Objektorientierte Softwareentwicklung datenintensiver Mehrbenutzeranwendungen für relationale Datenbanksysteme[1]

Joachim Reinert, Hans-Peter Steiert
Universität Kaiserslautern, FB Informatik
67663 Kaiserslautern
e-mail: {jreinert, steiert}@informatik.uni-kl.de

Kurzfassung

Von der Objektorientierung als Paradigma der Softwareentwicklung verspricht man sich vielfach eine höhere Produktivität der Entwickler/innen und eine verbesserte Wiederverwendung von Software-Komponenten. Über die Nutzung objektorientierter Programmiersprachen hinaus wird dabei auch eine objektorientierte Entwurfsmethodik propagiert. Ausgangspunkt dieses Papiers ist ein konkretes Industrieprojekt, in dem mittels objektorientierter Technologie datenintensive Mehrbenutzeranwendungen entwickelt werden. Innerhalb dieses Projektes wurde deutlich, daß, unabhängig von der Datenbanktechnologie, die objektorientierte Entwurfsmethodik wichtige Faktoren eines leistungsfähigen Gesamtsystems in dieser Domäne nicht ausreichend berücksichtigt. In der vorliegenden Arbeit schlagen wir daher für die systematische Erfassung dieser Faktoren eine Verfeinerung der Entwurfsmethodik vor.

Die Sicherstellung der Persistenz von Daten erfordert die Integration von Programmiersprachen und Datenbanktechnologie. Für eine leistungsfähige Integration müssen die beteiligten Teilsysteme optimal aufeinander abgestimmt werden. Zur systematischen Realisierung entwickeln wir in dieser Arbeit ein Architekturmodell, das es erlaubt, Applikationsentwicklung und Entwicklung des Datenbankschemas weitgehend eigenständig zu behandeln und zu optimieren, und so die geforderte Gesamtleistung zu erreichen. Dies ist nur möglich, wenn im Entwicklungsprozeß alle verfügbaren Informationen genutzt werden können. Deren Vielfalt und gegenseitige Abhängigkeit erfordert jedoch eine geeignete Werkzeugunterstützung, die heute vielfach noch nicht gegeben ist. Ein solches Werkzeug haben wir aus der konkreten Projekterfahrung unter Berücksichtigung der verfeinerten Entwurfsmethodik und der von uns verfolgten Architektur konzipiert. Wir berichten in dieser Arbeit über die prinzipiellen Anforderungen und Leistungen dieses Werkzeuges.

Stichworte: Objektorientierte Analyse und Design, objektorientierte Softwareentwicklung, Abbildung auf relationale Datenbanksysteme

1. Einleitung

Objektorientierte Softwareentwicklung scheint eine geeignete Technologie zu sein, um die steigende Komplexität moderner Softwaresysteme in den Griff zu bekommen. Man erwartet, daß die entstehenden Systeme schneller entwickelt werden können, weil die Methodik einen höheren Grad an Wiederverwendung erlaubt. Außerdem erhofft man sich leichter wartbare Systeme, da die Komponenten stärker gekapselt sind. In der Literatur werden verschiedene Vorgehensweisen vorgeschlagen, die bekanntesten sind von Booch [Bo94], Rumbaugh [Ru91] und Coad/Yourdon [CY91a, CY91b]. Es bestehen Bestrebungen, die ersten beiden zu einer umfassenderen Methodik zu vereinen [BR95]. Bisher fehlen aber noch Erfahrungen beim Einsatz dieser Entwicklungstechniken für Anwendungssysteme, die eine (logisch) zentralisierte Haltung großer Datenmengen erfordern [GLL96].

Im Rahmen einer Kooperation unterstützen wir die „Markant SW Software und Dienstleistung GmbH". Sie entwickelt für ihr Mutterhaus, ein Handelsunternehmen, unter dem Projektnamen „WWS2000" ein neues Warenwirtschaftssystem. Man hat sich in diesem Projekt ebenfalls für objektorientierte Entwicklungsmethoden entschieden, da man hofft, so zu einem leicht zu wartenden und flexibel anpaßbaren System zu kommen. Der Entwicklungsprozeß orientiert sich an dem Vorschlag „Objektorientierte Analyse und Design" von Booch. Die Modellierung wird mit Hilfe eines kommerziellen Werkzeuges durchgeführt, die Realisierung erfolgt in der objektorientierten Programmiersprache (OOPL) Smalltalk. Wie in jedem praxisrelevanten Projekt wird für die zentralisierte Datenhaltung ein Datenbanksystem eingesetzt. Auf den ersten Blick erscheint der Einsatz eines objektorientierten Datenbanksystems (OODBMS) als passende Lösung. Dennoch hat man sich für ein relationales Datenbanksystem (RDMBS) entschieden. Die Gründe sind sowohl betriebwirtschaftlicher, als auch technischer Natur:

- Der Umstieg auf objektorientierte Datenbanktechnologie führt zu großen Investitionen, da die vorhandene relationale Lösung komplett ersetzt werden muß. Während im Umgang mit RDBMS bereits ausreichend Erfahrung bei den Mitarbeitern vorhanden ist, würde der Umstieg zusätzliche Schulungen und damit weitere Kosten erfordern.
- Die Kontinuität in der Entwicklung relationaler Systeme haben OODBMS noch nicht erreicht. Sowohl die verwendete Technologie als auch die Qualifikation der Mitarbeiter drohen schnell zu veralten. Die Sicherheit der Investitionen ist daher nicht gewährleistet.
- Während die Standardisierung der Anfragsprache SQL für relationale Datenbanksysteme bereits weit fortgeschritten ist, hat sich bei objektorientierten Systemen noch kein einheitlicher Standard durchgesetzt. Mit dem Einsatz eines OODBMS bindet man sich daher in den meisten Fällen an einen Hersteller. Der Einsatz eines RDMBS ermöglicht dagegen eine weitgehende Unabhängigkeit von der verwendeten Datenbankplattform.
- Im Anforderungskatalog der Anwender wird die Offenheit des Datenbanksystem für den Einsatz von Fremdwerkzeugen, beispielsweise Report-Generatoren, verlangt. Für SQL-fähige Datenbanksysteme sind diese Werkzeuge in großer Zahl vorhanden, OODBMS werden diesbezüglich noch schlecht unterstützt.

Diese Konstellation, objektorientierte Techniken in Entwurf und Implementierung einzusetzen und gleichzeitig auf die bewährte relationale Datenbanktechnologie zurückzugreifen, dürfte typisch für viele aktuelle Projekte sein. In dieser Arbeit wollen wir unsere Erfahrungen im beschriebenen Umfeld darstellen.

