Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Diplomarbeit

FAME-DBMS: Entwurf ausgewählter Aspekte der Transaktionsverwaltung

Verfasser:

Mario Pukall

2. Januar 2006

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Wirtsch.-Inf. Thomas Leich

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg
Germany

Pukall, Mario:
FAME-DBMS: Entwurf ausgewählter Aspekte der Transaktionsverwaltung
Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, 2005.

OTTO-VON-GUERICKE-UNIVERSITÄT MAGDEBURG

FAKULTÄT FÜR INFORMATIK

INSTITUT FÜR TECHNISCHE
UND BETRIEBLICHE
INFORMATIONSSYSTEME

Thema der Diplomarbeit:
FAME-DBMS: Entwurf ausgewählter Aspekte der Transaktionsverwaltung

Aufgabenstellung:

Im Bereich der eingebetteten DBMS ergeben sich aufgrund des oftmals stark limitierten Ressourcenangebotes bzgl. Hard- und Software vielfältige Einschränkungen, welche die Installation eines flexiblen und leistungsstarken DBMS behindern. Existierende, kommerzielle Systeme wurden in ihrer Funktionalität in der Vergangenheit derart erweitert, dass ihr Betrieb hohe Anforderungen gerade an die Hardware stellt. Daher ist der Einsatz solcher Systeme im Bereich der eingebetteten DBMS zumeist undenkbar. Ein denkbarer Ansatz wird unter dem Begriff FAME-DBMS (*Family of Embedded Data Base Management Systems*) zusammengefasst. Ausgehend von Basis wird die Funktionalität schrittweise erweitert. Diese Erweiterungen werden in einer Programmfamilie organisiert. Der große Vorteil liegt in der flexiblen Wiederverwendbarkeit der entwickelten Software.

Ausgehend von allgemeinen Grundlagen zu konfigurierbaren Datenmanagement beschäftigt sich die Diplomarbeit detailliert mit dem Bereich der Transaktionsverwaltung. Dabei soll zunächst untersucht werden, welche Ansätze bezüglich flexibeler, konfigurierbarer DBMS in der Vergangenheit verfolgt wurden. Den zentralen Bereich der Arbeit nimmt der Entwurf einer konfigurierbaren Transaktionsverwaltung ein, welcher in Form einer Programmfamilie organisiert wird (z.B. aus dem Bereich Protokollierung oder Sperrverwaltung). Für die Bearbeitung der Arbeit sind folgende Teilaufgaben zu bearbeiten:

- Diskussion existierender Ansätze konfigurierbarer DBMS mit dem Schwerpunkt Transaktionsverwaltung
- Domänenanalyse der Transaktionsverwaltung
- Entwurf feingranularer Komponenten gemäß des Ansatzes der Programmlinie/-familie
- Untersuchungen bzgl. heterogener und homogener "crosscutting concerns"
- Prototypische Implementierung einzelner Komponenten der Transaktionsverwaltung

Name:	Pukall	**geb. am:**	04.09.1978
Vorname:	Mario	**geb. in:**	Salzwedel

Betreuer der Arbeit: Prof. Dr. rer. nat. habil. Gunter Saake,
Dipl.-Wirtsch.-Inf. Thomas Leich

Tag der Ausgabe der Aufgabenstellung: 15.06.2005
Tag der Einreichung der Arbeit: 15.11.2005

Magdeburg, 20. Juni 2004

Betr. Hochschullehrer

Danksagung

An dieser Stelle möchte ich mich bei allen Menschen, die mich bei der Fertigstellung dieser Diplomarbeit unterstützt haben, herzlich bedanken. Besonderen Dank für die hervorragende Betreuung möchte ich Prof. Dr. rer. nat. habil. Gunter Saake und Dipl.-Wirtsch.-Inf. Thomas Leich aussprechen. In intensiven Diskussionen gelang es Herrn Thomas Leich die wichtigen Aspekte der Aufgabenstellung in den Fokus zu rücken und dadurch die Qualität der Arbeit entscheidend zu verbessern. Zusätzlich war Dipl.-Inf. Sven Apel bei Fragen zu FeatureC++ ein verlässlicher Ansprechpartner. Für die freundliche Bereitstellung des Informationsmaterials zum PLENTY-System möchte ich ebenfalls Prof. Dr. Gerhard Weikum danken.

Darüber hinaus möchte ich mich bei meiner Freundin, Familie und meinen Freunden für ihre Geduld und ihr Verständnis bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Verzeichnis der Abkürzungen

ADD	Attribute Driven Design
AHEAD	Algebraic Hierarchical Equations for Application Design
AML	Aspectual Mixin Layer
AOP	Aspect-Oriented Programming
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction-Set Processor
ATS	AHEAD Tool Suite
COMET	Component-Based Embedded Real-Time (DBMS)
CDL	Component Definition Language
DBMS	Database Management System
DRC	Design Rule Check
DSP	Digital Signal Processor
DSSA	Domain Specific Software Architecture
FAME-DBMS	Family of Embedded Data Base Management Systems
FIFO	First In First Out
FODA	Feature-Oriented Analysis
FOP	Feature-Oriented Programming
FPGA	Field-Programmable Gate-Array
GUI	Graphical User Interface
IDL	Interface Definition Language
ODM	Organization Domain Modeling
OOP	Object-Oriented Programming
PDA	Personal Digital Assistant
PLENTY	Parallel Execution of Nested Transactions on Plenty of Processors
SQL	Structured Query Language

Kapitel 1

Einleitung

Ein Ziel dieses Kapitels ist die Formulierung eines motivierenden Beispiels, das die Bedeutung des Themengebiets darlegt. Aufbauend auf der Themenstellung werden die Ziele diskutiert. Abschliessend werden Angaben zur Gliederung der Arbeit gemacht.

1.1 Motivation

Computer werden mittlerweile auf so vielfältige Art und Weise eingesetzt, dass es schwierig ist, überhaupt ein Gebiet zu benennen, in dem auf die Hilfe solcher Maschinen verzichtet werden kann. Vereinfacht beschrieben, haben Computer die Aufgabe, unter Verwendung geeigneter Anwendungsprogramme, aussagekräftige Informationen zu gewinnen. Die Leistungsfähigkeit dieser Maschinen ermöglicht es in Bruchteilen einer Sekunde so grosse Mengen an Daten zu produzieren, dass es zwingend nötig ist, diese zu organisieren. Zu dem Zweck wurde schon frühzeitig mit der Entwicklung von geeigneten Datenhaltungssystemen begonnen. Sie sollten wichtige Aspekte wie: Redundanz, Konsistenz und Ausfallsicherheit der Daten aufgreifen und effizient behandeln. Weit verbreitet sind heute so genannte *Datenbankmanagementsysteme* (kurz: *DBMS*), wie z.B. *Oracle* oder *MySQL*, welche die Daten einer Datenbank verwalten.

DBMS sind heute so mächtig und in ihrer Funktionsvielfalt so umfangreich, dass die damit verbundenen Hardware-Anforderungen längst nicht mehr von jedem System erfüllt werden. Sicherlich ist es, aus dem Blickwinkel der Entwickler betrachtet, sinnvoll ein Produkt zu entwickeln, welches möglichst viele Anwendungsszenarien abdeckt. In den meisten Fällen wird jedoch ein Grossteil der Funktionen nicht benötigt. Soll trotzdem auf die Dienste eines derartigen Systems zurückgegriffen werden, so muss ein ″Overhead″ an Funktionalität in Kauf genommen werden, der die Anforderungen an die Hardware künstlich erhöht. Besonders schwer wiegt diese Tatsache, wenn die vorhandenen Systemressourcen grundsätzlich für den Anwendungsfall ausreichen, aber die Voraussetzungen für den Betrieb des gewählten DBMS nicht erfüllt werden.

Hierbei kristallisiert sich, bezogen auf verfügbare Ressourcen und Anforderungen, eine besonders sensible Systemklasse heraus: die *eingebetteten Systeme* (engl.: Embedded

Systems). Innerhalb dieser Systemklasse ist es zwingend notwendig mit vorhandenen Ressourcen, wie: Rechenleistung, Speicherplatz oder Energiereserven, sparsam umzugehen. Ideal wäre hier der Einsatz eines *schlanken* DBMS, das nur so viel Funktionalität mitbringt, wie für den Anwendungsfall nötig ist.

Um derartige DBMS auf verschiedenen Systemen (PDA, Notebook, verschiedene Rechnerarchitekturen, etc.) einsetzen zu können, muss sie möglichst flexibel gestaltet werden, da erneute aufwendige Anpassungen, aufgrund von Spezialfällen, zu aufwendig und letztendlich teuer sind. Dies führt zu der Überlegung, die Software und ihre einzelnen Funktionen noch restriktiver zu modularisieren, als dies herkömmliche Entwicklungsansätze ohnehin tun. Damit ist es möglich, die Komponenten für den aktuellen Anwendungsfall passend zu kombinieren, ohne grössere Umstellungen an den einzelnen Programmmodulen vornehmen zu müssen.

1.2 Zielstellung

Speziell im Bereich der eingebetteten Systeme ergeben sich gewisse Einschränkungen und Forderungen bezüglich der eingesetzten Software. Dies trifft insbesondere auf den Betrieb von Datenbankmanagementsystemen und deren Datenbanken zu. Gerade hier gilt es, eine gute Balance aus Geschwindigkeit und Funktionalität zu finden. Ausgehend von diesem Betrachtungswinkel, widmet sich die Arbeit einem zentralen Bereich der DBMS - der *Transaktionsverwaltung*. Ohne Transaktionsverwaltung ist es unmöglich, die grosse Menge von Operationen zu koordinieren, welche im Datenbankbetrieb typischerweise anfällt. Daher sollte jedes, noch so primitive, DBMS zumindest grundlegende Funktionen der Transaktionsverwaltung bereitstellen.

Wie aus der Aufgabenstellung hervorgeht, ist der Aspekt der modularisierten Entwicklung von zentraler Bedeutung. In der Vergangenheit hat es verschiedene, mehr oder weniger vollständige Versuche gegeben, sich diesem Thema zu nähern. Daher werden einige der Ansätze näher betrachtet, und so weit dies möglich ist, mit der eigenen Umsetzung verglichen.

Der zweite Teilbereich dieser Arbeit beinhaltet die modularisierte, prototypische Entwicklung ausgesuchter Teilaspekte der Transaktionsverwaltung. Die zugrunde liegende Idee verbirgt sich hinter dem Begriff *FAME-DBMS* (*Family of Embedded Data Base Management Systems*). Hier werden dem Prototypen, schrittweise, benötigte Funktionen hinzugefügt. Dadurch ist es möglich, nur die geforderten Funktionen in das Endprodukt einzugliedern, und ein schlankes, minimales Programm zu erzeugen, welches trotzdem alle Aufgaben erfüllt. Die technische Umsetzung dieser Vorgehensweise erfolgt durch Erweiterungen des herkömmlichen Paradigmas der objektorientierten Programmierung, in Form von merkmals- (engl.: **F**eature-**O**riented **P**rogramming - *FOP*) und aspektorientierter Programmierung (engl.: **A**spect-**O**riented **P**rogramming - *AOP*). Dabei gilt es die Erweiterungen möglichst feingranular zu gestalten, um ein breites Spektrum an Anwendungsfällen abzudecken und trotzdem, eine möglichst minimale Lösung anzubieten. Die Entwicklung wird begleitet und unterstützt von grundlegenden Techniken, insbesondere

aus dem Bereich der Domänenentwicklung.

1.3 Gliederung

Die Arbeit gliedert sich wie folgt. In Kapitel 2 wird die nötige Wissensbasis gelegt, um sich der Thematik zu nähern. Dabei wird genauer auf verschiedene Strategien und Technologien zur restriktiven modularen Software-Entwicklung eingegangen. Die erste Phase der gewählten Entwicklungsstrategie (Domänenentwicklung) wird in Kapitel 3 beschrieben. Die Ergebnisse der Domänenanalyse werden in Kapitel 4 aufgegriffen und für die Entwurfsentwicklung (Domänenentwurf) genutzt. Kapitel 5 befasst sich mit der Phase der prototypischen Implementierung, auf Basis der aspekt- und merkmalsorientierten Programmierung. Das Erreichen der formulierten Ziele wird in Kapitel 6 evaluiert. Einen zusammenfassenden Überblick über den Inhalt und die Ergebnisse der Arbeit gibt Kapitel 7.

Kapitel 2

Grundlagen

Für das Verständnis des Themengebiets bedarf es einiger einführender Erläuterungen zu verwendeten Technologien, Werkzeugen und Vorgehensweisen. Auf diese soll im Kapitel der Grundlagen näher eingegangen werden. Dabei handelt es sich nicht nur um datenbankspezifische Themen, sondern auch um Konzepte zu Softwareentwicklung und Implementierungstechniken.

2.1 Software-Produktlinien vs. Programmfamilien

Die Idee der Wiederverwendung von existierenden, bewährten Lösungen in neuen Umgebungen ist eine viel versprechende Variante zur Entwicklung neuer Produkte, die viele Vorteile aber auch einige Probleme mit sich bringt. Zu den Vorteilen gehört sicherlich eine wesentlich kürzere Zeitspanne bis zur Marktreife des Produktes bei deutlicher Verringerung des Entwicklungsaufwands. Hinzu kommt, wie schon angedeutet, eine positive Beeinflussung der Produktqualität durch den Einsatz erprobter Komponenten. Daraus ergibt sich gleichzeitig ein nicht zu vernachlässigender Nachteil - das Problem der Fehlerfortpflanzung. Es kann schliesslich vorkommen, dass ein Produkt im Detail fehlerhaft ist, dieser Umstand aber noch nicht zum Tragen gekommen ist. In der neuen Anwendungsumgebung kommt dieser durch neue Parameter und Anforderungen plötzlich zum Vorschein.

Während die Vorteile der Produktlinie in hardware-basierten Industriezweigen, wie etwa der Automobilindustrie, schon lange genutzt wurden, gab es im Bereich der Software-Entwicklung einige Schwierigkeiten, sich dem Thema zu nähern. Erste Ansätze, um Programmteile einer mehrfachen Nutzung zugänglich zu machen finden sich in der *prozeduralen* Programmierung. Hier wird die Wiederverwendbarkeit von Prozeduren realisiert. Weitergeführt wurde dieser Ansatz in der *objektorientierten* Programmierung durch Eigenschaften wie z.B. Vererbung. Die Idee der Produktlinie verleiht solchen Ansätzen zusätzliche strukturelle Möglichkeiten. Ein sehr allgemeiner Annäherungsversuch zur Klärung des Produktlinienbegriffs wird von Bass et. al [Bas03] geliefert. Er beschreibt ihn als den Übergang vom einzelnen System hin zu vielen, auf dem Initialsys-

tem gründenden, Systemen.

Clements und Northrop [Cle01] fassen eine genauere Definition der Software-Produktlinie in folgende Worte:

> *"A* software product line *is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way "*

Für Bosch [Bos00] steht die Software-Architektur im Mittelpunkt der Entwicklung von Software-Produktlinien, da die Produkte einer Linie auf Basis vergleichbarer Strukturen entworfen, organisiert und umgesetzt werden können.

Der von Parnas [Par79] eingeführte Begriff der *Programmfamilien* und Produktlinien werden fälschlicherweise oft synonym verwendet. Während jedoch die Zugehörigkeit zu einer Produktlinie aufgrund gemeinsamer Produkteigenschaften und identischer Zielgruppe am Markt festgelegt wird, gelten für die Einordnung in Programmfamilien andere Kriterien. Die Zugehörigkeit wird über die Menge der *funktionalen* Übereinstimmungen der beteiligten Programme definiert. So können z.B. Videospiele der gleichen Programmfamilie angehören, wie auch Video-Software. Es ist offensichtlich, dass es sich hier um Vertreter gänzlich verschiedener Produktlinien handelt. Werden die beiden Systeme jedoch in Hinblick auf die Umsetzung der graphischen Benutzerschnittstelle (engl.: *GUI - **G**raphical **U**ser **I**nterface*) betrachtet, so sind möglicherweise eindeutige funktionale Gemeinsamkeiten zu erkennen. Während sich die beiden Produkte Funktionen zur Entwicklung und dem Betrieb der GUI teilen, können sie auch weitere Funktionen enthalten, die einer anderen Programmfamilie angehören. Dadurch ist klar, dass eine Produktlinie vielen verschiedenen Programmfamilien angehören kann, welche wiederum mehreren Produktlinien zugeordnet sein können.

Abbildung 2.1 verdeutlicht die Unterschiede zwischen Softwarefamilien und Produktlinien anhand von Produktbeispielen der Firma Microsoft. Es werden 3 Produktlinien (Zielmärkte: Standard, Small Business, Professional) und 4 unterschiedliche Programmfamilien (Anwendungsgebiete: Money, Office, Project, WinServer) dargestellt. Hier ist deutlich zu erkennen, dass eine einzelne Produktlinie mehreren Programmfamilien angehören kann und umgekehrt Mitglieder einer Programmfamilie unterschiedlichen Produktlinien. Aus der Darstellung geht hervor, dass die beiden Begriffe unabhängig voneinander betrachtet werden können. In der Abbildung ist beispielsweise zu erkennen, dass jede Produktlinie jeweils nur ein Mitglied einer einzigen Programmfamilie beinhaltet. Das bedeutet, dass die Vorteile der Wiederverwendung von Funktionen, im Sinne von Programmfamilien, nicht greifen, da es diesbezüglich keine Gemeinsamkeiten gibt.

Ziel des Ansatzes der Programmfamilien ist es die gemeinsam genutzten Funktionen konsequent auf Wiederverwendbarkeit aus zulegen. Es ist ein System anzustreben, in dem die Funktionen sich gegenseitig benutzen [Spi02]. Das Fazit der vorhergehenden Betrachtungen ist die Erkenntnis, dass Programmfamilien für die modularisierte Entwicklung von Funktionen des Transaktions-Managements eine viel versprechende Methode darstellen. Wie bei anderen Formen der Software-Entwicklung auch, ist es sinnvoll

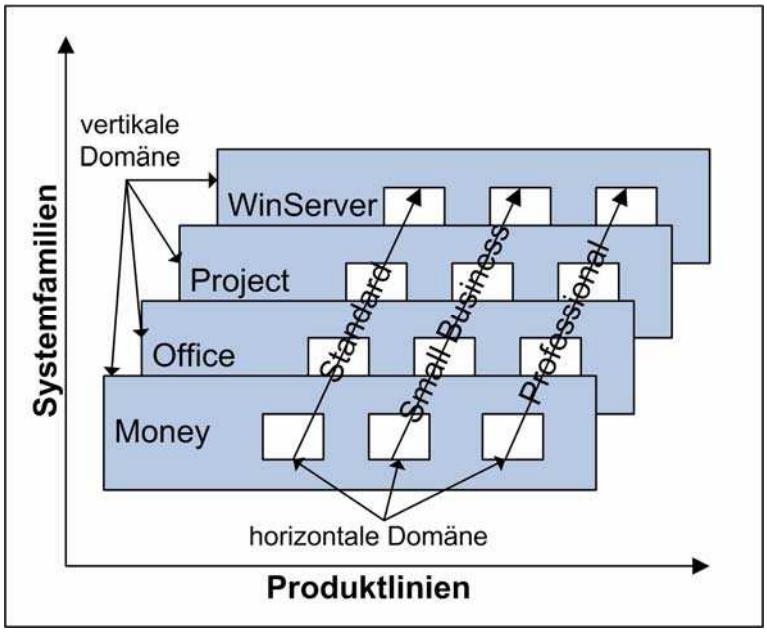

Abbildung 2.1: Einordnung Softwarefamilien / Softwareproduktlinien nach [Fra04]

den Entwicklungsprozess einer Programmfamilie in geeigneter Weise zu strukturieren. Diese Funktion leistet das *Domain Engineering*, welches im folgenden Abschnitt näher erläutert wird.

2.2 Domain Engineering

Das *Domain Engineering* (übersetzt: Domänenentwicklung) stellt die strukturelle Basis für die Entwicklung von Programmfamilien dar. Es kann gleichzeitig als Leitfaden und grundlegendes Werkzeug der Software-Entwicklung in diesem Bereich angesehen werden. Kang et al. [Kan90] beschreiben den Begriff der *Domäne* als eine Menge von bestehenden und zukünftigen Produkten die Gemeinsamkeiten besitzen. Czarnecki und Eisenecker [Cza00] heben die zwei verschiedenen Sichtweisen auf den Begriff der Domäne hervor. In Abbildung 2.1 wird dies deutlich. Die Grenzen der *horizontalen* Domäne einer Programmfamilie definieren sich unter anderem über die gemeinsam verwendbaren Funktionen. Während die *vertikale* Domäne teilweise durch die Eigenschaften des Zielmarkts beschrieben wird.

Der Domänenbegriff beschreibt den Wissens- oder Anwendungsbereich, der von den involvierten Interessenvertretern (so genannte *Stakeholder*) festgelegt und durch folgende Punkte charakterisiert wird [Cza00]:

- beinhaltet eine Sammlung an Konzepten und Begrifflichkeiten, die von Anwendern des Fachgebiets interpretiert werden können

- Eingrenzen der Sichtweise bezüglich optimaler Abdeckung der Anforderungen, welche durch die Interessenvertreter entwickelt werden
- Know-How für die komponentenbasierte Systementwicklung

Eine umfassende Definition des Domain Engineering wird von Czarnecki und Eisenecker [Cza00] angegeben:

> ″Domain Engineering *is the activity of collecting, organizing and storing past experiences in building systems or parts of systems in a particular domain in the form of reuseable assets (i.e., reusable work products), as well as providing an adequate means for reusing these assets (i.e., retrieval, qualification, dissemination, adaption, assembly, and so on) when building new systems.*″

In einem Artikel des Carnegie Mellon Software Engineering Institute [Sof96] wird das Domain Engineering als der Prozess der Analyse, Spezifikation und Implementierung von Software-Produkten einer Domäne, welche in verschiedenen Programmen zum Einsatz kommt, bezeichnet. Die Domänenentwicklung zielt also auf ein Höchstmaß an Wiederverwendung von Programmartefakten vorhandener Systeme in neuen Softwareprodukten hin. Neben den in der vorhergehenden Sektion genannten Vorteilen ergeben sich auch Vorteile in der Software-Wartung. Innerhalb einer Programmfamilie müssen die Änderungen nicht in jedem Mitglied bewerkstelligt werden. Dies geschieht für alle Mitglieder gleichzeitig auf der Ebene der Familie. Auch wenn die Domänenentwicklung immer in Verbindung mit der Software-Entwicklung behandelt wird, muss erwähnt werden, dass ausserhalb dieses Themengebiets Anwendungsbereiche existieren, in denen dieser Ansatz genutzt wird.

Wie Abbildung 2.2 zu entnehmen ist, verläuft neben der Domänenentwicklung noch ein zweiter Prozess. Dabei handelt es sich um die Phasen der Anwendungsentwicklung (engl.: *Software-Engineering*), beginnend mit der Anforderungsanalyse entsprechend des Software-Lebenszyklus wie er beispielsweise von Dumke [Dum03] beschrieben wird, über die Produktkonfiguration auf Basis der Resultate des Domain Engineering bis hin zu den klassischen Phasen der Integration und Erprobung. Aufbauend auf den Ergebnissen der einzelnen Phasen des Domain Engineering wird mit der Anwendungsentwicklung begonnen. Es existiert jedoch eine Rückkopplung zwischen den Prozessen. Dies bedeutet, dass sich Änderungen innerhalb eines Prozesses auf den benachbarten Prozess niederschlagen und umgekehrt.

In den folgenden Abschnitten werden die drei Phasen der Domänenentwicklung erläutert. Obwohl die Einteilung der Domänenentwicklung in einzelne Phasen unpräzise ist, da mit Phasen eine zeitliche Entwicklung assoziiert wird. Die drei Abschnitte der Domänenentwicklung können, unabhängig von einer durch die Zeit vorgegebenen Reihenfolge, abgearbeitet und bei Bedarf ″angesprungen″ werden. Es handelt sich also vielmehr um einen kontinuierlichen Prozess, in dem die Ergebnisse durch ständiges Nachbessern und Überdenken verfeinert werden.

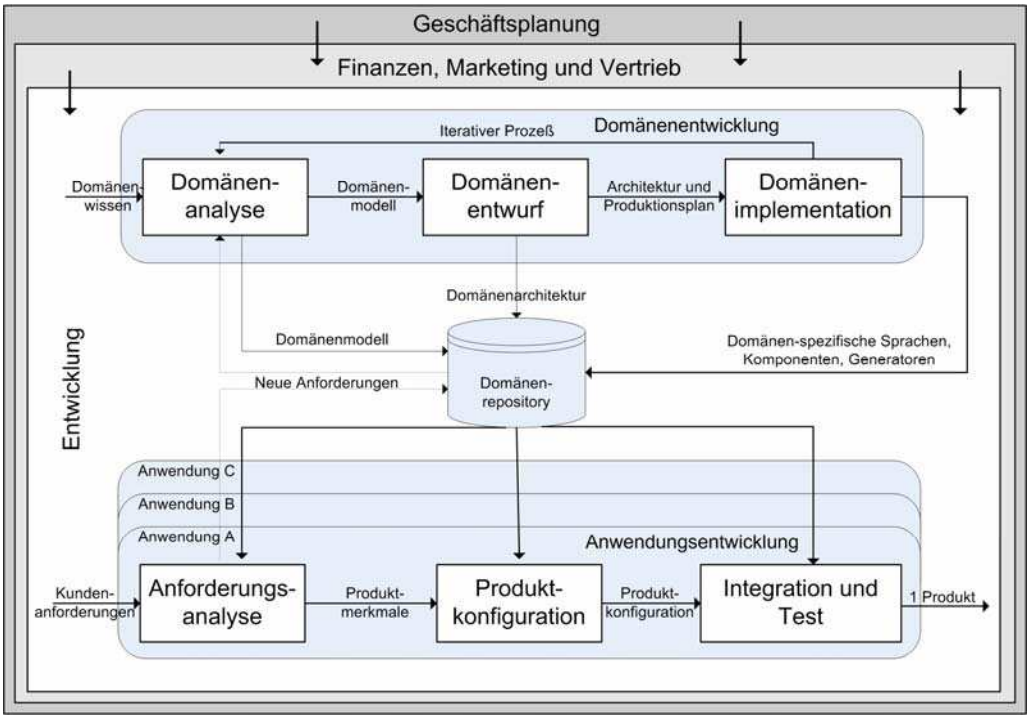

Abbildung 2.2: Domänenentwicklung im Zusammenspiel mit der Anwendungsentwicklung nach [Fra04]

2.2.1 Domain Analysis

Die *Domain Analysis* (übersetzt: Domänenanalyse) ist die initiale Phase der Domänenentwicklung. Aus der Bezeichnung leitet sich die Aufgabe dieser Phase ab. Hier erfolgt die Eigenschaftenextrahierung für die gesuchte Domäne. Zu diesem Zweck muss die Domäne eingegrenzt werden. Zunächst müssen die involvierten *Stakeholder* identifiziert und bewertet werden. Im Anschluss werden die Forderungen der beteiligten Parteien aufgegriffen und darauf aufbauend die Auswahl und Definition der Domäne verabschiedet. Wichtige Formen sind: *funktionale*, *marktspezifische*, *qualitative*, *design-spezifische*, *konzeptuelle* Anforderungen in Addition zum *Anwendungsbereich*. Dabei ist zu beachten, dass die einzelnen Parteien ihre ganz eigenen Schwerpunkte bezüglich der Anforderungen setzen. Umso wichtiger ist es, eine geeignete Gewichtung der verschiedenen Forderungen zu beschliessen, und damit eine Domäne zu definieren, die die entscheidenden Kriterien möglichst genau abbildet.

Die Domänenanalyse wird oft als anwendungsübergreifende Anforderungsanalyse bezeichnet. Da in einer Domäne unterschiedliche Systeme angesiedelt sein können, die ausser den domänenspezifischen Eigenschaften nichts gemeinsam haben, ist es durchaus sinnvoll, die einzelnen Systeme fachübergreifend zu analysieren. Während der Analyse wird das existierende bzw. aufgezeichnete Wissen einer Domäne, mit dem Ziel der Wiederverwendung bei der Erstellung neuer Systeme, systematisch identifiziert, erfasst

und organisiert [Ara91]. Hier kommt es insbesondere darauf an, dieses Wissen durch das Beschreiten neuer, kreativer Wege zu erweitern [Cza00].

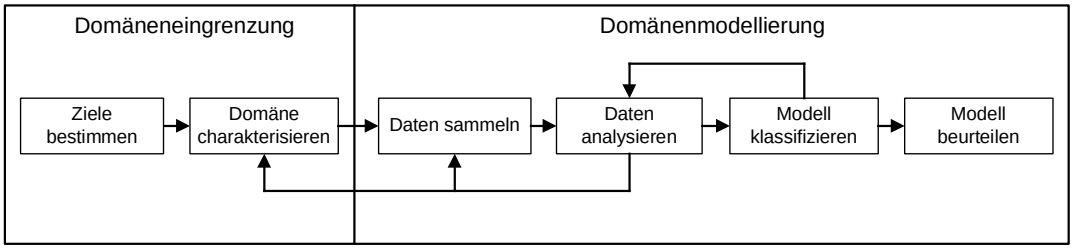

Abbildung 2.3: Allgemeiner Ablauf der Domänenanallyse nach [Fra04]

Das erworbene Wissen fliesst in das *Domänenmodell* ein, welches die Gemeinsamkeiten und Unterschiede der beteiligten Systeme repräsentiert. Es stellt eine höhere Abstraktionsebene des Wissens dar und ist somit plattformunabhängig. In Abbildung 2.3 ist der allgemeine Ablauf der Domänenanalyse skizziert.

Nach Czarnecki und Eisenecker [Cza00] sind in dem Domänenmodell folgende Elemente zusammengefasst. Zunächst wird die schon beschriebene *Domänendefinition* eingebunden. Sie gibt gewissermaßen die Richtung für die weiteren Schritte vor. Im so genannten *Domänen-Lexikon* (engl.: Domain Lexicon) wird das domänenspezifische Vokabular organisiert. Weitere Elemente stellen Modelle dar, die wichtige *konzeptuelle* Eigenschaften der Domäne beschreiben. Dies können z.B. Fluss-, Objekt-, Interaktions- oder Zustandsdiagramme sein. Besonders wichtige Bestandteile des Domänenmodells sind die *Merkmalsmodelle* (engl.: Feature Model). Die Entwicklung und Verwendung derartiger Modelle ermöglicht es, die einzelnen für die Wiederverwendung bestimmten Komponenten zu kennzeichnen und sie bezüglich ihrer Funktionalität, zu ordnen. In diesem Zusammenhang ist es wichtig, verschiedene Regeln für die (Re-)kombination der Komponenten zu erstellen, um zu verhindern, dass eine unsinnige Systemkonfiguration abgeleitet wird. Den Techniken zur Merkmalsmodellierung sind im Folgenden weiterführende Abschnitte gewidmet. Laut Arango [Ara94] folgen noch die *taxonomische Klassifizierung* des Domänenmodells (Generalisierung und Abstraktion der domänenrelevanten Aspekte) und dessen *Beurteilung*. Damit sind die Grundlagen für den *Domänenentwurf* gelegt.

2.2.2 Domain Design

Im Bereich der Software-Entwicklung stellt die Phase des Entwurfes den Teil der Arbeit dar, in dem festgelegt wird *wie* die spezifizierten Ergebnisse der Problemdefinition und Anforderungsanalyse konkret umgesetzt werden. Der Domänenentwurf behandelt die Frage nach der Art der Umsetzung. Es muss geklärt werden, welche Struktur die Programmfamilie haben sollte, um die Entwicklung konkreter Anwendungen, gerade in Hinblick auf Wiederverwendbarkeit, optimal zu unterstützen. Im Rahmen des Domain Engineering werden solche Strukturen auch als *Software-Architektur* bezeichnet (genauer: DSSA - **D**omain **S**pecific **S**oftware **A**rchitecture).

Shaw und Garlan [Sha96] geben eine Definition für den Begriff der Software-Architektur:

> *"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 these components. Such a system may in turn be used as a (composite) element in a larger system design."*

Auch Buschmann et al. [Bus96] kennzeichnen die Software-Architektur als ein beschreibendes Stilmittel, dass die Beziehungen der Komponenten und Teilsysteme zu einander zum Ausdruck bringt. Ergänzend wird jedoch angeführt, dass die einzelnen Elemente aus verschiedenen Blickwinkeln betrachtet werden. Durch diese Methode werden die relevanten funktionalen und nichtfunktionalen Eigenschaften des Software-Systems aufgezeigt.

Die Entwicklung und Dokumentation einer geeigneten Software-Architektur wird von Nilson et al. [Nil94] als Schlüssel für den Entwurf von Systemen, die auf Wiederverwendbarkeit ausgelegt sind, angesehen. Die Architektur sollte so allgemein gehalten sein, dass sie die meisten Familienmitglieder im Prozess ihrer Entwicklung unterstützt. Die Entscheidung für eine konkrete Form der Architektur sollte so früh wie möglich fallen, da alle folgenden Festlegungen daran ausgerichtet werden.

Das Design der Software-Architektur kann durch verschiedene Methoden systematisiert werden. Das Carnegie Mellon SEI [Sof] erläutert das *Attribute-Driven-Design* (kurz: ADD) als eine rekursive Methode der Dekomposition für den Entwurf einer Software-Architektur (auch [Bas01]). Zentrale Grössen der Methode sind die so genannten architekturbeeinflussenden Merkmale (engl.: Architectural Drivers). Als architekturbestimmend werden Merkmale eingestuft, die in Bezug auf Qualtität, Funktionalität und Wirtschaftlichkeit besondere Beachtung verdienen. Die Identifizierung relevanter Merkmale geschieht auf Basis der Ergebnisse der Domänenanalyse (insbesondere der dort entwickelten Merkmalsmodelle). Aufbauend auf dem Resultat des grundlegenden Architektur-Entwurfs muss darauf hin festgelegt werden, wie die einzelnen Merkmale im Hinblick auf die Modularisierung zur Wiederverwendung am besten aufbereitet werden können. Dabei spielt die Klärung der verwendeten Strategie zur Komponentenerzeugung eine wesentliche Rolle. Da es an dieser Stelle nicht um eine einzelne Anwendungsentwicklung geht, sondern ein Prozess beschlossen werden soll, der die Entwicklung ganzer Programmfamilien ermöglicht, ist es von grosser Bedeutung festzulegen, wie die einzelnen Komponenten zu den verschiedenen Anwendungen kombiniert werden können. Diese Festlegungen müssen in geeigneter Form dokumentiert werden. Dabei sind sowohl nutzlose, als auch komplett unsinnige Kombinationen auszuschliessen. Die erforderlichen Techniken, um die genannten Probleme und Forderungen zu bewältigen, werden in nachfolgenden Abschnitten näher erläutert.

2.2.3 Domain Implementation

In den Abschnitten der Domänenanalyse und des Domänenentwurfs werden die strukturellen und theorethischen Grundlagen für die konkrete Umsetzung einer Programmfamilie gelegt. Die *Domänenimplementierung* stellt die Realisierung dieser Grundlagen dar.

Laforme und Stropky [Laf95] geben für den Implementierungsprozess folgende Beschreibung:

> ″Domain Implementation *is the DE activity that realizes the reuse opportunities identified during domain analysis und design in the form of common requirements and design solutions, respectively. It includes the identification, re-engineering and/or construction, and maintenance of reusable assets that support those common requirements and design solutions. It faciliates the integration of those reusable assets into a particular application.*″

Es werden demnach alle für die Wiederverwendung gekennzeichneten Komponenten implementiert. Entscheidend für den Erfolg einer Programmfamilie ist die Zuverlässigkeit der Komponenten. Aus diesem Grund wird die Durchführung komplexer Software-Tests empfohlen. Hierbei kommt es insbesondere auf die Zuverlässigkeit im gegenseitigen Wechselspiel an, da gerade die flexible Gestaltung das Hauptmerkmal dieser Entwicklung darstellt. Während der Implementierung ist, analog dem herkömmlichen Entwicklungsprozess der Software-Entwicklung, darauf zu achten, dass entstandene Probleme adäquat behandelt und gelöst werden. Das erfordert in manchen Fällen, Umstellungen in den zuvor erarbeiteten Ergebnissen der Analyse und/oder des Entwurfs, worauf hin der Domänenentwicklungsprozess unter den geänderten Parametern nochmals durchlaufen wird. Der Einstiegspunkt wird durch die Art der neuen Parameter festgelegt.

Zusammengefasst werden die Ergebnisse der Analyse, des Entwurfes und der Implementierung in der *Domain Reuse Library* [Laf95] (siehe auch: Abb. 2.2 *Domain Repository*). Sie stellt eine Art Speicher und gleichzeitig auch Schnittstelle dar, durch die den einzelnen Prozessen der Software-Entwicklung der Zugriff auf die Ergebnisse der Domänenentwicklung ermöglicht wird.

2.2.4 Application Engineering

Das *Application Engineering* wird als der Prozess bezeichnet, in dem unter Verwendung der Ergebnisse des Domain Engineering die gewünschten Anwendungen im Rahmen der Programmfamilie entwickelt werden. Domänen- und Anwendungsentwicklung sind zwei parallele Prozesse, die sich gegenseitig bedingen und in Interaktion stehen. Gemäss Abbildung 2.2 wird zunächst die Anforderungsanalyse durchgeführt. In dieser Phase werden die Anforderungen (im Dialog mit dem Auftraggeber), denen das System gerecht werden muss, ermittelt. Dabei ist es wichtig, dass die Entwickler während der Verhandlung ein für

den Kunden verständliches, fachspezifisches Vokabular benutzen, um Missverständnisse zu verhindern und die Forderungen so genau wie möglich festlegen zu können. Als Grundlage für die Analyse dient das entwickelte Domänenmodell. Aus diesem Modell wählen Kunde und Entwickler die für das Programm nötigen Merkmale (verwendet: Merkmalsmodell) aus. Wichtig in diesem Zusammenhang ist die Sicherstellung einer funktionierenden Interaktion der Merkmale untereinander. Dabei müssen durch die Merkmalsauswahl entstandene, zusätzliche Anforderungen berücksichtigt werden. Weiterhin ist bezüglich der Anforderungen und in Hinblick auf eine möglichst ressourcen-schonende Anwendung, ein vollständiges und minimales System anzustreben. Dies erfordert den Verzicht auf überflüssige Funktionen.

Für den Fall, dass der Auftraggeber eine Funktion wünscht, die nicht im Domänenmodell beschrieben ist, muss der Kundenwunsch in geeigneter Form behandelt werden. Dafür existieren zwei allgemeine Lösungsansätze, die jeweils unterschiedliche Auswirkungen auf der Ebene der Domänen- und Anwendungsentwicklung haben. Eine Möglichkeit wäre die einmalige Implementierung und Integration der Eigenschaft in die Kundenanwendung. Dabei kann auf eine Modifikation innerhalb des Domain Engineering verzichtet werden, da es sich um den klassischen Fall einer Entwicklung im Sinne des Software Engineering handelt und allenfalls dafür gesorgt werden muss, dass eine geeignete Schnittstelle zu den Komponenten der Programmfamilie bereitgestellt wird. Für den Fall, dass die gewünschte Funktion eine sinnvolle Erweiterung des Programmfamilien-Funktionsraumes darstellt, sollte sie, gemäss Domänenentwicklung, der Wiederverwendbarkeit zugeführt werden.

Phase	Produkt - Design	Detailiertes Design	Implementierung	Integration
Aufwand	16%	25%	40%	19%

Abbildung 2.4: Anteiliger Programmentwicklungsaufwand nach *COCOMO* (***Co****nstructive* ***Co****st* ***Mo****dell*) [Boe81]

Um die richtige Entscheidung für eine der Methoden treffen zu können, ist es erforderlich die Kosten abzuschätzen, die diese hervorrufen. Abbildung 2.4 verdeutlicht die Notwendigkeit der Kostenabschätzung. Der Vorteil einer einmaligen Nutzung besteht darin, dass auf die Funktionsintegration in das Konzept der Programmfamilie verzichtet werden kann. Wird jedoch eine Wiederverwendung angestrebt, ist es unsinnig, eine erneute Entwicklung der Komponente für jede einzelne Anwendung zu bemühen.

An die Phase der Anforderungsanalyse schliesst sich die *Konfiguration* an. Dabei handelt es sich um einen Prozess, der dafür verantwortlich ist, dass die verfügbaren Komponenten der Programmfamilie zur angestrebten Anwendung kombiniert werden. Die zugrunde liegende Implementierungstechnik wird als *generative Programmierung* bezeichnet. Eine Programmiersprache, die derartige Formen der Programmierung unterstützt, sollte über Mechanismen verfugen, die eine unsinnige oder sogar unzulässige Konfiguration verhindern. Daher ist bei der Konfiguration schon im Vorfeld darauf zu achten, dass die gewählten Komponenten in die End-Anwendung passen und sich nicht gegenseitig be-

hindern. Sollte es mehrere Komponenten geben, die der gleichen Funktion angehören, so ist die ″beste″ Lösung zu wählen. Es ist sinnvoll schon vor Beginn des Konfigurationsprozesses adäquate Auswahlkriterien festzulegen. Wie die Anforderungsanalyse zuvor, baut auch die Konfiguration auf den Ergebnissen der Domänenentwicklung auf. Grundlage hierfür sind, unter anderem, die Merkmalsmodelle und *Varianten-Management-Systeme* (Bedeutung: Regelung der Kombinationsmöglichkeiten von Komponenten und Steuerung der Lösungsauswahl).

