal« ist allerdings der folgende Zielkonflikt verbunden:

- Der Gegenstandsbereich muß, in Bezug auf die Anzahl der darin modellierten Begriffe, so umfangreich sein, daß er eine gemeinsame fachliche Basis für alle zu konstruierenden Produktbereiche darstellt. Ansonsten kommt es zu einer Mehrfachmodellierung von Begriffen in unterschiedlichen Produktbereichen. Dies erschwert die Entwicklung integrierter Anwendungssysteme.
- Der Gegenstandsbereich sollte, damit er einfach zu konstruieren ist und die Abhängigkeiten zwischen den Produktbereichen und dem Gegenstandsbereich möglichst gering sind, minimal sein.

Eine wesentliche Eigenschaft des Gegenstandsbereichs ist, daß er nur im fachlichen und softwaretechnischen Modell konstruiert wird. Innerhalb einer Organisation gibt es keinen Bereich, der dem Gegenstandsbereich entspricht. Denn er umfaßt jene Konzepte, die das Kerngeschäft eines Anwendungsbereichs vergegenständlichen und auf denen alle Produktbereiche basieren.

Die Modellierung des Gegenstandsbereichs wird insbesondere dann notwendig, wenn mindestens zwei sich überlappende Produktbereiche existieren, die softwaretechnisch zu unterstützen sind. Bei der Konstruktion des ersten Produktbereichs sollten allerdings Generalisierungen, die sich auf Grund von Erfahrungswerten des Softwareentwicklers durch die Hinzunahmen eines weiteren Produktbereichs herausbilden würden, in dem Produktbereich vorausschauend herausgearbeitet werden.

Wird zu einem bestehenden Produktbereich ein zweiter hinzugefügt, so muß bei seiner Konstruktion sukzessive der Gegenstandsbereich gebildet werden. Auf diesem werden dann beide Produktbereiche weiterentwickelt. Dieser Schritt stellt einen Schlüsselfaktor für einen optimalen Ausbau eines Anwendungssystems hin zu einer Familie integrierter Anwendungssysteme dar. Wird der Schritt unterlassen und statt dessen der zweite Produktbereich auf der

Basis des bereits entwickelten konstruiert, so ist der zweite Produktbereich vom ersten abhängig. Erweiterungen, die am ersten Produktbereich vorgenommen werden, können somit zu Anpassungsaufwänden im zweiten Produktbereich führen. Die beiden Bereiche lassen sich nicht mehr getrennt voneinander weiterentwickeln. Dies sollte aber gerade durch die Zerlegung des Anwendungsbereichs in mehrere Produktbereiche vermieden werden. Diese Vorgehensweise ist allerdings in vielen kommerziellen Projekten aus Kostengründen vorherrschend.

Mit jeder weiteren Hinzunahme eines Produktbereichs ergibt sich potentiell die Notwendigkeit, den Gegenstandsbereich zu überarbeiten. Folgende Fälle lassen sich unterscheiden:

- Im ersten Fall muß ein Konzept, das bereits in einem Produktbereich modelliert wurde, in den Gegenstandsbereich überführt werden, da es von dem neuen Produktbereich ebenfalls benötigt wird.
- Im zweiten Fall muß ein Konzept, das bereits Bestandteil des Gegenstandsbereichs ist, auf Grund von Anforderungen des neuen Produktbereichs erweitert werden.
- Im dritten Fall muß ein Konzept des Gegenstandsbereichs bei Hinzunahme eines neuen Produktbereichs weiter generalisiert werden. Dies ist notwendig, wenn das Konzept im Gegenstandsbereich noch zu spezifisch durch die Sicht der existierenden Produktbereiche geprägt wird. Es stellt dann auch solche Umgangsformen bereit, die aus Sicht des neu hinzukommenden Produktbereichs fachlich irrelevant sind. Typischerweise wird der (produktbereichs-) spezifische Anteil in einem separaten Konzept ausfaktorisiert.

Der erste Fall führt zu einem nicht vermeidbarem Änderungsaufwand sowohl im Gegenstandsbereich als auch in den bereits existierenden Produktbereichen. Im dritten Fall muß nur der Gegenstandsbereich geändert werden, da ein dort modelliertes Konzept in eine Generalisierung und eine Spezialisierung zerlegt wird. Zusätzliche Änderungen an den existierenden Produktbereichen sind nur notwendig, wenn die Spezialisierung aus dem Gegenstandbereich in die betroffenen Produktbereiche ausgelagert werden soll.

Im zweiten Fall läßt sich jeglicher Änderungsaufwand vermeiden, wenn der Gegenstandsbereich im softwaretechnischen Modell so konstruiert wird, daß er sich fachlich erweitern läßt, ohne dabei selber verändert zu werden. Dieser Grundsatz entspricht dem von Meyer definierten Offen-Geschlossen-Prinzip (vgl. [Mar96a] und [Mey88]). Offenheit und Geschlossenheit sind im Sinne von Meyer wie folgt definiert:

- *Offenheit* bedeutet, daß die Module eines Systems offen für Erweiterungen der Datenstrukturen und Operationen sind.
- Unter *Geschlossenheit* versteht Meyer, daß ein Modul für die Verwendung durch andere Module bereitsteht, was bedingt, daß die Schnittstelle des Moduls im Sinne des Geheimnisprinzips wohldefiniert und stabil ist. Programmtechnisch geschlossene Module werden gewöhnlich kompiliert in eine Bibliothek eingebracht und dann für den allgemeinen Gebrauch freigegeben.

Die Forderung nach Offenheit steht normalerweise im Widerspruch zur notwendigen Geschlossenheit, die Voraussetzung für die sichere Verwendung der einzelnen Module ist. Im

objektorientierten Modell läßt sich der konventionelle Widerspruch zwischen Offenheit und Geschlossenheit von Systemkomponenten überbrücken. Vererbung ist der Schlüsselfaktor für entsprechende Konstruktionsprinzipien. Da diese Konstruktionsprinzipien neben ihrer fachlichen auch eine starke technische Motivation besitzen, werden sie erst eingeführt, nachdem das bis jetzt ausgearbeitete Modell aus Einsatzkontext, Produkt- und Gegenstandsbereich mit den beiden in Kapitel 2 identifizierten Kontexten »Modell der verwendeten Technologie« und »Systembasis« zu einem Gesamtmodell verknüpft worden ist. Dieses wird im Abschnitt 5.4 vorgestellt.

4.5 Zusammenfassung und Ausblick

In der Einleitung zu diesem Kapitel habe ich sechs Kriterien festgelegt, die eine Makrostruktur für das fachliche Modell aufweisen muß. Die erste Eigenschaft, nach der die Modellierungskontexte im fachlichen Modell auf der Basis von Makrostrukturen eines Unternehmens zu bilden sind, konnte für Anwendungsbereiche, die nach dem Produkt- oder Prozeßprinzip organisiert sind, wobei die Prozesse überwiegend an Produkten orientiert sein müssen, erfüllt werden. Ebenso ließ sich die bruchlose Übertragbarkeit der definierten Struktur auf das fachliche Modell unter Berücksichtigung der beiden softwaretechnischen Modularisierungsprinzipien minimierte Kopplung und maximierte Kohäsion zeigen. Unberücksichtigt blieb allerdings die Struktur des softwaretechnischen Modells. Eine entsprechende Diskussion erfolgt daher im nächsten Kapitel.

Eine evolutionäre Vorgehensweise wird durch die vorgestellte Strukturierungsmethode ebenso unterstützt. So kann in einem Projekt mit der Modellierung eines ersten Produktbereichs begonnen werden, ohne daß dazu die Aufarbeitung des gesamten Anwendungsbereiches notwendig ist. Das Gesamtsystem kann somit evolutionär durch sukzessive Modellierung der einzelnen Produktbereiche zu einem Gesamtsystem ausgebaut werden. Weiterhin ermöglicht die hier vorgestellte Zerlegung des fachlichen Modells in die Bereiche Einsatzkontext, Produkt- und Gegenstandsbereich die parallele Modellierung mehrerer Produktbereiche. Lediglich produktbereichsübergreifende Begriffe müssen in einem Gegenstandsbereich, der die gemeinsame fachliche Basis bildet, modelliert werden (vgl. Kriterium (5) in der Einleitung dieses Kapitels). Damit die Konstruktion des Gegenstandsbereichs nicht zu derselben Problematik führt, die mit der Entwicklung eines unternehmensweiten Modells verbunden ist, habe ich Eigenschaften und Vorgehensweisen beschrieben, die dies vermeiden. Dies sind insbesondere:

- Der Gegenstandsbereich stellt ein rein softwaretechnisches Konstrukt dar. Innerhalb einer Organisation gibt es keine Unternehmenseinheit, die das Pendant zum Gegenstandsbereich darstellt. Er umfaßt jene Konzepte, die das Kerngeschäft eines Anwendungsbereichs vergegenständlichen und daher allen Produktbereichen gemein sind.
- Die Begriffe des Gegenstandsbereichs werden stets auf der Basis der Begriffe der Produktbereiche gebildet.
- Wird zu einem Produktbereich ein weiterer hinzugefügt, so muß bei dessen Konstruktion sukzessive ein Gegenstandsbereich gebildet werden.

- Die Verwendung des ersten Produktbereichs als Basis (Gegenstandsbereich) für den zweiten führt zu Abhängigkeiten zwischen den Produktbereichen und sollte daher auf jeden Fall vermieden werden.

Mit jedem weiteren Produktbereich ergibt sich zwangsläufig die Notwendigkeit, den Gegenstandsbereich zu überarbeiten. Es sind daher zwingend Konstruktionsprinzipien erforderlich, die die Entwicklung des Gegenstandsbereichs nach dem Offen-Geschlossen-Prinzip ermöglichen. Diese sind bisher aber nur skizziert worden. Ebenso ist die Forderung, daß die entstandenen Produktbereiche sich mittels einer rahmenwerkbasierten Softwarearchitektur realisieren lassen, unberücksichtigt geblieben. Auch die technische Kombinierbarkeit und Integrierbarkeit der durch die rahmenwerkbasierte Softwarearchitektur entstandenen Komponenten zu einer Familie von Anwendungssystemen ist noch nicht gelöst. Eine Diskussion, die speziell diese Punkte berücksichtigt, muß daher in den nächsten beiden Kapiteln erfolgen.

Für die Organisation des fachlichen Modells in die Bereiche Gegenstands-, Produktbereich und Einsatzkontext ergibt sich nunmehr das folgende Bild, bei dem die einzelnen Bereiche wieder mit Beispielen aus dem Bankgeschäft unterlegt werden (vgl. Abb. 4.10):

- *Der Gegenstandsbereich*: Der Gegenstandsbereich wird pro modelliertem Anwendungsbereich nur einmal konstruiert. In ihm sind die Kernkonzepte, die über verschiedene Produktbereiche hinweg existieren, vergegenständlicht. Beispiele für Kernkonzepte im Gegenstandsbereich sind:
  - Fachliche Werte: das Konzept der fachlichen Werte repräsentiert die in einer Bank verwendeten elementaren Werte wie Betrag, Zinssatz, Adresse oder Wechselkurs.
  - Person: das Konzept Person umfaßt Informationen zur Identität von Personen und Bemerkungen, warum die Person zu der Bank in Beziehung steht. Personen müssen nicht zwangsläufig Kunden der Bank sein.
  - Kunde: das Konzept Kunde erweitert das Konzept Person um allgemeine bankspezifische Umgangsformen wie die Verwaltung von Konten, die Berechnung des Gesamtengagements, die Verwaltung einer Kundenakte oder die Vergabe einer eindeutigen Stammnummer.
  - Konto: das Konzept Konto umfaßt den allgemeinen Umgang, der mit jeder Kontoart in einer Bank verbunden ist. Dies sind beispielsweise `einzahlen`, `auszahlen`, `berechneSaldo` und `berechneZinsen`.
  - Produkt: das Konzept Produkt repräsentiert die abstrakten Umgangsformen, die mit einem Bankprodukt, wie etwa einem Sparbuch, einem Bundesschatzbrief oder einer Baufinanzierung verbunden sind. Dies sind etwa: die Verwaltung von Verkaufsinformationen zu einem Produkt, die Produktkonditionen oder das Erstellen einer Modellrechnung.
  - Sicherheit: das Konzept Sicherheit umfaßt den allgemeinen Umgang, der mit jeglicher Art von Sicherheiten in der Bank verbunden ist, wie etwa die Berchnung des Beleihungswertes einer Sicherheit.
  - elektronisches Formularwesen: anhand der Konzepte Formular, Antrag, Vertrag, Dokument, Formularordner und Kundenakte wird das bestehende Formularwesen einer Bank elektronisch nachgebildet.

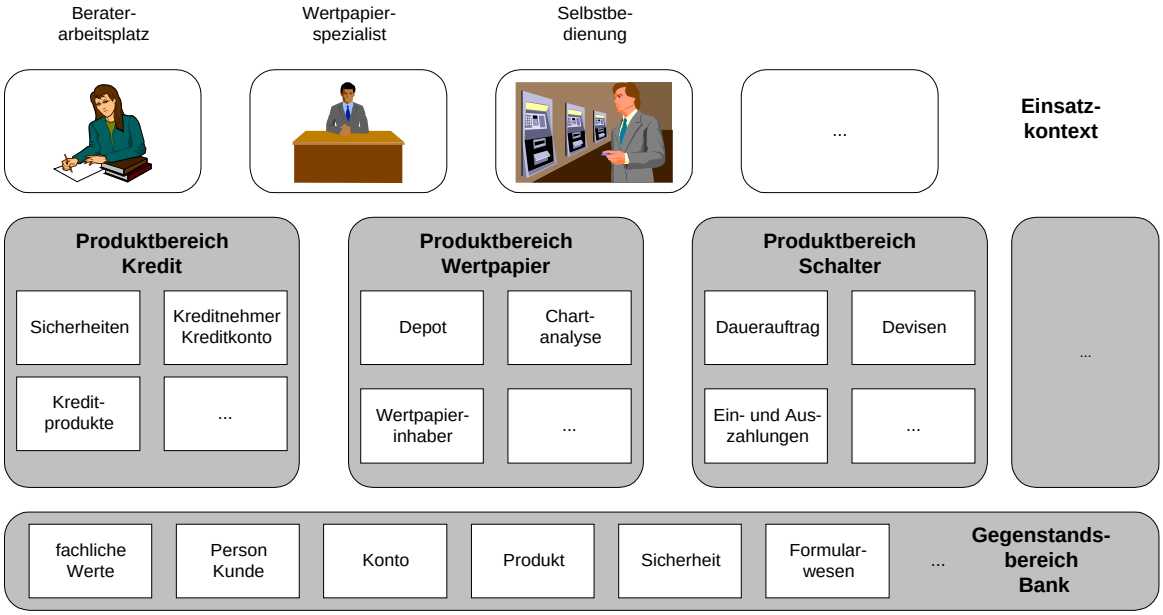


Abb. 4.10: Gegenstandsbereich, Produktbereiche und Einsatzkontexte im Bankenbereich

- *Die Produktbereiche*: Das fachliche Modell kann, analog zu der Aufbauorganisation, in n Produktbereiche unterteilt sein. Pro Produktbereich wird jeweils ein von den anderen Produktbereichen unabhängiges fachliches und softwaretechnisches Modell konstruiert. Diese erweitern entweder Konzepte aus dem Gegenstandsbereich oder bilden neue, nur für diesen Produktbereich relevante Begriffe oder Geschäftsprozesse ab. Das Kredit-, Anlage-, Wertpapier- und Schaltergeschäft stellen Beispiele für Produktbereiche dar. Typische Konzepte im Kreditgeschäft sind:
  - Kreditnehmer: der Kreditnehmer erweitert das Konzept des Kunden aus dem Gegenstandsbereich um für das Kreditgeschäft typische Umgangsformen wie das Verwalten von Sicherheiten, das Führen von Kreditkonten oder die Berechnung der Gesamtverschuldung.
  - Kreditprodukte: Kreditprodukte erweitern das im Gegenstandsbereich modellierte Konzept Produkt. Typische neue Umgangsformen sind: Berechnung des effektiven Zinssatzes oder Angaben zum Verhältnis von Tilgung und Zinssatz. Zudem realisiert dieser Produktbereich den Geschäftsprozeß für die Vergabe eines Kredits.
  - Sicherheiten: dieser Komplex umfaßt die Modellierung der bereits in Abschnitt 2.3.1 vorgestellten Sicherheitenhierarchie.
- *Die Einsatzkontexte*: Analog zum Anwendungsbereich existieren im fachlichen Modell m Einsatzkontexte. Die Anwendungssysteme innerhalb eines Einsatzkontextes setzen sich aus einer Menge von Komponenten aus den Produktbereichen zusammen. Zwischen Einsatzkontext und Produktbereich existiert somit eine n : m Beziehung. Beispiele für Einsatzkontexte sind der Beraterarbeitsplatz, ein Arbeitsplatz für einen Wertpapierspezialisten oder ein Selbstbedienungsterminal.

5 Architekturstile für große rahmenwerkbasierte Anwendungssysteme

"As the size and complexity of software systems increase, the design and specification of overall system structure become more significant issues than the choice of algorithms and data structures of computation." ([SG96], S. 1)

Die bisherige Diskussion war auf organisatorische Strukturen eines Unternehmens und deren Übertragung auf das fachliche Modell beschränkt. In Kapitel 4 habe ich eine Struktur für das fachliche Modell vorgeschlagen, die sich bereits technologisch motivieren ließ, aber weder die Systembasis noch die zur Programmierung verwendete Technologie (vgl. Abschnitt 2.5) explizit berücksichtigt hat. Ein entsprechendes Gesamtbild soll nun in Form zweier Architekturstile präsentiert werden.

Da die Konstruktion der Anwendungssysteme unter Verwendung einer objektorientierten Methode erfolgt, müssen beim Entwurf der Softwarearchitekturen objektorientierte Charakteristika wie Abstraktion, Polymorphie oder dynamisches Binden (vgl. [Mey88] und [Weg87]) berücksichtigt werden. Zudem sind allgemein gültige softwaretechnische Konstruktionsprinzipien wie etwa die Maximierung von Kohäsion, die Minimierung von Kopplung (vgl. [PB93]), „separation of concern“ sowie das Offen-Geschlossen-Prinzip (vgl. [Mey88] und Abschnitt 4.4.1) von den Softwarearchitekturen zu erfüllen.

In Abschnitt 5.1 diskutiere ich zunächst den zentralen Begriff des Architekturstils und beschreibe relevante Anforderungen an eine Architekturbeschreibungssprache. Als Grundlage werden anschließend in Abschnitt 5.2 die Begriffe White-Box und Black-Box Rahmenwerk thematisiert. Aufbauend auf den beiden Abschnitten führe ich dann in Abschnitt 5.3 einen rahmenwerkbasierten Architekturstil ein, der in Abschnitt 5.4 durch einen objektorientierten Schichtenarchitekturstil komplettiert wird. Der Abschnitt 5.5, in dem ich die Umsetzung der beiden Architekturstile als statische und dynamische Bibliotheken vorstelle, bildet den Abschluß dieses Kapitels.

5.1 Softwarearchitekturen und Architekturstile

5.1.1 Der Begriff des Architekturstils

Eine Softwarearchitektur strukturiert das fachliche und softwaretechnische Modell als eine Menge interagierender Komponenten. Verbindungsstücke beschreiben die Kopplung und das Zusammenspiel der Komponenten untereinander. Eine Softwarearchitektur ist als solche immer projekt- bzw. anwendungssystembezogen. Daraus läßt sich allerdings nicht schlußfolgern, daß jedem Softwaresystem eine Architektur unterliegen muß. Viele Systeme zeichnen sich nur durch einen Entwurf aus, der einige architekturelle Eigenschaften wiedergibt (vgl.

[Tra95a]). Existieren vergleichbare Softwarearchitekturen in mehreren Softwaresystemen, so können diese durch einen Abstraktionsprozeß auf ihre nominale Essenz gebracht werden (vgl. Begriff 3.10). Das Ergebnis dieses Abstraktionsprozesses wird als Architekturstil bezeichnet. Monroe et al. schreiben zum Begriff des Architekturstils:

> "An architectural style characterizes a family of systems that are related by shared structural and semantic properties. An architectural style provides a specialized design language for a specific class of systems." ([MKM+97], S. 44)

In diesem Sinne ist ein Architekturstil mit einem Entwurfsmuster (vgl. Begriff 2.11) vergleichbar. Ein Entwurfsmuster wird typischerweise anhand eines Beschreibungsschemas dokumentiert, das seine charakteristischen Eigenschaften dargelegt (vgl. [GOF95]). Dazu zählen beispielsweise der Zweck, die Motivation, die Anwendbarkeit, eine Beispielimplementierung oder bekannte Verwendungen des Musters. Diese Form der Musterbeschreibung hat sich in den letzten Jahren schrittweise herausgebildet und bewährt. Ein vergleichbares Schema existiert auch für die Beschreibung von Architekturstilen. Als charakteristische Eigenschaften von Architekturstilen werden in [SG96] und [MKM+97] genannt:

- *ein Vokabular für die Entwurfselemente*. Ein Entwurfselement ist dabei eine Komponente oder ein Verbindungsstück. Teile eines entsprechenden Vokabulars wären: Client und Server oder Pipe und Filter.
- *Regeln, wie die Entwurfselemente miteinander kombiniert werden dürfen*. Eine Regel legt zum Beispiel für den Pipe-und-Filter Architekturstil fest, daß zyklische Strukturen nicht zulässig sind.
- *eine Semantik*, die die Bedeutung kombinierter Entwurfselemente, deren Anordnung durch Regeln festgelegt wird, beschreibt. Zum Beispiel führt die Kombination von Filtern und Pipes wieder zu neuen Komponenten des Typs Filter.
- *ein Katalog von Analysen und Messungen*, die auf Grund des Aufbaus des Anwendungssystems durchgeführt werden können. Ein Beispiel hierfür ist die Deadlock-Erkennung, die für ein Client-Server System durchgeführt werden sollte, das auf bidirektionalen Message-Queues aufgebaut ist.

In der Literatur (vgl. [MG92], [MKM+97], [SG96] und [Tra95b]) werden eine Reihe von Vorteilen aufgeführt, die mit der Verwendung von Architekturstilen verbunden sind. Diese werden, ergänzt um eigene Erfahrungen, nachfolgend aufgezählt. Der erste und wichtigste Vorteil ist sicherlich die Wiederverwendbarkeit von guten und erprobten Architekturen, was den Entwurf neuer Softwaresysteme stark vereinfachen kann. Zweitens fördert ein gemeinsamer Architekturstil bei der Entwicklung einer Familie von Anwendungssystemen die Standardisierung ihrer Struktur unter einem einheitlichen Gesichtspunkt. Dies wiederum ermöglicht die übergreifende Wiederverwendung von Komponenten, die in verschiedenen Anwendungssystemen oder, in Bezug auf diese Arbeit, in verschiedenen Produktbereichen entwickelt worden sind. Architekturstile ermöglichen es auch, uns vertraute Eigenschaften in Systemen wiederzuerkennen (vgl. Begriff 3.10). Der dritte Vorteil eines Architekturstils ist somit die Verbesserung der Verständlichkeit. Softwareentwickler können Anwendungssysteme, deren Struktur auf der Basis eines bekannten Architekturstils organisiert ist, leichter verstehen.

Viertens ermöglicht ein entsprechender Stil die Verwendung einer angepaßten oder gar speziellen Analysemethode. In diesen Bereich fällt beispielsweise die im letzten Kapitel beschriebene Vorgehensweise, wann und wie die Konstruktion eines Gegenstandsbereichs durchgeführt werden sollte. In Anlehnung an Monroe et al. wird der Begriff des Architekturstils wie folgt definiert:

Begriff 5.1 Architekturstil

Ein Architekturstil definiert die charakteristischen Merkmale einer Menge ähnlicher Softwarearchitekturen. Er wird beschrieben durch ein Vokabular für die Komponenten und Verbindungsstücke, Regeln für ihre Kombination und eine Semantik, die die Bedeutung von kombinierten Entwurfselementen festlegt. Die Beschreibung eines Architekturstils nennt weiterhin Analysen und Messungen, die auf Grund des Aufbaus des Anwendungssystems sinnvoll durchgeführt werden können.

5.1.2 Merkmale einer Architekturbeschreibungssprache

Die Beschreibung einer individuellen Softwarearchitektur oder eines Architekturstils erfolgt entweder graphisch oder textuell. Graphische Architekturbeschreibungen enthalten meist Diagramme aus Kästen und Linien. Kästen sollen wichtige Komponenten der Softwarearchitektur, Linien die Kommunikation zwischen diesen Komponenten darstellen. Leider wird die Semantik der Linien häufig nur sehr ungenau beschrieben, da gängige Architekturbeschreibungen fast ausschließlich auf die Komponenten des Systems fokussieren. Zudem sind die Komponenten i.d.R. nur unzureichend dokumentiert. Eine typische graphische Beschreibung einer Softwarearchitektur ist in Abb. 5.1 dargestellt.

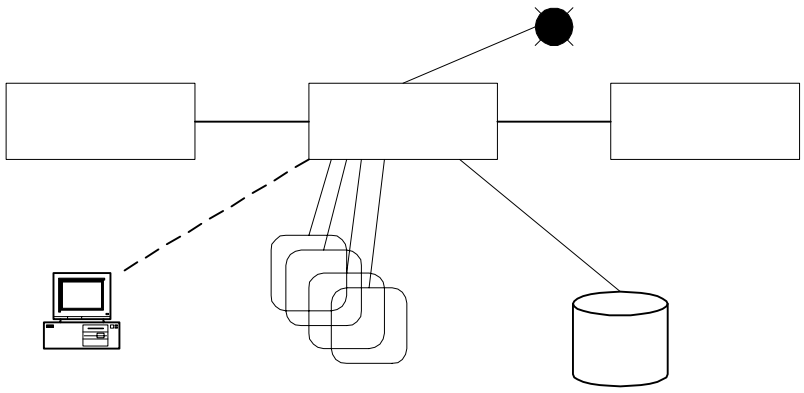


Abb. 5.1: Eine typische graphische Beschreibung einer Softwarearchitektur (aus [SG96], S. 160)

Um die Funktionsweise eines Softwaresystems vermitteln zu können, ist es nicht nur wichtig, die Komponenten geeignet darzustellen, sondern auch das Zusammenspiel zwischen den Komponenten explizit zu beschreiben. Verbindungsstücke sind daher, im Gegensatz zu der heute gängigen Praxis, für die Beschreibung einer Softwarearchitektur von erstrangiger Bedeutung (engl. first-class entity). Eine entsprechende Architekturbeschreibungssprache muß

demnach für Komponenten und Verbindungsstücke gleichwertig graphische oder textuelle Beschreibungselemente mit eindeutiger Semantik bereitstellen.

Nach Shaw & Garlan muß eine Architekturbeschreibungssprache daher sowohl Komponenten als auch Verbindungsstücke definieren, wobei sich die Definition in einen Spezifikations- und Implementierungsteil unterteilt. Bei einer Komponente legt der Spezifikationsteil den Komponententyp und die Schnittstelle in Form von Einheiten (sog. *players*) fest, die mit Verbindungsstücken zusammenarbeiten können. Der Spezifikationsteil eines Verbindungsstücks legt gleichfalls Typ und Schnittstelle fest, wobei die Schnittstelle hier Einheiten (sog. *roles*) definiert, die mit Komponenten zusammenarbeiten können. Die Bezeichner *player* und *role* wurden von den Autoren gewählt, um losgelöst von einem konkreten Architekturstil die Zusammenarbeit zwischen Komponenten und Verbindungsstücken zu symbolisieren. Shaw et al. schreiben in [SDK+95] zu den beiden Begriffen:

> "It is helpful to think of the connector as defining a set of roles that specific named entities of the components must play." ([SDK+95], S. 317)

Neben der Forderung, Komponenten und Verbindungsstücke in der Architekturbeschreibungssprache gleichwertig zu handhaben, geben Shaw & Garlan in [SG96], S. 151 ff. sechs weitere Aspekte an, die eine Sprache aufweisen muß. Diese sind wie folgt beschrieben:

- *Komposition*: Die Sprache muß es ermöglichen, ein Softwaresystem als eine Menge interagierender Komponenten und Verbindungsstücke zu beschreiben. Ein komplexes System kann dadurch in kleinere verständlichere Teile zerlegt werden oder, umgekehrt betrachtet, aus einzelnen Komponenten und Verbindungsstücken zusammengesetzt werden.
- *Abstraktion*: Komponenten und Verbindungsstücke sollten auf einer angemessenen Abstraktionsebene beschrieben werden, so daß die mit den Elementen verbundenen architekturellen Inhalte bereits klar aus der Beschreibung der Softwarearchitektur hervorgehen. So reicht es beispielsweise für eine Client-Server Architektur nicht aus, den Client und den Server als Komponenten, die eine Reihe von Prozeduren zur Verfügung stellen, zu charakterisieren. Denn dadurch wird die Client- und Server-Eigenschaft der beiden Komponenten noch nicht deutlich.
- *Wiederverwendbarkeit*: Bereits existierende Komponenten, Verbindungsstücke und bewährte Kombinationen sollten sich bei der Beschreibung neuer Softwarearchitekturen bzw. neuer Architekturstile wiederverwenden lassen.
- *Konfiguration*: Eine Architekturbeschreibungssprache muß es ermöglichen, die Struktur eines Anwendungssystems zu beschreiben, ohne dabei den inneren Aufbau der Komponenten und Verbindungsstücke zu kennen. Dies soll es ermöglichen, Konfigurationen des Softwaresystems abhängig von verschiedenen Randbedingungen abzuleiten, ohne daß Wissen über die Komponenten oder Verbindungsstücke vorausgesetzt wird. Gegebenenfalls muß die Sprache sogar die Rekonfiguration des Softwaresystems zur Laufzeit unterstützen.

- *Heterogenität*: Komponenten und Verbindungsstücke sollten über Architekturstile hinweg verwendbar sein. Einzelne Schichten einer Schichtenarchitektur sollten beispielsweise mit Hilfe unterschiedlicher Architekturen (z.B. Pipes und Filters oder Event-based) implementiert werden können.
- *Analyse und Messungen*: Die Sprache soll die Durchführung von Analysen und Messungen unterstützen, die auf Grund der Softwarearchitektur sinnvoll an einem Anwendungssystem durchführbar sind.

Die dargelegten Eigenschaften einer Architekturbeschreibungssprache sollen nun anhand eines Beispiels veranschaulicht werden. Dazu soll ein System konstruiert werden, welches einen in Zeilen strukturierten Text von `stdin` einliest, auf die darin einmalig vorkommenden Zeilen reduziert und wieder auf `stdout` ausgibt. Dazu werden die beiden Unix-Filter `sort` und `uniq` sowie eine Unix-Pipe verwendet. Die textuelle Beschreibung des Systems erfolgt dabei in der UniCon-Notation (vgl. [SDK+95]). Sie wird, soweit dies notwendig ist, am Beispiel erläutert. Prinzipiell gilt allerdings: fett und großgeschriebene Bezeichner stellen Schlüsselworte der Sprache UniCon dar (z.B. **`COMPONENT`**), Bezeichner, die mit einem Großbuchstaben beginnen, repräsentieren die Typen der Komponenten, Verbindungsstücke, *player* und *roles* (z.B. `Filter` oder `StreamIn`), kursive und kleine Bezeichner sind eigendefinierte Namen (z.B. *eingabe*). Zuerst werden die beiden Filter als Komponenten definiert, danach eine Unix-Pipe als Verbindungsstück und abschließend das Gesamtsystem als neue Komponente. Die Komponente `sort`:

```
COMPONENT sort
     INTERFACE IS
               // Eine Komponente vom Typ Filter
          TYPE Filter
               // Definition einer Schnittstelle (player) eingabe
               // vom Typ StreamIn
          PLAYER eingabe IS StreamIn
          PLAYER ausgabe IS StreamOut  // analog zu eingabe
     END INTERFACE
     IMPLEMENTATION
          BUILTIN       // Ist als Standardelement vorhanden
     END IMPLEMENTATION
END sort
```

Die Komponente `uniq`:

```
COMPONENT uniq
     INTERFACE IS
          TYPE Filter    // Eine Komponente vom Typ Filter
          PLAYER eingabe IS StreamIn
          PLAYER ausgabe IS StreamOut
     END INTERFACE
     IMPLEMENTATION
          BUILTIN       // Ist als Standardelement vorhanden
     END IMPLEMENTATION
END uniq
```

Das Verbindungsstück `Unix-Pipe`:

```
CONNECTOR unix-pipe
     PROTOCOL IS
          TYPE Pipe     // Ein Verbindungsstück vom Typ Pipe
               // Definition einer Schnittstelle quelle vom
               // Typ Source
          ROLE quelle IS Source   //
          ROLE senke IS Sink // analog zu quelle
     END PROTOCOL
     IMPLEMENTATION
          BUILTIN       // Ist als Standardelement vorhanden
     END IMPLEMENTATION
END unix-pipe
```

Das Gesamtsystem *redu* als neue Komponente:

```
COMPONENT redu
     INTERFACE IS
          TYPE Filter
          PLAYER eingabe IS StreamIn
          PLAYER ausgabe IS StreamOut
     END INTERFACE
     IMPLEMENTATION
               // Definiere eine Variable sortierer vom Typ
               // sort, eine Variable eindeutig vom Typ uniq
               // und eine Variable p vom Typ unix-pipe.
          USES sortierer INTERFACE sort
          USES eindeutig INTERFACE uniq
          USES p PROTOCOL unix-pipe

               // Verbinde die Eingabe von redu mit der Eingabe
               // des Filters sortierer und die Ausgabe von
               // eindeutig mit der Ausgabe von redu.
          BIND eingabe TO sortierer.eingabe
          BIND ausgabe TO eindeutig.ausgabe

               // Verbinde die Ausgabe von sortierer mit der
               // Quelle der Pipe und die Eingabe von eindeutig
               // mit der Senke der Pipe.
          CONNECT sortierer.ausgabe TO p.quelle
          CONNECT eindeutig.eingabe TO p.senke
     END IMPLEMENTATION
END redu
```

Eine graphische Darstellung des Gesamtsystems sieht wie folgt aus (vgl. [SDK+95]):

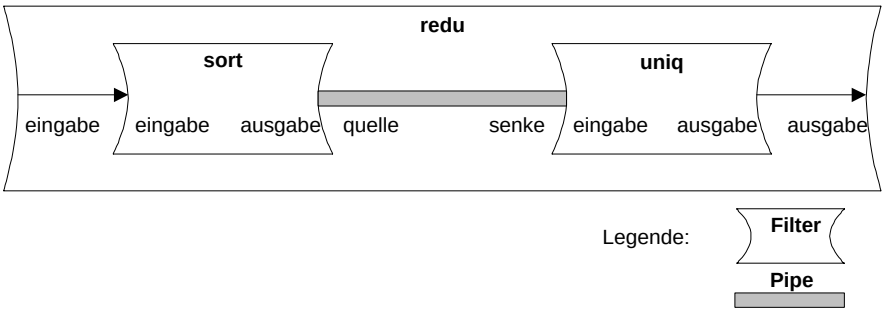


Abb. 5.2: Eine graphische Darstellung des Redu-Systems

Die graphische Darstellung der Architektur eines Systems ermöglicht es einem Softwareentwickler in der Regel, einfach und schnell einen Überblick über das Anwendungssystem zu gewinnen. Textuelle Beschreibungssprachen sind dagegen darauf ausgerichtet, Beschreibungen einer Architektur in ausführbare Einheiten zu übersetzen (vgl. UniCon, [SDK+95]), oder das Verhalten eines Softwaresystems vor dessen Konstruktion zu simulieren (vgl. Rapide, [LKA+95]). Sowohl textuelle als auch graphische Beschreibungen einer Architektur sollten aber so gestaltet sein, daß sie die charakteristischen Merkmale des Softwaresystems widerspiegeln. Monroe et al. schreiben hierzu:

> "[…] an architectural description must be simple enough to permit system-level reasoning and prediction; practically speaking, it should fit on a page or two. Consequently, it is usually hierarchical: atomic architectural elements at one level of abstraction are often described by a more detailed architecture at a lower level." ([MKM+97], S. 44)

Textuelle Architekturbeschreibungen werden schon für kleine Softwaresysteme schnell umfangreich und komplex (siehe vorheriges Beispiel). Da weder die oben angesprochene automatische Generierung noch die Simulation von Anwendungssystemen im Fokus dieser Arbeit stehen, werde ich im folgenden mit einer graphischen Notation arbeiten. Sie ist speziell darauf zugeschnitten, die Architektur großer rahmenwerkbasierter Anwendungssysteme zu beschreiben und wird daher parallel zu den von mir vorgestellten Architekturstilen entwickelt. Einleitend soll zunächst die aktuelle Diskussion über Rahmenwerke zusammengefaßt werden.

5.2 Zum Begriff des Rahmenwerks

Bei der Entwicklung einer Familie objektorientierter Anwendungssysteme werden durch die Konstruktion der ersten Systeme meistens Rahmenwerke bereitgestellt, die als Komponenten für die Entwicklung weiterer Systeme wiederverwendet werden. Ein Architekturstil für die Beschreibung derartiger Anwendungssysteme muß Rahmenwerke als Komponenten auffassen (vgl. Abschnitt 2.6). Bevor ich den Verwendungszusammenhang eines Rahmenwerks detaillierter untersuche, soll zuerst eine technisch fundiertere Fassung des Begriffs Rahmenwerk gegeben werden (vgl. [JR91] und Begriff 1.2)):

Begriff 5.2 Rahmenwerk (2. Fassung)

Ein Rahmenwerk stellt eine softwaretechnische Lösung für eine Reihe verwandter Probleme zur Verfügung. Statisch besteht es aus einer Menge von Klassen, die, bezogen auf das zu lösende Problem, das Zusammenspiel (Kontrollfluß) einer Gruppe von Objekten zur Laufzeit eines Softwaresystems festlegen. Im Unterschied zu Entwurfsmustern sind Rahmenwerke implementierte Einheiten. Sie ermöglichen somit die Wiederverwendung von Analysen, Entwurfsentscheidungen, Architekturen und Implementierungen. Ein Rahmenwerk definiert dabei auch die Stellen, an denen seine Funktionalität erweitert und angepaßt werden kann.

In der Literatur (vgl. [JF88]) werden Rahmenwerke auf Grund ihrer Verwendung in Black-Box und White-Box Rahmenwerke unterschieden. Ein Black-Box Rahmenwerk wird bei der

Entwicklung von Softwaresystemen verwendet, indem von speziell dafür vorgesehenen Rahmenwerkklassen Objekte erzeugt werden. Diese Objekte lassen sich häufig mit Hilfe von weiteren Objekten konfigurieren. Solche Konfigurationsobjekte werden ebenfalls von Klassen desselben oder eines anderen Black-Box Rahmenwerks instanziiert. Dies soll an dem bereits in Abschnitt 2.6 eingeführten Konto-Verzinser Beispiel erläutert werden. Die graphische Darstellung des Beispiel wird daher in Abb. 5.3 nochmals präsentiert.

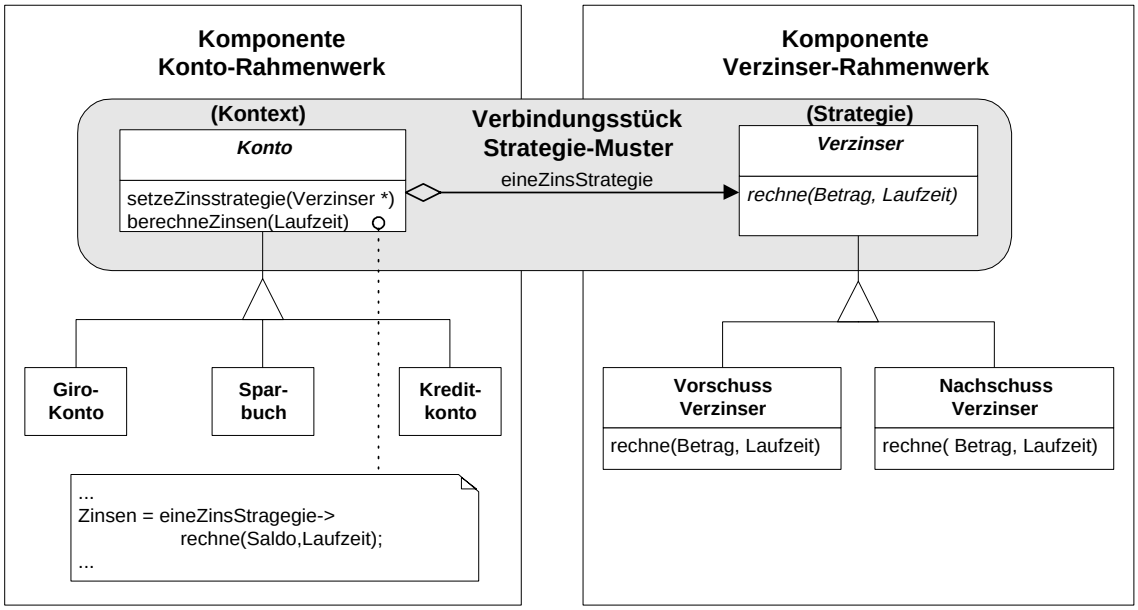


Abb. 5.3: Das Konto-Verzinser Beispiel

Möchte ein Softwareentwickler etwa ein Sparbuch mit einer nachschüssigen Verzinsung erzeugen, so instanziiert er zunächst ein Objekt der Klasse `Sparbuch`, dem er dann ein Objekt der Klasse `NachschussVerzinser` übergibt. Dadurch konfiguriert er die Zinsberechnung des Sparbuchs auf eine nachschüssige Verzinsung. Die Bezeichnung Black-Box bringt dabei zum Ausdruck, daß der Softwareentwickler bei der Verwendung des Rahmenwerks keine Kenntnisse über die Interna des Rahmenwerks benötigt.

Die Verwendung eines White-Box Rahmenwerks hingegen ist nur mit Wissen über den internen Aufbau des Rahmenwerks, insbesondere über das in den Klassen modellierte Zusammenspiel der Objekte möglich. White-Box Rahmenwerke werden eingesetzt, indem ein Softwareentwickler von speziell dafür vorgesehenen Klassen des Rahmenwerks Unterklassen bildet. Die von diesen Unterklassen erzeugbaren Objekte lassen sich auf zwei unterschiedliche Arten verwenden.

Im ersten Fall werden sie direkt in einem Anwendungssystem benutzt. Das Konto-Rahmenwerk könnte beispielsweise so gestaltet sein, daß es die Klasse `Konto` und das abstrakte Zusammenspiel mit Objekten der Klasse `Verzinser` implementiert. Konkrete `Konto` Unterklassen wären in dem Rahmenwerk nicht enthalten und müßten daher in den Anwendungssystemen durch Unterklassenbildung konstruiert werden. Aus dem Rahmenwerk erbt eine

Unterklasse dann das Zusammenspiel mit Verzinsern, das sie demnach nicht selber implementiert haben muß.

Im zweiten Fall werden Objekte der Subklassen zur Konfiguration eingesetzt. Das Rahmenwerk könnte etwa konkrete Kontoklassen (`GiroKonto`, `Sparbuch`, usw.), aber keine Verzinsungsstrategien bereitstellen. In einem Anwendungssystem ist daher eine geeignete Unterklasse der abstrakten Klasse `Verzinser` zu bilden, dessen Objekte dann zur Konfiguration konkreter Kontoobjekte benutzt werden.

Wie die vorherigen Beispiele zeigen, sind Black-Box Rahmenwerke einfach zu verstehen, während sich White-Box Rahmenwerke in ihrer Verwendung flexibler gestalten. Erfahrungen aus dem RWG-Kontext zeigen, daß Black-Box Rahmenwerke häufig durch eine sukzessive Erweiterung von White-Box Rahmenwerken um weiterreichende Standardlösungen entstehen. Johnson & Foote schreiben hierzu:

> "In fact, as the design of a system becomes better understood, black-box relationships should replace white-box ones. Black-Box relationships are an ideal towards which a system should evolve." ([JF88], S. 26)

Hier wird bereits deutlich, daß Rahmenwerke in der Softwareentwicklung nicht nur bei der Konstruktion von Anwendungssystemen eingesetzt werden, sondern auch als Basis für die Entwicklung weiterer Rahmenwerke dienen.

Black-box und white-box sind keine statischen Eigenschaften eines Rahmenwerks. Sie ergeben sich vielmehr erst aus dem jeweiligen Verwendungszusammenhang bei der Softwareentwicklung. Zwar werden Rahmenwerken, die für eine White-Box Verwendung konzipiert sind, hauptsächlich auch als solche verwendet, sie können aber dennoch vorgefertigte Standardlösungen enthalten, die Softwareentwickler black-box-artig verwenden. Umgekehrt erlauben Black-Box Rahmenwerke, wenn die notwendige Dokumentation existiert, natürlich auch immer eine white-box Verwendung.

Leider wird diese Unterscheidung in der Literatur zumeist nicht getroffen. Für meine weitere Arbeit möchte ich den jeweiligen Verwendungszusammenhang jedoch explizit machen. In Anlehnung an [JF88] und basierend auf der Definition eines Rahmenwerks (vgl. Begriff 5.2) definiere ich die beiden Begriffe deshalb wie folgt:

Begriff 5.3 White-Box Rahmenwerk

Ein White-Box Rahmenwerk ist in seiner Verwendung primär darauf ausgerichtet, von speziell dafür vorgesehenen Klassen Unterklassen zu bilden. Objekte dieser Unterklassen lassen sich sowohl direkt in Anwendungssystemen oder weiteren Rahmenwerken benutzen, als auch zu Konfiguration von Objekten einsetzen, die ebenfalls von Klassen eines Rahmenwerks instanziiert wurden.

Enthält ein White-Box Rahmenwerk Klassen, die sich black-box artig verwenden lassen, beziehe ich mich darauf als ein *geschlossen verwendbares* White-Box Rahmenwerk.

Begriff 5.4 Black-Box Rahmenwerk

Ein Black-Box Rahmenwerk ist in seiner Verwendung primär darauf ausgerichtet, Objekte von darin implementierten Klassen zu erzeugen, die sich in der Regel durch Konfigurationsobjekte anwendungsspezifischen Erfordernissen anpassen lassen.

Soll ein Black-Box Rahmenwerk mittels Unterklassenbildung auch white-box artig benutzt werden, so beziehe ich mich darauf als ein *offen verwendbares* Black-Box Rahmenwerk.

Neben Rahmenwerken werden bei der Entwicklung von objektorientierten Softwaresystemen auch Klassenbibliotheken eingesetzt. Ich möchte den Begriff der Klassenbibliothek an dieser Stelle explizit von statischen und dynamischen Bibliotheken (engl. libraries) abgrenzen. Die technische Umsetzung von Rahmenwerken und Klassenbibliotheken in solche vorübersetzten Einheiten werde ich detailliert in Abschnitt 5.5 diskutieren.

Eine Klassenbibliothek bündelt, analog zu einem Rahmenwerk, eine Menge von Klassen. Allerdings legt sie dabei nicht das Zusammenspiel einer Gruppe von Objekten zur Laufzeit fest. In der Regel ist jede Klasse einer Klassenbibliothek unabhängig von anderen Klassen der Bibliothek (ausgenommen Hilfsklassen) verwendbar. Objekte, die von Klassen einer Bibliothek erzeugt werden, kooperieren daher auch nicht mit weiteren aus der Bibliothek instanziierten Objekten. Das Zusammenspiel zwischen den Objekten, d.h. der Kontrollfluß, wird erst im Programmkode des Anwendungssystems festgelegt.