Der erste Teil beschäftigt sich mit der gewählten Entwicklungsmethodik und ihrer Eignung zum Erstellen datenintensiver Anwendungen. Nach unserer Meinung wird im Vorschlag von Booch der Aspekt der persistenten Datenhaltung vernachlässigt. Die Modellierung liefert nur den Ausgangspunkt für die Implementierung des fachlichen Kerns der Anwendungen, die Frage nach einer leistungsfähigen Anbindung einer Datenbank bleibt offen. Dieses Problem tritt besonders dann auf, wenn das Datenbanksystem ein anderes Datenmodell bereitstellt als die verwendete Programmiersprache. Im Rahmen unserer Kooperation haben wir daher eine Verfeinerung der Entwurfsmethodik vorgeschlagen, um bereits frühzeitig die Anforderungen an die persistente Datenhaltung zu ermitteln und die gegenseitigen Abhängigkeiten zwischen dem Entwurf der Applikationslogik und dem Datenzugriff zu analysieren. Damit soll verhindert werden, daß der objektorientierte Entwurf in der Implementierung zu einer prozeduralen Programmierung degeneriert. Dies wird Inhalt von Abschnitt 2 sein.

Anschließend (Abschnitt 3) gehen wir auf die angestrebte Realisierung ein. Dazu führen wir

1. Die Arbeiten zu diesem Beitrag wurden im Rahmen eines Kooperationsprojektes und mit finanzieller Unterstützung der „Fraunhofer Einrichtung für experimentelles Software Engineering" und der „Markant SW Software und Dienstleistung GmbH" an der Universität Kaiserslautern durchgeführt.

eine geeignete Systemarchitektur ein und verfeinern die Anbindung des RDBMS an die objektorientierte Programmiersprache mit Blick auf unser Projekt. Die Frage nach der Eignung einer objektorientierten Entwicklung im Umfeld eines solchen Datenbanksystems entscheidet sich in realen Projekten an der Leistungsfähigkeit dieser Anbindung. Es wird sich zeigen, daß unsere Entwicklungsmethodik einen sauberen Übergang unterstützt und damit die Basis für eine optimale Realisierung bietet.

In Abschnitt 4 werden wir ein integriertes Werkzeug vorstellen, das augenblicklich entwickelt wird, um uns im Entwicklungsprozeß sowohl bei der Modellierung als auch bei der Implementierung zu unterstützen. Abschließend fassen wir in Abschnitt 5 unsere Ergebnisse nochmals zusammen und schließen mit einem Ausblick.

2. Softwareentwickungsprozeß für datenintensive Anwendungen

In der Einleitung haben wir bereits deutlich gemacht, daß in unserem Projekt die Nutzung eines OODBMS aufgrund der Anforderungen nicht möglich ist. Da die Objektorientierung wegen ihrer Vorteile für die Entwicklung der Applikationslogik maßgebend sein soll, besteht die Notwendigkeit einer Abbildung der entstehenden objektorientierten Strukturen mit ihrer Funktionalität auf relationale DBMS. In den folgenden Abschnitten wird dargestellt, wie sich diese Abbildung in den Entwurfsprozeß einbettet, den unser Projektpartner verfolgt.

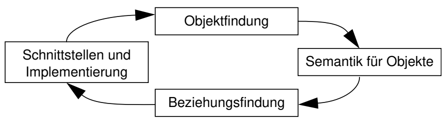

Abb. 1: Mikrozyklus

2.1 Der objektorientierte Softwareentwicklungsprozeß

Ziel eines Entwicklungsprozesses für objektorientierte Software ist, wie der Name schon sagt, die Erstellung von objektorientierten Softwaremodulen. Nun muß der Frage nachgegangen werden, ob die objektorientierte Entwicklungsmethodik spezielle Eigenschaften besitzt, die sie von bisher eingesetzten Methodiken zur Entwicklung datenintensiver Anwendungen unterscheidet.

Unser Projektpartners verfolgt einen Entwicklungsprozeß, der sich an den Vorschlägen von Booch orientiert. Es handelt sich um ein inkrementelles, iteratives Vorgehen, in dem immer wieder die verschiedenen Phasen des sogenannten Mikroprozesses durchlaufen werden (Abb. 1).

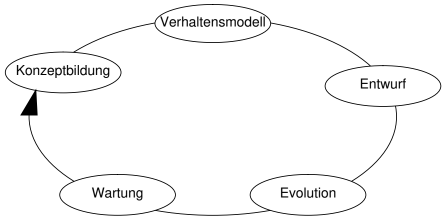

Abb. 2: Makrozyklus

Der Mikroprozeß beginnt jeweils mit der Suche nach Objekten. Daran schließt sich die Ermittlung der konkreten Semantik einzelner Objekte an, d.h., man identifiziert ihre Attribute und Methoden. Wesentlich für die Gesamtsemantik ist die Findung der Beziehungen zwischen den Objekten. Dafür ist ein separater Schritt im Mikroprozeß vorgesehen. Während die ersten drei Schritte die Objekte im wesentlichen von außen betrachten, konkretisiert man im vierten Schritt die interne Realisierung.

Innerhalb eines Entwicklungsprozesses wird der Mikrozyklus ständig durchlaufen und das entstehende System dabei zyklisch erweitert und verfeinert. Darin unterscheidet sich das Vorgehen von klassischer Entwicklungsmethodiken. Man versucht im Mikrozyklus zu einem stabilen Modell bezüglich einer „gewissen“ Abstraktions- bzw. Verfeinerungsstufe zu gelangen. Diese wird durch die unterschiedlichen Phasen des Makroprozesses bestimmt (Abb. 2). Kriterien für das Erreichen der angestrebten Abstraktionsstufe sind z. B.

- Reviews *verschiedener* Experten
  Da im Prozeß Domänenexperten, Systemanalytiker, Entwickler, Benutzer und andere beteiligt sind, werden die Ergebnisse (Dokumente) der Prozeßschritte aus sehr unterschiedlichen Blickwinkeln betrachtet, die zur Vervollständigung des Modelles führen.

- Explizite Ziele
  Einige Makroprozeßschritte haben explizite Ziele, so soll (nach Booch) der Mikroprozeß der Phase „Entwurf“ solange iteriert werden, bis es zu einem lauffähigen Kernsystem kommt.

- Dynamisches Objektverhalten
  Für die „Entdeckung“ fehlender Objekte spielen vor allem die *Interaktionsdiagramme* von Booch eines wesentliche Rolle. Sie beschreiben den (Auslöse-) Zusammenhang zwischen verschiedenen Nachrichten eines oder mehrerer Objekte aus einer übergeordneten Sicht. Insbesondere können mit diesen Diagrammen *Abläufe* in der Anwendungswelt modelliert werden.