Ergänzend zu den Komponenten der Programmfamilie müssen die kundenspezifischen Funktionalitäten entwickelt und integriert werden. Hier gilt es, genau wie bei den Bausteinen der Programmfamilie, die Kombatibilität zu gewährleisten. Eine weit verbreitete Methode der Systemimplementierung stellt das *Prototyping* dar. Hier werden dem Prototypen, beginnend mit den Basiseigenschaften, Schritt für Schritt die geforderten Funktionen hinzugefügt. Dabei sollten die ergänzten Programmteile bezüglich ihrer Funktion eingehend getestet werden. Im Fehlerfall ist dann eine effektive Eingrenzung der Ursache möglich.

Damit gewährleistet werden kann, dass die entwickelte Anwendung die zuvor erarbeiteten Anforderungen erfüllt, ist es notwendig diese Eigenschaften unter Verwendung geeigneter Testroutinen zu validieren. Weiterhin müssen Qualitätskriterien (z.B. Laufzeiteigenschaften, Benutzerfreundlichkeit, Systemkompatibilität, Ressourcenverbrauch, etc.) definiert werden, die eine Abschätzung der Marktreife des entwickelten Produkts ermöglichen. Nicht zuletzt ist darauf zu achten, dass die Anwendung als Gesamtsystem fehlerfrei funktioniert. Um die Gefahr der Schädigung von Systemen der Anwendungsumgebung, zu minimieren, wird es in einer Testumgebung auf korrekte Funktion überprüft. Gleichzeitig werden erste Anwendungserfahrungen gesammelt, die dem Kunden bereitgestellt werden können. Erst wenn sichergestellt ist, dass alle Gütekriterien erfüllt sind, kann das Programm an den Kunden ausgeliefert werden. Dort erfolgt die Eingliederung der Anwendung in die dafür bestimmte Umgebung. Sollten während der Programmanwendung, trotz aller Tests, Fehler auftreten, so muss eine domänengestützte Überarbeitung auch zu diesem Zeitpunkt möglich sein. Die entwickelte Programmfamilie muss dafür die nötigen Mechanismen bereitstellen.

2.2.5 Techniken und Methoden des Domain Engineering

Die Eigenschaft der Wiederverwendbarkeit steht beim Domain Engineering im Mittelpunkt der Betrachtung. Die Wiederverwendung erstreckt sich hierbei nicht ausschliesslich über Software-Komponenten, sondern wird auch bezüglich strukturgebender Daten, wie z.B. der Software-Architekur, angestrebt. Weiterhin bietet sich eine Wiederverwendung von Erfahrungswerten aus vergangenen Projekten der gleichen Programmfamilie in neuen Anwendungen an. Dies betrifft unter anderem Ergebnisse aus Anforderungsanalysen früherer Entwicklungen. Es ist ersichtlich, dass es sich um ein sehr vielschichtiges Thema handelt, welchem sich, unter Anwendung verschiedenster Methoden, genähert werden kann. Zentrale Konzepte und Ideen werden an den Beispielen *FODA* und *ODM*

erläutert. Auf Ausführungen zu anderen Methoden wird an dieser Stelle verzichtet, da diese in vielen Punkten deckungsgleich mit den beschriebenen Technologien sind.

Feature-Oriented Domain Analysis - FODA

Die Analyse und Beschreibung auf Basis von Merkmalen ist eine gängige Methode, um die bestimmenden Eigenschaften einer Domäne zielsicher zu identifizieren. Die *merkmalsorientierte Domänenanalyse* (engl.: **F**eature-**O**riented **D**omain **A**nalysis) ist eine Entwicklung des Software Engineering Institute (kurz: SEI) der Carnegie Mellon Universität. Die Technologie ermöglicht die Durchführung einer strukturierten Domänenanalyse.

Kang et al. [Kan90] beschreiben drei grundsätzliche Phasen der Merkmalsorientierten Domänenanalyse (siehe Abbildung 2.5):

- Kontextanalyse
- Domänenmodellierung
- Architekturmodellierung

In der *Kontextanalyse* werden die Grenzen der Domäne und ihre Beziehungen zu externen Elementen untersucht und definiert. Das zentrale Resultat der Analyse ist das Kontextmodell. Es beinhaltet das Struktur- und Kontextdiagramm. Beim Strukturdiagramm handelt es sich um ein Blockdiagramm, das die Domäne in Beziehung setzt zu über- und untergeordneten Domänen. Im Kontextdiagramm werden die Datenströme, in Bezug auf eine verallgemeinerte Anwendung der Programmfamilie, über die Domänengrenzen hinweg, dargestellt [Kru93].

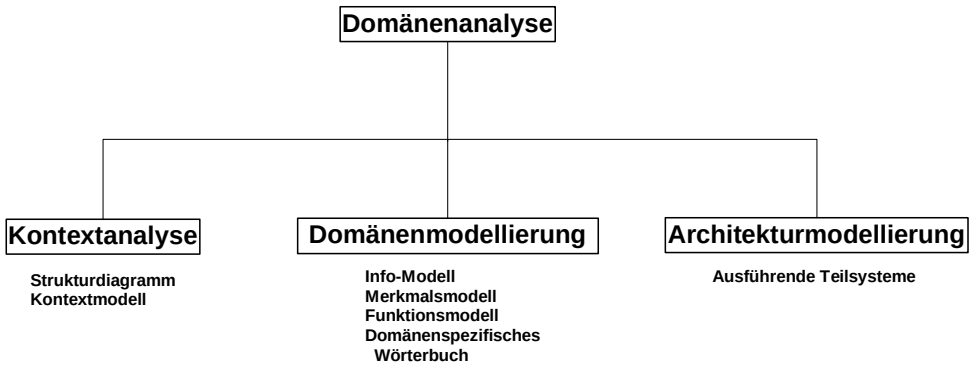

Abbildung 2.5: Phasen der merkmalsorientierten Domänenanalyse [Kru93]

Die Domänenmodellierung hat die Aufgabe, Gemeinsamkeiten und Unterschiede der potentiellen Familienmitglieder zu markieren. Sie setzt sich aus drei Abschnitten zusammen. Bei der *Informationsanalyse* geht es hauptsächlich um das Zusammentragen des

existierenden, domänenspezifischen Fachwissens. Durch dieses Wissen wird das Verständnis für die Problemstellung entwickelt. Dabei können die Beziehungen der einzelnen Elemente durch verschiedene Modelle dargestellt werden (z.B. ER-Modelle). In der *Merkmalsanalyse* wird das Merkmalsmodell entwickelt. Die Herangehensweise und Notation bei der Merkmalsmodellierung wird nachfolgend erläutert. Daten- und Kontrollflüsse innerhalb der Domäne werden während der *Funktionsanalyse* untersucht. Erste Ansätze zum Entwurf einer für die Domäne geeigneten Software-Architektur werden während der *Architekturmodellierung* generiert. Ergänzend erfolgt die Modellierung allgemeiner und grundsätzlicher Eigenschaften der Architektur. Die Phase der Architekturmodellierung fällt, genau genommen, in den Bereich des Domänenentwurfs. Dadurch wird klar das FODA den fliessenden Übergang zwischen Domänenanalyse und -entwurf ermöglicht.

Zusammengefasst ist FODA eine Methode, die die Wiederverwendung von Architekturen und Design-Elementen unterstützt [Fer99]. Es werden domänenspezifische Merkmale gekennzeichnet, die in späteren Prozessen der Anwendungsentwicklung, auch mit Beteiligung der Stakeholder, genutzt werden können.

Feature-Modellierung

Zentraler Aspekt der merkmalsorientierten Domänenentwicklung ist die Entwicklung des Merkmalsmodells. Dieses Modell beinhaltet Merkmalsdiagramme, in denen die Merkmale in graphischer Form aufbereitet werden. Als zugrunde liegende Darstellungsform wird in den meisten Fällen eine azyklische Baumstruktur gewählt. Dies ermöglicht eine semantisch reichhaltige, hierarchische Anordnung der Elemente. Neben den Merkmalsdiagrammen werden im Merkmalsmodell noch ergänzende Informationen zu den einzelnen Merkmalen festgehalten. Dabei kann es sich um Angaben zu bekannten Einschränkungen, Verfügbarkeit, Beziehungen oder aber potentiellen Interessenten handeln [Cza00]. Die Darstellung in Form einer azyklischen Baumstruktur kann als Graph aufgefasst werden, wobei die Features durch Knoten repräsentiert werden, und die Darstellung der Beziehungen (Baum: Vater-Kind-Beziehung) zwischen den Merkmalen durch die Kanten erfolgt.

Abbildung 2.6 veranschaulicht an einem kleinen Beispiel die gängigsten Notationen von Merkmalsdiagrammen. Das Konzept *Auto* bildet die Wurzel des Baumes und ist damit allen Merkmalen übergeordnet. Es setzt sich aus diesen Merkmalen zusammen. Ein Auto besitzt grundsätzlich eine *Karosserie* und einen *Antrieb*. Die zwingende Notwendigkeit des Vorhandenseins dieser beiden Merkmale wird durch einen ausgefüllten Punkt gekennzeichnet. Der Antrieb wiederum kann sich optional aus den Merkmalen *Elektro* und *Verbrennungsmotor* zusammensetzen. Dabei signalisiert der gefüllte Kreisausschnitt, dass *mindestens* eines der Merkmale ausgewählt werden muss. Soll das Auto einen Hybrid-Antrieb bekommen, so können auch beide Merkmale gewählt werden. Die Kreisausschnitte veranschaulichen also eine Gruppierung der involvierten Merkmale. Bei der Entscheidung nach der Art des Verbrennungsmotors kann zwischen *Diesel* und *Otto-Motor* entschieden werden. Jedoch darf nur eine Variante gewählt werden. Die beiden Merkmale schliessen sich gegenseitig aus. Dies entspricht in der Logik einem Exklusiv-

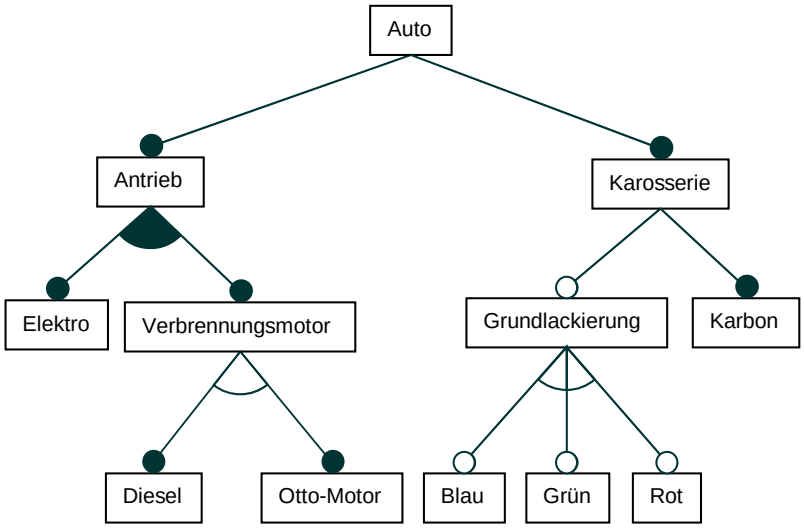

Abbildung 2.6: Einfaches Merkmalsdiagramm - AUTO

Oder und wird im Merkmalsdiagramm durch einen transparenten Kreisausschnitt zwischen den beiden Merkmalen markiert. Die Karosserie besteht grundsätzlich aus *Karbon*. Eine Lackierung zur Verhinderung von Korrosion ist also nicht zwingend erforderlich. Die *Grundlackierung* kann optional erfolgen (Ausdruck durch leeren Kreis). Darüber hinaus kann eine farbgebende Lackierung optional erfolgen, wobei sich die einzelnen Merkmale erneut ausschliessen. Weitere Möglichkeiten wären Gruppierungen von optionalen und zwingend erforderlichen Merkmalen. Czarnecki und Eisenecker [Cza00] führen jedoch aus, dass alle möglichen Varianten auf die dargestellten Basis-Konstrukte zurückzuführen sind.

Merkmalsdiagramme bieten den Vorteil, dass eine komplexe Problemstellung in viele, kleine Einzelteile (Merkmale) zerlegt werden kann. Ist die Problemstellung so umfangreich, dass eine Darstellung in einem einzigen Merkmalsdiagramm zu unübersichtlich wird, so kann sie in Teilkomplexe aufgeteilt werden, die jeweils ein eigenes Merkmalsdiagramm besitzen. Obwohl Merkmalsdiagramme sehr aussagekräftig sind, können sie nicht alle Beziehungen zwischen den Merkmalen beschreiben. Beispielsweise können Merkmale der Karosserie unabhängig von den Antriebseigenschaften aus dem Merkmalsdiagramm gewählt werden. Bei näherer Betrachtung fällt jedoch auf, dass es für die Konstruktion der Karosserie von grosser Bedeutung ist, ob ein Hybrid-Antrieb verbaut wird oder nicht. Im ersten Fall muss die Karosserie jeweils eine Aufhängung für jeden Antrieb bereitstellen. Solche überkreuzenden Beziehungen müssen in geeigneter Form behandelt werden. In nachfolgenden Abschnitten wird diese Problemstellung genauer beleuchtet.

Organization Domain Modeling - ODM

Organization Domain Modeling (kurz: ODM) ist eine weitere Methode zur Unterstützung der Domänenentwicklung. Sie wurde ursprünglich, unter anderem von Mark Simos, ent-

wickelt und in verschiedenen Projekten, wie z.B. dem STARS-Projekt, verfeinert und verbessert. Ähnlich zu FODA bietet sie formale, komfortabel steuerbare und reproduzierbare Mechanismen als Grundlage für eine effektive Domänenentwicklung [Kea97].

Ziel ist es brauchbare Artefakte, existierender Systeme, für die Wiederverwendung aufzubereiten. Dies können verschiedenste Elemente, wie: Quellcode, design- und anforderungsspezifische Informationen oder Testdaten, sein. Diese Elemente werden in kleinen Informationseinheiten (engl.: *Assets*) organisiert. Anders als bei FODA werden nicht nur Merkmale aufgearbeitet, die für den Endnutzer interpretierbar sind. Vielmehr werden *alle* Merkmale, die für die beteiligten Stakeholder von Interesse sind, berücksichtigt. Grundsätzlich kann jedes Produkt einer (früheren) Entwicklung, in Form eines Assets, der Wiederverwendung zugeführt werden.

Die Verwendung von ODM bietet sich insbesondere für die Entwicklung von Domänen an, die folgende Kriterien erfüllen [STA96]:

- Ausgereiftheit (Verfügbarkeit grosser Mengen von wieder verwendbaren Elementen aus Altsystemen)
- Stabilität (kein Veralten der Domäne aufgrund technologischer Veränderungen)
- Wirtschaftlichkeit (grosse Nachfrage nach Entwicklungen dieser Domäne)

Kern des Organization Domain Modeling ist das *ODM-Prozessmodell*. Es beinhaltet drei Hauptphasen, in denen der ODM-Lebenszyklus verläuft. Die *Planungsphase* umfasst das Formulieren von Zielen, die innerhalb der Domäne erreicht werden sollen. Dabei ist eine geeignete Domäne, bezüglich der Bedürfnisse der Organisation zu wählen. Um diesen Bedürfnissen gerecht zu werden, kommt der Stakeholder-Definition eine besondere Bedeutung zu. Diese Phase entspricht in etwa der Kontextanalyse in FODA [Cza00]. Ziel der *Modellierungsphase* ist die Erzeugung eines geeigneten Domänenmodells, basierend auf den Ergebnissen der Planungsphase. Das Modell inkludiert Beschreibungen und Informationen der verschiedenen Teilsysteme innerhalb der Domäne in Form von Merkmalen. Hier werden ebenfalls die Beziehungen und Abgrenzungen der Merkmale genauestens beschrieben und festgehalten. Dies ist notwendig, um die Qualität der herausgearbeiteten Merkmale zu erhöhen. In der *Engineering-Phase* werden unter anderem die Architektur und Implementierung der einzelnen Assets bewältigt. Dabei müssen, in Hinblick auf die Wirtschaftlichkeit der Domänenentwicklung möglichst Asset-Basen[1] gewählt werden, denen eine hohe Nachfrage am Markt bescheinigt werden kann.

Bei ODM handelt es sich um ein Konzept, das nicht an bestimmte Methoden oder Technologien angelehnt ist. Es beschreibt vielmehr eine allgemeine Vorgehensweise auf höherer Abstraktionsebene, bei der die verwendeten Hilfsmittel nach Bedarf gewählt werden können. Dadurch ist ein Höchstmaß an Verträglichkeit zwischen dieser Form der Modellierung und weiterführenden Technologien, wie etwa objektorientierten Ansätzen, gewährleistet.

[1] Asset-Basen sind Ansammlungen von domänenspezifischen Informationseinheiten. Sie beinhalten neben den Informationseinheiten auch Informationen zur Organisation dieser Einheiten.

2.3 Programmiertechniken

Es existieren zwei grundlegende Programmierparadigmen, auf die sich heutige Technologien zurückführen lassen. In der *imperativen Programmierung* (auch: *prozedurale Programmierung*) werden Programme als Folge von Befehlen angesehen, die durch den Rechner in einer definierten Reihenfolge abgearbeitet werden. Die zu berechnenden Werte werden in Variablen gespeichert. Die Werte der Variablen können sich zur Laufzeit ändern. Wird in der imperativen Programmierung der Lösungsweg beschrieben, so ist die Herangehensweise bei der *deklarativen* Programmierung von anderer Art. Hier wird die Form der gewünschten Lösung definiert. Der Rechner muss von dieser Definition ausgehend den richtigen Lösungsweg schlussfolgern und die geforderten Ergebnisse errechnen. Wichtige Ausprägungen stellen die *funktionale Programmierung* (Idee: Berechnungen als Auswertung mathematischer Funktionen) und die *Logikprogrammierung* (Idee: Lösungsberechnung auf Basis vorgegebener Axiome) dar.

Grundlage der nachfolgend vorgestellten Technologien, wie *merkmals-* und *aspektorientierte Programmierung*, bildet die *objektorientierte Programmierung*. Sie baut auf den Ansätzen zur imperativen Programmierung auf und erweitert diese durch zusätzliche Eigenschaften.

2.3.1 Objektorientierte Programmierung

Bei der *objektorientierten Programmierung* (kurz: OOP) handelt es sich um eines der am weitesten verbreiteten Programmierparadigmen überhaupt. Grundlage dieses Verfahrens bildet die Zusammenfassung von artverwandten Daten und der darauf arbeitenden Programmlogik zu Einheiten, den so genannten *Objekten* [Erg05b]. Dies bedeutet, dass der Programmablauf nicht mehr streng sequentiell (wie bei der *prozeduralen* Programmierung) verläuft, sondern sich aus der Kommunikation zwischen den einzelnen Objekten und deren Zustandsänderungen ergibt. Ausserdem verzichtet diese Art der Programmierung auf strikte Trennung von Daten und Funktionen, die diese manipulieren. Simula (**Sim**ulation **La**nguage) gilt als die erste objektorientierte Programmiersprache. Sie wurde in den 1960er Jahren von Ole-Johann Dahle und Kristen Nygaard entwickelt.

Es gibt vier wesentliche Eigenschaften, die charakteristisch sind für die objektorientierte Programmierung:

- Abstraktion
- Kapselung
- Vererbung
- Polymorphie

Die Eigenschaften eines Objektes werden in *Klassen* festgelegt. Wichtige Elemente einer Klasse sind: Variablen (für Zustandsspeicherung) und Methoden. Letztere beschreiben z.B. Schnittstellen zur Kommunikation mit anderen Objekten und Algorithmen zur

Manipulation eines Objektes (Erzeugung, Änderung, Löschung). In der Interaktion mit anderen Objekten spielt es überhaupt keine Rolle, wie diese Eigenschaften und Funktionen implementiert sind (zumindest solange sie ihre Funktion erfüllen). Ein Objekt stellt also die *Abstraktion* eines ″Akteurs″ im Programm dar, die es ermöglicht, seine Funktionen, ohne das Wissen über die konkrete Umsetzung zu nutzen. Eng verbunden mit dem Begriff der Abstraktion ist auch die Eigenschaft der *Kapselung*. Fremde Objekte können nicht auf interne Strukturen des Objektes, mit dem sie in Kontakt treten wollen, zugreifen, sondern müssen dafür die bereitgestellten Schnittstellen nutzen. Die Kapselung der Objektdaten gewährleistet einen kontrollierten Zugriff auf das Objekt.

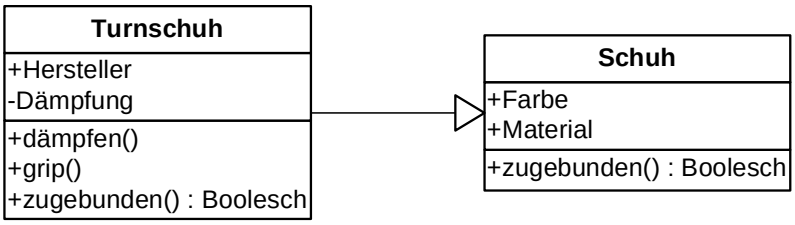

Abbildung 2.7: Einfaches Vererbungsbeispiel Schuh

Oft ist es so, dass eine neue Objektklasse erzeugt wird, die Eigenschaften besitzt, welche in einer anderen Objektklasse (teilweise) schon definiert wurden. Es ist in Hinblick auf die Wiederverwendbarkeit von Programmteilen sinnvoll, wenn die neue Klasse Eigenschaften der existierenden Klasse nutzen kann. Genau dies leistet die *Vererbung*. In Abbildung 2.7 ist ein einfaches Beispiel zum Vererbungsaspekt dargestellt. Hier erbt die Klasse *Turnschuh* die Definition (Attribute: *Farbe*, *Material*, Methode: *zugebunden()*) der Basisklasse *Schuh*. Manche Programmiersprachen lassen sogar das Erben von mehreren Klassen zu (Stichwort: *Mehrfachvererbung*).

Zum Verständnis der *Polymorphie* (auch: Vielgestaltigkeit) wird ein Objekt betrachtet, welches seine Eigenschaften sowohl aus der zugehörigen Objektklasse als auch mittels Vererbung aus einer oder mehrerer *Basisklassen* erhält. Dieses Objekt kann während der Programmabarbeitung verschiedene Gestalten annehmen. Für den Fall, dass nur die Eigenschaften einer Basisklasse benötigt werden, kann es den Typ der entsprechenden Basisklasse annehmen (bezeichnet als: Up-Cast). Dabei werden die Eigenschaften und Funktionen der ursprünglichen Objektklasse ″ausgeblendet″. Bei Methoden die, sowohl in den Basisklassen als auch in der aktuellen Klasse implementiert sind, ist es von besonderer Bedeutung, die passende Methode zum Einsatz zu bringen (Problem der Vielgestaltigkeit von Methoden). Bezüglich des Beispiels aus Abbildung 2.7 kann ein Objekt *Turnschuh* den Typ *Schuh* annehmen. Somit sind auf direktem Weg nur noch die Eigenschaften und Methoden aus der Klasse *Schuh* erreichbar. Auf weiterführende Informationen zur objektorientierten Porgrammentwicklung wird an dieser Stelle verzichtet und auf entsprechende Literatur verwiesen (u.a.: [Mey90], [Jac94]).

2.3.2 Komponentenbasierte Software-Entwicklung

Die objektorientierte Programmierung beinhaltet erste Ansätze zur Wiederverwendung von Programmteilen (z.B.: Vererbung). An dieser Form der Programmierung setzt das Paradigma der *komponentenbasierten Programmierung* an [Szy98]. Hier wird die Idee der Wiederverwendung aufgegriffen und weiterentwickelt. Eine Möglichkeit zur Lösung komplexer Problemstellungen ist die Aufteilung dieser in viele kleine Teilprobleme. Die Teilprobleme werden, jedes für sich, in geeigneter Form gelöst. Abschliessend werden die entwickelten Teillösungen so zusammengesetzt, dass das Ergebnis die Lösung der ursprünglichen Problemstellung widerspiegelt.

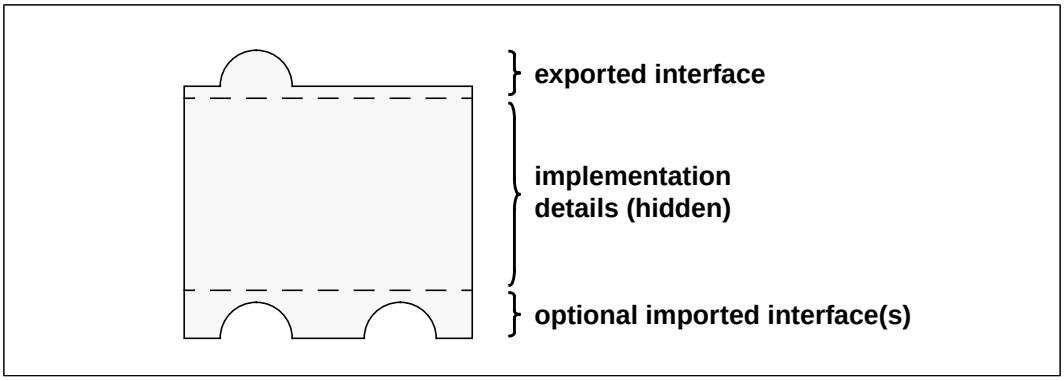

Abbildung 2.8: Zusammenhang Komponente - Schnittstelle [Sin96]

Komponenten sind geradezu prädistiniert für derartige Problemlösungen. Insbesondere können existierende Teillösungen vergangener Software-Entwicklungen, sofern sie komponentenorientiert organisiert sind, ohne grossen Aufwand in neuen Projekten wieder verwendet werden. Zu den bekanntesten Komponentenmodellen (auch: Komponenten-Frameworks) zählen *JavaBeans*, *Component Object Modell (COM+)*, *CORBA* oder *ActiveX*.

Der Komponentenbegriff wird in den verschiedensten Zusammenhängen verwendet. Daher ist es notwendig, genau festzulegen, welche Bedeutung ihm im Kontext der komponentenbasierten Software-Entwicklung zukommt.

Szyperski [Szy98] gibt eine umfassende Definition des Begriffes der Software-Komponente:

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

Demnach sind Komponenten Fragmente, welche über festgelegte Schnittstellen mit der Aussenwelt kommunizieren. Durch die Trennung von Schnittstelle und Implementierung wird die Kapselung der Komponente realisiert (Abb.: 2.8). Gleichzeitig sollte eine

Komponente als eine, vom Einsatzgebiet und von anderen Komponenten, unabhängige Funktionseinheit konzipiert sein. Hervorzuheben ist, dass eine Komponente als Ganzes eingesetzt wird (eine partielle Nutzung ist nicht erwünscht).

Nierstrasz und Dami geben folgende kurze Definition einer Software-Komponente [Nie95]:

> ″A software component *is a static abstraction with plugs.*″

Hier liegt die Betonung auf der Dauerhaftigkeit einer entwickelten Komponente. Durch diese Eigenschaft ist gewährleistet, dass die Komponente für eine Wiederverwendung auch über einen grösseren Zeitraum zur Verfügung steht.

Der Entwicklungsprozess von komponentenbasierten Programmen verläuft, nach Berg in [Hub95], grundsätzlich in zwei Phasen:

- Komponentenerstellung (bezeichnet als: programming in the small)
- Systemerstellung (bezeichnet als: programming in the large)

Der Prozess der *Komponentenerstellung* muss möglichst flexibel gestaltet werden. Er bedient sich zumeist herkömmlicher Werkzeuge und Entwicklungstechniken, wie sie auch in ordinären Anwendungsentwicklungen üblich sind. Jedoch werden in [Kar95] wichtige Hinweise für die Vorgehensweise der Erstellung gegeben. Demnach muss die Komponentenentwicklung *für* eine spätere Wiederverwendung berücksichtigen, dass die Komponente auch in einem anderen als dem ursprünglich geplanten Kontext einsetzbar sein soll. Darauf muss schon in frühen Phasen des Domain Engineering geachtet werden.

Bezüglich der Wiederverwendung von Komponenten gibt es verschiedene Ausprägungen. Zwei davon sollen an dieser Stelle kurz genannt werden. Die *Black-Box-Wiederverwendung* (engl.: Black Box Reuse) betrachtet die Komponente als eine Programmeinheit, die unverändert in die Zielumgebung integriert werden soll. Dem steht die *Glass-Box-Wiederverwendung* (engl.: Glass Box Reuse) gegenüber. Hier kann eine Komponente auch nachträglich an die Eigenschaften des Einsatzgebietes angepasst werden.

Bei der Systemerstellung kommt es darauf an, die vorhandenen Komponenten so zu kombinieren (auch: *Komposition*), dass als Resultat die gewünschte Anwendung entsteht. Von zentraler Bedeutung bei der Komposition einzelner Komponenten ist die *Schnittstelle* [Gri98]. Die Schnittstellenqualität einer Komponente entscheidet darüber, ob die Kommunikation mit anderen Komponenten reibungslos verläuft. Sie bietet einen wichtigen Anhaltspunkt bei der Bewertung der Korrektheit des erstellten Gesamtsystems, da sie die Beziehungen der Komponenten untereinander beschreibt. Um auf verschiedene Anwendungsfälle adäquat reagieren zu können und die Variabilität von Komponenten zu steigern, können diese auch mehrere verschiedene Schnittstellen besitzen. Dies ergibt sich oft aus der Tatsache, dass eine Komponente nachträglich für eine Anwendung angepasst werden muss, weil die vorhandenen Schnittstellen die Anforderungen nicht erfüllen.

Einen weiteren Schritt zur Verbesserung der Einsatzmöglichkeiten von Komponenten stellt die Einführung von *IDLs* (engl.: *Interface Definition Language*) dar. Dabei handelt es sich um plattformunabhängige Sprachen, welche die Schnittstelle beschreiben. Aufbauend auf den Schnittstellenbeschreibungen gibt es auch Beschreibungssprachen, die Informationen über die Komponente bereitstellen. Aus den so genannten *CDLs* (engl.: *Component Definition Language*) können dann plattformspezifische Komponentenbeschreibungen generiert werden.

Die komponentenbasierte Software-Entwicklung birgt eine Vielzahl von Problemen, die dem Vorteil der Wiederverwendung gegenüberstehen [Ber01]:

- Die Komplexität einer Problemstellung behindert eine optimale Dekomposition (Zerlegung in Teilprobleme) des Problems in unabhängige Fragmente. Es kann vorkommen, dass wichtige Details der Problemstellung durch Komponentenbeschreibungen nicht in ausreichendem Umfang dargestellt werden können.

- Sollen neue Komponenten unter Wiederverwendung vorhandener Komponenten erzeugt werden, so ergeben sich daraus verschiedene Anforderungen an das Komponentenmodell. Um beispielsweise eine optimale Komposition zu gewährleisten, ist es wichtig, dass die genutzten Komponenten ihrer Zielbestimmung möglichst ohne Modifikation zugeführt werden. Diese Forderung muss der verwendete Kompositionsmechanismus leisten.

- Erfordert der Betrieb von Komponenten eine Synchronisation verschiedener Prozesse (z.B. wenn eine Komponente als Input den Output einer anderen Komponente benötigt), so muss ein entsprechender Synchronisationsmechanismus eingeführt werden.

- Die Schwierigkeit der Adaption von Komponentenstrukturen zur Laufzeit

Dies sind nur einige Probleme, die den Nutzen einer komponentenbasierten Software-Entwicklung mindern. Durch den Einsatz von *Middleware* können einige der bestehenden Probleme gelöst werden. Wie in Abbildung 2.9 angedeutet, fungiert die Middleware als eine Technologie, die verschiedene Basisdienste zur Verfügung stellt, welche beispielsweise die Kommunikation von Komponenten unterstützen. Sie abstrahiert die angebotenen Dienste von den systemspezifischen Architekturen, auf denen diese Technologie aufsetzt.

Ein weiteres Problem liegt in der Effektivität von komponentenbasierten Entwicklungen. Da Komponenten so entwickelt werden sollen, dass sie bei Wiederverwendung nicht angepasst werden müssen, tragen sie möglicherweise einen Überschuss an Funktionalität mit sich, der im Einzelfall gar nicht benötigt wird. Dies ist gerade im Bereich der eingebetteten Systeme nicht erwünscht, da dort sparsam mit den vorhandenen Ressourcen umgegangen werden muss. Das beschriebene Problem führt zum Begriff der *Granularität*. Es ist wichtig, das richtige Verhältnis aus Funktionalität und minimaler Umsetzung zu finden. Ist eine Komponente zu ″grob″ strukturiert, so kommt es zum beschriebenen Problem der Ressourcenverschwendung. Bei einer zu feinen Strukturierung kommt der

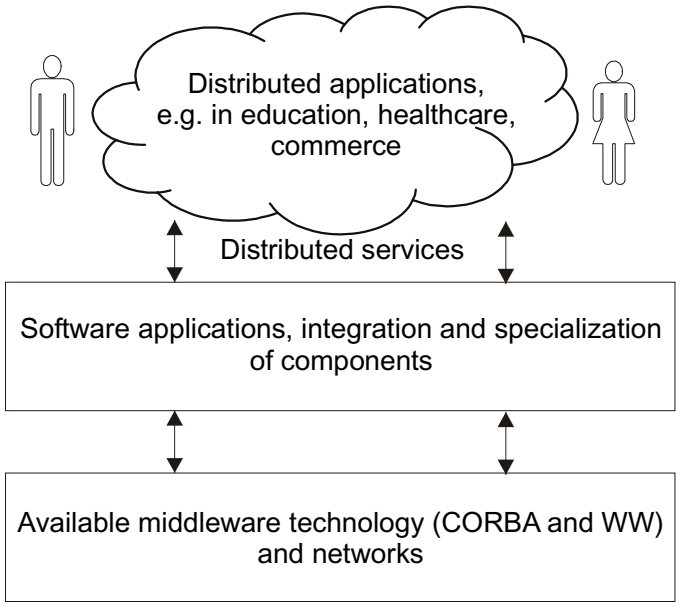

Abbildung 2.9: Einordnung von Middleware nach [Ber01]

eigentliche Sinn der Wiederverwendung, die Vermeidung der Entwicklung immer neuer Funktionalitäten, abhanden [Gri98].

Trotz der Problemvielfalt im Umgang mit komponentenbasierten Technologien, lohnt eine genauere Betrachtung. In vielen Bereichen kommt die komponentenbasierte Entwicklung mittlerweile zum Einsatz. Während die Mehrheit der herkömmlichen Ansätze mehr oder weniger ausgereizt ist, birgt diese Disziplin grosses Entwicklungspotential.

2.3.3 GenVoca

Für die Umsetzung der in Abschnitt 2.3.2 eingeführten komponentenbasierten Software-Entwicklung bedarf es, gerade in der Phase der Komposition der einzelnen Elemente, einiges an technischen Grundvoraussetzungen. Es muss eine Technologie eingesetzt werden, die in der Lage ist, eine zulässige Kombination der Komponenten zu generieren. Dabei spielt es natürlich eine wichtige Rolle, dass diese Kombination dem gewünschten Endprodukt entspricht. Diese Aufgabe wird von *Software System Generators (kurz: SSG)* erfüllt. Generatoren rekrutieren die Ausgangsmaterialien (u.a.: Komponenten und architekturspezifische Informationen) aus entsprechenden Ressourcen-Bibliotheken und konstruieren daraus das Zielobjekt. Damit bei der Komposition der Elemente ein sinnvolles Ergebnis herauskommt, ist es ganz besonders wichtig den Generator mit zusätzlichen Modellbeschreibungen zu versorgen, die auch festgelegte Kombinationsregeln (hier: in Bezug auf die Schnittstelleneigenschaften der Komponenten beinhalten. Ein solches Modell ist *GenVoca*.

Die Basis der Software-Entwicklung bildet das Prinzip der *schrittweisen Verfeine-*

rung (engl.: step-wise Refinement). Dieses Prinzip gründet auf der Idee, dass jedes Programm durch Hinzufügen kleinster Ergänzungen erstellt werden kann. Ein häufig auftretendes Problem in diesem Zusammenhang besteht darin, dass die Granularität solcher Ergänzungen so fein und die Abhängigkeiten so vielfältig sind, dass der Aufwand für die Einführung neuer Funktionalitäten, gerade bei grossen Software-Projekten, steigt (bekannt als: Skalierbarkeitsproblem) ([Bat02]). In Abbildung 2.10 ist darüber hinaus zu erkennen, dass bei nicht-skalierbaren Systemen der Zugewinn an Funktionalität in keinem günstigen Verhältnis zur eingeführten Anzahl (genauer: exponentiell steigender Aufwand) neuer Module steht. Ganz im Gegensatz zu skalierbaren Systemen (genauer: linear steigender Aufwand).

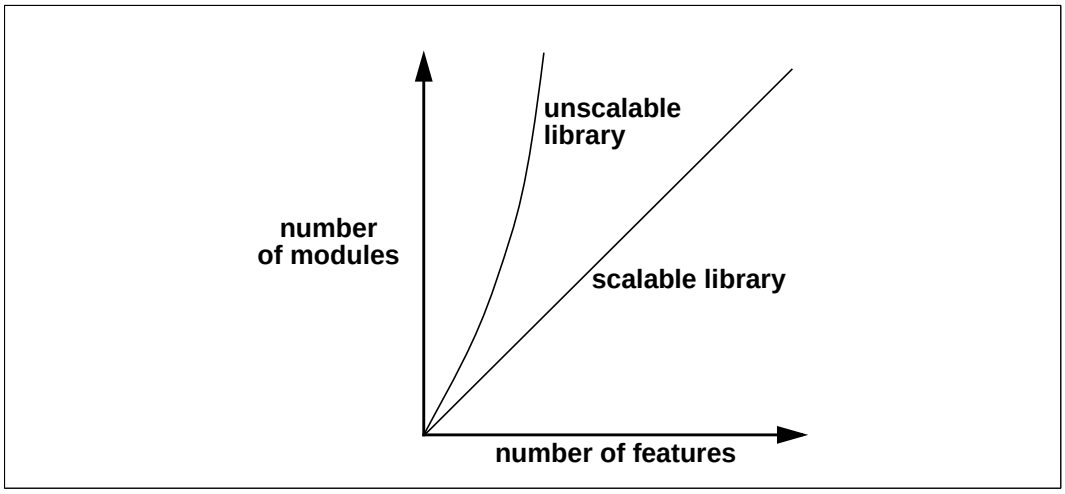

Abbildung 2.10: Skalierbarkeitsproblem [Sin96]

Es ist also wünschenswert die kleinste erstellbare Software-Einheit so zu wählen, dass unter Verwendung weniger Elemente eine möglichst grosse Anzahl an Programmen generiert werden kann. Bei GenVoca handelt es sich um eine skalierbare Modellbeschreibung. Die Basiseinheiten in GenVoca sind die Komponenten ([Bat94], [Sin96]). Sie beschränken die kleinste mögliche Software-Einheit auf Merkmale. Jedes dieser Features kann, beginnend bei Basisfunktionalitäten, schrittweise verfeinert werden.