Container-Bibliotheken repräsentieren ein typisches Beispiel für eine Klassenbibliothek. Normalerweise erzeugt ein Klient in einem Anwendungssystem ein einzelnes Objekt aus einer Container-Bibliothek, beispielsweise eine sortierte lineare Liste, und benutzt die an seiner Schnittstelle angebotenen Dienstleistungen. Weitere Objekte, die zur Realisierung der Dienstleistung benötigt werden, sind für den Klienten nicht sichtbar. Wird allerdings die Container-Bibliothek um robuste Iteratoren ergänzt (vgl. [Gam92], S. 33), so evolviert die Klassenbibliothek zu einem Black-Box Rahmenwerk. Denn durch die zwei abstrakten Oberklassen `Container` und `Iterator` wird ein Zusammenspiel zwischen klientenseitig instanziierten Container- und Iterator-Objekten festgelegt. Das definierte Zusammenspiel garantiert, daß ein auf einer Liste operierender Iterator vom Entfernen eines Objektes aus der Liste in Kenntnis gesetzt wird. Der Listeniterator kann somit nie auf ein bereits entferntes Objekt verweisen. Der Begriff der Klassenbibliothek wird im weiteren wie folgt verwendet:

Begriff 5.5 Klassenbibliothek

Eine Klassenbibliothek faßt eine Menge von einzeln verwendbaren Klassen zu einer konzeptionellen Einheit zusammen. Im Gegensatz zu einem Rahmenwerk legt eine Klassenbibliothek allerdings nicht das Zusammenspiel von Objekten zur Laufzeit eines Anwendungssystems fest.

5.3 Ein rahmenwerkbasierter Architekturstil

Ein Architekturstil definiert unter anderem ein Vokabular für Komponenten und Verbindungsstücke, das auch graphisch repräsentierbar sein muß. Für die Darstellung von Klassenbibliotheken und Rahmenwerken wird die in Abb. 5.4 beschriebene Notation verwendet. Sie basiert auf informellen Notationen, die in [Lew95] zur Beschreibung von bekannten rahmenwerkbasierten Systemen (ET++ und Taligents CommonPoint) benutzt werden.

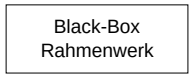
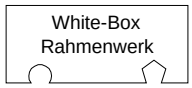
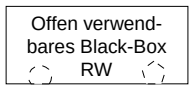
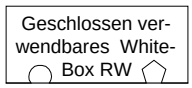
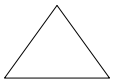
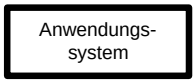
Graphisches Element	Interpretation
	Black-Box Rahmenwerke werden durch Rechtecke dargestellt.
	White-Box Rahmenwerke werden durch Rechtecke mit Einbuchtungen dargestellt. Die Einbuchtungen symbolisieren dabei die Verwendung des Rahmenwerks durch Unterklassenbildung.
	Offen verwendbare Black-Box Rahmenwerke werden durch Rechtecke mit einer gestrichelten Einbuchtung dargestellt.
	Geschlossen verwendbare White-Box Rahmenwerke werden durch Rechtecke mit Einbuchtungen dargestellt. Die Einbuchtungen sind durch eine Linie geschlossen.
	Klassenbibliotheken werden durch Dreiecke dargestellt.
	Anwendungssysteme werden durch Rechtecke mit dicker Umrandung dargestellt.

Abb. 5.4: Komponenten eines rahmenwerkbasierten Architekturstils, ihre graphische Repräsentation und ihre Interpretation

Bereits in Abschnitt 2.6 wurde dargelegt, daß sich Entwurfsmuster zur Beschreibung von komplexen Verbindungsstücken einsetzen lassen. Diese Aussage ist nun insbesondere vor dem Hintergrund zu detaillieren, daß die black-box und white-box Verwendung von Rahmenwerken die Konstruktion eines Softwaresystems unterschiedlich beeinflussen.

Entwurfsmuster beschreiben die Verantwortlichkeiten und die Zusammenarbeit einer Reihe von Klassen, die ein kontextbezogenes Entwurfsproblem lösen. Teilen sich nun die Klassen des Entwurfsmusters auf mehrere miteinander zu koppelnde Rahmenwerke auf, so beschreibt das Entwurfsmuster das Protokoll, mit dem die einzelnen Rahmenwerke interagieren. Um dies zu verdeutlichen, betrachten wir erneut die Rahmenwerke Konto und Verzinser. Das Verzinser-Rahmenwerk ist als Black-Box Rahmenwerk vorhanden und wird zur Konstruktion des Konto-Rahmenwerks verwendet. Die Verzinser-Klassen lassen sich als Strategien in den Konto-Klassen einsetzen. Um das Beispiel zu vereinfachen, werden die entsprechenden Strategie-Objekte in den konkreten Konto-Klassen erzeugt.

Betrachtet man nun das Strategie-Muster als Verbindungsstück, so muß in einem Diagramm erkennbar sein, welche Klassen des Konto- und Verzinser-Rahmenwerks welche Verantwortlichkeiten des Strategie-Musters übernehmen. Dies wird durch eine Zuordnung der Klassen im Verzinser- und Konto-Rahmenwerk zu den entsprechenden Klassen im Strategie-Muster repräsentiert. Ein Pfeil (➔) drückt diese Zuordnung aus. Er ist wie folgt zu lesen ist: Konto ➔ Kontext bedeutet, daß die `Konto` Klasse innerhalb des Strategie-Musters die Verantwortlichkeiten des Kontextes wahrnimmt. Am Anfang des Pfeils steht dabei immer ein Klassenname des Rahmenwerks, am Ende der entsprechende Klassenname des Entwurfsmusters. Stimmt der Klassenname des Rahmenwerks mit dem des Entwurfsmusters überein, so kann die Assoziation weggelassen werden.

Als graphisches Element für ein Verbindungsstück vom Typ Entwurfsmuster wird eine Linie mit einer Ellipse verwendet. Die Ellipse ist mit dem Namen des Entwurfsmusters beschriftet.

Abb. 5.5: Das Strategie-Muster als Verbindungsstück zwischen zwei Black-Box Rahmenwerken

Die dargestellte Abbildung drückt nicht aus, welche konkrete Verzinsungsstrategie aus dem Verzinser-Rahmenwerk für die Implementierung einer Konto-Klasse im Konto-Rahmenwerk verwendet wird. Diese Information hat keinen architekturellen Charakter, sondern stellt ein fachliches Detail dar, über dessen Umsetzung bei der Konstruktion der konkreten Konto-Klassen zu entscheiden ist.

Würde das Konto-Rahmenwerk das Verzinser-Rahmenwerk nicht als Black-Box, sondern als White-Box Rahmenwerk verwenden, so müßte im Konto-Rahmenwerk eine eigene Verzinsungsstrategie implementiert werden. Dies wird deutlich gemacht, indem eine Klasse im Konto-Rahmenwerk die Verantwortlichkeit einer konkreten Strategie übernimmt. Ein entsprechendes Diagramm sieht dann wie folgt aus:

Abb. 5.6: Das Strategie-Muster als Verbindungsstück zwischen einem Black-Box und einem White-Box Rahmenwerk

Nicht jedes Entwurfsmuster repräsentiert zwangsläufig ein Verbindungsstück. Komplexe Verbindungsstücke, die Protokollcharakter haben, können nur sehr einfach in Form von Entwurfsmustern beschrieben werden.

Rahmenwerke werden des öfteren einfach nur über Unterklassenbildung oder durch die Verwendung einer konkreten Klasse miteinander verbunden. Für diese einfachen Verbindungs-

stücke werden ebenfalls entsprechende graphische Symbole eingeführt, die an die im Entwurfsmuster-Buch (vgl. [GOF95]) verwendete Notation für Klassendiagrammen angelehnt ist. Eine Linie mit einem Dreieck steht für Vererbung, ein gerichteter Pfeil für eine Benutzt-Beziehung. Benutzt-Beziehungen werden allerdings nicht weiter in Aggregation, Assoziation oder »Menge von« unterschieden. Für Verbindungsstücke ergeben sich abschließend die in der Abb. 5.7 dargestellten graphischen Elemente.

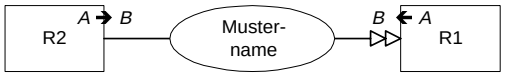
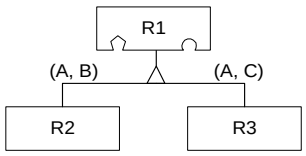
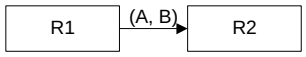
Graphisches Element	Interpretation
	Verbindungsstücke, die sich durch ein Entwurfsmuster beschreiben lassen, werden durch eine Linie mit einer Ellipse dargestellt. Die Notation A → B zeigt an, daß die Klasse A im Rahmenwerk R_2 die Verantwortlichkeiten der Klasse B des Entwurfsmusters wahrnimmt. Die Assoziation kann weggelassen werden, wenn der Klassenname des Rahmenwerks und des Entwurfsmusters übereinstimmen. Ein Doppelpfeil symbolisiert die Verwendung der Komponente R_1 durch die Komponente R_2.
	Eine Linie mit einem integrierten Dreieck symbolisiert Vererbung zwischen Klassen, die in unterschiedlichen Rahmenwerken liegen. Die Annotation (A,B) zeigt dabei an, daß die Klasse A des Rahmenwerks R_1 Oberklasse der Klasse B des Rahmenwerks R_2 ist. Diese Form von Verbindungsstück impliziert zugleich mögliche Operationsaufrufe entlang der Vererbungsbeziehung.
	Eine Benutzt-Beziehung zwischen zwei Klassen in unterschiedlichen Rahmenwerken wird durch einen gerichteten Pfeil vom benutzenden zum benutzten Rahmenwerk ausgedrückt. Eine Annotation der Form (A,B) benennt dabei die sich benutzenden Klassen. Analog zur Vererbung werden auch durch eine Benutzt-Beziehung mögliche Operationsaufrufe impliziert.

Abb. 5.7: Verbindungsstücke eines rahmenwerkbasierten Architekturstils, ihre graphische Repräsentation und ihre Interpretation

Entlang des in Abschnitt 5.1.1 vorgestellten Beschreibungsschemas (Vokabular, Kombinationsregeln, Semantik, Analysen und Messungen) wird im folgenden informell ein rahmenwerkbasierter Architekturstil definiert.

Das *Vokabular* stellt sich wie folgt dar (vgl. Abb. 5.4 und 5.7):

- Komponenten: Klassenbibliothek, Black-Box, offen verwendbares Black-Box, White-Box, geschlossen verwendbares White-Box Rahmenwerk und Anwendungssystem.
- Verbindungsstücke: Entwurfsmuster, Vererbung und Benutzt-Beziehung.

Die *Kombinationsregeln* für Entwurfselemente sind wie definiert:

- Rahmenwerke können über mehrere Verbindungsstücke, insbesondere über mehrere Entwurfsmuster, miteinander verbunden sein.

- Zyklische Strukturen sollten vermieden werden, da ansonsten die Entwicklung und Konfiguration neuer Komponenten erschwert wird (vgl. [Lak96]).
- Klassenbibliotheken dürfen nur als Basiskomponenten verwendet werden. Sie können nicht als Ergebnis einer Kombination mit Rahmenwerken entstehen. Eine Klassenbibliothek, die auf einem Rahmenwerk basiert, würde zwangsläufig auch das Zusammenspiel von Objekten beschreiben. Dies ist jedoch erklärtermaßen für Klassenbibliotheken ausgeschlossen (vgl. Begriff 5.5).
- Die Definition neuer Verbindungsstücke unter Verwendung von bereits existierenden Entwurfselementen wird nicht unterstützt. Entwurfsmuster müssen auf herkömmliche Weise (vgl. [GOF95]) beschrieben werden.

Die Entwicklung neuer Komponenten und Verbindungsstücke durch alleinige Kombination von bereits vorhandenen Entwurfselementen, wie etwa beim Pipes und Filters Stil, ist nicht möglich. Die Konstruktion ist stets mit Programmieraufwand verbunden. Eine Semantik für kombinierte Entwurfselemente, die über die bereits definierte Semantik für die Komponenten und Verbindungsstücke hinausgeht, kann daher nicht festgelegt werden. Sie hängt immer von der Semantik ab, die mit dem zusätzlichen Programmkode verbunden ist. Messungen und Analysen, die mit dem Architekturstil verbunden sind, beziehen sich vor allem auf die Überprüfung von zyklischen Strukturen.

In Abschnitt 5.1.2 wurden verschiedene Aspekte vorgestellt, die nach Shaw & Garlan (vgl. [SG96]) von einer Architekturbeschreibungssprache zu erfüllen sind. Es ist daher zu zeigen, inwieweit diese Aspekte von der vorgestellten graphischen Notation erfüllt werden.

- *Komposition*

  Rahmenwerkbasierte Anwendungssysteme können als untereinander verbundene Menge von White-Box sowie Black-Box Rahmenwerken und Klassenbibliotheken beschrieben werden, wobei als Verbindungsstücke die Entwurfselemente Muster, Vererbung und Benutzt-Beziehung verwendet werden. Das Merkmal Komposition als Eigenschaft, ein Softwaresystem als Menge interagierender Komponenten und Verbindungsstücke zu beschreiben, ist somit gegeben.

- *Abstraktion*

  Sowohl Komponenten als auch Verbindungsstücke sind Abstraktionen von den elementaren Strukturen einer objektorientierten Programmiersprache. Dadurch können die architekturellen Eigenschaften eines Systems einfach und kompakt beschrieben werden (vgl. Zitat auf Seite 93).

- *Wiederverwendbarkeit*

  Bereits definierte Komponenten und Verbindungsstücke können bei der Beschreibung einer Softwarearchitektur wiederverwendet werden. Das Konto-Rahmenwerk beruht zum Beispiel auf dem bereits existierenden Verzinser-Rahmenwerk. Die Notation erfüllt somit auch das Merkmal der Wiederverwendbarkeit.

- *Konfiguration*

  Die statische Konfiguration eines rahmenwerkbasierten Softwaresystems läßt sich durch die vorgestellte Notation beschreiben, die Charakterisierung der dynamischen Konfiguration, etwa zum Startzeitpunkt des Anwendungssystems, wird nicht unterstützt. Insbesondere ist die dynamische Rekonfiguration eines Softwaresystems zur Laufzeit nicht möglich. Dies bedeutet nicht, daß sich rahmenwerkbasierte Softwaresysteme nicht dynamisch konfigurieren lassen. In Kapitel 6 wird das Entwurfsmuster Produkthändler vorgestellt, das speziell die Forderung der Konfigurierbarkeit unterstützt.

- *Heterogenität*

  Heterogenität ist nur insoweit gegeben, als daß sich Komponenten eines anderen Architekturstils mit Hilfe von Komponenten und Verbindungsstücken des vorgestellten rahmenwerkbasierten Architekturstils beschreiben lassen. So läßt sich etwa eine Schicht oder ein Filter mit dessen Hilfe repräsentieren. Die Beschreibung von rahmenwerkbasierten Komponenten und Verbindungsstücken unter Einsatz eines anderen Stils ist nicht möglich, da beide Entwurfselemente sehr eng mit der objektorientierten Programmiermethodik verbunden sind.

- *Analysen und Messungen*

  Analysen und Messungen können im Rahmen der auf der vorherigen Seite definierten Regeln an einer Softwarearchitektur, die in der vorgestellten Notation beschrieben ist, vorgenommen werden. Sie sind allerdings vom Softwareentwickler durchzuführen. Eine maschinelle Überprüfung entsprechender Kriterien ist auf Grund einer fehlenden textuellen Beschreibung des Architekturstils nur begrenzt möglich.

5.4 Ein objektorientierter Schichtenarchitekturstil für interaktive Anwendungssysteme

In Abschnitt 4.4 habe ich das fachliche Modell in einen Gegenstandsbereich und darauf aufbauende Produktbereiche und Einsatzkontexte strukturiert. Wie die in diesem Abschnitt geführte Diskussion gezeigt hat, existieren zwischen den Bereichen Abhängigkeiten, die eine Umsetzung der Struktur als Schichtenarchitektur nahelegen. Dabei bilden der Gegenstandsbereich, die Produktbereiche und die Einsatzkontexte jeweils eine eigene Schicht. Der Gegenstandsbereich muß sich unter Verwendung objektorientierter Konstruktionsprinzipien für Produktbereiche erweitern lassen, ohne dabei selber verändert zu werden (Offen-Geschlossen-Prinzip). Ziel dieses Abschnitts ist es daher, Konzepte von Schichtenarchitekturen mit objektorientierten Techniken zusammenzuführen. Dazu diskutiere ich zuerst die Merkmale von Schichtenarchitekturen am Beispiel des OSI-Schichtenmodells. Die so erarbeiteten Charakteristika werden dann in den Kontext objektorientierter Softwareentwicklung eingebettet. Auf den Ergebnissen dieser Diskussion läßt sich dann ein objektorientierter Schichtenarchitekturstil definieren, der die bereits erarbeitete Struktur des fachlichen Modells reflektiert. Das OSI-Schichtenmodell wurde dabei als einführendes Beispiel gewählt, da sein prinzipieller Aufbau und seine Funktionsweise vielen Informatikern bekannt sind und daher auf eine detaillierte Erklärung des Modells verzichtet werden kann. Eine inhaltliche Verbindung zwischen dem OSI-Schichtenmodell und dem im Abschnitt 5.4.4 definierten objektorientierten Schichtenarchitekturstil, die über die konzeptionellen Merkmale hinausgeht, ist nicht gegeben.

5.4.1 Merkmale von Schichtenarchitekturen

Das wohl am weitesten verbreitete Beispiel einer Schichtenarchitektur ist das OSI-Schichtenmodell[16], das einen Kommunikationsvorgang zwischen einem Sender und einem Empfänger beschreibt (vgl. [McC91]). Das Modell teilt dabei den Kommunikationsvorgang in sieben funktionale Schichten ein: physikalische Übertragungsschicht (physical layer), Datenverbindungsschicht (data link layer), Netzwerkschicht (network layer), Transportschicht (transport layer), Kommunikationsschicht (session layer), Präsentationsschicht (presentation layer) und Anwendungsschicht (application layer).

Typischerweise verbindet ein Softwareentwickler folgende Eigenschaften mit einer Schichtenarchitektur, die wohlgemerkt nicht zwangsläufig auf alle in der Literatur diskutierten Schichtenarchitekturen zutreffen:

(a) *Abstraktion*: Eine Schichtenarchitektur organisiert ein System hierarchisch. Ziel der Hierarchisierung ist die Bildung unterschiedlicher Abstraktionsstufen. Dabei abstrahieren höherliegende Schichten von den Eigenschaften tieferliegender Schichten. Im OSI-Schichtenmodell wird beispielsweise mit jeder höherliegenden Schicht stärker von der technischen und physikalischen Realisierung eines Kommunikationsvorgangs abstrahiert. Zur Dualität von Schichten und Abstraktion schreibt Ossher:

> "A system based on levels of abstraction has a layered structure: each layer corresponds to an abstraction level." ([Oss87], S. 220)

(b) *Durchlässigkeit*: Jede Schicht erbringt eine definierte Leistung, die sie ausschließlich Elementen ihrer oder nächsthöheren Schichten zur Verfügung stellt. Eine Schicht kennt somit nur darunterliegende Schichten, aber keine höherliegenden Schichten. Objektorientiert interpretiert bedeutet dies, daß Klassen nur von anderen Klassen der eigenen oder der nächsthöheren Schicht verwendet werden dürfen.

Rumbaugh unterscheidet in [Rum91] offene und geschlossene Schichtenarchitekturen. In geschlossenen Architekturen greifen Elemente einer Schicht ausschließlich auf Elemente der direkt darunterliegenden Schicht zu, während in offenen Architekturen auch Zugriffe über mehrere Schichten hinweg zulässig sind. Um eine Begriffsverwirrung mit der offenen und geschlossenen Verwendbarkeit von Rahmenwerken sowie dem Meyer'schen Offen-Geschlossen-Prinzip zu vermeiden, bezeichne ich im weiteren offene als *durchlässige Schichtenarchitekturen* und geschlossene als *undurchlässige Schichtenarchitekturen.*

(c) *Lokalität von Änderungen*: Mit Schichtenarchitekturen ist häufig der Anspruch verbunden, daß sich Änderungen bzw. Erweiterungen an einer Schicht lediglich lokal auswirken. Änderungen an der Implementierung einer Schicht sollten keine Auswirkungen auf andere Schichten zeigen, Änderungen an der Schnittstelle bei undurchlässigen Schichtenarchitekturen wenn möglich nur auf die nächsthöhere Schicht. Denn dann können die Schichten teilweise unabhängig voneinander entwickelt werden. Die nachfolgend geführte Diskussion wird allerdings zeigen, daß sich auch bei undurch-

[16] Anstelle des Begriffs OSI-Schichtenarchitektur wird hier in Anlehnung an die gängige Verwendung des Begriffs in der Literatur von dem OSI-Schichten*modell* gesprochen.

lässigen Schichtenarchitekturen Änderungen über Schichten hinweg fortpflanzen können. Bei durchlässigen Schichtenarchitekturen sind alle Schichten betroffen, die die geänderte Schicht benutzen.

Das OSI-Schichtenmodell sieht darüber hinaus ein Referenzmodell vor, das die zu erbringenden Leistungen und öffentlichen Schnittstellen der einzelnen Schichten festlegt. Dies ist jedoch nicht bei jeder Schichtenarchitektur der Fall. Die oben aufgezählten drei Merkmale werden auch von Shaw & Garlan in ihrer Beschreibung einer Schichtenarchitektur aufgeführt. Sie halten fest:

> "A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below. In some layered systems inner layers are hidden from all except the adjacent layer, except for certain functions carefully selected for export. […]. In other layered systems the layers may be only partially opaque." ([SG96], S. 25)

Werden Schichtenarchitekturen im Kontext objektorientierter Softwareentwicklung diskutiert, muß ausdifferenziert werden, wie die Klassen als Anbieter der Leistungen einer Schicht verwendet werden dürfen. In traditionellen Schichtenarchitekturen werden Leistungen in Form von Operationsaufrufen (Benutzt-Beziehung) in Anspruch genommen. In objektorientierten Architekturen besteht darüberhinaus die Möglichkeit, eine Klasse mittels Vererbung zu verwenden, d.h. Klassen einer höherliegenden Schicht können Subtypen von Klassen aus tieferliegenden Schichten sein (vgl. Abb. 5.8). Damit ergibt sich speziell für objektorientierte Schichtenarchitekturen ein weiteres Merkmal:

(d) *Striktheit*: Ist Vererbung zwischen den Schichten auf Grund der Architektur ausgeschlossen, so spricht man von einer strikten Schichtenarchitektur. Ist sie zulässig, so spricht man von einer nicht-strikten Schichtenarchitektur. Es sei darauf hingewiesen, daß die Charakterisierung einer Schichtenarchitektur als durchlässig bzw. undurchlässig orthogonal zu dem Merkmal der Striktheit erfolgt.

Da ich diese Ausdifferenzierung für die weitere Arbeit als wichtig erachte, werden die beiden Begriffe wie folgt definiert:

Begriff 5.6 Strikte Schichtenarchitektur

Eine Schichtenarchitektur, die Vererbung über Schichten hinweg *ausschließt*, wird als *strikte* Schichtenarchitektur bezeichnet. Vererbung innerhalb einer Schicht wird dadurch allerdings nicht ausgeschlossen.

Begriff 5.7 Nicht-strikte Schichtenarchitektur

Eine Schichtenarchitektur, die Vererbung über einzelne Schichten hinweg *zuläßt*, wird als *nicht-strikte* Schichtenarchitektur bezeichnet.

Die Darstellung der Schichten- und Klassenhierarchie in einer gemeinsamen Abbildung ist mit Problemen verbunden, die sich darauf zurückführen lassen, daß die Hierarchien nach verschiedenen Abstraktionskriterien gebildet werden. In objektorientierten Modellen abstrahieren Oberklassen von den Gemeinsamkeiten ihrer Unterklassen. Der Begriff Abstraktion wird

daher synonym zum Begriff der Oberklasse bzw. Generalisierung verwendet. In Graphiken werden Oberklassen typischerweise oberhalb ihrer Unterklassen angeordnet. Bei einer Schichtenarchitektur werden abstraktere Schichten (etwa die Anwendungsschicht des OSI-Schichtenmodells, die von der konkreten technischen Realisierung eines Kommunikationsvorgangs abstrahiert) ebenfalls oberhalb der konkreteren (technischen) Schichten angeordnet. Abstraktere Schichten werden demnach gebildet, indem neue Leistungen auf der Basis bereits vorhandener Leistungen realisiert werden, die in tieferliegenden Schichten implementiert sind. Wird mittels Vererbung eine Klasse um Funktionalität erweitert, so geschieht dies durch Unterklassenbildung. Für objektorientierte Schichtenarchitekturen folgt daraus, daß Unterklassen typischerweise in höherliegenden Schichten als ihre Oberklassen liegen (vgl. Abb. 5.8).

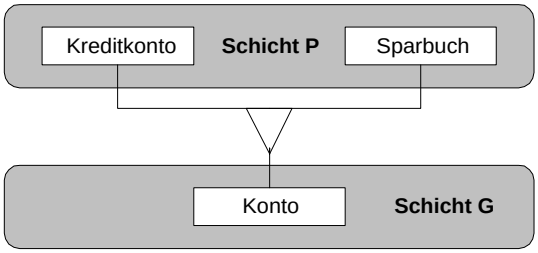


Abb. 5.8: Schichtenarchitektur und Vererbung

Bei der Darstellung einer Klassenhierarchie innerhalb einer Schichtenarchitektur muß daher eine der beiden Hierarchien horizontal gespiegelt werden (vgl. Abb. 5.9). Im weiteren werde ich jenachdem, welche der beiden Sichten im Zentrum des Interesses steht, die Graphik entsprechend ausrichten. Der Klarheit halber verwende ich den Begriff der Abstraktion nur im Zusammenhang mit Schichtenarchitekturen, bei Klassenhierarchien benutze ich die Begriffe Generalisierung und Spezialisierung. Abstrakte Klassen werden als aufgeschobene Klasse bezeichnet (vgl. [Mey88]).

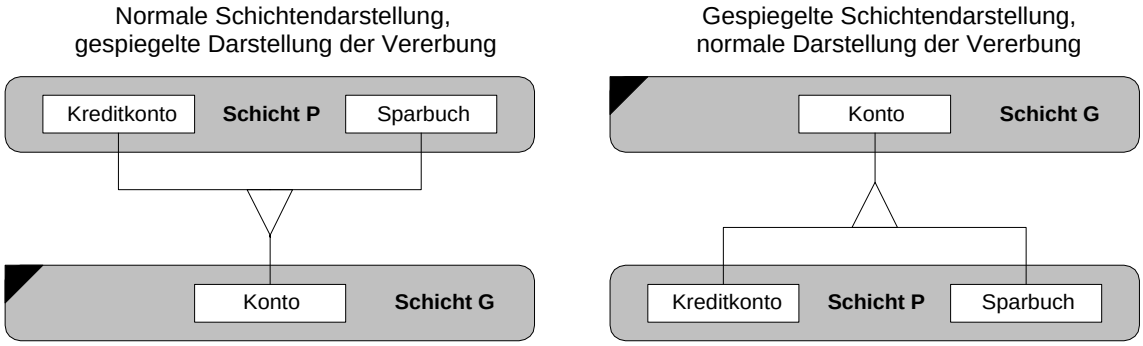


Abb. 5.9: Normale und gespiegelte Darstellung von Schichten- und Klassenhierarchien

Die beiden Begriffe »tieferliegend« und »höherliegend« beziehen sich stets auf die klassische Darstellung einer Schichtenarchitektur, bei der eine konkrete Schicht jeweils unterhalb der von ihr abstrahierenden Schichten dargestellt wird (linke Darstellung in der Abb. 5.9). Für die Bedeutung der beiden Begriffe ist dabei die graphische Anordnung der Schichten irrelevant.

Auch wenn die Darstellung gespiegelt wird, bezeichne ich im weiteren die aus Sicht von Schichtenarchitekturen konkreteren Schichten als tieferliegende Schichten. Demnach wird in der Abb. 5.9 sowohl in der linken als auch in der rechten Darstellung die Schicht G als die tieferliegende Schicht im Sinne einer Schichtenarchitektur bezeichnet. Um in einer Abbildung unabhängig von der graphischen Darstellung jederzeit die am tiefsten liegende Schicht leicht erkennen zu können, wird sie in der linken oberen Ecke mit einem schwarzen Dreieck markiert.

5.4.2 Nicht-strikte Schichtenarchitekturen und das Offen-Geschlossen-Prinzip

In der Objektorientierung kommt dem Offen-Geschlossen-Prinzip (vgl. Abschnitt 4.4.1) eine besondere Bedeutung zu. Wird das Prinzip im Kontext einer objektorientierten Schichtenarchitektur reinterpretiert, so sollte eine ganze Schicht offen für Erweiterungen, aber gleichzeitig in ihrer Verwendung geschlossen sein. Dieses ist nur mit einer nicht-strikten Schichtenarchitektur möglich, da bekanntermaßen Vererbung, Polymorphie und dynamisches Binden den Schlüssel zur Umsetzung bilden. Meyer argumentiert folgendermaßen (vgl. [Mey88]):

- Eine Klasse ist geschlossen, da sie kompiliert und in einem Rahmenwerk allgemein zur Verfügung gestellt werden kann.
- Eine Klasse ist offen, da sie als Oberklasse einer entsprechend spezialisierten Unterklasse dienen kann. Auf diese Weise können sowohl neue Operationen und Attribute hinzugefügt, als auch bereits vorhandene Merkmale redefiniert werden.

Die Schichten einer nicht-strikten objektorientierten Schichtenarchitektur sind wegen der Möglichkeit polymorpher Zuweisungen prinzipiell offen. Den Variablen in tieferliegenden Schichten können jederzeit Objekte von (Unter-) Klassen aus höherliegenden Schichten zugewiesen werden.

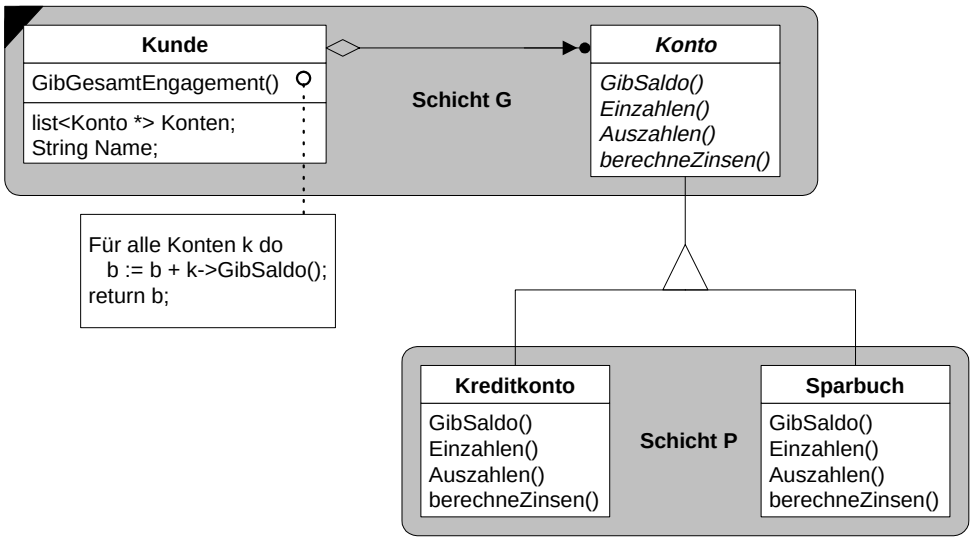


Abb. 5.10: Das Offen-Geschlossen-Prinzip am Beispiel von Kunde und Konto

Dies soll anhand des nachfolgenden Beispiels verdeutlicht werden (vgl. Abb. 5.10). Eine Klasse `Kunde` verwaltet die zu einem Kunden gehörenden Konten in einer linearen Liste. Die

Konten sind dabei vom generalisierten Typ `Konto`. Beide Klassen werden in einer Schicht G zusammengefaßt. In einer Schicht P werden die beiden konkreten Kontotypen `Kreditkonto` und `Sparbuch` implementiert. Diese sind als Unterklassen von `Konto` realisiert. Die Operation `GibGesamtEngagement` an der Klasse `Kunde` wird implementiert, indem an jedem Kontoobjekt, das in der Liste enthalten ist, die Operation `GibSaldo` aufgerufen und das Ergebnis aufsummiert wird.

Betrachtet man nun ein Objektgeflecht zur Laufzeit des Systems, so referenziert ein Kundenobjekt, das Element der Schicht G ist, über die lineare Liste Kontoobjekte vom Typ `Kreditkonto` und `Sparbuch`. Die entsprechenden Klassen sind Bestandteil der abstrakteren Schicht P (vgl. Abb. 5.11). Das Verhalten der Schicht G wird somit zur Laufzeit von Elementen beeinflußt, die in der abstrakteren Schicht P implementiert sind.

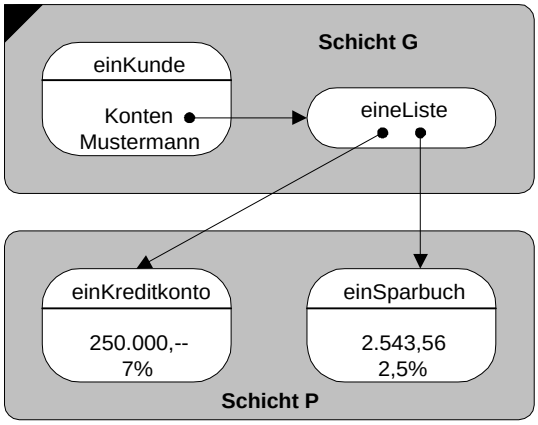


Abb. 5.11: Ein Objektgeflecht zur Laufzeit des Systems

Die Verwendung von Vererbung zur Umsetzung des Offen-Geschlossen-Prinzips setzt allerdings voraus, daß die Vererbungshierarchie eine Typhierarchie darstellt (vgl. [Lis88], [LW93], [Mar96b] und [Win92]). Eine Typhierarchie ist nach [Lis88] unter folgender Bedingung gegeben:

> "A type hierarchy is composed of subtypes and supertypes. The intuitive idea of a subtype is one whose objects provide all the behavior of objects of another type (the supertype) plus something extra. If for each object o_1 of type S there is an object o_2 of type such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T." ([Lis88], S. 25)

In dem vorgestellten Beispiel bildet die Vererbungshierarchie zwischen der Klasse `Konto` und den Klassen `Kreditkonto` sowie `Sparbuch` eine Typhierarchie. Denn an allen Stellen, an denen in der Klasse `Kunde` ein Objekt vom Typ `Konto` erwartet wird, kann ein Objekt vom Typ `Kreditkonto` oder `Sparbuch` verwendet werden.

Eine Vererbungsbeziehung, die die Typhierarchie verletzen würde, stellt die Ableitung der Klasse `Wertpapierdepot` von der aufgeschobenen Klasse `Konto` dar. Zwar kann durch die

Implementierung der Operation `GibSaldo` in Form einer aktuellen Bewertung der Aktien, die in dem Wertpapierdepot verwahrt werden, das Zusammenspiel von `Kunde` und `Konto` sehr elegant um das Konzept des Wertpapierdepots erweitert werden. Eine korrekte Implementierung der Operationen `einzahlen`, `auszahlen` und `berechneZinsen` ist in der Klasse `Wertpapierdepot` hingegen nicht möglich. Sie stellt keinen Subtyp der Klasse `Konto` dar, denn nicht an allen Stellen, an denen ein Objekt vom Typ `Konto` erwartet wird, kann ein Objekt vom Typ `Wertpapierdepot` verwendet werden.

Schichten, Rahmenwerke oder Klassen lassen sich durch den Mechanismus der Vererbung nicht beliebig an veränderte Umstände anpassen. Entsprechende Erweiterungen müssen in den Entwurfselementen explizit vorgesehen sein. Dazu bildet ein Softwareentwickler i.d.R. über die Menge der antizipierbaren Erweiterungen eine oder mehrere Generalisierungen, die er in Form von aufgeschobenen Klassen ausdrückt. Liskov schreibt hierzu:

> "A good design principle is to think about expected modifications and organize the design by using abstractions that encapsulate the changes." ([Lis88], S. 20)

Kilberth et. al. betonen, daß neben den technischen Modellierungsmitteln vor allem eine fachliche Orientierung zur Umsetzung des Offen-Geschlossen-Prinzips notwendig ist. Sie schreiben im Kontext der WAM-Methode, die auch die methodische Grundlage dieser Arbeit bildet:

> "Denn erst in Verbindung mit der hier vorgestellten methodischen Orientierung an den Konzepten der Anwendung kann sichergestellt werden, daß Klassenhierarchien fachlich und softwaretechnisch sinnvolle Abstraktionen darstellen. Erst dann ist zu erwarten, daß sich eine Klassenhierarchie als stabil herausstellt. Sonst können notwendige Modifikationen zwar über den Vererbungsmechanismus mehr oder minder elegant in den Vererbungsbaum eingebracht werden, aber mit zunehmenden Änderungen verliert diese Hierarchie an Verständlichkeit und fachlicher Klarheit." ([KGZ+94], S. 130)

Vererbung, Polymorphie und dynamisches Binden stellen somit nur eine technische Möglichkeit dar, das Offen-Geschlossen-Prinzip zu realisieren.

In objektorientierten Schichtenarchitekturen lassen sich Änderungen und Erweiterungen selbstverständlich auch durch andere Techniken realisieren. Ist eine statische Erweiterbarkeit des Softwaresystems nicht mehr ausreichend, da sie beispielsweise von dynamischen Faktoren abhängt, so werden häufig reflexive Architekturen eingesetzt (vgl. [Bus96]). Diese erlauben es, zur Laufzeit eines Systems das Verhalten und den Zustand der Objekte im System zu inspizieren und, wenn nötig, an veränderte Bedingungen anzupassen.

In diesem Zusammenhang soll noch das Interpreter-Muster (vgl. [GOF95]) erwähnt werden, das ein weiteres häufig eingesetztes Mittel zur dynamischen Erweiterbarkeit darstellt. Das Muster sieht vor, mögliche Erweiterungen in Form einer eigenen Sprache zu formulieren, die dann ein Interpreter zur Laufzeit des Systems in entsprechende Anweisungen umsetzt. Der Zweck des Muster wird in [GOF95] auch beschrieben als:

> "Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language." ([GOF95], S. 243)

Sowohl eine reflexive Architektur als auch das Interpreter-Muster setzen allerdings gleichermaßen voraus, daß Änderungen durch einen vorherigen Abstraktionsprozeß antizipiert werden. Die denkbaren Änderungen, die sich mit den beiden Techniken realisieren lassen, können sich im Gegensatz zur Subtypisierung auch auf die Architektur beziehen. Insbesondere reflexive Architekturen ermöglichen einen dynamischen Zugriff auf die Architektur des Softwaresystems, wodurch sich die in einem Rahmenwerk verankerte Architektur zur Laufzeit des Systems verändern läßt. Diese dynamische Veränderbarkeit der Architektur sollte in großen Softwaresystemen sehr sorgfältig eingesetzt werden, da die scheinbar durch das Rahmenwerk festgelegte und in der Dokumentation beschriebene statische Architektur so zur Laufzeit des Systems nicht mehr vorhanden ist.

Das Konzept, Erweiterungen in Form von Unterklassenbildung zu realisieren, läßt sich auch auf einzelne Elemente innerhalb einer Schicht anwenden. So lassen sich etwa Rahmenwerke, die Elemente einer Schicht darstellen, in ihrer Implementierung schließen und gleichzeitig für Veränderungen, die in Rahmenwerken einer höherliegenden Schichten umgesetzt werden, offen halten. Die vorausgegangen Überlegungen sollen in einem Ergebnis festgehalten werden:

Ergebnis 5.2

In nicht-strikten Schichtenarchitekturen lassen sich antizipierte Erweiterungen an einer Schicht mit Hilfe von Vererbung, Polymorphie und dynamischem Binden auch in höherliegenden Schichten realisieren.

5.4.3 Beispiele objektorientierter Schichtenarchitekturen

In der objektorientierten Literatur werden im wesentlichen zwei verschiedene Formen von Schichtenarchitekturen diskutiert, die nachfolgend vorgestellt werden. Mein Ziel dabei ist zu prüfen, inwieweit sie sich zur Realisierung der in Abschnitt 4.4 vorgestellten Makrostruktur des fachlichen Modells eignen.

Die beiden Architekturen unterscheiden sich primär in dem Abstraktionskriterium, das bei der Bildung der Schichten angewendet wird. Die erste (vgl. Abb. 5.12) unterteilt ein Softwaresystem in eine Präsentationsschicht, eine funktionale Schicht und eine Datenschicht. Die Präsentationsschicht realisiert die Benutzungsschnittstelle einer Anwendung, die funktionale Schicht implementiert anwendungsfachliche Funktionen, die auf den Objekten der Datenschicht aufsetzen. Die Datenschicht bildet gleichzeitig die Schnittstelle zu der eingesetzten persistenten Datenbasis (vgl. Kapitel 4 sowie [CD94], [CY91a] und [Tra95b]).

Im Umfeld von Client-Server Anwendungen wird diese dreischichtige Architektur auch als „three-tier-architecture“ bezeichnet. Jede der Schichten wird dabei als eigener Prozess auf separate Computer verteilt. Die Präsentationsschicht ist dabei Client der funktionalen Schicht, die selbst wiederum Client der Datenschicht ist. Ich werde diese Form der Schichten-

architektur im weiteren als Drei-Schichten-Architektur bezeichnen, unabhängig davon, ob mit ihr eine Client-Server Architektur verbunden ist oder nicht.

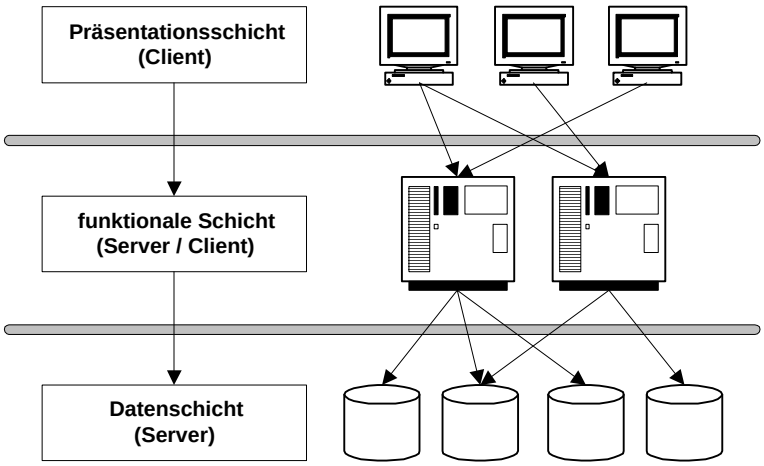


Abb. 5.12: Die Drei-Schichten-Architektur

Ein erheblicher Nachteil der Drei-Schichten-Architektur als Basis für die zu erarbeitende Makrostuktur des softwaretechnischen Modells besteht darin, daß Änderungen an der Präsentationsschicht fast immer Änderungen an der funktionalen Schicht und der Datenschicht nach sich ziehen. Denn neu zu präsentierende Informationen führen meistens zu neuen Attributen in den Datenobjekten, und fachlich motivierte Erweiterungen in der Handhabung des Systems, etwa durch neue Menüeinträge vergegenständlicht, bedingen Änderungen an der funktionalen Schicht. Im Gegenzug führen Änderungen an den Datenobjekten ebenso zu Anpassungen an der Präsentation, da neue Attribute i.d.R. auch an der Benutzungsschnittstelle dargestellt werden. So führt beispielsweise die Erweiterung der Datenschicht um die Eigenschaft, an einem Kunden seinen Quellensteuerfreibetrag führen zu können, sowohl zu fachlichen Anpassungen an der funktionalen Schicht (der Freibetrag muß fachlich verwaltet werden) als auch zu Erweiterungen der Präsentationsschicht (der Freibetrag muß dargestellt und erfaßbar sein).

Die so entstandenen drei Schichten sind in keiner Richtung unabhängig voneinander. Die unter dem Punkt *Lokalität von Änderungen* in Abschnitt 5.4.1 diskutierte Eigenschaft ist somit bei einer Drei-Schichten-Architektur nicht vorhanden. Die einzelnen Schichten können daher auch nicht getrennt voneinander entwickelt werden. Die drei Schichten zerlegen einen Anwendungsbereich demnach nicht in kleinere Bereiche, die die Eigenschaften eines Modellierungskontextes aufweisen. In [BBL+94] wird zur Drei-Schichten-Architektur ausgeführt:

> "Dies führt z.B. in objektorientierten Systemen oft dazu, daß die fachliche Komponente softwaretechnisch explizit als reine ‘Datenobjekte’ (sic!) abgebildet wird, wobei dann große Teile der fachlichen Dynamik implizit in der Oberflächenkomponente realisiert werden. Architekturen dieser Art sind schwer verständlich und kaum weiterentwickelbar." ([BBL+94], S. 109)

In [BBL+96] wird daher empfohlen, den fachlichen Kern und die Benutzungsschnittstelle zusammengehörend zu entwickeln:

> "Domain specific kernels and interactive interface parts should be developed complementarily." ([BBL+96], S. 537)

Die Drei-Schichten-Architektur kann als strikte Schichtenarchitektur realisiert werden. Vererbung zwischen den einzelnen Schichten macht im allgemeinen wenig Sinn, da die Schichten in keiner Generalisierungs- oder Spezialisierungsbeziehung zueinander stehen. Klassen der funktionalen Schicht sind keine Subtypen von Klassen der Datenschicht. Analog verhält es sich mit der Präsentationsschicht und der funktionalen Schicht.

Die zweite bekannte objektorientierte Schichtenarchitektur organisiert die zur Konstruktion eines Anwendungssystems verwendeten Komponenten (Rahmenwerke) in logische Schichten (vgl. [And95], [Wei94] und [WG95]). Die Schichtenarchitektur von ET++, als ein Beispiel einer derartigen Architektur, ist in Abb. 5.13 dargestellt. Die logischen Schichten existieren nicht zwangsläufig als physische Einheiten mit eigener Schnittstelle, etwa als statische oder dynamische Bibliothek. Mit der logischen Schichtenbildung ist gleichzeitig eine hierarchische Struktur verbunden, bei der höherliegende Schichten einen höheren Abstraktionsgrad aufweisen als darunterliegende Schichten.

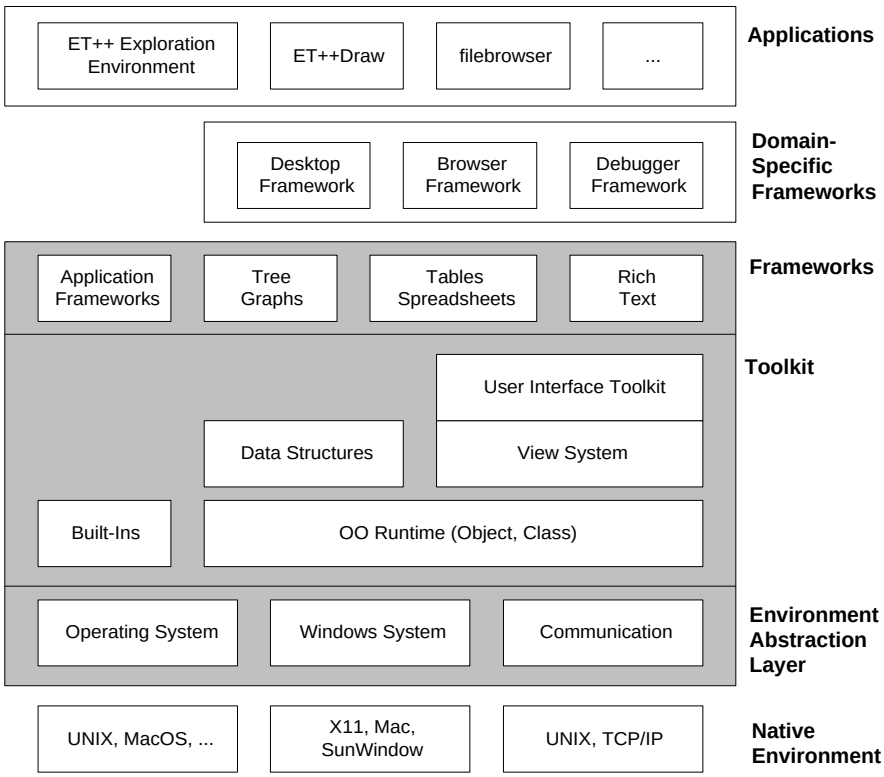


Abb. 5.13: Die Schichtenarchitektur von ET++ (aus [WG95], S. 158)

Logisch geschichtete Architekturen werden i.d.R. als durchlässige, nicht-strikte Schichtenarchitekturen verwirklicht. Klassen verwenden demnach typischerweise nicht nur Klassen der direkt untergeordneten Schicht. In dem in Abb. 5.13 dargestellten Beispiel können alle

Klassen, die in einer der Toolkit-Schicht übergeordneten Schicht liegen, die darin enthaltenen Container-Klassen (Data Structures) verwenden. Deshalb kann auch nicht ausgeschlossen werden, daß Änderungen an einer Schicht Auswirkungen auf eine Reihe anderer Schichten haben. So kann beispielsweise eine Änderung an der Toolkit-Schicht (siehe Abb. 5.13) zu entsprechenden Anpassungen in der Framework-Schicht führen, die wiederum weitere Anpassungen in der Domain-Specific-Framework-Schicht nach sich ziehen. Festgelegt ist indes, daß Klassen einer tieferliegenden Schicht keine Klassen einer höherliegenden Schicht verwenden dürfen.