Während nun der Mikroprozeß die eigentlichen Tätigkeiten in der Entwicklung steuert, legen die Makroprozeßschritte unterschiedliche Schwerpunkte für die jeweiligen Phase des Mikroprozesses fest. Über den gesamten Entwurfsprozeß hinweg kommen zusätzliche Objekte hinzu, deren Beziehungen, Semantik und Verhalten modelliert werden. Nach der Phase „Verhaltensmodell“ sollte diese Anpassung jedoch für die wichtigsten Objekte abgeschlossen sein. Im „Entwurf“ liegt der Schwerpunkt auf der Systemarchitektur. Das Ziel dieser Phase ist eine prototypische Implementierung des Gesamtsystems. Die „Evolution“ dient der stetigen Verfeinerung dieser Realisierung. Die Iterationsgeschwindigkeit ist nun deutlich verlangsamt, weil nun die Codierung im Vordergrund steht. Nachdem das System im Einsatz ist, sind die Fehler zu lokalisieren und zu beheben, die im Betrieb auftreten. Die Entwicklung befindet sich nun in der Wartungsphase. Hier werden auch zusätzliche Anforderungen aufgedeckt, die letztlich zu einer erweiterten Entwicklung führen. Dieses Vorgehen unterscheidet sich insofern deutlich von klassischen Lebenszyklus- und Wasserfallmodellen für die Softwareentwicklung [Ja91, So92], als sich die Entwicklungsschritte des Mikrozyklus in jeder Phase des Makrozyklus wiederfinden.

Wie in der Einführung bereits gesagt, besteht die Motivation für eine objektorientierte Entwicklung in der Hoffnung, die Wiederverwendung bereits bestehender Software besser zu unterstützen als herkömmliche Entwicklungsmethoden. Dazu ist die Abstraktion von konkreten Objekten zu Klassen und deren weitere Generalisierung der Hauptansatzpunkt. Die vorgestellte Entwicklungsmethodik begünstigt einen „Bottom-Up“-Ansatz, der letztlich zu einer Menge von Klassen führt, die zusammen mit ihren Instanzen die vollständige Funktionalität realisieren.

2.2 Datenintensive Mehrbenutzeranwendungen

Betriebliche Informationssysteme benötigen zur Erfüllung ihrer Aufgaben einen leistungsfähigen Zugriff auf aktuelle und konsistente Daten. Diese Anforderungen an eine logisch zentralisierte Datenhaltung werden von Datenbanksystemen erfüllt. Für die Anwendungsentwicklung ergeben sich zusätzliche Aspekte, beispielsweise aus dem Mehrbenutzerbetrieb, die beachtet werden müssen, um ein leistungsfähiges Gesamtsystem zu gewährleisten.

Der Ansatz von Booch bietet lediglich die Möglichkeit, ein Objekt als persistent zu kennzeichnen, d.h. explizit zu spezifizieren, daß es eine beliebige Lebensdauer hat, die auch über die Laufzeit der Anwendung hinausgeht. Wird eine Klasse so gekennzeichnet, so bedeutet dies, daß ihre Instanzen persistent sein *können*. Es muß dann zur Laufzeit entschieden werden, ob eine Instanz tatsächlich dauerhaft gespeichert wird.

In der Methodik sind keine Konstrukte enthalten, mit denen die Realisierung der Persistenz beschrieben werden kann, vielmehr soll davon gerade abstrahiert werden. Man nimmt also implizit an, daß ein Objekt, welches im Rahmen der Verarbeitung benötigt wird, zu diesem Zeitpunkt *vorhanden*, *gültig* und *sichtbar* ist.

Aus unserer Sicht ist dies für reale Projekte jedoch nicht akzeptabel, da die Probleme der effizienten Speicherung, des effektiven Zugriffs und zu großen Teilen auch der Synchronisation vernachlässigt werden.

In der Entwicklung von Datenbankanwendungen mit herkömmlichen Methodiken stand bisher die Modellierung der persistenten Daten im Vordergrund. Anwendungsprogramme realisierten die vom Anwender gewünschte Funktionalität als Operationen auf den Daten. Die Orientierung an Funktionen mit relativ geringer Stabilität führte zu einer schlechten Wartbarkeit, erlaubte aber eine initial hohe Effizienz. Bei objektorientierter Vorgehensweise werden die Daten so modelliert wie sie dem Anwender erscheinen und direkt mit den Methoden versehen, welche die strukturellen Eigenschaften nach außen verbergen und die gewünschte Funktionalität realisieren. Dadurch ist eine bessere Wartbarkeit zu erwarten, da Klassen stabiler sind als Funktionen. Allerdings werden viele Details verborgen, die man berücksichtigen muß, um ein optimales Laufzeitverhalten zu erreichen. Durch die stärker an der Vorstellung der Anwender ausgerichtete Entwicklung entstehen zudem Anwendungen mit verändertem Zugriffsverhalten. So führt der Wunsch nach einer graphischen Benutzerschnittstelle mit flexibler und komfortabler Benutzerführung zu Applikationen, bei denen der Ablauf der Aktionen nicht starr vorgegeben ist, sondern im wesentlichen von den Entscheidungen des Anwenders gesteuert wird. Es kommen beispielsweise verstärkt Browser und ähnliche Werkzeuge zum Einsatz, wodurch man auf größere Datenmengen zugreift, als für die eigentliche Aktion notwendig sind. Dies verlangt zum einen nach einer größeren Systemleistung, zum anderen besteht im Mehrbenutzerbetrieb eine höhere Konfliktwahrscheinlichkeit. Beides muß schon im Entwurf berücksichtigt werden.

Die Abstraktion von der Realisierung der Persistenz in der objektorientierten Entwicklung führt zudem dazu, daß bei der Modellierung der Applikationslogik von den Verhältnissen ausgegangen wird, die man bei der Verarbeitung transienter Objekte vorfindet, also einer kleinen Zahl von Objekten, auf die schnell zugegriffen werden kann. Bei der Verarbeitung persistenter Objekte ist jedoch mit großen Datenmengen und einem teuren Zugriff zu rechnen. Dies muß man bei der Auswahl der Algorithmen zur Realisierung der Funktionalität beachten.

Die Aspekte der persistenten Speicherung von Objekten können also nicht vollständig vernachlässigt werden. Hinzu kommt die Abbildung des modellierten Objektmodells auf das Datenmodell der Programmiersprache. Sie muß ebenso im Rahmen des Entwicklungsprozesses entworfen werden wie die Umsetzung des objektorientierten Programmiermodells auf das API (application programming interface) des DBS (Abschnitt 3.2.2).

Wir sehen es daher als notwendig an, die Entwurfsmethodik so zu verfeinern, daß dabei die zusätzlichen Anforderungen ebenfalls systematisch erfaßt werden. Ziel ist die Berücksichtigung der persistenten Datenhaltung, ohne die Entkopplung von Applikationsentwurf und Datenbank-Design aufzugeben, um die Vorteile der objektorientierten Entwicklung nicht zu verlieren. Wir verstehen unter Datenbank-Design neben dem klassischen Datenbankentwurf auch alle anderen datenbankorientierten Entwicklungsschritte, wie beispielsweise die Überbrückung der unterschiedlichen Programmiermodelle.