Die Beschreibung (u.a. der Komponentenschnittstellen) und Organisation der Merkmale und ihrer Verfeinerungen erfolgt in Schichten (engl.: Layer). Jede Schicht stellt gegenüber einer anderen Schicht eine *virtuelle Maschine* dar, die dieser, über ihre Schnittstellen, ihre Dienste anbietet. Verschiedene Schichten und Merkmale können nur kombiniert werden, wenn die zugehörigen Schnittstellen dies zulassen. GenVoca beinhaltet Mechanismen, die eine Validierung der Zulässigkeit von Merkmalskombinationen aufgrund von Schnittstellenanalysen ermöglicht. Merkmale mit identischer Schnittstellenbeschreibung können beliebig getauscht werden. Derartige Merkmale gehören der gleichen Klasse, bezeichnet als *Realm*, an. Nachfolgend sind die Realms S, T und W dargestellt:

$$\begin{aligned} S &= \{\ a,\ b,\ c\ \} \\ T &= \{\ d[S],\ e[S],\ f[S]\ \} \\ W &= \{\ g[W],\ h[W],\ m[T],\ i,\ j[T,\ S]\ \} \end{aligned}$$

In den eckigen Klammern werden die Schnittstellen angegeben, die von der jeweiligen Komponente selbst benötigt werden, um beispielsweise mit anderen Komponenten zu kommunizieren. Es kommt vor, dass eine Komponente (hier: $j[T, S]$) mehrere Interfaces beansprucht. Komponenten, die keine Schnittstellen fordern, werden als *terminal* bezeichnet. Wenn eine Komponente die gleiche Schnittstelle bereitstellt (exportiert), wie sie selber fordert (importiert), dann können dadurch rekursive Konstrukte modelliert werden. Eine solche Komponente wird als *symmetrisch* bezeichnet [Bat97]. Im obigen Beispiel sind innerhalb des Realms W die Komponenten $g[W]$ und $h[W]$ symmetrisch. Damit ist es möglich, Kompositionen der Art: $g[h[i]]$ oder $h[g[j[T, S]]]$ zu generieren. Dabei sind solche Kompositionen schon als einfache Programme zu verstehen. Nachfolgend sind beispielhaft zwei Programme angegeben, die bezüglich der obigen Realms korrekte Kombinationen darstellen:

$$\begin{aligned} Prog_1 &= g[m[f[c]]] \\ Prog_2 &= h[g[j[e[c],\ b]]] \end{aligned}$$

Aus den obigen Beschreibungen (Realms) kann eine Grammatik (Chomsky-Grammatik vom Typ 2) $G = (N,\ T,\ P,\ S)$ abgeleitet werden, deren Wörter die potentiell erzeugbaren Programme beschreiben (N ist die Menge der Nichtterminalsymbole, T ist die Menge der Terminalsymbole, P ist die Menge der Produktionsregeln und S ist das nicht-terminale Startsymbol, und die kontextfrei ist. Zulässige Produktionsregeln einer Chomsky-Grammatik ($G \in Typ_2$) haben die Form: $\forall(w_1 \rightarrow w_2) \in P : w_1 \in N \wedge w_2 \in (T \vee N)$. Die auf einer solchen Grammatik aufsetzende Sprache $L(G)$ beschreibt genau die Systemfamilie, welche mit einem GenVoca-Generator erzeugt werden kann [Par79].

Die aus den angegebenen Realms resultierende Menge der Produktionsregeln lautet wie folgt:

$$P = \{\ S ::= a \mid b \mid c\ ,\ T ::= d\ S \mid e\ S \mid f\ S\ ,\ W ::= g\ W \mid h\ W \mid m\ T \mid i \mid j\ T\ S\ \}$$

Dabei bilden die Komponenten a, b, c und i offensichtlich die Menge der terminalen Symbole (T), da sie keine Schnittstelle zu anderen Komponenten fordern. Die restlichen Komponenten bilden die Menge der nicht-terminalen Symbole (N), da sie jeweils nachfolgend benötigte Schnittstellenkategorien (auch: Parameter) angeben. Durch die obigen Festlegungen in Form von Realms und der daraus hervorgegangenen GenVoca-Grammatik wird die syntaktische Korrektheit von Kompositionen, wie etwa in $Prog_1$

und $Prog_2$ angegeben, gewährleistet. Auch neu hinzugefügte Komponenten, welche eine Spracherweiterung des GenVoca-Modells bedeuten, können auf diese Weise ordnungsgemäss eingepflegt werden.

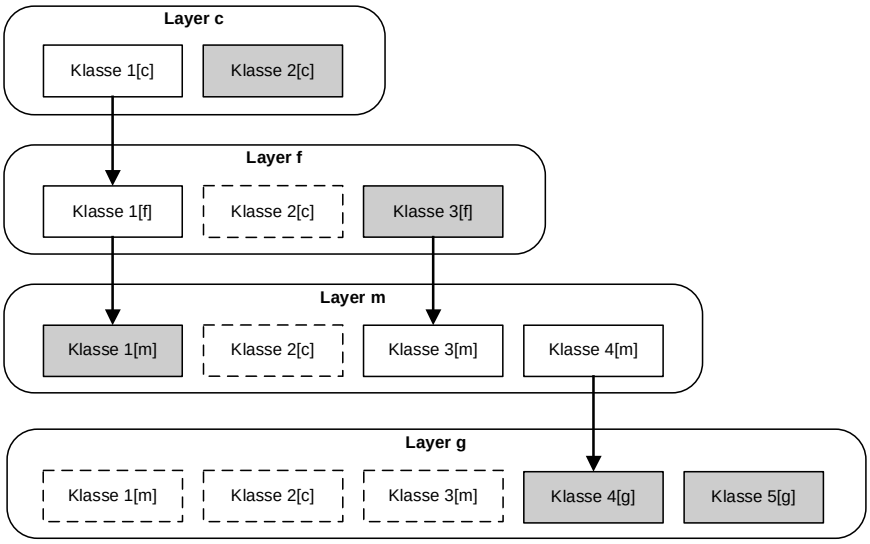

Abbildung 2.11: GenVoca - Schichtenarchitektur

Das in Abbildung 2.11 dargestellte Beispiel einer Schichtenhierarchie entspricht der Komposition $Prog_1$. Jede Schicht stellt eine der involvierten Komponenten (gemäss $Prog_1$ von innen nach aussen: c, f, m und g) dar. Layer c beinhaltet die Klassen 1 und 2. Hierbei handelt es sich um die ersten Entwicklungsstufen der durch die Klassen ausgedrückten Funktionalität (Fragmente von Methoden und Klassen). Eine neu eingeführte Klasse wird hier als *Konstante* bezeichnet, da sie keine direkten Vorgänger hat. In Layer f wird eine zusätzliche Klasse installiert. Darüber hinaus wird Klasse 1, ausgehend von der *Parent-Klasse* aus Layer c, *verfeinert* (Pfeilbeziehungen). Unveränderte Klassen werden durch einen gestrichelten Rahmen gekennzeichnet. Layer m enthält zwei *Refinements* (Klasse 1, Klasse 3) und eine neue Klasse (4). Diese Klasse wird in Layer g verfeinert. Des Weiteren wird hier Klasse 5 eingeführt. Die grau unterlegten Klassen stellen beispielhaft die im komponierten Endprogramm letztendlich verwendeten Elemente dar.

Der durch GenVoca-Grammatiken ermöglichte Test der syntaktischen Korrektheit von Kompositionen stellt noch keine hinreichende Bedingung für die darauf aufbauende generative Programmierung dar. Durch zusätzliche Informationsquellen, wie etwa der Domänenentwicklung, muss ergänzend die semantische Korrektheit der Kompositionen sichergestellt werden, um eine Applikation zu erschaffen, die in jeder Hinsicht den festgelegten Anforderungen gerecht wird. Solche anwendungsbereichsspezifischen Bedingungen (engl.: domain-specific Constraints) werden auch als Entwurfsregeln (engl.: Design Rules) bezeichnet.

Jede Komponente A muss definieren, welche Eigenschaften eine Komponente B besit-

zen muss, um in unmittelbarer ″Nachbarschaft″ von Komponente A eingesetzt werden zu können. Gleichzeitig muss Komponente A ihrerseits Angaben zur Entscheidungsfindung bezüglich einer Nachbarschaftsbeziehung für Komponente B bereitstellen. Die Flussrichtung solcher Informationen verläuft, beginnend bei der Top-Komponente (Komponente, die am weitesten links in der Komposition angegeben ist), zu den inneren Komponenten hin. Im Beispielprogramm $Prog_1$ ist der Ausgangspunkt die Komponente g. Hier wird zunächst geprüft, ob es sich tatsächlich um die Top-Komponente handelt. Aufbauend auf den definierten Forderungen von g bezüglich der Nachbarschaft, werden die umgebenden Komponenten (hier: m) analysiert. Erfüllt m die Anforderungen, so wird in der beschriebenen Weise mit der benachbarten Komponente von m fortgefahren (hier: f). Sollte dies nicht der Fall sein, so handelt es sich um eine ungültige Komposition, die mit einer entsprechenden Fehlermeldung abgebrochen wird. Es ist zu erkennen, dass es sich um ein rekursives Verfahren handelt, bei dem die Anforderungen von Komponente zu Komponente ″weitergereicht″ werden (ergänzt durch Bedingungen der aktuell betrachteten Komponente). Angekommen bei der innersten Komponente (hier: c) wird die Überprüfung in umgekehrter Richtung vollzogen (von innen nach aussen). Sollte die Überprüfung in Hin- und Rückrichtung ohne Beanstandung verlaufen sein, so handelt es sich um eine, den Entwurfsregeln entsprechend, gültige Komposition. Dies gewährleistet aber noch nicht, dass die resultierende Applikation ein, im Kontext der zugrunde liegenden Programmfamilie, sinnvolles Programm darstellt. Die Definition von Design Rules schränkt lediglich die Menge unzulässiger Programme ein [Bat97].

Ein anderes Problem von GenVoca bezeichnen Batory und Geraci [Bat97] als *Subjektivität*. Dieses Phänomen beschreibt Schwierigkeiten in der domänenübergreifenden Nutzung von Komponenten. Die Probleme rühren beispielsweise von den Komponentenschnittstellen her, da diese häufig für die native Domäne entwickelt wurden und andere Domänen diese nicht ausreichend unterstützen.

2.3.4 Feature-orientierte Programmierung

Seit einiger Zeit wird davon gesprochen, dass die Software-Entwicklung in einer Krise steckt. Etablierte Entwicklungsmethoden, wie sie gegenwärtig zum Einsatz kommen, werden den Forderungen nach einer übersichtlichen und effektiven Programmentwicklung nicht mehr in vollem Umfang gerecht. Dies liegt zumeist an der immer stärker ausgeprägten Komplexität neuer Anwendungen.

Die *merkmalsorientierte Programmierung* (engl.: **F**eature-**O**riented **P**rogramming - FOP) ist ein viel versprechender Ansatz, der viele Probleme heutiger Methoden löst. Dabei greift FOP die in Abschnitt 2.3.3 behandelten Ideen und Konzepte von GenVoca auf und erweitert diese. Während es in GenVoca nicht möglich ist, auf direktem Weg neue Komponenten aus vorhandenen zu erstellen, so bietet FOP diese Möglichkeit. Bei der Betrachtung eines Merkmalsdiagramms, wie es in Abschnitt 2.2.5 beschrieben ist, wird dieser maßgebliche Unterschied deutlich. Ein Merkmal kann sich aus verschiedenen Untermerkmalen zusammensetzen. Zum Beispiel setzt sich das Feature *Antrieb* (Abbil-

dung: 2.6) wahlweise aus verschiedenen Merkmalen (*Elektro*, *Verbrennungsmotor*) und deren Untermerkmalen zusammen.

Batory [Bat05] beschreibt die merkmalsorientierte Programmierung als ein Programmierkonzept, das die Organisation und Verwendung der Merkmale in Form von Modulen unterstützt. Merkmale bilden die zentralen Einheiten der Modularität. In [Bat05] wird die *schrittweise Entwicklung* (engl.: **s**tep-**w**ise **D**evelopment (SWD)) als eine wirkungsvolle Variante der FOP bezeichnet. Hier werden die bestehenden Programmteile (Merkmale) systematisch Schritt für Schritt erweitert (siehe auch: 2.3.3).

Der Merkmalsbegriff ist bisher an verschiedenen Stellen der Arbeit kommentiert und erläutert worden. Nachfolgend soll dieser kurz auf Quellcode-Ebene betrachtet werden. In aktuellen Programmierkonzepten ist es nicht ganz einfach, ein Programm durch neue Merkmale zu erweitern. Es sind in der Regel Ergänzungen über eine Vielzahl von Klassen und Methoden nötig (Problem: *Code Scattering*). Einen objektorientierten Ansatz für die teilweise Lösung dieses Problems stellt die Klassenerweiterung (engl.: *Class Extension*) dar. Durch eine derartige Erweiterung können innerhalb einer Vererbungshierarchie neue Methoden oder Variablen (auch: *Member*) hinzugefügt oder existierende Methoden erweitert werden.

Listing 2.1: Klasse StartBasis

```
1   class StartBasis {
2     Button start;
3     void starten() {...}
4   };
5
6   class Startextended : public StartBasis {
7     Button End;
8     void beenden() {...}
9   };
10  typedef Startextended StartEnd;
```

In Listing 2.1 wird die Basisklasse *StartBasis* mit zwei Klassenelementen definiert. *StartBasis* wird durch Klasse *Startextended* erweitert. Zu den geerbten Elementen aus der Klasse *StartBasis* werden zwei weitere Elemente hinzugefügt. Listing 2.2 beinhaltet eine komplette Klassendefinition von *Startextended*, bei der alle Elemente, ohne Vererbung aus einer Basisklasse, integriert werden.

Listing 2.2: Klasse Startextended

```
1   class Startextended {
2     Button start;
3     Button End;
4     void starten() {...}
5     void beenden() {...}
6   };
```

Der Vorteil der Klassenerweiterung liegt auf der Hand. Durch die Verwendung einer Vererbungskette können komplexe Programme übersichtlich strukturiert werden. Gleichzeitig ist es möglich effektive Klassenerweiterungen zu realisieren. *Startextended* stellt eine Verfeinerung der Klasse *StartBasis* dar. Durch eine zusätzliche Typdefinition (Listing

2.1 Zeile: 10) kann die behandelte Klasse (hier: *Startextended*) von der Vererbungskette abstrahiert werden. Dadurch wird die schrittweise Verfeinerung einer Klasse (hier: *StartEnd*) angedeutet.

In FOP wird die Idee der Klassenerweiterung thematisiert und systematisiert. Dies ist im Sinne der Übersichtlichkeit und Automatisierung der schrittweisen Verfeinerung sehr sinnvoll. Listing 2.3 zeigt die Umsetzung des Beispiels mittels merkmalsorientierter Programmierung. Beginnend mit einem Basisprogramm werden erste Funktionalitäten implementiert (Listing 2.3 Zeilen 1-4). Eine solche initiale Klasse wird auch als *Value* bezeichnet. Die Verfeinerung einer Klasse (Listing 2.3 Zeilen 6-9) beginnt mit dem Signalwort *refines*. Da jede Verfeinerung ihren Ursprung in einem *Value* hat, und dieser durch sie eine Erweiterung erfährt, wird eine Verfeinerung auch *Function* genannt. Im Sinne von FOP wird die Komposition verschiedener *Values* und *Functions* als *Expression* bezeichnet.

Listing 2.3: Klasse StartEnd

```
1  class StartEnd {
2    Button start;
3    void starten() {...}
4  };
5
6  refines class StartEnd {
7    Button End;
8    void beenden() {...}
9  };
```

Wie aus Abbildung 2.11 in Abschnitt 2.3.3 hervorgeht, ist es notwendig, den bisher diskutierten Ansatz der Verfeinerung zu erweitern. Oftmals ist eine gleichzeitige Verfeinerung mehrerer Klassen erforderlich (bezeichnet als: *large-scale Refinement* [Bat05]), da der erforderliche Quellcode für das einzuführende Merkmal über verschiedene Klassen verteilt werden muss. Die von Batory et al. für AHEAD (**A**lgebraic **H**ierarchical **E**quations for **A**pplication **D**esign), einer auf GenVoca aufbauenden Umsetzung der merkmalsorientierten Programmierung, genutzte Notation und die Idee der *large-scale Refinements* werden nachfolgend erläutert. Dabei sei F eine Menge von Merkmalen (in [Bat05] als Einheiten (engl.: Units) bezeichnet), die sich aus *Values* und/oder *Functions* zusammensetzen.

$$F \quad = \quad \{\ a,\ b,\ c,\ d,\ ...\ \}$$

Hervorzuheben ist, dass diese Einheiten (hier z.B.: a, b, c) sich selbst rekursiv aus Features zusammensetzen können. Die Komposition der einzelnen Merkmale (Einheiten) wird durch einen Verbindungsoperator (Binär-Operator) (hier: $\bullet$) gekennzeichnet. Zur beispielhaften Erläuterung einer Komposition seien zwei Units X (X sei in diesem Fall eine Konstante, deren Menge die beteiligten konstanten Klassen sind) und Y gegeben. Die Indizes der einzelnen Mitglieder von X und Y signalisieren jeweils, welcher Herkunft das verwendete Artefakt ist.

$$\begin{aligned} X &= \{\ a_x, \quad\ c_x,\ d_x\ \} \\ Y &= \{\ a_y,\ b_y, \quad\ d_y\ \} \end{aligned}$$

Die paarweise Komposition der beiden Units X und Y erfordert das Zusammenfügen aller Mengenmitglieder, die (in eventuell unterschiedlicher Ausprägung) in beiden Mengen vorkommen:

$$X \bullet Y = \{\ a_x \bullet a_y,\ b_y,\ c_x,\ d_x \bullet d_y\}$$

Da der Verfeinerungsoperator nicht kommutativ ist, weil bestehende Verfeinerungen von neuen Verfeinerungen überlagert werden, kommt es beim Zusammenfügen der Elemente auf die richtige Reihenfolge, bezogen auf den Verfeinerungsoperator, an. Darüber hinaus gilt die Assoziativität. Eine Kombination aus Verfeinerungen ist wiederum eine Verfeinerung. Aus einer Komposition $Feature_1 \bullet Feature_2$ entsteht eine Konstante, wenn $Feature_2$ eine Konstante ist. Die Kompositionsgleichungen werden von Batory et al. [Bat03] als *Equations* bezeichnet. Ergänzend wird darauf hingewiesen, dass die Verwendung der erläuterten Kompositionsgleichungen die Nutzung von automatisierten Optimierungsverfahren (unter Vorgabe von Optimierungszielen) ermöglicht. Dabei wird zur Verdeutlichung dieser Eigenschaft auf die *Structured Query Language* (kurz: SQL) verwiesen, welche ebenfalls auf algebraischen Ausdrücken basiert, deren Effizienz durch Optimierungsverfahren verbessert werden kann.

Da der Verfeinerungsoperator das zentrale Element von Kompositionen ist, wird an dieser Stelle genauer auf dieses Thema eingegangen. Um die oben beschriebenen Funktionen des Operators in einem realen System bereitzustellen, bedarf es einer entsprechenden Implementierung. Ein Basiskonzept stellt die Verwendung von *Mixins*[2] dar. Während in anderen Implementierungen des Verfeinerungsoperators (siehe: *Jampack* [Bat03]) die einzelnen Fragmente in einer grossen Repräsentation zusammengefasst werden (wodurch die Fehlersuche und Wartbarkeit erschwert wird, da die Verfeinerungen von einander nicht abgegrenzt sind), bleiben bei der Verwendung von Mixins die Verfeinerungen in ihrer ursprünglichen Form (als Klasse) erhalten. Gleichzeitig wird die Flexibilität (bezogen auf die Vererbung) erhöht, da bei der Implementierung der Verfeinerungen keine konkrete Basisklasse angegeben werden muss.

Listing 2.4: Mixins in C++

```
1  template <class Basis>
2  class Mixin : public Basis {
3                ...
4  };
```

[2] auch: Mixin Class

In C++ können Mixins unter Verwendung von Templates (auch: Typvorlagen) realisiert werden (siehe: Listing 2.4) [Bat00]. Statt eines Basisklassentyps wird ein Parameter angegeben, der zum Zeitpunkt der Instanziierung des Templates durch einen konkreten Typ ersetzt wird. Das Prinzip der Mixins kann auf mehrere Klassen und somit auf Features angewandt werden. Diese Erweiterung wird als *Mixin Layer* bezeichnet. Dabei beinhaltet jede Mixin-Schicht die für die Umsetzung des jeweiligen Merkmals nötigen Mixins (engl.: Collaborating Mixins). Die in Abbildung 2.11 als Klassen bezeichneten Elemente stellen im gegenwärtigen Kontext die Mixins dar, während die Schichten (Layer c, f, m und g) die Mixin Layers sind.

Da die merkmalsorientierte Programmierung auf den Konzepten von GenVoca aufbaut, ist auch die in Abschnitt 2.3.3 beschriebene Entwurfsregelprüfung (engl.: Design Rule Check) Bestandteil des diskutierten Ansatzes. Daher muss die Realisierung des Verfeinerungsoperators um Mechanismen zur Entwurfsregelprüfung erweitert werden.

Die merkmalsorientierte Programmierung ist als eine recht ausgereifte Technologie einzustufen. Jedoch gibt es auch hier Probleme, die nicht unerwähnt bleiben sollen. Eine Schwierigkeit besteht darin, dass sich Klassenerweiterungen durch eine Subklasse nur auf die Elternklasse niederschlagen. Bestehende Subklassen dieser Elternklasse werden dadurch nicht verfeinert (bekannt als: *Extensibility Problem*) [Fin98]. Ein weiteres Problem betrifft die Klassenkonstruktoren, die in herkömmlichen objektorientierten Sprachen nicht automatisch weitervererbt werden. Sie müssen für jede Subklasse redefiniert werden. Dieses Phänomen ist auch bekannt als das Problem der *Konstruktorpropagation* (engl.: Constructor Propagation).

FeatureC++, als die in dieser Arbeit genutzte Technologie zur merkmals- und aspektorientierten Programmierung, löst die genannten Probleme und stellt damit eine wirkungsvolle Erweiterung der von Batory et al. entwickelten Technologien dar [Ape05].

2.3.5 Aspektorientierte Programmierung

Ein weiteres Programmierparadigma stellt die *aspektorientierte Programmierung* dar. Ziel dieser Entwicklung ist es, die Wartbarkeit objektorientierter Software und die Wiederverwendbarkeit von Komponenten zu verbessern [Cza01]. Diese recht allgemeine Beschreibung trifft auch auf andere Technologien, wie z.B. die der merkmalsorientierten Programmierung (siehe: Abschnitt 2.3.4), zu. Daher sollen in diesem Abschnitt die Besonderheiten und Unterschiede zu den vorhergehend erläuterten Methoden hervorgehoben werden.

Wie aus der Begrifflichkeit ersichtlich ist, steht im Blickpunkt dieses Porgrammieransatzes der *Aspekt*. Kiczales et al. [Kic97] führen den Begriff des Aspekts wie folgt ein:

> ″Aspects *tend not to be units of the system's functional decomposition, but rather to be properties that affect the performance or semantics of the components in systemic ways. Examples of aspects include memory access patterns and synchronization of concurrent objects.*″

Aspekte sind demnach Systemeinheiten, die mehrere (verteilte) Programmbestandteile gleichzeitig adressieren. Dabei wird versucht, geeignete Funktionen und den davon betroffenen Programm-Code von den übrigen Programmfragmenten zu trennen (bezeichnet als: Kapselung der Code-Fragmente), und von zentraler Stelle aus verfügbar zu machen. Diese Vorgehensweise wird *Separation of Concerns* (Übersetzung: Kapselung von Forderungen oder Belangen) genannt. Die aspektorientierte Programmierung nimmt sich einer ganz speziellen Form von Concerns an - den *homogenen Crosscutting Concerns* (Übersetzung: homogene querschneidende Forderungen oder Belange). Dabei handelt es sich um Belange, welche die durch herkömmliche Techniken implementierten Kernfunktionalitäten querschneiden, sich also nicht eindeutig einem Software-Modul zuordnen lassen. Der Begriff *homogen* signalisiert, dass identischer Programm-Code an verschiedenen Stellen des Programms eingefügt wird. Dem gegenüber widmet sich die feature-orientierte Programmierung *heterogenen Crosscutting Concerns*, Belangen, die ebenfalls alle möglichen Programmteile adressieren können, aber in jedem einzelnen Fall verschiedenartigen (heterogenen) Programm-Code umfassen, der keine (direkten) Wiederholungen birgt. Im Zusammenhang mit der Entwicklung von Merkmalsdiagrammen umfasst der Merkmalsbegriff sowohl Features, als auch Aspekte. Ist demnach die Rede von Merkmalen, so muss unterschieden werden, um welche Form von Merkmal (homogene oder heterogene Crosscutting Concerns) es sich handelt und wie dieses letztendlich umgesetzt werden soll (FOP vs. AOP).

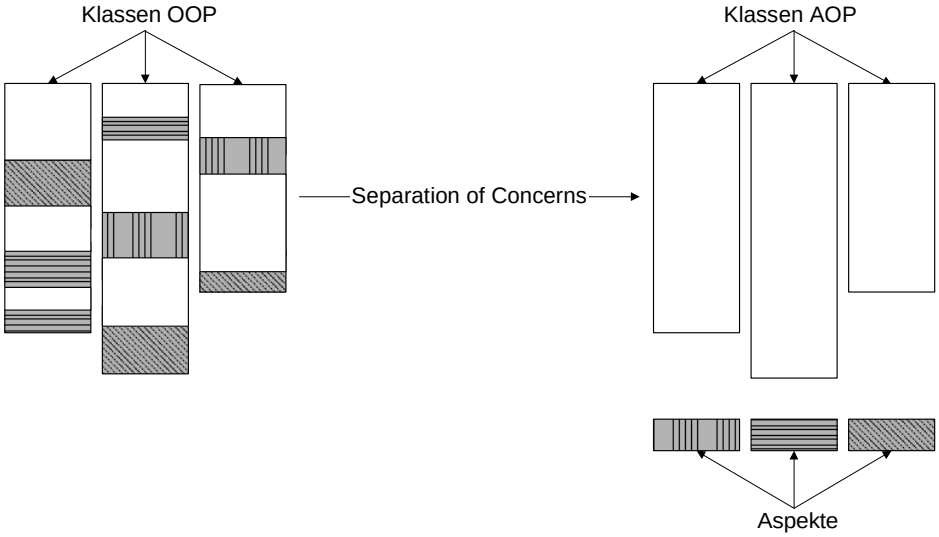

Abbildung 2.12: Kapselung der Belange

Der klassische Anwendungsfall, der merkmalsorientierten Programmierung, ist die Realisierung von Logging-Funktionalitäten. Dabei können sicherlich viele der benötigten Eigenschaften modular (in Form von Merkmalen) implementiert werden. Zumindest die Aufrufe der Eigenschaften erfolgen aber dezentral in allen Programmteilen, die von der Logging-Funktion profitieren wollen. Die Probleme, welche sich dahinter verbergen, haben weitreichende Konsequenzen für die Programmpflege und Erweiterung. Zunächst

kann festgestellt werden, dass homogene Crosscutting Concerns eine saubere modulare (merkmalsorientierte) Programmentwicklung unmöglich machen, da der zugehörige Quellcode über weite Teile des Programms verstreut sein kann. Im Falle einer Änderung von Programmteilen, die einem homogenen Crosscutting Concern zuzuordnen sind, müssen diese Änderungen manuell an jeder betroffenen Stelle vollzogen werden. Dies kann sich gerade in komplexen Programmen als sehr schwierig und fehlerträchtig erweisen.

Um das beschriebene Problem zu umgehen, werden die homogenen Crosscuts durch Aspekte beschrieben, die wie folgt eingeführt werden. Zunächst müssen die existierenden homogenen Crosscutting Concerns identifiziert werden, um sie anschliessend vom übrigen Code trennen zu können. Abbildung 2.12 zeigt drei Klassen, welche an verschiedenen Stellen identischen, querschneidenden Quellcode beinhalten. Nach der Separation liegt der Quellcode der homogenen querschneidenden Belange in Form von Aspekten vor, während die restlichen Programmteile wie bisher in den Klassen organisiert sind.

Listing 2.5: Aspekt-Code

```
1   //protocol.ah
2   aspect protocol {
3
4       //Konstruktor
5       protocol()
6       {
7         //Konstruktordefinition
8       }
9
10      //Ausfuehrung nach Funktionsaufruf print()
11      pointcut log() = call("%␣Liste::␣print(...)");
12      advice log() : before()
13      {
14        //auszufuehrender Code
15      }
16  };
```

Abbildung 2.13: Aspektweber nach [Ros05]

Da die abgespaltenen Aspekte, ähnlich wie Merkmale, spezielle Verfeinerungen von Klassen darstellen, muss geklärt werden, wie dieser Programmcode in die entsprechenden Klassen eingearbeitet wird. Die Ankerpunkte für den Aspekt-Code in den Klassen bilden die *Joinpoints*. Diese Joinpoints werden innerhalb der Aspektbeschreibungen (Aspekt-Code) mittels *Pointcuts* (Listing 2.5: Zeile 11) definiert. Die in den Pointcuts angegebenen Suchausdrücke (auch: *Wildcards*) machen es möglich, dass der Aspekt-Code an

jeder beliebigen Stelle des Programms eingefügt werden kann. Der Aspekt-Code muss ergänzend die auszuführende Funktionalität beinhalten. Zu diesem Zweck wird so genannter *Advice-Code* (Listing 2.5: Zeilen 12-15) definiert, der dann je nach Definition, an der durch den Pointcut festgelegten Stelle, eingefügt wird.

Der Vorgang des Zusammenführens von Aspekten und herkömmlichen Code-Fragmenten wird als *Weben* bezeichnet. Dabei ermittelt der *Aspektweber* auf Basis des Aspekt-Codes (Pointcuts) alle Joinpoints, an denen der zugehörige Advice-Code ausgeführt werden muss. Es wird zwischen statischem (vor der Programmlaufzeit) und dynamischem (zur Laufzeit) Weben unterschieden. Abbildung 2.13 veranschaulicht den Prozess des Aspektwebens schematisch (hier: statisches Weben vor der Programmausführung).

Nachdem allgemeine Eigenschaften und das prinzipielle Vorgehen in AOP beschrieben wurden, sollen FOP und AOP kurz verglichen werden, um die Eignung für verschiedene Anwendungsfälle herauszuarbeiten. Beide Ansätze machen es sich zur Aufgabe, Eigenschaften, die über dem Programmcode verteilt sein können, zu modularisieren. FOP verfeinert den betreffenden Code entlang einer Vererbungslinie (hierarchisch) auf Basis der Parent-Klassen. Homogene Crosscutting Concerns müssten, mittels merkmalsorientierter Programmierung umgesetzt, an jeder geforderten Stelle manuell implementiert werden. Dies ist einerseits ineffektiv und führt andererseits zu Redundanzen im Programm-Code. Weiterhin führen derartige Verfeinerungen in FOP oft zu Schnittstellenerweiterungen (Problem: komplexe Schnittstellen sind unkomfortabel zu handhaben). FOP ist eher dafür geeignet, neuartigen Programm-Code zu implementieren (Bezeichnung: *heterogene Crosscuts*). Ausserdem unterstützt FOP keine *dynamischen Crosscuts*, wie sie in AOP möglich sind. Es kann zur Laufzeit keine Entscheidung für die Ausprägung einer Verfeinerung getroffen werden. Mixin Layers in FOP beinhalten demnach *statische Crosscuts*.

Mittels aspektorientierter Programmierung wird ein Merkmal, im Vergleich zu FOP, aus den beteiligten Klassen und den merkmalsspezifischen Aspekten zusammengestellt. Dabei ermöglichen Aspekte Verfeinerungen die unabhängig von Vererbungshierarchien implementiert werden können. Auf diese Weise ergänzt es hervorragend FOP. Die Idee von AOP beruht darauf, dass eine Funktion an verschiedenen Stellen (Szenario: wiederkehrende Funktionalitäten (*homogene Crosscuts*)) im Programm ausgeführt werden kann, ohne redundanten Code zu erzeugen. Die Kapselung von Aspekten verhindert darüber hinaus das Problem der Schnittstellenerweiterung.

In FeatureC++ wird die aspektorientierte Programmierung im Sinne einer Schichtenarchitektur, wie sie in FOP beschrieben wird, erweitert. Unter Verwendung von *Aspectual Mixin Layers* werden Aspekte mit den aus Abschnitt 2.3.4 bekannten Mixin Layers kombiniert. In Abbildung 2.14 wird das aus Abschnitt 2.3.3 bekannte Beispiel (Abbildung 2.11) zunächst um $Aspekt1[x]$ erweitert. Der Einfluss von $Aspekt1[x]$ erstreckt sich über mehrere Mixins (durch Pfeilbeziehungen gekennzeichnet). $Aspekt1[x]$ kann zudem um weitere Eigenschaften erweitert werden. Dies wird durch Einführung von Aspectual Mixin Layer y und $Aspekt1[y]$ skizziert.

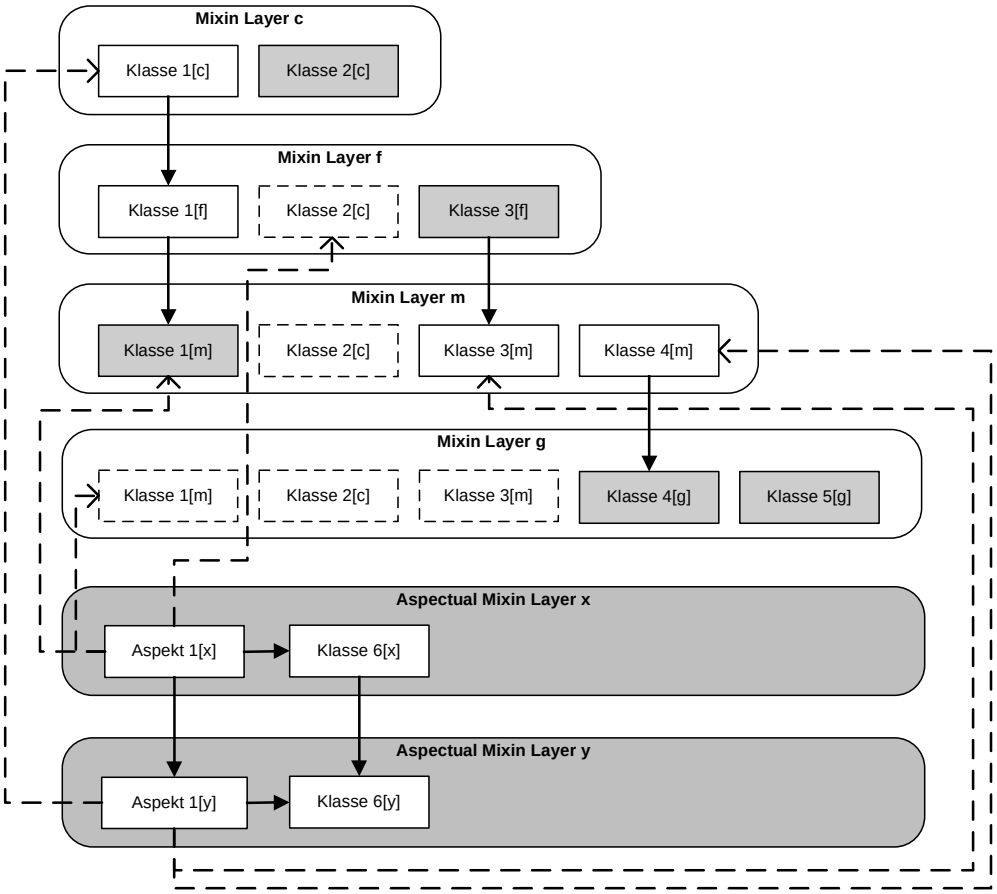

Abbildung 2.14: Aspectual Mixin Layers (mit Verfeinerung)

2.4 Zusammenfassung

In diesem Kapitel wurden die grundlegenden Technologien und Lösungsansätze für den Entwurf und die effektive, flexible Implementierung ausgewählter Abschnitte der Transaktionsverwaltung eingeführt und betrachtet. Zunächst wurden *Programmfamilien* als mögliches Verfahren für die auf Wiederverwendung ausgelegte Programmentwicklung untersucht. Dieser Ansatz erfüllt die Forderung nach Wiederverwendung und soll für die konkrete Umsetzung genutzt werden. Die für Programmfamilien verwendete Methode des Software-Engineering wird *Domain Engineering* genannt. Sie umfasst konkrete Methoden zu: domänenspezifischer Analyse, Entwurf und Implementierung. Zentrale Elemente der Modellierung im Domain Engineering sind die *Merkmalsmodelle*. Sie veranschaulichen die hierarchische Organisation der Komponenten einer Programmfamilie. In diesem Zusammenhang wurden Methoden erläutert, die das Domain Engineering unterstützen - *FODA* und *ODM*.

In Abschnitt 2.3 wurden Programmiertechniken beschrieben, die eine modularisierte Software-Entwicklung, im Sinne einer Programmfamilie, unterstützen. Beginnend bei

grundlegenden Paradigmen wie der *objektorientierten Programmierung*, die schon erste Ansätze für diesen Belang bietet, bis hin zur Untersuchung von *komponentenbasierten* Entwicklungsstrategien. Dabei stellte sich heraus, dass die reine *komponentenbasierte Programmentwicklung* einige Defizite (z.B.: mangelnde Transparenz und Flexibilität von *Black-Box Komponenten*) aufweist, die einer Nutzung im Kontext der Aufgabenstellung im Weg stehen. Trotzdem stellt sie die Grundlage für besser geeignete Technologien dar. Einen weiteren Schritt in die richtige Richtung stellt *GenVoca* dar. Hier werden erste Mechanismen zur Sicherung der syntaktischen und semantischen Korrektheit von automatisch generierten Komponentenzusammenstellungen angewandt. Die *merkmalsorientierte Programmierung* (kurz: FOP) erweitert diese Ansätze. Beispielsweise ist es möglich, eine neue Komponente aus vorhandenen Komponenten zu generieren. Des Weiteren ermöglichen die verwendeten algebraischen Kompositionsgleichungen eine Optimierung der Programmzusammenstellung. Ergänzend zur merkmalsorientierten Programmierung erlaubt die *aspektorientierte Programmierung* (kurz: AOP) die Modularisierung von schwer separierbaren Belangen - *homogenen Crosscutting Concerns*.

Kapitel 3

Domänenanalyse

Gemäss den Vorgaben und Erläuterungen zur Domänenanalyse aus Abschnitt 2.2.1 und zu FODA (2.2.5) erfolgt in diesem Kapitel die Domänenanalyse für den Themenkomplex der Transaktionsverwaltung. Da die zu entwickelnde Programmfamilie für eine Nutzung in eingebetteten Systemen vorgesehen ist, erfolgt im Rahmen der Kontextanalyse eine Untersuchung des Fachgebiets, um alle daraus resultierenden Abhängigkeiten zu extrahieren und die Domänengrenzen zu bestimmen. In diesem Zusammenhang wird darüber hinaus die grundsätzliche Architektur eines Datenbankmanagementsystems betrachtet. Des Weiteren muss geklärt werden, welchen Einschränkungen das DBMS bei der Verwendung in eingebetteten Systemen Rechnung tragen muss. Die Transaktionsverwaltung, als Bestandteil eines DBMS, ist dabei zentraler Aspekt der Betrachtungen. Der Abschnitt der Domänenmodellierung befasst sich mit der Analyse existierender Lösungsansätze bezogen auf die modulare Entwicklung von DBMS im Allgemeinen und Aspekten der Transaktionsverwaltung im Speziellen. Schwerpunkt dieser Untersuchungen sind konkrete Systeme wie *PLENTY* und *COMET*. Aufbauend auf dem erarbeiteten Wissen über die Domäne wird ein entsprechendes Merkmalsmodell entwickelt. Dort werden die domänenspezifischen Merkmale, ihre Eigenschaften und Abhängigkeiten bestimmt. Die Merkmale werden dann in geeigneten Merkmalsdiagrammen angeordnet. Abschliessend wird eine Entscheidung darüber getroffen, welche Zielgruppe die Programmfamilie hat. Diese Vorgabe ermöglicht das zielgerichtete Ausführen der Entwurfs- und Implementierungsprozesse der Domänenentwicklung, da frühzeitig auf die Bedürfnisse der im Fokus stehenden, potentiellen Nutzer eingegangen werden kann.

3.1 Eingebettete Systeme

Eingebettete Systeme (engl.: Embedded Systems) finden in vielen Bereichen des täglichen Lebens ihre Anwendung. Oftmals rückt die Nutzung dieser Systeme gar nicht ins Bewusstsein, weil sie subtil und unscheinbar ihren Dienst verrichten. Daher lautet eine erste allgemeine Definition zu *eingebetteten Systemen* wie folgt [Erg05a]:

> ″Embedded Systems *ist der englische Fachbegriff für eingebettete (Computer-)Systeme, die - weitestgehend unsichtbar - ihren Dienst in einer Vielzahl von Anwendungsbereichen und Geräten versehen, wie z.B. in Flugzeugen, Autos, Kühlschränken, Fernsehern, DVD-Playern oder allgemein Geräten der Unterhaltunselektronik.*″

Wie in Abbildung 3.1 dargestellt, wird das eingebettete System in die Umgebung integriert, in der es seine Funktionen und Dienste zur Verfügung stellen soll. Wichtige Bestandteile eines derartigen Systems sind die Prozessoren und Speicherstrukturen, welche noch gesondert betrachtet werden, da ihre Beschaffenheit besonderen Einfluss, gerade auf den Bereich der Datenbankmanagementsysteme hat. Die Standard-Komponente stellt die Funktion des Systembestandteils dar. Sie kann soft- und/oder hardware-technisch realisiert sein. ASICs (auch: **A**pplication-**S**pecific **I**ntegrated **C**ircuits) stellen eine spezielle Form der Implementierung dar, bei der die Hardware eigens für einen konkreten Anwendungsfall entwickelt wird.