Die nachfolgende Tabelle faßt die Diskussion über die beiden vorgestellten Schichtenarchitekturen zusammen. Die in der Tabelle verwendeten Symbole - und + weisen die folgende Bedeutung auf:

- -: das Merkmal wird von der vorgestellten Schichtenarchitektur nicht erfüllt.
- +: das Merkmal wird erfüllt.

Merkmal / Architektur	Durchlässigkeit	Lokalität von Änderungen	Striktheit (Offen-Geschlossen-Prinzip)
Drei-Schichten-Architektur	-	-	-
logische Schichtenbildung	+	-	+

5.4.4 Ein objektorientierter Schichtenarchitekturstil

Die diskutierten Schichtenarchitekturen zeigen, daß Schichten sowohl als ausführbare Komponenten mit eigener Schnittstelle realisierbar sind als auch zur logischen Gruppierung (wieder)verwendbarer Einheiten eingesetzt werden. Definieren Schichten eine eigene Schnittstelle, mit der sie ihre Leistung höherliegenden Schichten zur Verfügung stellen, bleibt der innere Aufbau dieser Schichten für Klienten verborgen (opak). Ein Beispiel hierfür bilden die Implementierungen des OSI-Schichtenmodells.

In Begriff 2.12 habe ich eine Komponente definiert als benannte, (wieder)verwendbare Einheit eines Softwaresystems, die ihre Dienste in Form von Schnittstellen zur Verfügung stellt. Diese Definition stimmt mit der Sichtweise von Shaw & Garlan überein. Sie schreiben in [SG96]:

> "The framework we will adopt is to treat an architecture of a specific system as a collection of computational components – or simply components – together with a description of the interactions among these components – the connectors." ([SG96], S. 20)

Dieser Komponentenbegriff wurde somit zu eng festgelegt, um darunter auch nicht ausführbare, logische Schichten ohne eigene Schnittstelle fassen zu können. Denn diese können nicht direkt als „computational components“ verwendet werden. Klienten einer logischen Schicht benutzen vielmehr die in ihr gruppierten Komponenten. Es kann demzufolge auch keine Kopplung logischer Schichten geben, sondern nur eine Kopplung der von den Schichten zur

Verfügung gestellten Komponenten. Die Kopplung basiert folglich auf Verbindungsstücken, die für den Architekturstil charakteristisch sind, nach dem die gruppierten Komponenten realisiert sind. Werden die Komponenten einer Schicht beispielsweise mittels eines rahmenwerkbasierten Architekturstils realisiert, so sind die Komponenten einer Schicht Klassenbibliotheken und Rahmenwerke (vgl. Abb. 5.4). Um sie über Schichten hinweg zu koppeln, werden die Verbindungsstücke Entwurfsmuster, Spezialisierung und Benutzt-Beziehung (vgl. Abb. 5.7) eingesetzt.

Da eine logische Schichtenarchitektur in vielen objektorientierten Systemen als die zugrundeliegende Architektur angegeben wird (vgl. [And95], [Vli95] und [WG95]), führe ich den Begriff der logischen Schichtenarchitektur ein. Dieser erweitert das bisher vorgestellte Verständnis von Komponente und Verbindungsstück.

Begriff 5.8 Logische Schichtenarchitektur

Eine logische Schichtenarchitektur gruppiert Komponenten in schnittstellenlosen Schichten, die ebenfalls als Komponenten bezeichnet werden. Logische Schichten sind hierarchisch angeordnet, benannt und anhand eines Abstraktionskriteriums gebildet. Die Kopplung der Komponenten einer Schicht basiert auf den Verbindungsstücken, die für den Architekturstil dieser Komponenten charakteristisch sind. Eine logische Schichtenarchitektur definiert keine eigenen Verbindungsstücke. Sie sollte aber solche Verbindungsstücke explizit machen, die sich besonders zur Kopplung von Komponenten über Schichten hinweg eignen. Die Unterteilung einer Schicht in weitere logische Einheiten ist zulässig.

In den Kapiteln 2, 3 und 4 habe ich eine Reihe von Anforderungen erarbeitet, die das softwaretechnische Modell auf der Ebene der Makrostruktur erfüllen sollte. Im weiteren werde ich einen objektorientierten Schichtenarchitekturstil ausarbeiten, der sich an diesen Anforderungen orientiert. Zusammenfassend sind dies:

(1) Die zur Konstruktion eines Softwaresystems verwendete Technik zerfällt in eine Systembasis und in ein Modell der verwendeten Technik. Dieses wurde mit den Kontexten »Leitbilder und Entwurfsmetaphern« sowie »Softwarearchitekturen und Entwurfsmuster« unter dem Begriff Technologie zusammengefaßt. Ein Architekturstil muß demnach die beiden Bereiche Systembasis und Technologie geeignet abbilden können.

(2) Die erarbeitete Makrostruktur des fachlichen Modells, bestehend aus Gegenstandsbereich, Produktbereichen und Einsatzkontexten, muß sich auf das softwaretechnische Modell übertragen lassen. Denn nur so kann die Struktur des softwaretechnischen Modells an den Strukturen des Anwendungsbereichs und der verwendeten Technologie ausgerichtet werden.

(3) Der Schichtenarchitekturstil muß es erlauben, eine technische Basis umzusetzen, die die technische Kombinierbarkeit und Integrierbarkeit der in den Produktbereichen entwickelten Komponenten sicherstellt.

(4) Der Gegenstandsbereich und die technische Basis müssen sich flexibel erweitern lassen (vgl. Offen-Geschlossen-Prinzip).

(5) Die einzelnen Schichten sollen mit Hilfe des rahmenwerkbasierten Architekturstils realisiert werden.

Die Drei-Schichten-Architektur kann diesen Anforderungen offensichtlich nicht gerecht werden. Die für die Makroebene vorgeschlagene Trennung von Funktionalität und Präsentation hat sich nichtsdestotrotz als Organisationskriterium für Mikrostrukturen sehr bewährt. Die Trennung stellt beispielsweise ein Grundprinzip bei der Konstruktion interaktiver graphischer Anwendungssysteme dar. Beispiele hierfür finden sich im SMALLTALK-80 System (MVC-Konzept), im ET++ Rahmenwerk (vgl. [WG95]), in Taligents CommonPoint (vgl. [And95]) und im GEBOS-Rahmenwerk (vgl. [BBS+95] und [BKG+95]). Die Trennung auf Mikroebene wird daher auch nicht kritisiert.

Entlang der in 5.1.1 beschriebenen Charakteristika von Architekturstilen entwickle ich im folgenden einen eigenen objektorientierten Schichtenarchitekturstil. Die Strukturierung des softwaretechnischen Modells als *durchlässige nicht-strikte logische Schichtenarchitektur* ermöglicht es, die gestellten Anforderungen zu verwirklichen. Denn die Systembasis, die Technologie und die Struktur des fachlichen Modells können in jeweils eigenen logischen Schichten abgebildet werden. Die in den Schichten zusammengefaßten Komponenten lassen sich dabei mit Hilfe des rahmenwerkbasierten Architekturstils realisieren. Die Nicht-Striktheit bildet hierbei die Voraussetzung, Schichten nach dem Offen-Geschlossen-Prinzip erweitern zu können. Das Vokabular für die Komponenten und Verbindungsstücke stellt sich wie folgt dar:

- *Komponenten*: Als Komponenten werden die Schichten Systembasis, Technologie, Gegenstandsbereich, Produktbereich und Einsatzkontext definiert. Die Komponenten bilden logische Schichten, in denen Klassenbibliotheken, White-Box und Black-Box Rahmenwerke zusammengehörend gruppiert werden. Als graphisches Symbol für eine Schicht wird ein abgerundetes Rechteck verwendet. Um die untere Schicht leicht erkennen zu können, wird sie mit einem schwarzen Dreieck markiert (vgl. Abb. 5.9).
- *Verbindungsstücke*: Neue Verbindungsstücke werden nicht definiert (vgl. Begriff 5.8). Die Kopplung der Komponenten erfolgt mittels der für den rahmenwerkbasierten Architekturstil bereits festgelegten Verbindungsstücke Benutzt-Beziehung, Spezialisierung und Entwurfsmuster. Neben einer Reihe bekannter Entwurfsmuster wie etwa Dekorierer, Besucher, Zuständigkeitskette, „Separation of Powers“ und Serialisierer (vgl. [GOF95], [RZ95] und [RSB+97]) haben sich insbesondere zwei neue Entwurfsmuster zur schichtenübergreifenden Kopplung von Komponenten im GEBOS-Kontext bewährt: das Rollenmuster und der Produkthändler. Beide werden ausführlich in Kapitel 6 diskutiert.

Im weiteren werden die einzelnen Schichten erklärt und an Beispielen veranschaulicht. Die Beispiele beziehen sich auf die Abb. 5.14, welche die im RWG-Kontext entstandenen Rahmenwerke präsentiert. Um die mit den beiden Schichten *Technologie* und *Gegenstandsbereich* verbundene Aufgabe, die fachliche und technische Integrierbarkeit der Komponenten sicherzustellen, besser hervorzuheben, bilden diese Schichten in Abb. 5.14 eine U-Form. Das Schichtenprinzip einer logischen Schichtenarchitektur wird dadurch nicht verletzt.

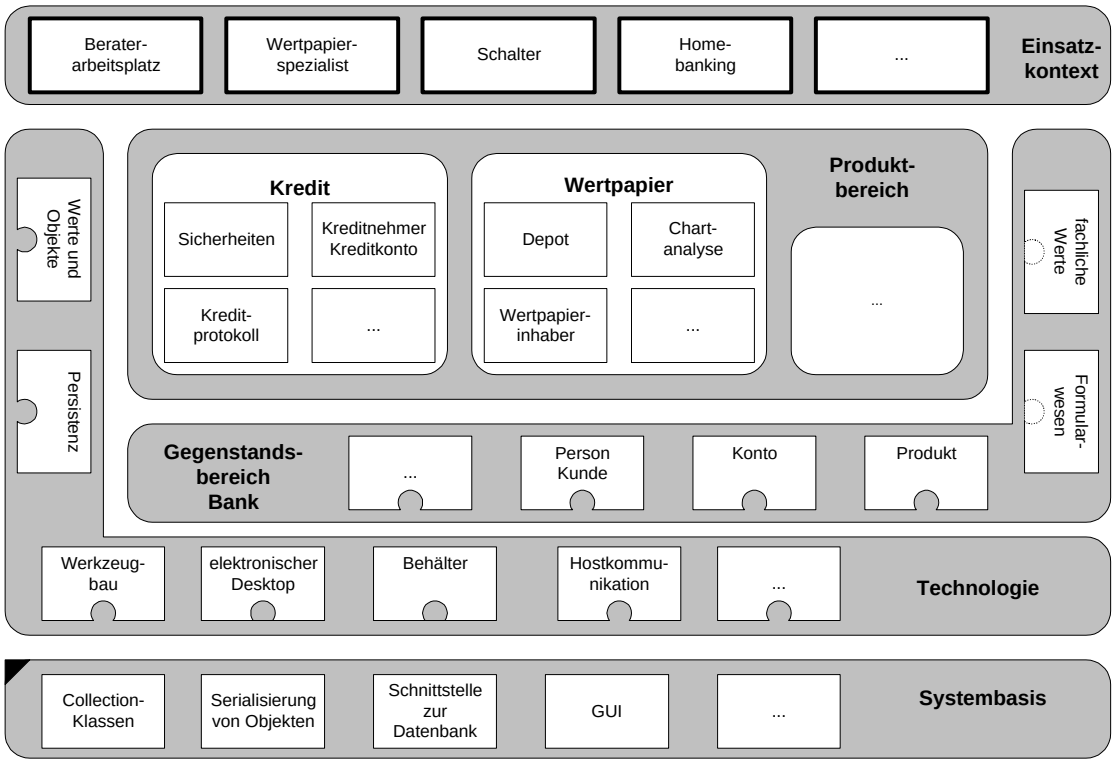


Abb. 5.14: Der Schichtenarchitekturstil am Beispiel von GEBOS

Schicht: Systembasis

Diese Schicht realisiert die in Abschnitt 2.5 beschriebene Systembasis, die aus der zur Entwicklung eines Anwendungssystems verwendeten Technik herausgelöst wurde. Die Systembasisschicht umfaßt hauptsächlich Black-Box Rahmenwerke und Klassenbibliotheken, die sich in zwei Gruppen einteilen lassen. Die erste Gruppe umfaßt solche Rahmenwerke und Klassenbibliotheken, die allgemein benötigte (objektorientierte) Konzepte bereitstellen. Hierzu zählen Meta-Objekt-Protokolle (vgl. [Bus96]), Container-Klassen, Garbage-Kollektoren und Serialisierungsmechanismen zur Transformation von Objektgeflechten in und aus flachen Strukturen (vgl. [RSB+96]). Die zweite Gruppe kapselt das zugrundeliegende Betriebs- und Fenstersystem, eventuell verwendete Middlewareprodukte und persistente Datenspeicher, wie etwa relationale Datenbanken und elektronische Archive. Beide Gruppen sind Bestandteil dieser Schicht, da sie Schnittstellen zur verwendeten Technik bereitstellen. Der Serialisierungsmechanismus und die Schnittstelle zu einer relationalen Datenbank stellen zum Beispiel nur die technische Möglichkeit zur Verfügung, Objekte persistent zu machen. Eine Verbindung zwischen diesen beiden Komponenten besteht in dieser Schicht nicht. Die Komponente, die das Protokoll festlegt, mit dem Objekte in relationale Datenbanken gespeichert werden, muß, da es das verwendete Technikmodell realisiert (vgl. Abschnitt 2.5), Bestandteil der nachfolgend definierten Technologieschicht sein.

Die Rahmenwerke und Klassenbibliotheken der Systembasisschicht sind vollkommen unabhängig von einem speziellen Anwendungsbereich. Sie können daher zur Konstruktion verschiedenster interaktiver, graphischer Anwendungssysteme eingesetzt werden. Benutzt werden die Komponenten dieser Schicht von allen höherliegenden Schichten.

Schicht: Technologie

Komponenten der Technologieschicht vergegenständlichen die bei der Softwareentwicklung eingesetzten Modelle über die verwendete Technik und das zugrunde liegende Leitbild mit den dazu passenden Entwurfsmetaphern (vgl. Abschnitt 2.5). Die Schicht bildet die technologische Basis der darauf konstruierten Anwendungssysteme.

Die Technologieschicht besteht größtenteils aus White-Box Rahmenwerken, die vereinzelt um offen verwendbare Black-Box Rahmenwerke ergänzt werden. Hierzu zählen typischerweise Rahmenwerke zur Konstruktion und Integration von interaktiven Werkzeugen auf einem elektronischen Schreibtisch (vgl. [BKG+95]) sowie Rahmenwerke, die den Umgang mit anwendungsfachlichen Behältern (Ablagen, Mappen und Stapeln) zur Organisation des Desktops definieren. Diese Komponenten realisieren somit das in den Anwendungssystemen gewünschte Modell von einem elektronischen Schreibtisch. Darüberhinaus werden Rahmenwerke bereitgestellt, die generische Werkzeuge (z.B. Browser) sowie Protokolle und Standardimplementierungen für das Speichern und Laden von Objekten in bzw. aus einer relationalen Datenbank implementieren. Das entsprechende Persistenz-Rahmenwerk vergegenständlicht dabei ein Modell über das Zusammenspiel von Objekten mit einer relationalen Datenbank und wird auf der Basis der elementaren Serialisierungsmechanismen und der Datenbankschnittstelle, die beide in der Systembasisschicht angesiedelt sind, implementiert (vgl. Beispiel in der Schicht Systembasis).

Die Rahmenwerke dieser Schicht bilden die Grundlage für die Konstruktion einer Familie von Anwendungssystemen, denen ein gemeinsames Modell der verwendeten Technik zugrundeliegt. Die Technologieschicht von GEBOS eignet sich etwa zur Konstruktion interaktiver Anwendungssysteme für den Dienstleistungssektor (nicht beschränkt auf Banken).

Schicht: Gegenstandsbereich

Diese Schicht bildet das Konzept des in Abschnitt 4.4.1 eingeführten Gegenstandsbereichs ab und wird auf der Basis der Systembasis- und Technologieschicht gebildet. Sie stellt die fachliche Basis für die auf ihr konstruierten Rahmenwerke und Anwendungssysteme bereit. Die Komponenten dieser Schicht ermöglichen es nicht nur, bewährte Analysen, Architekturen und Implementierungen wiederzuverwenden, sondern haben vor allem die Aufgabe, die fachliche Integrationsfähigkeit der in den Produktbereichen entwickelten Rahmenwerke sicherzustellen. Die Gegenstandsbereichsschicht besteht größtenteils aus White-Box Rahmenwerken, die vereinzelt um offen verwendbare Black-Box Rahmenwerke ergänzt werden.

In GEBOS generalisiert diese Schicht in White-Box Rahmenwerken die abstrakten fachlichen Kernkonzepte, wie etwa Person, Kunde, Produkt oder Sicherheit (vgl. Abschnitt 4.5 und Abb. 4.10), die in allen Produktbereichen benötigt werden, dort aber noch zu spezialisieren sind. Zum anderen stellt sie in offen verwendbaren Black-Box Rahmenwerken direkt instanziierbare Klassen zur Verfügung, die sich ebenfalls übergreifend verwenden lassen. Ein typisches Beispiel hierfür sind die in Abschnitt 4.5 diskutierten bankfachlichen Werte.

Da die Rahmenwerke dieser Schicht ein Modell vergegenständlichen, das spezifisch für ein bestimmtes Geschäftsfeld ist, lassen sie sich auch nur zur Implementierung von Anwendungs-

systemen in eben diesem Geschäftsfeld wiederverwenden (im Beispiel von GEBOS das Bankgeschäft der von der RWG betreuten Volks- und Raiffeisenbanken).

Schicht: Produktbereich

Die Produktbereichsschicht bildet das bereits im Anwendungsbereich in verschiedene Produktbereiche unterteilte Geschäftsfeld ab. Jeder Produktbereich entspricht einer vertikalen Partitionierung der Schicht. Jede Partition besteht vorwiegend aus Black-Box Rahmenwerken, die auf der Basis der in den tieferliegenden Schichten konstruierten White-Box und Black-Box Rahmenwerke erstellt werden. Die Komponenten dieser Schicht implementieren die für den Produktbereich relevanten Geschäftsobjekte, bespielsweise Kreditnehmer, Kreditkonto und Sicherheit für den Produktbereich Kredit, und die dazugehörigen Geschäftsprozesse, wie die Vergabe eines Baufinanzierungskredits. Die wechselseitige Verwendung von Rahmenwerken aus verschiedenen Partitionen ist nicht zulässig. Denn damit werden unerwünschte Abhängigkeiten in die ansonsten voneinander unabhängigen Produktbereiche (vgl. Begriff 4.2) hineinmodelliert. Neue White-Box Rahmenwerke entstehen lediglich, wenn sie zur Konstruktion anderer Rahmenwerke desselben Produktbereichs (d.h. derselben Partition) wiederverwendet werden können.

Die Rahmenwerke der Produktbereichsschicht konkretisieren und erweitern über Spezialisierung und Entwurfsmuster verschiedene White-Box Rahmenwerke der beiden tieferliegenden Schichten Technologie und Gegenstandsbereich. Bestes Beispiel hierfür ist das Black-Box Rahmenwerk für Kreditnehmer/Kreditkonto (vgl. Abb. 5.15). Es konkretisiert

- das Person/Kunde- sowie das Konto-Rahmenwerk aus dem Gegenstandsbereich. Die Konkretisierung erfolgt beim Person/Kunde-Rahmenwerk mittels des Rollen-Musters (vgl. Abschnitt 6.3) und beim Konto-Rahmenwerk durch Spezialisierung.
- das Werkzeug- und das Persistenz-Rahmenwerk aus dem Technologiebereich. Auf Basis der beiden Rahmenwerke werden im Kreditnehmer/Kreditkonto-Rahmenwerk interaktive Werkzeuge für die Materialien `Kreditnehmer` und `Kreditkonto` sowie ein Automat zum Speichern und Laden dieser Materialien konstruiert. Die Konkretisierung findet anhand der zwei Entwurfsmuster „separation of powers“ und „Serialisierer“ statt (vgl. [RZ95], [Rie95a], [RSB+97]).

Schicht: Einsatzkontext

Diese Schicht bildet die verschiedenen Einsatzkontexte eines Geschäftsfeldes ab und ist daher ebenfalls vertikal partitioniert. Jede Partition entspricht dabei einem Einsatzkontext. Die Konstruktion von Anwendungssystemen in einem Einsatzkontext erfolgt auf der Basis der Black-Box Rahmenwerke, die in den verschiedenen Produktbereichen entwickelt wurden. Beispiele für Einsatzkontexte in einer Bank sind der Berater- und der Wertpapierarbeitsplatz (vgl. Abschnitt 4.3.2). Neue Rahmenwerke werden für einen Einsatzkontext nur entwickelt, wenn er besondere technologische oder fachliche Anforderungen aufweist. Ein Beispiel hierfür wäre die Entwicklung eines Rahmenwerks für die Verwendung von Touch-Screens an Selbstbedienungsterminals.

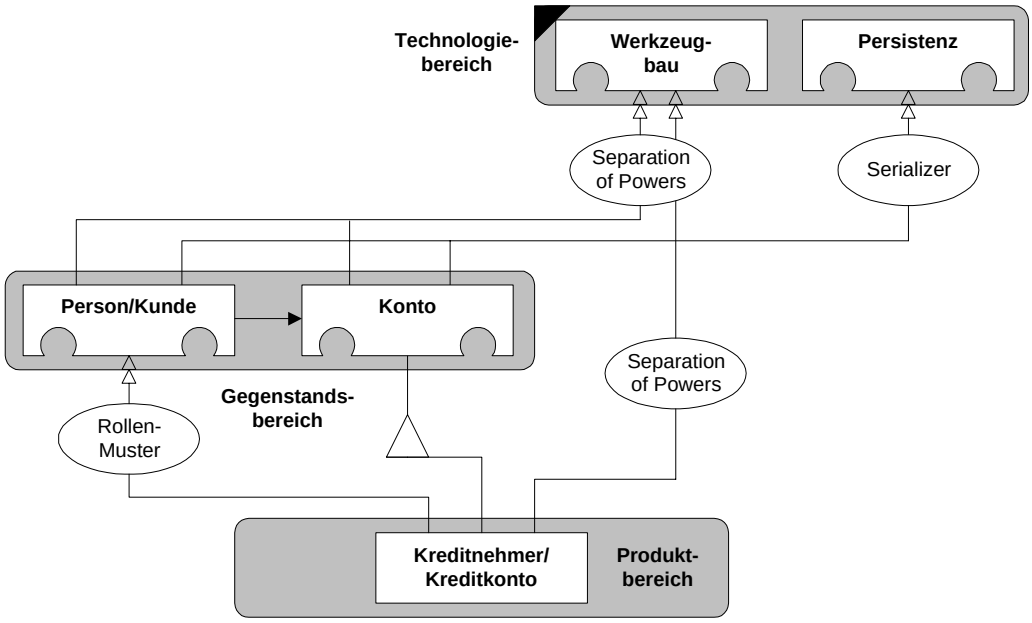


Abb. 5.15: Das Black-Box Rahmenwerk für Kreditnehmer/Kreditkonto

Aufbauend auf den fünf logischen Schichten (Komponenten), die durch den Architekturstil definiert wurden, lassen sich nun einige Regeln des rahmenwerkbasierten Architekturstils konkreter fassen. Die Regeln beziehen sich auf die erlaubten Beziehungen zwischen Komponenten (Klassenbibliotheken, Black-Box und White-Box Rahmenwerke) aus unterschiedlichen Schichten und sind wie folgt festgelegt:

- Komponenten einer Schicht dürfen ausschließlich Komponenten derselben oder einer beliebigen darunterliegenden Schicht verwenden.
- Die Komponenten eines Produktbereichs dürfen keine Komponente eines anderen Produktbereichs verwenden.
- Klassen eines Produktbereichs dürfen nicht gleichzeitig (z.B. mittels Mehrfachvererbung) von aufgeschobenen Klassen aus dem Gegenstands- und Technologiebereich abgeleitet sein. Dadurch wird sichergestellt, daß Klassen entweder technisch oder fachlich motiviert sind. Damit wird nicht ausgeschlossen, daß Komponenten der Produktbereichsschicht gleichzeitig Komponenten des Gegenstandsbereichs und des Technologiebereichs benutzen (vgl. Abb. 5.15).

Der rahmenwerkbasierte Architekturstil definiert bekanntermaßen keine eigene Semantik für kombinierte Entwurfselemente (vgl. Abschnitt 5.3). Da der Schichtenarchitekturstil lediglich eine logische Gruppierung der Komponenten des rahmenwerkbasierten Architekturstils vornimmt und keine Kombination von Schichten zu neuen Schichten unterstützt wird, kann auch hier keine spezielle Semantik angegeben werden. Die an der beschriebenen Schichtenarchitektur durchführbaren Messungen und Analysen sind durch die oben aufgeführten Regel vorgegeben. Die Architekturbeschreibungssprache wird um das Symbol für eine logische Schicht erweitert.

Die Konstruktion der Rahmenwerke in den verschiedenen Produktbereichen basiert auf den Rahmenwerken der Technologie- und Gegenstandsbereichsschicht. Sie bilden die technische respektive fachliche Basis und stellen daher die technische und fachliche Kombinierbarkeit und Integrierbarkeit bei der Entwicklung von Anwendungssystemen in den unterschiedlichen Einsatzkontexten sicher. In beiden Bereichen werden die entsprechenden technischen und fachlichen Grundkonzepte und die Möglichkeiten ihrer Erweiterbarkeit im Rahmen des Offen-Geschlossen-Prinzips definiert. So garantiert das White-Box Rahmenwerk für den Werkzeugbau aus dem Technologiebereich (vgl. Abb. 5.15), daß sich alle Werkzeuge in einen elektronischen Schreibtisch eines Anwendungssystems einfügen lassen. Analog sorgt das Persistenz-Rahmenwerk für eine gleichartige Speicherung von Objekten in einer relationalen Datenbank. Dadurch kann jedes Anwendungssystem auf die dort gespeicherten Objekte zugreifen.

Die beiden White-Box Rahmenwerke Person/Kunde und Konto aus dem Gegenstandsbereich bilden die fachliche Basis für die Konstruktion des Kreditnehmer/Kreditkonto-Rahmenwerks (vgl. Abschnitt 4.4.1). Neu ist allerdings, daß sich die White-Box Rahmenwerke des Gegenstandsbereichs auf bereits vorhandene Rahmenwerke des Technologiebereichs abstützen. Diese Schichtung der Rahmenwerke garantiert die technische Kombinierbarkeit und Integrierbarkeit von Rahmenwerken des Produktbereichs, die sich ausschließlich auf Rahmenwerken des Gegenstandsbereichs abstützen. Rahmenwerke des Gegenstandsbereichs können zudem fachliche Konzepte in eine gemeinsame technische Basis einbetten und so deren einheitliche technische Verwendung garantieren. In dem in Abb. 5.15 dargestellten Beispiel konkretisieren bereits das Person/Kunde- und Konto-Rahmenwerk des Gegenstandsbereichs die Transformation der entsprechenden Objekte in eine relationale Datenbank. Eine weitere Spezialisierung muß im Kreditnehmer/Kreditkonto-Rahmenwerk des Produktbereichs nicht vorgenommen werden.

Die Vorteile, die mit der vorgestellten objektorientierten Schichtenarchitektur verbunden sind, ergeben sich zum größten Teil aus den in Kapitel 4 und auf der Seite 112 dieses Abschnitts formulierten Anforderungen. Ihre wichtigsten Vorteile sind:

- die bruchlose Übertragbarkeit von anwendungsfachlichen und technologischen Makrostrukturen auf das softwaretechnische Modell. Der Schichtenarchitekturstil kann daher zur Konstruktion von objektorientierten Softwaresystemen verwendet werden, deren Anwendungsbereich nach dem Produkt- oder Prozeßprinzip organisiert ist (vgl. Ergebnis 4.1). Der homologe Aufbau der Modelle und Kontexte erleichtert zudem die Etablierung von Codes im Entwicklungsprozeß.
- die getrennte Modellierbarkeit der Produktbereiche. Dies ermöglicht eine Zerlegung des Anwendungsbereichs, die die evolutionäre Entwicklung von großen Anwendungssystemen unterstützt.
- die fachliche und technische Kombinierbarkeit der Komponenten. Dies erleichtert die Konfektionierung von Anwendungssystemen aus Komponenten unterschiedlicher Produktbereiche. Damit kann flexibel auf neue oder geänderte Anforderungen reagiert werden.
- die saubere Trennung zwischen der verwendeten Technik und dem Modell der verwendeten Technik in Form der beiden Schichten Systembasis und Technologie. Dies ermöglicht es, Softwaresysteme, die nach diesem Architekturstil konstruiert worden

sind, auf unterschiedliche Systemplattformen zu portieren. Außerdem ist die einheitliche Verwendung eines Technikmodells innerhalb einer Familie von Anwendungssystemen gewährleistet.

Die Vorstellung der Schichtenarchitektur, insbesondere die beispielhafte Präsentation von Komponenten aus dem GEBOS-Kontext für die einzelnen Schichten, macht einen weiteren interessanten Sachverhalt deutlich: je „besser“ eine Komponente in einen gegebenen Kontext integriert werden soll, desto ausgeprägter sind die architekturellen Vorgaben, die an die Konstruktion der Komponente gestellt werden. So muß eine Komponente, die sowohl fachlich als auch technisch integrierbar ist, die Architekturvorgaben der Schichten Technologie und Gegenstandsbereich erfüllen. Eine Komponente, die nur technisch zu integrieren ist, muß dagegen nur die Vorgaben der Technologieschicht einhalten.

	Klassen	Black-Box	Off. ver. Black-Box	White-Box	Geschl. ver. White-Box
Einsatzkontexte	242	3	0	1	1
Schalter	*3*	*0*	*0*	*0*	*0*
Anlage	*3*	*0*	*0*	*0*	*0*
Kredit	*23*	*0*	*0*	*0*	*0*
Wertpapier	*20*	*0*	*0*	*0*	*0*
Telefonbanking	*3*	*0*	*0*	*0*	*0*
Selbstbedienung	*92*	*2*	*0*	*1*	*0*
Administration	*17*	*0*	*0*	*0*	*0*
Technik	*8*	*0*	*0*	*0*	*0*
DB-Technik	*73*	*1*	*0*	*0*	*1*
Produktbereiche	1511	35	2	7	1
Schalter	*256*	*11*	*0*	*2*	*0*
Anlage	*273*	*6*	*1*	*0*	*0*
Kredit	*430*	*8*	*0*	*2*	*1*
Wertpapier	*552*	*10*	*1*	*3*	*0*
Gegenstandsbereich	453	4	2	10	3
Technologiebereich	445	2	3	8	6
Systembasis	201	7	0	1	0

Abb. 5.16: Zahlen aus dem GEBOS-System

Die Tabelle der Abb. 5.16 präsentiert Zahlen aus dem GEBOS-System, die entsprechend der vorgestellten Architekturstile aufbereitet sind. Leider ist eine saubere Strukturierung auf der Basis der in diesem Kapitel vorgestellten Architekturstile bis heute für das GEBOS-System nur für die Schichten Technologie (die auch die Schicht Systembasis enthält), Gegenstandsbereich sowie für einige Produktbereiche und Einsatzkontexte durchgeführt worden. Die Zuordnung der restlichen Klassen und Rahmenwerke wurde von mir daher auf den Erkenntnissen dieser Arbeit und bestehender Zuordnungen vorgenommen.

5.4.5 Ein Vergleich mit der Schichtenarchitektur von ET++

Sowohl der objektorientierte Schichtenarchitekturstil (vgl. Abb. 5.14) als auch die ET++ Schichtenarchitektur (vgl. Abb. 5.13) organisieren die Komponenten eines Gesamtsystems in insgesamt fünf Schichten. Da die zur Bezeichnung der einzelnen Schichten gewählten Namen gewisse Ähnlichkeiten aufweisen, mag die Vermutung naheliegen, daß auch die Kriterien zu ihrer Bildung ähnlich sind. So könnte die Systembasis dem Environment Abstraction Layer, die Technologieschicht dem Toolkit Layer, der Gegenstandsbereich dem Framework Layer, die Produktbereichsschicht dem Domain Specific Framework Layer und die Einsatzkontextschicht dem Application Layer entsprechen. Im weiteren soll aufgezeigt werden, in welchen Bereichen die Bedeutung der Schichten voneinander abweicht und inwieweit sie sich decken.

Ein wesentlicher Unterschied zwischen dem objektorientierten Architekturstil und der Schichtenarchitektur von ET++ besteht in ihren voneinander abweichenden Zielsetzungen. Dies wirkt sich insbesondere auf die drei Schichten Toolkit (Technologie), Frameworks (Gegenstandsbereich) und Domain Specific Frameworks (Produktbereiche) aus.

Der Domain Specific Frameworks Layer von ET++ ist insbesondere nicht darauf ausgerichtet, das fachliche und softwaretechnische Modell in unabhängige Bereiche, die die Eigenschaften eines Modellierungskontextes aufweisen, zu strukturieren. Es existiert daher weder eine Partitionierung dieser Schicht in eigenständige Modellierungskontexte noch sind Regeln definiert, inwieweit die Rahmenwerke dieser Schicht miteinander gekoppelt sein dürfen.

Um Toolkit Layer und Framework Layer besser mit den Schichten des objektorientierten Schichtenarchitekturstils vergleichen zu können, wird nachfolgend ihre Definition gemäß [WG95] angegeben.

Toolkit-Layer:

> "The toolkit layer contains the most important low level building blocks of the ET++ class hierarchy: the class `Object`, the root of the overall class hierarchy, implements the common behavior for all ET++ classes. Data Structures are general purpose classes like arrays, lists, sets etc. […]. The *View System* contains a small number of graphical building blocks which implement the common behavior of all graphical classes and defines a framework to easily build new components from existing ones. The *User Interface Toolkit* contains all the graphical and interactive components found in almost every user interface toolbox, such as menus, dialogs, or scrollbars." ([WG95], S. 158)

Framework-Layer:

> "The Framework layer contains a small number of high level frameworks. The Application Framework classes are high level abstract classes that factor out the common control structure of applications running in a graphic environment. They define the abstract model of a typical ET++ application and together form a generic ET++ application. The other frameworks provide high level and extensible

> building blocks for rich text, tables and spreadsheets, and trees and graphs." ([WG95], S. 159)

Die Definition des Toolkit Layers entspricht weitgehend der Definition der Systembasis-Schicht, die des Framework Layers der des Technologiebereichs. Dies liegt in der Tatsache begründet, daß der Anwendungsbereich von ET++ vor allem technisch geprägt ist. Neben der Konstruktion von Rahmenwerken für die Entwicklung von Editoren (ET steht für Editor Toolkit) ging es auch um die Beantwortung technologischer Fragestellungen bei der Entwicklung graphischer Fenstersysteme oder objektorientierter Softwareentwicklungsumgebungen. Der Gegenstandsbereich von ET++ ist somit Technik. Demzufolge stellt der Toolkit Layer die Technologie zur Verfügung, mit deren Hilfe der technikorientierte Gegenstandsbereich, abgebildet im Framework Layer, konstruiert werden kann. Dies erklärt, warum bei ET++ viele der technischen Rahmenwerke eine Schicht höher liegen, als in dem von mir vorgestellten objektorientierten Schichtenarchitekturstil. So sind etwa das Data Structure- und das OO-Runtime Framework in ET++ Bestandteil des Toolkit Layers, während die entsprechenden GEBOS-Rahmenwerke in der Systembasisschicht liegen. Ein weiteres Beispiel stellt das Application Framework dar, das dem Rahmenwerk Elektronischer Schreibtisch im GEBOS-System entspricht. In ET++ liegt dieses im Framework Layer, im GEBOS-System dagegen bereits in der Technologieschicht.

Der Anspruch, eine Familie integrierter Anwendungssysteme zu entwickeln, hat bei dem objektorientierten Schichtenarchitekturstil erstens zu einer sauberen Trennung von Technik, Technologie und anwendungsfachlichen Bereichen sowie zweitens zur vertikalen Partitionierung der Produktbereichs- und der Einsatzkontextschicht in eigenständige Subsysteme geführt. Diese Punkte werden von ET++ nicht adressiert, stellten aber eine notwendige Erweiterung für die Entwicklung des GEBOS-Systems dar.

5.5 Technische Realisierung der beiden Architekturstile

Die bisherige Diskussion war darauf fokussiert, die getrennte Modellierbarkeit einzelner Kontexte des Anwendungsbereichs sowie die Konfektionierung von Anwendungssystemen auf der Basis von Komponenten zu ermöglichen. Schichten wurden unter dem Gesichtspunkt der logischen Einheitenbildung diskutiert. Ihre technische Umsetzung, beispielsweise in Form von Bibliotheken, wurde bisher nicht untersucht und bildet den Gegenstand dieses Abschnitts.

Als erstes thematisiere ich am Beispiel von C++ die prinzipiellen Möglichkeiten, wiederverwendbare Komponenten zur Verfügung zu stellen, die über das Konstrukt einer übersetzten Klasse hinausgehen. Dies sind statische und dynamische Bibliotheken. Mit den beiden Begriffen ist das folgende Verständnis verbunden:

- *statische Bibliothek*: Eine statische Bibliothek faßt vorübersetzte Einheiten, sogenannte Object-Dateien, zu größeren Einheiten zusammen, ohne daß dabei Referenzen zwischen den einzelnen Object-Dateien aufgelöst werden. Beim Erstellen von Anwendungssystemen löst dann ein entsprechendes Werkzeug, der Binder (Linker), die Referenzen zwischen den Object-Dateien der statischen Bibliotheken und des Anwendungssystems auf. Das entstandene ausführbare Softwaresystem enthält dabei den Programmkode der

aus den Bibliotheken verwendeten Object-Dateien. In diesem Sinne entspricht eine statische Bibliothek der klassischen Modulbibliothek. Sehr häufig spricht man in diesem Zusammenhang auch von Programmbibliotheken.

- *dynamische Bibliothek*: Bei dynamischen Bibliotheken werden hingegen die Referenzen zwischen den Object-Dateien bei ihrer Bildung aufgelöst. Sie sind somit bereits gebunden und stellen in diesem Sinne *fertige, ablauffähige Einheiten* dar. Sie entsprechen daher eher einem ausführbaren System als einer statischen Bibliothek. Anwendungssysteme, die auf der Basis von dynamischen Bibliotheken erstellt werden, enthalten nur den Programmkode, der zur Konstruktion des Anwendungssystems erstellt wurde, nicht aber den der dynamischen Bibliothek. Dieser wird, da er ja bereits fertig gebunden ist, erst zu Laufzeit des Anwendungssystems hinzugeladen. Ein spezielles Werkzeug im Betriebssystem löst dabei die Referenzen zwischen Anwendungssystem und verwendeten dynamischen Bibliotheken auf. Im Speicher eines Betriebssystems vorhandene dynamische Bibliotheken können von verschiedenen Anwendungssystemen gemeinsam benutzt werden. Dynamische Bibliotheken können dabei wiederum von weiteren dynamischen Bibliotheken abhängig sein.

Rahmenwerke lassen sich sinnvoll als dynamische Bibliotheken realisieren. Ein Rahmenwerk definiert große Teil des Kontrollflusses zur Laufzeit eines Softwaresystems und ermöglicht so die Wiederverwendung von Analysen, Entwurfsentscheidungen, Architekturen und Implementierungen (vgl. Begriff 5.2). Rahmenwerke werden daher eher als fertige Systeme denn als Programmbibliotheken genutzt. Die Realisierung eines Rahmenwerks als dynamische Bibliothek hat in großen Softwaresystemen zudem den Vorteil, daß Entwickler, da die Bibliotheken ja schon fertig gebunden sind, keine Klassen bzw. Object-Dateien des Rahmenwerks austauschen und so die Architektur des Systems verändern können. Neben diesen fachlichen und organisatorischen Vorteilen weisen dynamische Bibliotheken auch einige technische Vorzüge auf. So führen sie in großen Projekten zu kürzeren Bindezeiten des Anwendungssystems, da die dynamischen Bibliotheken bereits gebunden sind und somit beim Linken des Anwendungssystems nicht bearbeitet werden müssen. Zudem können sich die Anwendungssysteme unterschiedlicher Einsatzkontexte die dynamischen Bibliotheken der Produktbereiche, des Gegenstandsbereichs, der Technologieschicht sowie der Systembasisschicht im Speicher des Betriebssystems teilen. Klassenbibliotheken lassen sich, da große Teile des Kontrollflusses vom Anwendungssystem und nicht von der Klassenbibliothek implementiert werden, durchaus als statische Bibliotheken realisieren. Die für Rahmenwerke aufgeführten organisatorischen und technischen Vorzüge treffen aber auch auf das Konzept der Klassenbibliothek zu. Eine technische Umsetzung als dynamische Bibliothek liegt somit ebenfalls nahe. Die Diskussion in Abschnitt 5.2 hat gezeigt, daß mit Rahmenwerken bezüglich des implementierten Kontrollflusses eine höhere Komplexität verbunden ist als mit Klassenbibliotheken. Ich werde daher die Gründe für die vorgeschlagene technische Abbildung von Klassenbibliotheken und Rahmenwerken nur am Beispiel von Rahmenwerken diskutieren. Die Ergebnisse lassen sich aber einfach auf Klassenbibliotheken übertragen.

Als nächstes ist somit die Frage zu diskutieren, wie die Rahmenwerke auf dynamische Bibliotheken abzubilden sind. Ein erster Ansatz ist beispielsweise, jedes Rahmenwerk durch eine dynamische Bibliothek zu repräsentieren. Dieser Ansatz setzt die in Kapitel 3 geforderte

homologe Struktur von Anwendungsbereich, fachlichem und softwaretechnischem Modell sauber um. Allerdings bleiben strukturelle sowie softwaretechnische Aspekte der verwendeten Technologie unberücksichtigt. Um diese Gesichtspunkte besser analysieren zu können, führe ich vor ihrer Diskussion ein vereinfachtes Beispiel aus dem GEBOS-Kontext ein (vgl. Abb. 5.17).

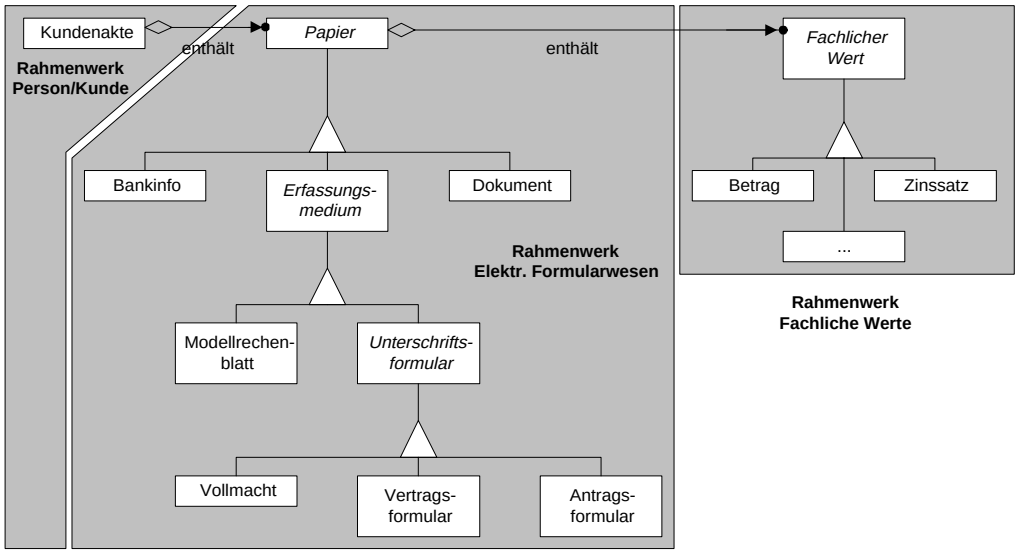


Abb. 5.17: Ein Beispiel aus Kundenakte, Papieren und fachlichen Werten

Eine Kundenakte, die Element des bereits vorgestellten Person/Kunde-Rahmenwerks ist, enthält überwiegend Schriftstücke in Papierform. Diese Benutzt-Beziehung wird durch eine entsprechende Assoziation von der Klasse `Kundenakte` zur Klasse `Papier` ausgedrückt. Die Klasse `Papier` stellt dabei die aufgeschobene Wurzelklasse des Rahmenwerks für das elektronische Formularwesen dar und basiert wiederum auf der Klasse `FachlicherWert`, die den allgemeinen Umgang mit fachlichen Werten festlegt. Die gängigen fachlichen Werte für die Konstruktion eines Banksystems sind in dem entsprechenden Rahmenwerk fachliche Werte enthalten. Eine 1 : 1 Abbildung dieser Rahmenwerksstruktur auf entsprechende dynamische Bibliotheken ist mit den folgenden Problemen verbunden:

(1) *unnötige Übersetzung*: Aufgeschobene Klassen stellen eines der Grundprinzipien der Objektorientierung dar (vgl. [Mey88] und [Lak96]). Sie erleichtern insbesondere die Umsetzung des Offen-Geschlossen-Prinzips (vgl. Abschnitt 5.4.2). Beispielsweise enthält die Klasse `Kundenakte` lediglich eine Referenz auf die Klasse `Papier`, die für die Kundenakte alle Unterklassen des Rahmenwerks elektronisches Formularwesen kapselt. Bei einer 1 : 1 Umsetzung der Rahmenwerks- in eine Bibliotheksstruktur werden sowohl aufgeschobene als auch konkrete Klassen in einer dynamischen Bibliothek zusammengefaßt. In einem System, das in der Sprache C++ entwickelt und dessen Übersetzung unter Verwendung des Make-Programms durchgeführt wird, führen Änderungen an konkreten Klassen eines Rahmenwerks zu unnötigen Übersetzungen von Rahmenwerken, die dieses verwenden. Ändert ein Softwareentwickler etwa die Klasse `Vollmacht` des Rahmenwerks elektronisches Formularwesen, so wird auch die dynamische Bibliothek des Rahmenwerks Person/Kunde unnötigerweise neu erstellt, da diese von

der dynamischen Bibliothek elektronisches Formularwesen abhängig ist. Auf Klassenebene existiert allerdings nur eine Abhängigkeit von der Klasse `Kundenakte` zu der Klasse `Papier`. Eine Übersetzung wäre somit nur notwendig, wenn sich die Klasse `Papier` ändert.

(2) *Verständlichkeit*: Wichtige Abhängigkeiten auf Klassenebene (Benutzt- und Vererbungsbeziehungen) sind auf der Ebene der Bibliotheksstruktur nicht mehr unmittelbar zu erkennen. So kann ein Softwareentwickler nur durch Browsen der Klassenstruktur feststellen, daß die Klasse `Kundenakte` lediglich die Oberklasse `Papier` der Bibliothek elektronisches Formularwesen und keine Unterklassen verwendet.

(3) *Konfektionierung*: Bei einer 1 : 1 Abbildung würden dynamische Bibliotheken immer den gesamten Klassenumfang eines Rahmenwerks enthalten. Verwendet ein Anwendungssystem nur Teile eines Rahmenwerks, so muß dennoch die gesamte dynamische Bibliothek zur Laufzeit des Systems geladen werden. Wird das System mittels einer Client-Server Architektur realisiert, so enthält die Bibliothek sowohl die Client- als auch die Serverteile, da sie ja unter fachlichen Gesichtspunkten in einem Rahmenwerk zusammengefaßt werden. Die Trennung von Rahmenwerks- und Bibliotheksstruktur ermöglicht es hingegen, Bibliotheken unter anwendungsspezifischen oder technischen Gesichtspunkten so zu konfektionieren, daß sie nur die für ein Softwaresystem notwendige Funktionalität enthalten.