Die folgende Vorgehensweise berücksichtigt dies und unterstützt uns insbesondere bei der Anbindung des objektorientierten Systems an ein RDBMS. Beim Einsatz eines OODBMS gestalten sich die Prozeßschritte zwar weniger aufwendig, sind aber ebenfalls notwendig, um die Gesamtleistung des Systems sicherzustellen.

2.3 Die Erweiterung des Entwurfsprozesses

Die bisher dargestellte Entwicklungsmethodik (Abb. 1, Abb. 2) stellt das Objekt mit seiner Funktionalität aus der Sicht des Anwenders in den Vordergrund. Nicht-funktionale Anforderungen, wie z. B. Zugriffshäufigkeit oder Anzahl paralleler Zugriffe werden nicht oder nur indirekt erfaßt. In der 3-Schema-Architektur nach ANSI/SPARC sind solche Informationen zur Realisierung und Optimierung des „Internen Schema“ jedoch unbedingt erforderlich. Faßt man in einer solchen Architektur das Objektmodell zusätzlich „nur“ als externes Schema auf, so müssen mindestens die persistenten Objekte auch noch auf ein konzeptuelles Schema abgebildet werden, das dann als Schema der persistenten Daten gelten kann (nicht zu verwechseln mit dem Makroschritt der Konzeptualisierung im Prozeß).

Eine Schwierigkeit ergibt sich aus der Tatsache, daß gute Datenbankschemata (im Sinne einer optimalen Leistung für das Gesamtsystem) erst dann erstellt werden können, wenn der *gesamte* Informationsbedarf der Miniwelt bekannt ist.

Weiterhin muß für eine leistungsfähige Datenbankanbindung die Anfragemächtigkeit des DBMS ausgenutzt werden. In unserem Fall ist es deshalb notwendig an dedizierten Punkten die Kapselung der Programmiersprachenobjekte aufzugeben und uns auf die relationale Schnittstelle zu begeben. Dies gefährdet im laufenden Prozeß die Vorteile objektorientierter Entwicklung (vor allem der Wiederverwendung). Darum sollten solche Stellen bereits im Entwurf gefunden und dokumentiert werden. Hinzu kommt, daß die Fähigkeiten von DBMS zur dynamischen Schemaevolution eher beschränkt sind. Ein iterativer Entwurfsprozeß mit hoher Dynamik für den Datenbankentwurf ist aus diesen Gründen nur bedingt geeignet. Deshalb müssen die Prozeßschritte des Datenbank-Design zu stabileren Ergebnissen führen als diejenigen in den OO-Phasen.

Die Entwicklungsmethodik ist so zu verfeinern, daß die Problembereiche der Entwicklung datenintensiver Mehrbenutzeranwendungen ihre direkte Entsprechung in definierten Prozeßschritten finden. Ziel dabei ist, die persistente Datenhaltung zu berücksichtigen, ohne Applikationsentwurf und logisches, bzw. physisches, Datenbank-Design zu vermischen, d.h., die wechselseitigen Abhängigkeiten zwischen DB-Design und OO-Entwurf möglichst minimal zu halten.

Aus diesem Grund haben wir einen zweischichtigen Beschreibungsansatz gewählt und den OO-Mikrozyklus um eine Ebene erweitert (Abb. 3): Der DB-Mikrozyklus wird vom OO-Mikrozyklus getrieben, weist jedoch auch Rückkopplungen auf diesen auf: z. B., in dem Fall, daß aufgrund einer nicht-funktionalen Anforderung, die sich nicht allein durch ein geschicktes Schema erfüllen läßt, die OO-Modellierung neu überdacht werden muß. Die grundlegende Vorstellung besteht darin, in jedem Makroprozeßschritt zunächst den bekannten objektorientierten Mikrozyklus zu iterieren, bis dieser eine „gewisse“ Stabilität gewinnt. Nach und nach wird dann die zweite Ebene des Prozesses, der DB-Mikrozyklus, „mitgenommen“. Gekoppelt sind diese

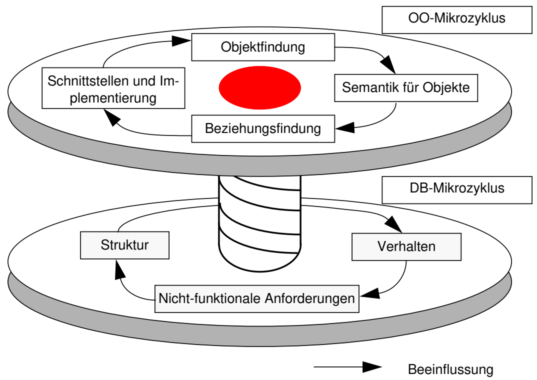

Abb. 3: DB-Mikrozyklus

Ebenen über das gemeinsame Phasenziel des Makroschrittes, so z. B. im „Entwurf“ die Realisierung eines Prototypen des Kernsystems. Die Frage, wann der Übergang vom einen zum anderen Mikrozyklus durchgeführt wird, ist dabei ein Erfahrungsparameter. Geschieht der Übergang zwischen den beiden Zyklen zu früh, so müssen eventuell viele Entscheidung des DB-Schemaentwurfs später revidiert werden. Ähnliches kann passieren, wenn Erkenntnisse aus der Schemaentwicklung zu spät in OO-Mikrozyklus rückgekoppelt werden (z. B. Design-Entscheidungen auf der Basis der Größen von beteiligten Mengen).

Wie schon der OO-Mikrozyklus so ist auch der DB-Mikrozyklus zwar in jedem Makroprozeßschritt enthalten, es verschieben sich jedoch auch hier die Schwerpunkte bzw. die Iterationsgeschwindigkeit. Während in der Analysephase sicherlich die nicht-funktionalen Anforderungen im Vordergrund stehen (die Struktur ist nur informell erfaßbar), liegt im „Entwurf“ der Schwerpunkt auf der Struktur und in der „Evolution“ letztlich auf dem Verhalten. Alle Entwurfsentscheidungen für die Anwendungslogik werden im OO-Zyklus getroffen, die Ergebnisse des DB-Zyklus dienen hier nur als Entscheidungsgrundlage. Die einzelnen Phasen der DB-Schemaebene werden im folgenden näher diskutiert:

2.3.1 Nicht-funktionale Anforderungen

Zweck: Die nicht-funktionalen Anforderungen beschreiben im wesentlichen die geforderte Leistungsfähigkeit eines Systems. Die bestimmenden Parameter müssen aus der Anwendungsdomäne ermittelt werden. Hier sind insbesondere folgende zu nennen:

- Mengengerüste für die beteiligten Objektmengen
- Häufigkeiten von Methodenausführungen
- Geforderte Leistungsparameter wie „turn-around“-Zeiten und Durchsatzanforderungen

Vorgehen: Ein Vorgehen für die Erhebung dieser Daten kann in etwa wie folgt aussehen:

- Für jede Klasse, die nicht abstrakt ist, wird abgeschätzt, wieviele Instanzen zu erwarten sind. Diese Information stellt die Grundlage für die Entscheidung über die Auswahl geeigneter Speicherungsstrukturen, z. B. Zugriffspfade, für das Datenbanksystem dar. Zusätzlich sind diese Größen bei der Abschätzung des Umfanges von Ergebnismengen mit zu berücksichtigen.
- Für jede Beziehung wird die durchschnittlich zu erwartende Kardinalität abgeschätzt. Auch diese Information dient zum einen der Auswahl der „Implementierung“ in der relationalen Datenbank und zum anderen der Entscheidung zwischen den möglichen Auflösungsstrategien dieser Beziehungen während der Laufzeit.
- Für jedes Interaktionsdiagramm, an dem persistente Objekte beteiligt sind, wird die Häufigkeit des Auftretens dieses Szenarios in einem vorgegebenen Zeitrahmen (Minute, Stunde, Tag, ...) bestimmt. Hier geht es um die Einschätzung der relativen Wichtigkeit von Optimierungen. Da nicht alle Anwendungen gleich gut unterstützt werden können, kann also nur bezüglich der zu erwartenden Mischung von Anwendungen ein optimales Ergebnis erzielt werden.
- Für jede Methode wird die Menge der Objekte bestimmt, die minimal bzw. maximal direkt von ihr verwendet werden (~ Anzahl der Nachrichten, die eine Methode auslöst). Damit kann abgeschätzt werden, wie aufwendig der Aufruf der Methode ist.
- Für jede dem Benutzer sichtbare Funktionalität (function point) wird die Obergrenze der Ausführungszeit ermittelt, d. h., für die einzelnen Arbeitschritte des Benutzers muß festgelegt werden, welche Antwortzeiten durch das System während der Durchführung maximal zulässig sind. So muß z. B. festgelegt werden, wie lang die höchstens zulässige Zeit für die Suche nach einem Kunden bei vorgegebener Kundennummer sein darf.

Ergebnis: Das Ergebnis dieser Phase ist die Dokumentation der gewonnenen Anforderungen in der Domäne.

Meßgrößen: Eine Bewertung des Ergebnisses ist schwierig, da sich die Qualität erst im konkreten Einsatz beweist. Allerdings kann man von einer erfolgreichen Analyse ausgehen, wenn dem Ergebnis sowohl Domänen-Experten als auch Datenbank-Experten zustimmen können.

2.3.2 Struktur

Zweck: Ziel der Phase „Struktur“ ist die Aufstellung eines konzeptuellen DB-Schemas für die vorhandenen Objekte. Je nach Datenmodell des Ziel-DBMS muß hierbei unterschiedlich viel Aufwand betrieben werden. Im einzelnen sind folgende Abbildungen vorzunehmen:

- Abbildung von Objekten
  Im Fall eines RDBMS als Zielsystem muß hierbei eine Aufteilung der Objektdaten auf Relationen ermittelt werden. Handelt es sich um ein OODBMS, so ist die Abbildung vielfach sehr einfach. Allerdings kann es auch hierbei sinnvoll sein, die Aufteilung oder Zusammenfassung von Objekten zu betrachten, um den nicht-funktionalen Anforderungen gerecht werden zu können (z. B. Aufteilung sehr großer Objekte, um Sperrkonflikte zu reduzieren, Clusterbildung).
- Abbildung von Beziehungen
  Bei den Beziehungen ist insbesondere die Abbildung der definierten Abstraktionsbeziehungen zu realisieren.
- Abbildung impliziter Eigenschaften
  In der objektorientierten Modellierung setzt man bei den Objekten implizit Eigenschaften voraus, die sich im verwendeten Datenbanksystem eventuell nicht wiederfinden. Bei Verwendung eines RDBMS muß beispielsweise ein eigenes Konzept zur Sicherung der Objektidentiät realisiert werden.

Vorgehen: Bei der Abbildung der Objekte gibt es einige Standardvorschläge (siehe auch Abschnitt 3.2.1). Wir gehen davon aus, daß in den ersten Phasen der Realisierung (Makroprozeßschritt „Entwurf“) zunächst eine direkte Umsetzung gewählt wird. Optimierungen, die beispielsweise einer Leistungsverbesserung dienen, werden erst dann vorgenommen, wenn ausreichende Kenntnis des Systemverhaltens vorliegt.

Ergebnis: Das Ergebnis dieser Phase ist ein DB-Schema, in dem die Daten aller persistenten Objekte inklusive ihrer Beziehungen abgebildet werden können. Da in den ersten Makroschritten die Objekte oft noch informell definiert sind, kann es jedoch sein, daß noch kein gültiges Schema im Sinne des verwendeten DBMS entsteht.

Meßgrößen: Der Prozeßschritt kann abschlossen werden, wenn alle definierten Objekte und Beziehungen abgebildet sind. Besondere Meßgrößen für die Bewertung ergeben sich aus der Kombination der Meßgrößen für den Datenbankentwurf (z. B. 1., 2., 3.-Normalform im Falle von RDBMS) und der Beachtung der nicht-funktionalen Anforderungen. Sie sind in den frühen Prozeßphasen mittels Review-Techniken durch Datenbank-Experten zu validieren. Später werden auf bereits realisierten Ausschnitten Testläufe durchgeführt und bewertet.

2.3.3 Verhalten

Zweck: Für die Erfassung des dynamischen Verhaltens von Anwendungen sind vor allem folgende Aspekte wesentlich:

- Anwendertransaktionen
  Unter Anwendertransaktionen verstehen wir eine Folge von Arbeitsschritten, die nur

zusammen eine Semantik aus Anwendersicht besitzen. Anwendertransaktionen können sich später aus mehreren Datenbanktransaktionen zusammensetzen, beschreiben also ein gröberes Granulat. Hier stehen vor allem die Isolationseigenschaft des bekannten ACID-Transaktionskonzeptes im Vordergrund. Dadurch wird in der Anwendung sichergestellt, daß Entscheidungen, die innerhalb dieser Arbeitsschritte getroffen werden, auf einer konsistenten Sicht auf den Datenbestand basieren.

- Rücksetzpunkte im Fehlerfall
  Da DBMS immer auch im Hinblick auf den Fehlerfall eingesetzt werden, ist es ebenfalls erforderlich, anwendungsorientierte Einheiten für mögliche Rücksetzungen bei Betriebsmittelverlusten (Rechnerabsturz, Deadlock, usw.) zu ermitteln. Sie stellen dem Anwender im wesentlichen die Atomaritätseigenschaft des ACID-Prinzips zur Verfügung.

- Ermittlung von Anwendungskontexten
  Ein Optimierungspotential, insbesondere in C/S-Umgebungen, ist die Möglichkeit die Anfrageverarbeitung an unterschiedlichen Orten, z. B. auf dem Client oder auf dem Server, bzw. zu unterschiedlichen Zeiten, z. B. erst bei Bedarf oder im Voraus, durchzuführen. Gerade beim Einsatz von RDBMS ist es entscheidend, die deskriptiven Anfragemöglichkeiten geeignet einzusetzen. Nach Möglichkeit sollten die benötigten *Datenmengen* mit wenigen Anfragen vom Server geholt werden. Hinweise darauf gewinnt man aus dem Kontext, den die Anwendung in nächster Zeit verarbeitet. Dies werden wir in Abschnitt 3.2.2 noch genauer aufgreifen.