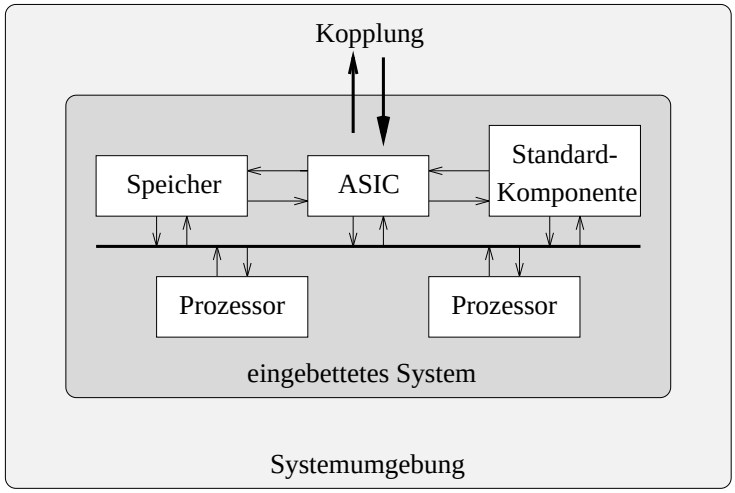

Abbildung 3.1: Schematische Darstellung eines eingebetteten Systems nach [Thi03])

Einen anderen Implementierungsansatz verfolgen FPGAs (auch: **F**ield-**P**rogrammable **G**ate-**A**rray). Dabei handelt es sich um universell einsetzbare Hardware, die für den speziellen Anwendungsfall programmiert und konfiguriert wird. FPGAs ermöglichen die Kombination von Soft- und Hardware-Komponenten. Darüber hinaus ist es möglich, die zu entwickelnde Funktionalität in einer reinen Software-Lösung zu implementieren. Die Software wird dann auf der Basis von Prozessoren ausgeführt.

Die Entwicklung von eingebetteten Systemen wird maßgeblich davon bestimmt, welche Bedingungen in der Zielumgebung vorherrschen. Bei einem System, dass einem mobilen Einsatz (beispielhaft: Betrachtungen treffen auf viele Anwendungsumgebungen zu) zugeführt werden soll, ergeben sich gleich mehrere Anforderungen, die erfüllt werden müssen. Zunächst wird eine lange netzunabhängige Nutzung angestrebt. Daher müssen

die integrierten Komponenten energiesparend arbeiten. Sparsame Komponenten sind in den meisten Fällen nicht so leistungsfähig wie Komponenten mit hohem Stromverbrauch. Der Leistungsbegriff hat verschiedene Dimensionen, wobei die ersten beiden in Abschnitt 3.1.1 und 3.1.2 genauer betrachtet werden:

- Beschaffenheit des Speichers (primär, sekundär): Kapazität, Geschwindigkeit, Ausfallsicherheit, Leistungsaufnahme
- Rechenleistung: z.B. Rechengeschwindigkeit der Hardware bei Ausführung von systemtypischen Anwendungen
- Eingabe- und Ausgabeoptionen: z.B. Displays(mobile Systeme haben meist kleine und unübersichtliche Displays) und Tastaturen (Beispiel PDA: keine leistungsfähige Tastatur, meist Stifteingabe))

Soll das System in Addition noch möglichst klein und leicht sein, so bedarf es weiterer Kompromisse in Bezug auf die Systemzusammenstellung (ein grosser und schwerer Akku wirkt der Forderung nach einem leichten Gerät entgegen). Der Platzbedarf ist eine weitere Grösse, die den Handlungsspielraum bei der Systemkonfiguration einschränkt. Weitere denkbare Einflussgrössen sind Geräuschentwicklung[1], Robustheit[2] oder der Kostenfaktor[3].

Ziel der Entwicklung ist es, die benötigte Systemleistung unter Beachtung der leistungsbeeinflussenden Faktoren bereitzustellen. Gerade in Hinblick auf den auszuführenden Programmcode ergeben sich einige Optionen, um die Leistungsreserven des Systems zu schonen. Beispielsweise kann leistungsoptimierter Code auf deutlich langsamerer Hardware arbeiten, als dies ohne Optimierung möglich ist. Sehr gutes Optimierungspotential besitzt Assembler-Code. Jedoch ist solch ein maschinennaher Programmcode schwer wart- und wieder verwendbar. Bei häufig genutzten Programmroutinen bietet sich die Integration der Routinen in die Hardware an. Auf diese Weise kann die entsprechende Funktion schnell und stromsparend ausgeführt werden.

Da eingebettete Systeme den obig beschriebenen Restriktionen unterliegen, ist es bei der Systementwicklung besonders wichtig, ein ausgewogenes Verhältnis aus Soft- und Hardware-Bestandteilen zu erhalten. Dies wird in Abbildung 3.2 deutlich. Während eine weitestgehend durch Software realisierte Implementierung ein Höchstmaß an Flexibilität bei erhöhtem Energieverbrauch bedeutet, kann eine hardware-lastige Implementierung durch Leistung überzeugen. Jedoch ist eine derartige Lösung aufgrund hoher Entwicklungs- und Produktionskosten nur bei grösseren Stückzahlen sinnvoll, da die resultierende Lösung so speziell ist, dass sie in den meisten Fällen in anderen Systemen nicht wieder verwendet werden kann. Aufgrund des hohen Entwicklungsaufwands ist zudem mit einer längeren Zeitspanne bis zur Marktreife (auch: Time-To-Market) zu rechnen.

[1] Herausforderung bei der Kühllösung (eventuell Passivkühlung)
[2] beeinflusst beispielsweise den Gewichtsfaktor, da stabile Gehäuse schwerer sind
[3] Kleine, leichte, sparsame und schnelle Systeme sind teuer.

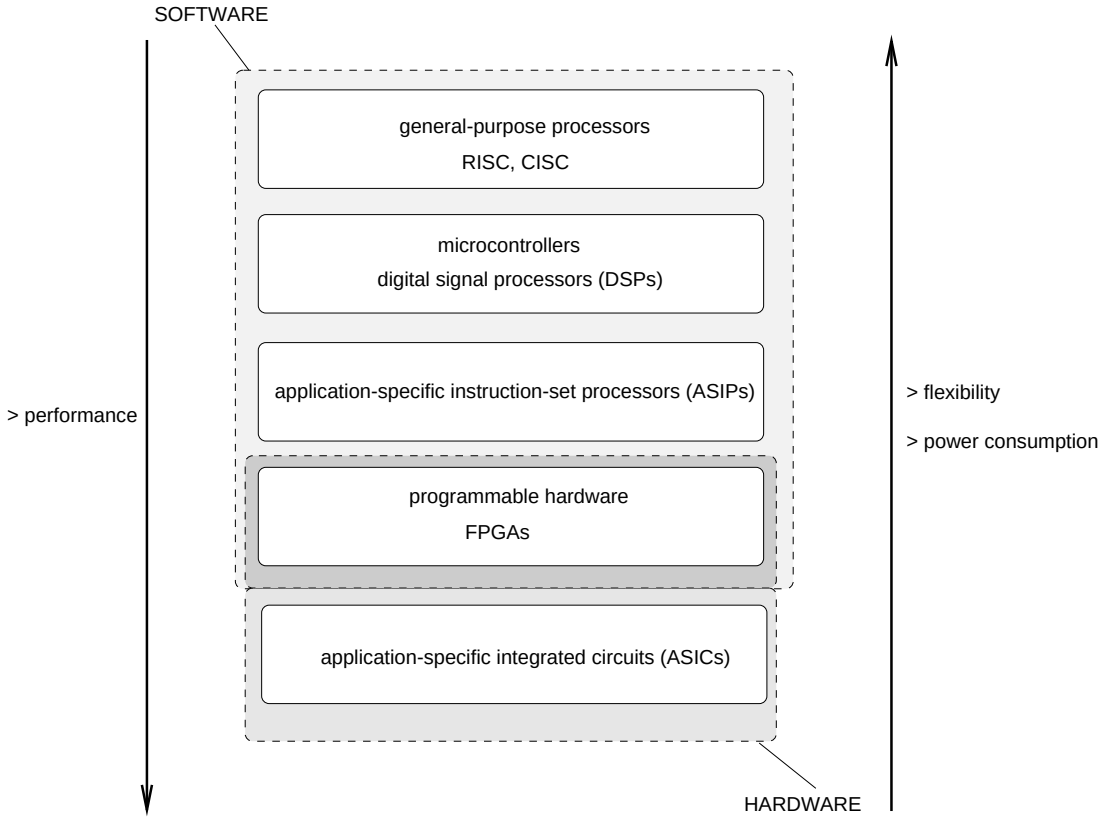

Abbildung 3.2: Implementierung Hardware vs. Software nach [Pla01])

Die Leistungsreserven von eingebetteten Systemen können auch dadurch geschont werden, dass nur die *minimal* erforderlichen Funktionen implementiert werden. Hier setzt die merkmals- und aspektorientierte Programmierung (in Verbindung mit Programmfamilien) an, durch die eine feingranulare Software-Entwicklung möglich wird. Nähere Erläuterungen zu diesem Lösungsansatz erfolgen im weiteren Verlauf dieser Arbeit.

3.1.1 Prozessorarchitekturen in eingebetteten Systemen

Aufgrund der ressourcenspezifischen Restriktionen in eingebetteten Systemen kommt der Wahl einer geeigneten Prozessorarchitektur besondere Bedeutung zu. Die Klasse der *Mikrocontroller* beschreibt Prozessoren, die auf einen speziellen Anwendungsbereich zugeschnitten werden. Häufig liegt ihr Einsatzgebiet in der Steuerung von Prozessen. Die Wortbreite der Mikrocontroller kann, je nach Anwendung, zwischen 4-64 Bit variieren. Ihr Befehlssatz (engl.: Instruction Set) beinhaltet Zugriffsfunktionen (Eingabe/Ausgabe) auf vorhandene Peripherie [Thi03]. Ein Minuspunkt von Mikrocontrollern ist die geringe Performanz. Durch die anwendungsbasierte Konfiguration der Prozessoren ergibt sich dem gegenüber eine minimale Lösung, die sehr sparsam mit vorhandenen Ressourcen umgeht.

In vielen Anwendungsfällen ist es trotz Ressourcen-Knappheit wichtig, ein möglichst leistungsfähiges System bereitzustellen. Gerade im Workstation-Bereich oder in Systemen mit rechenintensiven Prozessen (z.B. Multimedia, Graphik, Datenbanken, Kommunikation) ist die Leistungsfähigkeit des Prozessors ein entscheidendes Qualitätsmerkmal. In solchen Fällen bietet sich die Verwendung von *Mikroprozessoren* (Beispielarchitekturen: RISC, CISC) an. Sie zeichnen sich unter anderem dadurch aus, dass sie nebenläufige Prozesse (Stichwort: Parallelität) unterstützen. Aufgrund ihrer Performanz haben Mikroprozessoren gegenüber Mikrocontrollern eine erhöhte Leistungsaufnahme. Darüber hinaus ist es oftmals schwierig einen Prozessor zu wählen, der genügend Leistungsreserven bietet und gleichzeitig ressourcen-schonend operiert.

ASIPs (auch: **A**pplication-**S**pecific **I**nstruction **S**et **P**rocessor) besitzen speziell auf die Anwendung zugeschnittene Befehlssätze, Funktionseinheiten, Register und Verbindungsstrukturen [Thi03]. Die Vorteile einer an den Anwendungsfall orientierten Prozessorarchitektur sind [Thi03]:

- *Flexibilität*: allgemein programmierbare Prozessoren sind meist recht langsam
- *Kosten*: Platzersparnis gegenüber allgemein programmierbaren Prozessoren, aufgrund der Einsparung von Pins und vereinfachter Schnittstellen- und Speicherarchitektur
- *Leistungsverbrauch*: geringe Leistungsaufnahme (Verminderung von thermischen Problemen, Schonung der Akkus) ermöglicht Einsatz in mobilen Lösungen

Den genannten Vorteilen steht/stehen ein verringertes Anwendungsspektrum (aufgrund der Spezialisierung), für welches diese Prozessorarchitektur in Frage kommt, und geringere Leistungsreserven (für zukünftige Erweiterungen) gegenüber. *Digitale Signalprozessoren* (kurz: DSP) gehören ebenfalls der Klasse der anwendungsspezifischen Prozessoren an. Sie finden häufig Anwendung in Programmen mit hoher Nebenläufigkeit, hohem Datenflussaufkommen und/oder komplexen, regelmässigen arithmetischen Operationen auf Feldern. Die Wortlänge dieser Prozessoren variiert zwischen 16-64 Bit. Ein wesentlicher Vorteil liegt in der geringen Leistungsaufnahme.

Aufgrund der erläuterten Eigenschaften der einzelnen Prozessoren erfolgt die Auswahl für den Anwendungsfall. Speziell für die Transaktionsverwaltung sind Prozessoren, die Nebenläufigkeit unterstützen, von besonderem Interesse.

3.1.2 Speichertechnologien in eingebetteten Systemen

Die Vielfalt an Speichertechnologien im Bereich der eingebetteten Systeme ist aufgrund der verschiedenartigen Einsatzmöglichkeiten solcher Systeme naturgemäss sehr gross. Die Eigenschaft der Ressourcen-Knappheit ist das primäre Auswahlkriterium bei der Entscheidungsfindung für eine Speichertechnologie. Ähnlich zur Wahl der Prozessorarchitektur, kommt es in dieser Disziplin darauf an, den richtigen Kompromiss zwischen

Performanz, Leistungsaufnahme und Platzbedarf zu finden. Dabei müssen im Einzelfall häufig noch weitere Faktoren berücksichtigt werden. Es liegt in der Natur von DBMS, dass diese zum Zweck der Datenspeicherung, -löschung, -manipulation und des Datenabrufs ausserordentlich häufig mit der genutzten Speichertechnologie in Interaktion treten. Das verstärkt die Forderung nach einer optimalen Antwort auf die Frage der Speicherwahl zusätzlich.

In Bezug auf DBMS ergibt sich zunächst eine Speicherhierarchie, die beginnend mit der schnellsten Ebene, wie folgt gegliedert ist:

1. *Primärspeicher*: oft gleichbedeutend mit Hauptspeicher, sehr schnell, aber teuer
2. *Sekundärspeicher*: beispielsweise Festplatten, Speicherkarten, langsam, aber günstig
3. *Tertiärspeicher*: Magnetbänder, sehr langsam, sehr günstig

An dieser Stelle wird auf eine Analyse der Speichertechnologien in Prozessoren verzichtet, da die Entscheidung für eine Technologie mit der Wahl der genutzten Prozessorarchitektur fällt. Dabei ist jedoch zu beachten, dass nicht jede Hardwarearchitektur gleichermaßen Funktionen[4] für die Speicherverwaltung zur Verfügung stellt. Daraus resultierende Einschränkungen können einen grossen Einfluss auf die Datensicherheit haben.

Die erste Systemklasse umfasst Geräte die ausschliesslich über Primärspeicher verfügen. Hier wird der Primärspeicher zusätzlich zur ″persistenten″ (Persistenz: nur solange das System mit Strom versorgt wird) Speicherung der Daten genutzt. Er fungiert also gleichzeitig als eine Art Sekundärspeicher. Der Vorteil dieser Lösung liegt in der hohen Geschwindigkeit bei Schreib- und Leseoperationen. Zudem lassen sich derartige Speichermedien byte-weise auslesen und beschreiben, wodurch die Verwendung von effizienten Zugriffsstrukturen (z.B.: Bäumen) nicht ausgebremst wird[5]. Nachteilig ist die Notwendigkeit einer permanenten Stromversorgung[6] des flüchtigen Speichers (verglichen mit Festplatten ist der Energiebedarf recht gering). Darüber hinaus ist die Speicherkapazität aufgrund des Kostenfaktors begrenzt. Daher ist es besonders wichtig den Overhead eines ″aufgeblähten″ (Forderung nach einer minimalen Lösung) DBMS zu verhindern.

In anderen Systemen wird ″echte″ Persistenz durch die Verwendung von nichtflüchtigen Speichern realisiert. Dazu zählen in erster Linie Festplatten, die in punkto Schreib- und Lesezugriff nicht mit Hauptspeicherlösungen mithalten können. Zudem

[4] z.B.: fehlende *MMU* (Memory Management Unit) zur Steuerung der Speicherzugriffe

[5] Entartete Binärbäume führen im Zusammenhang mit blockorientierten Medien beispielsweise häufig zu ineffizienten Speicherzugriffen und Speicherverschwendung, da die Daten über viele Blöcke verteilt abgelegt werden.

[6] Häufig werden derartige Systeme, zur Überbrückung von Ernergieversorgungsengpässen, durch eine alternative Stromquelle gepuffert.

kann die blockorientierte Speicherstruktur zu Problemen mit bestimmten Zugriffsstrukturen führen. Dafür sind sie in der Lage, eine deutlich höhere Speicherkapazität bereitzustellen. Weil die Festplattenaktivitäten beim Schreiben und Lesen grösstenteils mechanischer Natur sind, ist ihr Energiebedarf recht hoch. Um die Festplatte im Ruhezustand halten zu können, bietet sich eine Pufferung der Daten im Hauptspeicher an. Trotz der genannten Probleme kommen in vielen eingebetteten DBMS-Lösungen Festplatten zum Einsatz [Nys02].

So genannte *Flash-Speicher* vermeiden mechanische Schreib-/Lesekopfbewegungen (Ergebnis ist ein Stromverbrauch von wenigen Milliampere). Zusätzlich sind die Speicherzugriffe deutlich schneller zu bewerkstelligen als bei Festplatten. Sie ermöglichen das byte-weise Auslesen und blockweise Schreiben (nach vorheriger Löschung) von Daten (dadurch sind zumindest effektive Lesezugriffe auf Byte-Ebene gewährleistet). Jedoch beschränkt sich die Lebenszeit eines Flash-Speichers auf 100.000 - 1.000.000 Schreibzugriffe. Da gerade Datenbanken viele Schreibvorgänge tätigen, ist die Eignung solcher Speichermedien in diesem Anwendungsbereich eingeschränkt.

Eine weitere Form der Datenspeicherung stellen vernetzte Systeme dar. Dort erfolgt die Datenspeicherung auf entfernten Systemen, die mit dem Client verbunden sind. Hier besteht ebenfalls das Problem der hohen Verzögerung von Schreib- und Leseoperationen. Ausserdem ist davon auszugehen, dass der Betrieb einer Netzwerkverbindung zusätzlich Strom benötigt.

3.1.3 Datenhaltung in eingebetteten Systemen

In diesem Abschnitt soll der Stellenwert einer datenbankbasierten Datenhaltung diskutiert werden. Die gewählten Beispiele sollen nur einen kurzen Abriss vorhandener Anwendungsszenarien geben und keineswegs dem Anspruch der Vollständigkeit genügen.

Automobilbau - Elektronik

Automobile beherbergen eine grosse Menge an elektronischen Bauteilen. Dazu gehören Navigationssysteme, Motorsteuerung, Fahrwerkssteuerung[7] oder Steuerung des Bremssystems[8]. Daneben finden sich in einem Fahrzeug viele verschiedene Sensorsysteme, die alle denkbaren Fahrzeugdaten messen und verarbeiten.

Den meisten integrierten Systemen ist eine Eigenschaft gemein - Echtzeitfähigkeit. Die einzelnen Systeme müssen sich in Bruchteilen einer Sekunde auf neue Situationen einstellen und angemessen darauf reagieren. Dem entsprechend muss auch der Abruf benötigter Daten zur Berechnung der Reaktion schnellstmöglich erfolgen. Auf der anderen Seite ist eine langfristige Speicherung der Daten in den meisten Fällen nicht erforderlich, da die Einschätzung einer neuen Fahrsituation auf aktuell gemessenen Daten beruht.

[7] z.B.: ESP (**E**lektronisches **S**tabilitäts**p**rogramm)
[8] z.B.: ABS (**A**nti-**B**lockier-**S**ystem)

Aus diesem Grund müssen die verwendeten Speichertechnologien keine hohen Speicherkapazitäten bereitstellen, sondern in punkto Geschwindigkeit verlässliche Partner sein. Dem gegenüber gibt es aber auch Daten, die langfristig gespeichert werden müssen. Dabei handelt es sich um Setup-Daten, die eine optimale Einstellung der verschiedenen Bord-Systeme garantieren.

Unterhaltungselektronik - Haushaltsgeräte

Ähnlich der Kfz-Elektronik werden die Funktionen dieser Geräteklasse vor dem Nutzer versteckt[9] oder durch ein Benutzer-Interface zugänglich gemacht.

Der Umfang an einstellbaren Funktionen wird zu Gunsten einer nutzerfreundlichen Bedienung auf die notwendige Menge reduziert, wodurch auch die zu speichernde Datenmenge[10] sehr gering ausfällt. Auch kann in solchen Systemen in den meisten Fällen auf Echtzeitfähigkeit verzichtet werden, da eine verzögerte Ausführung der Befehle toleriert werden kann. Daher muss die Datenhaltungskomponente keinen gesteigerten Anforderungen bezüglich Geschwindigkeit und Kapazität genügen.

Mobile Geräte

Der Klasse der mobilen Geräte gehört ein funktional weit gefächertes Spektrum von Devices an, die vielfältige Anforderungen an die Datenhaltungskomponente stellen. Schon allein ein PDA (engl.: **P**ersonal **D**igital **A**ssistant) muss je nach Anwendungsbereich unterschiedlichen Anforderungen genügen. Soll in das Gerät eine komfortable Telefonbuch-, Termin- oder Aufgabenverwaltung integriert werden, so wäre es sicherlich sehr ungünstig, wenn diese Daten aufgrund eines leeren Akkus verloren gehen. Um dies zu verhindern, gibt es zwei wesentliche Strategien, die verfolgt werden können. Eine denkbare Lösung ist die permanente Speicherung auf einem geeigneten Medium (Festplatte, Flash-Speicher). Eine Alternative dazu stellt die Synchronisation mit dem PC dar.

Neben der Art der Speicherung hängt auch die verwendete Speichergrösse sehr stark vom Einsatzzweck ab. Während Termine nicht viel Speicherplatz beanspruchen, belegen ausgewachsene Datenbanken[11] viel Speicherplatz. Dem entsprechend leistungsfähig muss das mit den Datenbanken betriebene DBMS sein, was zusätzliche Rechenleistung und Speicherkapazität erfordert. Die Frage nach der Geschwindigkeit einer Datenhaltungskomponente und der beteiligten Bauteile wird vom Einsatzgebiet beantwortet.

[9] sie verrichten ihre Arbeit im Hintergrund und ohne Eingriffsmöglichkeit im Fehlerfall

[10] vorwiegend Setup-Einstellungen der Geräte (Zeit, Sprache, Weckfunktion, etc.)

[11] In Unternehmen werden zunehmend mobile Geräte für den Abruf und die Manipulation von Daten genutzt (beispielsweise zentral über WLAN-Lösungen oder dezentral auf dem Gerät selbst).

3.2 Datenbankmanagementsysteme

Ein Datenbanksystem setzt sich aus den zu verwaltenden Datenbanken und einem *Datenbankmanagementsystem* zusammen. Eine Definition des Datenbankmanagementsystem-Begriffes kann wie folgt gegeben werden [Pre00]:

> *"A software system that facilitates (a) the creation and maintenance of a database or databases, and (b) the execution of computer programs using the database or databases."*

Edgar Frank Codd entwickelte ab Anfang der 80er Jahre verschiedene Regeln (bestehend aus 12 wesentlichen Regeln, die Codd in [Cod90] weiter differenziert), nach denen ein allgemeines DBMS aufgebaut sein sollte. Diese Forderungen betreffen unter anderem die Form der Dateiorganisation, Konsistenzkontrolle oder Datensicherung. Bis heute gibt es aber kein DBMS, das alle Forderungen erfüllt.

3.2.1 DBMS - Allgemein

Egal wie ausgeprägt der Funktionsumfang eines realen DBMS auch ist, so lässt sich dessen Architektur doch auf ein allgemeines Modell zurückführen, da es auf Standardfunktionen zurückgreift, die in den meisten DBMS verankert sind. Die einzelnen Funktionen lassen sich nach Aufgabenbereichen[12] einteilen. Eine der bekanntesten Formen der Architekturbeschreibung erfolgt in dem von Härder entwickelten *5-Schichten-Modell* [Hä87]. Abbildung 3.3 veranschaulicht die einzelnen Schichten und deren Aufgaben aus dem Blickwinkel der Funktionen. Dabei fällt auf, dass die dargestellten Schichten (beginnend beim Datensystem) die anfänglich abstrahierten Anfragen und Änderungen auf Datenbankmodellebene mit jeder Schicht, schrittweise, auf die Ebene der konkreten Speichermedien herunter brechen [Saa99].

Die Schichten sind dabei durch verschiedene Schnittstellen verbunden, welche die durch die Vorgängerebene transformierten Informationen weitergeben. In der Mengenorientierten Schnittstelle erfolgt die Kommunikation zwischen DBMS und externen Applikationen. Das Datensystem ist unter anderem dafür zuständig, dass die Ausdrücke der, in der Mengenorientierten Schnittstelle realisierten, Datenmanipulationssprache (z.B. SQL), zur weiteren Verarbeitung in den unteren Schichten, korrekt übersetzt werden. Weiterhin erfolgen hier Zugriffskontrolle, Integritätskontrolle und Zugriffspfadwahl. Die aufbereiteten Informationen können über die Satzorientierte Schnittstelle an das Zugriffssystem weitergereicht werden. Dies beinhaltet eine einheitliche Verwaltung interner Tupel (unabhängig von relationsabhängiger Typisierung). Hervorzuheben ist die Realisierung von Teilen der Transaktionsverwaltung (Mehrbenutzerbetrieb), wobei Sperrverwaltung und Log/Recovery im Speichersystem beheimatet sind. Die Interne Satzschnittstelle realisiert die Speicherstrukturen der Zugriffspfade (Hash-Tabellen oder B+-Bäume) [Saa99].

[12] Wobei eine klare Abgrenzung oftmals schwierig zu realisieren ist, und darum, auch zum besseren Verständnis, geringfügig idealisiert erfolgen muss.

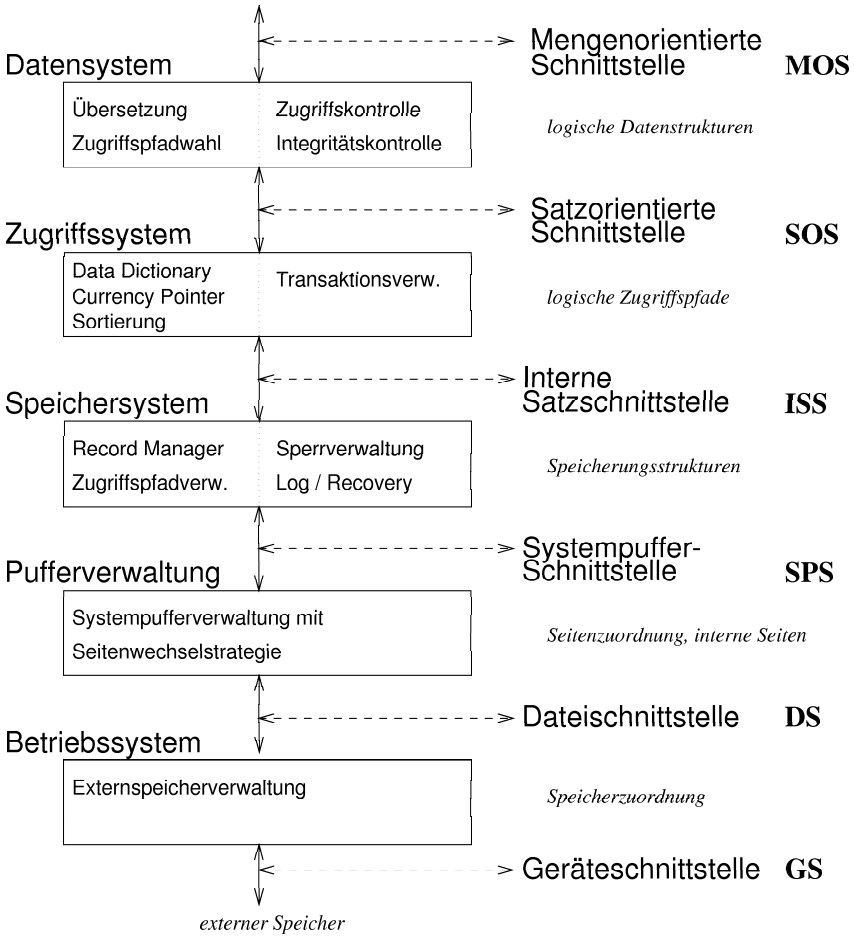

Abbildung 3.3: Funktionale Sicht auf die 5-Schichten-Architektur nach [Hä87] aus [Saa99]

Dem Speichersystem kommt die Aufgabe zu, die in der Internen Satzschnittstelle definierten Speicherstrukturen auf interne, seiten-orientierte Konzepte (mit Adresszuweisungen) abzubilden. Über die Systempufferschnittstelle erfolgt die Kommunikation mit der Pufferverwaltung. Diese bildet die internen Seiten auf blockorientierte Speicherkonzepte ab (realisiert auf Dateischnittstelle). Die weiteren Umsetzungsmechanismen, für konkrete Speicherung/Abruf der Daten übernimmt das Betriebssystem, welches nicht Bestandteil des DBMS ist [Saa99].

3.2.2 DBMS - Transaktionsverwaltung

Da die Transaktionsverwaltung ein recht komplexer Vorgang ist, der viele Prozesse in sich vereint, überrascht es nicht, dass verschiedene DBMS-Komponenten jeweils Teilaufgaben der Verwaltung übernehmen. Da die 5-Schichten-Architektur eher ungeeignet erscheint, um die Belange der Transaktionsverwaltung abzugrenzen, wird eine komponentenbasierte Darstellung gewählt (Abbildung 3.4).

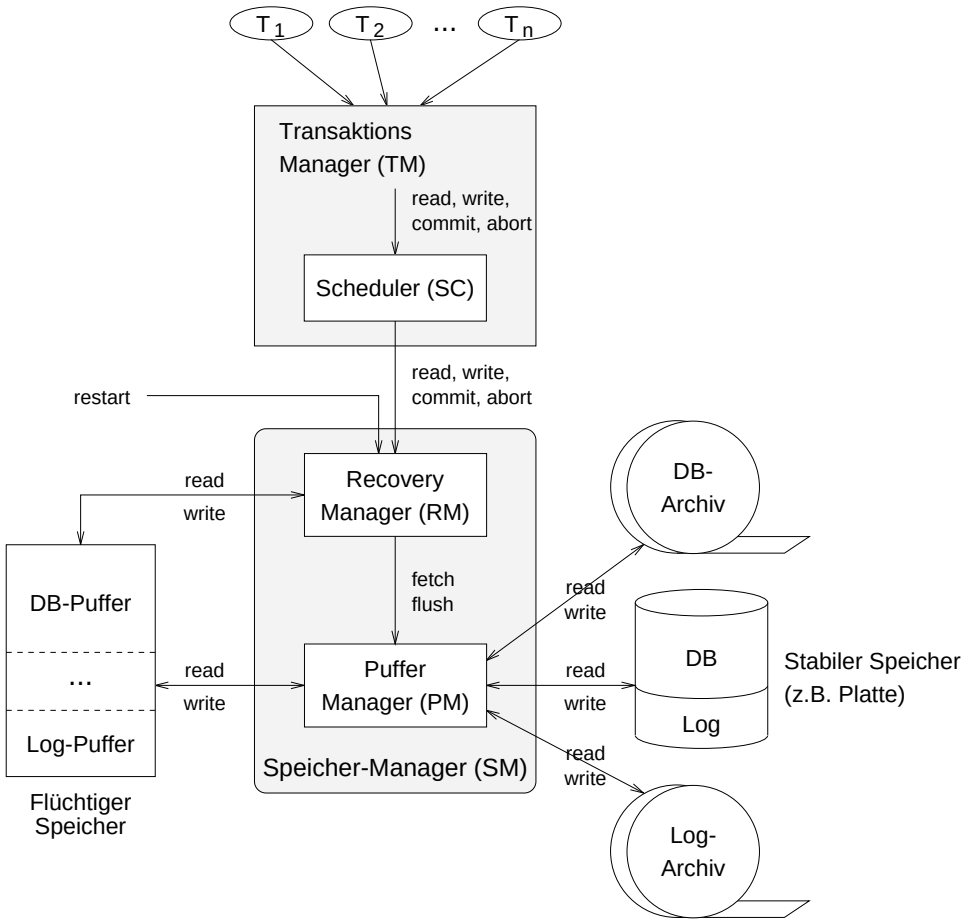

Abbildung 3.4: DBMS-Komponenten im Zusammenspiel aus [Saa99])

Der Transaktions-Manager (mit seiner Hauptkomponente, dem Scheduler) nimmt die von den Nutzern[13] in Auftrag gegebenen Transaktionen entgegen. Im Sinne eines effektiven Mehrbenutzerbetriebs eines Datenbanksystems hat er die Aufgabe, die eingehenden Transaktionen und deren Operationen in ihrer Ausführungsreihenfolge so zu mischen, dass die lokale Operationsreihenfolge erhalten bleibt und Konflikte (aufgrund von konkurrierenden Transaktionsoperationen auf gleichen Daten) entweder vermieden oder (nachträglich) gelöst werden [Saa99]. Dabei verfährt der Transaktions-Manager nach den ACID-Eigenschaften[14] (in der Praxis werden diese Eigenschaften zum Zweck eines höheren Transaktionsdurchsatzes häufig ″aufgeweicht″).

Transaktionsoperationen werden in der vom Scheduler festgelegten Reihenfolge an den Recovery-Manager weitergereicht, der deren Ausführung administriert. Er überwacht

[13]Der Nutzer steht in den meisten Fällen nicht direkt mit der Datenbank in Verbindung, sondern tätigt seine Eingaben in speziellen Anwendungen, die über definierte Schnittstellen mit der Datenbank in Verbindung stehen.

[14]ACID: **A**tomicity, **C**onsistency, **I**solation, **D**urability

den Puffer-Manager (Datenauslagerung, Datenpufferung) und manipuliert die im Puffer enthaltenen Daten gemäss der eingehenden Transaktionsoperationen. Dabei trägt er ebenfalls dafür Sorge, dass die ACID-Eigenschaften eingehalten werden. In dieser Funktion übernimmt er im Fehlerfall die Wiederherstellung der Daten (auch: Herbeiführen eines konsistenten Datenbankzustands). Als Grundlage für Wiederherstellungsmaßnahmen des Recovery-Managers dienen, je nach Recovery-Strategie, Log-Bucheinträge, aus denen sich ein aktueller konsistenter Datenbankzustand rekonstruieren lässt. Das ″Fahren″ der Logging-Protokolle übernimmt ebenfalls der Recovery-Manager.

Durch die Beschreibung der einzelnen Komponenten der Transaktionsverwaltung werden die Grenzen der betrachteten Domäne klar abgesteckt. Dies ist Grundvoraussetzung für die spätere Merkmalsanalyse.

Transaktionsverwaltung in eingebetteten Systemen

In Abschnitt 3.1.3 wurden Anforderungen an die Datenhaltung (allgemein) in unterschiedlichen eingebetteten Systemen betrachtet. Nachfolgendes Thema ist die Erfassung von Anforderungen rund um die Transaktionsverwaltung.

Essentiell für alle weiterführenden Betrachtungen ist die Beantwortung der Frage nach der Notwendigkeit von Mehrbenutzerbetrieb. Wird Mehrbenutzerbetrieb angeboten, so muss die Architektur des eingebetteten Systems nebenläufige Prozesse unterstützen, da dies Grundvoraussetzung für die verschränkte Ausführung konkurrierender Transaktionen ist. Es muss demnach ein Prozessor vorhanden sein, der genügend Leistungspotenzial für Nebenläufigkeit besitzt. Um die vorhandenen Ressourcen zu schonen, kann beispielsweise die maximale Anzahl nebenläufiger Transaktionen begrenzt werden. Soll das DBMS nur im Einzelnutzerbetrieb laufen, so kann auf Methoden zur Steuerung verschränkter Transaktionsausführungen verzichtet werden.

Der nächste Punkt betrifft die vorhandene Speicherarchitektur. Handelt es sich um ein System, das keine persistente Speicherung von Daten zulässt (Stichwort: Hauptspeicherlösung, Primärspeicher), so kann in den meisten Fällen auf die Implementierung von Recovery- und Logging-Mechanismen verzichtet werden, da die Recovery-Informationen im Fehlerfall (z.B. leerer Akku, Stromausfall) ebenfalls verloren gehen. Viel wichtiger ist in diesem Zusammenhang der Einsatz geeigneter Backup-Strategien, um den Systemzustand eines eingebetteten Systems auf externen Medien zu sichern. Ergänzend gilt für alle speicherspezifischen Operationen, die von der Transaktionsverwaltung angestossen werden, dass sie möglichst sparsam mit vorhandenem Speicherplatz umgehen sollten.

Im Bereich von Echtzeitlösungen ist eine priorisierte Ausführung von Transaktionen wünschenswert, um die benötigten Daten innerhalb des vorgegebenen Zeitfensters bereitzustellen. Dies kann z.B. durch die Verwendung von Zeitmarkenverfahren bewerkstelligt werden. Allgemein ist für die Transaktionsverwaltung, genau wie für jede andere Anwendung, im Bereich der eingebetteten Systeme die minimalste Lösung anzustreben, die den gestellten Anforderungen gerecht wird. Darüber hinaus ist darauf zu achten, dass die implementierten Mechanismen möglichst effizient arbeiten.

3.3 Ansätze zur Implementierung konfigurierbarer DBMS

Neben dem in dieser Diplomarbeit verfolgten Ansatz der Entwicklung von konfigurierbaren DBMS, mittels merkmals- und aspektorientierter Programmierung, gibt es eine Reihe weiterer interessanter Projekte, die sich mit dem Thema beschäftigen. Wie angekündigt, sollen in diesem Abschnitt schwerpunktmmässig das *COMET*-DBMS und *PLENTY* (auch: **P**arallel **E**xecution of **N**ested **T**ransactions on Plent**y** of Processors) genauer betrachtet werden (insbesondere in Hinblick auf die Realisierung der Transaktionsverwaltung). Weitere Ansätze werden im Rahmen dieser Arbeit kurz erwähnt.

3.3.1 COMET-DBMS

Beim *COMET-DBMS* handelt es sich nach Nyström et. al [Nys03] um eine experimentelle Datenbankplattform, die für den Einsatz in, von Ressourcenknappheit geprägten, Fahrzeugkontrollsystemen entwickelt wurde. Das DBMS ist für den Betrieb von Echtzeitanwendungen ausgelegt. Es wird generell zwischen Kontrollsystemen unterschieden, die statisch installiert werden und solchen, die flexibel anpassbar sein sollen und eine priorisierte Ausführung ermöglichen. Allein diese Klassifizierung bedingt ein möglichst flexibel anpassbares DBMS. Den Forderungen versucht COMET gerecht zu werden, indem es nach dem Baukastenprinzip für einen konkreten Anwendungsfall zusammengestellt wird. Dabei bedient es sich zweier verschiedener Arten von Bauteilen: Aspekten und Komponenten.

Als Grundlage für die Entwicklung von COMET wurde *ACCORD* (**A**spe**c**tual **Co**mponent-Based **R**eal-Time System **D**evelopment) verwendet. Dabei handelt es sich um ein Konzept, das die Kombination von Komponenten und Aspekten unterstützt. Auf der Basis von ACCORD wird ein System in seine funktionalen Bestandteile (Komponenten und Aspekte) zerlegt. Dazu werden die Aspekte in drei Kategorien unterteilt: Application Aspects[15], Run-Time Aspects[16] und Composition Aspects[17]. Zur Beschreibung, wie eine Komponente im Kontext eines Echtzeitsystems aufgebaut sein sollte, wird das **R**eal-**T**ime **Co**mponent **M**odel (kurz: RTCOM) verwendet.

In Abbildung 3.5 sind die drei, in ACCORD beschriebenen, aspektspezifischen Dimensionen dargestellt, aus denen sich eine Komponente zusammensetzt. Zu erkennen ist, dass sich der funktionale Bestandteil der Komponente aus statischen Funktionen (Mechanism) und modifizierbaren Teilen (Policy) zusammensetzt. Der Policy-Part kann von funktionsspezifischen Aspekten verändert werden. Der Composition-Part beinhaltet Informationen zur Kompatibilität der Komponente zu externen Einheiten (Betriebssystem, Ressourcen, Aspekten, etc.). Die formale Beschreibung des Real-Time Component-Model kann in [Nys03] nachgeschlagen werden.