(4) *Zuordnung*: ein Rahmenwerk ist immer genau einer logischen Schicht zugeordnet. Unter softwaretechnischen Gesichtspunkten (z.B. minimierte Kopplung) kann es allerdings notwendig sein, dynamische Bibliotheken schichtenübergreifend zu bilden.

Die vorige Diskussion führt zu dem Ergebnis, daß Rahmenwerke (Module) und Bibliotheken (Übersetzungseinheiten) nach unterschiedlichen Strukturierungskriterien gebildet werden sollten. Zur Organisation von Modulen und Übersetzungenseinheiten und ihrem Verhältnis zueinander schreibt Leroy in [Ler94]:

> "Modularization is the process of decomposing a program in[to] small units (modules) that can be understood in isolation by the programmers, and making the relations between those units explicit to the programmers. Separate compilation is the process of decomposing a program in[to] small units (compilation units) that can be type-checked and compiled separately by the compiler, and making the relations between these units explicit to the compiler and linker."([Ler94] aus [ND95], S. 16)

Auf Grund der genannten Punkte ist die Bibliotheksstruktur von der Rahmenwerksstruktur zu trennen, wissend, daß damit ein Strukturbruch zwischen softwaretechnischem Modell und der Organisation der Systemkomponenten in Bibliotheken verbunden ist[17]. Ich schlage daher die nachfolgende Abbildung von Komponenten auf eine Bibliotheksstruktur vor. Der Begriff Komponente ist dabei im Sinne des in Abschnitt 5.3 vorgestellten rahmenwerkbasierten

[17] Die beschriebenen Nachteile treffen so nicht auf statische Bibliotheken zu. Insbesondere die Punkte (1) und (3) sind, da statische Bibliotheken nur eine Bündelung von vorübersetzten Einheiten darstellen, nicht übertragbar.

Architekturstils zu verstehen. Er bezieht sich somit auf eine Klassenbibliothek oder ein Rahmenwerk.

Rahmenwerke und Klassenbibliotheken werden logisch in einen Konzeptteil und n, mit $n \in N$, Realisierungsteile unterteilt. Der Konzeptteil umfaßt dabei die aufgeschobenen Klassen einer Komponente. Für Rahmenwerke beschreiben die Konzeptteile i.d.R. das Zusammenspiel von Objekten zur Laufzeit eines Systems (vgl. Begriff 5.2). Der Realisierungsteil dagegen enthält die konkreten Klassen einer Komponente. Beide, der Konzept- und der Realisierungsteil, besitzen sowohl eine Schnittstellendefinition als auch eine Implementierung. Ausgehend von dieser Unterteilung werden Konzeptteile verschiedener Rahmenwerke und Klassenbibliotheken unter Berücksichtigung von Modularisierungsprinzipien, wie etwa minimierte Kopplung und maximierte Kohäsion, zu dynamischen Bibliotheken zusammengefaßt. Die Bündelung der Implementierungsteile erfolgt analog. Sie muß allerdings nicht mit der Bündelung der Konzeptteile übereinstimmen (vgl. Abb. 5.18).

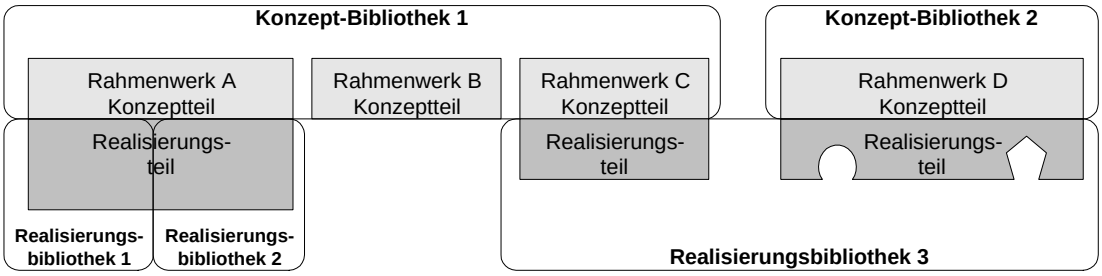


Abb. 5.18: Aufteilung von Komponenten in einen Konzept- und einen Realisierungsteil

Wird diese Vorgehensweise auf die Rahmenwerksstruktur des Beispiels aus Abb. 5.17 angewendet, so ergibt sich die folgende Bibliotheksstruktur: die Konzeptklasse `Kundenakte` des Rahmenwerks Person/Kunde, die Konzeptklassen `Papier` und `Erfassungsmedium` des Rahmenwerks elektronisches Formularwesen sowie die Konzeptklasse `FachlicherWert` des Rahmenwerks fachliche Werte werden in einer Konzeptbibliothek zusammengefaßt. Die restlichen Klassen des Rahmenwerks elektronisches Formularwesen und fachliche Werte bilden je eine Realisierungsbibliothek. Die Klassen des Rahmenwerks elektronisches Formularwesen werden also auf zwei dynamische Bibliotheken aufgeteilt. Die resultierende Bibliotheksstruktur ist in der Abb. 5.19 dargestellt.

Die technische Umsetzung der in Abschnitt 5.4.4 definierten fünf Schichten erfolgt durch eine entsprechende Verzeichnisstruktur. Jede Schicht wird dabei durch ein Verzeichnis repräsentiert, wobei sich die jeweiligen Verzeichnisse für die Produktbereichs- und Einsatzkontextschicht entsprechend der n Produktbereiche bzw. den m Einsatzkontexten weiter untergliedern. Die Klassenbibliotheks- und Rahmenwerksstruktur der einzelnen Schichten wird ebenfalls durch Verzeichnisse abgebildet. Die Bibliotheksstruktur läßt sich dann als zweite eigenständige Sicht unter Verwendung eines virtuellen Dateisystems oder von symbolischen Verweisen, wie sie etwa mit dem Unix Befehl `ln` eingerichtet werden können, etablieren. Die

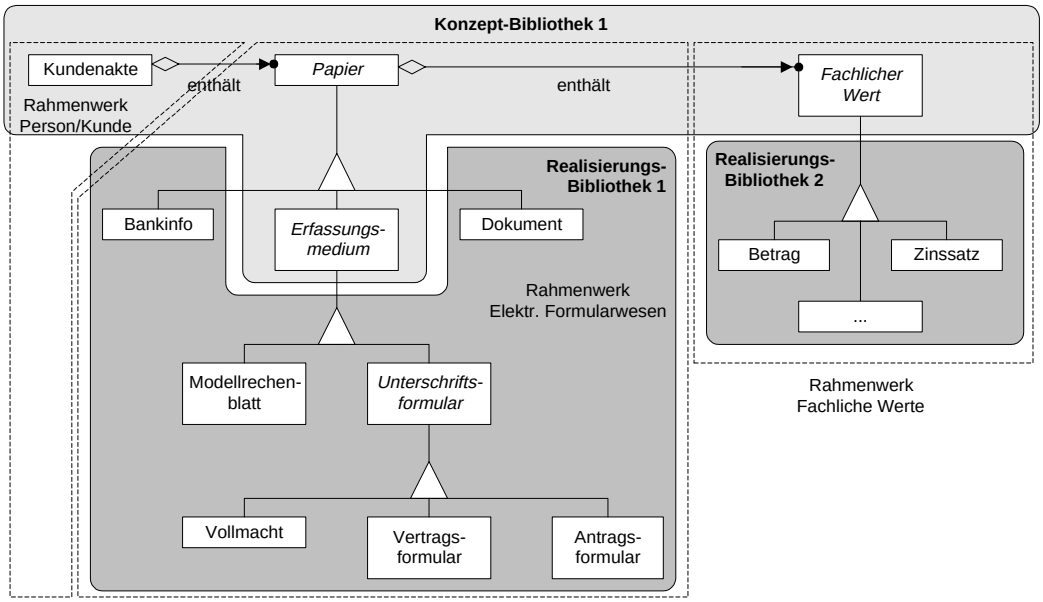


Abb. 5.19: Eine konkrete Umsetzung von Konzept- und Realisierungsbibliotheken

Abb. 5.20 stellt eine auf dem vorangegangenen Beispiel basierende Schichten-, Rahmenwerks- und Bibliotheksstruktur dar. Die Verbindung zwischen Klassen in den Bibliotheken zu den Klassen in den Rahmenwerken wurde dabei mittels symbolischer Verweise hergestellt. Diese sind in der Abbildung *kursiv* und durch einen Pfeil auf die Klasse im Rahmenwerk dargestellt.

Die vorgestellte Trennung der logischen Schichten- und Komponentenstruktur des softwaretechnischen Modells von der Bibliotheksstruktur schafft die Voraussetzung, um verschiedene Bibliotheksstrukturen in einem Projekt zu etablieren. Aus Entwicklungssicht ist es beispielsweise vorteilhaft, viele kleine dynamische Bibliotheken einzusetzen. Änderungen, die sich nur auf die Implementierung der Bibliothek auswirken, und nicht ihre Schnittstelle betreffen, führen nur zur Neuübersetzung dieser einen Bibliothek. Ein neuer Bindevorgang des Anwendungssystems oder gar dessen Übersetzung ist nicht notwendig. In der Konsequenz führt dies zu kürzeren Entwicklungszyklen. Allerdings ziehen viele kleine dynamische Bibliotheken einen höheren Speicherbedarf und eine längere Ladezeit des Anwendungssystems nach sich. Die Auslieferungsversion eines Softwaresystems sollte daher aus wenigen großen Bibliotheken bestehen. Eine derartige Bibliotheksstruktur kann durch das vorgestellte Prinzip der Trennung in einer zweiten Verzeichnisstruktur abgebildet werden.

Leider wird dieses Prinzip von gängigen Entwicklungsumgebungen nur unzureichend unterstützt. Sie verbinden i.d.R. eine Projektstruktur mit einer Bibliotheksstruktur. Bei großen Projekten führt dies in der Konsequenz zu einer Übertragung der physischen Auslieferungsstruktur auf die logische Komponentenstruktur, was mit den diskutierten Nachteilen verbunden ist.

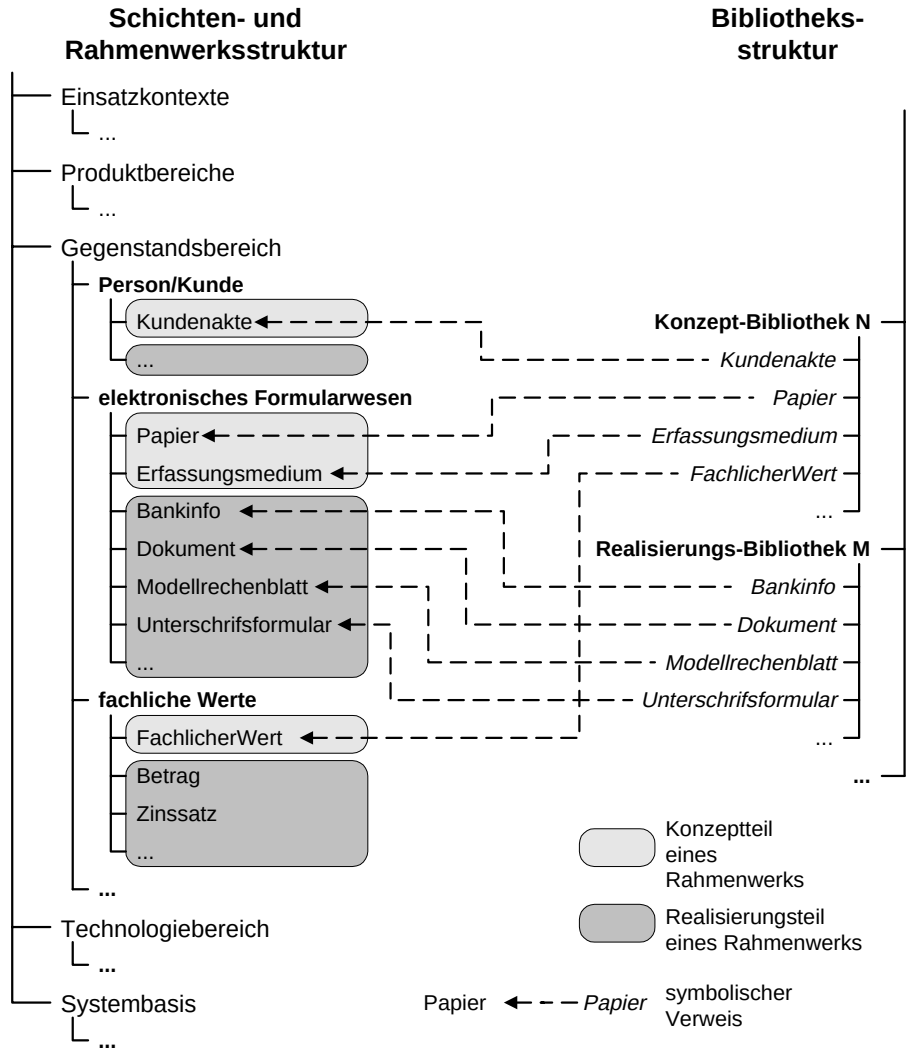


Abb. 5.20: Die technische Umsetzung von Schichten und Rahmenwerken in eine Bibliotheksstruktur

5.6 Zusammenfassung und Ausblick

Ich habe in diesem Kapitel auf der Basis der von Shaw & Garlan definierten Merkmale von Architekturstilen und Architekturbeschreibungssprachen zwei neue Architekturstile vorgestellt. Dazu wurden zunächst in Abschnitt 5.2 Begriffe für die unterschiedlichen Verwendungsformen eines Rahmenwerks eingeführt. Dabei habe ich neben der in der Literatur gängigen Praxis, zwischen White-Box und Black-Box Rahmenwerken zu unterscheiden, zusätzlich die Begriffe des geschlossen verwendbaren White-Box Rahmenwerks sowie des offen verwendbaren Black-Box Rahmenwerks herausgearbeitet. Ergänzt um die Komponente Klassenbibliothek und die Verbindungsstücke Entwurfsmuster, Spezialisierung und Benutzt-Beziehung habe ich dann einen rahmenwerkbasierten Architekturstil und eine entsprechende graphische Notation definiert. Dieser Architekturstil ist speziell darauf ausgerichtet, die Architektur eines Anwendungssystems als Komposition unterschiedlicher Rahmenwerke zu beschreiben.

Im Anschluß daran wurden die Eigenschaften von Schichtenarchitekturen diskutiert. Dabei habe ich vor dem Hintergrund des Offen-Geschlossen-Prinzips die Unterscheidung in strikte und nicht-strikte Schichtenarchitekturen getroffen. Durch die Konzepte aufgeschobene Klasse, Vererbung, Polymorphie und dynamisches Binden ermöglichen es nicht-strikte Schichtenarchitekturen, die Funktionalität einer tieferliegenden Schicht in einer höherliegenden Schicht zu erweitern, ohne daß die tieferliegende Schicht dazu geöffnet werden muß. Nicht-strikten Schichtenarchitekturen kommt daher bei der Entwicklung von komplexen Anwendungssystemen eine besondere Bedeutung zu. Meine Kritik der weitverbreiteten Schichtenarchitekturen führte zur Definition des objektorientierten Schichtenarchitekturstils. Dieser basiert, ähnlich wie ET++, auf einer logischen Schichtenbildung. Die einzelnen Schichten sind: Systembasis, Technologiebereich, Gegenstandsbereich, Produktbereiche und Einsatzkontexte. Die beiden Schichten Produktbereiche und Einsatzkontexte sind dabei entsprechend der in Kapitel 4 definierten Struktur für das fachliche Modell vertikal in einzelne Produktbereiche bzw. Einsatzkontexte partitioniert. Die Entwurfselemente in den jeweiligen logischen Schichten entsprechenden denen eines rahmenwerkbasierten Architekturstils. Die dadurch etablierte Kombination der beiden Architekturstile ist neben den bereits in Kapitel 4 beschriebenen Eigenschaften mit den folgenden Charakteristika verbunden:

- Die Makrostrukturen des Anwendungsbereichs und des fachlichen Modells lassen sich in das softwaretechnische Modell übertragen, d.h. das softwaretechnische Modell weist auch die beschriebene Struktur aus Gegenstands-, Produktbereich und Einsatzkontext auf.
- Die Abbildung der in Kapitel 2 identifizierten Kontexte Systembasis und verwendete Technologie ermöglichen die technische Kombinierbarkeit und Integrierbarkeit von Komponenten in unterschiedlichen Anwendungssystemen.
- Der Gegenstandsbereich stellt als fachliche Basis die fachliche Kombinierbarkeit und Integrierbarkeit der in den Produktbereichen entwickelten Komponenten sicher.

Ausgehend von der Struktur im softwaretechnischen Modell habe ich in Abschnitt 5.5 die technische Umsetzung der beiden Architekturstile diskutiert. Technologische Argumente sprechen dafür, Bibliotheksstrukturen nicht homolog zur Struktur des softwaretechnischen Modells zu bilden. Folgende Vorgehensweise ist dabei festgelegt worden: Klassenbibliotheken und Rahmenwerke werden logisch in einen Konzept- und in n Realisierungsteile unterteilt. Ausgehend von dieser Unterteilung werden Konzeptteile verschiedener Klassenbibliotheken und Rahmenwerke unter Berücksichtigung von Modularisierungsprinzipien zu dynamischen Bibliotheken zusammengefaßt. Die Bündelung der Implementierungsteile erfolgt analog. Die bibliotheksorientierte Sicht auf das aus Schichten, Klassenbibliotheken und Rahmenwerken bestehende softwaretechnische Modell wird mittels virtueller Dateisysteme oder symbolischer Verweise umgesetzt.

Durch die in diesem Kapitel definierten Architekturstile und deren Umsetzung in eine entsprechende Bibliotheksstruktur ist ein Konzept für die Entwicklung großer rahmenwerkbasierter Anwendungssysteme entstanden. Insbesondere die Diskussion über nicht-strikte Schichtenarchitekturen hat gezeigt, daß das Offen-Geschlossen-Prinzip einen wichtigen Faktor bei der Umsetzung von Schichtenarchitekturen darstellt. Im weiteren ist zu zeigen, wie mit Hilfe von

Entwurfsmustern das Problem der gemeinsamen fachlichen Identität (vgl. Abschnitt 4.4.1) von Objekten der Gegenstandsbereichs- und Produktbereichsschicht gelöst werden kann. Zudem muß die Entkopplung der Objekterzeugung vom Typsystem der Programmiersprache sichergestellt werden. Denn nur so ist es möglich, daß sich Realisierungsteile einer Komponente ausschließlich auf die Konzeptteile anderer Komponenten beziehen können. Beide Problemstellungen werden im folgenden Kapitel adressiert.

6 Händler und Rollen als schichtenübergreifende Verbindungsstücke zwischen Rahmenwerken

"Design patterns identify, name, and abstract common themes in object-oriented design. They preserve design information by capturing the intent behind a design." ([GOF95], S. 407)

In der vorausgegangenen Diskussion habe ich deutlich gemacht, daß sich bestimmte Entwurfsmuster als schichtenübergreifende Verbindungsstücke zwischen Rahmenwerken einsetzen lassen. In diesem Kapitel sollen zwei wichtige Muster unter diesem Gesichtspunkt diskutiert werden.

Dienstleistungen werden in der Objektorientierung von Objekten erbracht. Will ein Klient also eine gewünschte Dienstleistung in Anspruch nehmen, so muß ihm entweder zuvor ein Objekt übergeben werden, oder er muß sich ein Objekt, das die gewünschte Dienstleistung erbringen kann, erzeugen. Die Instanziierung ist unter folgender Rahmenbedingung problematisch: wird die Dienstleistung von einem Objekt erbracht, dessen Klasse in einer höherliegenden Schicht oder in einem benachbarten Produktbereich enthalten ist, so kann dieses auch dann nicht erzeugt werden, wenn der spätere Zugriff nur unter der Schnittstelle seiner aufgeschobenen Oberklasse erfolgt, die Bestandteil dergleichen oder einer tieferliegenden Schicht ist (z.B. Schicht Gegenstandsbereich oder Technologie). Denn der Aufruf der `new` Operation erzeugt eine Abhängigkeit zu einer höherliegenden Schicht oder zu einem benachbarten Produktbereich. Die in [GOF95] vorgestellten Erzeugungsmuster lösen diese Problematik im Zusammenhang mit einer Schichtenarchitektur nur unzureichend. In Abschnitt 6.1 wird daher als Lösung das Händler-Muster vorgestellt.

Die Modellierung eines Anwendungssystems wird immer von einem ganz bestimmten Blickwinkel aus durchgeführt. Daher sind Klassen, die die Gegenstände des Anwendungsbereichs auf ihre nominale Essenz bringen, geprägt durch die Sichtweise, unter der ein Softwareentwickler den Anwendungsbereich betrachtet. Dies gilt insbesondere für die unterschiedlichen Produktbereiche. Diese differenzierte Sichtweise ist unproblematisch, solange aus Sicht der verschiedenen Produktbereiche unterschiedliche Gegenstände des Anwendungsbereichs modelliert werden. Weist allerdings ein Konzept des Gegenstandsbereichs aus dem Blickwinkel der Produktbereiche verschiedene Attribute und Umgangsformen auf, so sind spezifische Lösungsmöglichkeiten erforderlich, um nicht alle Sichtweisen an diesem einen Konzept modellieren zu müssen. Denn dann würde das Konzept als Bestandteil des Gegenstandsbereichs Abhängigkeiten zu den Produktbereichen besitzen. Derartige Abhängigkeiten führen zu einem gemeinsamen Bereich, der alle Produktbereiche und den Gegenstandsbereich umfaßt. Dies entspräche wieder dem zu Anfang dieser Arbeit kritisierten unternehmensweiten Modell. Ich führe daher als Lösung dieser Problematik in Kapitel 6.2 das Modellierungsmittel der Rolle ein (vgl. [GSR95] und [KØ96]). Rollen ermöglichen es, die aus der Sicht eines Produktbereichs fachlich zusammengehörigen Attribute und Umgangsformen eines Konzepts

aus dem Gegenstandsbereich in einer eigenen Klasse zu modellieren. In Abschnitt 6.3 werde ich ein Entwurfsmuster vorstellen, das die technische Realisierung dieser Klassen in Produktbereichen erlaubt.

Die bisher eingeführten Techniken und Architekturstile erlauben es, Anwendungssysteme auf der Basis von Rahmenwerken zu erstellen. Die Konfiguration solcher Anwendungssysteme läßt sich aber bisher nicht beschreiben. Abschnitt 6.4 beschäftigt sich daher mit der Konfektionierung von Anwendungssystemen unter Verwendung von Konfigurationsskripten.

Ich möchte hier darauf hinweisen, daß der Einsatz der zwei vorgestellten Entwurfsmuster nicht auf die schichtenübergreifende Kopplung von Rahmenwerken beschränkt ist. Beide eignen sich ebenso als Verbindungsstücke zwischen Rahmenwerken, die in derselben logischen Schicht liegen. Zudem lassen sich alle hier vorgestellten Konzepte auch auf Klassenbibliotheken übertragen (vgl. Abschnitt 5.2 und 5.5). Da aber Klassenbibliotheken bei der Realisierung von großen Anwendungssystemen eine eher untergeordnete Rollen spielen (vgl. Abschnitt 5.4.4 und Abb. 5.14) und die Schreibweise »Klassenbibliothek und Rahmenwerk« einem flüssigen Lesen entgegenwirkt, werden die in diesem Kapitel vorgestellten Konzepte nur am Beispiel von Rahmenwerken erläutert. Dieses Kapitel beruht teilweise auf den Artikeln [BR97], [RSB+97] und [BRS+97], an denen ich als Autor mitgewirkt habe.

6.1 Händler

In Zusammenhang mit der in Abschnitt 5.4 diskutierten Schichtenarchitektur habe ich gezeigt, daß Objekte in einer tieferliegenden Schicht nach polymorpher Zuweisung Objekte einer höherliegenden Schicht referenzieren können (vgl. Abb 5.10 und 5.11). Der Umgang mit einem referenzierten Objekt erfolgt über die Schnittstelle einer aufgeschobenen Oberklasse, die ebenfalls Bestandteil der tieferliegenden Schicht ist. Dieses Verfahren bildet die Grundlage des Offen-Geschlossen-Prinzips, das es ermöglicht, bereits antizipierte Erweiterungen einer Schicht auch in höherliegenden Schichten zu realisieren.

Die Instanziierung von Objekten aus höherliegenden Schichten ist in tieferliegenden Schichten nicht über den direkten Aufruf des `new` Operators möglich. Denn dadurch würden Abhängigkeiten von Klassen in tieferliegenden Schichten zu Klassen höherliegender Schichten entstehen. Solche Abhängigkeiten sind allerdings in einer Schichtenarchitektur nicht erlaubt (vgl. Abschnitt 5.4.1).

In vielen Fällen ist eine derartige Instanziierung von Objekten jedoch wünschenswert. Denn dann könnte nicht nur auf der Ebene von aufgeschobenen Klassen das Zusammenspiel von Objekten beschrieben, sondern auch deren konkrete Erzeugung festgelegt werden. Dies soll an einem Beispiel aus dem GEBOS-Kontext erläutert werden.

Für die Konstruktion bankfachlicher Anwendungssysteme wurde eine Klassenhierarchie entworfen und implementiert, die die relevanten (bank-) fachlichen Werte (kurz: fachlichen Werte) abbildet. Beispiele für fachliche Werte sind etwa die Klassen `Kontonummer`, `Betrag`, `Zinssatz` oder `Datum`. Fachliche Werte werden in fast allen Rahmenwerken der Schichten

Gegenstandsbereich und Produktbereich sowie in den Anwendungssystemen der Schicht Einsatzkontext verwendet. Das elektronische Formularwesen ist hierfür ein Beispiel. Es modelliert die Formulare des Bankgeschäfts in Klassen. Dabei werden die auf den Formularen enthaltenen Felder durch ihre korrespondierenden fachlichen Werte repräsentiert. Abb. 6.1 gibt einen Überblick über die vorgestellte Klassenhierarchie.

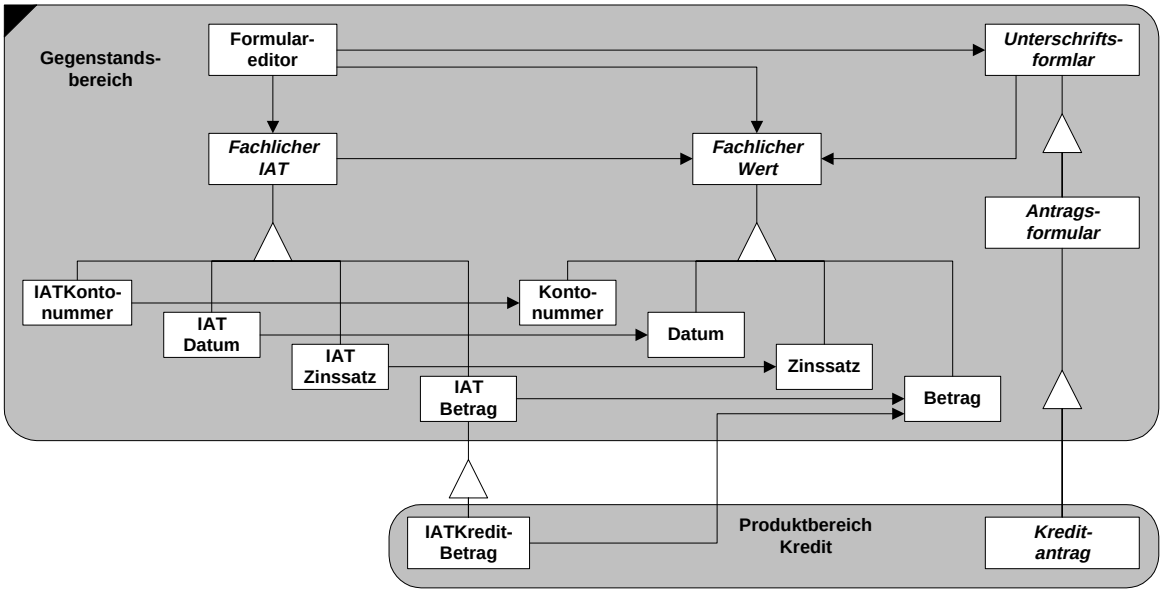


Abb. 6.1: Das Zusammenspiel von Klassen zur Darstellung von Formularen

Ein auf die wesentlichen Merkmale beschränktes Programmfragment der Klasse `Kredit-antrag`, die den fachlichen Gegenstand Kreditantrag modelliert, präsentiert die Abb. 6.2:

```
class Kreditantrag: public Antragsformular
{
      public:
            list <FachlicherWert *> GibFelder( void );
      private:
            ...
            Kontonummer _Kontonummer;
            Datum _EroeffungsDatum;
            Betrag _KreditBetrag;
            ...
};
```

Abb. 6.2: Ein Programmfragment der Klasse `Kreditantrag`

Ein Formulareditor, der Formulare am Bildschirm darstellt und es ermöglicht, diese auszufüllen, benötigt zur Präsentation der fachlichen Werte korrespondierende Interaktionstypen (Controls, Widgets). Die dazugehörenden Interaktionstypen berücksichtigen die spezifischen Darstellungs- und Editiermöglichkeiten der fachlichen Werte. So wird beispielsweise ein Betrag im Bankenbereich mit Tausenderpunkten präsentiert, die Nachkommastellen werden durch ein Komma abgetrennt, und das Vorzeichen wird dem Betrag nachgestellt. Eine den fachlichen Werten entsprechende Klassenhierarchie der fachlichen Interaktionstypen zeigt ebenfalls die Abb. 6.1[18]. Sie beinhaltet zusätzlich einen auf das Kreditgeschäft spezialisierten

[18] Aus Platzgründen wurde in der Abbildung der Bezeichner Interaktionstyp durch die Abkürzung IAT ersetzt.

Interaktionstyp für den fachlichen Wert `Betrag`, der das Vorzeichen voranstellt. Die Abbildung verdeutlicht die Aufteilung der Klassen auf die in Abschnitt 5.4.4 eingeführte objektorientierte Schichtenarchitektur. Die Klassen `IATKreditbetrag` und `Kreditantrag` sind Bestandteil des Produktbereichs Kredit. Alle anderen Klassen sind in der Schicht Gegenstandsbereich enthalten, da sie Kernkonzepte für die Produktbereiche darstellen.

Ein Formulareditor, der das Bearbeiten von Unterschriftsformularen ermöglicht, muß für alle fachlichen Werte, die er durch Aufruf der Methode `GibFelder` von einem Formular anfordert, einen passenden Interaktionstyp erzeugen. Folgende Lösungsmöglichkeiten bieten sich hier an:

- Der `Formulareditor` enthält eine große Case-Anweisung, in der abhängig vom Typ des fachlichen Wertes ein entsprechender Interaktionstyp erzeugt wird. Da der Editor im Kreditumfeld für einen Betrag den speziellen Interaktionstyp `IATKreditbetrag` erzeugen muß, entsteht in der Klasse `Formulareditor` eine nicht zulässige Abhängigkeit zu einer Klasse (`IATKreditbetrag`), die in einer höherliegenden Schicht enthalten ist. Zudem muß die Case-Anweisung bei jeder Erweiterung der fachlichen Wertehierarchie gepflegt werden. Jede Änderung an der Klassenhierarchie der fachlichen Werte kann demnach zu einer Öffnung des Gegenstandsbereichs führen.
- Die fachlichen Werte implementieren eine Fabrikmethode (vgl. [GOF95], S. 107), in der ein Objekt des entsprechenden Interaktionstyps erzeugt wird. Die Zuordnung eines fachlichen Wertes zu einem Interaktionstyp ist dann allerdings nicht konfigurierbar. Sie wird einmalig statisch festgelegt und führt in allen Anwendungen bei Aufruf der Fabrikmethode zur Erzeugung desselben Interaktionstyps. Ist beispielsweise dem fachlichen Wert `Betrag` in der Fabrikmethode der Interaktionstyp `IATBetrag` zugeordnet, so könnte für eine Kreditanwendung diese Zuordnung ohne Kodeänderung bzw. zusätzliche Parametrisierung nicht angepaßt werden. Auch hier müßte demnach der Gegenstandsbereich geöffnet werden.
- Eine abstrakte Fabrik (vgl. [GOF95], S. 87) verbirgt die Objekterzeugung vor einem Klienten (dem `Formulareditor`), indem sie für jeden Fachwert eine aufgeschobene Operation zur Erzeugung des passenden Interaktionstyps anbietet. Eine konkrete Fabrik implementiert die aufgeschobenen Operationen, indem sie die gewünschten Interaktionstypen erzeugt (vgl. Abb. 6.3).

  Dieses Entwurfsmuster ermöglicht es zwar, daß eine konkrete Fabrik für den Kreditbereich eine spezielle Zuordnung des fachlichen Wertes `Betrag` zum Interaktionstyp `IATKreditbetrag` vornimmt. Das Muster weist jedoch die folgenden Nachteile auf: der `Formulareditor` kann nur anhand einer großen Case-Anweisung entscheiden, welche Operation der abstrakten Fabrik abhängig vom Typ des fachlichen Werts aufzurufen ist. Weiterhin führen neue fachliche Werte zu neuen aufgeschobenen Operationen an der abstrakten Fabrik. Demnach bedingen Änderungen an der Hierarchie der fachlichen Werte oder der Interaktionstypen die Öffnung des Gegenstandsbereichs, selbst dann, wenn diese Änderungen in einem Produktbereich realisiert werden.
- Alternativ zum Einsatz einer Fabrik könnte der `Formulareditor` in jedem Produktbereich mit spezifischen Anforderungen an die Darstellung und Bearbeitung von fachlichen Werten spezialisiert werden. Die Lösung führt jedoch zu Problemen, sobald der

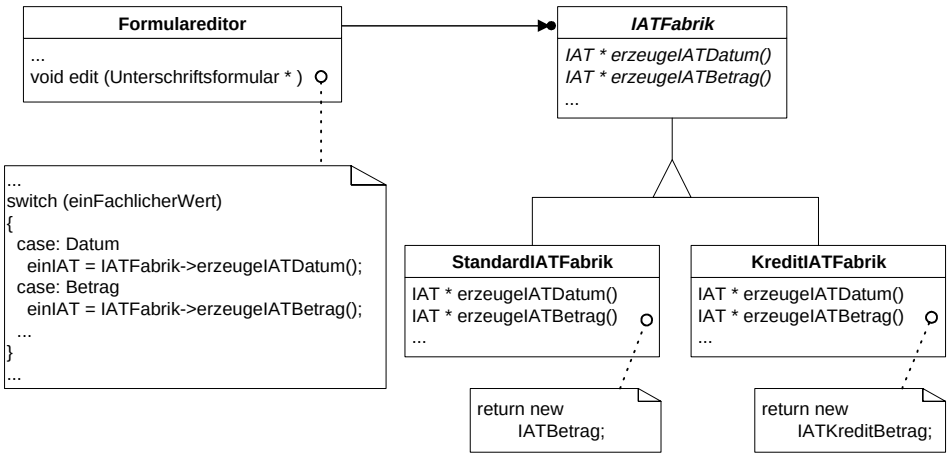


Abb. 6.3: Einsatz des Fabrikmusters für die Erzeugung von Interaktionstypen

Formulareditor in einem Einsatzkontext benötigt wird, in dem spezialisierte Wertdarstellungen aus unterschiedlichen Produktbereichen zusammenzuführen sind. Denn dann müßte im Einsatzkontext eine weitere Spezialisierung der Klasse `Formulareditor` gebildet werden. Diese Spezialisierung kann nur mittels Mehrfachvererbung oder durch Kopieren von Kode implementiert werden. Keine der beiden Realisierungsformen stellt aus softwaretechnischer Sicht eine elegante Lösung dar.

Die beschriebenen Nachteile können vermieden werden, wenn ein Klient eine Stelle im System damit beauftragt, für eine bestimmte Dienstleistung einen Anbieter zu ermitteln. Der Formulareditor könnte sich für sämtliche fachlichen Werte an diese Stelle wenden und würde als Ergebnis den passenden Interaktionstyp geliefert bekommen. Eine Stelle, die zwischen einem Klienten und einem Leistungsanbieter vermittelt, wird als Händler (engl. trader) bezeichnet. Mätzel & Bischofberger führen zum Begriff des Händlers aus:

> "The process of finding a service provider, e.g. the matching of requests with offers is called trading [ISO93a]. The instance that performs trading is called a trader." ([MB96], S. 109)

Beispiele für Komponenten in kommerziellen Anwendungssystemen, die die Aufgaben eines Händlers wahrnehmen, sind etwa die Namensverwaltung im Distributed Computing Environment (DCE, vgl. [Sch93]) oder der Object Request Broker (ORB) in CORBA (vgl. [OMG96]).

In der Objektorientierung werden Dienstleistungen (Services) von Objekten[19] erbracht, die sie als Operationen anderen Objekten zur Verfügung stellen. Klient und Leistungsanbieter werden also in der Objektorientierung durch Objekte realisiert. Die Dienstleistung, die der Leistungsanbieter erbringen soll, beschreibt der Klient in Form einer Spezifikation. Spezifikationen lassen sich wiederum durch Objekte realisieren. Es können die Objekte sein, mit denen der Leistungsanbieter zusammenarbeiten soll, Leistungsbeschreibungen wie etwa das

[19] Dies können sowohl Exemplare einer Klasse als auch Klassenobjekte sein.

Laufzeitverhalten einer Operation, aber auch eindeutige Identifikationen wie etwa eine Klassennummer oder eine Metaklasse.

Objekte, die eine bestimmte Dienstleistung erbringen und sie an andere Objekte vermitteln lassen, werden in einem Entwurfsmuster in der Regel als Produkt bezeichnet. Ich werde daher im weiteren von einem Produkt anstelle eines Leistungsanbieters reden. Den Begriff des Händlers definiere ich für meine Arbeit wie folgt:

Begriff 6.1 Händler

Ein Händler vermittelt einem Klienten ein Produkt, das für den Klienten eine gewünschte Dienstleistung erbringen kann. Die gewünschte Dienstleistung beschreibt der Klient dabei in Form einer Spezifikation.

Würde man die Zuordnung eines fachlichen Werts zu einem Interaktionstyp nicht am Editor, den fachlichen Werten oder einer Fabrik führen, sondern in einem Händler realisieren, dann könnte dieser zu einem fachlichen Wert einen entsprechenden Interaktionstyp erzeugen. Die Aufgabe des Händlers kann dabei die Klasse `FachlicherIAT` übernehmen. In diesem Sinne erzeugt (vermittelt) die Klasse `FachlicherIAT` auf Grund einer Spezifikation (dem fachlichen Wert, der editiert werden soll) ein Exemplar eines konkreten fachlichen Interaktionstyps. In C++ müßte dazu die Klasse `FachlicherIAT` die folgende Klassenoperation anbieten:

```
static FachlicherIAT * FachlicherIAT :: erzeugeFuer(FachlicherWert *);
```

Der `Formulareditor` würde dann die Erzeugung eines Interaktionstyps, mit dem sich ein fachlicher Wert darstellen und editieren läßt, folgendermaßen implementieren:

```
class Formulareditor {
public:
      ...
      void edit(Unterschriftsformular * dasFormular)
      {
            list<FachlicherWert *> eineWerteListe = dasFormular->GibFelder();
            iterator<FachlicherWert *> i(eineWerteListe);
            FachlicherWert * einWert;
            FachlicherIAT * einIAT;

            while( einWert = i.getNext() )
            {
                  // erzeuge den entsprechenden Interaktionstyp
                  einIAT = FachlicherIAT :: erzeugeFuer(einWert);
                  // Positioniere den IAT
                  ...
                  einIAT->show();
            }
      }
      ...
};
```

Intern müßte die Klasse `FachlicherIAT` lediglich eine Zuordnung (beispielsweise als Hashtabelle) führen, die den dynamischen Typ[20] eines fachlichen Wertes einem konkreten Interaktionstyp zuordnet. Dieser Prozeß ist effizient und läßt sich in konstanter Zeit durchführen.

[20] Der dynamische Typ eines Objektes kann in C++ über die Runtime Type Information (RTTI) Schnittstelle ermittelt werden.

Der eigentliche Interaktionstyp, von dem ein Objekt zu erzeugen ist, wird somit erst zur Laufzeit des Systems festgelegt. Da dem Klienten das erzeugte Objekt unter einer abstrakten Schnittstelle geliefert wird, kann er darauf auch nur über die Schnittstelle der aufgeschobenen Klasse `FachlicherIAT` zugreifen. Die konkreten Interaktionstypen bleiben vor dem Klienten somit vollständig verborgen. Ferner kann die Zuordnung zur Laufzeit des Systems an geänderte Bedingungen angepaßt werden. Der Mechanismus des Händlers schafft die Voraussetzung, Rahmenwerke einer Schicht bezüglich der Objekterzeugung zu schließen, aber trotzdem für Veränderungen offen zu halten. Händler erleichtern somit die Evolution von Klassenhierarchien, Rahmenwerken und Anwendungssystemen.

Im weiteren präsentiere ich die technische Realisierung eines Händlers. Insbesondere wird die Konfiguration von Händlern, d.h. die Zuordnung einer Spezifikation, die eine gewünschte Dienstleistung beschreibt, zu einem Objekt, das die Dienstleistung erbringen kann, ausführlich dargelegt. Die Realisierung wird in Form eines Entwurfsmusters beschrieben. Ich beschränke mich dabei auf die Anwendbarkeit, Struktur, Teilnehmer, Interaktionen, Konsequenzen, Implementierung und den Beispielkode des Entwurfsmusters. Für eine detaillierte Darstellung des Musters verweise ich auf den Artikel [BR97].

6.1.1 Das Entwurfsmuster Produkthändler

Anwendbarkeit

Der Produkthändler sollte in den folgenden Situationen verwendet werden:

- wenn ein Klient kein Wissen darüber besitzen sollte, welche Dienstleistung von welchem konkreten Produkt erbracht wird. Der Klient ist somit vollständig von der Existenz konkreter Produkte entkoppelt.
- wenn die Spezifikation, die die zu erbringende Dienstleistung beschreibt, erst zur Laufzeit bestimmt werden kann.
- wenn die Zuordnung zwischen der Spezifikation und dem Produkt, das die Dienstleistung erbringt, statisch oder zur Laufzeit des Systems angepaßt werden soll, ohne dabei das Offen-Geschlossen-Prinzip zu verletzen.

Der Produkthändler sollte nicht als genereller Ersatz für die direkte Objekterzeugung (`new` Operator) oder für die Entwurfsmuster Fabrikmethode bzw. abstrakte Fabrik eingesetzt werden. Denn mit dem Einsatz von Produkthändlern sind auch Nachteile verbunden, wie etwa eine höhere Komplexität und damit auch eine höhere Fehleranfälligkeit des Anwendungssystems (vgl. Abschnitt Konsequenzen).

Struktur

Die nachfolgend aufgezeigte Struktur geht von der Annahme aus, daß Produkte, die durch einen Händler vermittelt werden, jeweils neu zu erzeugen sind. Diese Einschränkung habe ich getroffen, da dadurch die Präsentation der Struktur vereinfacht wird. Wie der Abschnitt über die konzeptionelle Erweiterung des Entwurfsmusters allerdings zeigen wird, läßt sich das Muster einfach ausbauen, um auch im System bereits existierende Produkte zu vermitteln.

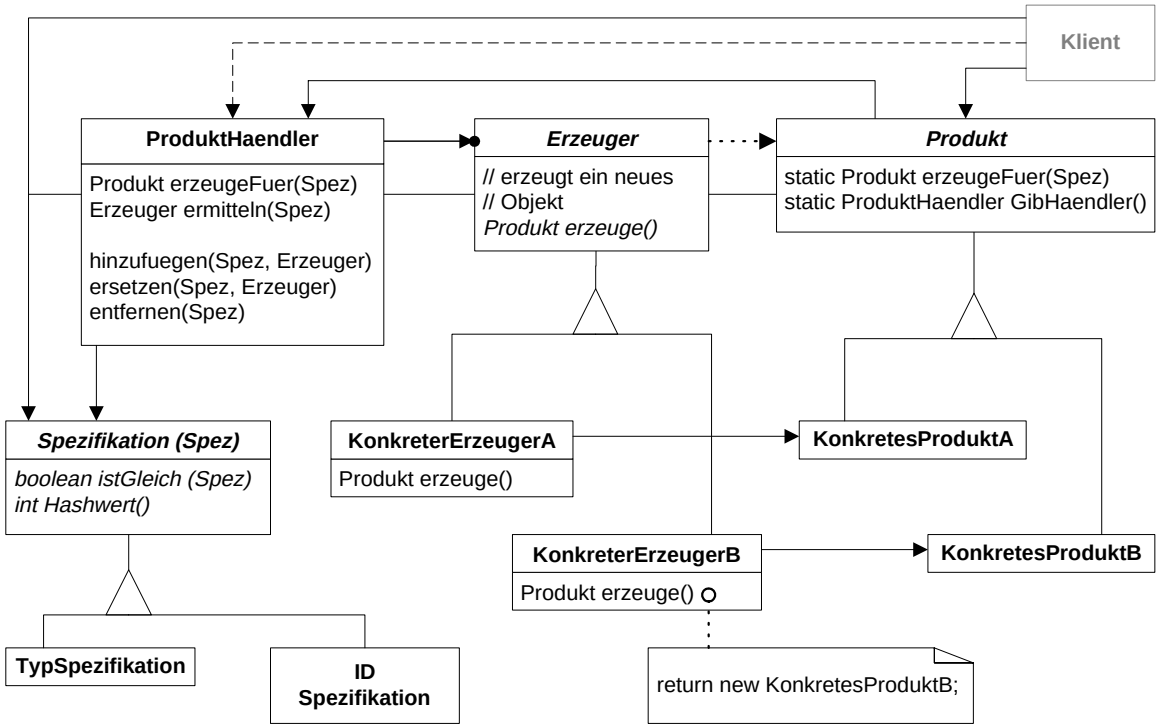


Abb. 6.4: Die Struktur des Objektvermittlers

Der gepunktete Pfeil von der aufgeschobenen Klasse `Erzeuger` zur aufgeschobenen Klasse `Produkt` soll verdeutlichen, daß die Klasse `Erzeuger` lediglich den Typ der Klasse `Produkt` kennen muß, um den Rückgabewert (einen Zeiger vom Typ `Produkt`) der Operation `erzeuge()` zu deklarieren. Der Compiler benötigt zur Übersetzung der Klasse `Erzeuger` kein Wissen über den Aufbau der Klasse `Produkt`. In C++ reicht daher die Deklaration (Klassenbekanntgabe) des Typs `Produkt` in der Klasse `Erzeuger` aus. Es entsteht somit keine zyklische Struktur zwischen den Klassen `ProduktHaendler`, `Erzeuger` und `Produkt`. Der Produkthändler wird i.d.R. nie direkt von einem Klienten benutzt. Die Objekterzeugung erfolgt stets über die Produktklasse. Dies soll durch den gestrichelten Pfeil vom Klienten zum Produkthändler symbolisiert werden.

Teilnehmer

- `Klient` (`Editor`)
  - erzeugt eine Spezifikation, die das gewünschte Produkt (Objekt) beschreibt.
  - initiiert den Erzeugungsprozeß, indem er entweder an der Klasse `Produkt` oder direkt an einem Produkthändler die Erzeugung anstößt. Das gewünschte Produkt wird dabei mittels einer Spezifikation beschrieben.
- `Produkt` (`FachlicherIAT`)
  - definiert die Schnittstelle der Klassenhierarchie, von der Produkte erzeugt (vermittelt) werden sollen.
  - kann unter Verwendung eines Produkthändlers Produkte erzeugen.
- `KonkretesProdukt` (`IATBetrag, IATDatum, IATKreditbetrag`)
  - repräsentiert ein konkretes Produkt.

- `ProduktHaendler`
  - verwaltet und pflegt die Zuordnung von Spezifikationen zu einem konkreten Produkt bzw. einem Erzeugerobjekt, das ein konkretes Produkt instanziieren kann.
  - läßt anhand einer Spezifikation ein konkretes Produkt erzeugen.
- `Erzeuger`
  - definiert die Schnittstelle, um ein Exemplar eines konkreten Produkts erzeugen zu können.
- `KonkreterErzeuger`
  - erzeugt genau ein konkretes Produkt.
- `Spezifikation` (ein fachlicher Wert)
  - beschreibt eine Dienstleistung, so daß der Produkthändler ein entsprechendes Objekt erzeugen kann.
  - muß eine eindeutige Identifikation der gewünschten Dienstleistung sein.