Vorgehen: Die Grundlage zur Erfassung des Verhaltens einer Datenbankanwendung sind in der Booch-Methodik die Interaktionsdiagramme, in denen die Zusammenhänge zwischen Methodenaufrufen erfaßt werden. Im einzelnen sind nun folgende zusätzliche Schritte notwendig:

- Atomare Schritte definieren
  Für die Definition atomarer Schritte werden zusätzlich die referenzierten Objekte nach Informationsobjekten und Entscheidungsobjekten klassifiziert:

  Informationsobjekte dienen in der Applikation nur als prinzipielle Information, so daß deren Änderung keine Auswirkung auf die Applikation besitzt. Beispielsweise ist im Rahmen der Auftragserfassung die Adresse des Kunden vielfach nicht von Interesse, sie stellt lediglich eine Information am Bildschirm dar. Falls sie sich während der Auftragserfassung ändert, ist dies nicht problematisch.

  Entscheidungsobjekte sind Objekte, auf deren Basis eine Entscheidung getroffen wird. Daher dürfen sich diese Objekte nach der Entscheidung bis zum Abschluß des Anwendungsschrittes nicht mehr ändern. Ein solches Objekt ist im Rahmen der Auftragserfassung z. B. die Höhe der offenen Posten eines Kunden, an Hand derer entschieden wird, ob dem Kunden überhaupt noch etwas geliefert werden kann. Auch kann die Kundenadresse, die in der Auftragserfassung nur ein Informationsobjekt darstellte, in der Tourenplanung als Entscheidungsobjekt auftreten, so daß Objekte innerhalb der Gesamtapplikation sehr unterschiedlich benutzt werden.

- Rücksetzpunkte im Fehlerfall
  Die Rücksetzpunkte im Fehlerfall stellen Punkte im Rahmen der Verarbeitung dar, die als *anwendungsbezogene* Aufsetzpunkte dienen können. In der Realisierung muß gewährleistet sein, daß im Fehlerfall automatisch auf diese Punkte zurückgesetzt werden kann *und* daß der Anwender durch die Anwendung auch wieder zu diesem Punkt geführt wird.

- Abstrakte Beschreibung des Anwendungskontextes
  Ist die Datenmenge bekannt, die innerhalb eines Anwendungsschrittes verarbeitet wird, so kann die Datenanforderung leichter auf die Fähigkeiten des Servers zugeschnitten werden. Sie muß formal spezifiziert werden, um eine Optimierung zu ermöglichen (siehe Abschnitt 3.2.2).

Ergebnis: Als Ergebnis dieser Phase entstehen verfeinerte Interaktionsdiagramme, die um die genannten Aspekte erweitert sind.

Meßgrößen: Die Benutzertransaktionen sowie die Rücksetzpunkte stellen die Abstraktion der Datenbanktransaktionen dar. Bekanntermaßen sind lange Transaktionen für die Parallelität von Bearbeitungsvorgängen hinderlich. Daher sind Anwendertransaktionen *so kurz wie möglich* zu definieren. Zusätzlich muß in den Implementierungsphasen eventuell ein Kompromiß zwischen den Rücksetzpunkten und den Transaktionen gefunden werden, falls, wie bei vielen kommerziellen Systemen, zwischen diesen Konzepten keine Unterscheidung möglich ist.

2.4 Zusammenfassung

Die Erstellung objektorientierter Software wird durch eine objektorientierte Entwicklungsmethodik besonders vorteilhaft unterstützt, da die iterative Vorgehensweise der Realisierung solcher Software entgegenkommt. Wie wir aufgezeigt haben, muß die Methodik von Booch für datenintensive Mehrbenutzeranwendungen verfeinert werden, um den Anforderungen an derartige Applikationen gerecht zu werden. Unser Ziel war, die persistenten Datenhaltung zu berücksichtigen, ohne beim Applikationsentwurf den objektorientierten Ansatz aufzugeben. Ansonsten besteht die Gefahr, die Vorteile der Objektorientierung zu verlieren. Dies gilt insbesondere beim Einsatz eines DBS, das diese Form der Entwicklung schlecht unterstützt, beispielsweise einem RDBMS. Dazu haben wir einen separaten Mikrozyklus eingeführt, in dem die nicht-funktionalen Anforderungen sowie die Aspekte von Struktur und Verhalten abgeleitet werden. Hauptargument für diese Erweiterung ist die Tatsache, daß in der ursprünglichen Vorgehensweise viele der nicht-funktionalen Anforderungen an DB-gestützte Applikationen (Mehrbenutzerbetrieb, Durchsatzforderungen, ...) nicht berücksichtigt werden, sondern im wesentlichen die funktionalen Eigenschaften (Attribute und Methoden) im Vordergrund stehen. Damit haben wir den Entwurf des DB-Schemas und der Anbindung des DBMS für datenintensive Mehrbenutzeranwendungen in die objektorientierten Entwurfsmethodik nach Booch eingebettet, ohne frühzeitig die Vorteile der objektorientierten Softwareentwicklung aufzugeben.

3. Systemarchitektur

Im vorangegangenen Abschnitt haben wir den Softwareentwurfsprozeß beschrieben, der unserem Projekt zugrunde liegt. Der von Booch vorgeschlagene OOA/D-Entwicklungszyklus wurde mit dem Ziel erweitert, die Aspekte der persistenten Datenhaltung in einer Mehrbenutzerumgebung zu berücksichtigen und die Vorteile der Objektorientierung beizubehalten.

Wir wollen uns nun der Realisierung des Systems zuwenden. Die Applikationslogik soll sich auch dann objektorientiert implementieren lassen, wenn das verwendete DBMS diese Form der Entwicklung schlecht unterstützt. Ansonsten besteht die Gefahr, daß der objektorientierte Entwurf in der Realisierung zu einer prozeduralen Programmierung degeneriert. Dazu werden wir eine Systemarchitektur vorstellen, die eine weitgehend unabhängige Realisierung von Applikationslogik und Datenzugriff erlaubt. Die wechselseitigen Rückkopplungen werden bereits im Entwurfsprozeß berücksichtigt. Um ein leistungsfähiges System zu gewährleisten müssen die Ergebnisse aus dem datenbankorientierten Entwurfszyklus in die Entwicklung einer Anbindung einfließen.

Wie eine solche Anbindung realisiert werden kann, soll in den folgenden Abschnitten beschrieben werden. Die Betrachtungen orientieren sich dabei an unserem Projekt, also an der Anbindung eines RDBMS an eine Implementierung der Applikationslogik in einer OOPL.