[15] Bedeutung: Änderung des funktionalen Verhaltens der Anwendung
[16] Bedeutung: Änderung des Verhaltens zur Laufzeit
[17] Bedeutung: Steuerung der Kombinierbarkeit von Aspekten und Komponenten

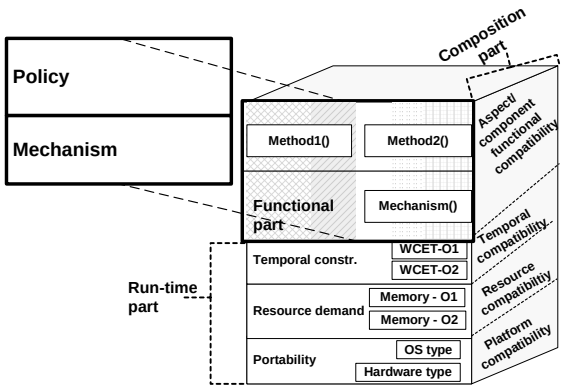

Abbildung 3.5: RTCOM aus [Nys03])

Architektur

Zur Identifikation der wichtigsten Bestandteile, aus denen sich das COMET-DBMS zusammensetzen soll, wurde eine Analyse der einzelnen Aktivitäten eines DBMS durchgeführt. Das Ergebnis der Analyse ist die in Abbildung 3.6 dargestellte Architektur. Die Entwickler weisen gleichzeitig darauf hin, dass das COMET-DBMS in der Zukunft um weitere Funktionalitäten erweitert werden soll.

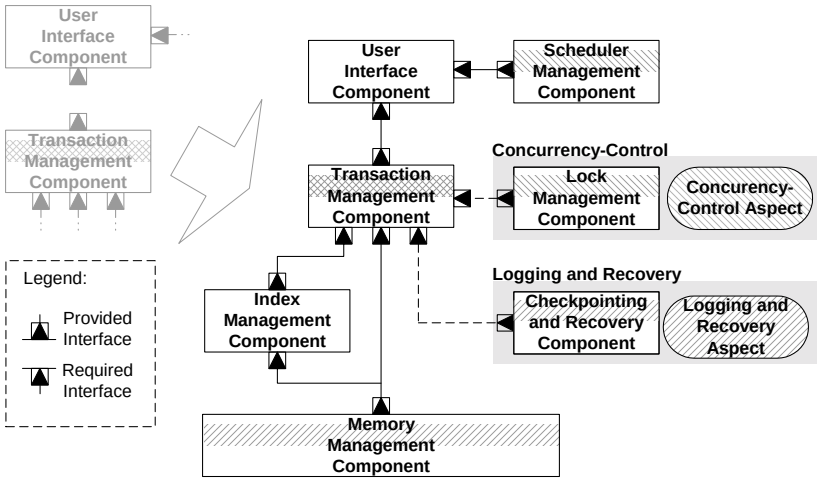

Abbildung 3.6: Architektur des COMET-DBMS aus [Nys03])

Bezogen auf die lokalisierten Aktivitäten wurde eine Einteilung dieser bezüglich ihrer Umsetzung vorgenommen. Dabei wurden klar abgrenzbare Aktivitäten in Form von Komponenten (rechteckige Umrandung) realisiert und Funktionen, die in verschiedenen Programmteilen genutzt werden, als Aspekte (runde Umrandung). Die Art des in den Komponenten verwendeten Aspekt-Codes wird durch die Ausrichtung der Schraffierung gekennzeichnet. Des Weiteren sind in Abbildung 3.6 die Beziehungen der Komponenten

untereinander in Form der Komponentenschnittstellen dargestellt. Die *User Interface Component* stellt die Schnittstelle[18], zu der auf dem DBMS aufsetzenden Anwendung bereit. Ausführungspläne für Transaktionen, die im Zusammenspiel zwischen User Interface Component und *Scheduler Management Component* generiert wurden, gelangen in die *Transaction Management Component*, wo sie gemäss dem Serialisierbarkeitsgedanken bezüglich konkurrierender Transaktionen ausgeführt werden. Mit der Verhinderung von Schreib/Lese-Konflikten bei der Ausführung der Transaktionen wird das *Lock Management Component* beauftragt. Die Umsetzung erfolgt durch die Sperrverwaltung. Die innerhalb der Transaktionen benötigte Daten (Tupel) werden mittels der *Index Management Component* indiziert. Die so ermittelten Tupel Identifier (kurz: TID) enthalten Informationen darüber, an welcher Stelle im Speicher sich die zugehörigen Tupel befinden. In der *Memory Management Component* erfolgen die Datenzugriffe. Weiterhin werden hier die von der Transaction Management Component benötigten Sperrinformationen verwaltet. Damit im Fehlerfall eine Rekonstruktion eines aktuellen und konsistenten Datenbankzustands möglich ist, wird der Datenbankzustand in regelmässigen Abständen archiviert. Die Regelmässigkeit einer derartigen Operation wird durch Checkpoints definiert. Auf der Basis der gesicherten Zustände erfolgt die Wiederherstellung (Recovery). Die benötigten Methoden stellt die *Checkpoint and Recovery Component* bereit.

Wie schon erwähnt, gibt es Belange, deren Funktionalität an mehrere Stellen des DBMS benötigt werden. Dazu gehört der *Concurrency-Control Aspect*. Er stellt sicher, dass die nebenläufigen Transaktionen serialisierbar sind. Daher wird dieser Aspect unter anderem in die Transaction Management Component eingewebt. Zusammen mit der Lock Management Component realisiert sie die Concurrency-Control. Auch verschiedene Funktionen zur Protokollierung und Wiederherstellung lassen sich mittels Aspekten beschreiben und leicht in Komponenten einweben, die auf die Funktionen zurückgreifen wollen. Diese werden im *Logging and Recovery Aspect* zusammengefasst.

Konfiguration

Zu Beginn der Konfiguration des COMET-DBMS werden die Anforderungen formuliert (Abbildung 3.7). Ausgehend von diesen Informationen werden die benötigten Kontrollsysteme, Laufzeiteigenschaften und ihre Beziehungen identifiziert.

Die Modellierung einer für das Szenario geeigneten Datenbank erfolgt mit Unterstützung des *Data Engineering Tools*. Das *Configuration Tool* bedient sich der bis dahin gesammelten Informationen und wählt, darauf aufbauend, die benötigten Aspekte und Komponenten für die COMET-Konfiguration aus. Die resultierende Konfiguration wird, mittels verschiedener *Analysis Tools*, auf Eignung für das Einsatzgebiet untersucht.

[18] Die Schnittstelle beinhaltet eine entsprechende **D**ata **M**anipulation **L**anguage (kurz: DML).

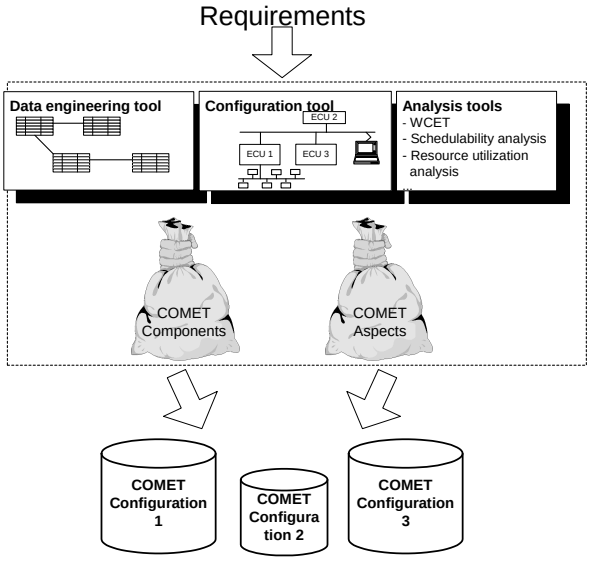

Abbildung 3.7: Konfigurationsprozess des COMET-DBMS aus [Nys04])

3.3.2 PLENTY-System

Hasse betrachtet in [Has95] Möglichkeiten zur Steigerung und Optimierung des Transaktionsdurchsatzes für Datenbanksysteme, bei denen OLTP-Transaktionen[19] und Decision-Support-Transaktionen[20] kombiniert werden. Der verfolgte Lösungsansatz zielt auf eine effektive Ausnutzung von Mechanismen zur Inter- und Intratransaktionsparallelität in Datenbanken ab. Passend zum gewählten Lösungsansatz, bilden Mehrschichten-Transaktionen die theorethische Grundlage für die weiteren Betrachtungen.

Nach dieser knappen Einleitung zum PLENTY-Projekt, soll an dieser Stelle die Beziehung zum Gegenstand der zugrunde liegenden Diplomarbeit hergestellt werden. Für weiter führende Informationen zu konkreten Optimierungsalgorithmen und Modellen sei auf [Has95] verwiesen, da diese nicht im Mittelpunkt der Betrachtungen der gegenwärtigen Arbeit stehen.

Damit die entwickelten Mechanismen zur Inter- und Intratransaktionsparallelität einer Leistungsüberprüfung unterzogen werden können, wurde die Datenbanksystemfamilie PLENTY[21] (Erklärung: der in PLENTY enthaltene Begriff des Prozessors beschreibt Ressourcen, derer sich zur Bewältigung der vorgegebenen Aufgaben (engl.: Tasks), bedient wird [Has95]) entwickelt. Dabei handelt es sich um drei Datenbanksysteme, wobei das Basisdatenbanksystem PLENTY die wesentlichen Funktionalitäten eines relationalen DBS bereitstellt und die Zweischichten-Transaktionsverwaltung unterstützt. Das zwei-

[19] engl.: **O**nline **T**ransaction **P**rocessing, eingesetzt in zeitkritischen Applikationen

[20] Bedeutung: behandeln komplexe Informationen und Zusammenhänge, welche wichtige Grundlage für Planungsprozesse und Analysen sind

[21] auch: **P**arallel **E**xecution of **N**ested **T**ransactions on Plenty of Processors

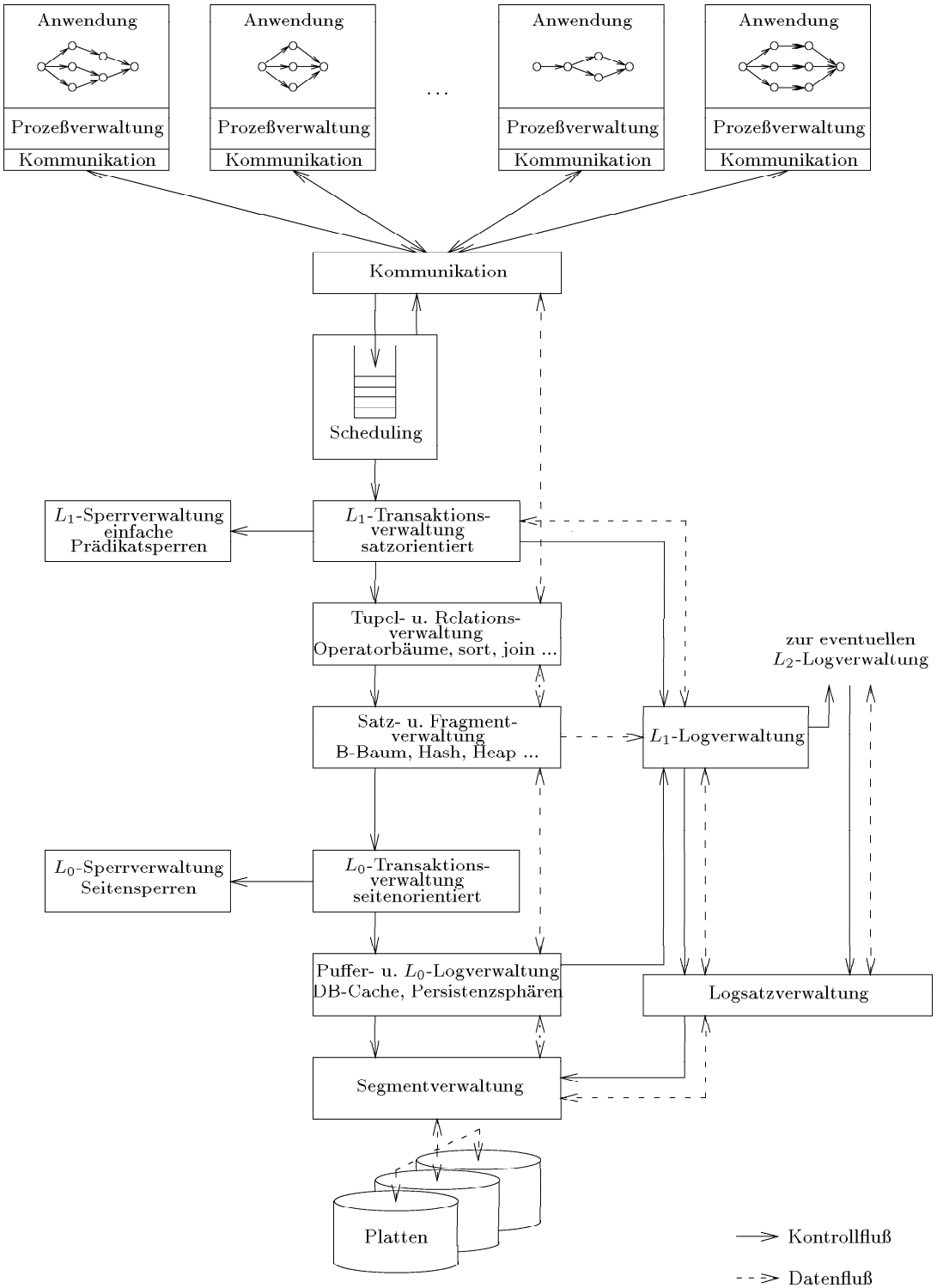

Abbildung 3.8: Client/Server-Architektur - PLENTY [Has95]

te Mitglied der Familie ist das PLENTY/Sim-System. Mit Hilfe dieses Systems ist eine gezielte Analyse von potentiellen Systemkonfigurationen möglich. Beim dritten Bestandteil des PLENTY-Projekts handelt es sich um die Anwendung/Umsetzung des PLENTY-Ansatzes auf das relationale DBS *Ingres* (Bezeichnung: PLENTY/Ingres-System). Kernpunkte der Entwicklung waren eine flexible Erweiterbarkeit und Konfiguration des DBS, nicht zuletzt im Bereich der Transaktionsverwaltung. Im Folgenden soll die Konkretisierung dieser Eigenschaften im PLENTY-DBS näher beleuchtet werden.

Bei PLENTY handelt es sich um ein Kernel-System, das in seiner Grundkonfiguration nur die nötigsten Funktionen bereitstellt und durch nachträgliches Hinzuladen von Modulen weitere Fähigkeiten erwirbt und dadurch an die gestellten Anforderungen angepasst werden kann. Es besteht die Möglichkeit, dass der Kernel zur Laufzeit benötigte Module durch dynamisches Binden (ähnlich zu Unix-Systemen) nachnominiert, um die Aufgaben bewältigen zu können. In Abbildung 3.8 sind die verschiedenen Bestandteile des PLENTY-Systems dargestellt. Die einzelnen Anwendungen treten als Clients an das Datenbanksystem (Server) heran und stellen ihre Anfragen.

Tatsächlich sind die Beziehungen zwischen Client und Server recht komplex. Die Client-Anwendung stellt ihre Anfragen an den Server in Form von Graphen (auch: Präzedenzgraph), wobei ein Knoten des jeweiligen Graphen jeweils eine Task darstellt, die ihrerseits eine Menge von Datenbankaufträgen umfassen kann. Die Anwendung mit ihrem Präzedenzgraphen wird als Task-Gruppe bezeichnet (wie in Abbildung 3.8 angedeutet). Diese Datenstruktur tritt in Interaktion mit dem Server und tauscht notwendige Informationen mit ihm aus (z.B.: verwendetes Scheduling-Verfahren, gegenwärtige und zu erwartende Systemzustände, etc.). Für genauere Informationen sei an dieser Stelle abermals auf [Has95] verwiesen, da die Zusammenhänge nicht so trivial sind, als dass sie in einem kurzen Abschnitt abgehandelt werden könnten. Nachfolgend werden die Bereiche, in denen eine flexible Anpassung des Funktionsumfanges möglich ist, näher beleuchtet.

Scheduling-Verfahren

Wie erwähnt, werden zusätzliche Funktionen in Form von Modulen dynamisch geladen. Auf diese Weise erfolgt die Nominierung eines Scheduling-Verfahrens in PLENTY durch die folgende Funktion unter gleichzeitiger Angabe erforderlicher Parameter, wobei ↑ Ausgabeparameter und ↓ Eingabeparameter kennzeichnen:

- $InitStrategy(\uparrow Strategy, \downarrow Name)$, lädt das durch $Name$ angegebene Scheduling-Verfahren aus der zugehörigen Modul-Bibliothek

Hinzu kommen noch weitere Funktionen zur Bearbeitung der Datenbankaufträge, die teilweise, je nach gewähltem Scheduling-Verfahren, in ihrer Ausprägung variieren.

Operatoren und Operatorbäume

Aufgabe der Tupelschicht (siehe: Abbildung 3.8, Tupel- und Relationsverwaltung) ist die Bereitstellung von Methoden, auf die ein Query-Compiler Datenbankanfragen abbilden

kann [Has95]. Dafür stellt die Tupelschicht von PLENTY das Konzept der Tupelströme und Operatoren, die diese Ströme manipulieren und weiterreichen (genauer: die enthaltenen Tupel), bereit. Durch eine Verkettung der Tupelströme (gerichtete Kanten) und Operatoren (Knoten) lassen sich Operatorbäume (im Allgemeinen gerichtete azyklische Graphen) aufspannen, die dann gemäss den gestellten Aufgaben durchlaufen werden.

Um verschiedene Operationen auf dem Tupelstrom (und letztendlich den entsprechenden Tupeln) durchführen zu können, muss der Operator entsprechende Funktionen bereitstellen (z.B.: *Get* (Entnahme eines Tupels aus dem Tupelstrom)). Dabei ist es wichtig, dass der Operatortyp das adressierte Tupel überhaupt verarbeiten kann. Das Kernsystem von PLENTY enthält keine eigenen Operatoren. Daher müssen die benötigten Operatoren aus einer Operatorbibliothek geladen werden. Dies geschieht über folgenden Befehl:

- $InitOperator(\downarrow OperatorName, \uparrow Operator)$, lädt einen Operator unter Angabe von $OperatorName$ und initialisiert die Datenstruktur *Operator*

Durch das flexible Laden der benötigten Operatoren, können ebenfalls die resultierenden Operatorbäume ohne grossen Aufwand an neue Anforderungen angepasst werden.

Datentypen

Da das Kernsystem von PLENTY keine eigenen Datentypen beinhaltet, müssen Mechanismen installiert werden, die das Einbinden von Datentypen realisieren. PLENTY erlaubt das Laden von *extern definierten Datentypen* (kurz: *EDT*) aus einer Datentypbibliothek. Um komplexere Operationen wie Verbunde, Selektionen oder Projektionen auf den Attributen der adressierten Tupel durchführen zu können, muss eine flexible Erweiterung der Datentypen möglich sein. Eigens dafür stellt das Kernsystem von PLENTY eine so genannte *Stapelmaschine* zur Verfügung, deren Befehlssatz eine Funktion bereitstellt, die jede beliebige Methode aufruft. So kann im Bedarfsfall eine Funktion aufgerufen werden, die eine Erweiterung eines verwendeten *EDTs* darstellt.

Flexible Speicherstrukturen

Die Satzschicht (siehe: Abbildung 3.8, Satz- und Fragmentverwaltung) trägt dafür Sorge, dass geeignete Speicherstrukturen für die Abbildung von Datensätzen auf Seiten bereitgestellt werden. Auch hier erfolgt die Konfiguration mittels entsprechender Funktionen:

- $Open(\uparrow\ Fragment, \downarrow\ FragmentName, \downarrow\ StorageM, \downarrow\ StorageMData)$, öffnet ein Fragment mit $FragmentName$, die Speicherstruktur wird mittels $StorageM(ethod)$ gewählt (BTree, Hash oder Heap), falls die Funktionen der Speicherstruktur noch nicht verfügbar sind, so werden sie aus einer Speicherstrukturbibliothek geladen, mit $StorageM(ethod)Data$ können Flags gesetzt werden, welche die Eigenschaften der Speicherstruktur festlegen

Sperrverwaltung

Die Sperrverwaltung in PLENTY basiert auf der Verwendung und Prüfung von Sperrprädikaten, die den Zugriff auf festgelegte Bereiche einer Relation steuern. Da die vom Sperrprädikat zu erfüllenden Eigenschaften von Anwendungsfall zu Anwendungsfall variieren, werden die Datenstruktur Sperrprädikat und ihre Ausprägungen nicht innerhalb der Sperrverwaltung des Kernsystems implementiert, sondern als *EDTs* in separaten Modulen, die bei Bedarf nach dem bekannten Schema geladen werden. In jeden Fall muss ein Sperrprädikat die Vergleichsfunktion (hier: $CompareProdicate(\uparrow Result, \downarrow Predicate1, \downarrow Predicate2)$) implementieren, unter deren Verwendung entschieden werden kann, ob sich zwei Sperrprädikate vertragen. Die Datenstruktur kann darüber hinaus um weitere Funktionen ergänzt werden (z.B.: Kopieren und Freigeben von Sperren).

Zusammenfassung

Das PLENTY-Datenbanksystem beruht auf einer Client/Server-Architektur, bei der die einzelnen Anwendungen ihre Anfragen in Form von graphenbasierten Datenstrukturen an das PLENTY-DBS richten. Ein Kernsystem stellt Basisfunktionalitäten zur Verfügung, die nach Bedarf durch weitere Funktionen ergänzt werden. Dies geschieht durch das dynamische Binden der entsprechenden Funktionsmodule aus dafür bereitgestellten Modulbibliotheken.

3.3.3 Konzepte im Überblick

In den Abschnitten 3.3.1 und 3.3.2 wurden mit COMET und PLENTY zwei Systeme näher betrachtet, die sich auf verschiedenen Wegen dem Thema der flexiblen, konfigurierbaren DBMS nähern. Gegenstand der folgenden Betrachtungen soll die Einordnung verschiedener Lösungsansätze, auf der Basis charakteristischer Merkmale, sein.

Kernel-Systeme

Dittrich und Geppert [Dit01] beschreiben ein kernel-orientiertes DBMS als ein Gebilde, das in einem Basissystem (auch: Kernel) grundlegende Funktionen bereitstellt. Obwohl der Funktionsumfang eines derartigen Systems beachtlich ist, muss dieser, für konkrete Anwendungsszenarien, entsprechend den Anforderungen, erweitert werden, da der Kern häufig noch kein ausführbares DBMS darstellt. Das Kernsystem muss geeignete Schnittstellen für die Kopplung der Erweiterungen zur Verfügung stellen. Die Leistungsfähigkeit eines Kernel-Systems hängt stark von der Wahl des richtigen Verhältnisses, aus Kernel-Umfang und zusätzlich implementierten Funktionen, ab. Ein grosszügig dimensionierter Kernel erübrigt, je nach Funktionszusammenstellung, in vielen Fällen eine Erweiterung durch externe Lösungen. Gleichzeitig ist er weniger flexibel konfigurierbar und im schlimmsten Fall für den Einsatz im Bereich der eingebetteten Systeme

ungeeignet. Bei einem kleinen Kernsystem wird der Grossteil an Funktionalität ausgelagert. Dies ermöglicht vielseitige Konfigurationsmöglichkeiten. Kommt es jedoch zu einer exzessiven Funktionsauslagerung, so führt dies zu Problemen bei der Schnittstellenbereitstellung und -verwaltung, da jede externe Methode ein geeignetes Interface benötigt, um mit dem Kernel kooperieren zu können. Wie in Abschnitt 3.3.2 angedeutet und ausgehend von den hier beschriebenen Eigenschaften, handelt es sich bei PLENTY um ein Kernel-System.

Anpassbare Systeme

Bei anpassbaren Systemen (engl.: Customizable Systems) handelt es sich um vollständige DBMS, die an neue Aufgaben durch Anpassungsmechanismen herangeführt werden. Um neue Funktionen zu integrieren, wird das alte System derart überarbeitet, dass die Programmteile ausgetauscht werden, die den neuen Anforderungen nicht mehr genügen. Für den Erfolg dieses Anpassungsprozesses ist es wichtig, die entsprechenden Programmstellen (auch: Variationspunkte) zuverlässig zu identifizieren und durch passenden Programm-Code zu ersetzen. Im Fall, dass nicht alle funktionssensiblen Stellen modifiziert wurden, kann es zu fehlerhaften Programmausführungen kommen.

Transformationssysteme

Als Vorläufer des, in dieser Arbeit thematisierten Ansatzes der Entwicklung konfigurierbarer DBMS mittels merkmalsorientierter Programmierung, kann der Ansatz der Transformationssysteme (engl.: Transformational Systems) bezeichnet werden. Ausgangspunkt bildet eine Schichtenarchitektur, die in ihren einzelnen Elementen verschiedene Funktionen implementiert. Durch die Einführung neuer Schichten wird die vorhandene Architektur mit neuen Funktionen ausgestattet. Dabei kann sich eine Schicht über festgelegte Schnittstellen vorhandener Funktionen aus übergeordneten Schichten bedienen. Bei der Zusammenstellung eines DBMS werden die für den Anwendungsfall benötigten Schichten ausgewählt. Unterstützt wird diese Form der Konfiguration durch die Angabe von Kombinationsregeln, die eine unsinnige Zusammenstellung verhindern sollen.

Frameworks

Frameworks ermöglichen die Modellierung von Datenbankfamilien. Funktionen für die verschiedene Alternativen bereitgestellt werden sollen, werden auf der Basis abstrakter Klassen implementiert. Die konkrete Implementierung einer variabel einsetzbaren Funktion wird von der zugehörigen abstrakten Klasse abgeleitet. Dittrich und Geppert [Dit01] nennen diesen Vorgang: Wiederverwendung durch Vererbung. Jede alternative Funktion wird in einer neuen Klasse implementiert.

Die Ausnutzung des Prinzips der Vererbung gibt dem Entwickler ein effektives Werkzeug für eine flexible DBMS-Entwicklung und -Konfiguration in die Hand. Da die Klassen

für ein bestimmtes Framework konzipiert werden, kann es jedoch zu Problemen bei der Intergration in Projekte, die ausserhalb des ursprünglichen Frameworks liegen, kommen.

Generatoren

Ausgangspunkt für die Erstellung eines DBMS auf der Basis von Generatoren ist die präzise Spezifikation der umzusetzenden Funktionalitäten. Zentrales Element der Spezifikation ist das Modell der zu implementierenden Funktion. Die in der Spezifikation erarbeiteten Daten werden anschliessend in den Generator gespeist, der daraus ein (den Input-Werten entsprechendes) DBMS generiert. Natürlich muss der Generator in der Lage sein, die eingespeisten Informationen in angemessener Weise verarbeiten zu können. Er muss Routinen und Informationen bereitstellen, die dies ermöglichen. Die Entwicklung von konfigurierbaren DBMS mittels dieses Ansatzes ist ein interessantes Konzept. Jedoch ist es ein schwieriges Unterfangen, den Generator mit dem Know-How auszustatten, der es ihm ermöglicht, die Input-Informationen korrekt zu deuten, weil die dafür benötigten Transformationsregeln, gerade bei umfangreichen Projekten, sehr komplex werden können.

Toolkit-Systeme

Wie der Name verrät, handelt es sich bei Toolkit-Systemen um Bausätze, die verschiedene alternative Funktionen in Bibliotheken organisieren, welche für den konkreten Anwendungsfall zusammengefügt werden. Dittrich und Geppert [Dit01] betonen, dass dieses Konzept auch mit anderen diskutierten Lösungen kombiniert werden kann. PLENTY ermöglicht beispielsweise die Erweiterung des Funktionsumfanges in Form von zusätzlichen Modulen, die in geeigneten Modulbibliotheken organisiert und im Bedarfsfall nachgeladen werden.

Aufgrund der komplexen Modulabhängigkeiten in einem DBMS kann es bei der Zusammenstellung und im Betrieb zu vielfältigen Problemen, aufgrund von Inkompatibilitäten zwischen den einzelnen Komponenten, kommen. Da die Konfiguration des in Abschnitt 3.3.1 beschriebenen COMET-Systems auf der Basis von Aspekten und Komponenten, die verschiedene Aufgabenbereiche eines DBMS kapseln, erfolgt, kann es am ehesten den Toolkit-Systemen zugeordnet werden.

Echt erweiterbare Datenbanksysteme

Carey und Haas [Car90] bezeichnen ein Datenbanksystem als erweiterbar, wenn es eine beliebige Form von Leistungssteigerung, Erweiterung oder Anpassung unterstützt. Ausgehend von dieser Festlegung, handelt es sich beim erweiterbaren Datenbanksystem um einen Sammelbegriff, der die bisher beschriebenen Lösungen umfasst, weil jedes dieser Systeme der vorgegebenen Definition genügt. Echt erweiterbare Datenbanksysteme (engl.: Pure Extensible Database Systems) spezialisieren den Begriff der erweiterbaren

Datenbanksysteme, da sich die Art der Erweiterung auf ADTs (lang: Abstrakte Datentypen) oder Indexstrukturen beschränkt.

3.4 Merkmalsanalyse

Nachdem die Anforderungen, die das Einsatzgebiet der eingebetteten Systeme vorgibt, weitestgehend bekannt sind, verschiedene alternative Ansätze zum Themenkomplex der konfigurierbaren DBMS, genauer der Transaktionsverwaltung, in eingebetteten Systemen analysiert und somit eine grosse Menge an Domänenwissen gesammelt wurde, ist es an der Zeit, mit der Modellierung der Domäne zu beginnen. Zunächst muss sichergestellt werden, dass die Abgrenzung zwischen Transaktionsverwaltung und Systemen, die mit ihr in Interaktion stehen, klar formuliert worden ist (siehe Abschnitt 3.2.2). Erst dann kann damit begonnen werden, das System in übersichtliche funktionale Einheiten (Features) zu unterteilen. Ergebnis der Einteilung sind verschiedene Merkmalsdiagramme, in denen die identifizierten Features hierarchisch angeordnet sind. Ziel dieser Arbeit ist es, wichtige Bereiche der Transaktionsverwaltung in Beziehung zur aspekt- und feature-orientierten Anwendungsentwicklung zu setzen, und die Verträglichkeit zwischen beiden Komplexen zu untersuchen. Im Sinne der Übersichtlichkeit soll auf die vollständige Identifikation aller Merkmale verzichtet werden.

3.4.1 Transaktions-Manager

Der Transaktions-Manager ist die Instanz, welche die in Auftrag gegebenen Transaktionen aufnimmt und deren Ausführung steuert und überwacht. Abbildung 3.9 zeigt das Feature-Diagramm mit den identifizierten Merkmalen.

Das erste Merkmal, welches vom Konzept des Transaktions-Managers (TM) gefordert (*mandatory*) wird, ist $Transaktion$. Es beinhaltet Basisfunktionalitäten für die Verwaltung der eingegangenen Transaktionen. Dieses Feature besitzt seinerseits Kinder, welche ebenfalls angewählt werden müssen. $TransStatus$ übernimmt die Funktion der Abfrage und das Setzen von Transaktionsstati (*Commit*, *Abort*, etc.). $QueueTransOp$ hat die Aufgabe, alle zur Transaktion gehörenden Operationen in ihrer ursprünglichen Reihenfolge aufzunehmen und für die weitere Verarbeitung bereitzustellen. Die durch den Teilbaum mit der Wurzel $Transaktion$ repräsentierte Datenstruktur ist Grundlage für alle folgenden Operationen des Transaktionsmanagers. Damit die einzelnen Transaktionen einer ordnungsgemässen Bearbeitung zugeführt werden können, müssen sie in geeigneter Form verwaltet werden. Dies leistet das Feature $TransaktionsListe$. In diese Liste werden alle eingehenden Transaktionen aufgenommen.

Alle bisher diskutierten Merkmale bilden die Grundlage für den zentralen Gegenstand des Transaktions-Managements - den $Scheduler$. Der Scheduler benötigt zunächst Möglichkeiten zur Steuerung der Statusinformationen aller beteiligten Transaktionen. $AdminTransStatus$ übernimmt das Setzen der Stati. Im späteren Ausführungsprozess kann mit Hilfe dieses Merkmals der Zustand jeder einzelnen Transaktion abgefragt und

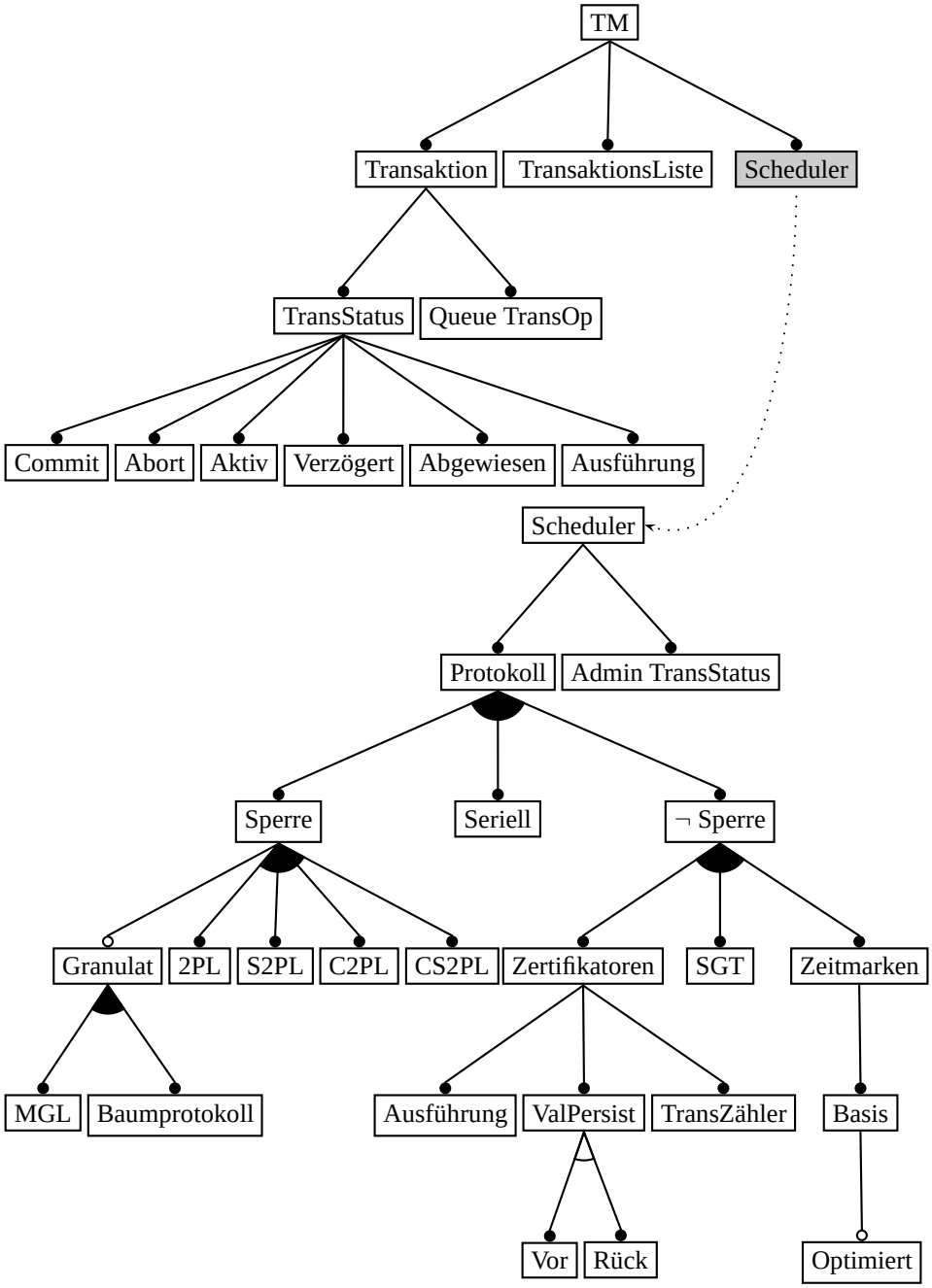

Abbildung 3.9: Merkmalsdiagramm Transaktions-Manager

auf Basis der Informationen die passende Methode (Deadlock-Beseitigung, Verzögerung von kritischen Transaktionsoperationen, Transaktionsabbrüche, etc.) angestossen werden. Ausgangspunkt des Schedulers ist ein vom Transaktions-Manager zufällig gewählter Input-Schedule, der alle Operationen der beteiligten Transaktionen umfasst. Für die Abarbeitung dieses Schedules bedarf es einer Ausführungsvorschrift - einem *Protokoll*. Zur genaueren Beschreibung dieser Vorschrift gibt es drei grundsätzliche Möglichkeiten - die serielle Ausführung, sperrende- und nicht-sperrende Verfahren. Die Oder-Gruppierung legt fest, dass jede nichtleere Menge, bestehend aus den drei genannten Verfahren, eine korrekte Scheduler-Konfiguration darstellt.

Die sperrenden Verfahren stellen, auf der Basis einer allgemeinen Sperrdisziplin, die korrekte Ausführung des gegenwärtigen Schedules sicher und verhindern, je nach Protokoll, verschiedene Formen von Mehrbenutzeranomalien[22] (Bedeutung: auftreten von Konfliktsituationen bei der gleichzeitigen Benutzung eines Datenobjekts durch mehrere Transaktionen). Zu diesem Zweck werden die Zugriffe auf benötigte Datenobjekte durch das Setzen von Lese- oder Schreibsperren sanktioniert. Die Auswahl der zu nutzenden Sperrprotokolle ($2PL$, $S2PL$, $C2PL$ und $CS2PL$) wird ebenfalls über eine Oder-Gruppierung gesteuert.

Wird die Locking-Information direkt im Datenobjekt gespeichert, so erfordert das Ermitteln, Setzen und Aufheben von Sperrinformationen, für jeden Ausführungsschritt das Aufsuchen der entsprechenden Daten in der Speicherstruktur (häufig baumartig). Sind grosse Datenmengen vorhanden, so kann es beim Zugriff (auch: Durchlaufen der Speicherstruktur bis zum Zielobjekt) zu erheblichen Verzögerungen kommen. Es ist aus diesem Grund wünschenswert, durch möglichst wenige Zugriffsoperationen, an die gesuchte Sperrinformation zu gelangen. Diese Eigenschaft wird von *Sperrgranulaten* erfüllt. Dabei wird zwischen zwei verschiedenen Formen unterschieden. Das Verfahren des *hierarchischen Sperrens*(auch: ***M**ulti **G**ranularity **L**ocking - MGL*) ermöglicht das Sperren einzelner Abschnitte der Datenbank auf logischer Ebene (z.B.: Datenbank -¿ Relation -¿ Tupel -¿ Attribut). Statt jedes einzelne Datenobjekt mit einer Sperre zu belegen, können durch die Vergabe von Sperren für ganze Teilbereiche von Datenbanken mehrere Objekte gesperrt werden. Dies hat den Vorteil, dass der Aufwand der Sperrverwaltung geringer ausfällt. Die Verwendung von *Baumprotokollen* lässt es zu, dass einzelne Elemente von Indexstrukturen gesperrt werden können. Da es sich bei dem Merkmal (*Granulat*) um eine Optimierung handelt, die nicht zwangsläufig für den Betrieb verschiedener Sperrverfahren vorgesehen ist, wird es als optional angegeben.

Einen anderen Ansatz zur systematischen Transaktionsausführung verfolgen *nichtsperrende Verfahren*. Wie der Name vermuten lässt, wird auf die Vergabe von Sperrattributen für die beteiligten Datenobjekte verzichtet. Zertifikatoren gehören zu den optimistischen Scheduling-Verfahren. Die Idee dabei ist, dass die gegebenen Transaktionen zunächst in einer sicheren Umgebung im Datenbankenpuffer ausgeführt werden,

[22]Verletzt eine Transaktionsoperation das gefahrene Protokoll und muss aus diesem Grund abgebrochen werden, so können die Folgen des Abbruchs gemindert werden (z.B. Vermeidung kaskadierender Abbrüche beim *Strikten 2-Phasen Sperrprotokoll*).

ohne auf Serialisierbarkeit zu testen. Anschliessend erfolgt in der Validierungsphase die Überprüfung des Auftretens eventueller Konflikte. Dabei kann zwischen zwei alternativen Validierungsstrategien gewählt werden (Vor, *Rück*), die sich jeweils gegenseitig ausschliessen (Exklusiv-Oder-Gruppierung). Im Anschluss an die Validierung erfolgt die persistente Speicherung der Ausführungsergebnisse. *SGT* (engl.: *Serialization Graph Tester*) ist ein weitere Variante eines nicht-sperrenden Verfahrens. Hier wird ein Konfliktgraph verwaltet, der bei jedem Ausführungsschritt auf Zyklen untersucht wird. Bei Zeitmarkenverfahren handelt es sich um Methoden, die jeder Transaktion ein Zugriffsattribut (engl.: *Timestamp*) zuweisen, durch dessen Charakteristik die Vorrangigkeit beim Zugriff auf gemeinsame Datenobjekte geregelt wird. Optional kann das Verfahren bezüglich des *blinden Schreibens* optimiert werden.

3.4.2 Recovery-Manager

Nachdem die wichtigsten Merkmale des Transaktions-Managers identifiziert sind, sollen die Features der zweiten wichtigen funktionalen Einheit im Transaktionsverwaltungsprozess - dem Recovery-Manager - markiert werden. Der Recovery-Manager sorgt dafür, dass die vom Scheduler[23] erzeugte Operationsreihenfolge der laufenden Transaktionen auf Datenbasis ausgeführt werden. Seine Arbeit verrichtet der Recovery-Manager auf den betroffenen Datenobjekten, die zur Manipulation in den Datenbankpuffer geladen werden.