Interaktionen

Die folgende in Abb. 6.5 dargestellte Zusammenarbeit wird zur Laufzeit des Systems zwischen den Objekten etabliert:

- der Klient übergibt dem `ProduktHaendler` eine Spezifikation mit der Aufforderung, ein entsprechendes Objekt zu erzeugen.
- der `ProduktHaendler` verwaltet eine Hashtabelle, aus der er mit Hilfe der Spezifikation den entsprechenden `Erzeuger` ermittelt.
- der `ProduktHaendler` ruft an einem konkreten Erzeuger (z.B. einem Objekt vom Typ `KonkreterErzeugerA`) die Operation `erzeuge` auf.
- ein `KonkreterErzeuger` instanziiert durch Aufruf des `new` Operators ein konkretes Produkt und gibt dieses an den Produkthändler zurück.

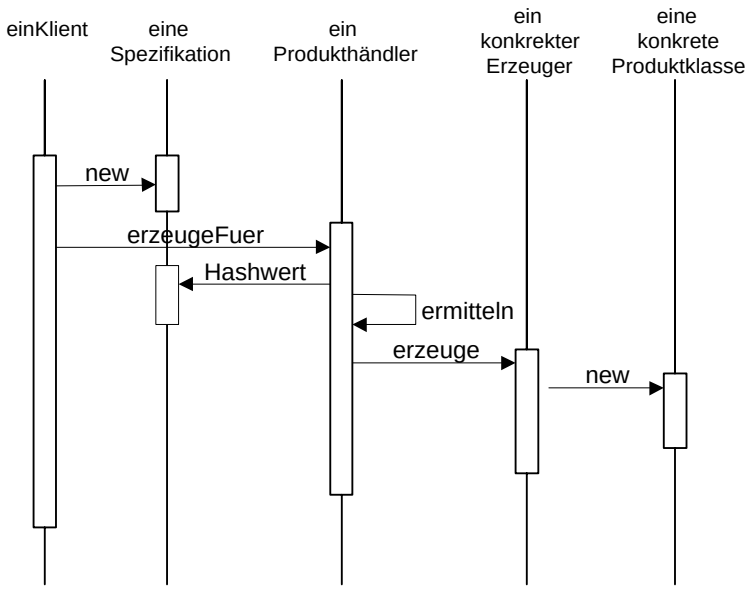


Abb. 6.5: Die Zusammenarbeit der Objekte im Entwurfsmuster

Konsequenzen

Mit dem Entwurfsmuster Produkthändler sind folgende Konsequenzen und Vorteile verbunden:

1. *Klienten können vollständig von konkreten Produkten entkoppelt werden,* da sie die Objekterzeugung nicht direkt unter Verwendung der `new` Operation, sondern indirekt über einen Produkthändler durchführen. Der Klient beschreibt das angeforderte Produkt über eine Spezifikation.

2. *Klienten können in tieferliegenden Schichten Produkte aus höherliegenden Schichten instanziieren, ohne daß dadurch eine Abhängigkeit entsteht.* Greift der Klient auf ein konkretes Produkt lediglich unter der Schnittstelle der aufgeschobenen Produktklasse zu, dann kann er mittels eines Händlers ein konkretes Produkt referenzieren, ohne die konkrete Produktklasse zu kennen (vgl. Abb. 5.10 und 5.11).

3. *Die konkrete Produktklasse, von der ein Objekt erzeugt werden soll, kann zur Laufzeit ermittelt werden.* Die gewünschte Dienstleistung wird dann in Form einer Spezifikation beschrieben und an den Produkthändler zur Erzeugung übergeben.

4. *Die Zuordnung einer Spezifikation zu einem konkreten Produkt ist konfigurierbar*. Unter Verwendung der am Produkthändler implementierten Operationen `hinzufuegen`, `entfernen` und `ersetzen` kann die Zuordnung statisch beim Start des Systems oder zur Laufzeit verändert werden. Dadurch lassen sich Anwendungssysteme auf der Basis von Rahmenwerken und Klassenbibliotheken konfektionieren.

5. *Kontextspezifisch lassen sich für eine Klassenhierarchie unterschiedliche Produkthändler verwenden*. Eine aufgeschobene Produktklasse könnte mehrere Produkthändler verwalten (in Abb. 6.4 der Einfachheit halber nicht dargestellt). Dadurch kann ein Anwendungssystem zwischen verschiedenen Produkthändlern wählen und sich kontextabhängig unterschiedliche Produkte erzeugen lassen.

   In dem vorgestellten Beispiel des Formulareditors könnte die Klasse `FachlicherIAT` zwei unterschiedliche Produkthändler verwalten. Abhängig von dem zu editierenden Formular würde dann entweder der allgemeine Produkthändler ausgewählt, bei dem der fachliche Wert `Betrag` mit einem `IATBetrag` editiert wird, oder ein spezieller Produkthändler, der einem `Betrag` den `IATKreditbetrag` zuordnet. Um Zuordnungen nicht mehrfach führen zu müssen, sollten die verschiedenen Produkthändler über eine Zuständigkeitskette (Chain of Responsibility) untereinander verbunden sein (vgl. [GOF95], S. 223).

6. *Die Klassenhierarchie der Produkte kann einfach weiterentwickelt werden*. Da der Klient lediglich von der aufgeschobenen Produktklasse abhängig ist, können Änderungen an konkreten Produktklassen einfach vorgenommen werden. Zudem lassen sich neue Produktklassen leicht zu einem bestehenden Rahmenwerk oder Anwendungssystem hinzufügen. Es müssen lediglich die Konfigurationsbeschreibungen, die die Zuordnung von Spezifikationen zu konkreten Produktklassen enthalten, angepaßt werden. Wenn zusätzlich die in Abschnitt 5.5 vorgestellte Trennung von Konzept- und Realisierungsteil eingehalten wird, ist nur eine Neuübersetzung der Realisierungsbibliothek notwendig.

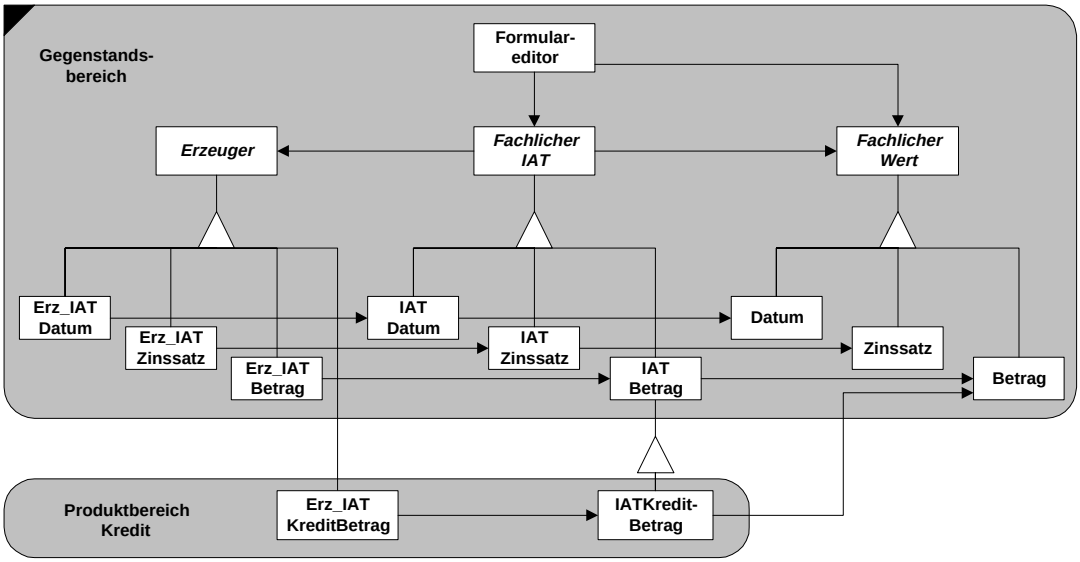


Abb. 6.6: Aufteilung der Erzeuger auf Gegenstands- und Produktbereich

Das Entwurfsmuster Produkthändler ermöglicht es somit, Rahmenwerke einer Schicht auch im Punkt Objekterzeugung zu schließen und trotzdem für Veränderungen offen zu halten. Dies war, wie die eingangs geführte Diskussion gezeigt hat, bisher nur bedingt möglich. Das Entwurfsmuster erleichtert daher die Weiterentwicklung von Klassenbibliotheken und Rahmenwerken und ermöglicht zudem eine einfache Konfektionierung von Anwendungssystemen (vgl. Abb. 6.6 und Abschnitt 6.4). Die Verwendung des Musters ist allerdings auch mit Nachteilen verbunden:

1. *Erhöhte Komplexität der Strukturen und Abhängigkeiten in Programmen*: der Erzeugungsprozeß von Objekten wird zugunsten einer Flexibilisierung nicht mehr statisch im Programmkode beschrieben. Dadurch kann die Objekterzeugung nicht mehr vom Übersetzer geprüft werden. Zudem ist der Programmkode für einen Softwareentwickler schwerer verständlich, da die Klasse, von der ein Objekt erzeugt werden soll, nicht mehr direkt angegeben ist.

2. *Erhöhte Fehleranfälligkeit*: falsche Konfigurationen, d.h. Zuordnungen von Spezifikationen zu konkreten Produktklassen, können zu Fehlern führen, die wegen der fehlenden `new` Operation im Programmkode nur schwer zu lokalisieren sind.

3. *Erweiterte Konfigurationsbeschreibung*: mittels eines Produkthändlers kann die Konfiguration eines Systems sowohl statisch als auch zur Laufzeit einfach an geänderte Bedingungen angepaßt werden. Um erstens eine Konfiguration überhaupt beschreiben zu können und zweitens dafür zu sorgen, daß bei einer C++ Implementierung der Linker konkrete Produktklassen beim Binden einer Bibliothek oder eines Anwendungssystems nicht unberücksichtigt läßt, sind spezielle Konfigurationsbeschreibungen notwendig. In der Konfigurationsbeschreibung wird die konkrete Produktklasse explizit referenziert. Diese Referenzierung ist notwendig, da der Aufruf der `new` Operation im Kode des Klienten fehlt (das Produkt wird über den Händler vermittelt) und konkrete Produktobjekte nur über die Schnittstelle der aufgeschobenen Produktklasse angesprochen werden, also nirgends eine Operation der konkreten Produktklasse aufgerufen wird. Im

Implementierungsteil werde ich verschiedene Alternativen für eine Konfigurationsbeschreibung diskutieren.

4. *Mehrdeutige Spezifikation*: abhängig vom Typ einer Spezifikation kann es vorkommen, daß zu einer Spezifikation mehrere konkrete Produkte existieren. Vermittelt ein Produkthändler beispielsweise zu einem Material ein passendes Werkzeug, so kann es in einem Anwendungssystem, das sich aus mehreren Produktbereichen zusammensetzt, mehrere passende Werkzeuge geben. Entweder muß dann durch zusätzliches Kontextwissen die Spezifikation eindeutig gemacht werden, oder es sind Regel anzugeben, wie dieser Konflikt zu lösen ist. Zusätzliches Kontextwissen wäre etwa der Produktbereich, in dem das Werkzeug implementiert sein soll. Regeln könnten sich auch auf den Spezialisierungsgrad der konkreten Produktklasse beziehen. Dieser könnte durch den Abstand beschrieben werden, den die konkrete Produktklasse in der Klassenhierarchie von der aufgeschobenen Produktklasse aufweist. Im Beispiel der Interaktionstypen für die fachlichen Werte hätte die Klasse `IATKreditbetrag` einen höheren Spezialisierungsgrad als die Klasse `IATBetrag`.

5. *Argumente für die Objekterzeugung*: benötigen konkrete Produktklassen bei der Objekterzeugung zusätzliche Argumente, so müssen diese mit übergeben werden. So braucht beispielsweise ein Interaktionstyp üblicherweise einen Verweis auf sein Vaterfenster. Wird zur Übergabe der Argumente der gängige Mechanismus einer Parameterliste verwendet, so lassen sich allerdings nur solche Argumente übergeben, die für alle Produktklassen gleich sind. Denn der Produkthändler kennt vom Typ her nur die aufgeschobene Produktklasse.

   Unterschiedliche Argumente lassen sich in generischen Parameterlisten übergeben (vgl. [MB96]). Dabei muß unbedingt sichergestellt sein, daß der Klient keine impliziten Annahmen über das zu erzeugende Produkt trifft. Ansonsten können Änderungen an der Konfiguration zu Fehlern führen, wenn ein anderes als das implizit angenommene Objekt instanziiert wird.

Implementierung

Die Implementierung des Entwurfsmusters muß sich mit den folgenden vier Aspekten beschäftigen: den Spezifikationen, den Erzeugern, dem Produkthändler und den Konfigurationsbeschreibungen. Die Zuordnung von Spezifikation zu konkretem Produkt (mittels eines Erzeugungsobjekts) kann sehr einfach in Form einer Hashtabelle realisiert werden. Für eine ausführlichere Diskussion dieser Thematik verweise ich daher auf [BR97].

1. *Spezifikationen* können sowohl als einfache Datentypen (z.B. Zahlen oder Zeichenketten) als auch in einer eigenständigen Klassenhierarchie realisiert werden. Die Modellierung einer Klassenhierarchie hat den Vorteil, daß sich eine einheitliche Schnittstelle für Spezifikationsobjekte in Form einer aufgeschobenen Spezifikationsklasse definieren läßt (vgl. Abb. 6.4). Produkthändler können dann der Basis dieser Schnittstelle implementiert werden und sind daher einfach um neue Spezifikationstypen erweiterbar.

2. *Erzeuger* lassen sich in Form von Prototypen oder speziellen Erzeugerobjekten realisieren. Unabhängig von der Methode, die gewählt wird, sollte die Realisierung nicht mit der händischen Implementierung einer Erzeugerklasse pro konkreter Produktklasse

verbunden sein. Die beiden unterschiedlichen Realisierungsformen stellen sich wie folgt dar:

- *Prototypen*: Bei dieser Form der Implementierung verwaltet der Produkthändler jeweils ein prototypisches Objekt pro konkretem Produkt. Soll ein neues Exemplar dieses Produkts erzeugt werden, so wird der Prototyp kopiert (vgl. [Gam92], S. 117).
- *Verwendung einer jeweils eigenständigen Erzeugerklasse.* Um die entsprechenden Klassen in C++ nicht einzeln kodieren zu müssen, können Templates zur Generierung eingesetzt werden. Dabei ist sowohl ein Template für die Klasse `Erzeuger` als auch eines für die Klasse `KonkreterErzeuger` notwendig. Das Template für die Klasse `Erzeuger` läßt sich mit der Klasse des aufgeschobenen Produktes konfigurieren, dasjenige für `KonkreterErzeuger` wird mit einer konkreten Produktklasse instanziiert. Konkrete Erzeuger implementieren die im `Erzeuger` aufgeschobene Operation `erzeuge`. Die nachfolgend beschriebenen Templates können somit für jede Produkthierarchie verwendet werden, bei denen die Konstruktoren der konkreten Produktklassen keine Parameter erwarten. Werden Parameter benötigt, so müssen die beiden Templates repliziert und mit zusätzlichen Templateparametern für den Argumenttyp versehen werden.

```
template<class ProduktTyp>
class Erzeuger {
public:
        virtual ProduktTyp * erzeuge() = 0;
};

template<class ProduktTyp, class KonkreterProduktTyp>
class KonkreterErzeuger : public Erzeuger<ProduktTyp> {
public:
        ProduktTyp * erzeuge(){return new KonkreterProduktTyp};
};
```

3. Produkthändler verwalten die Zuordnung von Spezifikationen zu konkreten Produkten in Form einer Hashtabelle. Da er die eigentliche Objekterzeugung anstößt, ist er produktspezifisch. Für jede Klassenhierarchie, deren Klassen über einen Produkthändler instanziierbar sein sollen, muß daher ein separater Produkthändler implementiert werden. Durch den Einsatz von Templates läßt sich aber ein allgemeiner Händler einmalig beschreiben. Ein entsprechendes Template läßt sich folgendermaßen formulieren:

```
template<class ProduktTyp>
class ProduktHaendler {
public:
      ProduktTyp * erzeugeFuer(const Spez &);

      void hinzufuegen(Spez *, Erzeuger<ProduktTyp> *);
      void ersetzen(const Spez &, Erzeuger<ProduktTyp> *);
      void entfernen(const Spez &);

private:
      Erzeuger<ProduktTyp> * ermittle(const Spez &);
      ...
};
```

4. Der Klient beschreibt gewünschte Dienstleistungen in Form von Spezifikationen. Die Konfiguration des Produkthändlers, d.h. die Zuordnung von einer Spezifikation zu einem konkreten Produkt, muß demnach für jedes Anwendungssystem definiert werden. Folgende Möglichkeiten bieten sich an:

- *Registrierungsobjekte*: Für Produkte, bei denen die Zuordnung von Spezifikation zu konkreter Produktklasse statisch definiert werden kann, lassen sich sogenannte Registrierungsobjekte einsetzen (vgl. [BR97]). Diese sind als Klassenvariablen der konkreten Produktklasse implementiert und werden daher beim Start eines Anwendungssystems automatisch erzeugt und initialisiert. Bei ihrer Initialisierung melden sie dann ein für ihre Produktklasse passendes Erzeugungsobjekt unter einer statisch festgelegten Spezifikation beim Produkthändler der aufgeschobenen Produktklasse an. In dem vorgestellten Beispiel der fachlichen Werte und Interaktionstypen würde das Registrierungsobjekt der Klasse `IATDatum` ein Erzeugungsobjekt vom Typ `KonkreterErzeuger<IATDatum>` instanziieren und beim Produkthändler der Klasse `FachlicherIAT` anmelden. Als Spezifikation wird dabei der Typ der Klasse `Datum` verwendet. Dieser läßt sich in C++ durch die Operation `typeid` ermitteln. Registrierungsobjekte sind allerdings mit den folgenden beiden Nachteilen verbunden: erstens sind Änderungen der Konfiguration mit Anpassungen der konkreten Produktklasse verbunden. Dies führt somit jeweils zu einer Öffnung des korrespondierenden Rahmenwerks respektive der Schicht. Unter diesem Gesichtspunkt sollte man Registrierungsobjekte nur einsetzen, wenn die Konfiguration eine hohe Stabilität aufweist oder Standardfälle beschreibt. Zweitens führt die Verwendung von Registrierungsobjekten nicht zur externen Referenzierung der konkreten Produktklasse (das Registrierungsobjekt referenziert die Produktklasse nur in der Produktklasse selbst). Der Linker kann somit nicht sicherstellen, daß alle zur Ausführungszeit des Systems benötigten Klassen zum System hinzugebunden werden.

  Befindet sich die Produktklasse in einer dynamischen Bibliothek, so reicht es aus, wenn eine beliebige Klasse der dynamischen Bibliothek vom Anwendungssystem referenziert wird, um die Produktklasse zu laden und das Registrierungsobjekt zu aktivieren. Denn der Inhalt einer dynamischen Bibliothek wird immer vollständig vom Betriebssystem geladen, auch dann, wenn nicht alle in der Bibliothek angebotenen Operationen vom Anwendungssystem auch tatsächlich benutzt werden.

  Befindet sich die Produktklasse hingegen nicht in einer dynamischen Bibliothek und wird sie auch an keiner anderen Stelle des Systems referenziert, dann ist es notwendig, daß die Produktklasse durch eine Pseudoreferenz zum Anwendungssystem hinzugebunden wird.
- `Konfigurationsskripten`: Die Konfiguration der einzelnen Produkthändler wird in einem speziellen Konfigurationsskript festgelegt, das mittels eines Präprozessors in C++ Kode übersetzt wird. Der Kode instanziiert ein Erzeugungsobjekt und registriert es unter der konfigurierten Spezifikation bei einem Produkthändler. Die Konfiguration des Produkthändlers läßt sich somit aus den konkreten Produktklassen herausziehen. Der generierte Kode für die Standardkonfiguration fachlicher Interaktionstypen sieht wie folgt aus:

```
void StandardIATKonfiguration(void) {
 ...

 ProduktHaendler<FachlicherIAT> * einHaendler =
      FachlicherIAT :: GibHaendler();

      // ordne Betrag einen IATBetrag zu, indem unter dem Typ der Klasse
      // Betrag (new TypSpezifikation(typeid(Betrag))) ein Erzeuger für den
      // IATBetrag (new KonkreterErzeuger<FachlicherIAT, IATBetrag>) einge-
      // tragen wird.
```

```
 einHaendler->fuegeHinzu(
      new TypSpezifikation(typeid(Betrag)),
      new KonkreterErzeuger<FachlicherIAT, IATBetrag>);

      // ordne einem Datum ein IATDatum zu (analog zu oben)
 einHaendler->fuegeHinzu(
      new TypSpezifikation(typeid(Datum)),
      new KonkreterErzeuger<FachlicherIAT, IATDatum>);
 ...
}
```

Soll diese Standardkonfiguration an besondere Bedürfnisse angepaßt werden, weil beispielsweise für einen `Betrag` der spezielle `IATKreditbetrag` zu verwenden ist, dann läßt sich diese Anpassung einfach in einem gesonderten Konfigurationsskript beschreiben:

```
KreditKonfiguration() {
 ...
 ProduktHaendler<FachlicherIAT> * einHaendler =
      FachlicherIAT :: GibHaendler();

      // ordne einem Betrag einen IATKreditBetrag zu
 einHaendler->ersetze(
      new TypSpezifikation(typeid(Betrag)),
      new KonkreterErzeuger<FachlicherIAT, IATKreditBetrag>);
 ...
}
```

Das Hauptprogramm des Anwendungssystems muß nun die beiden Operationen `StandardIATKonfiguration` und `KreditKonfiguration` aufrufen. Dadurch wird die notwendige Konfiguration des Produkthändlers vollzogen. Zudem wird sichergestellt, daß alle zur Ausführungszeit des Systems benötigten Klassen referenziert werden, so daß der Binder diese zum Anwendungssystem hinzufügt.

Eine Kombination aus Konfigurationsskripten und Registrierungsobjekten ermöglicht es, Standardfälle im Programmkode der jeweiligen Klassen zu hinterlegen und Änderungen in den Skripten zu beschreiben. Die Interaktionstypen könnten sich beispielsweise mittels Registrierungsobjekten für bestimmte fachliche Werte beim Händler `FachlicherIAT` anmelden. Dann muß ein Softwareentwickler im Konfigurationsskript lediglich die Änderungen beschreiben. Allerdings muß sichergestellt sein, daß die Abarbeitung von Registrierungsobjekten und Konfigurationsskripten zweistufig erfolgt. Zuerst muß ein System alle Registrierungsobjekte verarbeiten. Erst danach dürfen die Konfigurationsskripten abgearbeitet werden. Zudem müssen die Produktklassen, die durch Registrierungsobjekte konfiguriert werden, extern referenziert werden, oder sie müssen Bestandteil einer dynamischen Bibliothek sein.

Läßt sich kein sinnvoller Standardfall beschreiben oder werden für die Konstruktion des Anwendungssystems keine dynamischen Bibliotheken eingesetzt, sind Konfigurationsskripten den Registrierungsobjekten vorzuziehen. Sie sind einfacher zu verstehen, beschreiben die Konfiguration eines Systems an einem Ort und stellen sicher, daß die Produktklassen referenziert werden. Da sie Teil des anwendungsspezifischen Kodes sind, lassen sich die Konfigurationsskripten ausserdem einfach anpassen, ohne daß dazu ein Rahmenwerk geöffnet werden muß.

Beispielkode

Bereits durch die Abschnitte Motivation und Implementierung ist ein vollständiges Beispiel eingeführt worden. Ich werde daher die beiden folgenden Beispiele nur kurz skizzieren. Für eine ausführliche Diskussion verweise ich auf [BR97]. Das erste Beispiel betrifft die

Erzeugung von Objekten, die aus einer Datei oder einer relationalen Datenbank geladen werden (vgl. [RSB+97]). Das zweite Beispiel zeigt, wie die zu einem Material passenden Werkzeuge erzeugt werden können (vgl. [Rie95]).

In fast jedem größeren Anwendungssystem besteht die Notwendigkeit, Objekte persistent in einem Dateisystem oder einer relationalen Datenbank zu speichern und anschließend wieder zu laden. Beim Speichern wird dabei der dynamische Typ von jedem Objekt in Form einer Klassenidentifikation oder des Klassennamens mit abgespeichert. Beim Laden wird nun anhand der Identifikation ein „leeres“ Objekt erzeugt, daß anschließend mit den gespeicherten Werten gefüllt wird. Ein Händler könnte dabei die Zuordnung der Identifikation zu einem Erzeugerobjekt verwalten. Zur Erzeugung eines Objektes müßte ein Klient dann lediglich die Identifikation an den Händler übergeben. Existiert beispielsweise ein Protokoll `Serialisierbar` zum Speichern und Laden von Objekten, so könnte diese Klasse einen entsprechenden Händler verwalten. Dies stellt sich wie folgt dar:

```
class Serialisierbar {
public:
      // aufgeschobene Operationen zum Speichern und Laden
      ...

      static Serialisierbar * Serialisierbar ::
            erzeugeFuer( const long Klassennummer)
      {
       return einHaendler.erzeugeFuer(IDSpezifikation(Klassennummer));
      }
private:
      static ProduktHaendler<Serialisierbar> einHaendler;
};
```

Das zweite Beispiel skizziert den Einsatz des Musters bei der Erzeugung von Werkzeugen. In einem Anwendungssystem bearbeiten wir Materialien mit Werkzeugen. Je nach Konfiguration oder abhängig von Einflußfaktoren zur Laufzeit des Systems muß ein Material mit unterschiedlichen Werkzeugen bearbeitbar sein. Die dynamische Zuordnung eines Material zu einem Werkzeug sollte daher über einen Händler realisiert werden (vgl. Abschnitt 6.4).

Konzeptionelle Erweiterung des Entwurfsmusters

Die Erzeugerhierarchie ist in ihrer jetzigen Form darauf ausgelegt, bei jedem Erzeugungsauftrag ein neues Produkt zu instanziieren. Auf Grund von knappen Ressourcen oder infolge fachlicher Anforderungen kann es notwendig sein, bereits existierende Produkte im System zu vermitteln. Diese Anforderung läßt sich realisieren, indem der Händler nicht mit der aufgeschobenen Klasse `Erzeuger` arbeitet, sondern einen allgemeinen Vermittler verwendet. Dieser könnte dann als Spezialisierung die `Erzeuger` Hierarchie enthalten. Zusätzlich wird die (Template-) Klasse `Geber` eingeführt, die nach dem Singleton Muster (vgl. [GOF95], S. 127) genau ein Exemplar eines konkreten Produkt verwaltet. Eine auf den vorgestellten Templates basierende Klassenhierarchie gestaltet sich dann wie folgt (vgl. Abb. 6.7): die Klasse `Erzeuger` wird zur Klasse `Vermittler`, die Klasse `KonkreterErzeuger` zur Klasse `Erzeuger` und neu hinzu kommt die Klasse `Geber`.

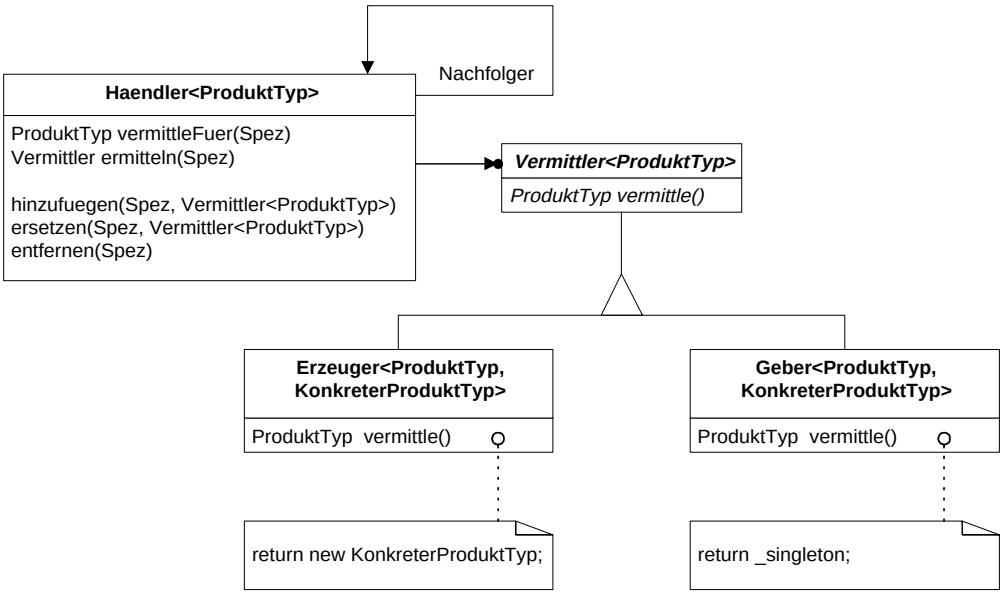


Abb. 6.7: Produkthändler mit Erzeuger- und Geberklassen

Das Spektrum der Spezifikationen kann, da ein Händler nur das Protokoll der aufgeschobenen Klasse `Spezifikation` kennen muß, beliebig erweitert werden. Neben den bereits vorgestellten Spezifikationen, die eine Klassenidentifikation oder den dynamischen Typ (`typeid`) eines Objektes verwalten, hat sich die Verwendung von Spezifikationen bewährt, die nicht den konkreten Typ eines Objektes führt, sondern den Typ einer Oberklasse. Dadurch kann ein gesamter Klassenbaum einem konkreten Produkt zugeordnet werden. Eine solche Spezifikation könnte bei einem Händler, der Werkzeuge vermittelt, beispielsweise den Typ `Unterschriftsformular` einem `Formulareditor` zuweisen (vgl. Abb. 5.19). Dadurch würde allen Objekten, die Exemplare einer Unterklasse von `Unterschriftsformular` sind, automatisch der `Formulareditor` als Werkzeug zugewiesen. Spezielle Zuordnungen der Unterklassen können durch einen zweiten Händler realisiert werden, der unerfüllbare Anforderungen an den ersten über eine Zuständigkeitskette (Chain of Responsibility Muster, vgl. Punkt 4 im Abschnitt Konsequenzen) weitergibt. Will ein Softwareentwickler dem Material `Vertragsformular` das Werkzeug `Vertragseditor` zuweisen, so stellt sich die Situationen in einem Objektdiagramm wie folgt dar:

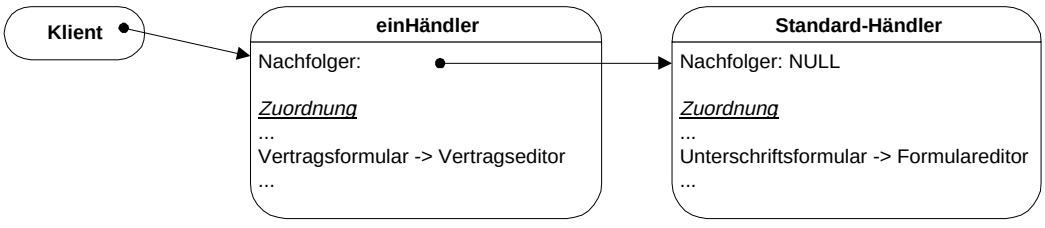


Abb. 6.8: Zwei über eine Chain of Responsibility miteinander verbundene Händler

Der letzte Händler in der Kette sollte immer einen Standardfall verwalten, damit keine Standardspezifikationen beim Händler registriert werden müssen. Dies vereinfacht die Konfigurationsbeschreibungen erheblich.

In der hier vorgestellten Form kapselt ein Produkthändler immer eine spezifische Produkthierarchie, d.h. über den Händler können jeweils nur Produkte vermittelt werden, die als Unterklasse der aufgeschobenen Produktklasse realisiert sind. Ein allgemeiner Händler, der jedes Produkt in einem System vermitteln kann, läßt sich in einer typisierten Programmiersprache wie C++ nur realisieren, wenn ein gemeinsame Wurzelklasse existiert.

Die vorgestellten Konfigurationsskripten geben einem Softwareentwickler die Möglichkeit, ein Anwendungssystem sowohl zur Übersetzungszeit als auch zur Laufzeit zu konfektionieren. So kann er beispielsweise festlegen, welcher fachliche Wert mit welchem Interaktionstyp dargestellt und editiert wird, welches Material mit welchem Werkzeug bearbeitet werden kann oder welches Material sich aus der Datenbank laden läßt. Da die Konfigurationsskripten jeweils den Typ des zu vermittelnden Produkts referenzieren, stellen sie zudem sicher, daß die entsprechenden Produktklassen von einem Linker zu einer Rahmenwerksbibliothek oder zu einem Anwendungssystem hinzugebunden werden. Die Konfigurationsskripten sind daher geeignet, die Anwendungssysteme eines Einsatzkontextes basierend auf den verschiedenen Produktbereichen zu konfektionieren. Darauf werde ich detailliert in Abschnitt 6.4 eingehen.

Die beiden nächsten Abschnitte thematisieren zunächst das Konzept der Rolle. Rollen werden bei der Entwicklung von Rahmenwerken und Anwendungssystemen eingesetzt, um die Begriffs- und Klassenbildung aus unterschiedlichen anwendungsfachlichen Blickwinkeln zu ermöglichen.

6.2 Rollen

In Abschnitt 4.4 habe ich bei der Motivation des Gegenstandsbereichs ausgeführt, daß sich derselbe fachliche Gegenstand aus der Sicht verschiedener Produktbereiche unterschiedlich modellieren läßt. Beispielsweise realisieren die beiden Konzepte Kreditnehmer (Produktbereich Kredit) und Anleger (Produktbereich Anlage) das Konzept des Kunden aus unterschiedlichen Blickwinkeln. Ein Kreditnehmer verwaltet etwa die Sicherheiten, die im Rahmen der Kreditvergabe an die Bank übereignet werden. Derartige Informationen sind für das Konzept Anleger ohne Bedeutung und werden daher auch nicht an ihm modelliert. Zudem gilt, daß die aus Sicht eines Produktbereichs modellierten Eigenschaften dynamische Aspekte eines fachlichen Gegenstands darstellen können. Ein Kunde wird beispielsweise nur dann zum Kreditnehmer in einer Bank, wenn er einen Kredit in Anspruch nimmt. Ansonsten weist er die mit einem Kreditnehmer verbundenen Eigenschaften nicht auf.

Eine naive auf den Mitteln der Objektorientierung basierende Lösung ist die Modellierung der drei Begriffe Kunde, Kreditnehmer und Anleger als Klassen. Das Konzept Kunde wäre Bestandteil des Gegenstandsbereichs, das des Kreditnehmers Teil des Produktbereichs Kredit und das des Anlegers Element des Produktbereichs Anlage. Da die Konzepte Anleger und Kreditnehmer das Konzept des Kunden erweitern und Anleger- bzw. Kreditnehmerobjekte

anstelle eines Kundenobjekts verwendbar sind (ein Kreditnehmer „ist-ein“ Kunde), wird softwaretechnisch die Klasse `Kunde` als Generalisierung der Klassen `Kreditnehmer` und `Anleger` realisiert. Die Klasse `Kunde` modelliert dabei Eigenschaften wie die `Kundennummer` oder eine Operation `GibGesamtEngagement` (vgl. Abb. 6.9).

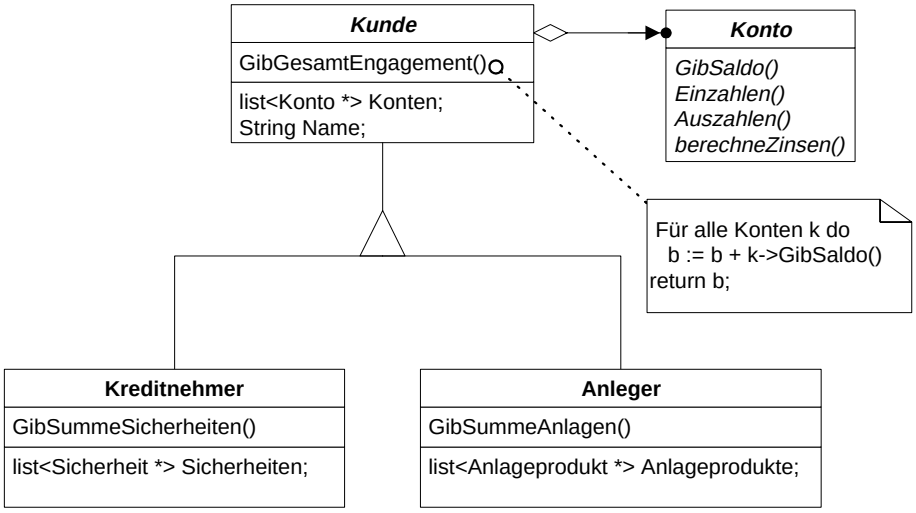


Abb. 6.9: Produktbereichsspezifische Spezialisierung eines Kunden mittels Vererbung

Wird nun der Kunde Hans Müller zur Laufzeit eines Systems als Anleger und als Kreditnehmer verwaltet (d.h. es wird von beiden Klassen ein Objekt instanziiert), so führt die vorgestellte Realisierung zur Verdopplung der in der Klasse `Kunde` modellierten Attribute. Denn sowohl ein Objekt der Klasse `Anleger` als auch ein Objekt der Klasse `Kreditnehmer` besitzen die Attribute `Konten` und `Name`. Es sind daher aufwendige Synchronisierungsmechanismen notwendig, um die fachliche Identität der Objekte zu wahren. Änderungen an einem Objekt müssen jeweils in dem anderen Objekt nachgezogen werden, um mittels Wertegleichheit ihre fachliche Identität zu simulieren.

Die technische Umsetzung in Form von Mehrfachvererbung, bei der eine Klasse `KreditnehmerUndAnleger` als Unterklasse der beiden Klassen `Anleger` und `Kreditnehmer` gebildet wird, verhindert zwar die Verdopplung der Attribute der Klasse `Kunde`, führt aber bei n Klassen zu n Fakultät möglichen Unterklassen. Diese Flut von Unterklassen wird schnell sehr unübersichtlich. Die kombinierten Klassen werden zudem von allen Produktbereichen verwendet, die an der Kombination beteiligt sind (bei der Klasse `KreditnehmerUndAnleger` die beiden Produktbereiche Kredit und Anlage). Die dadurch entstehenden Abhängigkeiten zwischen den einzelnen Produktbereichen führen dazu, daß diese nicht mehr getrennt voneinander analysiert, modelliert und konstruiert werden können. Weiterhin ist diese Lösung mit dem Problem der Objektmigration verbunden. So ist beispielsweise ein Kunde nicht notwendigerweise gleichzeitig Kreditnehmer und Anleger. Er wird vielleicht zunächst nur als Anleger betrachtet. Erst wenn er einen Kredit in der Bank aufnimmt, wird er zusätzlich auch zum Kreditnehmer. Dabei muß nun ein neues Objekt der Klasse `KreditnehmerUndAnleger` erzeugt werden, in das die Attribute des Anlegerobjekts übertragen werden. Weiterhin müssen alle Referenzen, die auf das „alte“ Anlegerobjekt verweisen, auf

das neue Objekt gesetzt werden. Dieser Prozeß der Referenzänderung ist ohne geeignete Unterstützung (beispielsweise `become:` in Smalltalk) nicht durchführbar.

In der Literatur wird die Problematik, fachliche Gegenstände abhängig vom jeweiligen Blickwinkel (Produktbereich) und vom Zeitpunkt der Betrachtung unterschiedlich zu modellieren, unter dem Begriff der Rolle diskutiert (vgl. [GSR95], [KØ96] und [Ree96]). Kristensen & Østerbye definieren in [KØ96] den Begriff der Rolle folgendermaßen:

> "A role of an object is a set of properties which are important for an object to be able to behave in a certain way expected by a set of other objects." ([KØ96], S. 143)

Ein Werkzeug, das der Bearbeitung von Kundensicherheiten dient, arbeitet beispielsweise mit Kundenobjekten unter dem speziellen Blickwinkel des Kreditnehmers. Die Definition von Kristensen & Østerbye impliziert zwei weitere Eigenschaften von Rollen, die sie wie folgt beschreiben:

- Ein Objekt kann mehrere Rollen aufweisen, d.h. es kann aus mehreren unterschiedlichen Blickwinkeln verwendet werden.
- Eine Rolle ist ein integraler Bestandteil des Objekts, der das Objekt unter einem bestimmten Blickwinkel zeigt.

Die beiden Merkmale führen zu einer Reihe weiterer Frage, etwa ob eine Rolle eigenständig existieren kann oder welche Eigenschaften Objekte neben ihren rollenspezifischen Eigenschaften noch aufweisen. Im nächsten Abschnitt erfolgt daher eine Diskussion der Merkmale, die mit einer Rolle verbunden sind.

6.2.1 Merkmale von Rollen

Eine Rolle repräsentiert die Menge der Attribute und Operationen, die ein Gegenstand unter einem bestimmten Blickwinkel aufweist. So verwaltet etwa die Rolle Kreditnehmer die Sicherheiten, die ein Kunde im Zuge der Kreditvergabe an die Bank übereignet. Im Kontext der Objektorientierung wird eine Einheit bestehend aus Attributen und Operationen als Objekt bezeichnet. Rollen lassen sich daher als Objekte modellieren. Ein solches Objekt wird als Rollenobjekt bezeichnet. Es ist definiert als:

> **Begriff 6.2 Rollenobjekt**
>
> Ein Rollenobjekt repräsentiert die Menge der Attribute und Operationen, die ein Gegenstand unter einem anwendungsbezogenen Blickwinkel aufweist.

Aus dieser Definition ergibt sich eine weitere wichtige Eigenschaft von Rollen (vgl. [KØ96], S. 144). Rollenobjekte verfügen über eine eigene technische Identität, die sich von der anderer Rollenobjekte unterscheidet. So ist beispielsweise das Rollenobjekt Kreditnehmer von dem Rollenobjekt Anleger verschieden, auch wenn beide Rollenobjekte denselben Kunden modellieren.

Rollenobjekte sind, obwohl sie über eine eigene technische Identität, einen eigenen Zustand und eigene Operationen verfügen, alleine nicht lebensfähig. Sie können immer nur in Verbindung mit dem Objekt (Gegenstand) existieren, das die betreffende Rolle gerade spielt. So macht die Rolle Kreditnehmer ohne einen Kunden, der die Rolle spielt, keinen Sinn. Um im weiteren ein Rollenobjekt von dem Objekt zu unterscheiden, das die Rolle spielt, führe ich den Begriff des Kernobjekts ein. Das Kernobjekt ist wie folgt definiert:

Begriff 6.3 Kernobjekt

Ein Kernobjekt ist ein Objekt, das in einem System verschiedene Rollen spielen kann. Es umfaßt diejenigen Zustände und Operationen, die unabhängig von einem spezifischen Blickwinkel von all seinen Rollenobjekten nach außen repräsentiert werden. Ein Kernobjekt kann unabhängig von seinen Rollenobjekten existieren.

Bei einem Kunden umfaßt das Kernobjekt beispielsweise seine Adresse und Kundennummer sowie sein Girokonto. Sowohl das Kernobjekt als auch die Rollenobjekte sind dabei nach dem Prinzip des Information Hiding realisiert. Sämtliche Attribute werden ausschließlich über Operationsaufrufe manipuliert. Ein Rollenobjekt hat somit keinen direkten Zugriff auf die Attribute seines Kernobjekts. Da ein Kernobjekt unabhängig von seinen Rollenobjekten lebensfähig sein muß, darf es nicht mittels Operationsaufrufen auf sie zugreifen. Andernfalls könnten über das Kernobjekt indirekte Abhängigkeiten zwischen den Rollenobjekten entstehen, die wiederum zu Abhängigkeiten zwischen den Produktbereichen führen würden.

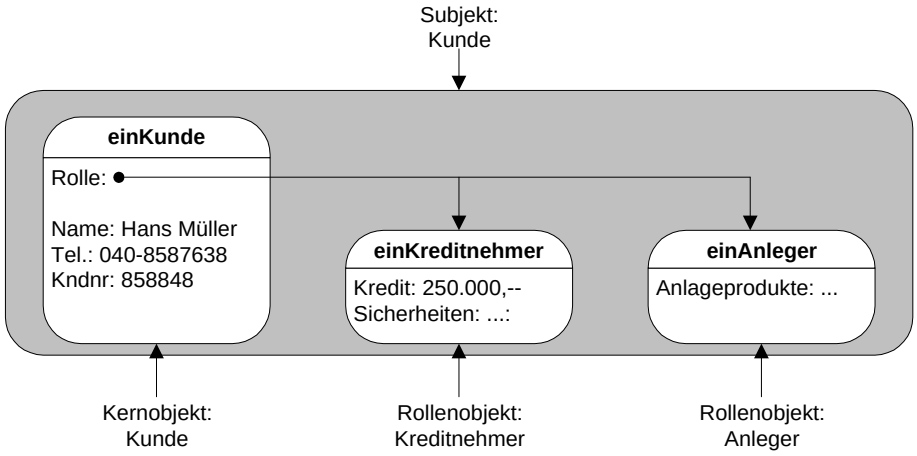


Abb. 6.10: Ein Kernobjekt Kunde mit den Rollenobjekten Kreditnehmer und Anleger

Die technische Unterscheidung zwischen Kern- und Rollenobjekt führt zur Ausdifferenzierung des Identitätsbegriffs. Das Kernobjekt verfügt, wie seine Rollenobjekte auch, über eine eigene technische Identität, die sich von denen der Rollenobjekte unterscheidet. Daneben bildet das Kernobjekt zusammen mit seinen Rollenobjekten eine logische Einheit, die als ganzes eine fachliche Identität besitzt. Existiert beispielsweise ein Kernobjekt Kunde mit zwei Rollenobjekten Kreditnehmer und Anleger, so besitzen die drei Objekte sowohl ihre eigene technische Identität als auch eine gemeinsame fachliche Identität, die der logischen Einheit bestehend aus dem Kernobjekt Kunde und seinen Rollenobjekten anhaftet (vgl. Abb. 6.10). In

Anlehnung an [KØ96] bezeichne ich diese logische Einheit als Subjekt[21]. Der Begriff des Subjekts wird wie folgt definiert:

Begriff 6.4 Subjekt

Als Subjekt wird die logische Einheit aus Kernobjekt und allen dem Kernobjekt momentan anhaftenden Rollen bezeichnet. Ein Subjekt verkörpert eine fachliche Identität, die im weiteren Subjektidentität genannt wird.

Bezogen auf die unterschiedlichen Identitätsformen lassen sich nun folgende Aussagen treffen:

- Ein Rollenobjekt ist technisch verschieden von seinem Kernobjekt.
- Ein Rollenobjekt besitzt dieselbe Subjektidentität wie sein Kernobjekt.
- Technische Identität impliziert Subjektidentität; die Umkehrung gilt nicht.

Ähnliche Gegenstände werden in der Objektorientierung durch einen Abstraktionsprozeß auf den Begriff gebracht und anschließend im softwaretechnischen Modell durch eine oder mehrere Klassen modelliert. Analog lassen sich ähnliche Rollen, die von einem Gegenstand im Anwendungsbereich gespielt werden können, auf ihre nominale Essenz bringen. Ich führe daher die beiden Begriffe Kern- und Rollenkonzept ein. Sie sind wie folgt definiert:

Begriff 6.5 Kern- und Rollenkonzept

Ein Kernkonzept ist eine Abstraktion über ähnliche Kernobjekte, ein Rollenkonzept eine Abstraktion über ähnliche Rollenobjekte. Mit der Definition eines Kern- bzw. Rollenkonzepts ist in einer typisierten Programmiersprache die Etablierung eines Typs verbunden (analog zur Klasse).

Rollenobjekte erweitern ein Kernobjekt zur Laufzeit des Systems unter einer anwendungsspezifischen Sicht. Da Rollenkonzepte die Abstraktion über ähnlichen Rollenobjekten darstellen, modellieren sie demnach diesen Blickwinkel. Diese Überlegung soll in einem Ergebnis festgehalten werden:

Ergebnis 6.1

Rollenkonzepte modellieren die Erweiterung eines Kernkonzepts unter einem anwendungsspezifischen Blickwinkel.