3.1 Implementierungsmodell

Der Wunsch, Applikationslogik und Datenzugriff weitgehend zu trennen, legt eine „klassische" 3-Ebenen-Architektur (Abb. 4) nahe [HTW95].

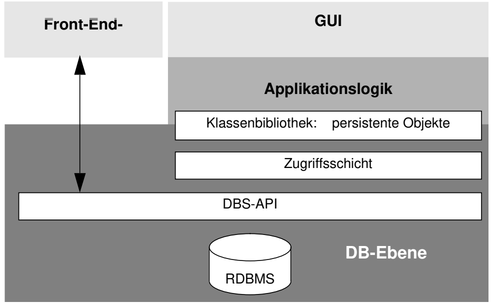

Abb. 4: 3-Ebenen-Architektur

Im Mittelpunkt der herkömmlichen OOA/D-Entwurfsmethodik steht die Entwicklung der Applikationslogik. Diese bildet die mittlere Ebene der Architektur und soll sowohl von der graphischen Benutzerschnittstelle, als auch vom Zugriff auf die persistenten Daten isoliert sein. Die Trennung nach oben vermeidet, daß der Benutzer sich schon bei kleinen Änderungen an eine neue Schnittstelle und Benutzerführung gewöhnen muß. Am Übergang von der Datenbankebene zur Applikationslogik wird von den Aspekten persistenter Datenhaltung abstrahiert. Statt dessen ist eine rein objektorientierte Programmierung angestrebt, bei der die Aspekte der Anbindung einer Datenbank verborgen sind.

Um diese Anforderung zu erfüllen, bietet die Datenbankebene nach oben einen Zugriff auf persistente Objekte im Objektmodell der Programmiersprache an, der sich übergangslos in deren Programmiermodell einbettet. Wird eine Datenbank mit anderem Objektmodell oder einem ungeeigneten Objektzugriff verwendet, beispielsweise ein RDBMS, so muß die Datenbankebene nochmals unterteilt werden. Auf der Schnittstelle des Datenbanksystems setzt eine Zugriffsschicht auf, mit der dessen Programmier- und Datenmodell verborgen werden. In die Entwicklung dieser Zugriffsschicht fließen die Ergebnisse aus dem datenbankorientierten Entwurfszyklus ein. Die Verbindung zwischen Applikationslogik und der Datenbankebene wird durch eine Klassenbibliothek realisiert, die dem Entwickler die persistenten Objekte zur Verfügung stellt.

Die Aufteilung der Gesamtarchitektur in drei grobe Ebenen ermöglicht es, jeden Aspekt weitgehend isoliert zu betrachten. Für die Datenbankebene bedeutet dies, daß deren interne Realisierung sich nicht auf die Applikationslogik auswirkt, solange am Übergang zwischen den beiden Ebenen die geforderte Funktionalität gesichert ist. Damit wird die Entwicklung vereinfacht und die Wartbarkeit erhöht. In objektorientierten Systemen läßt sich aufgrund der Kapselung der Datenzugriff in der Klassenbibliothek gut vor der Applikationslogik verbergen. Man gewinnt durch den Einsatz der Zugriffsschicht einen gewissen Grad der Unabhängigkeit vom Daten- und Programmiermodell des verwendeten Datenbanksystems (Datenmodellunabhängigkeit).

Im nächsten Abschnitt wollen wir die Datenbankebene verfeinern. Dazu werden die wesentlichen Aufgaben einer Zugriffsschicht analysiert, mit der ein RDBMS in eine OOPL integriert wird.

3.2 Zugriffsschicht

In der angestrebten Systemarchitektur abstrahiert die Zugriffsschicht vom verwendeten Datenbanksystem. Die Aufgaben der Zugriffsschicht hängen vom Datenmodell des DBS und dessen API ab. Es gilt, die Unterschiede im Daten- und Programmiermodell zu überbrücken.

In unserem Projekt wird ein RDBMS verwendet. Hier muß sowohl der Übergang vom Objektmodell der Programmiersprache Smalltalk in das relationale Datenmodell durch die Zugriffschicht realisiert werden als auch eine Abbildung der navigierenden Verarbeitung auf die mengen- bzw. cursororientierte SQL-Schnittstelle. Hinzu kommen Aufgaben, die sich aus dem Einsatz in einer Client/Server-Umgebung im Mehrbenutzerbetrieb ergeben. Gleichzeitig ist die Gesamtleistung des Systems zu beachten. Bei der Realisierung der Zugriffsschicht müssen daher die Stärken des DBMS ausgenutzt werden. Mit den Anforderungen, die sich in daraus ergeben, wollen wir uns im folgenden beschäftigen.

3.2.1 Statische Anbindung

Dieser Abschnitt wird sich mit der statischen Anbindung objektorientierter Anwendungen an ein RDBMS befassen, also den Aufgaben der Zugriffsschicht, die nicht direkt mit dem Datenzugriff zusammenhängen.

Eine wesentliche Eigenschaft objektorientierter Systeme ist die Identität eines Objektes, die für alle Zeiten garantiert werden muß. Im Gegensatz zum relationalen Datenmodell, in dem Tupel sich allein durch unterschiedliche Werte unterscheiden, kann es in der objektorientierten Welt mehrere Objekte mit gleichem Zustand aber unterschiedlicher Identität geben. Für jedes Objekt ist ein Objektidentifikator von der Zugriffsschicht zu vergeben, der im gesamten System eindeutig sein muß und einen direkten Zugriff auf das Objekt erlaubt. Es reicht daher nicht, einen Schlüssel zu benutzen, der nur innerhalb einer Relation eindeutig ist, weil dann kein direkter Zugriff nur über den Schlüssel möglich ist.

Die Zugriffsschicht muß auch die Abbildung der Objekte auf das Relationenschema, in dem die Daten gespeichert werden, realisieren. Nach oben bietet die Zugriffsschicht eine objektorientierte Sicht auf die Daten an, während sie selbst auf dem relationalen System aufsetzt. Für die Freiheitsgrade bei der Abbildung eines Objektmodells in ein Relationenschema verweisen wir auf [KGZ93, AKK95, Sch95]. Im wesentlichen werden Attribute komplexer Objekte, die nur atomare Werte annehmen können, auf Attribute einer Relation abgebildet. Komplexe Komponenten speichert man in einer eigenen Relation und verbindet sie mit dem Aggregat durch eine Fremdschlüsselbeziehung. Mengenwertige Attribute können teilweise auf Fremdschlüsselbeziehungen abgebildet werden, teilweise muß man eigene Relationen dafür anlegen. Das Verhalten der Objekte läßt sich nicht in der Datenbank ablegen. Die Zuordnung von Objektdaten und Verhalten wird durch die Zugriffsschicht zur Laufzeit durchgeführt. Mit Blick auf die Verwendung von Fremdwerkzeugen ist von eigenwilligen Speziallösungen abzusehen und ein verständliches Relationenschema zu verwenden.