Um die Manipulation der Datenobjekte gemäss den eintreffenden Transaktionsoperationen ausführen zu können, benötigt der Recovery-Manager Methoden zum Zugriff auf den Datenbankpuffer. Das Konzept RM fordert das Merkmal $Interface$, das einfache Schnittstellenbeschreibungen beinhaltet. Diese werden durch $Puffer$ so erweitert, dass der Recovery-Manager in die Lage versetzt wird, die Elemente des Datenbankpuffers bearbeiten zu können. Kommt es vor, dass transaktionsspezifische Datenobjekte nicht im Datenbankpuffer vorhanden sind, so müssen sie aus der Datenbank geladen werden. Für diesen Vorgang ist der Puffer-Manager verantwortlich, der vom Recovery-Manager den entsprechenden Auftrag über die Schnittstelle $PufferM$ erhält. Über diese Schnittstelle signalisiert der Recovery-Manager dem Puffer-Manager zusätzlich, dass ein manipuliertes Datenobjekt zur persistenten Speicherung ausgelagert werden soll.

Der Recovery-Manager muss verschiedene Strategien zur Wiederherstellung eines konsistenten Datenbankzustandes (nach einem Fehlerfall) anbieten. *Recovery* umfasst einfache Seitenersetzungs- und Propagierungsstrategien, die alle Änderungen auf einer Kopie im Datenbankpuffer vornehmen und diese, im Fall eines Commit der zugehörigen Transaktion, sofort und vollständig in den persistenten Speicher verlagern. Diese Strategie wird als *No-Undo/No-Redo* bezeichnet, weil keinerlei Information für das Aufheben (Undo) von Änderungen im persistenten Speicher unvollständiger Transaktionen und Wiederholen (Redo) von Änderungen (nicht persistent gespeichert) erfolgreich beende-

[23] Gemäss Abbildung 3.4 in Abschnitt 3.2.2 muss eine geeignete Schnittstelle zwischen Scheduler und Recovery-Manager bestehen, über die die Elemente des Output-Schedules transportiert werden.

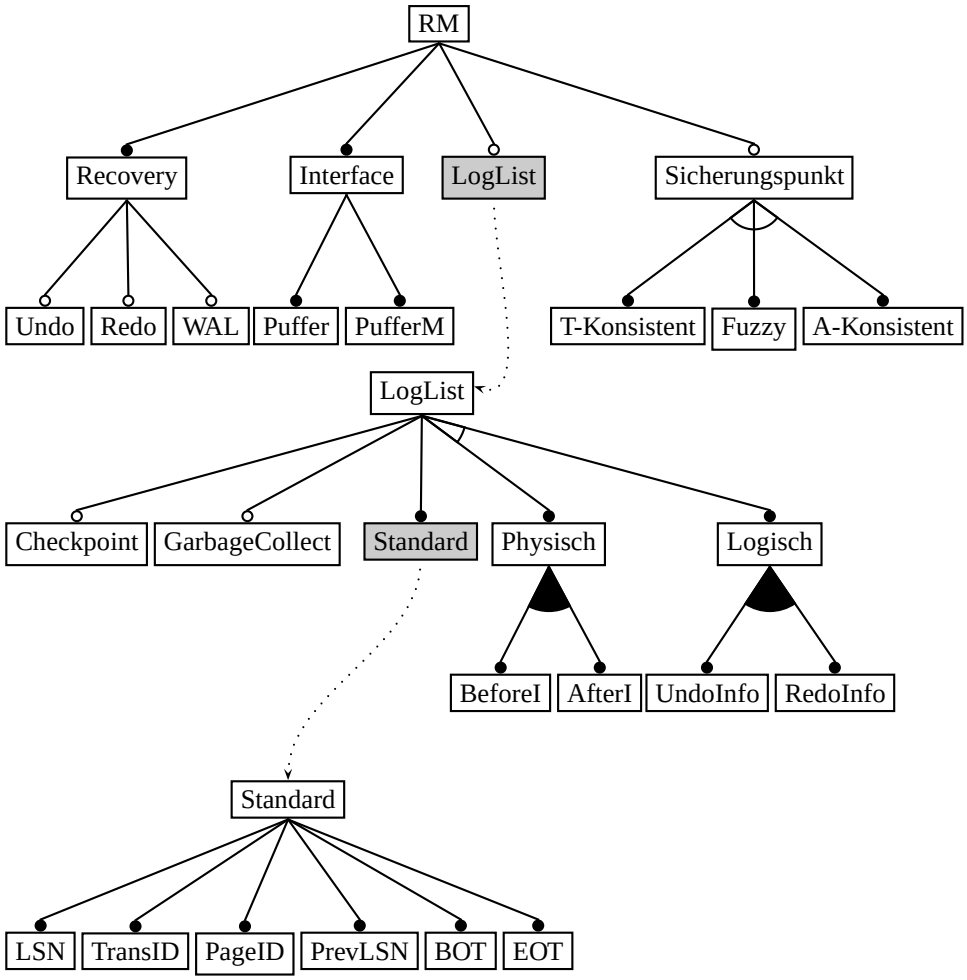

Abbildung 3.10: Merkmalsdiagramm Recovery-Manager

ter Transaktionen benötigt wird. *Recovery* kann im Sinne der Optimierung von Zugriffen auf den meist langsamen persistenten Speicher so abgeändert werden, dass drei alternative Recovery-Strategien gefahren werden können (*No-Undo/Redo*, *Undo/No-Redo* und *Undo, Redo*). Neben den Methoden zur Handhabung von Datenobjektänderungen besteht ergänzend die Möglichkeit das Auslagern von Log-Informationen zu systematisieren (durch: *WAL* - **W**rite **A**head **L**og).

Ausser *No-Undo/No-Redo* benötigt jede der übrigen Recovery-Strategien Informationen, um nach Fehlfunktionen (Transaktionsabbrüche, Systemfehler, etc.) wieder einen konsistenten Datenbankzustand herbeizuführen. Die Grundlagen für eine entsprechende Datenstruktur werden im Merkmal *LogList* vereinbart. Dazu gehören Methoden zur Manipulation und Verwaltung von Listen und deren Einträgen. *LogList* fordert zunächst die Implementierung von Verwaltungsinformationen[24] und Markern für Start (*BOT*)

[24]in *Standard*: *LSN* (**L**og-**S**equence-**N**umber), $TransID$, $PageID$ (Seitennummer), $PrevLSN$ (vor-

und Ende (*EOT*) einer Transaktion. Die Protokollierung der tatsächlichen Wiederherstellungsinformation kann auf zwei verschiedene Arten erfolgen, die alternativ gewählt werden. Die Charakteristik des *physischen Protokollierens* leitet sich aus der Tatsache ab, dass Undo- und/oder Redo-Informationen die Werte eines Datenobjekts vor (auch: *Before-Image*) und nach (auch: *After-Image*) der Manipulation enthalten. Dabei ist unter einem Datenobjekt der komplette Inhalt einer Datenbankseite zu verstehen. Unter *logischer Protokollierung* wird die Speicherung von Wiederherstellungsfunktionen verstanden. Funktionen, die die Werteänderung eines Datenobjekts wieder rückgängig machen, stellen die Undo-Information dar. Noch nicht persistent gespeicherte Änderungen können bei Verlust ebenfalls durch Funktionen (Redo-Information) nachvollzogen werden.

Damit der Umfang der Log-Liste keine schwer beherrschbaren Ausmaße annimmt, kann optional ein *Garbage Collector* hinzugezogen werden, der nicht mehr benötigte Informationen aus dem Log-Buch entfernt. Diese Maßnahme erleichtert das Handling der Log-Datei bei der Wiederherstellung.

Von Zeit zu Zeit ist es sinnvoll sicherzustellen, dass alle Änderungen des Datenbankzustands in den stabilen Speicher geschrieben werden. Damit dieser Vorgang auf einer Systematik gründet, besteht die Möglichkeit der Einführung von Sicherungspunkten. Wird diese Option wahrgenommen, so muss die Wahl auf eine der angebotenen Alternativen fallen. *Transaktionskonsistente Sicherungspunkte* melden zuerst das Vorhaben der Sicherung an. Alle zu diesem Zeitpunkt laufenden Transaktionen müssen erfolgreich beendet werden. Erst dann werden die geänderten Datenbankseiten in den stabilen Speicher geschrieben. Bei *aktionskonsistenten Sicherungspunkten* werden die aktiven Transaktionen nach Ablauf festgelegter Zeitabschnitte blockiert und die bis dato geänderten Datenbankseiten persistent gespeichert [Saa99]. Unter *unscharfen Sicherungspunkten* wird verallgemeinernd das persistente Ablegen von Datenbankseiten, die vor dem letzten Sicherungspunkt nicht ausgeschrieben wurden, verstanden. Da die Durchführung von Sicherungspunkten die Auslagerung aktueller Datenbankänderungen in den stabilen Speicher gewährleistet, sollte ihr auch ein Log-Buch-Eintrag gewidmet werden. Je nach Art der Sicherung muss das Log-Buch im Fall der Wiederherstellung nur noch bis zum letzten *Checkpoint* durchlaufen (beim letzten Eintrag beginnend rückwärts) werden.

3.5 Nutzergruppe vs. Nutzungsform

Letztendlich handelt es sich bei den Mitgliedern der Programmfamilie Transaktionsverwaltung nicht um vollständige DBMS, welche eine Datenbank administrieren können. Vielmehr müssen die Mitglieder erst in ein DBMS integriert werden, um ihre Dienste anbieten zu können. Daher ist die Gruppe potentieller Nutzer der Programmfamilie im Bereich der Datenbankentwicklung, insbesondere für eingebettete Systeme, zu suchen. Die Entwickler verfügen über Wissen, welches nötig ist, um die Transaktionskomponente zunächst dem Anwendungsfall entsprechend zu konfigurieren.

heriger Log-Eintrag der Transaktion)

Bärecke [Bä05] schlägt für seinen konfigurierbaren Speichermanager die Bereitstellung der Funktionen in Form von Bibliotheken vor. Die Verwendung der Funktionen erfolgt über eine geeignete, ausreichend dokumentierte *API* (auch: **A**pplication **P**rogramming **I**nterface). Das beschriebene Konzept bietet sich ebenfalls für die Programmfamilie Transaktionsverwaltung an. Der Entwickler stattet die Bibliothek mit den geforderten Funktionen aus (betrifft die Konfiguration für konkrete Anwendungsfälle auf Basis der Programmfamilie). Anschliessend kann die generierte Funktionsbibliothek über die definierten Schnittstellen angesprochen werden. Unter Verwendung weiterer Systembibliotheken kann ein vollständiges, funktionsfähiges DBMS zusammengestellt werden.

Vorteil von Funktionsbibliotheken ist die einfache Erweiterbarkeit, welche auch ausserhalb der Domänenentwicklung (unter Verwendung der angebotenen API) erfolgen kann. Wesentliche Eigenschaft der vorgestellten Lösung ist die Nutzung der Funktionen auf Sourcecode-Ebene. Der Entwickler greift über die API auf die benötigten Funktionen und Bibliotheken zu und entwickelt daraus das gewünschte System.

Denkbar ist auch ein anderer Ansatz, der die Nutzung der Transaktionsverwaltungsfunktionen von der Quellcode-Ebene und somit vom Entwickler loslöst. Das System, bestehend aus Transaktions- und Recovery-Manager, wird als eigenständige Komponente entwickelt, die nur über die nötigsten Kommunikationskanäle verfügt. Die Implementierung der vereinfachten Schnittstellen erfolgt innerhalb der Programmfamilie. Gleichzeitig wird dafür gesorgt, dass die externen Programmteile mit dem nötigen Wissen über die Nutzung der Schnittstellen versorgt werden.

Das System kann als eine Art Automat aufgefasst werden, der seinen Dienst auf der Basis geeigneter Input-Werte (u.a.: Input-Transaktionen) verrichtet und das Produkt seiner Arbeit mittels entsprechender Output-Werte (u.a.: Befehle zur Datenmanipulation im Puffer, Fetch- und Flush-Befehle an den Puffermanager) in seine Umgebung delegiert. Durch die Kapselung der einzelnen Funktionen ist es dem Entwickler nicht möglich, diese direkt anzusprechen. Er trägt lediglich dafür Sorge, dass das System entsprechend dem Anwendungsfall konfiguriert wird. Vorteil dieser Lösung ist der Verzicht auf die Verwendung einer umfangreichen API. Durch die einfachen Verbindungsstellen kann das System klar von seiner Umgebung abgegrenzt werden. Dem gegenüber können Erweiterungen und systemspezifische Anpassungen leider nicht auf API-Ebene erfolgen, sondern müssen im Rahmen der Domänenentwicklung durchgeführt werden. Letzt genannter Punkt ist ein wesentlicher Nachteil des diskutierten Ansatzes, da der Datenbankentwickler im Falle von notwendigen Erweiterungen auf die Hilfe der Programmfamilienentwickler angewiesen ist. Ferner würde es mit erheblichen Schwierigkeiten verbunden sein, die Kompatibilität der im Automaten definierten Schnittstellenbeschreibungen für jeden erdenklichen Anwendungsfall zu gewährleisten.

Unter Abwägung aller Vor- und Nachteile zu den erläuterten Ansätzen, fällt die Wahl der Nutzungsform auf das Konzept der api-gestützten Funktionsbibliotheken.

3.6 Zusammenfassung

Das Kapitel der Domänenanalyse befasst sich mit der Beschaffung, Entwicklung und Aufbereitung von domänenspezifischem Wissen. Zu diesem Zweck wurden zunächst wesentliche Aspekte eingebetteter Systeme betrachtet. Hauptaugenmerk lag auf dem Begriff der Ressourcenknappheit. Verschiedene Dimensionen des Begriffs sind: Speicherausstattung, Energiereserven, Rechenleistung, Ergonomie. Zu Beginn des Kapitels wurden allgemeine Betrachtungen zu Systemen mit unterschiedlichen Anteilen an Hard- und Software und die Erläuterung der Vor- und Nachteile derartiger Lösungen durchgeführt. Anschliessend wurden verschiedene Prozessorarchitekturen analysiert und auf ihre Eignung für typische Transaktionsprozesse untersucht. Dabei stellte sich heraus, dass insbesondere Prozessoren die nebenläufige Prozesse ermöglichen, als geeignet einzustufen sind, da es sich bei parallel auszuführenden Transaktionen um eben solche Prozesse handelt. In eingebetteten Systemen können verschiedene Speicherarchitekturen zum Einsatz kommen. Dieser Umstand erzwang die Analyse aller Abhängigkeiten, aufgrund der Verwendung unterschiedlicher Architekturen, die sich für den Bereich der Transaktionsverwaltung ergeben. Besondere Aufmerksamkeit muss in diesem Zusammenhang Systemen ohne persistenten Speicher geschenkt werden, da ihre Beschaffenheit gravierenden Einfluss auf Recovery- und Backup-Strategien hat. Im Anschluss an die Betrachtungen zu Prozessorarchitekturen und Speichertechnologien wurden verschiedene Systeme untersucht, die sich einer Datenhaltungskomponente bedienen. Dabei stellte sich heraus, dass das Spektrum der geforderten Eigenschaften sehr gross ist (z.B. Echtzeitfähigkeit, Datensicherheit, etc.). Die Beschreibung des allgemeinen Aufbaus eines DBMS und der Transaktionsverwaltung diente der Identifikation der Aufgabenbereiche und Systemgrenzen. Gemäss der 5-Schichten-Architektur nach Härder [Hä87] sind die Belange der Transaktionsverwaltung im Zugriffs- und Speichersystem verankert. Um die Leistungsfähigkeit des Entwicklungsansatzes der Programmfamilie und der Domänenentwicklung beurteilen zu können, musste eine Untersuchung alternativer Lösungen erfolgen. Etwas genauer geschah dies für das kernel-basierte PLENTY-System und das Baukastensystem COMET. Erste Schritte zur Modellierung der Domäne wurden mit der Durchführung der Merkmalsanalyse für den Transaktions- und Recovery-Manager vollzogen. Mittelpunkt der Modellierung bilden die zugehörigen Merkmalsdiagramme. Als Zielgruppe für die zu entwickelnde Programmfamilie wurden Datenbankentwickler identifiziert. Ihnen obliegt die Aufgabe der anwendungsspezifischen Konfiguration der Transaktionsverwaltungskomponente.

Kapitel 4

Entwurf

Dieses Kapitel beschäftigt sich mit dem Entwurf ausgewählter Aspekte der Transaktionsverwaltung. Auf Grundlage der für die prototypische Umsetzung nominierten Merkmale, wird ein überarbeitetes Merkmalsdiagramm entwickelt. Ausgehend von diesem Diagramm, im Zusammenhang mit Zusatzinformationen, wie etwa Entwurfsregeln, fällt die Entscheidung für das strukturgebende Konzept.

4.1 Ausgesuchte Merkmale

Im Rahmen dieser Diplomarbeit können aufgrund des Funktionsumfangs nicht alle identifizierten Merkmale implementiert werden. Daher verfolgt die Umsetzung ausgewählter Merkmale, anhand derer mögliche Probleme und Herausforderungen im Umgang mit der Transaktionsverwaltung und den Methoden der merkmals- und aspektorientierten Programmierung verdeutlicht werden können. Für weitere Vereinfachungen der Prototypenentwicklung werden einzelne funktionale Passagen auf höheren Abstraktionsebenen behandelt. Ziel der Entwicklung ist ein Prototyp, dessen Eigenschaften die Eignung (für die Realisierung von konfigurierbaren DBMS (besser: Teilen von DBMS)) der verwendeten Technologien belegen.

Abbildung 4.1 zeigt das reduzierte Merkmalsdiagramm des Transaktions-Managers, welches, als eine Art Zielvorgabe, die zu implementierenden Merkmale beinhaltet. Im Aufgebot stehen alle Merkmale, die für transaktionsspezifische Verwaltungsprozesse benötigt werden. Weiterhin stehen für die parallele Transaktionsausführung die vier bekannten Sperrprotokolle zur Verfügung. In Ergänzung zu den Sperrprotokollen ist ebenfalls die serielle Transaktionsausführung realisierbar.

Zentrale Aufgabe des Recovery-Managers ist die tatsächliche Ausführung der Transaktionsoperationen. Da die Verwendung der No-Redo/No-Undo-Strategie mit sehr vielen Schreib- und Lesezugriffen (insbesondere auf dem zumeist verhältnismäßig langsamen Sekundärspeicher) verbunden ist und dies nicht das einzige Performance-Problem dieser Strategie darstellt, empfiehlt sich die Verwendung von Mechanismen (genauer: Undo, Redo), die den Zeitpunkt der persistenten Speicherung von Änderungen zu Gunsten

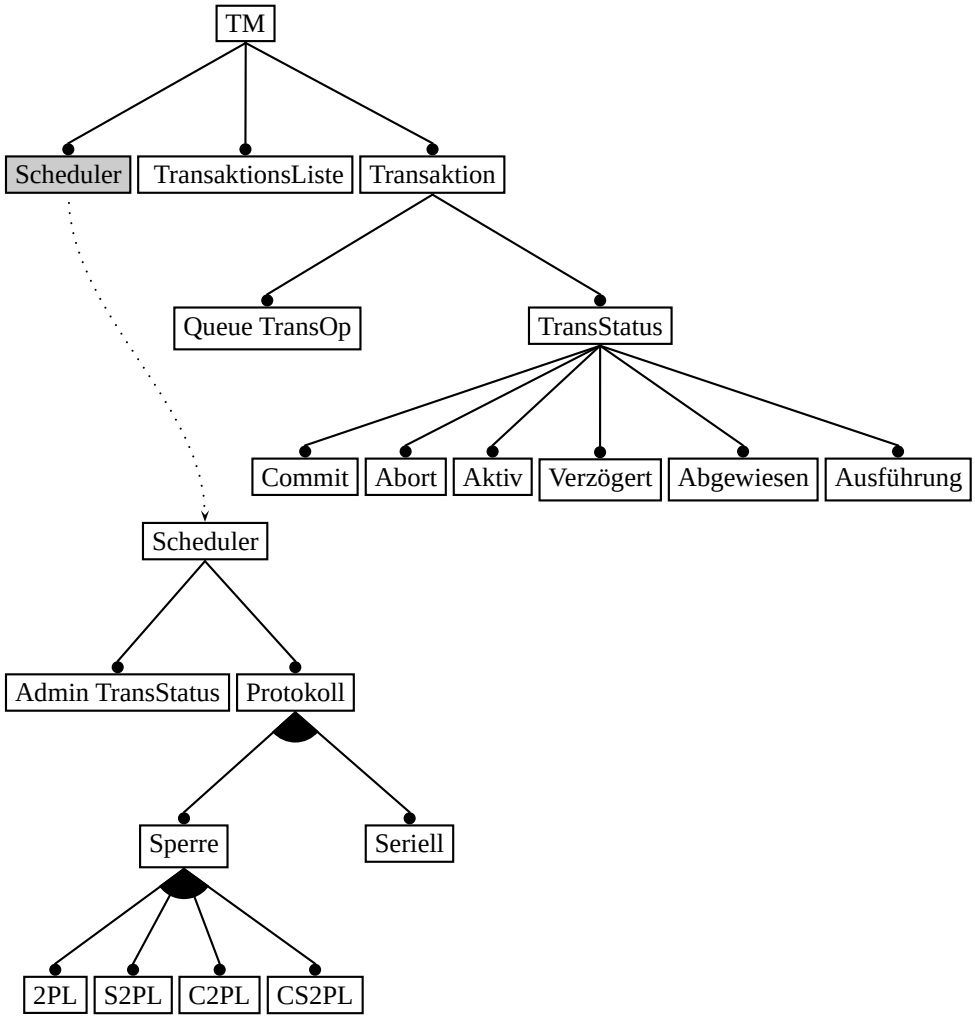

Abbildung 4.1: reduziertes Merkmalsdiagramm Transaktions-Manager

einer effektiven Speicherverwaltung (bzgl.: Speicherzugriff, Speicherplatz), verschieben. Während die Vorteile derartiger Strategien im laufenden, fehlerfreien Betrieb deutlich zum Tragen kommen, werden die negativen Effekte im Fehlerfall deutlich. Hier müssen persistent gespeicherte Datenänderungen abgebrochener Transaktionen rückgängig (Undo) gemacht und Änderungen (nicht persistent) abgeschlossener Transaktionen nachvollzogen (Redo) werden. Die so genannten Recovery-Maßnahmen können unmöglich ohne geeignete Datenbasis ergriffen werden. Daher kommt dem Aufzeichnen der relevanten Wiederherstellungsinformationen eine tragende Rolle zu. Neben dem hohen Stellenwert für die Transaktionsverwaltung birgt die Disziplin der Protokollierung entwicklungsspezifische Besonderheiten, die eine Implementierung interessant machen. Aus diesem Grund konzentriert sich die prototypische Entwicklung im Bereich der Wiederherstellung auf den Logging-Aspekt. Alle damit verbundenen Merkmale sind in Abbildung 4.2 dargestellt.

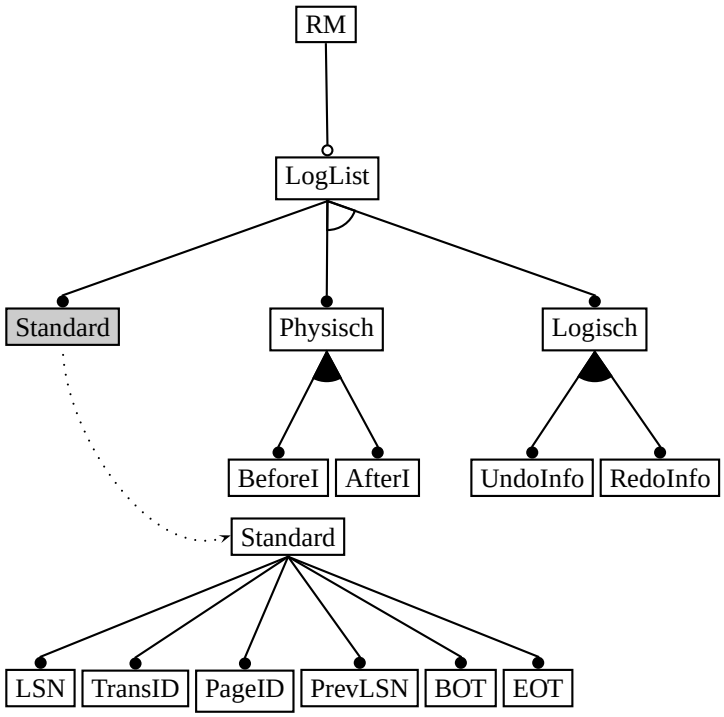

Abbildung 4.2: reduziertes Merkmalsdiagramm Recovery-Manager

4.2 Software-Architektur

In diesem Abschnitt sollen die strukturgebenden Informationen für die gewünschte Programmfamilie entwickelt werden. In Abschnitt 3.5 wurden erste architekturspezifische Festlegungen verabschiedet. Demnach soll jedes Mitglied der Programmfamilie in Form einer Funktionsbibliothek angeboten und unter Verwendung der dazugehörigen API genutzt werden. Vor der Nutzung muss der Entwickler die Anwendung, den Anforderungen entsprechend, konfigurieren. Mit Abschluss der Konfiguration steht der Funktionsumfang des Programms fest. Er wird demnach zum Erstellungszeitpunkt der Transaktionsverwaltungsbibliothek festgelegt. Möchte der Entwickler weitere Funktionen, die innerhalb der Programmfamilie verfügbar sind, hinzufügen, so muss das System einem neuen Konfigurationsprozess unterzogen werden. Die Nutzung der resultierenden Bibliothek erfolgt auf Quellcode-Ebene.

Da das System in eingebetteten Systemen eingesetzt wird, wo der sparsame Umgang mit vorhandenen Ressourcen erwünscht ist, muss darauf geachtet werden, dass die Granularität der angebotenen Funktionen besonders fein ist. Diese Eigenschaft, in Verbindung mit den Forderungen der Merkmalsdiagramme (Gruppierungen, optionale und geforderte Merkmale), wird unter anderem durch die Definition geeigneter Entwurfsregeln überwacht.

4.2.1 Grundlegende Datenstrukturen

Da sich die Ausgangssituation so darstellt, dass es an dringend benötigten Datenstrukturen für das Testen und den unabhängigen Betrieb des Transaktionsverwaltungssystems mangelt, muss dafür gesorgt werden, dass diese Defizite vor Beginn der konkreten Systemumsetzung beseitigt werden. Fundamentaler Gegenstand der Entwicklung ist die Bereitstellung einer geeigneten Struktur zur Speicherung von Datenobjekten. Die Objekte werden zur Erprobung testweise generierter Transaktionsverwaltungssysteme benötigt. Die Speicherstruktur simuliert die Funktion des Datenbankenpuffers und erlaubt die regelgerechte Manipulation (bezüglich Sperren und Ausführung der Transaktionsoperationen) der Testdaten, Somit kann auf die Dienste eines Speicher-Managers und einer realen Datenbasis verzichtet werden. Das Transaktionsverwaltungssystem kann als unabhängige (engl.: stand-alone) Applikation erprobt werden.

Für die Speicherung der Testdaten bieten sich B+-Bäume an. Dabei handelt es sich um Strukturen, die in ihren Blättern das Datenobjekt speichern, während in allen übrigen Knoten Indexstrukturen (für den effektiven Zugriff auf das Datenobjekt) angeordnet werden. Da in den einzelnen Knoten mehrere Indizes bzw. Datenobjekte gespeichert werden sollen, wird zur Organisation der Elemente eine zweite Datenstruktur benötigt. Dieses Defizit wird durch verkettete Listen, die verschiedene Operationen zur Verwaltung der gespeicherten Elemente anbieten, ausgeglichen.

Bei der Wahl der grundlegenden Datenstrukturen werden nicht nur Belange der formulierten Zielvorgaben berücksichtigt. Auch wenn der entwickelte Prototyp keine Merkmale für den Bereich der Sperrgranulate beinhalten wird, so bilden B+-Bäume dennoch eine vorzügliche Testgrundlage (betrifft das Sperren von Teilbereichen einer Datenbasis, was durch hierarchische Baumstrukturen sehr gut abstrahiert werden kann) für die spätere Implementierung dieser Features.

4.2.2 Transaktionsorganisation

Die im Transaktions-Manager eintreffenden Transaktionsoperationen müssen, für die weitere Verarbeitung im Rahmen der Transaktionsverwaltung, in geeigneten Verwaltungsstrukturen (gemäss Abbildung 4.3 als Komponente *Transaktion* bezeichnet) organisiert werden, damit eine systematische und korrekte Ausführung gewährleistet werden kann. Es sei darauf hingewiesen, dass im Folgenden die Namensgebungen für Komponenten und Merkmale der entwickelten Merkmalsdiagramme synonym verwendet werden. Diese Vereinbarung ist sinnvoll, da eine einheitliche Namensgebung die Verständlichkeit der Komponentendiagramme verbessert. Aufgrund der Tatsache, dass Merkmale ihrerseits bestimmte Funktionalitäten kapseln und im Rahmen von Vater-Kind-Beziehungen weitere Merkmale bedingen, können sie durchaus als kleinste Komponenten aufgefasst werden.

Zunächst werden die Operationen auf einem Stack, der dem FIFO-Prinzip (FIFO: **F**irst **I**n **F**irst **O**ut) gehorcht, abgelegt. Dadurch wird die Einhaltung der relativen Operationsreihenfolge bei der Ausführung sichergestellt. Weiterhin benötigt *Transaktion*

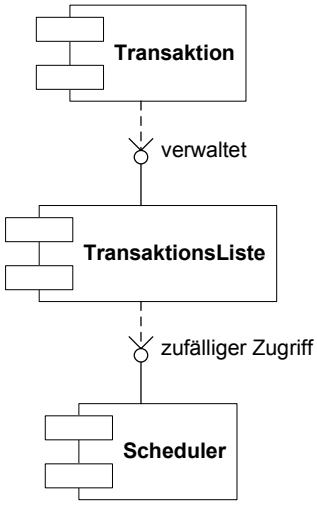

Abbildung 4.3: Komponenten des Transaktionsmanagers

zusätzliche Funktionen zum Setzen und Ändern von Statusinformationen der Transaktion. Um die einzelnen Transaktionen in Beziehung zu setzen und ihre Verwaltung zu organisieren, wird eine Listenstruktur (in Abbildung 4.3 als *TransaktionsListe* bezeichnet) verwendet. Die Liste verwaltet die aufgenommenen Transaktionen und stellt damit den Pool der auszuführenden Transaktionsoperationen bereit, aus dem der Scheduler zufällig Operationen rekrutiert und damit seinen Input-Schedule bestückt. Die Beziehungen der einzelnen Komponenten in Abbildung 4.3 verdeutlichen die Abhängigkeiten und damit die Implementierungsreihenfolge (in Richtung der Abhängigkeiten). Die daraus resultierenden Zusammenhänge werden weiter unten durch Entwurfsregeln beschrieben.

4.2.3 Scheduler

Gegenstand dieses Abschnitts ist der Architekturentwurf für die Merkmale des Schedulers. Hier muss die Architektur erstmalig semantische Zusammenhänge, die sich aus der Gruppierung von Merkmalen ergeben, berücksichtigen. Wie in Abschnitt 4.2.2 beschrieben, fordert der Scheduler das Vorhandensein einer Transaktionsliste, welche die aktuell zu bearbeitenden Transaktionen beinhaltet.

Zur Steuerung der Transaktionsstati muss der Scheduler die Komponente *Admin TransStatus* bereitstellen. Wie Abbildung 4.4 verdeutlicht, übernimmt diese Komponente sowohl das Setzen, als auch das Abfragen der aktuellen Statusinformation. Diese Fähigkeit ist Voraussetzung für den Betrieb der Transaktionsausführungsprotokolle.

Im nächsten Schritt wird ein allgemeines Protokoll entwickelt, das erste Routinen für die Organisation der Ausführung umfasst. Dieses Protokoll ist noch nicht in der Lage, eine vollständige Transaktionsausführung zu administrieren. Abhilfe schafft die Dreingabe der fehlenden Methoden durch die Komponenten *Sperre* und/oder *Seriell*. Die Kom-

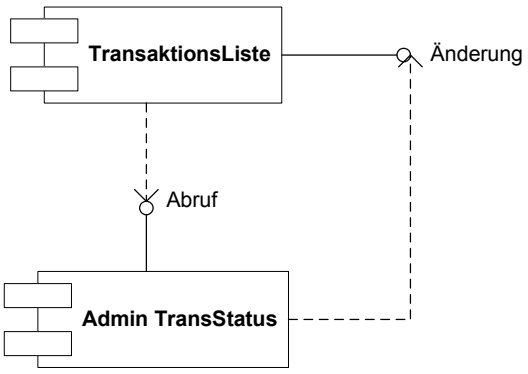

Abbildung 4.4: Abhängigkeiten vs. Transaktionsstatus

ponente *Seriell* gewährleistet die serielle Ausführung der eingehenden Transaktionen, während *Sperre* die Grundlage für eine parallele Transaktionsausführung, basierend auf einer Sperrverwaltung, bildet. Die Art der Gruppierung signalisiert, dass jede nicht-leere Auswahlmenge bezüglich der beiden Komponenten zulässig ist.

Abbildung 4.5 veranschaulicht den Zusammenhang. Die Generator-Komponente stellt aus dem Transaktionslisteninhalt zulässige Input-Schedules zusammen. Ist dies geschehen, so muss entschieden werden, welche der verfügbaren Ausführungsprotokolle genutzt werden sollen (zur Ansteuerung des gewählten Verfahrens muss eine geeignete Schnittstelle bereitgestellt werden). Dabei darf für jeden Input-Schedule nur ein Protokoll gewählt werden, da der Schedule nicht mehrmals ausgeführt werden soll. Das Arbeitsergebnis des gewählten Protokolls ist das Gegenstück des Inputs - der Output-Schedule. Dieser wird vom Recovery-Manager entgegengenommen und dient als Grundlage für die Prozesse des Recovery-Managements.

Die Sperr-Komponente bietet, ähnlich wie das allgemeine Ausführungsprotokoll, nicht genügend Funktionen, um einen gegebenen Schedule sicher zu verarbeiten. Sie stellt lediglich Routinen zum Setzen, Aufheben und Verwalten von Sperren bereit (allgemeine Sperrdisziplin). Trotz dieser Regeln können verschiedenartige Mehrbenutzer-Anomalien auftreten. Dem kann durch die verschiedenen in Abbildung 4.6 dargestellten Sperrprotokolle, mehr oder weniger, entgegengewirkt werden. Die Protokolle bauen auf der allgemeinen Sperrdisziplin auf, und erweitern diese um Regeln, die den Zeitpunkt und die Prozedur des Sperrens betreffen. Das Transaktionsverwaltungssystem kann so konfiguriert werden, dass die Funktionalität mehrerer Sperrprotokolle (gemäss Gruppierung gilt: jede nicht-leere Auswahlmenge stellt eine korrekte Zusammenstellung dar) integriert werden kann. Jedes einzelne Protokoll kann über eigene Schnittstellen angesprochen werden. In diesem Zusammenhang muss darauf geachtet werden, dass der aktuelle Input-Schedule am gewählten Sperrprotokoll anliegt. Der, gemäss vorliegendem Sperrprotokoll, generierte Output-Schedule wird an den Recovery-Manager weitergegeben.

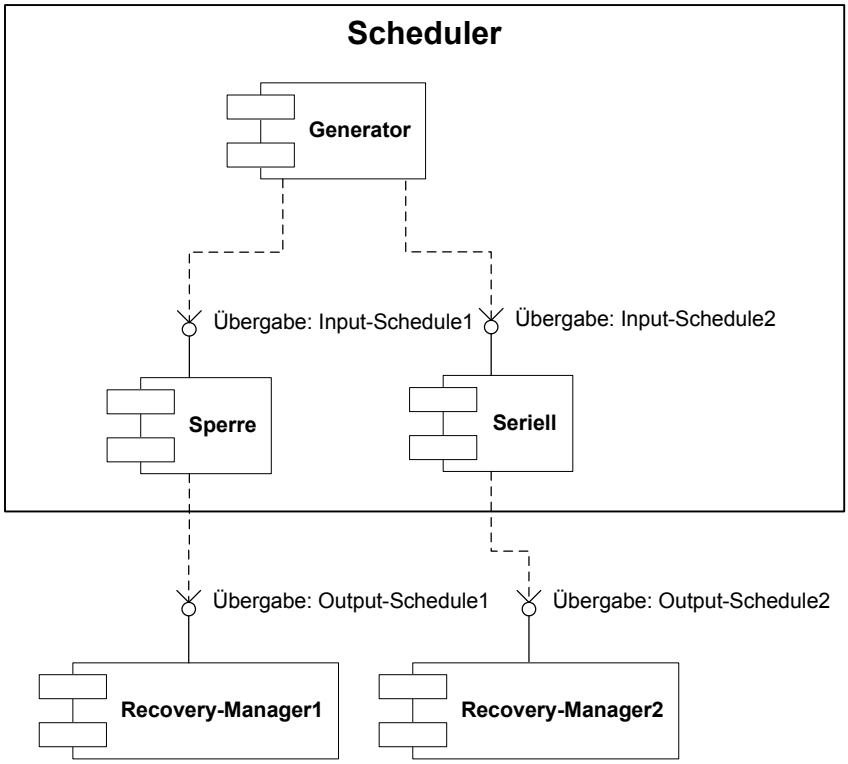

Abbildung 4.5: alternative Ausführungsprotokolle

4.2.4 Protokollierung der Transaktionsausführung

In Abschnitt 4.1 wurde die Reduktion der Prototypenfunktionalität beschlossen. Demnach konzentriert sich die Entwicklung im Bereich des Recovery-Managers auf den Themenkomplex Logging. Auf die Implementierung der Ausführungsfunktionalität für die Operationen des Output-Schedules wird verzichtet. Da Log-Buch-Einträge mit der Ausführung von Transaktionsoperationen gekoppelt sind (zumindest für schreibende Operationen und unter Verwendung von Undo- und/oder Redo-Protokollen), ist es erforderlich die Operationsausführung zu abstrahieren, um den Zeitpunkt des Eintrags zu bestimmen (möglich: Log-Buch-Eintrag unmittelbar nach Operationsausführung). Aus diesem Grund wird für die prototypische Entwicklung festgelegt, dass die Herausgabe der Output-Operationen durch den Scheduler mit deren Ausführung gleichgesetzt wird.

Das Log-Buch setzt sich aus den, in Abbildung 4.2 aufgeführten, Merkmalen zusammen. *LogList* symbolisiert das Gerüst der Datenstruktur. *Standard* fügt der Datenstruktur Spalten hinzu, deren Inhalt für den Recovery-Prozess in jedem Fall benötigt wird. Aufbauend auf der Implementierung der grundlegenden Listenfunktionen und -spalten erfolgt die Implementierung der alternativen Protokollierungsformen (*Physisch* und *Logisch*). Für Redo- und Undo-Informationen werden der Liste, analog zu *Standard*, weitere Spalten hinzugefügt, deren Inhalt der gewählten Protokollierungsart entspricht.

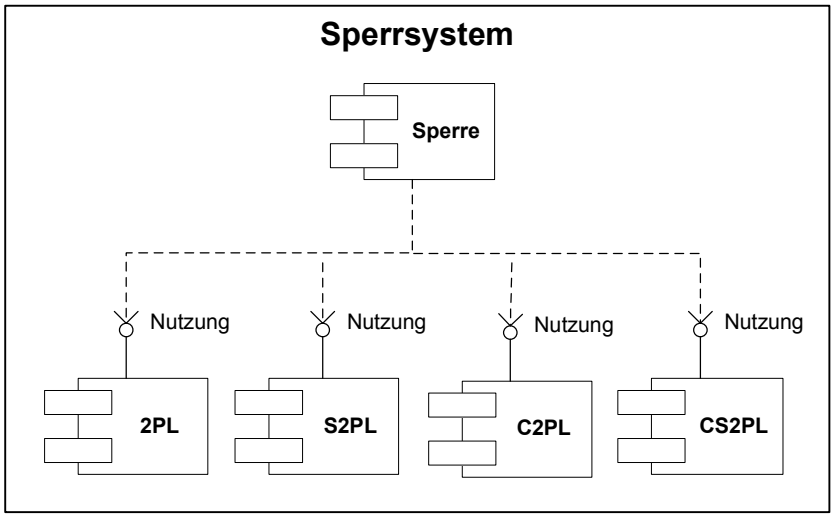

Abbildung 4.6: konkrete Sperrprotokolle auf Basis der allgemeinen Sperrdisziplin

Alle in diesem Abschnitt (4.2) diskutierten Funktionen, werden mittels merkmalsorientierter Implementierungstechniken umgesetzt. Auch die Implementierung der Logging-Funktionalität kann auf diese Weise erfolgen. Eine Sonderstellung nimmt dem gegenüber die Nutzung (Log-Buch-Eintragung) der Funktionen ein. Die entsprechenden Aufrufe erfolgen an vielen verschiedenen Stellen im Programm. Es ist daher am effizientesten, die Aufrufe unter Verwendung aspektorientierter Technologien zu implementieren. Abbildung 4.7 verdeutlicht das Vorgehen. $Aufruf$ wird als Aspekt implementiert. Durch die Verwendung adäquater *Pointcuts* wird der Code von $Aufruf$ (Stichwort: *Advice*) überall dort im Programm eingefügt, wo ein Log-Buch-Eintrag erfolgen muss.