Der anwendungsspezifische Blickwinkel läßt sich im Kontext dieser Arbeit konkretisieren. Er kann interpretiert werden als die Sicht, die ein Produktbereich auf ein Kernkonzept des Gegenstandsbereichs einnimmt. Der Einsatz von Kern- und Rollenkonzepten ermöglicht es

[21] Kristensen & Østerbye unterscheiden in [KØ96] zusätzlich zwischen einem ‘subject’ und einem ‘closured subject’. Ein ‘subject’ umfaßt das Kernobjekt plus eine beliebige Anzahl von Rollenobjekten, die es gerade spielt. Der Begriff ‘closured subject’ dagegen repräsentiert das Kernobjekt plus alle seine Rollenobjekte. Der von mir definierte Begriff Subjekt stimmt in seiner Definition demnach mit dem Begriff ‘closured subject’ von Kristensen & Østerbye überein. Ich verzichte in dieser Arbeit auf die Unterscheidung in ‘subject’ und ‘closured subject’, da sich im RWG-Kontext für sie keine fachliche Notwendigkeit ergeben hat.

somit, fachliche Gegenstände des Anwendungsbereichs in verschiedenen Produktbereichen unabhängig voneinander zu modellieren.

Um das Rollenkonzept respektive die Erweiterung eines Kernkonzepts mittels Rollen vom herkömmlichen Klassenkonzept und der Vererbung unterscheiden zu können, führe ich eine spezielle Notation ein (vgl. Abb. 6.11). Ein Rechteck mit oben abgerundeten Ecken symbolisiert ein Rollenkonzept. Kernkonzepte werden wie Klassen notiert. Die Erweiterung eines Kernkonzeptes durch Rollen wird mittels einer Linie, die mit einem Halbkreis versehen ist, dargestellt.

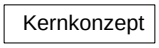
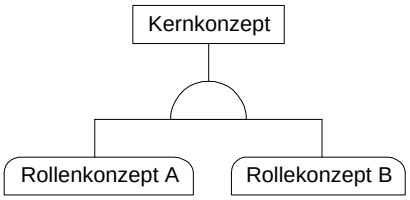
Graphisches Element	Interpretation
	Kernkonzepte werden (analog zu Klassen) durch ein Rechteck dargestellt.
	Rollenkonzepte werden durch ein Rechteck mit oben abgerundeten Ecken symbolisiert.
	Die Erweiterung eines Kernkonzeptes durch ein Rollenkonzept wird durch eine Linie mit einem Halbkreis symbolisiert.

Abb. 6.11: Kern- und Rollenkonzepte mit ihren graphischen Symbolen

Die Erweiterung von Kernkonzepten durch Rollen bildet eine Hierarchie, die ich im weiteren als Rollenhierarchie bezeichnen werde. Rollenhierarchien unterscheiden sich von Klassenhierarchien in folgender Form: die im Kernkonzept definierten Attribute werden von den Rollenkonzepten nicht im klassischen Sinne geerbt. Ein Rollenobjekt verfügt nicht über eigene Exemplarvariablen für die im Kernkonzept definierten Attribute. Alle Rollenobjekte, die zur Laufzeit an dasselbe Kernobjekt gebunden werden, verwenden die im Kernkonzept definierten Attribute gemeinsam. So teilen sich beispielsweise die Rollenobjekte `Kreditnehmer` und `Anleger` die Attribute `Name`, `Telefonnummer` und `Kundennummer` des Kernkonzepts `Kunde`. Die Hierarchie soll ausdrücken, welches Kernkonzept sich durch welche Rollenkonzepte erweitern läßt. Eine Rollenhierarchie ist definiert als:

Begriff 6.6 Rollenhierarchie

Die Erweiterung eines Kernkonzeptes durch ein oder mehrere Rollenkonzepte bildet eine Rollenhierarchie.

Auf der Basis der bisher eingeführten Begriffe, der beispielhaften Rollenhierarchie der Abb. 6.12 und des Subjekts „Hans Müller“ der Abb. 6.13 sollen nun die Eigenschaften von Kern- und Rollenobjekten respektive von Kern- und Rollenkonzepten unter Berücksichtigung der einschlägigen Literatur präzisiert werden (vgl. [GSR95], [KØ96], [OKK+96] und [Ree96]).

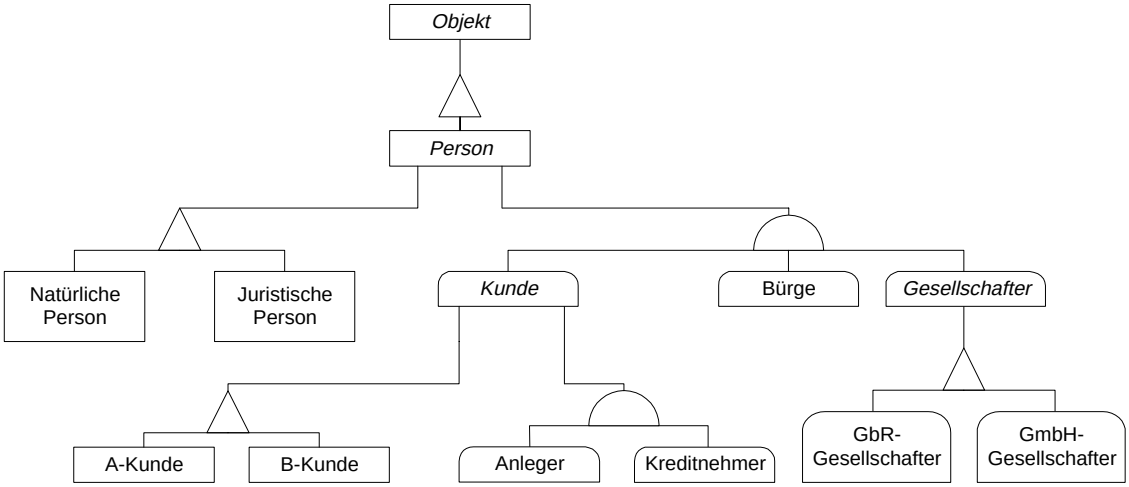


Abb. 6.12: Ein Beispiel für eine Rollenhierarchie aus dem Bankenumfeld

Das in Abb. 6.13 dargestellte Objektdiagramm des Subjekts „Hans Müller“ zeigt die natürliche Person „Hans Müller“ mit seinen Rollen GmbH-Gesellschafter und Kunde. Als Kunde tritt die Person „Hans Müller“ in der Rolle des Kreditnehmers auf.

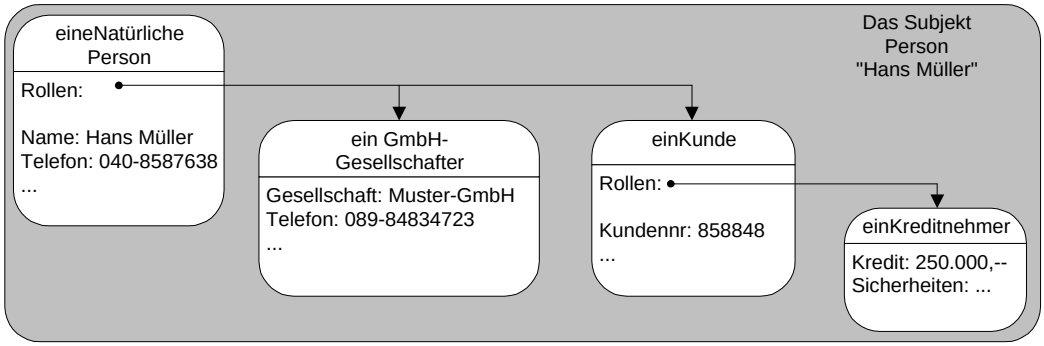


Abb. 6.13: Das Subjekt Person „Hans Müller“ dargestellt als Objektdiagramm

Die Eigenschaften von Rollen lassen sich wie folgt detaillieren:

(1) *Rollenobjekte erweitern Kernobjekte dynamisch, d.h. sie können zur Laufzeit einem Kernobjekt hinzugefügt und von diesem wieder entfernt werden.* So kann ein Kunde in einer Bank erst nur die Rolle Anleger spielen, wenn er einen Kredit in Anspruch nimmt, zum Kreditnehmer werden, und wenn er diesen abbezahlt hat, die Rolle wieder aufgeben.

(2) *Die Attribute des Kernobjekts werden von allen Rollenobjekten, die dem Kernobjekt anhaften, geteilt*. So haben alle Rollenobjekte, die ein Kernobjekt vom Typ `Person` erweitern, auf die in dem Kernkonzept `Person` definierten Eigenschaften, wie etwa das Attribut `Name`, über eine Operationenschnittstelle Zugriff.

(3) *Jedes Rollen- und Kernobjekt verfügt über eine eigene technische Identität*. So stellen die Rollenobjekte `GmbH-Gesellschafter` und `Kunde` sowie das erweiterte Kernobjekt `Person` technisch drei Identitäten dar.

(4) *Rollen- und Kernobjekte verfügen über eine gemeinsame fachliche Identität*. Diese wird, in Anlehnung an den Begriff des Subjekts, der die Zusammenfassung des Kernobjekts mit allen seinen Rollenobjekten repräsentiert, als Subjektidentität bezeichnet.

(5) *Rollen können selbst wieder Rollen spielen*, d.h. die rekursive Aufteilung eines Rollenobjekts in Kern- und Rollenobjekte ist möglich. Rollenkonzepte, die durch Rollenkonzepte erweitert werden, übernehmen eine Doppelfunktion: sie sind Kern- und Rollenkonzept in einem. Das obere Kernkonzept einer Rollenhierarchie bezeichne ich in Anlehnung an [GSR95] als Wurzelkonzept, das entsprechende Objekt als Wurzelobjekt. Beispielsweise ist das Rollenkonzept `Kunde` des Kernkonzepts `Person` wieder Kernkonzept für die Rollenkonzepte `Kreditnehmer` und `Anleger`. Das Kernkonzept `Person` ist dabei das Wurzelkonzept. Den Begriff des Wurzelkonzeptes fasse ich wie folgt:

> **Begriff 6.7 Wurzelkonzept**
>
> Ein Konzept, das innerhalb einer Rollenhierarchie kein Kernkonzept erweitert, selbst aber durch Rollenkonzepte erweitert wird, heißt Wurzelkonzept.

Konzepte, die sowohl Kern- als auch Rollenkonzept sind, werden als Rollenkonzept bezeichnet, wenn die Beziehung zu ihrem Kernkonzept diskutiert wird. Steht hingegen die Beziehung zu ihren Rollenkonzepten im Vordergrund, spreche ich von dem Konzept als Kernkonzept. Analoges gilt für Kern- und Rollenobjekt. Konzepte, die sowohl ein Kern- als auch ein Rollenkonzept darstellen, werden analog zu [GSR95] graphisch als Rollenkonzept symbolisiert.

(6) *Wurzelobjekte sind ohne ein erweiterndes Rollenobjekt lebensfähig.* So kann das Wurzelobjekt `Person` allein existieren.

(7) *Ein Kernobjekt kann eine Rolle mehrmals spielen, d.h. ein Kernobjekt läßt sich mit mehreren Rollenobjekten desselben Rollenkonzepts erweitern*. So kann eine Person Gesellschafter mehrerer GmbHs sein. Sie spielt dann die Rolle des GmbH-Gesellschafters mehrmals.

(8) *Rollenkonzepte können Attribute von Kernkonzepten überladen*. Kern- und Rollenkonzepte können gleichnamige Attribute definieren. So könnten sowohl das Kernkonzept `Person` als auch die Rollenkonzepte `GmbH-` und `GbR-Gesellschafter` das Attribut `Telefonnummer` mit entsprechenden Operationen zu ihrer Manipulation definieren.

(9) *Rollenobjekte können auf ihre eigenen und auf die Eigenschaften ihres Kernobjekts zugreifen*. Das Rollenobjekt `Kreditnehmer` kann beispielsweise über eine Operationenschnittstelle auf die Sicherheiten und die Kundennummer von `Kunde` und auf den Namen von `Person` zugreifen.

(10) *Aus der Sicht eines Klienten umfaßt die Schnittstelle eines Rollenobjekts sowohl die für diese Rolle spezifischen Operationen als auch die Operationen seines Kernobjekts. Dieses Prinzip setzt sich für Rollen von Rollen rekursiv fort.* Verwendet ein Klient beispielsweise ein Rollenobjekt `Anleger`, so kann er es durch Aufruf entsprechender Operationen nach der Kundennummer (Kernkonzept `Kunde`) und nach dem Namen

(Wurzelkonzept `Person`) fragen. Der Klient kann aber nicht auf die Sicherheiten zugreifen, die im Rollenkonzept `Kreditnehmer` festgelegt sind.

Hat ein Rollenkonzept ein Attribut des Kernkonzepts überladen (vgl. Punkt (8)), so ist am Rollenobjekt nur die überladene Eigenschaft sichtbar. Bearbeitet beispielsweise ein Klient ein Rollenobjekt `GbR-Gesellschafter`, so kann er nur auf die Telefonnummer des Gesellschafters, nicht auf die im Wurzelkonzept `Person` vereinbarte Telefonnummer zugreifen. Will ein Klient die Telefonnummer der Person erfragen, so muß er dazu das Subjekt unter dem Typ des Wurzelkonzeptes bearbeiten.

(11) *Rollenobjekte können in einem System anstelle ihres Kernobjekts verwendet werden.* Ein Rollenkonzept erweitert die Eigenschaften eines Kernkonzepts. Daher verfügt ein Rollenobjekt über dieselben Eigenschaften wie sein Kernobjekt (vgl. Punkt (2)). Das Rollenobjekt kann demnach anstelle des Kernobjekts verwendet werden. In diesem Sinne besteht zwischen dem Rollenkonzept und seinem Kernkonzept eine *ist-ein* Beziehung.

Erwartet beispielsweise eine Operation ein Kundenobjekt, um von ihm durch Aufruf einer Operation die Kundennummer zu erfragen, so kann anstelle des Kundenobjekts auch ein Kreditnehmer- oder Anlegerobjekt verwendet werden. Denn der Klient hat über beide Rollenobjekte Zugriff auf die Operationen des Kernobjekts Kunde (vgl. Punkt (10)). Das Rollenkonzept `Kreditnehmer` *ist-ein* Kernkonzept `Kunde`.

(12) *Kernobjekte können anstelle eines Rollenobjekts vom Typ R verwendet werden, wenn das Kernobjekt bereits mit dem Rollenobjekt vom Typ R erweitert ist (Rückkonvertierung, engl. dynamic cast).* Ein Kunde kann genau dann anstelle eines Kreditnehmers verwendet werden, wenn er bereits Kreditnehmer der Bank ist.

(13) *Rollenkonzepte können bereits existierende Operationen des Kernkonzepts erweitern*. Die Erweiterung von Operationen läßt sich fachlich motivieren. Da Rollenkonzepte über spezielleres Wissen bezüglich der Begriffseinheit aus Kern- und Rollenkonzept verfügen, können sie auch erweiterte Realisierungen von Operationen anbieten. Dies soll an einem Beispiel verdeutlicht werden. Das Kernkonzept Person bietet eine Operation `aendereAdresse` an. Bei Adreßänderung an einem Kreditnehmer muß, sobald die Kreditsumme einen festgesetzten Betrag überschreitet, eine entsprechende Mitteilung an eine kreditüberwachende Einrichtung verschickt werden. Diese Anforderung läßt sich erst mittels Erweiterung der Operation `aendereAdresse` im Rollenkonzept Kreditnehmer realisieren, da im Kernkonzept Person keine Informationen über die Kreditsumme modelliert werden können. Wie das Beispiel zeigt, ist die Erweiterung von Operationen auch über mehrere Hierarchiestufen möglich.

(14) *Rollen- und Klassenhierarchien sind miteinander kombinierbar*. Einerseits lassen sich Kern- und Rollenkonzepte mittels Vererbung spezialisieren. Andererseits können Kernkonzepte selbst auch wieder Spezialisierungen einer Klasse sein. Eine Spezialisierung verfügt, wie bei der Vererbung üblich, über alle Eigenschaften ihrer Generalisierung. So wird beispielsweise das Kernkonzept `Person` durch die Klassen `NatürlichePerson` und `JuristischePerson` spezialisiert. Das Rollenkonzept `Gesellschafter` ist die Generalisierung der beiden Spezialisierungen `GbR-` und `GmbH-Gesellschafter`. Wie beim Einsatz der Vererbung üblich, stellen Generalisierungen i.d.R. aufgeschobene Konzepte dar, von denen sich keine Objekte instanziieren lassen.

Alle spezialisierten Kern- und Rollenkonzepte können polymorph anstelle ihrer Generalisierung verwendet werden. So können natürliche und juristische Personen alle Rollen der Person spielen (Kunde, Bürge, usw.) und eine Person kann eine beliebige Gesellschafterrolle (GbR- oder GmbH-Gesellschafter) wahrnehmen.

Die nachfolgend aufgeführte Eigenschaft von Rollen ist unter dem Gesichtspunkt, daß mit den Produktbereichen unterschiedliche Modellierungskontexte verbunden sind, von besonderer Bedeutung. Denn sie stellt sicher, daß Rollen, die in verschiedenen Produktbereichen modelliert werden, voneinander unabhängig sind.

(15) *Zwischen Rollenkonzepten eines Kernkonzepts dürfen keine Abhängigkeiten existieren, d.h. Rollenkonzepte sind disjunkt zu konstruieren.* So darf beispielsweise ein Rollenobjekt kein anders Rollenobjekt an seinem Kernobjekt voraussetzen. Die Rollenobjekte `Kreditnehmer` und `Anleger` erweitern unabhängig voneinander das Kernobjekt `Kunde`.

Die in diesem Abschnitt erarbeiteten Ergebnisse ermöglichen es nun, Gegenstände in unabhängig voneinander modellierbare Kern- und Rollenkonzepte zu zerlegen und diese in getrennten Schichten bzw. Partitionen einer Schicht zu realisieren. Auf diese Weise lassen sich die Kernkonzepte des Gegenstandsbereichs unabhängig voneinander aus den verschiedenen Blickwinkeln der Produktbereiche in Rollenkonzepten erweitern. Sind neue Produktbereiche bei der Entwicklung zu berücksichtigen, können dem Softwaresystem neue Rollenkonzepte hinzugefügt werden, ohne daß bereits modellierte Kern- und Rollenkonzepte davon betroffen sind. So läßt sich das Subjekt Person aus Abb. 6.13 wie folgt auf die in Abschnitt 5.4.4 definierten logischen Schichten verteilen:

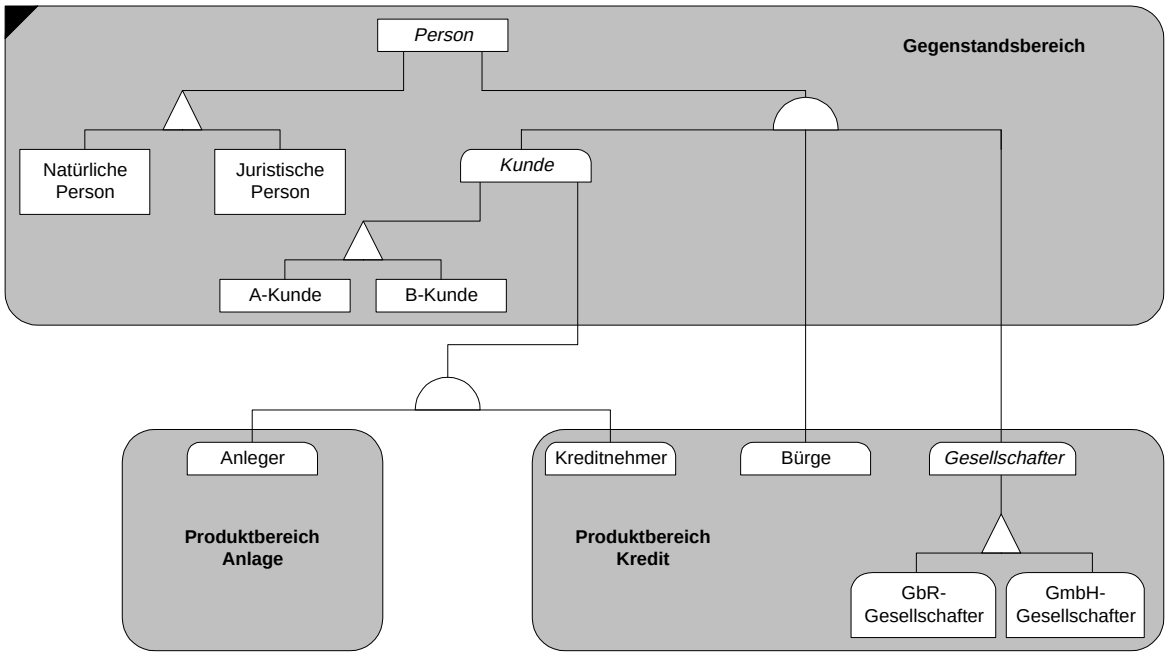


Abb. 6.14: Das Subjekt Person, verteilt auf die Schichten der objektorientierten Schichtenarchitektur

Die vorgestellten Modellierungsmittel Kern-, Rollenkonzept und Subjekt können, da sie keine Annahmen bezüglich ihrer objektorientierten Realisierung machen, bereits bei der Konstruktion des fachlichen Modells angewendet werden. Diese Überlegung soll in einem Ergebnis festgehalten werden:

Ergebnis 6.2

Die Modellelemente Kern-, Rollenkonzept und Subjekt ermöglichen es, die Gegenstände des Anwendungsbereichs bereits im fachlichen Modell aus unterschiedlichen Blickwinkeln der Produktbereiche zu modellieren. Als Notation lassen sich die Symbole aus der Abb. 6.11 verwenden.

6.2.2 Anforderungen an die technische Realisierung von Kern- und Rollenkonzepten

Die bisherige Diskussion hat Kern- und Rollenkonzepte auf einer rein konzeptuellen Ebene thematisiert. Auf der Basis der zuvor aufgeführten Merkmale sollen in diesem Abschnitt Anforderungen für ihre technische Umsetzung im Kontext gängiger objektorientierter Techniken abgeleitet werden. Im Anschluß daran kann für verschiedene aus der Literatur bekannte Verfahren und Ansätze geprüft werden, inwieweit sie sich als Grundlage für die Realisierung eignen.

Kernobjekte müssen sich während ihrer Lebenszeit dynamisch um Rollenobjekte erweitern lassen. Die Forderung nach Dynamik kann in einem objektorientierten Anwendungssystem nur durch eine Benutzt-Beziehung zwischen Kern- und Rollenobjekt realisiert werden, da sich nur Objekte dynamisch zu anderen Objekten hinzufügen und wieder von diesen entfernen lassen. Denn die gängigen objektorientierten Programmiersprachen unterstützen keine dynamische Reklassifikation, beispielsweise durch eine Veränderung der Vererbungsbeziehung zur Laufzeit des Systems[22]. Durch eine Benutzt-Beziehung zwischen Kern- und Rollenobjekten können die Merkmale (1) - (3), und (6) - (8) realisiert werden, da sie auf einer dynamischen Kopplung zwischen Kern- und Rollenobjekt beruhen.

Die Benutzt-Beziehung zwischen Kern- und Rollenobjekten kann weiterhin dazu verwendet werden, die in Merkmal (4) geforderte Subjektidentität zu realisieren. Wenn alle Rollenobjekte, die ein Kernobjekt erweitern, dieses über eine Benutzt-Beziehung kennen, so läßt sich die technische Identität des Kernobjekts als Subjektidentität verwenden. Rollenobjekte besitzen dann dieselbe Subjektidentität, wenn sie dasselbe Kernobjekt referenzieren.

Kern- und Rollenobjekte sind programmiersprachliche Objekte. Demnach werden die Kern- und Rollenkonzepte in Form von Klassen realisiert. Dadurch wird automatisch die Forderung erfüllt, daß mit einem Kern- und Rollenkonzept ein Typ verbunden ist. Die jeweiligen Klassen implementieren dabei die Eigenschaften ihrer Kern- bzw. Rollenobjekte. Die Einführung eines Typs für Kern- und Rollenobjekte gewährleistet den typisierten Zugriff auf die Eigenschaften der Objekte. Erweitert beispielsweise das Rollenkonzept `Kreditnehmer` das Kernkonzept

[22] Prototyp-orientierte Programmiersprachen wie etwa Self (vgl. [CUL89]) erlauben i.d.R. eine dynamische Reklassifikation. Solche Sprache sind aber in der Industrie so gut wie nicht verbreitet.

`Kunde` um die Verwaltung von Sicherheiten, so kann mittels einer Variablen, die vom Typ `Kunde` definiert ist, nicht auf die Sicherheiten des Kreditnehmers zugegriffen werden.

In Punkt (11) der Aufzählung wurde festgehalten, daß Rollenobjekte polymorph anstelle der von ihnen erweiterten Kernobjekte verwendet werden können. Diese Forderung läßt sich in einer typisierten objektorientierten Programmiersprache nur unter Einsatz von Vererbung realisieren. Rollenkonzepte müssen daher von Kernkonzepten erben. Die Vererbungsbeziehung ermöglicht zudem eine einfache Realisierung des Merkmals, Eigenschaften des Kernkonzeptes in einem Rollenkonzept zu erweitern (vgl. Punkt (13)). Zudem schafft sie die Voraussetzung zur Umsetzung der Merkmale (9) - (12), die auf einer *ist-ein* Beziehung zwischen Rollen- und Kernkonzept beruhen. Die Vererbungsbeziehung zwischen Kern- und Rollenkonzept ist wohlgemerkt fachlich motiviert. Sie kann nicht als Implementationsvererbung interpretiert werden. Wenn Rollenhierarchien mittels Vererbung realisiert werden, können diese auch sehr einfach mit Vererbungshierarchien kombiniert werden. Punkt (14) ist somit auch erfüllt.

In der Konsequenz muß demnach zwischen einem Kern- und einem Rollenkonzept technisch sowohl eine Benutzt- als auch eine Vererbungsbeziehung bestehen. Wird das Muster aus Benutzt- und Vererbungsbeziehung nun auch auf Rollenkonzepte angewendet, so lassen sich darüber Rollen von Rollen realisieren (vgl. Punkt (5)).

Die Eigenschaften eines Gegenstands im Anwendungsbereich lassen sich durch das Rollen-Konzept in verschiedenen technischen Klassen modelliert. Dadurch wird die mit dem Gegenstand verbundene Identität auf mehrere technische Identitäten verteilt. Abgesehen davon, daß nun zusätzlich wieder eine fachliche Subjektidentität konstruiert werden muß, kann diese Form der technischen Realisierung zu Problemen bei der Zustandsintegrität des Subjekts führen, sofern nicht geeignete Vorkehrungen getroffen werden. Denn der Zustandsraum eines Gegenstands des Anwendungsbereichs wird auf n Zustandsräume im softwaretechnischen Modell verteilt. Existieren Abhängigkeiten zwischen den einzelnen Zustandsräumen, so werden geeignete Mechanismen für ihre Synchronisation benötigt.

Auch die Möglichkeit, Eigenschaften eines Kernkonzeptes in einem Rollenkonzept zu erweitern, führt potentiell zu Problemen. Dies läßt sich am Beispiel der Kundenadreßänderung illustrieren. Angenommen, die Operation `aendereAdresse` wird nicht nur in der Rolle Kreditnehmer, sondern auch im Anleger erweitert (technisch kann dies durch die Redefinition der Operation `aendereAdresse` sowohl für den Kreditnehmer als auch für den Anleger realisiert werden). Dann führt ein Aufruf der Operation zu drei unterschiedlichen Ergebnissen, jenachdem ob `aendereAdresse` von dem Rollenobjekt Anleger, dem Rollenobjekt Kreditnehmer oder dem Kundenkernobjekt abgearbeitet wird. Dabei gilt:

- erfolgt der Aufruf am Kernobjekt, so wird keine der Erweiterungen ausgeführt.
- erfolgt sie an einem der Rollenobjekte, so wird zwar die dort angegebene Erweiterung (Redefinition) verarbeitet, die des anderen Rollenobjekts bleibt aber unberücksichtigt.

Das Problem besteht darin, daß ein Subjekt technisch in mehreren Klassen realisiert wird und daher nur jeweils eine der drei Operationen zur Ausführung kommt. Aus fachlicher Sicht

sollten sich die verschiedenen Objekte jedoch wie ein Subjekt verhalten. Demnach müßten bei einer Adreßänderung alle drei Operationen ausgeführt werden. Diese Überlegungen sollen in einem Ergebnis festgehalten werden:

Ergebnis 6.3

Bei der Realisierung eines fachlichen Konzepts des Anwendungsbereichs in Form eines Kern- und mehrerer Rollenkonzepte kann es durch nicht ausgeführten Programmkode von erweiterten Operationen des Kernkonzeptes zu Problemen bei der Zustandsintegrität von Kern- und Rollenobjekten kommen.

Technische Realisierungen des Kern- und Rollenkonzepts, die dieses Problem lösen, könnten sich auf das Beobachter Muster abstützen. Der Zweck des Beobachter Musters ist wie folgt beschrieben:

> "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" ([GOF95], S. 293)

Rollenkonzepte können allenfalls Operationen ihres Kernkonzepts erweitern. Das Kernkonzept legt demnach fest, welche Operationen von einem Rollenkonzept erweiterbar sind. Informiert nun ein Rollenobjekt sein Kernobjekt über den Aufruf einer erweiterbaren Operation, so kann dieses alle Rollenobjekte, von denen es momentan erweitert wird, vom Aufruf diesen Operation benachrichtigen. Ruft ein Klient die Operation direkt am Kernobjekt auf, so kann dieses die Rollenobjekte direkt informieren. Rollenobjekte, die über eine eigene Erweiterung der Operation verfügen, können dann geeignet darauf reagieren. Um nicht unnötige Benachrichtigungen an Rollenobjekte zu versenden, könnten sich diese vorher beim Kernobjekt für die Operationen registrieren, die sie erweitert haben und von deren Aufruf sie in Kenntnis gesetzt werden wollen. Übertragen auf das Beobachter Muster übernimmt ein Rollenkonzept also die Aufgabe der Klasse `Beobachter` und das Kernkonzept die der Klasse `Subject`. Rollenobjekte beobachten somit ihr Kernobjekt.

Die Eigenschaft (15), nach der Rollenkonzepte disjunkt zu konstruieren sind, läßt sich technisch nicht erzwingen, sondern muß von einem Softwareentwickler, der einen fachlichen Gegenstand in Form von Kern- und Rollenkonzepten realisiert, bei der Konstruktion berücksichtigt werden.

In den nächsten drei Abschnitten werde ich Ergebnisse aus der Literatur diskutieren, um zu überprüfen, inwieweit sie sich zur technischen Umsetzung des vorgestellten Kern- und Rollenkonzepts eignen.

6.2.3 Subjektorientierte Komposition

Harrison & Ossher führen in [HO93] den Begriff der subjektorientierten Programmierung ein, den sie mit weiteren Autoren in zwei nachfolgenden Artikeln (vgl. [OKH+95] und [OKK+96]) zur Technik der subjektorientierten Komposition ausbauen. Ihr Ziel ist es, auf der Basis von bereits entwickelten Komponenten neue Systeme zu konstruieren. Die einzelnen zu

kombinierenden Klassenhierarchien, Subjekte genannt, modellieren dabei aus einer je spezifischen Sicht einen Bereich des zu konstruierenden Anwendungssystems, der sich mit anderen Bereichen überschneiden kann. Mittels spezieller Werkzeuge kann nun ein Softwareentwickler daraus neue Klassenhierarchien bilden. Er muß dabei insbesondere spezifizieren, wie Klassen, die in den unterschiedlichen Bereichen mehrfach modelliert wurden, in der neuen Komponente zusammenzufügen sind. Die Autoren schreiben zu ihrer Vorgehensweise:

> "The subject-oriented programming paradigm […] supports packaging of object-oriented systems into subjects. Each subject is an object-oriented program or program fragment that models its domain in its own, subjective way. This subjective model is defined by the subject's class hierarchy, which contains just those details implemented and/or used by the subject. A *composition designer* can use a *compositor program* to compose subjects into larger subjects, and eventually entire systems." ([OKK+96], S. 179)

Diese Vorgehensweise soll anhand des eingangs eingeführten Beispiels aus Kunde, Kreditnehmer und Anleger erläutert werden. Nach der vorgestellten Methode würden jeweils zwei Subjekte gebildet, bei dem das erste die Klassen `Kreditnehmer` und `Kreditkonto` und das zweite die Klassen `Anleger` und `Sparbuch` enthält. Dabei verfügen sowohl die Klasse `Kreditnehmer` als auch die Klasse `Anleger` über eine Operation `GibGesamtEngagement` (vgl. Abb. 6.15).

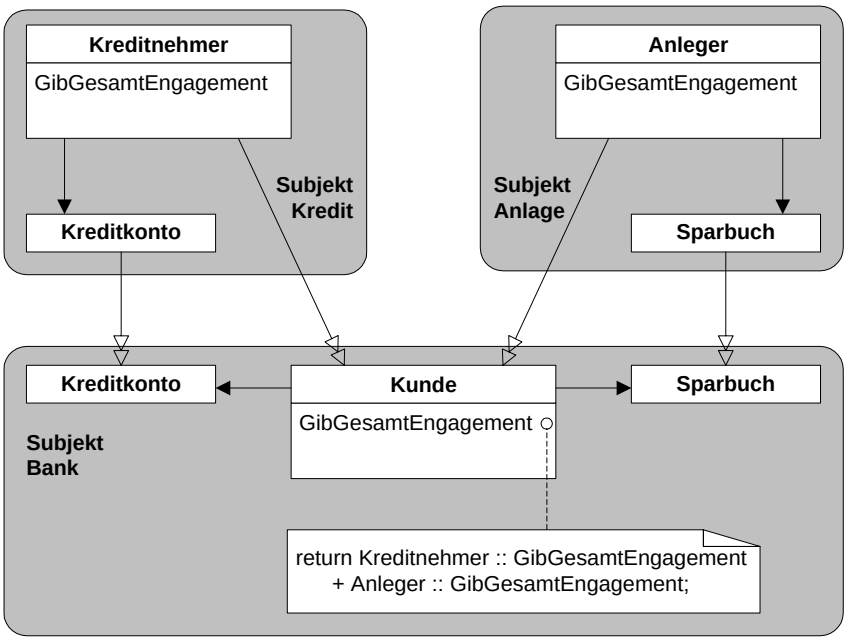


Abb. 6.15: Das kombinierte Subjekt Bank

Ein neues Subjekt `Bank`, das auf der Grundlage der beiden vorgestellten Subjekte gebildet wird, muß die beiden Klassen `Kreditnehmer` und `Anleger` zu einer neuen Klasse `Kunde` zusammenführen, da beide Klassen dasselbe Konzept (einen Kunden) aus unterschiedlichen Blickwinkeln modellieren. Bei der Komposition der neuen Klasse `Kunde` muß ein Softwareentwickler (der *composition designer*) insbesondere angeben, wie fachlich übereinstimmende

Attribute und Operationen aus unterschiedlichen Klassen zusammengeführt werden sollen. So sollte der Aufruf der Operation `GibGesamtEngagement` an der neuen Klasse `Kunde` das Gesamtengagement der beiden Klassen `Kreditnehmer` und `Anleger` aufsummieren. Zudem besitzt die neue Klasse `Kunde` Verweise sowohl auf die Klasse `Kreditkonto` als auch auf die Klasse `Sparbuch`. Die Komposition des neuen Subjekts `Bank` kann nach [OKK+96] auf Quellkodeebene (durch Generierung neuer Quelldateien) oder auf Binärebene erfolgen. Die Binärebene erfordert allerdings die Verwendung spezieller Subjekt-Compiler, die den Binärkode so aufbereiten, daß er sich nachträglich mit anderen Binärdateien verknüpfen läßt. Binärkodes sind allerdings mit dem Vorteil verbunden, weitgehend sprachenunabhängig zu sein.

Die Regeln, nach denen fachlich übereinstimmende Attribute und Operationen in den Klassen neuer Subjekte miteinander kombiniert werden, können sehr unterschiedlich sein. Ossher et al. weisen daher darauf hin, daß eine Entwicklungsumgebung, die das subjektorientierte Paradigma unterstützt, auf jeden Fall eine flexible Erweiterbarkeit der Regeln vorsehen muß. In [OKK+96] beschreiben sie als Beispiel die folgenden beiden Regeln:

- *Mischen*: Wenn die Signatur der zu kombinierenden Attribute bzw. Operationen gleich ist, dann werden bei Anwendung dieser Regel fachlich übereinstimmende Eigenschaften aus den verschiedenen zu kombinierenden Klassen in der kombinierten Klasse durch ein einziges Attribut bzw. eine einzige Operation repräsentiert. Für Operationen muß ein Softwareentwickler zusätzlich die Ausführungslogik spezifizieren. Im oben aufgeführten Beispiel wird die Operation `GibGesamtEngagement` in der Klasse `Kunde` unter Anwendung der Regel *Mischen* konstruiert.
- *Überschreiben*: Bei dieser Regel wird für fachlich übereinstimmende Eigenschaften explizit festgelegt, welches Attribut bzw. welche Operation aus welcher Klasse in die kombinierte Klasse übernommen wird. Eigenschaften einer Klasse können somit Eigenschaften einer anderen Klasse in der kombinierten Klasse überschreiben. Wäre die Operation `GibGesamtEngagement` in der Klasse `Kunde` unter Anwendung der Regel *Überschreiben* entstanden, so hätte sie entweder den Algorithmus aus der Klasse `Kreditnehmer` oder aus der Klasse `Anleger` enthalten.

Die von mir geforderte dynamische Adaptierbarkeit von Kernkonzepten mittels Rollen kann mit der von Ossher et al. vorgeschlagenen subjektorientierten Komposition nicht verwirklicht werden. Da die Kombination von Subjekten zu neuen Subjekten statisch definiert wird, ist die subjektorientierte Komposition in diesem Punkt mit denselben Problemen behaftet wie die Realisierung mittels Mehrfachvererbung. Dagegen sind die Regeln, die angeben, wie fachlich übereinstimmende Operationen und Attribute in die kombinierte Klasse überführt werden sollen, für die weitere Diskussion von Interesse. So muß etwa sichergestellt werden, daß die Operation `GibGesamtEngagement` bei einem Kunden, der gleichzeitig als Kreditnehmer und Anleger agiert, das richtige Resultat liefern (vgl. Ergebnis 6.3). Mein Lösungsvorschlag wird den Ansatz von Ossher et al. daher wieder aufgreifen.

6.2.4 Die OOram Methode

Ossher et al. fokussieren in ihren Arbeiten zur subjektorientierten Komposition auf technische Realisierungen. Reenskaug macht hingegen in [Ree96] deutlich, daß Rollen bereits bei der Modellierung eines Anwendungssystem unverzichtbar sind. Er argumentiert in seinem Buch,

das die von ihm entwickelten OOram Methode (Object Oriented Role Analysis and Modelling) beschreibt, folgendermaßen: die Modellierung eines Systems wird immer von einem ganz bestimmten Standpunkt aus durchgeführt. Daher sind Klassen geprägt durch die Sichtweise, unter der ein Softwareentwickler den Anwendungsbereich betrachtet.

Reenskaug empfiehlt daher, Anwendungssysteme rein unter Verwendung von Rollen zu modellieren, und verzichtet dabei vollkommen auf das Klassenkonzept. Das unter einer spezifischen Sichtweise wahrgenommene Zusammenspiel von Gegenständen im Anwendungsbereich beschreibt er dabei in Form eines Rollenmodells. Die Rollen repräsentieren ähnliche Gegenstände, die durch einen Abstraktionsprozeß auf den Begriff gebracht wurden. Reenskaug schreibt zum Begriff der Rolle:

> "The role is the why abstraction. All objects that serve the same purpose in a structure of collaborating objects, as viewed from the context of some area of concern, are said to play the same role." ([Ree96], S. 14)

Zum Begriff des Rollenmodells führt er aus:

> "The role model is an abstraction on the object model where we recognize a pattern of objects and describe it as a corresponding pattern of roles. The role model supports separation of concern and describes the static and dynamic properties of a phenomenon in a single, coherent model." ([Ree96], S. 43)

Einen so modellierten Ausschnitt des Anwendungsbereichs bezeichnet Reenskaug als „base model“. Da der gleiche Ausschnitt eines Anwendungsbereichs unter verschiedenen Gesichtspunkten modelliert werden kann (z.B. Kredit und Anlage), können für einen Anwendungsbereich verschiedene „base models“ entstehen. Um ein Modell des Gesamtsystems zu erhalten, werden in OOram mittels eines Prozesses, der als Synthese bezeichnet wird, mehrere „base models“ in ein „derived model“ überführt. Reenskaug beschreibt dies folgendermaßen:

> "Several base models may be combined into a composite, or derived, model by the synthesis operation. The phenomenon covered by the derived model is some combination of the phenomena described by the base models, and the derived model is complete in the sense that it presents a whole phenomenon" ([Ree96], S.70)

Bei diesem Prozeß werden, ähnlich zum Vorgehen bei der subjektorientierten Komposition, übereinstimmende Gegenstände, die mittels Rollen in den verschieden „base models“ aus unterschiedlichen Blickwinkel modelliert wurden, im „derived model“ in einer Rolle zusammengefaßt.

Die technische Umsetzung der Modelle sieht folgendermaßen aus (vgl. [Ree96], S. 116): die Rollen im „derived model“ werden entweder als neue Klassen implementiert oder aber erben von den bereits implementierten Klassen des „base models“, sofern bereits Implementierungen vorhanden sind. Dazu muß die verwendete Programmiersprache Mehrfachvererbung unterstützen. Die synthetisierten Modelle sind, wie bei der subjektorientierten Komposition

auch, statisch zu definieren. Die geforderte dynamische Adaptierbarkeit von Kernkonzepten wird demnach ebensowenig unterstützt. Diese Methode ist also zur Umsetzung der in Abschnitt 6.2.2 beschrieben technischen Anforderungen ungeeignet.

6.2.5 Das Extension Object Muster

In [Gam97] stellt Gamma das Extension Object Muster vor. Das Muster ermöglicht es, die bereits existierende Schnittstelle einer Klasse zur Laufzeit dynamisch zu erweitern. Gamma schreibt zum Zweck des Musters:

> "Anticipate extensions to an object's interface. Extension Object lets you add interfaces to a class, and it lets clients choose and access the interface they need." ([Gam97], S. 79)

Ein ähnlicher Zweck ist auch mit Rollenobjekten verbunden. Ein Rollenobjekt ermöglicht es, Kernobjekte mit neuen Schnittstellen (den Rollen) zu versehen. Ein Klient kann dann entscheiden, in welcher Rolle er ein Kernobjekt verwenden möchte. Ein Klassendiagramm, das die Struktur des Extension Object Musters wiedergibt, ist in Abb. 6.16 dargestellt.

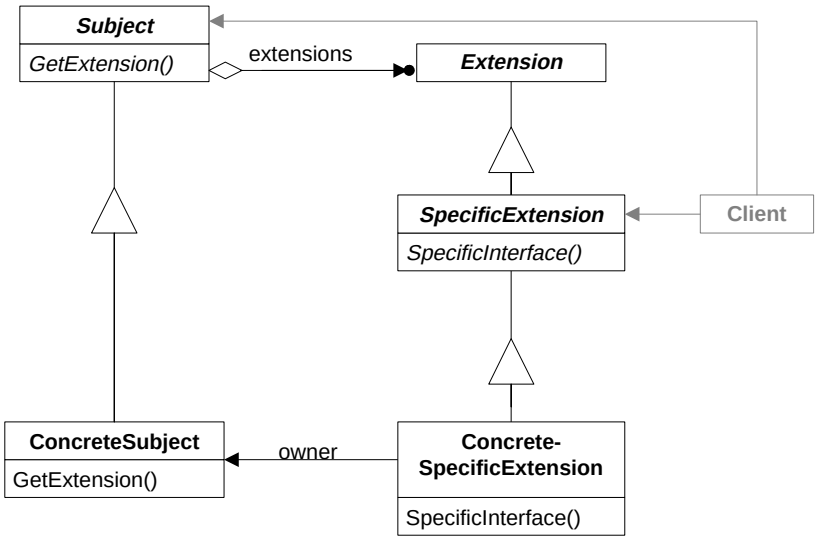


Abb. 6.16: Die Struktur des Extension Object Musters (aus [Gam97], S. 82)

Die Klassen des Entwurfsmusters übernehmen dabei die folgenden Aufgaben: Die Klassen `Subject` und `Extension` beschreiben auf abstrakter Ebene das Zusammenspiel zwischen einem Objekt und dessen möglichen Erweiterungen. Die Klasse `ConcreteSubject` steht für eine konkrete Klasse, die sich um eine konkrete Schnittstelle erweitern läßt. Diese wird durch die Klasse `ConcreteSpecificExtension` symbolisiert. `SpecificExtension` definiert die Schnittstelle, die eine konkrete Erweiterung für einen Klienten erfüllen muß. Dadurch können verschiedene Implementierungen für eine konkrete Erweiterung definiert werden.

Bewertet man dieses Entwurfsmuster vor dem Hintergrund der Anforderungen, die an Kern- und Rollenkonzepte gestellt werden, so kann die Hierarchie der Klasse `Subject` mit den

Kernkonzepten, die Hierarchie der Klasse `Extension` mit den Rollenkonzepten assoziiert werden. Dadurch lassen sich Kernobjekte durch Rollenobjekte dynamisch erweitern. Die Merkmale, die eine *ist-ein* Beziehung zwischen Rollenkonzept und Kernkonzept erforderlich machen, werden durch das Entwurfsmuster nicht abgedeckt, da keine gemeinsame Abstraktion der Klassen `Subject` und `Extension` existiert.

6.3 Das Rollenmuster

Als Lösungsvorschlag für die in Abschnitt 6.2.2 diskutierten Anforderungen möchte ich in diesem Abschnitt das Rollenmuster vorstellen. Es basiert in Teilen auf dem Extension Object Muster von Gamma (vgl. Abschnitt 6.2.5), addressiert aber die Aspekte Typ-Subtyp-Beziehung, Subjektidentität und Zustandsintegrität. Die Beschreibung des Entwurfsmusters beruht in weiten Teilen auf den Texten [BGK+97] und [BRS+97].

Bei der Musterbeschreibung (vgl. [GOF95], S. 6) beschränke ich mich auf die Bereiche Struktur, Teilnehmer, Interaktionen und Implementierung. Konsequenzen werde ich nur beschreiben, wenn sie nicht bereits in den vorherigen Abschnitten diskutiert wurden. Der Motivationsteil greift das bereits bekannte Beispiel aus dem Bankenbereich auf, an dem sich vor allem der Beispielkode orientieren wird. Für eine detaillierte Darstellung des Entwurfsmusters nach der in [GOF95] beschriebenen Form verweise ich auf den Artikel [BRS+97].

Das Entwurfsmusters basiert auf einer Kombination der Muster Extension Object, Dekorierer und Beobachter (vgl. [GOF95]) sowie auf dem in Abschnitt 6.1.1 vorgestellten Produkthändler. Das Dekorierer Muster setzt dabei die geforderte Typ-Subtyp-Beziehung zwischen den Rollenkonzepten und dem Kernkonzept um. Das Beobachter Muster schafft die Basis, um die Zustandsintegrität des Subjekts sicherzustellen. Rollenobjekte, die Operationen des Kernkonzepts erweitern, werden benachrichtigt, sobald die erweiterte Operation an einem anderen Rollenobjekt bzw. an dem Kernobjekt selbst aufgerufen worden ist. Der Produkthändler ermöglicht es, den Erzeugungsprozeß von Rollen vor dem Klienten zu verbergen, so daß er sich in einem Anwendungssystem anhand von Konfigurationsskripten (vgl. Abschnitt 6.1.1) einfach konfektionieren läßt.

Motivation

Als Beispiel für die Beschreibung des Entwurfsmusters wird die folgende Rollenhierarchie verwendet. Die Symbole entsprechen der in Abschnitt 6.2.1 eingeführten Notation. Die Rollenhierarchie beschreibt das Kernkonzept `Person`, das durch die beiden Rollenkonzepte `Kunde` und `GbR-Gesellschafter` erweitert wird.