4.2.5 Entwurf des Schichtenmodells

Die Beschreibung der Funktionen durch Merkmalsdiagramme ist der erste Schritt, um Zusammenhänge und Beziehungen zwischen den verschiedenen Programmteilen zu spezifizieren. Jedoch reichen die gewonnenen Informationen nicht aus, um Beziehungen, die losgelöst von der Semantik der hierarchischen Baumstrukturen und Gruppierungen der Diagramme existieren, zu beschreiben. Aus diesem Grund wird aufbauend auf den Theorien zu GenVoca ein Schichtenmodell und die dazu passende Grammatik für den Prototypen entwickelt.

Für die Entwicklung und Nutzung des zusammengestellten Transaktionsverwaltungssystems ergeben sich in der Praxis nicht zu vernachlässigende Besonderheiten, denen Rechnung getragen werden muss. FeatureC++ kombiniert die hinzugefügten Verfeinerungen mit den bestehenden Funktionen einer Klasse so, dass im Ergebnis eine terminale Klasse bereitsteht. Während Funktionen früherer Schichten im reinen GenVoca-Ansatz nur auf Verfeinerungen einer Klasse zugreifen können, die bis zu der entsprechenden

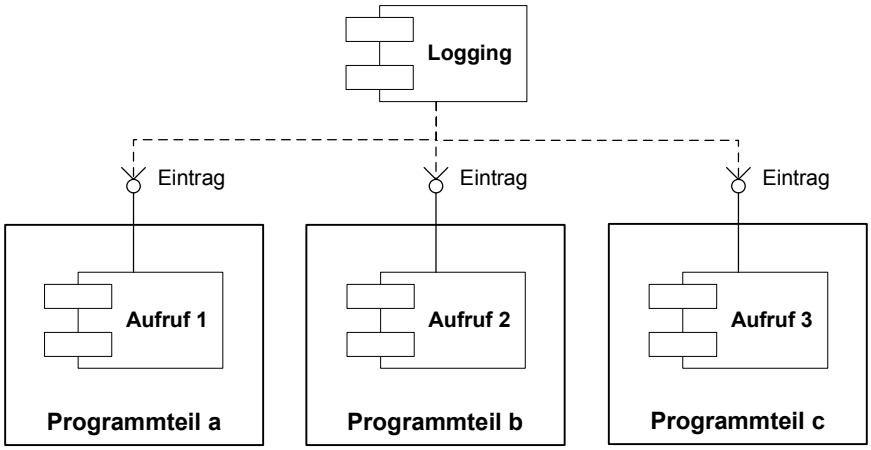

Abbildung 4.7: Logging-Komponente vs. Aufruf

Schicht eingefügt wurden, ist es den Funktionen in FeatureC++ möglich, auch auf Verfeinerungen nachfolgender Schichten zuzugreifen.

In jeder, der in Abbildung 4.8 dargestellten Schichten, wird ein neues Merkmal (in den Aspectual Mixin Layers wird zusätzlich der Aspect *Logging* behandelt) eingefügt, welches auf den Merkmalen der oberen Schichten aufbaut. Im Schichtenmodell zum Prototypen repräsentieren einige Schichten (aus Gründen der Übersichtlichkeit) mehrere Merkmale. Dabei handelt es sich um Merkmale, deren gemeinsame Einführung sich anbietet, weil ihre Funktionen sich gegenseitig bedingen. So implementiert beispielsweise die Schicht *Base_Scheduler*, in Addition zu den Scheduling-Mechanismen (Klasse *Scheduler* siehe Abbildung 4.8), gleichzeitig Funktionen der Komponente $Transaktion$.

Die Schichten *Base* bis *Base_Recovery_Manager* implementieren Funktionen, die durch feature-orientierte Implementierungstechniken realisiert werden (Bezeichnung: *Mixin Layers* (Abschnitt 2.3.4)). Dem gegenüber gibt es Funktionen, die effektiver durch Aspekte beschrieben werden können. Dazu gehören die Protokollierungsaufrufe des Recovery-Managers.

Die Abgrenzung der Aspekte von den herkömmlichen Merkmalsschichten erfolgt durch die Einführung von *Aspectual Mixin Layers* (*Log_Recovery_Manager* bis *Log_Recovery_Manager_After_Before*). Schichten, in denen der Programm-Code der *Aspectual Mixin Layers* wirkt (Stichwort: *Aspect Weaving*), werden durch Pfeilbeziehungen gekennzeichnet. Wird das physische Loggen favorisiert, so übernimmt *Log_Recovery_Manager_After_Before* die Pfeilbeziehungen der Alternative (*Log_Recovery_Manager_Redo_Undo*).

Zu dem entwickelten Schichtenmodell wird nachfolgend die den Festlegungen entsprechende Grammatik angegeben. Darin stellen P_0 bis $P_{Transaktionsverwaltungssystem}$ Produktionsregeln dar, die definieren, in welcher Form die Schichten kombiniert werden können. Bei der Mengenbeschreibung in P_9 handelt es sich nicht um *Realms* (gemäss Sektion 2.3.3). Vielmehr soll die Vielfalt der Kombinationsmöglichkeiten angedeutet werden.

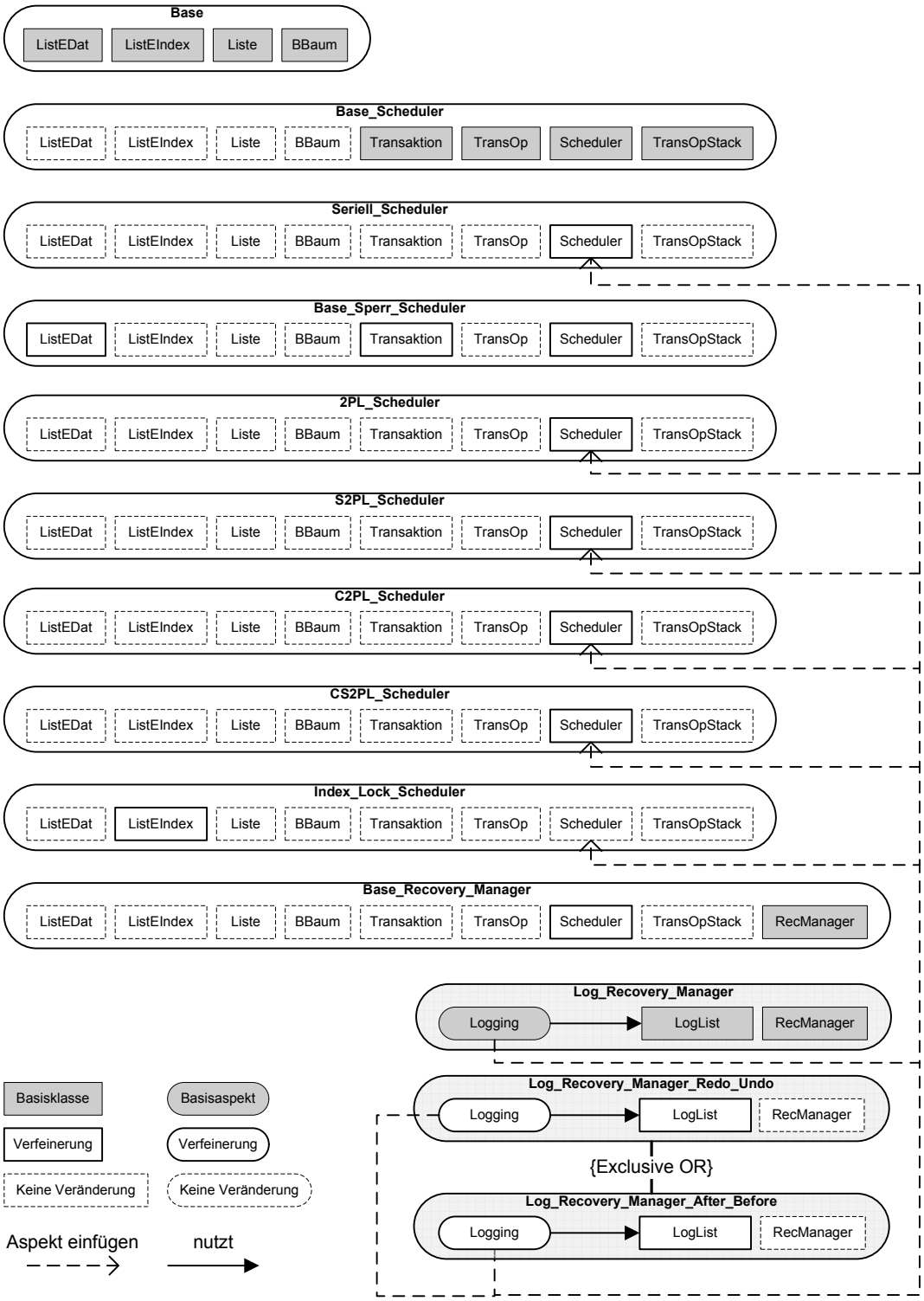

Abbildung 4.8: Schichtenmodell des Prototypen

$$
\begin{aligned}
P_0 &::= Base\ ; \\
P_1 &::= Base\_Scheduler\ P_0\ ; \\
P_2 &::= Seriell\_Scheduler\ P_1\ ; \\
P_3 &::= Base\_Sperr\_Scheduler\ P_1\ ; \\
P_4 &::= 2PL\_Scheduler\ P_3\ ; \\
P_5 &::= S2PL\_Scheduler\ P_3\ ; \\
P_6 &::= C2PL\_Scheduler\ P_3\ ; \\
P_7 &::= CS2PL\_Scheduler\ P_3\ ; \\
P_8 &::= Index\_Lock\_Scheduler\ P_4\ P_5\ P_6\ P_7\ ; \\
P_9 &::= Base\_Recovery\_Manager\ \{\ P_2,\ P_4,\ P_5,\ P_6,\ P_7\ \} \mid \\
&\quad\ Base\_Recovery\_Manager\ \{\ P_2,\ P_8\ \}\ ; \\
P_{10} &::= Log\_Recovery\_Manager\ P_9\ ; \\
P_{11} &::= Log\_Recovery\_Manager\_Redo\_Undo\ P_{10}\ ; \\
P_{12} &::= Log\_Recovery\_Manager\_After\_Before\ P_{10}\ ; \\
P_{Transaktionsverwaltungssystem} &::= Log\_Recovery\_Manager\_Redo\_Undo\ P_{10} \mid \\
&\quad\ Log\_Recovery\_Manager\_After\_Before\ P_{10}\ ;
\end{aligned}
$$

4.2.6 Entwurfsregelprüfung

Regelgrammatiken, wie sie in der vorhergehenden Sektion entwickelt wurden, beschreiben lediglich die kombinatorischen Möglichkeiten der Programmkonfiguration. Eine automatisierte Überprüfung erfolgt nicht. Im Rahmen der Konfiguration des Transaktionsverwaltungssystems wäre eine autonome Validierung wünschenswert, da nicht davon ausgegangen werden kann, dass die Klientel der Nutzer die tieferen Zusammenhänge der Schichtenkombination kennt. FeatureC++ integriert die in der *Ahead Tool Suite* enthaltene Entwurfsregelprüfung. In jeder Schicht wird eine Datei (*.drc) hinterlegt, die mit geeigneten Informationen versehen ist.

Listing 4.1 zeigt die Entwurfsregeln für die *Base*-Schicht. Es wird eine boolesche Variable *basis* (Zeile: 4) definiert, welche, ausgehend von einer Kompositionsgleichung nach rechts weitergegeben wird. Gleichzeitig wird *Basis_SchedulerL* (Zeile: 5) importiert. Dabei handelt es sich um eine von links hereingereichte Variable aus einer der Folgeschichten. Die Schlüsselworte *provides* und *requires* kennzeichnen die angebotenen, bzw. geforderten Variablen.

Listing 4.1: Base

```
1  single layer Base;
2
3  //define/use attributes
4  flowright Bool basis;
5  extern flowleft Bool Basis_SchedulerL;
6
7  //provides and requires
8  provides flowright basis;
9  requires flowleft Basis_SchedulerL;
```

Listing 4.2: Base_Scheduler

```
1  single layer Base_Scheduler;
2
3  //define/use attributes
4  extern flowright Bool basis;
5  flowleft Bool Basis_SchedulerL;
6  flowright Bool Basis_SchedulerR;
7
8  //provides and requires
9  provides flowleft Basis_SchedulerL;
10 provides flowright Basis_SchedulerR;
11 requires flowright basis;
```

Die Entwurfsregeln für *Base_Scheduler* sind in Listing 4.2 aufgeführt. Diese Schicht fordert das Vorhandensein von *Base*. Da *Base* die Variable *basis* nach rechts weiterreicht, wird die Forderung von *Base_Scheduler* genau dann erfüllt, wenn *Base* in der Kompositionsgleichung vor dieser Schicht liegt. Durch dieses Konstrukt wird nicht nur das Vorhandensein von *Base* sichergestellt. Gleichzeitig wird einer Vertauschung der beiden Schichten vorgebeugt. Da *Base* (für sich allein betrachtet) keinerlei Funktion eines Transaktionsverwaltungssystems bereitstellt, fordert es von linker Seite die Existenz verschiedener Scheduling-Funktionen (*Basis_SchedulerL*). Auf diese Weise wird sichergestellt, dass die kleinste mögliche Konfiguration ein funktional vollständiges System bereitstellt. Die *Design Rules* der übrigen Schichten werden in Analogie zu den Beispiel-Listings und der definierten Regelgrammatik aus 4.2.5 entwickelt und sind in Anhang A.1 aufgeführt.

4.3 Zusammenfassung

In diesem Kapitel wurde die Phase des Domänenentwurfs behandelt und die einzelnen Ergebnisse diskutiert. Zunächst müssen die in Abschnitt 3.4 entwickelten Merkmalsdiagramme aus Gründen der Überschaubarkeit reduziert werden. Die Auswahl der umzusetzenden Merkmale erfolgte priorisiert. Vorrang haben Features, die eine zentrale Rolle innerhalb der Transaktionsverwaltung einnehmen. Anschliessend wurden die für die Implementierung maßgeblichen Gesichtspunkte skizziert (unter Verwendung von Komponentendiagrammen). Ausgehend von grundlegenden Datenstrukturen, über die Themenkomplexe Transaktionsorganisation und Scheduling, bis hin zur Log-Verwaltung,

erfolgen erste Schritte in Richtung der (vorerst) endgültigen Software-Architektur. Danach wird das Schichtenmodell des Prototypen kreiert. Kernpunkt der Modellierung ist die Abgrenzung der zu implementierenden Aspekte. Parallel zur Schichtenmodellierung verläuft die Entwicklung einer geeigneten Regelgrammatik. Aufbauend auf der entwickelten Grammatik werden die Voraussetzungen für die automatisierte Entwurfsregelprüfung geschaffen.

Kapitel 5

Implementierung

Die Phase der Implementierung stellt die Konkretisierung der im Entwurf getroffenen Festlegungen dar. Dieses Kapitel soll nicht dem Anspruch der Vollständigkeit genügen. Vielmehr wird der gewählte Lösungsansatz auszugsweise erläutert und auf Probleme während der Entwicklung eingegangen. Wie in den vorhergehenden Kapiteln angedeutet, setzt die Implementierung auf dem Tool *FeatureC++* auf. Dabei handelt es sich um eine Art Precompiler, der den merkmals- und aspektorientierten Quell-Code auswertet und in herkömmlichen C++-Code übersetzt. Das Tool wird in Kombination mit der Entwicklungsplattform *Microsoft Visual Studio .NET 2003* verwendet.

Die Entwicklung ist in zwei Komplexe unterteilt. Zunächst werden die Features implementiert. Diese Vorgehensweise bietet sich an, da die Merkmale hierarchisch angeordnet sind. In gewisser Weise bilden sie die Grundlage für die Einführung von Aspekten, da diese in den verschiedenen Merkmalsschichten wirken. Um den Aspekt-Code testen zu können, müssen zunächst die entsprechenden Merkmalsschichten implementiert werden. Auf Basis der implementierten Merkmale erfolgt die Realisierung der aspektorientierten Programmteile. Die Entwicklung des Prototypen orientiert sich an dem in der Entwurfsphase erarbeiteten Schichtenmodell (Abbildung 4.8) und setzt die darin enthaltenen Festlegungen originalgetreu um.

5.1 Merkmale

Zunächst werden die grundlegenden Datenstrukturen, wie B+-Bäume und Listen, implementiert. Sie bilden zusammen die *Base*-Schicht (siehe: Abbildung 4.8). Da es sich bei den in dieser Schicht enthaltenen Datenstrukturen um initiale Klassen handelt, ergeben sich gegenüber der objektorientierten Entwicklung zunächst keine gravierenden Unterschiede. Aus diesem Grund wird auf weiterführende Erläuterungen verzichtet. Wichtig im Zusammenhang mit der Verwendung von FeatureC++ ist die Tatsache, dass die Konstruktor- und Methodendefinitionen ausserhalb der Klassendefinition erfolgen müssen. Wird dies nicht beachtet, so kommt es unter anderem zu Problemen bei der Erkennung von Objekttypen.

Nach der Realisierung der *Base*-Schicht erfolgt die Umsetzung der Merkmale von *Base_Scheduler*. Hier werden erste Funktionen des Schedulers und der Transaktions-Datenstrukturen entwickelt (gemäss Abbildung 4.8 werden die Basisklassen *Transaktion*, *TransOp*, *Scheduler* und *TransOpStack* eingeführt). Listing 5.1 zeigt einen Auszug aus der Klassendefinition der Klasse *Transaktion*. Die Transaktionsoperationen werden in einem Stack organisiert. Hinzu kommen Funktionen zur Verwaltung der Statusinformation und Transaktionskennung (ID). In Listing A.12 ist der komplette Sourcecode der Basisklasse *Transaktion* dargestellt.

Listing 5.1: Klasse Transaktion (Base_Scheduler)

```
1  class Transaktion {
2
3    public:
4
5      //Trans mit Operationsliste + ID
6      Transaktion(TransOpStack*, int);
7      //leere Transaktion + ID
8      Transaktion(int);
9      //OperationsStack
10     TransOpStack* getTransOpStack();
11     //Operation von Stelle int bekommen
12     TransOp* getTransOp(int);
13     //Status
14     void setStatus(int);
15     int getStatus();
16     //naechste Operation nominieren
17     void pushTransOp(TransOp*);
18 };
```

Listing 5.2: Klasse Scheduler (Base_Scheduler)

```
1  class Scheduler {
2
3    public:
4
5      //Scheduler-Konstruktoren
6      Scheduler(TransOpStack*);
7      Scheduler();
8      //Input-Schedule
9      TransOpStack* getInput();
10     void setInput(TransOpStack*);
11     void addInput(TransOp*);
12     //erzeugt Transaktionsvektor
13     vector<Transaktion>* parseOps();
14 };
```

Der wesentliche Inhalt der Klassendefinition des Basis-Schedulers ist in Listing 5.2 dargestellt. Der Scheduler verwaltet einen Input-Schedule (Stack), der entweder zum Zeitpunkt der Initialisierung übergeben oder schrittweise, durch das Hinzufügen einzelner Operationen, aufgebaut wird. Der dazugehörige Quell-Code ist Listing A.13 zu entnehmen. Bei den beiden beschriebenen Klassen handelt es sich um Initial-Klassen, deren Implementierung noch keine grösseren Besonderheiten birgt. Wird der Prototyp jedoch, in

einem der folgenden Entwicklungsschritte, um die Funktionalität der sperrenden Transaktionsausführung erweitert, so werden die Unterschiede zur objektorientierten Entwicklung deutlich. Listing 5.3 beschreibt diese Abweichungen. In der *Base_Sperr_Scheduler*-Schicht (siehe Abbildung 4.8) wird die Datenstruktur *Transaktion* um Verwaltungsroutinen für Lock-Informationen ergänzt. Die in früheren Schichten implementierten Funktionen zu *Transaktion* sind, gemäss Vererbungshierarchie, weiterhin (falls nicht von neuen Konstrukten überdeckt) nutzbar. Zusätzlich müssen weitere Datenstrukturen für den Einsatz der Sperrverfahren modifiziert werden. Beispielsweise wird der Lock-Status eines Datums direkt im Objekt gespeichert (Verfeinerung von *ListEDat* gemäss Abbildung 4.8).

Listing 5.3: Refinement Transaktion (Base_Sperr_Scheduler)

```
1   refines class Transaktion {
2
3     public:
4
5       //verwaltet gelockte Datenobjekte
6       vector<ListEDat>* getLockVec();
7       //fuegt gelockte Elemente hinzu
8       void pushLockInfo(ListEDat*);
9       //gibt gelockte Elemente frei
10      ListEDat* popLockInfo();
11  };
```

Auch die *Scheduler*-Klasse muss innerhalb der *Base_Sperr_Scheduler*-Schicht durch weitere Funktionen (Listing 5.4) zur Unterstützung der Sperrprozesse ergänzt werden.

Listing 5.4: Refinement Scheduler (Base_Sperr_Scheduler)

```
1   refines class Scheduler {
2
3     public:
4
5       //naechste Operation vom Input-Schedule nominieren
6       TransOp* schedule_pop(TransOpStack*);
7   };
```

Die verschiedenen Sperrprotokolle werden in *Scheduler* implementiert. Listing 5.5 deutet die Einführung des 2Phasen-Sperrprotokolls an (Zeile 6). Da der Umfang des dazugehörigen Sourcecodes (gerade der interessanten Passagen) recht gross und folglich die Lesbarkeit stark eingeschränkt ist, muss auf eine vollständige Präsentation verzichtet werden. Alle übrigen Ergänzungen (S2PL, C2PL, CS2PL und Sperrgranulate) erfolgen analog zu den angegebenen Listings in den dafür zuständigen Schichten (siehe Abbildung 4.8).

Listing 5.5: Refinement Scheduler (2PL_Scheduler)

```
1  refines class Scheduler {
2
3    public:
4
5      //2Phasen-Sperrprotokoll
6      void _2PL();
7  };
```

5.2 Aspekte

Während der vorhergehenden Phasen der Domänenentwicklung stellte sich heraus, dass das Protokollieren der Transaktionsausführungen am effektivsten unter Verwendung des aspektorientierten Programmierparadigmas implementiert werden kann. Datenstrukturen des Themenkomplexes, die sich nahtlos in die hierarchischen Strukturen der Merkmale (Log-Listen, Recovery-Manager, etc.) einfügen, werden weiterhin als Features implementiert (jedoch werden sie nun (teilweise) in *Aspectual Mixin Layers* organisiert). Der Protokollierungsaufruf erfolgt dem gegenüber an verschiedenen Stellen im Programm-Code und setzt sich somit über die bestehenden hierarchischen Strukturen hinweg (siehe Abbildung 4.8). Daher wird er als Aspekt implementiert. Die Datenstruktur, die durch den Aspekt *Logging* manipuliert werden soll, heisst *LogList* (siehe Abbildung 4.8) und wird in Aspectual Mixin Layer *Log_Recovery_Manager* eingeführt. Listing 5.6 skizziert die enthaltenen Funktionen der Datenstruktur (der vollständige Quellcode ist dem Anhang (A.15) zu entnehmen).

Listing 5.6: Klasse LogList (Log_Recovery_Manager)

```
1  class LogList {
2
3    public:
4
5      //Konstruktor
6      LogList();
7      //Log-Eintrag
8      void newEntry(TransOp*, TransOpStack*);
9      //Log-Eintrag anschauen
10     void showEntry(int);
11 };
```

Listing 5.7 enthält einen Auszug des initialen Log-Aspekts (wird in der Schicht *Log_Recovery_Manager* implementiert). Innerhalb der Methode *logExecution* werden die für Log-Eintragungen nötigen Prozesse ausgeführt. Dem Signalwort *virtual* kommt eine besondere Bedeutung zu. Bei der späteren Verfeinerung des Aspekts *Log_Recovery_Manager* (genauer: *logExecution*) traten Schwierigkeiten auf. FeatureC++ konnte die aktuelle Ausprägung von *logExecution* nicht korrekt referenzieren, wenn die Methode nicht als virtuell angegeben wurde. Es wurde in jedem Fall nur die Basis-Methode ausgeführt. Unter Verwendung von *virtual* trat dieses Problem nicht mehr

auf. Ist schon während der Implementierung eines Aspekts klar, dass keine Verfeinerung erfolgen wird, so kann auf die Nutzung von virtuellen Methoden verzichtet werden, da es nur eine einzige Ausprägung der Methode gibt, die referenziert werden kann. Um den Ort, an dem *logExecution* ausgeführt werden soll, zu beschreiben, wird ein entsprechender *Pointcut* (Zeilen 10-13) angegeben. Die Methode *execution* erwartet als Parameter einen String, der die Einfüge-Information enthält. Im Beispiel erfolgt die Ausführung von *logExecution* an jeder Stelle im Programm, an der die Methode *schedule_pop* aufgerufen wird. Durch die Verwendung der Funktionen *result*, *that* und *args* können Objekte aus der Einfüge-Umgebung an den Aspekt-Code weitergereicht werden.

Listing 5.7: Aspekt Logging (Log_Recovery_Manager)

```
1   aspect Logging {
2
3       virtual void logExecution(TransOp* nam, Scheduler* sch_nam,
4                                 TransOpStack* op_stac)
5       {
6           //Ausfuehrung des Log-Eintrags
7       }
8
9       //Pointcut
10      pointcut log(TransOp* name, Scheduler* sch_name,TransOpStack* op_stack) =
11       execution("%_Scheduler::schedule_pop(...)") && result(name)
12                                                   && that(sch_name)
13                                                   && args(op_stack);
14
15      //Advice
16      advice log(name, sch_name, op_stack) : after(TransOp* name,
17                                                   Scheduler* sch_name,
18                                                   TransOpStack* op_stack)
19      {
20          logExecution(name, sch_name, op_stack);
21      }
22  };
```

Der Advice-Code umfasst die Gesamtheit aller einzufügenden Operationen (hier nur: *logExecution* (Zeile 20)). Das Attribut *after* gibt an, dass der Programm-Code nach dem Auftreten von *schedule_pop* eingefügt wird.

Listing 5.8: LogList (Log_Recovery_Manager_Redo_Undo)

```
1   refines class LogList {
2
3     public:
4
5       //Konstruktor-Refinement
6       LogList();
7       //Redo-Info
8       void setRedo(string);
9       //Undo-Info
10      void setUndo(string);
11      //Ausgabe des Log-Eintrags
12      void showEntry(int);
13  };
```

Da ein Log-Eintrag auch Redo- und/oder Undo-Informationen (alternativ: After- und/oder Before-Images) enthalten sollte, müssen *LogList* und *Logging* um entsprechende Funktionen erweitert werden. In den Zeilen 8 und 10 von Listing 5.8 wird *LogList* um die Merkmale *Redo* und *Undo* erweitert (Anhang A.16 enthält den vollständigen Sourcecode). Listing 5.9 spiegelt den Vorgang der Verfeinerung des Aspekts wieder. Zunächst werden die für den Eintrag der Redo- und/oder Undo-Information benötigten Prozeduren aufgerufen (Zeilen 5-7). Darauf hin erfolgt die Ausführung der im Eltern-Aspekt definierten *logExecution*-Methode (Zeile 10). Weil der Aspekt-Code der Verfeinerung ebenfalls an der in Listing 5.7 definierten Stelle eingefügt werden soll, kann auf eine Verfeinerung von Pointcut *log* verzichtet werden. Die Verfeinerung beschränkt sich auf die Modifikation der *logExecution*-Methode.

Listing 5.9: Refinement Logging (Log_Recovery_Manager_Redo_Undo)

```
1   refines aspect Logging {
2
3       void logExecution(TransOp* nam, Scheduler* sch_nam, TransOpStack* op_stac)
4       {
5           //Erweiterung des Log-Eintrags
6           //durch Hinzufuegen von Redo-
7           //und Undo-Information
8
9           //Methode des Eltern-Aspekts aus Layer Log_Recovery_Manager
10          super::logExecution(nam, sch_nam, op_stac);
11      }
12  };
```

5.3 Zusammenfassung

Thema dieses Kapitels war die beispielhafte Beschreibung der Prototypen-Implementierung. Der Prozess der Implementierung gliedert sich in Merkmals- und Aspektumsetzung. Aufgetretene Probleme bei der Realisierung der Merkmale betreffen die Erkennung von Objekttypen. Abhilfe schafft die externe Konstruktor- und Methodendefinition (ausserhalb der Klassendefinition).

Die Aspekt-Implementierung beschränkt sich beim Prototypen auf das Protokollieren der Transaktionsausführung. Verfeinerungen des Aspekts setzen voraus, dass die Methode *logExecution* des Basis-Aspekts als virtuell angegeben wird. Andernfalls kann die aktuell benötigte Version von *Logging* nicht korrekt erkannt und ausgeführt werden. Abbildung 5.1 deutet die Abhängigkeiten zwischen den einzelnen Klassen des Prototypen an. Es ist ersichtlich, dass die bestehenden Abhängigkeiten, je nach Konfiguration des Prototypen, variieren. Wird beispielsweise auf das Protokollieren der Transaktionsausführung verzichtet, so existieren die mit *LogList* verbundenen Abhängigkeiten nicht.

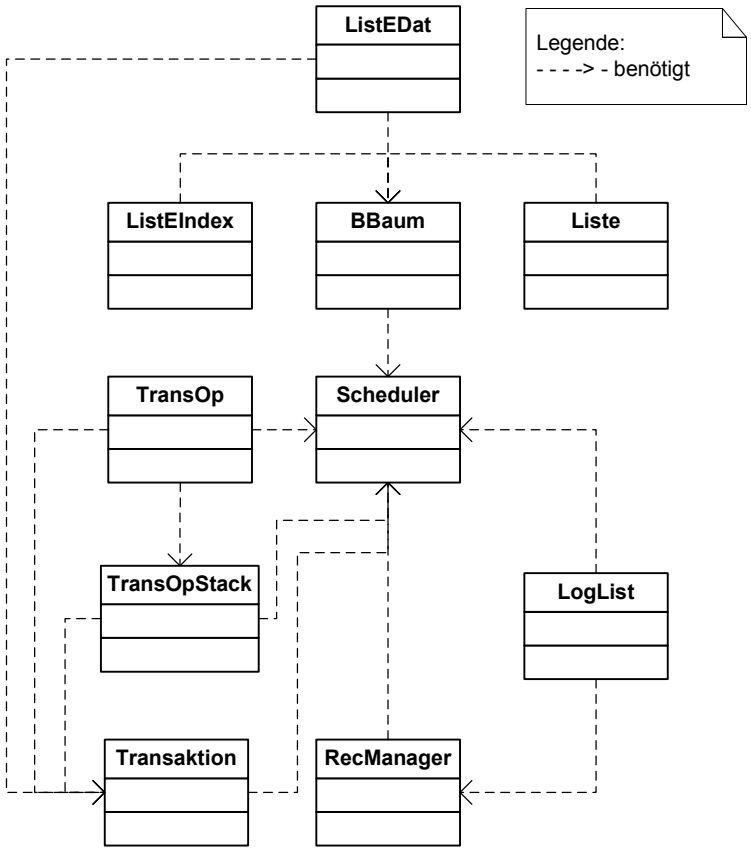

Abbildung 5.1: Klassendiagramm einer möglichen Prototypenkonfiguration

Kapitel 6

Evaluierung

Mit Abschluss der Implementierungsphase sind die ausgewählten Funktionen der Programmfamilie vollständig realisiert und stehen zur weiteren Verwendung bereit. Da die Funktionen teilweise abstrahiert oder idealisiert umgesetzt wurden ist es schwierig praxisnahe Tests zu konzipieren, die Aufschluss über die Leistungsfähigkeit des Prototypen geben (etwa: Effizienz der verwendeten Algorithmen). Zudem wurden derartige Leistungsaspekte in der Vergangenheit schon vielfach diskutiert (beispielsweise befassen sich Saake und Heuer [Saa99] u.a. mit Ansätzen zur Optimierung von Transaktionsverwaltungsprozessen). Daher soll in dieser Arbeit auf eine erneute Vertiefung verzichtet werden. Vielmehr zielt dieses Kapitel auf die Bewertung der Flexibilität (Stichworte: Konfigurierbarkeit, Speicherbedarf, etc.) der Software-Architektur und den sich daraus ergebenden Vor- und Nachteilen, auch im Vergleich mit Architekturen alternativer Systeme, ab.

6.1 Konfigurierbarkeit

Die flexible Konfigurierbarkeit auf Basis der entwickelten Entwurfsregeln ist eine der Hauptforderungen, die an die Programmfamilie gestellt wurden. Aus diesem Grund ist es notwendig, die Leistungsfähigkeit des Prototypen auf diesem Gebiet zu bewerten. Wie Abbildung 4.8 zeigt, kann der Prototyp mit Funktionen versehen werden, die in 13 verschiedenen Schichten organisiert sind. Jede Schicht kann genau einmal ausgewählt werden. Wird das Regelwerk zur Konfiguration nicht durch spezielle Festlegungen (u.a. Entwurfsregeln) eingeschränkt, so ergibt sich eine grosse Zahl an möglichen Kombinationen. Die Bildungsvorschrift für die Ermittlung des genauen Ergebnisses leitet sich aus dem Beispiel der Urnenziehung ohne Zurücklegen ab und lautet wie folgt:

$$Kombinationen_{ultd} = \sum_{1 \leq k \leq 13} \frac{n!}{(n-k)!}$$

Dabei beschreibt n die Gesamtzahl der Schichten, während k die Anzahl der gewähl-

ten Schichten widerspiegelt. Die Zwischenergebnisse werden addiert. Es ist sofort klar, dass nur ein geringer Teil der Kombinationen im Kontext der Transaktionsverwaltung ein sinnvolles Ergebnis darstellt. Beispielsweise sind alle Konfigurationen in denen die *Base_Scheduler*-Schicht fehlt unzulässig, weil die Scheduler-Erweiterungen anderer Layer auf den Funktionen (Klassen) dieser Schicht aufbauen. Ebenso, wie auf das bloße Vorhandensein einzelner Schichten, kommt es auf die Reihenfolge an, in der diese eingefügt werden. Es treten immer dann Fehlermeldungen auf, wenn eine Schicht eingeführt wird, die Funktionen noch nicht vorhandener Schichten nutzt. Es ist daher sinnvoll die Kompositionsgleichung gemäss der Implementierungsreihenfolge aufzustellen (siehe Abbildung 4.8 beginnend mit *Base* abwärts), da auf diese Weise derartige Konflikte vermieden werden.

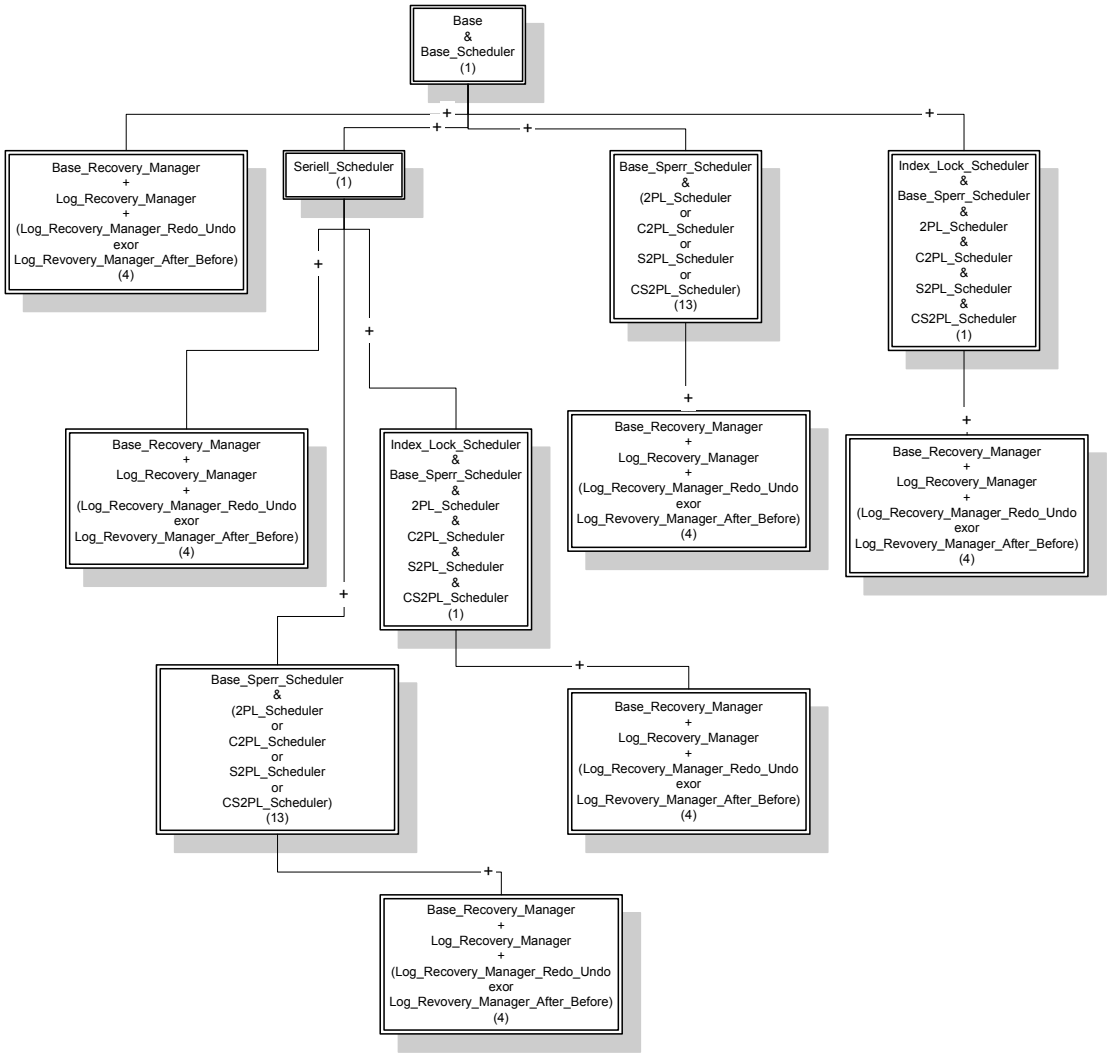

Abbildung 6.1: Kombinationsbaum zur Prototypenkonfiguration

Unter Berücksichtigung der getroffenen Einschränkungen (auch Entwurfsregeln) für

die Schichtenkombination, reduziert sich die Menge der möglichen Prototypenkonfigurationen drastisch. Gleichzeitig stellt die Beachtung der Einschränkungen sicher, dass nur sinnvolle Konfigurationen gewählt werden können. Aus dem Kombinationsbaum in Abbildung 6.1 lässt sich die Menge der Kombinationsmöglichkeiten ableiten. Die Wurzel (*Base & Base_Scheduler*) stellt die minimale Konfiguration dar. Die Ziffer in jedem Knoten dient der Orientierung und gibt die Anzahl der realisierbaren Kombinationen an, da aus Gründen der Übersichtlichkeit einige Schichten in einem Knoten zusammengefasst werden. Die übrigen Symbole werden nachfolgend erläutert:

- *... & ...* - nur gleichzeitige Nominierung zweier Schichten
- *... + ...* - nachfolgendes Element optional wählbar (jedes Element nur einmal)
- *(... or ...)* - jede nichtleere Untermenge der eingeklammerten Schichten
- *... exor ...* - exklusives ODER

Ausgehend von der Wurzel, werden alle Knoten des Baumes besucht und die einzelnen Schichtenkombinationen zusammengerechnet. Daraus ergibt sich die Gesamtzahl der sinnvollen Kombinationen wie folgt:

$$Kombinationen_{ltd} = (1) + (4) + (1 + 4 + (13 * 4) + 1 + 4) + (13 * 4) + (1 + 4) = 124$$

Die Menge der möglichen und sinnvollen Konfigurationen zeigt, dass der Prototyp in höchstem Maße konfigurierbar und flexibel anpassbar ist. Diese Eigenschaft unterstreicht gleichzeitig den hohen Wiederverwendungswert der entwickelten Programmteile.

Zum besseren Verständnis des Konfigurationsvorgangs zeigt Listing 6.1 eine Beispielkonfiguration, die aus der Menge der gültigen Kombinationen stammt. Sie beinhaltet alle Funktionen (ausser physisches Logging), die während der Prototypenentwicklung implementiert wurden. Ausgehend von einer derartigen Gleichung und unter Berücksichtigung der erläuterten Festlegungen erfolgt die Kompilierung des Prototypen (genauer: Funktionsbibliotheken) durch FeatureC++.