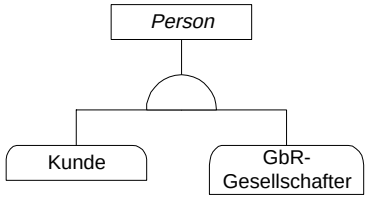


Abb. 6.17: Eine Rollenhierarchie

Struktur

Die folgende Abbildung präsentiert die Struktur des Rollenmusters. Sie zeigt keine Operationen zur Verwaltung der Subjektidentität. Dieser Sachverhalt wird von mir im Implementierungsteil diskutiert. Die in der Abbildung verwendete Spezifikation (`Spec`) ist mit der bereits beim Produkthändler beschriebenen Spezifikation vergleichbar.

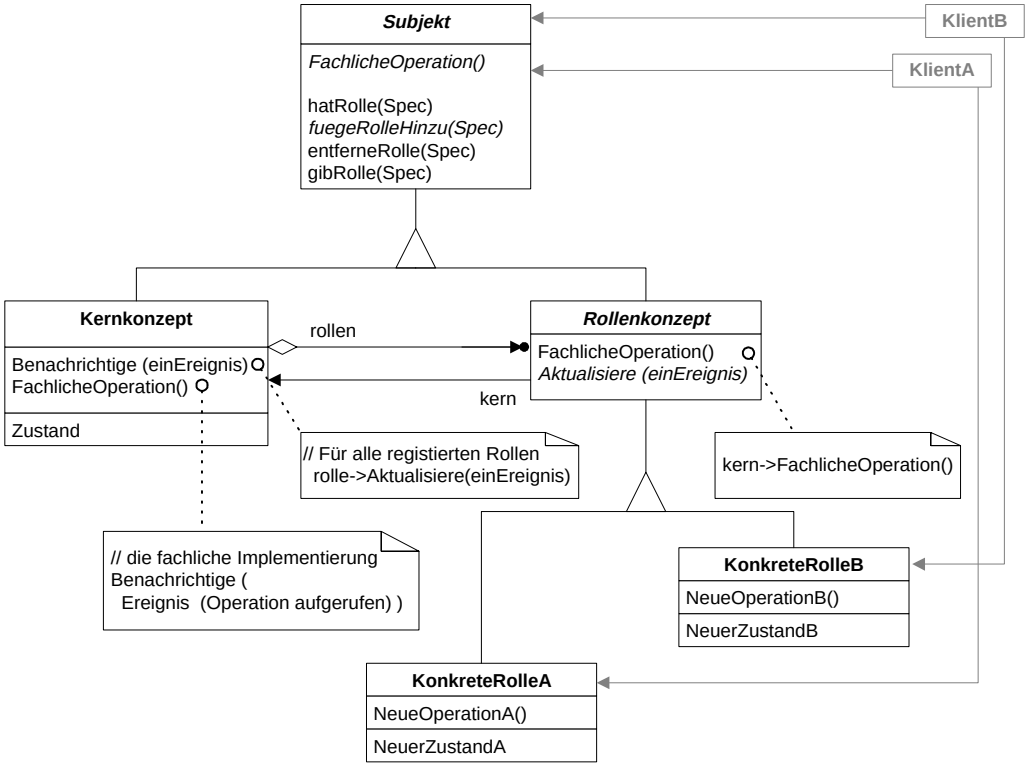


Abb. 6.18: Die Struktur des Rollenmusters

Teilnehmer

- `Subjekt` (das Subjekt Person)
  - modelliert das Subjekt in Form von fachlichen Operationen, im Beispiel der Person etwa `aendereAdresse` oder `gibTelefonnummer`. Die Operationen sind als aufgeschobene Operationen definiert und müssen im Kernkonzept implementiert werden. Da Rollenobjekte polymorph anstelle des Kernobjekts verwendbar sein müssen, implementieren die Rollenkonzepte ebenfalls die Schnittstelle des Subjekts.
  - bietet Operationen für die Rollenverwaltung an (Rollen erfragen, hinzufügen, entfernen und ihr Vorhandsein prüfen). Als Spezifikation wird im einfachsten Fall der Name der Rolle als Zeichenkette oder der Typ des Rollenkonzepts verwendet.
  - definiert keine eigenen Attribute, da es die logische Einheit aus Kernobjekt und Rollenobjekten repräsentiert.
- `Kernkonzept` (das Kernkonzept der Person)
  - verwaltet die zu einem Kernobjekt hinzugefügten Rollenobjekte.
  - implementiert die aufgeschobenen fachlichen Operationen des Subjekts.

- `Rollenkonzept`
  - verwaltet eine Referenz auf das Kernobjekt.
  - bietet eine Standardimplementierung der aufgeschobenen fachlichen Operationen des Subjekts an. Diese delegiert den Operationsaufruf an das Kernobjekt.
- `KonkreteRolle` (Kunde, GbR-Gesellschafter)
  - modelliert und implementiert eine Erweiterung des Kernkonzepts unter einem anwendungsspezifischen Blickwinkel. In der Regel ist das die Sicht, die ein Produktbereich auf ein Kernkonzept des Gegenstandsbereichs einnimmt.

Das Rollenmuster ist auf der Basis von vier bekannten Mustern realisiert. Die nachfolgende Tabelle gibt an, welche Aufgaben die Klassen des Rollenmusters bezogen auf die anderen Entwurfsmuster wahrnehmen (vgl. [Rie97]). Die Tabelle ist dabei wie folgt organisiert: die Entwurfsmuster mit ihren Klassen sind in Zeilen angeordnet, die Klassen aus der Struktur des Rollenmusters in Spalten. Die Tabelle bezieht sich nur auf die aufgeschobenen Klassen. Das Symbol + bedeutet, daß die Klasse der Spalte die Aufgabe der Klasse der Zeile wahrnimmt.

Bekannte Muster \ Das Rollenmuster		*Subjekt*	*Kernkonzept*	*Rollenkonzept*
Extension Object	*Subject*		+	
	Extension			+
	ConcreteSubject		+	
Dekorierer	*Komponente*	+		
	KonkreteKomponente		+	
	Dekorierer			+
Beobachter	*Subjekt*		+	
	Beobachter			+
Produkthändler	*Produkt*			+
	Produkthändler			+
	Klient		+	

Abb. 6.19: Zusammenhang zwischen den Klassen des Rollenmusters und den Klassen der Basismuster

Interaktionen

Die Interaktion zwischen Kern- und Rollenobjekt läßt sich folgendermaßen beschreiben:

- Ein Rollenobjekt leitet die an ihm aufgerufenen Subjektoperationen an das Kernobjekt weiter.
- Alle Rollenobjekte eines Subjekts verwenden dasselbe Wurzelobjekt.

Ein Klient interagiert folgendermaßen mit einem Kernobjekt und dessen Rollenobjekten:

- Ein Klient kann ein Rollenobjekt zu einem Kernobjekt hinzufügen. Dies geschieht, indem der Klient die Rollen in Form eines Spezifikationsobjekts beschreibt.

- Ein Klient erfragt an einem Kern- oder Rollenobjekt eine gewünschte Rolle. Existiert ein entsprechendes Rollenobjekt, so wird dieses zur weiteren Bearbeitung verwendet.
- Verfügt das Kernobjekt nicht über das gewünschte Rollenobjekt, so signalisiert es dies dem Klienten in Form einer Fehlermeldung. Kernobjekte erzeugen nie selbständig Rollenobjekte. Diese müssen immer von einem Klienten zu einem Kernobjekt hinzugefügt werden.

Konsequenzen

Die Vorteile und Konsequenzen, die mit dem Einsatz von Rollen verbunden sind, habe ich bereits ausführlich in Abschnitt 6.2.1 beschrieben. Zu ergänzen sind lediglich zwei Punkte: Erstens verfügen Kern- und Rollenkonzepte über eine hohe Kohäsion, da sie nur Operationen und Attribute enthalten, die aus einem bestimmten Blickwinkel (Gegenstands- oder Produktbereich) motiviert sind. Zweitens führt die Verwendung von Rollen zu einer verbesserten technischen Entkopplung des Systems. Ein Klient, der ein Kernobjekt Person in der Rolle Kunde betrachtet, ist beispielsweise nicht von weiteren Rollen abhängig, die das Kernobjekt spielen kann. Ein Anwendungssystem muß für nicht verwendete Rollenkonzepte daher auch keinen Programmkode enthalten. Demnach führen Änderungen an solchen Rollenkonzepten auch nicht zur unnötigen Neuübersetzung des Systems.

Der Nachteil des Rollenmusters ist sein zusätzlicher Kodierungsaufwand. Klienten müssen i.d.R. zuerst von einem Kernobjekt das passende Rollenobjekt anfordern. Ist dies noch nicht vorhanden, so muß es vom Klienten erst erzeugt werden. Dabei muß der Klient fachlich sicherstellen, daß das Kernobjekt diese Rolle tatsächlich spielen kann. Erst dann läßt sich das Subjekt vom Klienten in der gewünschten Rolle ansprechen.

Implementierung

Die Anforderungen an eine technische Realisierung von Kern- und Rollenkonzepten habe ich bereits in Abschnitt 6.2.2 diskutiert. Nachfolgend werden daher nur ergänzende Punkte dargelegt. Das Klassendiagramm der Abb. 6.20 zeigt bereits die technische Umsetzung einiger zu diskutierender Punkte. Dies sind die Kombination von Vererbungs- und Rollenhierarchien sowie die rekursive Anwendung des Rollenmusters, um Rollen von Rollen zu realisieren.

Die folgenden Implementierungsdetails wurden bisher nicht thematisiert:

- *Verwaltung der Rollenobjekte*. Damit ein Kernobjekt seine Rollenobjekte verwalten kann, muß die Klasse `Subjekt` ein geeignete, abstrakte Schnittstelle zur Verfügung stellen, die von dem konkreten Kernkonzept implementiert wird. Das Kernobjekt führt seine Rollenobjekte in einer Hashtabelle. Kann eine Rolle nur einmal zu einem Kernobjekt hinzugefügt werden und ist sichergestellt, daß mehrere Rollen nicht unter einer gemeinsamen, abstrakten Rolle betrachtet werden, kann als Schlüssel die Rollenspezifikation verwendet werden, die bereits zur Erzeugung eingesetzt wurde. Ansonsten muß für die Rollenerzeugung eine Spezifikation, die das Rollenkonzept (z.B. GmbH-Gesellschafter) beschreibt, und für die Rollenverwaltung eine Spezifikationen, die ein Rollenobjekt (z.B. GmbH-Gesellschafter der Firma „Computer GmbH“) charakterisiert, verwendet werden. Greift ein Klient auf die Rollen eines Kernobjektes unter

Verwendung einer Rollenkonzeptspezifikation zu (z.B. GmbH-Gesellschafter), so erhält er alle Rollenobjekte, die von diesem Rollenkonzept instanziiert sind (z.B. alle GmbH-Gesellschafterrollen). Diese Zugriffsweise läßt sich auch auf abstrakte Rollenkonzepte (z.B. Gesellschafter) ausdehnen, um so alle Gesellschafterrollen eines Kernobjekts zu erfragen. Die Operationen `hatRolle`, `entferneRolle` und `gibRolle` müssen, um beide Möglichkeiten zu realisieren, hierfür sowohl mit einem Parameter vom Typ Rollenobjektspezifikation (um auf eine konkrete Rolle zuzugreifen) als auch mit einem Parameter vom Typ Rollenkonzeptspezifikation (um auf alle Rollenobjekte eines bestimmten Rollenkonzeptes zuzugreifen) angeboten werden.

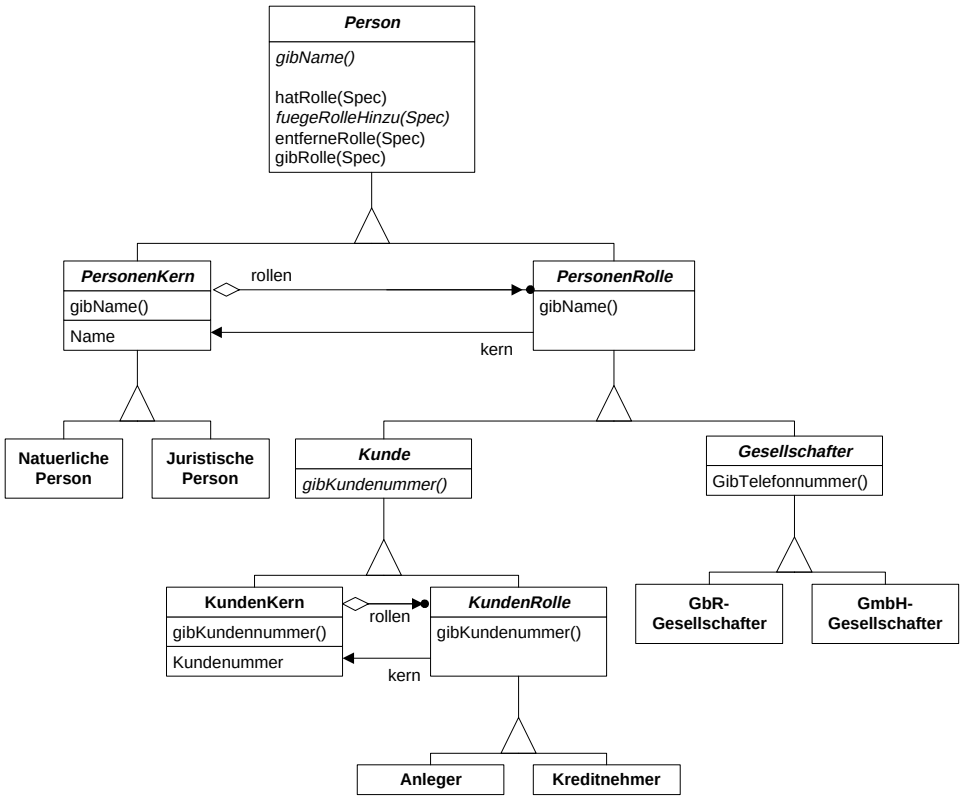


Abb. 6.20: Rekursive Anwendung des Rollenmusters

- *Erweiterung von Operationen.* Das Problem, Rollenobjekte vom Aufruf einer Operation, die sie erweitert haben, zu informieren, wurde bislang auf der Basis des Beobachter Musters gelöst. Eine Alternative stellt die Verwendung des Schablonenmusters dar. So könnte die Klasse `Rollenkonzept` (vgl. Abb. 6.18) für jede fachliche Operation, die von einem Rollenkonzept erweitert werden darf, eine Einschuboperation definieren (z.B. für die Operation `aendereAdresse` die Einschuboperation `_aendereAdresse`). Die Einschuboperation besitzt im Rollenkonzept eine leere Standardimplementierung. Sie wird dann in den konkreten Rollenkonzepten redefiniert, um so die fachlichen Operationen des Kernkonzeptes zu erweitern. Zur Laufzeit ruft nun das Kernkonzept aus der fachlichen Operation an all seinen Rollenobjekten die entsprechende Einschuboperation auf. Die Operation `Aktualisiere` des Beobachter Musters zerfällt somit in einzelne fachliche Einschuboperationen.

  Analog zu der in Abschnitt 6.2.3 präsentierten subjektorientierten Komposition werden bei der Erweiterung einer fachlichen Operation Regeln benötigt, die angeben, wie die

Einschuboperationen der Rollenkonzepte miteinander kombiniert werden sollen. Für die fachliche Operation `aendereAdresse` würden die Erweiterungen typischerweise auf Basis der Regel Mischen miteinander kombiniert. Die entsprechende Regel kann in der fachlichen Operation des Kernkonzepts in Form eines Algorithmus hinterlegt werden. Somit ist jede fachliche Operation in der Lage, ihre individuelle Regel zu implementieren.

- *Rollen von Rollen.* Durch die rekursive Anwendung der Rollenmusters auf Rollenkonzepte, lassen sich Rollen von Rollen realisieren. So ist in der Abb. 6.20 die Rolle `Kunde` des Kernkonzeptes `Person` wieder durch Anwendung des Rollenmusters in ein Kernkonzept `KundenKern` und die Rollenkonzepte `Anleger` und `Kreditnehmer` zerlegt worden.

  Durch die rekursive Anwendung des Musters sind zwei Kernkonzeptklassen entstanden. In einer Implementierung muß daher technisch zwischen einem Wurzelkonzept und einem Kernkonzept unterschieden werden (vgl. Abschnitt 6.2.1). Um die Verdopplung der Exemplarvariable `kern` in einem rekursiven Rollenkonzept zu vermeiden, sollte der Zugriff auf einen Kern nur über eine virtuelle Operation erfolgen. Der eigentliche Verweis auf das Kernobjekt ist dann in den jeweiligen konkreten Rollenkonzepten zu definieren.

- *Verwalten einer Subjektidentität.* Neben der technischen Identität von Kern- und Rollenobjekten existiert eine fachliche Identität, die die Rollenobjekte und ihr Kernobjekt als ein Subjekt identifiziert. Da alle Rollenobjekte ihr Kernobjekt kennen, kann diese Subjektidentität einfach über die technische Identität des Kernobjekts hergestellt werden. Zwei Rollenobjekte sind Teil desselben Subjekts, wenn sie dasselbe Kernobjekt erweitern. Bei einer rekursiven Anwendung des Rollenmusters bezieht sich die Subjektidentität auf das Wurzelobjekt. Zwei Rollenobjekte `Anleger` und `GbR-Gesellschafter` sind subjektidentisch, wenn sie auf dasselbe Wurzelobjekt `NatuerlichePerson` verweisen.

- *Kombination von Vererbungs- und Rollenhierarchien.* Rollenhierarchien können, da sie mittels normaler Vererbung realisiert sind, einfach mit Vererbungshierarchien kombiniert werden. Sowohl Kern- als auch Rollenkonzepte lassen sich mittels Vererbung spezialisieren. So spezialisieren die Klassen `NatuerlichePerson` und `JuristischePerson` das Kernkonzept `Kunde` und die Klassen `GbR-Gesellschafter` und `GmbH-Gesellschafter` das Rollenkonzept `Gesellschafter`.

Das Objektdiagramm der Abb. 6.21 stellt ein Subjekt `Person` dar, das aus dem Kernkonzept `NatuerlichePerson` und den beiden Rollenkonzepten `GmbH-Gesellschafter` und `Kunde` besteht. Die Rolle `Kunde` nimmt zusätzlich die Rolle `Kreditnehmer` war. Die Abbildung entspricht dem bereits in Abb. 6.13 dargestellten Subjekt `Person`.

Beispielkode

Der folgende Programmkode zeigt die Realisierung des Beispiels aus dem Motivationsteil. Die Erweiterung von fachlichen Operationen des Kerns wird dabei durch das Konzept der Einschuboperation realisiert. Der Einfachheit halber unterscheidet die Implementierung nicht zwischen Rollenkonzept- und Rollenobjektspezifikationen. Es lassen sich daher nicht mehrere

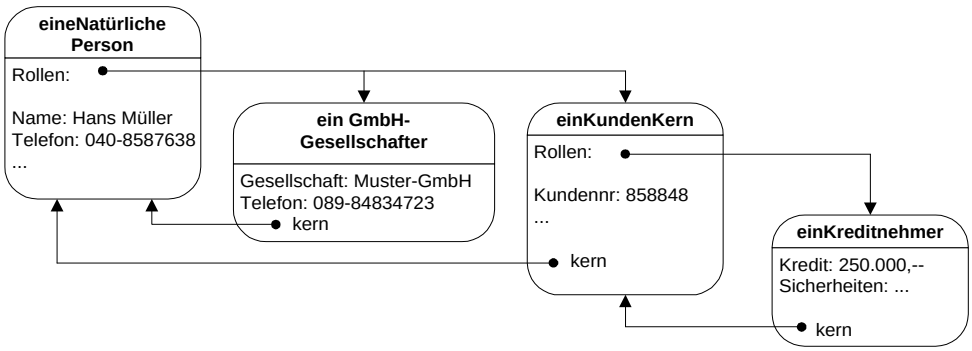


Abb. 6.21: Ein Objektdiagramm mit Rollen von Rollen

Rollen eines Rollenkonzeptes verwalten. Zunächst wird die Implementierung der Klasse `Person` vorgestellt:

```
class PersonenKern;          // Deklaration des Typs PersonenKern;
class PersonenRolle;         // Deklaration des Typs PersonenRolle;
class Person {
public:
            //    fachliche Operationen
      virtual Telefonnummer gibTelfonnummer() = 0;
      virtual void setzeName(const String & einName) = 0;
      ...
            //    Rollenmanagement
      bool istDasselbeSubjektWie(Person * einePerson)
      {
            return (gibWurzelKern() == einePerson->gibWurzelKern());
      }
      PersonenRolle * gibRolle(const Spez & Spez)
      {
            return ((* gibRollen())[eineSpez]);
      }
      virtual void fuegeRolleHinzu(const Spez & eineSpez) = 0;
      ...
protected:
      virtual Person * gibKern() = 0;
      virtual map<Spez, PersonenRolle *> * gibRollen() = 0;
private:
      Person * gibWurzelKern()
      {
            Person * p;
            if ((p = gibKern()) == NULL return this;
                  else return p->gibWurzelKern();
      }
};
```

Diese Implementierung bildet die Oberklasse für die Klassen `PersonenKern` und `PersonenRolle`. Die Klasse `PersonenKern` ist dabei wie folgt realisiert:

```
class PersonenKern : public Person {
public:
      Telefonnummer gibTelefonnummer() { return meineNummer; };
      void setzeName(const String & einName)
      {
            meinName = einName;
            // ermoegliche die Erweiterbarkeit der Operation
            // Fuer alle Rollen, die _setzeName erweitert haben
            eineRolle->_setzeName(einName);
      };
      ...
      void fuegeRolleHinzu(const Spez & eineSpez)
      {
            *Rollen[eineSpez] =
             PersonenRolle :: vermittleFuer(eineSpez, this);
      }
protected:
      Person * gibKern() { return NULL; };
      map<Spez, PersonenRolle *> * gibRollen(){ return Rollen; };
```

```
private:
      Telefonnummer meineNummer;
      String meinName;
      map<Spez, PersonenRolle *> * Rollen;
};
```

Als nächstes wird die Klasse `PersonenRolle` als Oberklasse aller Rollenkonzepte realisiert. Sie leitet alle aufgeschobenen Operationen an das von einem Rollenobjekt umwickelte Kernobjekt weiter. Die Klasse definiert zudem für die fachlichen Operationen, die ein Rollenkonzept erweitern kann, eine leere Einschuboperation.

```
class PersonenRolle : public Person {
public:
      friend class PersonenKern;
      Telefonnummer gibTelefonnummer()
      {
            return gibPersonenKern()->gibTelefonnummer();
      };
      void setzeName(const String & einName)
      {
            gibPersonenKern()->setzeName(einName);
      };
      void fuegeRolleHinzu(const Spez & eineSpez)
      {
            gibPersonenKern()->fuegeRolleHinzu(eineSpez);
      }
      ...
protected:
      virtual void _setzeName(const String & einName) {};
            // Händler Operationen
      static PersonenRolle * vermittleFuer(const Spez &, PersonenKern *);
      virtual PersonenKern * gibPersonenKern() = 0;
      map<Spez, PersonenRolle *> * gibRollen()
      {
            return gibPersonenKern()->gibRollen();
      };
};
```

Die Unterklassen der Klasse `PersonenRolle` definieren konkrete Rollenkonzepte. Die nachfolgende Implementierung zeigt die konkrete Rolle `Kunde`:

```
class Kunde : public PersonenRolle {
public:
      virtual void _setzeName(const String & einName)
      {
            // erweiterte Implementierung
            ...
      };
      Kundennummer gibKundennummer( return meineKundennummer; );
protected:
      Kunde(PersonenKern * einKern) { kern = einKern; };
      Person * gibKern() { return kern; };
      PersonenKern * gibPersonenKern() { return kern; };
private:
      PersonenKern * kern;
      Kundennummer meineKundennummer;
};
```

Ein Klient würde wie folgt mit einer `Person` und einer Rolle `Kunde` umgehen:

```
Person * einePerson = Datenbank :: ladePerson("Hans Müller");
Kunde * einKunde =
      dynamic_cast<Kunde *>einePerson->gibRolle(typeid(Kunde));
if (einKunde) {
      einKunde->setzeName("Hans Müller-Ganzenbein"); // hat geheiratet
      cout << einKunde->gibName() << " " << einKunde->gibTelefonnummer();
};
```

Dies führt zu der Ausgabe: Hans Müller-Ganzenbein 040-8587638.

6.4 Konfektionierung von Anwendungssystemen

Die in den vorherigen Kapiteln erarbeiteten Ergebnisse bilden die Grundlage, Anwendungssysteme flexibel aus einer Menge von Rahmenwerken zu konfektionieren. Da Rahmenwerke die Wiederverwendung von Analysen, Entwurfsentscheidungen, Architekturen und Implementierungen ermöglichen, sollen Anwendungssysteme auf ihrer Basis erstellt werden. Der objektorientierte Schichtenarchitekturstil (vgl. Abschnitt 5.4.4) setzt die produktorientierte Makrostruktur des fachlichen Modells in das softwaretechnische Modell um. Der Technologiebereich stellt dabei die technische Integierbarkeit und Kombinierbarkeit der einzelnen Rahmenwerke sicher, der Gegenstandsbereich die fachliche. Die Zerlegung eines Rahmenwerks oder einer Klassenbibliothek in einen Konzeptteil und mehrere Realisierungsteile sowie die Verwendung spezieller Entwurfsmuster unterstützen zudem die Realisierung des Offen-Geschlossen-Prinzips. All diese Ergebnisse schaffen allerdings lediglich die Voraussetzung, Anwendungssysteme auf der Basis von Klassenbibliotheken und Rahmenwerken zu konfektionieren. Offengeblieben ist bisher, wie die Konfiguration eines solchen Systems zu beschreiben ist.

Anhand von Entwurfsmustern werden in dem vorgestellten rahmenwerkbasierten Architekturstil komplexe Verbindungsstücke beschrieben. Dabei stehen überwiegend statische Aspekte im Vordergrund. So ist die Notation darauf ausgerichtet zu beschreiben, welche Klasse eines Rahmenwerks welche Verantwortlichkeit des Entwurfsmusters übernimmt (vgl. Abb. 5.6). Dynamische Aspekte sind bisher unberücksichtigt geblieben. So beschreibt der rahmenwerkbasierte Architekturstil nicht, welche Objekte zur Laufzeit eines System zusammenarbeiten. Werden die beiden Rahmenwerke für die fachlichen Werte und die Interaktionstypen mittels der von mir in Abschnitt 5.3 beschriebenen Notation dargestellt, so ergibt sich folgendes Bild:

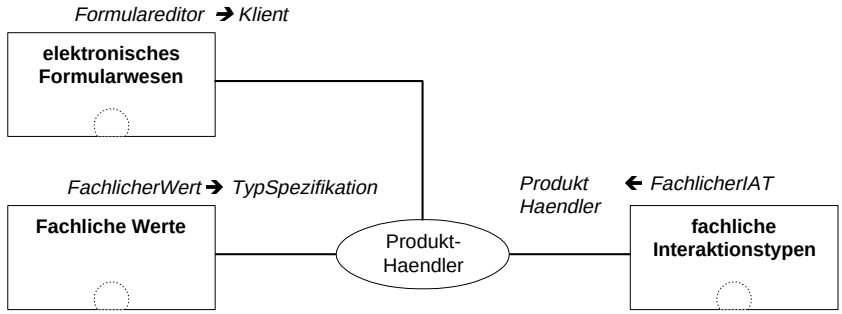


Abb. 6.22: Das Entwurfsmuster Produkthändler als Verbindungsstück

Die Zuordnung von konkreten fachlichen Interaktionstypen zu konkreten fachlichen Werten wird mittels der vorgestellten Konfigurationsskripten festgelegt. Diese beschreiben somit den Teil des Anwendungssystems, der sich an veränderliche Randbedingungen anpassen läßt, ohne daß dazu ein Rahmenwerk geöffnet werden muß. Die Konfigurationsskripten lassen sich somit zur Konfektionierung von Anwendungssystemen einsetzen. Nachfolgend ist daher zu untersuchen, an welchen Stellen in einem Anwendungssystem Händler etabliert werden müssen und welche Konfigurationsformen sich durch Skripten unterstützen lassen.

In Kapitel 4 habe ich dargelegt, daß viele Methoden die Unterteilung eines Anwendungssystems in die Bereiche Benutzungsschnittstelle, fachliche Funktionalität, Datenobjekte und Infrastruktur empfehlen (vgl. [CD94] und [CY91a]). Diese Unterteilung wurde als Grundlage für eine Schichtenarchitektur verworfen, da sich darüber insbesondere keine Modellierungskontexte etablieren lassen. Das damit verbundene Grundprinzip der Trennung von Zuständigkeiten ist von mir jedoch nicht kritisiert worden. Im Gegenteil, es bildet die Basis für viele heute eingesetzte Rahmenwerke oder Anwendungssysteme. Auch die WAM-Methode empfiehlt die Trennung von Zuständigkeiten in die Bereiche Werkzeug, Automat und Material (vgl. Abschnitt 2.3 und [Gry96]).

Um Zuständigkeitsbereiche bestmöglich voneinander zu entkoppeln, werden fachlich zusammengehörige Einheiten über das Entwurfsmuster des Produkthändlers miteinander gekoppelt. In Anwendungssystemen kann so die Zuordnung zwischen den verschiedenen Einheiten mittels Konfigurationsskripten beschrieben und dadurch der Aufbau und die Funktionsweise des Systems an die gegebenen Randbedingungen angepaßt werden. Dies soll an einem Beispiel erläutert werden, das aus den Komponenten Dantenbank, fachlichen Materialien und korrespondierenden Werkzeugen besteht. Die Komponenten werden nachfolgend beschrieben:

Datenbanksystem:

Das Anwendungssystem ist an ein relationales Datenbanksystem angeschlossen, das über einen Automaten (`DBAutomat`) gekapselt wird. Dieser Automat setzt bei den zu speichernden Materialien das bereits angesprochene `Serialisierbar` Protokoll voraus (vgl. Abschnitt 6.1.1). Zudem verwendet der Automat zwei aufgeschobene Klassen `Reader` und `Writer`, die das abstrakte Protokoll zum Laden und Speichern von Materialien festlegen. Das nachfolgende Klassendiagramm soll dies verdeutlichen (vgl. [RSB+97]):

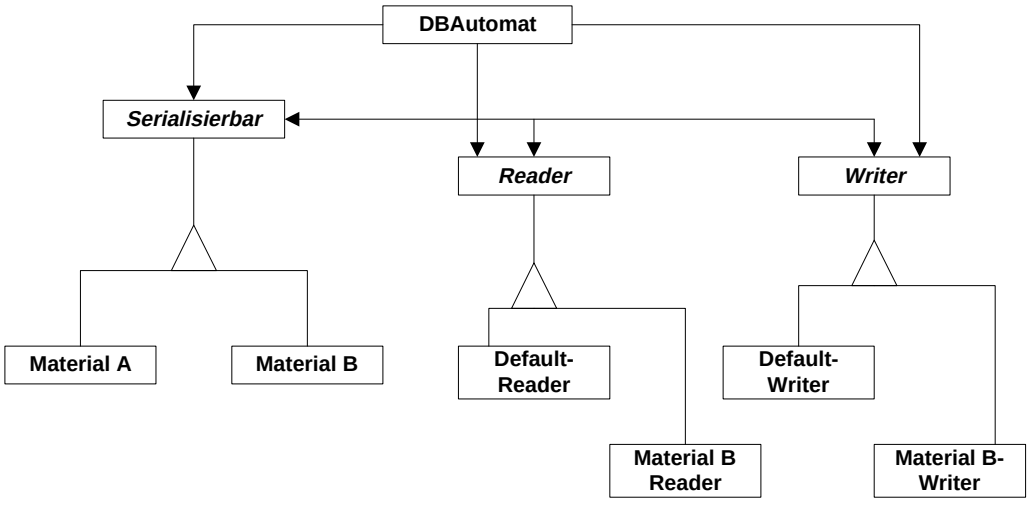


Abb. 6.23: Der Anschluß einer Datenbank unter Verwendung des Serialisierer Musters

Um das Zusammenspiel zwischen den Komponenten flexibel festlegen zu können, werden verschiedene Produkthändler eingerichtet. Die Klasse `Serialisierbar` fungiert als Händler, wie es bereits in dem Abschnitt *Beispielkode* des Entwurfsmusters beschrieben ist. Zusätzlich übernehmen folgende Klassen die Verantwortlichkeit eines Händlers:

- `Writer`: Als Spezifikation wird der Typ des zu speichernden Materials verwendet, anhand dessen ein konkreter `Writer` ausgewählt wird. Im Beispiel ist dies für das `MaterialA` der `DefaultWriter` und für das `MaterialB` ein spezieller `MaterialB-Writer`.
- `Reader`: Als Spezifikation wird hier die Klassenidenifikation verwendet, die auch beim Händler `Serialisierbar` eingesetzt wird. Die Zuordnung eines `Reader` zu einem konkreten Material erfolgt analog zum `Writer`. Dadurch kann sich der `DBAutomat`, nachdem er die Klassenidentifikation von der Datenbank gelesen hat, einen passenden `Reader` vermitteln lassen, der dann das eigentliche Objekt aus der Datenbank lädt.

Die drei etablierten Händler sorgen für eine lose Kopplung von konkreten Materialien zu ihrem `Reader` und `Writer`.

Materialien und Werkzeuge:

Neben dem Datenbankanschluß setzt sich das Anwendungssystem fachlich aus einer Menge von Materialien und dazu passenden Werkzeuge zusammen, die nachfolgend aufgezählt werden. In Klammern gebe ich dabei an, aus welcher Schicht die Elemente stammen: ein Depot (Produktbereich Wertpapier), eine Riskoerklärung (Produktbereich Wertpapier), ein Schreibtisch (Technologiebereich), ein Werkzeug zum Bearbeiten von Sicherheiten (Produktbereich Kredit), ein Formulareditor (Gegenstandsbereich) und ein Werkzeug, um den Schreibtisch zu präsentieren (Technologiebereich).

Nachfolgend wird nun in Form der Zuordnungen, die ein Händler führt, auszugsweise die Konfiguration des oben beschriebenen Anwendungssystems dargelegt. Die Beschreibung hat dabei folgenden Aufbau:

- Für jeden Händler wird angegeben, welche Zuordnungen er führt. In dem nachfolgenden Beispiel sind dies Händler für die abstrakten Klassen `Writer`, `Reader`, `Serialisierbar` und `Werkzeug`. Mit den Händlern ist dabei die folgende Dienstleistung verbunden:
  - `Writer`: speichert ein Material in der Datenbank
  - `Reader`: lädt ein Material aus der Datenbank
  - `Serialisierbar`: erzeugt ein Objekt im Hauptspeicher, das dann die aus der Datenbank geladenen Attribute aufnimmt.
  - `Werkzeug`: präsentiert ein Material und ermöglicht es dem Benutzer, dieses interaktiv zu manipulieren.
- Eine einzelne Zuordnung wird in der Form Spezifikation → Produkt notiert (vgl. Abschnitt 6.1.1). Die Spezifikation beschreibt dabei die zu erbringende Dienstleistung, das Produkt den Typ des konkreten Produkts, das die Dienstleistung erbringen kann. So wird bei dem Händler für die abstrakte Klasse `Writer` unter dem Standardfall (Speichern eines Materials) das konkrete Produkt `DefaultWriter` eingetragen und unter der Dienstleistung „Speichern eines Materials vom Typ `Depot`“ (`TypSpezifikation(Depot)`) das konkrete Produkt `DepotWriter`.

Diese Beschreibung läßt sich leicht in die beim Entwurfsmuster Produkthändler vorgestellten C++ Anweisungen umsetzen.

Händler: `Writer`
```
   // Dienstleistung ist das Speichern eines Materials
   // in die Datenbank
   Standardfall                    → DefaultWriter;
   TypSpezifikation(Depot)         → DepotWriter;
```
Händler: `Reader`
```
   // Dienstleistung ist das Laden eines Materials aus
   // der Datenbank
   Standardfall                    → DefaultReader;
   TypSpezifikation(Depot)         → DepotReader;
```
Händler: `Serialisierbar`
```
   // Dienstleistung ist das Erzeugen eines Objekts im Haupt-
   // speicher. Dieses Objekt nimmt die aus der Datenbank
   // geladenen Attribute auf.
   IDSpezifikation(Depot)          → Depot;
   IDSpezifikation(Risikoerklaerung) → Risikoerklaerung;
   IDSpezifikation(Schreibtisch)   → Schreibtisch;
```
Händler: `Werkzeug`
```
   // Dienstleisung ist die Präsentation und die interaktive
   // Bearbeitung eines Materials.
   TypSpezifikation(Schreibtisch)  → Schreibtischwerkzeug
   TypSpezifikation(Depot)         → Sicherheitenwerkzeug
   TypSpezifikation(Risikoerklärung) → Formulareditor
```

Durch die Konfiguration des Händlers `Serialisierbar` wird festgelegt, welche Materialien zu dem System gehören. Materialien, die von diesen Wurzelmaterialien referenziert werden, bindet der Linker automatisch zum Anwendungssystem hinzu. Damit sich die referenzierten Materialien ebenfalls beim Händler `Serialisierbar` anmelden, verfügen alle Materialien über ein entsprechendes Registrierungsobjekt (vgl. Abschnitt 6.1). Dieses Registrierungsverfahren wird auch für die restlichen Händler angewendet. Die Händler `Reader` und `Writer` besitzen eine Standardzuordnung für Materialien. Lediglich das Material `Depot` wird mit speziellen Algorithmen geladen und gespeichert. Die Konfiguration des Händlers `Werkzeug` ordnet jedem Material ein passendes Werkzeug zu.

Damit die Konfiguration wirksam wird, werden die Skripten vom Hauptprogramm bei der Systeminitialisierung abgearbeitet. Mittels der Skripten läßt sich nicht konfigurieren, welche Werkzeuge und Materialien dem Benutzer beim Einstieg in das Anwendungssystem präsentiert werden. Dies ist weiterhin im Hauptprogramm zu kodieren. Da der Schreibtisch Ausgangspunkt für die Arbeit des Benutzers ist, wird er zunächst aus der Datenbank geladen. Mittels des `Werkzeug` Händlers läßt sich dann ein passendes Werkzeug zur Präsentation und Bearbeitung des Schreibtisches erzeugen. Alle weitere Funktionalität stellen die Rahmenwerke der einzelnen Schichten zur Verfügung.

Die Vorteile, die mit der Verwendung von Händlern zur Konfiguration eines Systems verbunden sind, habe ich bereits im Teil *Konsequenzen* des Abschnitt 6.1 diskutiert. Sie sollen hier

um Vorteile, die sich auf die Konfektionierung und Entwicklung von Anwendungssystemen beziehen, ergänzt werden:

- Die Konfiguration eines Systems läßt sich sehr einfach anpassen. So bedingt etwa die Zuordnung eines anderen Werkzeugs für das Material `Depot` lediglich die Änderung eines Eintrags im Konfigurationsskript. Es muß dazu kein Rahmenwerk neu übersetzt oder gar geöffnet werden.
- Der Test neuer Komponenten wird stark vereinfacht, da sie sich leicht in eine bestehende Konfiguration einbinden lassen. Zudem kann eine neue Komponente anfangs nur einigen wenigen Elementen zu Versuchszwecken zugeordnet werden. Der Anschluß eines neuen `Readers` und `Writers` zum Test eines neuen Verfahrens, um Objekte zu speichern und zu laden, läßt sich erst mit einigen ausgesuchten Materialien erproben.
- Häufig verwendete Konfigurationen lassen sich, da sie aus C++ Anweisungen bestehen, übersetzen und in Form von dynamischen Bibliotheken zur Verfügung stellen. Sie können so einfach mit Anwendungssystemen kombiniert werden.
- Bereits in Abschnitt 5.5 habe ich herausgearbeitet, daß unter Entwicklungsgesichtspunkten eine Bibliotheksstruktur, die aus vielen kleinen Bibliotheken besteht, von Vorteil ist. Das `Erzeuger` Konzept (vgl. Abschnitt 6.1.1) läßt sich nun so modifizieren, daß der `Erzeuger` nicht direkt durch Aufruf der Operation `new` das gewünschte Objekte instanziiert, sondern zuerst die dynamische Bibliothek in den Speicher lädt und dann mittels einer Erzeugungsfunktion, die aus der Bibliothek an einen Funktionszeiger gebunden wurde, das Objekt erzeugt (vgl. [Sta91]). Änderungen die sich während der Entwicklung des Systems ergeben, lassen sich durch den Einsatz derartiger `Erzeuger` in das Anwendungssystem einfügen, ohne daß dieses gestoppt und neu gestartet werden muß. Es muß lediglich die Bibliothek aus dem Speicher entfernt, die Änderung durchgeführt und dann die Bibliothek durch Erzeugen eines Objektes wieder in den Speicher geladen werden. Um die Bibliothek allerdings entfernen zu können, müssen alle Objekte, deren Programmkode in der Bibliothek enthalten ist, aus dem Speicher entfernt sein. Dieses Verfahren bietet sich daher vorwiegend für Werkzeuge an, die ein Benutzer jederzeit interaktiv schließen und damit löschen kann. Bei großen Anwendungssystemen hat diese Vorgehensweise zudem den Vorteil, daß nur die benötigten Ausführungseinheiten in den Speicher geladen werden. So wird beispielsweise die Bibliothek, in der sich ein Werkzeug befindet, erst dann geladen, wenn von dem Werkzeug tatsächlich ein Objekt erzeugt wird.

Die Konfiguration eines Anwendungssystems muß nach Shaw & Garlan mittels einer Architekturbeschreibungssprache charakterisierbar sein (vgl. Abschnitt 5.1.2). Sie schreiben zu den Eigenschaften, die dabei von der Architekturbeschreibungssprache zu erfüllen sind:

> "Properties of configuration permit us to understand and change the architectural structure of a system without having to examine each of the system’s individual components. A consequence is that a language for architectural description should separate the description of composite structures from the elements in those compositions, so that we can reason about the composition as a whole. Dynamic reconfiguration is needed to allow architectures to evolve during the execution of a system." ([SG96], S. 154)

Die von mir vorgestellten Konfigurationsskripten ermöglichen es, den Aufbau eines Anwendungssystems getrennt von den zu strukturierenden Komponenten und Verbindungsstücken zu beschreiben. So kann die Konfiguration eines Systems analysiert werden, ohne dabei die einzelnen Komponenten und Verbindungsstücke zu untersuchen. Zudem wird die von Shaw & Garlan geforderte dynamische Rekonfiguration unterstützt.

Kombiniert man also den von mir vorgestellten rahmenwerkbasierten Architekturstil (vgl. Abschnitt 5.3) mit dem Entwurfsmuster Produkthändler, so vervollständigt dies den Architekturstil im Bereich der Konfigurationsbeschreibung. Die vorgestellten Konfigurationsskripten sind somit in die Architekturbeschreibungssprache des rahmenwerkbasierten Architekturstils aufzunehmen. Dies soll in einem Ergebnis festgehalten werden:

Ergebnis 6.4

Die Konfigurationsskripten des Entwurfsmusters Produkthändler ermöglichen es, den Aufbau eines Anwendungssystems getrennt von den Komponenten und Verbindungsstücken des rahmenwerkbasierten Architekturstils zu beschreiben. Sie erlauben zudem die dynamische Rekonfiguration eines Systems. Das Konzept der Konfigurationsskripten wird daher in die Architekturbeschreibungssprache des rahmenwerkbasierten Architekturstils aufgenommen, um diesen im Bereich der Konfigurationsbeschreibung zu vervollständigen.

6.5 Zusammenfassung

Ich habe in diesem Kapitel die beiden Entwurfsmuster Produkthändler und Rollenmuster sowie Konfigurationsskripten zur Konfektionierung von Anwendungssystemen vorgestellt. Da Konfigurationsskripten auf dem Konzept des Händlers beruhen, werden sie in der nachfolgenden Zusammenfassung gemeinsam mit diesem beschrieben. Den Abschluß dieser Zusammenfassung bildet das Rollenmuster.

Ein Produkthändler vermittelt einem Klienten eine Dienstleistung, die von einem konkreten Produkt erbracht wird. Die gewünschte Dienstleistung beschreibt der Klient dabei in Form einer Spezifikation. Ein Produkthändler übernimmt somit für einen Klienten die Auswahl eines konkreten Produkts sowie dessen Instanziierung. Durch den Einsatz eines Händlers können Klienten folglich vollständig von der Objekterzeugung entkoppelt werden und, sofern der Klient auf das konkrete Produkt nur über die Schnittstelle der aufgeschobenen Produktklasse zugreift, auch von der Existenz konkreter Produktklassen. Unter diesen Voraussetzungen kann ein Klient in einer tieferliegenden Schicht mit Hilfe eines Händlers ein konkretes Produkt, das in einer höherliegenden Schicht enthalten ist, erzeugen, ohne daß dadurch Abhängigkeiten zwischen Klassen der tieferliegenden Schicht zu Klassen der höherliegenden Schicht entstehen.

Seine Konfiguration verwaltet ein Händler in Form von Zuordnungen. Die Spezifikation ist dabei der Schlüssel, das Vermittlungsobjekt bzw. das Erzeugungsobjekt der Wert. In der Regel führt ein Händler diese Zuordnungen in einer Hashtabelle. Dies schafft die Voraussetzung, daß die Konfiguration eines Händlers dynamisch anpaßbar ist. Das Verhalten eines

Anwendungssystems kann somit zur Laufzeit verändert werden. Registrierungsobjekte und Skripten legen die Startkonfiguration eines Händlers fest. Beide tragen bei einem Händler unter einer Spezifikation ein entsprechendes Vermittlungsobjekt ein. Registrierungsobjekte werden in C++ typischerweise als Klassenvariablen realisiert. Sie sind demnach Bestandteil des Rahmenwerks, das die entsprechende Klasse enthält. Änderungen an der Konfiguration führen somit zur Öffnung des Rahmenwerks. Registrierungsobjekte sollten daher nur zur Beschreibung von relativ stabilen Standardkonfigurationen eingesetzt werden.

Im Gegensatz dazu sind Konfigurationsskripten Bestandteil des Anwendungssystems. Sie ermöglichen es folglich, die Konfiguration eines Anwendungssystems zu ändern, ohne dabei ein Rahmenwerk zu öffnen. Mit ihrer Hilfe lassen sich Anwendungssysteme auf der Basis bereits bestehender Rahmenwerke konfektionieren. Konfigurationsskripten verfügen dabei über die folgenden Vorteile: die Konfiguration eines Systems läßt sich sehr einfach anpassen, der Test neuer Komponenten wird erleichtert, da sie sich leicht in eine bestehende Konfiguration einbinden lassen, und häufig verwendete Konfigurationen lassen sich als dynamische Bibliothek zur Verfügung stellen. Weiterhin kann die Konfiguration eines Anwendungssystems mit Hilfe von Skripten getrennt von den zu strukturierenden Komponenten und Verbindungsstücken beschrieben werden. Da Skripten die dynamische Rekonfiguration eines Systems ermöglichen, konnte ich das Konfigurationskonzept in die Architekturbeschreibungssprache des rahmenwerkbasierten Architekturstils integrieren, um diesen im Bereich der Konfigurationsbeschreibung zu vervollständigen.

Die beiden Mechanismen Konfigurationsskript und Registrierungsobjekt können miteinander kombiniert werden. Auf diese Weise läßt sich die Standardkonfiguration eines Anwendungssystems durch Registrierungsobjekte festlegen, während Erweiterungen unter Verwendung von Konfigurationsskripten definiert werden. Dabei muß allerdings sichergestellt werden, daß alle Registrierungsobjekte abgearbeitet sind, bevor Skripten Änderungen an der Standardkonfiguration des Anwendungssystems vornehmen.

Das Konzept der Rolle wurde motiviert vor dem Hintergrund, daß Konzepte des Gegenstandsbereichs aus Sicht der verschiedenen Produktbereiche zumeist unterschiedliche Umgangsformen und Attribute aufweisen. Aufbauend auf der Literatur habe ich daher die Begriffe Rollen- und Kernobjekt eingeführt. Rollenobjekte können ein Kernobjekt zur Laufzeit eines Systems dynamisch erweitern. Mit ihrer Hilfe lassen sich die produktbereichsspezifischen Attribute und Umgangsformen modellieren, ohne daß dabei das Konzept im Gegenstandsbereich erweitert werden muß.

Rollenobjekte und Kernobjekt bilden zusammen eine fachliche Einheit. Es wurde daher der Begriff des Subjekts eingeführt. Ein Subjekt umfaßt das Wurzelobjekt und alle seine Rollenobjekte und verkörpert eine eigene fachliche Identität, die als Subjektidentität bezeichnet wird. Aufbauend auf diesen Begriffen, den Anforderungen an eine technische Realisierung und der Evaluierung bekannter Implementierungen habe ich dann in Abschnitt 6.3 das Rollenmuster vorgestellt. Dieses beschreibt die technische Realisierung der Konzepte Kernobjekt, Rollenobjekt und Subjekt in einer typisierten objektorientierten Programmiersprache.

7 Zusammenfassung und Ausblick

Die Entwicklung großer Anwendungssysteme sowohl im industriellen als auch im wissenschaftlichen Bereich stellt stets eine enorme Herausforderung dar. Diese Arbeit leistet einen Beitrag, diese Herausforderung im konstruktiven Bereich besser bewältigen zu können. Dazu wurden Ergebnisse aus den Gebieten Modellbildung im Softwareentwicklungsprozeß, Semiotik, Organisationstheorie, Softwarearchitekturen und objektorientierte Anwendungsentwicklung zueinander in Beziehung gesetzt. Durch die inhaltliche Verknüpfung dieser Bereiche weist der vorgestellte Ansatz insbesondere die folgenden Vorzüge auf:

- Der *objektorientierte Schichtenarchitekturstil organisiert*, im Gegensatz zu herkömmlichen Architekturstilen (vgl. [SG96]), sowohl das *fachliche* als auch das *softwaretechnische Modell*. Die Notwendigkeit hierzu ergab sich aus den Erkenntnissen, die zum einen bei der Aufarbeitung der Modellbildung im Softwareentwicklungsprozeß und zum anderen aus der Diskussion von Ergebnissen der Semiotik gewonnen wurden.
- Die *Makrostrukturen* des fachlichen Modells beruhen dabei auf der *Organisationsstruktur* des Anwendungsbereichs, die des softwaretechnischen Modells auf denen des fachlichen Modells und der verwendeten Technologie. Diese *Strukturähnlichkeit* zwischen dem Anwendungsbereich, dem fachlichem und softwaretechnischem Modell sowie der verwendeten Technologie ermöglicht, ebenso wie bei Mikrostrukturen, *eine bruchlose Umsetzung der fachlichen Konzepte* des Anwendungsbereichs bis hin in das softwaretechnische Modell. Der erarbeitete Schichtenarchitekturstil kann daher zur Konstruktion von objektorientierten Softwaresystemen verwendet werden, deren Anwendungsbereich nach dem Produkt- oder Prozeßprinzip organisiert ist. Der homologe Aufbau der Modelle und Kontexte erleichtert zudem sowohl die *Etablierung* als auch die *Reetablierung* von *Codes* im Softwareentwicklungsprozeß.
- Da die Makrostrukturen des fachlichen und softwaretechnischen Modells von den Organisationsstrukturen des Anwendungsbereichs abgeleitet wurden, konnte zudem eine *Zerlegung* von Teilen der beiden Modelle *in getrennt voneinander modellierbare Produktbereiche* erzielt werden. Insbesondere diese Zerlegung schafft die Grundlage für eine evolutionäre Entwicklung großer Anwendungssysteme.
- Die Schichtenarchitektur im softwaretechnischen Modell nimmt eine *saubere Trennung zwischen der verwendeten Technik und dem Modell der verwendeten Technik* in Form der beiden Schichten Systembasis und Technologie vor. Dies ermöglicht es, Softwaresysteme, die nach diesem Architekturstil konstruiert worden sind, auf unterschiedliche Systemplattformen zu portieren. Außerdem ist die einheitliche Verwendung eines Technikmodells innerhalb einer Familie von Anwendungssystemen gewährleistet.
- Eine *konstruktiv saubere Realisierung* der vorgestellten Architekturstile, die vor allem eine einfache Evolution der Schichten und der darin entwickelten Komponenten ermöglicht, wird durch die beiden Entwurfsmuster Produkthändler und Rollen gewährleistet.
- Die *Konfiguration von Anwendungssystemen* auf der Basis von Komponenten des präsentierten Schichtenarchitekturstils läßt sich mittels der Händlerkonfigurationsskripten realisieren.

Die nachfolgende Argumentationskette faßt nun die inhaltlichen Schwerpunkte und die daraus resultierenden Ergebnisse meiner Arbeit abschließend zusammen und gibt gleichzeitig einen Ausblick auf weitere Forschungsarbeiten.

Während des Softwareentwicklungsprozesses konstruieren die Mitarbeiter eines Projektteams eine Reihe von Modellen, die sich gegenseitig beeinflussen und durch äußere Kontexte geprägt werden. Als prägende Kontexte wurden in Kapitel 2 der Anwendungsbereich, die Technologie sowie die verwendete Technik herausgearbeitet. Der Anwendungsbereich bildet im Softwareentwicklungsprozeß die fachliche Ausgangslage für das zu konstruierende fachliche Modell. Daneben wird es von den eingesetzten Leitbildern und Entwurfsmetaphern geprägt. Das fachliche Modell stellt nun seinerseits wieder die Grundlage für das softwaretechnische Modell dar. Aufgabe des softwaretechnischen Modells ist dabei die Einbettung des fachlichen Modells in einen gegebenen technikgeprägten Kontext.

Diesen Kontext habe ich in meiner Arbeit in die Bereiche verwendete Technik und Modell der verwendeten Technik unterteilt. Die verwendete Technik umfaßt dabei die zur Konstruktion eine Anwendungssystems benötigte Technik sowie die eingesetzte Technik, um das Anwendungssystem so zu gestalten, daß es den gestellten Anforderungen bestmöglich genügen kann. Neben diesem rein auf die Verwendung von Schnittstellenoperationen fokussierten Einsatz der Technik benutzt ein Softwareentwickler Technik auch in Form eines Modells, das er über die zu verwendete Technik besitzt. Dieses Modell repräsentiert das Wissen des Softwareentwicklers oder eines Projektteams darüber, wie die Technik zur Konstruktion des softwaretechnischen Modells zu verwenden ist.

Neben der verwendeten Technik und dem korrespondierenden Modell beeinflussen insbesondere Entwurfsmuster und Softwarearchitekturen die Konstruktion des softwaretechnischen Modells. Entwurfsmuster beschreiben dabei auf Mikroebene, wie ein bestimmtes, kontextbezogenes Entwurfsproblem gelöst werden kann. Eine Softwarearchitektur organisiert und strukturiert das softwaretechnische Modell als eine Menge interagierender Komponenten und Verbindungsstücke. Sie beschreibt demnach den Gesamtaufbau eines Anwendungssystems auf Makroebene. Entwurfsmuster und Softwarearchitekturen stellen somit sich ergänzende Strukturierungsmittel dar, die ein Anwendungssystem auf unterschiedlicher Granularitätsebene organisieren.

Die Gesamtheit der drei Bereiche Leitbilder und Entwurfsmetaphern, Entwurfsmuster und Softwarearchitekturen sowie Modell der verwendeten Technik habe ich unter dem Begriff Technologie zusammengefaßt, da sie die technischen Kenntnisse, Fähigkeiten und Möglichkeiten repräsentieren, die die Konstruktion eines Anwendungssystems beeinflussen. Das so entstandene Prozeßmodell der objektorientierten Softwareentwicklung wurde dann in den anschließenden Kapiteln als Grundlage für die zu diskutierenden Fragestellungen verwendet.

Neben der Grundlage, die das erste Kapitel für meine Arbeit bildet, sehe ich als relevantes Ergebnis insbesondere die von mir vorgestellte Trennung in Modelle und ihre beeinflussenden Kontexte sowie die Unterscheidung in verwendete Technik und Modell der verwendeten Technik.

In Kapitel 3 habe ich zunächst untersucht, welche strukturellen Abhängigkeiten zwischen den erstellten Modellen und ihren prägenden Kontexten vorteilhaft sind. Motiviert wurde diese Fragestellung anhand der Tatsache, daß die mit der objektorientierten Anwendungsentwicklung verbundene bruchlose Modellierung von vielen Autoren als einer der entscheidenden Vorteile gegenüber herkömmlichen Analyse- und Designmethoden gesehen wird. Dieser Vorteil beruht auf der Möglichkeit, die Mikrostrukturen des Anwendungsbereichs (z.B. eine Begriffshierarchie) unter Einsatz einer passenden Technologie (z.B. Objektorientierung) homolog in das fachliche und softwaretechnische Modell zu übertragen (z.B. in eine Klassenhierarchie). Zur Beantwortung der Fragestellung nach strukturellen Abhängigkeiten habe ich Aussagen aus der Semiotik auf das vorgestellte Prozeßmodell der objektorientierten Softwareentwicklung übertragen.

Ausgehend von der erarbeiteten Erkenntnis, daß dem Prozeßmodell ein Signifikationssystem zugrunde liegt, wurde der Zusammenhang zwischen den Modellen und ihren beeinflussenden Kontexten mittels konnotativer und denotativer Semiotiken beschrieben. Die Semiotiken beruhen dabei auf der Etablierung der nachfolgend beschriebenen Codes:

- das Ausdruckssystem »softwaretechnisches Modell« wird durch einen Code mit dem Inhaltssystem »verwendete Technologie« korreliert;
- das Ausdruckssystem »softwaretechnisches Modell« wird durch einen weiteren Code mit dem Inhaltssystem »fachliches Modell« verbunden;
- das Ausdruckssystem »fachliches Modell« wird durch einen Code mit dem Inhaltssystem »Anwendungsbereich« korreliert;
- das Ausdruckssystem »fachliches Modell« wird auch durch einen Code mit dem Inhaltssystem »Leitbild und Entwurfsmetaphern« verbunden;
- das Ausdruckssystem »fachliches Modell« wird zudem durch einen Code mit dem Inhaltssystem »Softwarearchitektur« korreliert.

Für die Etablierung dieser Codes stellt die Struktur von Ausdrucks- und Inhaltssystem eine wichtige Eigenschaft dar. Die erstellten Modelle sollten daher homolog zu den beeinflussenden Kontexten sein. Eine Softwarearchitektur muß folglich immer auf Basis einer Technologie und auf Basis einer Klasse strukturell verwandter Anwendungsbereiche definiert werden. So lassen sich sowohl die im Anwendungsbereich vorhandenen Mikrostrukturen als auch die Makrostrukturen bruchlos zur verwendeten Technologie in das fachliche und softwaretechnische Modell übertragen.

Da in den weiteren Kapitel insbesondere Fragen in Bezug auf Mikro- und Makrostrukturen im Anwendungsbereich und der verwendeten Technologie diskutiert wurden, habe ich weiterhin die mit dem Begriff Struktur verbundenen Merkmale thematisiert. Diese wurden wie folgt angegeben: eine Struktur beschreibt den inneren Aufbau, das Gefüge eines Systems. Sie kann verwendet werden, um unterschiedliche Systeme von einem einheitlichen Gesichtspunkt aus zu standardisieren oder um in Systemen uns vertraute Eigenschaften zu identifizieren.

Offen geblieben, und damit potentiell Gegenstand weiterer Forschungsarbeit, ist in dem Kapitel 3 die Frage, welche Ergebnisse aus der Semiotik der Kommunikation ebenfalls auf den Softwareentwicklungsprozeß übertragbar sind.

Die in Kapitel 3 gestellte Forderung, daß die erstellten Modelle homolog zu ihren beeinflussenden Kontexten zu organisieren sind, implizierte eine Diskussion über Unternehmensorganisationen. Denn anwendungsfachliche Makrostrukturen, die als Grundlage für eine Softwarearchitektur dienen können, sind bei Softwaresystemen für den Dienstleistungsbereich in Organisationen oder in Bereichen einer Organisation zu suchen. Zudem verfügen die Strukturen eines Unternehmens über eine ausreichend hohe Lebensdauer, um als Basis für eine Softwarearchitektur zu dienen.

Die Diskussion der aus der Organisationstheorie bekannten Organisationsmodelle und ihre Gegenüberstellung mit einer rein technologisch motivierten Struktur, die nach den Modularisierungsprinzipien minimierte Kopplung und maximierte Kohäsion gebildet wurde, hat gezeigt, daß sich im Kontext objektorientierter Softwareentwicklung Organisationsstrukturen, die nach dem Produkt- oder Prozeßprinzip gebildet sind, als Grundlage für eine Softwarearchitektur eignen. Bei Unternehmen, die nach dem Prozeßprinzip strukturiert sind, müssen allerdings die Prozesse überwiegend an den Produkten orientiert sein.

Die Übertragung der Ergebnisse aus der Organisationstheorie auf das fachliche Modell hat dort zu den Bereichen Einsatzkontexte und Produktbereiche geführt. In einer daran anschließenden Diskussion über den Inhalt von Produktbereichen habe ich herausgearbeitet, daß zwischen verschiedenen Produktbereichen typischerweise Überschneidungen bestehen, die sich wie folgt charakterisieren lassen: (1) Begriffe werden über die einzelnen Produktbereiche hinweg gemeinsam verwendet, (2) zu Begriffen unterschiedlicher Produktbereiche ist eine gemeinsame Generalisierung vorhanden oder (3) mit Gegenständen, die anscheinend getrennt voneinander in den verschiedenen Produktbereichen existieren, ist eine gemeinsame fachliche Identität verbunden. Die Existenz dieser Überschneidungsarten und die Tatsache, daß die Konstruktion von fachlich integrierten Anwendungssystemen eine gemeinsame fachliche Basis voraussetzt, hat zu der Definition des Gegenstandsbereichs geführt. Dieser umfaßt somit als Modellierungsbereich die fachlichen Kernkonzepte der verschiedenen Produktbereiche.

Der Gegenstandsbereich repräsentiert demnach ein für alle Produktbereiche gemeinsames Modell. Um nicht mit denselben Problemen konfrontiert zu sein, die mit der Entwicklung eines unternehmensweiten Modells (Objekt- oder Datenmodell) verbunden sind, habe ich notwendige Eigenschaften des Gegenstandsbereichs aufgeführt sowie Vorgehensweisen und Konstruktionsprinzipien für seine technische Realisierung beschrieben. Dabei wurde insbesondere das Offen-Geschlossen-Prinzip als ein wichtiges Konstruktionsprinzip identifiziert.

Die von mir in Kapitel 4 vorgeschlagene Methode, die Makrostrukturen für das fachliche und softwaretechnische Modell von den Makrostrukturen des Anwendungsbereichs abzuleiten, sehe ich vor allem für Anwendungssysteme, die für den Dienstleistungssektor entwickelt werden, als wissenschaftlich relevant an. Denn sie ermöglicht es, nicht nur die Mikrostrukturen sondern auch die Makrostrukturen bruchlos bis in das softwaretechnische Modell zu

übertragen. Daher ergibt sich als weitere Forschungsfrage, inwieweit Organisationsstrukturen, die nach dem Verrichtungsprinzip gebildet worden sind, als Grundlage einer Softwarearchitektur für Anwendungssysteme dienen könnten, die nach der klassischen Methode der strukturierten Analyse und des strukturierten Entwurfs entwickelt werden sollen. Denn auch zwischen dem für die Organisation des Unternehmens und dem für die Architektur des Softwaresystems verwendeten Strukturierungsprinzip ist eine Strukturähnlichkeit unverkennbar. Ebenfalls unberücksichtigt blieb die Fragestellung, ob sich das Modell aus Gegenstandsbereich und Produktbereich wieder rekursiv auf einen Produktbereich anwenden läßt. Dies scheint eine konsequente und logische „Wiederverwendung“ des vorgestellten Musters zu sein. Die rekursive Anwendung des Modells wurde von mir aber nicht weiter beleuchtet, da hierzu noch keine Erfahrungen im Bereich großer objektorientierter Systeme existieren.

Ziel des Kapitel 5 war, die in Kapitel 4 erarbeitete Struktur des fachlichen Modells auf ein objektorientiert geprägtes softwaretechnisches Modell zu übertragen. Die Beschreibung der Makrostruktur des softwaretechnischen Modells erfolgte dabei in Form zweier Architekturstile. Ich habe mich in diesem Kapitel zuerst konzeptionell mit den Eigenschaften von Architekturstilen und Architekturbeschreibungssprachen, den Verwendungsformen von Rahmenwerken sowie den Merkmalen von Schichtenarchitekturen auseinandergesetzt. Aufbauend auf diesen konzeptionellen Erörterungen habe ich Beispiele objektorientierter Schichtenarchitekturen diskutiert. Dabei hat sich gezeigt, daß die Drei-Schichten-Architektur zwar eine geeignete Grundlage für eine Systemarchitektur darstellt, sich aber nicht als Basis einer Softwarearchitektur für große objektorientierte Anwendungssysteme eignet. Logische Schichtenarchitekturen, die Komponenten in schnittstellenlose Schichten gruppieren, wie dies etwa auch bei ET++ der Fall ist, eignen sich hingegen bestens. Insbesondere ermöglicht es diese Form der Schichtenarchitektur mittels Vererbung und Polymorphie, das Offen-Geschlossen-Prinzip umzusetzen. Dies bedeutet, daß sich Komponenten tieferliegender Schichten in höherliegenden Schichten einfach erweitern lassen. Eben diese Eigenschaft wurde zuvor als das Kernkonstruktionsprinzip herausgearbeitet, um die Nachteile zu vermeiden, die mit der Entwicklung von unternehmensweiten Objekt- bzw. Datenmodellen verbunden sind.

Ausgehend von dieser Diskussion, den Ergebnissen aus Kapitel 4 sowie der Unterscheidung in verwendete Technik und Technologie, die ich in Kapitel 2 vorgestellt habe, wurde ein objektorientierter Schichtenarchitekturstil definiert. Die Unterscheidung in verwendete Technik und Technologie hat dabei zu den beiden Schichten Systembasis und Technologie geführt. Die Schichten Gegenstandsbereich, Produktbereich und Einsatzkontext wurden strukturerhaltend vom fachlichen auf das softwaretechnische Modell übertragen.

Die durch die logischen Schichten des objektorientierten Schichtenarchitekturstils gruppierten Komponenten und deren Zusammenspiel habe ich mittels eines rahmenwerkbasierten Architekturstils beschrieben. Dabei stellen die Rahmenwerke der Technologieschicht die technische, die Rahmenwerke der Gegenstandsbereichsschicht die fachliche Kombinierbarkeit der in den Produktbereichen und Einsatzkontexten entwickelten Komponenten sicher. Zudem hat die Kombination der beiden Architekturstile eine detailliertere Angabe von Regeln, die das Zusammenspiel von Komponenten in unterschiedlichen Schichten beschreiben, ermöglicht.

Den Abschluß des Kapitels 5 bildet die technische Realisierung der beiden vorgestellten Architekturstile. Dabei habe ich gezeigt, daß dynamische Bibliotheken die mit einem Rahmenwerk verbundenen Ansprüche, Analysen, Entwurfsentscheidungen, Architekturen und Implementierungen wiederzuverwenden, sinnvoll umsetzen. Allerdings führten technische Argumente, wie unnötige Übersetzungen oder eine verbesserte Konfektionierung der Rahmenwerke, zu einer Trennung von Rahmenwerk- und Bibliotheksstruktur. Diese Trennung wurde ebenfalls auf die Komponente Klassenbibliothek übertragen.

Als Grundkonzept zur Lösung dieser technischen Probleme habe ich die Aufteilung von Rahmenwerken und Klassenbibliotheken in einen Konzeptteil und mehrere Realisierungsteile vorgeschlagen. Ausgehend von dieser Unterteilung werden Konzeptteile verschiedener Rahmenwerke und Klassenbibliotheken unter Berücksichtigung von Modularisierungsprinzipien zu dynamischen Bibliotheken zusammengefaßt. Die Bündelung der Implementierungsteile erfolgt analog. Die dadurch entstandene bibliotheksorientierte Sicht auf das aus Schichten, Klassenbibliotheken und Rahmenwerken bestehende softwaretechnische Modell wird mittels virtueller Dateisysteme oder symbolischer Verweise realisiert.

Die Mikrostrukturen eines Anwendungssystems werden durch die eingesetzte Programmiersprache, die Makrostruktur durch die gewählte Softwarearchitektur festgelegt. Die beiden Strukturen eines Anwendungssystems sind aber heute nur unzureichend miteinander gekoppelt. Eine Verbindung existiert i.d.R. nur in Form von Dokumentation. Wird als Softwarearchitektur beispielsweise eine undurchlässige Schichtenarchitektur verwendet, so kann heute auf Mikroebene eine Klasse trotzdem Klassen verwenden, die nicht nur in der angrenzenden, tieferliegenden Schicht enthalten sind. Ein Bruch zwischen der gemäß der gewählten Softwarearchitektur erwünschten Makrostruktur und der durch die implementierte Mikrostruktur tatsächlich realisierten Makrostruktur führt zum Verlust der Gesamtarchitektur des Anwendungssystems (vgl. [GAO95]). Es sind daher neben den Regeln, die für einen Architekturstil festlegen, wie dessen Komponenten und Verbindungsstücke miteinander kombiniert werden dürfen (vgl. [SG96]), Werkzeuge in Softwareentwicklungsumgebungen und Erweiterungen von Programmiersprachen erforderlich, mit denen sich die Makrostruktur eines Anwendungssystems beschreiben und so bei der Übersetzung gegen die realisierten Strukturen prüfen läßt. Diese Fragestellung eröffnet meines Erachtens einen für die Zukunft der Softwaretechnik interessanten Forschungsbereich.

Offengeblieben ist weiterhin die Problematik, welche Auswirkungen der Einsatz unterschiedlicher technologischer Konzepte auf die Schichtenarchitektur hat. Wenn nämlich Rahmenwerke im Gegenstandsbereich und in den Produktbereichen auf unterschiedlicher technologischer Basis konstruiert werden, dann können diese in den Einsatzkontexten nicht mehr einfach zu Anwendungssystemen kombiniert werden. Diese Fragestellung ist damit ebenfalls Gegenstand weiterer Forschungsarbeit.

In Kapitel 5 habe ich deutlich gemacht, daß sich bestimmte Entwurfsmuster als schichtenübergreifende Verbindungsstücke zwischen Rahmenwerken einsetzen lassen. Daher sind in Kapitel 6 die meines Erachtens wichtigen Konzepte Produkthändler und Rollen unter diesem Gesichtspunkt diskutiert worden.

Das Konzept des Händlers setzt sich mit dem Sachverhalt auseinander, daß ein Klient, wenn er eine gewünschte Dienstleistung von einem Objekt in Anspruch nehmen will, zuvor eine Referenz auf das Objekt besitzen muß. Diese kann ein Klient entweder dadurch erhalten, daß ihm das Objekt zuvor durch eine passende Methode übergeben wurde, oder indem er mittels Aufruf der `new` Operation ein entsprechendes Objekt erzeugt. Die Instanziierung des Objekts durch den Aufruf der `new` Operation ist aber unter folgender Rahmenbedingung problematisch: wird die Dienstleistung von einem Objekt erbracht, dessen Klasse in einer höherliegenden Schicht oder in einem benachbarten Produktbereich enthalten ist, so kann dieses auch dann nicht erzeugt werden, wenn der spätere Zugriff auf das Objekt nur unter der Schnittstelle seiner aufgeschobenen Oberklasse erfolgt. Denn der Aufruf der `new` Operation erzeugt eine Abhängigkeit zu einer höherliegenden Schicht oder zu einem benachbarten Produktbereich.

Wie eine ausführliche Diskussion der in [GOF95] vorgestellten Erzeugungsmuster gezeigt hat, lösen diese die aufgezeigte Problematik im Zusammenhang mit einer Schichtenarchitektur nur unzureichend. Ich habe daher das Entwurfsmuster des Produkthändlers präsentiert, bei dem ein Klient die Vermittlung (Instanziierung bzw. Wiederverwendung) eines bestimmten Objekts an einen Händler delegiert. Der Klient beschreibt dabei die Dienstleistung, die von dem Objekt erbracht werden soll, in Form einer Spezifikation. Der Zugriff auf das Objekt erfolgt dann lediglich unter Verwendung von Schnittstellen, die in der Schicht des Klienten oder in einer tieferliegenden Schichten deklariert sind. Folglich können Klienten durch den Einsatz eines Produkthändlers sowohl von der Objekterzeugung als auch vom Typ und somit von der Existenz konkreter Produktklassen vollständig entkoppelt werden.

Seine Konfiguration verwaltet ein Händler in Form von Zuordnungen. Diese Form der Implementierung verbunden mit dem Ansatz, den Aufbau eines Anwendungssystems unter Verwendung von Konfigurationsskripten zu beschreiben, führt zu den folgenden Eigenschaften von Softwaresystemen:

- Die Konfiguration eines Händlers läßt sich dynamisch anpassen. Das Verhalten eines Anwendungssystems kann somit zur Laufzeit einfach verändert werden.
- Anwendungssysteme lassen sich durch Konfigurationsskripten auf der Basis bereits bestehender Rahmenwerke konfektionieren. Die Summe aller Händlerkonfigurationen beschreibt somit den Aufbau des Gesamtsystems auf einer abstrakten Ebene.
- Die Konfiguration eines Anwendungssystems bzw. Rahmenwerks kann verändert werden, ohne daß dazu ein Rahmenwerk geöffnet werden muß. Dieses Merkmal ermöglicht insbesondere eine einfache Evolution von Rahmenwerken und Anwendungssystemen.

Bereits bei der Diskussion des Gegenstandsbereichs in Kapitel 4 wurde deutlich, daß das Konzept der Vererbung nicht ausreicht, um Begriffe des Anwendungsbereichs, die aus dem Blickwinkel verschiedener Produktbereiche unterschiedlich modelliert werden, softwaretechnisch befriedigend umzusetzen. Um diese Problematik im Kontext des vorgestellten Schichtenarchitekturstils sauber lösen zu können, habe ich das Konzept der Rolle eingeführt. Rollen ermöglichen es, die aus der Sicht eines Produktbereichs fachlich zusammengehörigen Zustände und Umgangsformen eines Konzepts in einer eigenen Klasse zu modellieren. Dazu wird ein

Konzept in ein Kern- und mehrere Rollenkonzepte aufgeteilt. Das Kernkonzept ist typischerweise Bestandteil des Gegenstandsbereichs, die einzelnen Rollenkonzepte sind Teil der verschiedenen Produktbereiche. Rollenkonzepte ermöglichen es somit, Kernkonzepte des Gegenstandsbereichs zu erweitern, ohne diesen dazu öffnen zu müssen.

Die objektorientierte Umsetzung von Kern- und Rollenkonzept erfolgt durch korrespondierende Kern- und Rollenobjekte. Um dennoch die fachliche Identität des durch Kern- und Rollenobjekt realisierten Konzepts zu wahren (Kern- und Rollenobjekte stellen eigene technische Identitäten dar) habe ich auf der Grundlage der einschlägigen Literatur die Begriffe Subjekt und Subjektidentität eingeführt. Anschließend habe ich mit dem Rollenmuster die technische Realisierung der Konzepte Kernobjekt, Rollenobjekt und Subjekt in einer typisierten objektorientierten Programmiersprache beschrieben.

Wie die Zusammenfassung zeigt, hat sich die Arbeit mit vielen scheinbar unabhängig voneinander existierenden Bereichen der Softwareentwicklung beschäftigt. Ziel dabei war es, diese Bereiche so miteinander in Relation zu setzen, daß dadurch die Entwicklung großer rahmenwerkbasierter Anwendungssysteme vereinfacht wird. Neben dem konzeptionellen Beitrag, den diese Arbeit hierbei geleistet hat, sind aus ihr auch eine Reihe von softwaretechnisch interessanten Fragestellungen für zukünftige Forschungsarbeiten hervorgegangen.

Literatur

[AB92] M. Aksit und L. Bergmans: Obstacles in Object-Oriented Software Development. OOPSLA ’92, ACM SIGPLAN Notices, 27(10), Oktober 1992, S. 341 - 358.

[AHU83] A. Aho, J. Hopcroft und J. Ullman: Data structures and algorithms. Addison-Wesley, 1983.

[And95] G. Andert: Object Frameworks in the Taligent OS. In: [Lew95], S. 221 - 235.

[And90] P. B. Anderson: A Theory of Computer Semiotics. Cambridge University Press, 1990.

[Bae93] D. Bäumer: GEBOS - Ein Praxisbeispiel zu objektorientierter Analyse und objektorientiertem Entwurf. HMD, Heft 170, März 1993, S. 64 - 78.

[BBE95] A. Birrer, W. R. Bischofberger und T. Eggenschwiler: Wiederverwendung durch Framework-Technik - vom Mythos zur Realität. OBJEKTSpektrum Nr. 5, 1995, S. 18 - 26.

[BBL+94] D. Bäumer, W. R. Bischofberger, H. Lichter, M. Schneider-Hufschmidt, V. Sedlmeier-Scholz und H. Züllighoven: Prototyping von Benutzeroberflächen. UBILAB Technical Report 94.9.2, 1994.

[BBL+96] D. Bäumer, W. R. Bischofberger, H. Lichter und H. Züllighoven: User Interface Prototyping - Concepts, Tools, and Experience. ICSE ’96, IEEE Computer Society Press, März 1996, S. 532 - 541.

[BBS+95] D. Bäumer, R. Budde, K.-H. Sylla, G. Gryczan und H. Züllighoven: Objektorientierte Konstruktion von Software-Werkzeugen und -Materialien. Informatik-Spektrum, 18(4), August 1995, S. 203 - 210.

[BGK+97] D. Bäumer, G. Gryczan, R. Knoll, C. Lilienthal, D. Riehle und H. Züllighoven: Framework Development for Large Systems. Communications of the ACM, 40(10), Oktober 1997, S. 52 - 59.

[BGZ95] D. Bäumer, G. Gryczan und H. Züllighoven: Objektorientierte Software-Entwicklung für Banken - Methodik und Erfahrungen aus einer mehrjährigen Projektpraxis. OBJEKTSpektrum Nr. 3, 1995, S. 45 - 61.

[BH97] F. X. Bea und J. Haas: Strategisches Management. Lucius und Lucius Verlag, Stuttgart, 1997.

[BKG+95] D. Bäumer, R. Knoll, G. Gryczan, W. Strunk und H. Züllighoven: Objektorientierte Entwicklung anwendungsspezifischer Rahmenwerke. OBJEKTSpektrum Nr. 6, 1995, S. 48 - 59.

[BKG+96] D. Bäumer, R. Knoll, G. Gryczan und H. Züllighoven: Large Scale Object-Oriented Software-Development in a Banking Environment - An Experience Report. In: ECOOP '96. Hrsg.: C. Pierre. Springer, Juli 1996, S. 73 - 90.

[BKK+96] R. J. Brachman, T. Khabaza, W. Kloesgen, G. Piatetsky-Shapiro und E. Simoudis: Mining Business Databases. Communications of the ACM, 39(11), November 1996, S. 42 - 48.

[BL79] L. A. Belady und M. M. Lehman: The Characteristics of Large Systems. In: Research Directions in Software Technology. Hrsg.: P.Wegner. MIT Press, 1979.

[Ble91] K. Bleicher: Organisation: Strategien – Strukturen – Kulturen. Gabler Verlag, 1991.

[Boo91] G. Booch: Object Oriented Design with Applications. The Benjamin/Cummings Publishing Company, 1991.

[BR97] D. Bäumer und D. Riehle: Product Trader. In: Pattern Languages of Program Design 3. Hrsg.: R. C. Martin, D. Riehle und F. Buschmann. Addison-Wesley, 1997, Kapitel 3, S. 29 - 46.

[Bro86] Brockhaus Enzyklopädie. F. A. Brockhaus, Mannheim, 1986.

[BRS+97] D. Bäumer, D. Riehle, W. Siberski und M. Wulf: Role Object. PLoP ’97, Conference Proceedings. Washington University Department of Computer Science, Technical Report WUCS-97-34, 1997, Paper 2.1, 10 Seiten.

[Bus96] F. Buschmann: Reflection. In: Pattern Languages of Program Design 2. Hrsg.: J. M. Vlissides, J. O. Coplien und N. L. Kerth. Addison-Wesley, 1996, S. 271 - 294.

[BZ90] R. Budde und H. Züllighoven: Software-Werkzeuge in einer Programmierwerkstatt. Berichte der Gesellschaft für Mathematik und Datenverarbeitung Nr. 182, Oldenbourg, 1990.

[CD94] S. Cook und J. Daniels: Designing Object Systems. Object-oriented modelling with Syntropy. Prentice Hall, 1994.

[CE95] M. Carroll und M. Ellis: Tradeoffs of interface classes. C++ Report, 7(1), Januar 1995, S. 29 - 32.

[Cot95] S. Cotter, mit M. Potel: Inside Taligent Technology. Addison-Wesley, 1995.

[CR90] J. M. Carroll und M. B. Rosson: Human computer interaction scenarios as design representation. Proceedings of the Hawaii International Conference on System Sciences. IEEE Computer Society Press 1990, S. 555 - 561.

[CUL89] C. Chambers, D. Ungar und E. Lee: An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. OOPSLA ‘89, ACM SIGPLAN Notices, 24(10), Oktober 1989, S. 49 - 70.

[Cun85] W. Cunningham: The Construction of SMALLTALK-80 Applications. Computer Research Laboratory Tektronix, Inc. Beaverton, Oregon 97077, 1995.

[CY91] P. Coad und E. Yourdon: Object-Oriented Analysis. Yourdon Press Computing Series, Prentice Hall, 1991.

[CY91a] P. Coad und E. Yourdon: Object-Oriented Design. Yourdon Press Computing Series, Prentice Hall, 1991.

[DeM71] T. De Mauro: Senso e significato. Bari, Adriatica, 1971.

[Dud90a] Duden: Das Fremwörterbuch. Duden Band 5, Dudenverlag, 1990.

[Eco72] U. Eco: Einführung in die Semiotik. Wilhelm Fink Verlag, 1972.

[Eco73] U. Eco: Zeichen, Einführung in einem Begriff und seine Geschichte. Suhrkamp Verlag, 1977.

[Eco76] U. Eco: Semiotik, Entwurf einer Theorie der Zeichen. Wilhelm Fink Verlag, 1987

[Eng95] T. Engelmann: Business Process Reegineering. Grundlagen - Gestaltungs-empfehlungen - Vorgehensmodell. Deutscher Universitäts-Verlag, 1995.

[ET65] F. E. Emery und E. L. Trist: The causal texture of organizational environments. Human Relations, 1965, S. 455 - 468.

[FKR+97] C. Floyd, A. Krabbel, S. Ratuski und I. Wetzel: Zur Evolution der evolutionären Systementwicklung: Erfahrungen aus einem Krankenhausprojekt. Informatik-Spektrum, 20(1), Februar 1997, S. 13 - 20.

[Flo87] C. Floyd: Outline of a Paradigm Change in Software Engineering. In: Computer and Democracy - A Scandinavian Challenge. Hrsg.: G. Bjerknes, P. Ehn und M. Kyng. Gower Publishing Company Limited, Avebury, 1987, S. 191 - 210.

[Flo94] C. Floyd: Software-Engineering - und dann? Informatik-Spektrum, 17(1), Februar 1994, S. 29 - 37.

[Fre92] E. Frese: Handwörterbuch der Organisation. C. E. Poeschel Verlag, Stuttgart, 1992.

[Gai83] M. Gaitanides: Prozeßorganisation: Entwicklung, Ansätze und Programme prozeßorientierter Organisationsgestaltung. Verlag Franz Vahlen, München, 1983.

[Gam92] E. Gamma: Objektorientierte Software-Entwicklung am Beispiel von ET++. Design-Muster, Klassenbibliothek, Werkzeuge. Springer, 1992.

[Gam97] E. Gamma: Extension Object. In: Pattern Languages of Program Design 3. Hrsg.: R. C. Martin, D. Riehle und F. Buschmann. Addison-Wesley, 1997, Kapitel 6, S. 79 - 88.

[GAO95] D. Garlan, R. Allen und J. Ockerbloom: Architectural Mismatch: Why Reuse Is So Hard. IEEE Software, 12(6), November 1995, S. 17 - 26.

[GOF95] E. Gamma, R. Helm, R. Johnson und J. Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[Gry96] G. Gryczan: Prozeßmuster zur Unterstützung kooperativer Tätigkeit. Deutscher Universitäts-Verlag, 1996.

[GSR95] G. Gottlob, M. Schrefl und B. Röck: Extending Object-Oriented Systems with Roles. ACM Transactions on Information Systems, 14(3), Juli 1996, S. 268 - 296.

[GZ92] G. Gryczan und H. Züllighoven: Objektorientierte Systementwicklung - Leitbild und Entwicklungsdokumente. Informatik-Spektrum, 15(5), Oktober 1992, S. 264 - 272.

[HO93] W. Harrison und H. Ossher: Subject-Oriented Programming (A Critique of Pure Objects). OOPSLA ’93, ACM SIGPLAN Notices, 28(10), Oktober 1993, S. 411 - 428.

[Inm96] W. H. Inmon: The Data Warehouse and Data Mining. Communications of the ACM, 39(11), November 1996, S. 49 - 50.

[ISO93a] ISO/ITU: ITU-T X.901 | ISO/IEC 10746-1 ODP Reference Model Part 1.

[Jac92] I. Jacobson, M. Christerson, P. Jonsson und G. Övergaard: Object-oriented Software Engineering. A Use Case Driven Approach. Addison-Wesley, 1992.

[Jac94] M. Jackson: Problems, Methods, and Specialisation. IEEE Software Engineering Journal, 9(6), November 1994, S. 249 - 255.

[JF88] R. E. Johnson und B. Foote: Designing Reusable Classes. The Journal of Object-Oriented Programming, 1(2), Juni/Juli 1988, S. 22 - 35.

[Joh92] R. Johnson: Documenting Frameworks using Patterns. OOPSLA ’92, ACM SIGPLAN Notices, 27(10), Oktober 1992, S. 63 - 70.

[JR91] R. E. Johnson und V. F. Russo: Reusing Object-Oriented Designs. Department of Computer Science, University of Illinois at Urbana-Champaign, 1991.

[KGZ+94] K. Kilberth, G. Gryczan, H. Züllighoven, D. Bäumer, R. Budde, K. Hasbron-Blume, K.-H. Sylla und V. Weimer: Objektorientierte Anwendungsentwicklung. Vieweg, 1994.

[KØ96] B. B. Kristensen und K. Østerbye: Roles: Conceptual Abstraction Theory and Practical Language Issues. Theory and Practice of Object Systems, 2(3), 1996, S. 143 - 160.

[KP88] G. Karsner und S. Pope: A CookBook for using the Model-View-Controller User Interface in SMALLTALK-80. ParcPlace Systems, 2400 GengRoad Palo Alto, CA 94303.

[Lak96] J. Lakos: Large Scale C++ Software Design. Addison-Wesley, 1996.

[Ler94] X. Leroy: Manifest Types, Modules, and Separate Compilation. Proceedings POPL ’94, ACM Press, 1994, S. 109 - 122.

[Lew95] T. Lewis: Object-Oriented Application Frameworks. Manning Publications, Greenwich, 1995.

[Lic93] H. Lichter: Entwicklung und Umsetzung von Architekturprototypen für Anwendungssoftware. Verlag der Fachvereine Zürich, 1993.

[Lis88] B. Liskov: Data Abstraction and Hierarchy. OOPSLA ‘87 Addendum to the Proceedings, ACM SIGPLAN Notices, 23(5), Mai 1988, S. 17 - 34.

[LKA+95] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan und W. Mann: Specification and Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering, 21(4), April 1995, S. 336 - 355.

[LW93] B. Liskov und J. M. Wing: Specifications and Their Use in Defining Subtypes. OOPSLA ‘93, ACM SIGPLAN Notices, 28(10), Oktober 1993, S. 16 - 28.

[Mad94] K. H. Madsen: A Guide to Metaphorical Design. Communications of the ACM, 37(12), Dezember 1994, S. 57 - 62.

[Mar96a] R. C. Martin: The Open - Closed Principle. C++ Report, 8(1), Januar 1996, S. 37 - 42.

[Mar96b] R. C. Martin: The Liskov Substitution Principle. C++ Report, 8(3), März 1996, S. 14 - 23.

[MB96] K.-U. Mätzel und W. R. Bischofberger: Evolution of Object Systems or How to tackle the Slippage Problem in object systems. In: Computer Science Research at UBILAB. Hrsg.: K.-U. Mätzel und H.-P. Frei. Universitätsverlag Konstanz, 1996, S. 99 - 117.

[McC91] G. R. McClain: Open Systems Interconnection Handbook. Intertext Publications McGraw-Hill, New York, 1991.

[McG60] D. McGregor: The human side of enterprise. New York, 1960.

[Mey88] B. Meyer: Object-Oriented Software Construction. Prentice Hall, 1988.

[Mey92] B. Meyer: Eiffel - The Language. Prentice Hall, 1992.

[MG92] E. Metalla und M. H. Graham: The Domain Specific Software Architecture Program. Special Report CMU/SEI-92-SR-9, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Juni 1992.

[MKM+97] R .T. Monroe, A. Kompanek, R. Melton und D. Garlan: Architectural Styles, Design Patterns, and Objects. IEEE Software, 14(1), Januar/Februar 1997, S. 43 - 52.

[ND95] O. Nierstrasz und Laurent Dami: Component-Oriented Software Technology. In: [NT95], S. 3 - 28.

[NT95] O. Nierstrasz und D. Tsichritzis: Object-oriented Software Composition. Prentice Hall, 1995.

[Ode92] J. J. Odell: Dynamic and multiple classification. The Journal of Object-Oriented Programming, 4(8), Januar 1992, S. 45 - 48.

[Ode92a] J. J. Odell: Managing object complexity, part I: abstraction and generalization. The Journal of Object-Oriented Programming, 5(5), September 1992, S. 19 - 22.

[Ode92b] J. J. Odell: Managing object complexity, part II: composition. The Journal of Object-Oriented Programming, 5(6), October 1992, S. 17 - 20.

[OKH+95] H. Ossher, M. Kaplan, W. Harrison, A. Katz und V. Kruskal: Subject-Oriented Composition Rules. OOPSLA ‘95, ACM SIGPLAN Notices, 30(10), Oktober 1995, S. 235 - 250.

[OKK+96] H. Ossher, M. Kaplan, A. Katz, W. Harrison und V. Kruskal: Specifying Subject-Oriented Composition. In: Theory and Practice of Object Systems, 2(3), 1996, S. 179 - 202.

[OMG96] Object Management Group: CORBA 2.0, Common Object Services and Common Facilities Specification. 1996.

[Oss87] H. L. Ossher: A Mechanism for Specifying the Structure of Large, Layered, Systems. In: Research Directions in Object-Oriented Programming. Hrsg.: B. Shriver und P. Wegner. MIT Press, 1987, S. 219 - 252.

[PB93] G. Pomberger und G. Blaschek: Software Engineering. Prototyping und objektorientierte Software-Entwicklung. Carl Hanser Verlag, 1993.

[PHV96] Y. Pant, B. Henderson-Sellers und J. M. Verner: Generalization of object-oriented components for reuse: Measurements of effort and size change. The Journal of Object-Oriented Programming, 9(2), Mai 1996, S. 19 - 31.

[Pie94] U. Piepenburg: Semantikerstellung in der Softwareentwicklung. Dissertation am Fachbereich Informatik der Universität Hamburg, 1994.

[Pin95] X. Pintado: Gluons and the Cooperation between Software Components. In: [NT95], S. 321 - 349.

[Ree96] T. Reenskaug, mit Per Wold und Odd Arild Lehne: Working with Objects. Manning, Greenwich, 1996.

[Rie95] D. Riehle: How and Why to encapsulate Class Trees. OOPSLA ‘95, ACM SIGPLAN Notices, 30(10), Oktober 1995, S. 251 - 264.

[Rie95a] D. Riehle: Muster - am Beispiel der Werkzeug und Material Metapher. UBILAB Technical Report 95.6.1, 1995.

[Rie97] D. Riehle: A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose. Ubilab Technical Report 97.1.1., 1997.

[Rol92] A. Rolf: Sichtwechsel - Informatik als (gezähmte) Gestaltungswissenschaft. In: Sichtweisen der Informatik. Hrsg.: W. Coy et al. Vieweg, 1992.

[RSB+97] D. Riehle, W. Siberski, D. Bäumer, D. Megert und H. Züllighoven: Serializer. In: Pattern Languages of Program Design 3. Hrsg.: R. C. Martin, D. Riehle und F. Buschmann. Addison-Wesley, 1997, Kapitel 17, S. 293 - 312.

[Rum91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy und W. Lorensen: Object-Oriented Modeling and Design. Prentice-Hall, 1991.

[Rum93] J. Rumbaugh: Objects in the Twilight Zone. The Journal of Object-Oriented Programming, 6(3), Januar 1993, S. 18 - 24.

[RWG95] Die GEBOS-Methode - Dokumenttypen und Vorgehensweisen. RWG GmbH Stuttgart, 1995.

[RZ95] D. Riehle und H. Züllighoven: A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor. In: Pattern Languages of Program Design. Hrsg.: J. O. Coplien und D. C. Schmidt. Addison-Wesley, 1995, S. 9 - 42.

[Sch93] A. Schill: DCE – Das OSF Distributed Computing Environment. Einführung und Grundlagen. Springer, 1993.

[SDK+95] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young und G. Zelesnik: Abstractions for Software Architecture and Tools to Support Them. IEEE Transactions on Software Engineering, 21(4), April 1995, S. 314 - 335.

[SG96] M. Shaw und D. Garlan: Software Architecture. Perspectives on an emerging discipline. Prentice Hall, 1996.

[SM93] S. Shlaer und S. J. Mellor: A deeper look at the transition from analysis to design. The Journal of Object-Oriented Programming, 5(9), Februar 1993, S. 16 - 21.

[SM93a] S. Shlaer und S. J. Mellor: A deeper look at partitioning the work on a project. The Journal of Object-Oriented Programming, 6(5), September 1993, S. 8 - 13, 22.

[Som95] I. Sommerville: Software Engineering. 5. Auflage, Addison-Wesley, 1995.

[Sta90] W. H. Staehle: Management. Eine verhaltenswissenschaftliche Perspektive Verlag Franz Vahlen, München, 1990.

[Sta91] M. Stadel: Object Oriented Programming Techniques to Replace Software Components on the Fly in a Running System. ACM SIGPLAN Notices, 26(1), Januar 1991, S. 99 - 108.

[Ste95] P. A. Steinbruch: Organisation. Kiehl Verlag, 1995.

[Str97] A. Stritzinger: Komponentenbasierte Softwareentwicklung: Konzepte und Techniken für das Programmieren in Smalltalk. Addison-Wesley-Longman, 1997.

[Str93] B. Stroustrup: The C++ Programming Language. Addison-Wesley, 1993.

[Tra95a] W. Tracz: Domain-Specific Software Architecture (DSSA). Frequently Asked Questions. Lockheed Martin, ADAGE-IBM-93-12B, März 1995.

[Tra95b] W. Tracz: DSSA (Domain-Specific Software Architecture) Pedagogical Example. Software Engineering Notes, 20(3), Juli 1995, S. 49 - 62.

[Vli95] J. M. Vlissides: Unidraw: A Framework for Building Domain-Specific Graphical Editors. In: [Lew95], S. 239 - 290.

[Weg87] P. Wegner: Dimensions of Object-Based Language Design. OOPSLA ‘87, ACM SIGPLAN Notices, 22(12), Dezember 1987, S. 168 - 182.

[Wei94] R. Weinreich: Concepts and Techniques for Object-Oriented Software Development. Universitätsverlag Rudolf Trauner, Linz 1994.

[WG95] A. Weinand und E. Gamma: ET++ A Portable Homogeneous Class Library and Framework. In: [Lew95], S. 154 - 194.

[Win92] J. F. H. Winkler: Objectivism: Class considered harmful. Communications of the ACM, 35(8), August 1992, S. 128 - 130.

[WWW93] R. J. Wirfs-Brock, B. Wilkerson und L. Wiener: Objektorientiertes Software-Design. Carl Hanser, 1993.

[YC79] E. Yourdan und L. L. Constantine: Structured Design. Prentice-Hall, 1979.

Index

H

I

K

L

M

O

P

R

S

T

U

V

W

Z