6.2 Speicherbedarf

Aufgrund der guten Konfigurierbarkeit des Prototypen kann das Transaktionsverwaltungssystem hervorragend an das vorhandene Speicherangebot der Zielumgebung angepasst werden. Im Rahmen der Domänenanalyse stellte sich in eingebetteten Systemen gerade dieser Leistungsparameter als entscheidend heraus, da ein System, das nicht über genügend Speicherressourcen verfügt, unmöglich die zu erwartende Datenflut, bedingt durch den Betrieb eines Datenbanksystems, erfassen und ordnungsgemäss organisieren

kann. Aus diesem Grund ist es sinnvoll die ohnehin stark beanspruchten Speicherreserven durch ein schlankes und minimales DBMS zu entlasten. Dieser Forderung kommt der Prototyp durch die erwähnt gute Konfigurierbarkeit nach, da diese ermöglicht, dass nur anwendungsfallspezifische Funktionen bereitgestellt werden.

Listing 6.1: TransRecManager.equation 1

```
1   //Feature
2   Base
3   Base_Scheduler
4   Seriell_Scheduler
5   Base_Sperr_Scheduler
6   2PL_Scheduler
7   S2PL_Scheduler
8   C2PL_Scheduler
9   CS2PL_Scheduler
10  Index_Lock_Scheduler
11  Base_Recovery_Manager
12
13  //Aspect
14  Log_Recovery_Manager
15  Log_Recovery_Manager_Redo_Undo
```

Um die mögliche Platzersparnis zu verdeutlichen, werden zwei völlig unterschiedliche Beispielkonfigurationen mit einander verglichen. Die erste Konfiguration erfolgt wie in Listing 6.1 angegeben. Listing 6.2 zeigt den Inhalt der Equation-Datei des zweiten Setups. Während es sich beim erst genannten Beispiel um eine bezüglich des Funktionsumfangs vollständige Konfiguration mit physischem Log-Buch handelt, umfasst das in Listing 6.2 angegebene Setup nur die Möglichkeit des seriellen Schedulings. Auf Log-Funktionalität wird verzichtet.

Listing 6.2: TransRecManager.equation 2

```
1   //Feature
2   Base
3   Base_Scheduler
4   Seriell_Scheduler
```

Bei den zwei Konfigurationen handelt es sich um gültige Einstellungen, deren resultierende Funktionsbibliothek jeweils ein funktionsfähiges Transaktionsverwaltungssystem bereitstellt. Abbildung 6.2 verdeutlicht die Speicherplatzersparnis des Konfigurationsbeispiels aus Listing 6.2 aufgrund des geringeren Funktionsumfangs. Während die dazugehörige Programmbibliothek nur ca. 57 Kilobyte an Speicherplatz beansprucht, belegt die Bibliothek gemäss Listing 6.1 273 Kilobyte im Speicher. Ist der Funktionsumfang der kleineren Konfiguration für den tatsächlichen Anwendungsfall ausreichend, so ist demnach eine Speicherplatzersparnis von ca. 78,8 Prozent (Differenz: 215,2 Kilobyte) gegenüber einer funktional vollständigen Konfiguration möglich. Dieses Verhältnis verdeutlicht die Vorteile einer flexibel konfigurierbaren Anwendung im Vergleich zu Programmen mit festem Funktionsumfang.

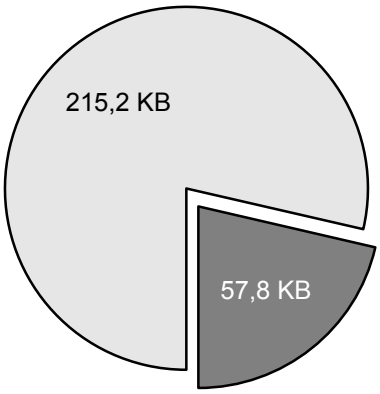

Abbildung 6.2: Speicherumfang: Bibliothek (vollständig) vs. Bibliothek (Seriell)

6.3 Heterogene vs. homogene Crosscuts

Wie dem Schichtenmodell aus Abbildung 4.8 zu entnehmen ist, wurde die Funktionalität des Prototypen hauptsächlich auf der Basis von heterogenen Crosscutting Concerns (durch Merkmale) entwickelt. Lediglich die Belange des Log-Eintrags (und dessen Verfeinerungen) werden in Form von Aspekten (Begründung: homogene Crosscuts lassen sich effektiv durch Aspekte ausdrücken) umgesetzt. Da sich die Funktionalität des Programms aus vielen verschiedenartigen Methoden und Algorithmen zusammensetzt und selten identischer Quell-Code an verschiedenen Stellen des Programms benötigt wird, überrascht es nicht, dass die merkmalsorientierten Entwicklungsprozesse dominieren. Der Prototyp wird mit jeder neuen Schicht um zusätzliche Funktionen ergänzt. Die Erweiterungen gehorchen demnach einer hierarchischen Ordnung (Vererbungshierarchie), wie sie durch die Merkmalsprogrammierung unterstützt wird. Ein weiterer Vorteil dieses Programmieransatzes ist die verbesserte Erweiter- und Wartbarkeit des Programm-Codes aufgrund der vereinfachten Fehlersuche (siehe Abschnitt 2.3).

Obwohl zu Beginn der Entwicklung klar war, dass sich ein Verhältnis zu Gunsten der Merkmalsprogrammierung einstellt, weil diese Art der Programmierung für die Implementierung der meisten identifizierten Belange am geeignetsten erschien, so konnte trotzdem die Nachhaltigkeit des Programmierparadigmas überraschen. In frühen Entwicklungsphasen wurde von der Annahme ausgegangen, dass der komplette Satz an Funktionen bezüglich der Protokollierung durch Aspekte ausgedrückt werden muss. Im Verlauf der Entwicklung stellte sich jedoch heraus, dass einzig der Log-Aufruf als Aspekt entwickelt werden muss, um die Vorteile der aspektorientierten Programmierung zu nutzen. Die dazugehörigen Datenstrukturen, wie *LogList* oder *RecManager* können weiterhin durch Features ausgedrückt werden, da es sich, wie bei den übrigen Merkmalen, um heterogene Crosscutting Concerns handelt.

6.4 Systemvergleich

Die Vorteile der thematisierten Programmfamilienentwicklung und der verwendeten Entwicklungsmethoden und -technologien sollen in diesem Abschnitt im Zusammenhang mit alternativen Lösungsansätzen diskutiert werden. Aufgrund der Vielfalt an verschiedenen Ansätzen, werden nur die beiden in den Abschnitten 3.3.1 und 3.3.2 behandelten Lösungen näher beleuchtet. Da die diskutierten Technologien Repräsentanten verschiedener Systemklassen sind, stehen sie stellvertretend für viele der nicht betrachteten Lösungsansätze. Daher ist eine Beschränkung auf die genannten Beispiele zulässig.

6.4.1 COMET

Um Redundanz zu vermeiden, soll an dieser Stelle auf eine wiederholte Erläuterung des COMET-Ansatzes verzichtet werden (die entsprechenden Passagen sind in Abschnitt 3.3.1 zu finden). Vielmehr soll der Lösungsansatz bezüglich Konfigurierbarkeit und Flexibilität bewertet werden. Wie aus Abschnitt 3.3.1 hervorgeht, setzt sich das COMET-DBMS aus verschiedenen Programmmodulen zusammen, die jeweils abgegrenzte Funktionsklassen beinhalten. Dabei können gemäss Abbildung 3.6 7 verschiedene Komponenten identifiziert werden, die zu einem funktionstüchtigen DBMS kombiniert werden können. Das im Verlauf dieser Arbeit entwickelte Transaktionsverwaltungssystem umfasst (nur) Funktionen der Transaktionsverwaltung. Es bietet allein für die dazugehörigen Prozesse sehr vielfältige Konfigurationsmöglichkeiten und lässt sich deutlich feiner auf den konkreten Anwendungsfall abstimmen, als dies dem COMET-System möglich ist. Ein weiterer Vorteil gegenüber dem COMET-DBMS ist die automatisierte Konfigurationsüberprüfung (Entwurfsregelprüfung). Während für die Konfiguration von COMET (zeit-)aufwendige Setup-Prozesse manuell durchlaufen werden müssen, erfolgt die Generierung des Transaktionsverwaltungssystems unter Angabe von Kompositionsgleichungen, die automatisch auf ihre Gültigkeit geprüft werden.

Das COMET-DBMS wurde auf der Basis aspektorientierter Programmierparadigmen entwickelt. Das Ergebnis zeigt, dass eine derartige Umsetzung möglich ist. Wie aus Abschnitt 6.3 hervorgeht, dominieren (zumindest im Bereich der Transaktionsverwaltung) Attribute, die am effektivsten durch die merkmalsorientierte Programmierung beschrieben werden können. Im COMET-DBMS wurden derartige Attribute durch Aspekte beschrieben, deren Quell-Code letztendlich nur an einer einzigen Stelle eingewebt wird. Diese Tatsache wirft die Frage nach der Zweckmäßigkeit der ausschliesslichen Aspektprogrammierung auf.

6.4.2 PLENTY

Das PLENTY-System stellt grundlegende Funktionalitäten in Form eines Kernsystems bereit. Zusätzliche Funktionen können zur Laufzeit (ähnlich zu Unix-Systemen) in Form von Modulen nachgeladen werden (siehe Abschnitt 3.3.2). Dieser Ansatz bietet ein

Höchstmaß an Flexibilität bei der Konfiguration. Das Laden von weiteren Modulen setzt jedoch voraus, dass die gewünschten Module verfügbar sind. Es ist also notwendig, zusätzliche Modulbibliotheken auf dem Zielsystem anzubieten. In dem Zusammenhang stellt es sich als schwierig dar, die Bibliotheken so zu bestücken, dass möglichst wenig Speicherplatz beansprucht und trotzdem die anwendungsfallspezifische Funktionalität integriert wird. Das Problem liegt nicht in der Machbarkeit, sondern in der Automatisierung derartiger Prozesse (wie sie die entwickelte Programmfamilie in Tateinheit mit den verwendeten Programmier- und Prüftechniken leistet).

Ein anderer Nachteil des PLENTY-Ansatzes ist der erhöhte Verwaltungsaufwand durch das dynamische Laden von zusätzlichen Modulen. Da der Funktionsumfang des hiesigen Transaktionsverwaltungssystems zum Zeitpunkt des Kompilierens festgelegt wird (statisch) und erst durch eine erneute Konfiguration verändert werden kann, sind derartige Verwaltungsprozesse nicht notwendig. Dies ermöglicht einen ressourcenschonenden Betrieb, der gerade in eingebetteten Systemen angestrebt wird.

6.5 Zusammenfassung

Im Mittelpunkt der Betrachtungen von Kapitel 6 steht die Bewertung des entwickelten Prototypen zur Transaktionsverwaltung. Der Aspekt der Konfiguration ist ein zentrales Thema der Bewertung. Durch die aufgestellten Entwurfsregeln, in Kombination mit weiteren Festlegungen, konnte die Menge der möglichen Konfigurationen auf die Anzahl der sinnvollen Setups reduziert werden (konkret: 124 gültige Kombinationen). Da das Transaktionsverwaltungssystem für den Betrieb in eingebetteten Systemen bestimmt ist, wurde der Speicherplatzbedarf im Zusammenhang mit verschiedenen Konfigurationen analysiert. Dabei stellte sich heraus, dass dramatische Speicherplatzeinsparungen möglich sind (erreichbar über den Verzicht auf nicht benötigte Funktionen). Anschliessend erfolgte eine Gegenüberstellung der aspekt- und merkmalsorientierten Programmierung und deren letztendliche Ausprägungen im Prototypen. Dabei konnte festgestellt werden, dass sich für die meisten Funktionen eine Umsetzung auf der Basis der merkmalsorientierten Programmierung anbietet. Zum Abschluss wurde das entwickelte System mit konkurrierenden Ansätzen verglichen. Die wichtigsten Vorteile der eigenen Lösung liegen in der feingranularen Konfigurierbarkeit und automatisierten Gültigkeitsüberprüfung des Setups.

Kapitel 7

Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Modellierung eines flexibel konfigurierbaren Transaktionsverwaltungssystems. Um das Verständnis für grundlegende Methoden und Technologien zu vermitteln, wurden diese im Grundlagenkapitel erläutert. Hauptthemen des Kapitels waren die Domänenentwicklung und Programmiertechniken zur Unterstützung einer konsequenten modularen Software-Entwicklung. Im Kapitel der Domänenanalyse wurden zunächst Informationen zu eingebetteten Systemen gesammelt. Als einer der bestimmenden Leistungsfaktoren derartiger Systeme kristallisierte sich das Speicherangebot heraus. Daher lag die Herausforderung darin, ein möglichst feingranulares Transaktionsverwaltungssystem zu entwickeln, um die Speicherressourcen zu schonen. Als Vorbereitung für die prototypische Entwicklung wurden grundlegende Eigenschaften von DBMS im Allgemeinen und Prozessen der Transaktionsverwaltung im Speziellen betrachtet. Weiterhin wurden existierende Lösungsansätze genauer betrachtet (COMET und PLENTY ausführlich), um einen Überblick über alternative Konzepte zu bekommen. Zum Abschluss des Kapitels der Domänenanalyse wurde eine Merkmalsanalyse durchgeführt, in deren Verlauf alle relevanten Features der Transaktionsverwaltung identifiziert wurden. In der Entwurfsphase fand die Modellierung der Software-Architektur für die Programmfamilie statt. Wichtigste Ergebnisse des Entwurfs sind das Schichtenmodell und die Bestimmung der Entwurfsregeln. An die Phase des Entwurfs schliesst sich die Implementierung an. Hier fliessen die Ergebnisse der vorhergehenden Entwicklungsabschnitte in ein konkretes Software-System ein. Im Evaluierungs-Kapitel findet eine Bewertung der Entwicklungsergebnisse bezüglich Konfigurierbarkeit, Speicherbedarf, Merkmalen und Aspekten statt. Je nach Konfiguration, sind signifikante Speicherplatzeinsparungen möglich. Aufgrund dieser Eigenschaft (weitere Vorteile siehe 6) kann der entwickelten Programmfamilie die Eignung für den Einsatz in eingebetteten Systemen bescheinigt werden. Die Implementierung des Prototypen basiert hauptsächlich auf der Verwendung von heterogenen Crosscuts. Durch die Kombination von Merkmalen und Aspekten ergeben sich für den Entwickler unzählige Möglichkeiten für die effiziente und komfortable Programmentwicklung. Aus diesem Grund kann den verwendeten Programmiertechniken (vereint in FeatureC++) für die Zukunft eine günstige Prognose ausgesprochen werden. Der genutzte Entwicklungsansatz erfüllt (eindrucksvoll) die

formulierten Erwartungen.

Zukünftig kann die Programmfamilie, neben den implementierten Funktionen (gemäss der reduzierten Merkmalsmodelle) durch zusätzliche Funktionen erweitert werden. Die verwendeten Technologien (in Gestalt von FeatureC++ und der Entwurfsregelprüfung aus der Ahead-Tool-Suite) stellen eine gute Basis für die Erweiterungen dar. Aufbauend auf den implementierten Funktionen, können die existierenden Klassenhierarchien in neuen Schichten erweitert (Stichwort: Verfeinerung) werden (das Hinzufügen neuer Klassen ist selbstverständlich auch möglich). Mit der Einführung neuer Schichten erfolgt eine Erweiterung des Entwurfsregelkatalogs. Einige Funktionen wurden aus Gründen der Vereinfachung abstrahiert. Diese können auf Basis der genutzten Technologien ebenfalls weiter konkretisiert und auf Praxistauglichkeit getrimmt werden. Wie im Entwurfskapitel (4) erwähnt, implementieren manche Schichten des Prototypen mehrere Merkmale. Werden die einzelnen Merkmale in zusätzlichen Schichten derart organisiert, dass jede Schicht nur noch ein einziges Merkmal implementiert, so kann die Granularität des Transaktionsverwaltungssystems und damit die Konfigurierbarkeit weiter verbessert werden.

Anhang A

A.1 Entwurfsregeln des Prototypen

Listing A.1: Seriell_Scheduler

```
1  single layer Seriell_Scheduler;
2
3  //define/use attributes
4  extern flowright Bool Basis_ScheduleR;
5  flowright Bool Seriell;
6
7  //provides and requires
8  provides flowright Seriell;
9  requires flowright Basis_ScheduleR;
```

Listing A.2: Base_Sperr_Scheduler

```
1  single layer Base_Sperr_Scheduler;
2
3  //define/use attributes
4  extern flowright Bool Basis_ScheduleR;
5  flowright Bool Basis_Sperr_ScheduleR;
6
7  //provides and requires
8  provides flowright Basis_Sperr_ScheduleR;
9  requires flowright Basis_ScheduleR;
```

Listing A.3: 2PL_Scheduler

```
1  single layer 2PL_Scheduler;
2
3  //define/use attributes
4  extern flowright Bool Basis_Sperr_ScheduleR;
5  flowright Bool B2Phase;
6
7  //provides and requires
8  provides flowright B2Phase;
9  requires flowright Basis_Sperr_ScheduleR;
```

Listing A.4: S2PL_Scheduler

```
1   single layer S2PL_Scheduler;
2
3   //define/use attributes
4   extern flowright Bool Basis_Sperr_ScheduleR;
5   flowright Bool S2Phase;
6
7   //provides and requires
8   provides flowright S2Phase;
9   requires flowright Basis_Sperr_ScheduleR;
```

Listing A.5: C2PL_Scheduler

```
1   single layer C2PL_Scheduler;
2
3   //define/use attributes
4   extern flowright Bool Basis_Sperr_ScheduleR;
5   flowright Bool C2Phase;
6
7   //provides and requires
8   provides flowright C2Phase;
9   requires flowright Basis_Sperr_ScheduleR;
```

Listing A.6: CS2PL_Scheduler

```
1   single layer CS2PL_Scheduler;
2
3   //define/use attributes
4   extern flowright Bool Basis_Sperr_ScheduleR;
5   flowright Bool CS2Phase;
6
7   //provides and requires
8   provides flowright CS2Phase;
9   requires flowright Basis_Sperr_ScheduleR;
```

Listing A.7: Index_Lock_Scheduler

```
1   single layer Index_Lock_Scheduler;
2
3   //define/use attributes
4   extern flowright Bool CS2Phase;
5   extern flowright Bool B2Phase;
6   extern flowright Bool C2Phase;
7   extern flowright Bool S2Phase;
8
9   //provides and requires
10  requires flowright B2Phase and CS2Phase
11                     and C2Phase and S2Phase;
```

Listing A.8: Base_Recovery_Manager

```
1  single layer Base_Recovery_Manager;
2
3  //define/use attributes
4  flowright Bool Basis_Recovery;
5  extern flowright Bool Basis_ScheduleR;
6
7  //provides and requires
8  provides flowright Basis_Recovery;
9  requires flowright Basis_ScheduleR;
```

Listing A.9: Log_Recovery_Manager

```
1  single layer Log_Recovery_Manager;
2
3  //define/use attributes
4  extern flowright Bool Basis_Recovery;
5  flowright Bool Basis_Log;
6
7  //provides and requires
8  provides flowright Basis_Log;
9  requires flowright Basis_Recovery;
```

Listing A.10: Log_Recovery_Manager_Redo_Undo

```
1  single layer Log_Recovery_Manager_redo_undo;
2
3  //define/use attributes
4  extern flowright Bool Basis_Log;
5  flowright Bool Logical_LogR;
6
7  //provides and requires
8  provides flowright Logical_LogR;
9  requires flowright Basis_Log;
```

Listing A.11: Log_Recovery_Manager_Before_After

```
1  single layer Log_Recovery_Manager_before_after;
2
3  //define/use attributes
4  extern flowright Bool Basis_Log;
5  flowright Bool Physical_Log;
6
7  //provides and requires
8  provides flowright Physical_Log;
9  requires flowright Basis_Log;
```

A.2 Ausgewählte Quell-Code-Abschnitte des Prototypen

Listing A.12: Transaktion.h

```
1  #include <string>
2  #include <iostream>
3  using namespace std;
4
5  class Transaktion
6  {
7          TransOpStack* operation;
8
9          //fuer gezieltes Lock-Management im B+-Baum
10         TransOpStack* archiv;
11         int TransId;
12         int Status;
13
14 public:
15
16         //Trans mit übergebener Operationsliste mit ID
17         Transaktion(TransOpStack*, int);
18
19         //leere Transaktion mit ID
20         Transaktion(int);
21
22         //ID abfragen
23         int getTransID();
24
25         //OperationsStack abfragen
26         TransOpStack* getTransOpStack();
27
28         TransOpStack* getArchivStack();
29
30         //Operation von Stelle int abfragen
31         TransOp* getTransOp(int);
32
33         //Status setzen
34         void setStatus(int);
35
36         //Status ermitteln
37         int getStatus();
38
39         //Transaktion anzeigen
40         void showTrans();
41
42         //Naechste Operation nominieren
43         void pushTransOp(TransOp*);
44
45         //Serielle Ausführung
46         void execution();
47
48 };
49
50 Transaktion::Transaktion(TransOpStack* n, int id)
51 {
52         archiv = new TransOpStack(n);
53     operation = new TransOpStack(n);
54         TransId = id;
55         Status = 1;
56 }
57
58 Transaktion::Transaktion(int id)
```

```
59 {
60     operation = new TransOpStack(1000);
61     archiv    = new TransOpStack(1000);
62     TransId = id;
63     Status = 1;
64 }
65
66 int Transaktion::getTransID()
67 {
68         return TransId;
69 }
70
71 TransOpStack* Transaktion::getTransOpStack()
72 {
73         return operation;
74 }
75
76 TransOpStack* Transaktion::getArchivStack()
77 {
78         return archiv;
79 }
80
81 TransOp* Transaktion::getTransOp(int stelle)
82 {
83         if((*operation).getTransOp(stelle) != NULL)
84           return (*operation).getTransOp(stelle);
85         else return NULL;
86 }
87
88 void Transaktion::setStatus(int state)
89 {
90         Status = state;
91 }
92
93 int Transaktion::getStatus()
94 {
95         return Status;
96 }
97
98 void Transaktion::showTrans()
99 {
100         cout<<"ID "<<getTransID()<<endl;
101         cout<<"Status "<<getStatus()<<endl;
102         cout<<"Operationen "<<endl;
103         (*operation).showStack();
104         cout<<endl;
105 }
106
107 void Transaktion::pushTransOp(TransOp* tr)
108 {
109         (*operation).push(tr);
110         (*archiv).push(tr);
111 }
112
113 void Transaktion::execution()
114 {
115         cout<<"Transaktion: "<<getTransID()<<" wird ausgefuehrt"<<endl;
116 }
```

Listing A.13: Scheduler.h

```
1 #include <string>
2 #include <iostream>
3 #include <vector>
4 using namespace std;
```

```
5
6  class Scheduler
7  {
8          TransOpStack* input;
9
10 public:
11
12         //Scheduler mit Operations-Stack
13         Scheduler(TransOpStack*);
14
15         //Scheduler mit leerem Operations-Stack
16         Scheduler();
17
18         //Operations-Stack abfragen
19         TransOpStack* getInput();
20
21         //Operations-Stack eingliedern
22         void setInput(TransOpStack*);
23
24         //schrittweiser Aufbau des Stacks
25         void addInput(TransOp*);
26
27         //Operationen zu initialisierten Transaktionen hinzufuegen
28         vector<Transaktion>* parseOps();
29
30 };
31
32 Scheduler::Scheduler(TransOpStack* in)
33 {
34         input = in;
35 }
36
37 Scheduler::Scheduler(){}
38
39 TransOpStack* Scheduler::getInput()
40 {
41     return input;
42 }
43
44 void Scheduler::addInput(TransOp* op)
45 {
46         (*input).push(op);
47 }
48
49 //parse die jeweiligen Operationen in die entsprechenden Transaktionen
50 vector<Transaktion>* Scheduler::parseOps()
51 {
52         bool flag = true;
53         int tempId;
54         TransOpStack* temp = new TransOpStack(input);
55         int elmenge = (*temp).getTip();
56         vector<Transaktion>* myTrans = new vector<Transaktion>;
57         while(flag == true)
58         {
59          if(elmenge > 0)
60          {
61                 if((*temp).getTransOp(0) != NULL)
62                 {
63                         tempId = (*(*temp).getTransOp(0)).getTransId();
64                         TransOpStack* neuSt = new TransOpStack(1000);
65                         for(int i = 0; i < elmenge; i++)
66                         {
67                                 if((*temp).getTransOp(i) != NULL
68                                   && (*(*temp).getTransOp(i)).getTransId() == tempId)
69                                 {
```

```
70                                          bool wahr =(*neuSt).push((*temp).delTransOp(i));
71                                          i = -1;
72                                          elmenge = (*temp).getTip();
73                                  }
74                          }
75                          Transaktion* neuTr = new Transaktion(neuSt, tempId);
76                          (*myTrans).push_back(*neuTr);
77                  }
78          }
79          else flag = false;
80         }
81         return myTrans;
82 }
83
84 void Scheduler::setInput(TransOpStack* cop)
85 {
86         input = cop;
87 }
```

Listing A.14: Refinement Transaktion.h

```
1 #include <string>
2 #include <iostream>
3 #include <vector>
4 using namespace std;
5
6 refines class Transaktion
7 {
8         vector<ListEDat>* lockvec;
9         bool unlockflag;
10
11 public:
12
13         //Trans mit übergebener Operationsliste mit ID
14         Transaktion(TransOpStack*, int);
15
16         //leere Transaktion mit ID
17         Transaktion(int);
18
19         //Vektor mit LockInfo abfragen
20         vector<ListEDat>* getLockVec();
21
22         //LockInfo speichern
23         void pushLockInfo(ListEDat*);
24
25         //LockInfo aus Vektor entfernen
26         ListEDat* popLockInfo();
27
28         //Kontrollstruktur
29         bool getULockFlag();
30 };
31
32 Transaktion::Transaktion(TransOpStack* op, int id):super(op, id)
33 {
34         lockvec = new vector<ListEDat>;
35         unlockflag = false;
36 }
37
38 Transaktion::Transaktion(int id):super(id)
39 {
40         lockvec = new vector<ListEDat>;
41         unlockflag = false;
42 }
43
44 vector<ListEDat>* Transaktion::getLockVec()
```

```
45 {
46         return lockvec;
47 }
48
49 void Transaktion::pushLockInfo(ListEDat* lockDat)
50 {
51         (*lockvec).push_back(*lockDat);
52         unlockflag = false;
53 }
54
55 ListEDat* Transaktion::popLockInfo()
56 {
57         ListEDat* locke = new ListEDat((*lockvec).at(0));
58         (*lockvec).erase(0);
59         unlockflag = true;
60
61         return locke;
62 }
63
64 bool Transaktion::getULockFlag()
65 {
66         return unlockflag;
67 }
```

Listing A.15: LogList.h

```
1  #include <string>
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  class LogList
7  {
8          //Listenspalten
9          vector<int>* LSN;
10         vector<int>* TID;
11         vector<int>* PrevLSN;
12         vector<string>* BOT;
13         vector<string>* EOT;
14
15         //Kontrollstruktur
16         int iter;
17
18 public:
19
20         //Konstruktor
21         LogList();
22
23         //Log-Eintrag
24         void newEntry(TransOp*, TransOpStack*);
25
26         //Log-Eintrag anschauen
27         void showEntry(int);
28
29         //Abruf von iter
30         int getIter();
31
32         //Manipulation von iter
33         void iterate();
34
35 };
36
37 LogList::LogList()
38 {
39         iter = 1;
```

```
40          LSN = new vector<int>;
41          TID = new vector<int>;
42          PrevLSN = new vector<int>;
43          BOT = new vector<string>;
44          EOT = new vector<string>;
45  }
46
47  void LogList::newEntry(TransOp* eintrag, TransOpStack* stack)
48  {
49          //LSN einfuegen
50          (*LSN).push_back(iter);
51
52          //TID einfuegen
53          (*TID).push_back((*eintrag).getTransId());
54
55          //PrevLSN einfuegen
56          (*PrevLSN).push_back(0);
57          for(int i = (*TID).size() - 2; i >= 0; i--)
58          {
59                  if((*TID).at(i) == (*TID).back())
60                  {
61                          (*PrevLSN).pop_back();
62                          (*PrevLSN).push_back((*LSN).at(i));
63                          break;
64                  }
65          }
66
67          //BOT einfuegen
68          if((*PrevLSN).back() == 0) (*BOT).push_back("BOT");
69          else (*BOT).push_back("");
70
71          //EOT einfuegen
72          if((*stack).empty()) (*EOT).push_back("EOT");
73          else (*EOT).push_back("");
74
75          cout<<"Log-Eintrag fuer aktuelle Operation vorgenommen!"<<endl;
76
77  }
78
79  void LogList::showEntry(int pos)
80  {
81          cout<<"LSN: "<<(*LSN).at(pos)<<endl;
82          cout<<"TID: "<<(*TID).at(pos)<<endl;
83          cout<<"PrevLSN: "<<(*PrevLSN).at(pos)<<endl;
84          cout<<"BOT: "<<(*BOT).at(pos)<<endl;
85          cout<<"EOT: "<<(*EOT).at(pos)<<endl;
86          cout<<endl;
87  }
88
89  int LogList::getIter()
90  {
91          return iter;
92  }
93
94  void LogList::iterate()
95  {
96          iter++;
97  }
```

Listing A.16: Refinement LogList.h

```
1  #include <string>
2  #include <iostream>
3  #include <vector>
4  using namespace std;
```

```
5
6  refines class LogList
7  {
8          vector<string>* REDO;
9          vector<string>* UNDO;
10
11
12 public:
13
14         //Konstruktor-Refinement
15         LogList();
16
17         //Redo-Info
18         void setRedo(string);
19
20         //Undo-Info
21         void setUndo(string);
22
23         //Ausgabe des Log-Eintrages
24         void showEntry(int);
25
26 };
27
28 LogList::LogList():super()
29 {
30         REDO = new vector<string>;
31         UNDO = new vector<string>;
32 }
33
34 void LogList::setRedo(string re)
35 {
36         (*REDO).push_back(re);
37 }
38
39 void LogList::setUndo(string un)
40 {
41         (*UNDO).push_back(un);
42 }
43
44 void LogList::showEntry(int pos)
45 {
46         super::showEntry(pos);
47         if((*REDO).size() != 0)cout<<"REDO: "<<(*REDO).at(pos)<<endl;
48         if((*UNDO).size() != 0)cout<<"UNDO: "<<(*UNDO).at(pos)<<endl;
49 }
```

Literaturverzeichnis

[Ape05] Apel, S.; Leich, T.; Rosenmüller, M.; Saake, G.: FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++. Technischer Bericht, Fakultät für Informatik, Universität Magdeburg, 2005. Online: `http://wwwiti.cs.uni-magdeburg.de/iti_db/forschung/fop/featurec/publications/FC++TechReport.pdf`, Abruf am 2. Januar 2006.

[Ara91] Arango, G.; Prieto-Diaz, R.: *Domain Analysis and Software Systems Modeling*. IEEECSP, 1991.

[Ara94] Arango, G.: Domain Analysis Methods. In *Software Reusability*, S. 17–49. Ellis Horwood, London, UK, 1994.

[Bä05] Bärecke, R.: Modellierung eines konfigurierbaren Speichermanagers für eingebettete Datenbankmanagementsysteme. Diplomarbeit, Fakultät für Informatik, Universität Magdeburg, 2005.

[Bas01] Bass, L.; Klein, M.; Bachmann, F.: Quality Attribute Design Primitives and the Attribute Driven Design Method. 2001.

[Bas03] Bass, L.; Clements, P.; Kazman, R.: *Software Architecture in Practice*. Addison-Wesley, Boston, 2. Auflage, 2003.

[Bat94] Batory, D.; Singhal, V.; Thomas, J.; Dasari, S.; Geraci, B.; Sirkin, M.: The GenVoca Model of Software-System Generators. *IEEE Software*, S. 89–94, 1994.

[Bat97] Batory, D.; Geraci, B.: Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering (special issue on Software Reuse)*, S. 62–87, 1997.

[Bat00] Batory, D.; Smaragdakis, Y.: Mixin-Based Programming in C++. *Second International Symposium on Generative and Component-Based Software Engineering (GCSE'2000)*, 2000.

[Bat02] Batory, D.; Lopez-Herrejon, R. E.; Martin, J.-P.: Generating Product-Lines of Product-Families. In *Proc. 17th IEEE Int'l Conf. on Automated Software Engineering*, S. 81–92, 2002.

[Bat03] Batory, D.; Sarvela, J. N.; Rauschmayer, A.: Scaling step-wise refinement. In *Proceedings of the 25th international conference on Software engineering*, S. 187–197. IEEE Computer Society, 2003.

[Bat05] Batory, D.: A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. *Summer school on Generative and Transformation Techniques in Software Engineering*, 2005.

[Ber01] Bergmans, L.; Akşit, M.: Guidelines for Identifying Obstacles when Composing Distributed Systems from Components. In Akşit, M. (Hrsg.): *Software Architectures and Component Technology: The State of the Art in Research and Practice*, S. 29–56. Kluwer Academic Publishers, 2001.

[Boe81] Boehm, B. W.: *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Bos00] Bosch, J.: *Design and Use of Software Architectures*. Addison-Wesley, 1. Auflage, 2000.

[Bus96] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M.: *Pattern-Oriented Software Architecture, A System of Patterns*. 1996. STARS Informal Technical Report STARS-VC-B017/001/00. Unisys Corporation.

[Car90] Carey, M. J.; Haas, L.: Extensible database management systems. *j-SIGMOD*, Band 19, Nr. 4, S. 54–60, 1990.

[Cle01] Clements, P.; Northrop, L. M.: *Software Product Lines: Practice and Patterns*. Addison-Wesley, 2001.

[Cod90] Codd, E. F.: *The Relational Model for Database Management: Version 2*. Addison-Wesley Publishing Company, Inc., 1990.

[Cza00] Czarnecki, K.; Eisenecker, U. W.: *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, 2000.

[Cza01] Czarnecki, K.; Dominick, L; Eisenecker, U. W.: Aspektorientierte Programmierung in C++. *iX Ausgaben 8-10 (dreiteilige Artikelserie)*, 2001.

[Dit01] Dittrich, K.; Geppert, A.: *Component Database Systems*. Morgan Kaufmann, 2001.

[Dum03] Dumke, R.: *Software Engineering*. Friedrich Vieweg & Sohn Verlag /GWV Fachverlage GmbH, Wiesbaden, 4. Auflage, 2003.

[Erg05a] Ergänzend zusammengetragen: Embedded System, 2005. Online: `http://de.wikipedia.org/wiki/Embedded_System`, Abruf am 4. Oktober 2005.

[Erg05b] Ergänzend zusammengetragen: Objektorientierte Programmierung, 2005. Online: `http://www.de.wikipidia.org/wiki/Objektorientierte_Programmierung`, Abruf am 14. August 2005.

[Fer99] Ferré, X.; Vegas, S.: An Evaluation of Domain Analysis Methods. 4th CAiSE/IFIP8.1 International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design (EMMSAD'99). 1999.

[Fin98] Findler, R. B.; Flatt, M.: Modular Object-Oriented Programming with Units and Mixins. In *Proceedings of ICFP*, 1998.

[Fra04] Franczyk, B.; Kiebusch, S.; Werner, A.: Domänenanalyse (Stakeholder) und Qualitätmetriken für Softwareproduktlinien. 2004. Online: `http://www.pesoa.org/pages/Publications/Fachberichte022004/PESOA_TR_2-2004.pdf`, Abruf am 22. August 2005.

[Gri98] Griffel, F.: *Componentware: Konzepte und Techniken eines Softwareparadigmas*. dpunkt.verlag, 1998.

[Hä87] Härder, T.: Realisierung von operationalen Schnittstellen. In Peter C. Lockemann and Joachim W. Schmidt (Hrsg.): *Datenbankhandbuch*, S. 163–335. Springer, 1987.

[Has95] Hasse, C.: *Inter- und Intratransaktionsparallelität in Datenbanksystemen: Entwurf, Implementierung und Evaluation eines Datenbanksystems mit Inter- und Intratransaktionsparallelität*. Dissertation, Departement Informatik, ETH Zürich, 1995.

[Hub95] Huber-Wäschle, F.; Schauer, H.; Widmayer, P. (Hrsg.): *GISI 95 - Herausforderungen eines globalen Informationsverbundes für die Informatik*. Springer-Verlag, Zürich, 1995.

[Jac94] Jacobson, I.; Christerson, M.; Jonsson, P.; Övergaard, G.: *Object-Oriented Software Engineering*. Addison-Wesley, 1994.

[Kan90] Kang, K.; Cohen,S.; Hess, J.; Novak, W.; Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Nr. CMU/SEI-90-TR-21, 1990.

[Kar95] Karlsson, E.-A. (Hrsg.): *Software Reuse: A Holistic Approach*. WILEY, 1995.

[Kea97] Kean, L.: Organization Domain Modeling. 1997. Online: `http://www.sei.cmu.edu/str/descriptions/odm_body.html`, Abruf am 8. September 2005.

[Kic97] Kiczales, G.; Lamping, J.; Menhdhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwin, J.: Aspect-Oriented Programming. In Akşit, M.; Matsuoka, S. (Hrsg.): *Proceedings European Conference on Object-Oriented Programming*, S. 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[Kru93] Krut, R. W.: Integrating 001 Tool Support into the Feature-Oriented Domain Analysis Methodology. Nr. CMU/SEI-93-TR-011, 1993.

[Laf95] Laforme, D.; Stropky, M. E.: An Automated Mechanism for Effectively Applying Domain Enginnering in Reuse Activities. 1995. Online: `http://archive.adaic.com/docs/reports/stropky/domain/paper.doc`, Abruf am 2. Januar 2006.

[Mey90] Meyer, B.: *Objektorientierte Softwareentwicklung*. Carl Hanser Verlag München Wien, Prentice-Hall International Inc., 1990.

[Nie95] Nierstrasz, O.; Tsichritzis, D. (Hrsg.): *Object-Oriented Software Composition*. Prentice-Hall, 1995.

[Nil94] Nilson, R.; Kogut, P.; Jackelen, G.: Component Provider's and Tool Developer's Handbook Central Archive for Reusable Software (CARDS). 1994.

[Nys02] Nyström, D.; Tešanović, A.; Hansson, J.; Norström, C.: Embedded Databases for Embedded Real-Time Systems: A Component-Based Approach. Technischer Bericht, Linköping University, Mälardalen University, 2002.

[Nys03] Nyström, D.; Tešanović, A.; Norström, C.; Hansson, J.: The COMET Database Management System. Technischer Bericht, Mälardalen Real-Time Research Centre, Mälardalen University, 2003.

[Nys04] Nyström, D.; Tešanović, A.; Norström, C.; Hansson, J.: COMET: A Component-Based Real-Time Database for Automotive Systems. In *Proceedings of the Workshop on Software Engineering for Automotive Systems at 26th International Conference on Software engineering (ICSE'04)*. IEEE Computer Society Press, Edinburgh, Scotland, 2004.

[Par79] Parnas, D. L.: Designing Software For Ease Of Extension And Contraction. *IEEE Transaction On SSoftware Engineering*, 1979.

[Pla01] Platzner, M.: Hardware/Software Codesign (Vorlesungsskript), 2001.

[Pre00] Prepared by ATIS Committee T1A1 Performance and Signal Processing: ATIS Telecom Glossary 2000 (T1.523-2001), 2000. Online: `http://www.atis.org/tg2k/_database_management_system.html`, Abruf am 2. Januar 2005.

[Ros05] Rosenmüller, M.: Merkmalsorientierte Programmierung in C++. Diplomarbeit, Fakultät für Informatik, Universität Magdeburg, 2005.

[Saa99] Saake, G.; Heuer, A.: *Datenbanken: Implementierungstechniken*. MITP-Verlag, Bonn, 1999.

[Sha96] Shaw, M.; Garlan, D.: *Software Architecture: Perspectives on a Emerging Discipline*. Prentice-Hall, 1996.

[Sin96] Singhal, V.: *A Programming Language for Writing Domain-Specific Software System Generators*. Dissertation, University of Texas at Austin, 1996.

[Sof] Software Engineering Institute: Attribute-Driven Design Method. Online: `http://www.sei.cmu.edu/architecture/add_method.html`, Abruf am 26. August 2005.

[Sof96] Software Engineering Institute: What is Model Based Software Engineering (MBSE)? 1996. Online: `http://www.sei.cmu.edu/mbse/is.html`, Abruf am 2. Januar 2006.

[Spi02] Spinczyk, O.: Aspektorientierung und Programmfamilien im Betriebssystembau. 2002.

[STA96] STARS: *Organization Domain Modeling (ODM) Guidebook, Version 2.0*, 1996.

[Szy98] Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Thi03] Thiele, L.; Teich, J.: Eingebettete Systeme (Vorlesungsskript), 2003.

Selbständigkeitserklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 2. Januar 2006

Mario Pukall