Schwieriger ist die Abbildung der Abstraktionskonzepte. Weder das Objektmodell von Smalltalk noch das relationale Datenmodell kennen beispielsweise eine Aggregationssemantik, wie sie in der Methode von Booch modelliert werden kann. Daher muß die gewünschte Semantik durch die Zugriffsschicht gewährleistet werden.

Bei der Abbildung der Generalisierung gibt es einen Konflikt zwischen dem Verlangen nach hoher Leistung und dem Wunsch nach einem normalisierten Relationenschema. Bildet man die gesamte Vererbungshierarchie in eine Relation ab, so erzielt man tendenziell ein besseres Verhalten, weil die Daten von Instanzen einer Klasse immer mit einer einfachen Anfrage geladen werden können. Dabei nimmt man jedoch spärlich besetzte Tupel in Kauf. Eine partitionierte Abbildung, bei der jede Klasse eine eigene Relation erhält, erfordert entweder einen Verbund (bei Anfragen auf Unterklassen) oder eine Mengenvereinigung (bei Anfragen auf Oberklassen).

Eine Denormalisierung ist auch bei (1:1)-Beziehungen sinnvoll, die eine Existenzabhängigkeit ausdrücken. Man spart sich Datenanforderungen, wenn die Daten beider Objekte in einer Relation gespeichert werden, und steigert damit die Systemleistung.

Das verwendete Relationenschema hat wesentlichen Einfluß auf die Leistung des Systems, da es die Zahl und die Komplexität der Anfragen an den DB-Server bestimmt, mit denen die Objektdaten geladen werden. Daher muß die Zugriffschicht mit allen Abbildungsmöglichkeiten arbeiten können, um so für jede Situation ein leistungsfähiges System zu ermöglichen.

3.2.2 Dynamische Anbindung

Die Verarbeitung der Objekte erfolgt mit den Mitteln der Programmiersprache, in OO-Umgebungen also in erster Linie navigierend. Ausgehend von bekannten Objekten werden Beziehungen traversiert. Dieser Abschnitt soll die Ideen vorstellen, die wir im Projekt verfolgen, um eine leistungsfähige Abbildung des Smalltalk-Programmiermodells auf die cursor-orientierte SQL-Schnittstelle zu realisieren. Dies bezeichnen wir als die dynamische Anbindung.

Die Aufgabe der Zugriffschicht besteht darin, Einstiegspunkte in die Verarbeitung bereitzustellen und die durch Navigation erreichten Objekte unsichtbar nachzuladen. Für den Einstieg in die Verarbeitung bietet die Zugriffsschicht deskriptive Objektanfragen auf Klassen, die eine Menge von Objekten für die nachfolgende Verarbeitung liefern. Diese müssen in SQL-Anfragen an den DB-Server umgesetzt werden, welche die Daten der gewünschten Objekte liefern. Aus den Daten erzeugt die Zugriffsschicht Objekte (Objektgenerierung), welche in der Applikationslogik mittels Navigation entlang der Beziehungen verarbeitet werden. Dazu müssen eventuell die Daten weiterer Objekte aus der Datenbank geladen werden.

Um eine schnellere Verarbeitung im Hauptspeicher zu erlauben, liegt es nahe, die persistenten Beziehungen in Smalltalk-Referenzen umzusetzen (pointer swizzling [Mos92]). Die Umsetzung kann entweder sofort geschehen (*eager swizzling*), indem alle Beziehungen rekursiv verfolgt werden, oder bei Bedarf (*lazy swizzling*), wenn das Fehlen eines Objektes bemerkt wird. Die sofortige Umsetzung bringt eine schnellere Verarbeitung im Hauptspeicher, führt aber dazu, daß sehr viele nicht benötigte Objekte geladen werden. Andererseits steigt der Aufwand für die Verarbeitung, wenn erst bei Bedarf nachgeladen wird, da für jede Beziehung überprüft werden muß, ob sie bereits umgesetzt wurde. Eine ausführliche Diskussion der Vor- und Nachteile beider Verfahren kann der Literatur entnommen werden [KK93].

Wir wollen beide Verfahren mit Blick auf den Einsatz der SQL-Schnittstelle des DBMS betrachten. Das Nachladen von Objekten bei Bedarf kommt den Fähigkeiten relationaler Systeme nicht entgegen. Anfragen können erst formuliert werden, wenn ein benötigtes Objekt fehlt. Das führt zu sehr vielen einfachen Anfragen mit kleinen Ergebnismengen. Der mengenorientierte Zugriff und die Mächtigkeit der Anfragesprache SQL werden nicht ausgenutzt, da für jedes Objekt eine Anfrage abgesetzt wird. Das RDBMS dient in diesem Fall als Objekt-Server. Bei einer sofortigen Umsetzung aller Beziehungen kann der Datenzugriff anhand des Schemas vorgeplant werden, mit dem Ziel, den Aufwand zum Laden der benötigten Daten zu minimieren. Dadurch läßt sich das RDBMS als Query-Server nutzen. Dabei sind die Zahl der Serveranfragen, deren Komplexität und der Aufwand für die Objektgenerierung zu berücksichtigen.

Um die Vorteile beider Verfahren ausnutzen zu können, fordern [KK93] die Anpassungsfähigkeit der Umsetzungsstrategie an das jeweilige Anwendungsprofil. In unserer Anwendungswelt läßt sich das Zugriffsverhalten einer Applikation gut vorhersagen. Im Gegensatz zu Entwurfs- und Entwicklungsumgebungen (CAD, CASE, ...) bestehen die Arbeitschritte nicht aus kreativen Prozessen, sondern im wesentlichen aus häufig auftretenden Routinearbeiten. Die Aktionen innerhalb eines Arbeitsschrittes werden während des OOA/D-Zyklus modelliert und dokumentiert, sind also bekannt. Die Zugriffsschicht kann dies ausnutzen, indem sie eine kontextspezifische Umsetzung anbietet. Die dazu notwendigen Informationen stellen wir mit sogenannten Anwendungskontexten bereit.

Ein Anwendungskontext beschreibt die Objektmenge, die eine Anwendung in nächster Zeit verarbeiten wird und daher sofort vollständig geladen werden kann. Zur Beschreibung gehören außerdem die Zugriffsstrategie, mit der die benötigten Daten geladen werden, und die Umsetzungsstrategien für die Beziehungen innerhalb des Kontextes.

Kontexte werden nicht vom Anwendungsprogrammierer festgelegt, sondern ergeben sich aus der Modellierung, wenn in späten Entwicklungsphasen Optimierungsmöglichkeiten gesucht werden. Sie müssen von Modellierern spezifiziert und dann in ein Format übersetzt werden, das die Zugriffsschicht interpretiert.

Die Zugriffsstrategie legt fest, mit welcher Folge von Anfragen die benötigten Daten vom Server geladen werden. Um die Zahl der Anfragen zu reduzieren, kann man einen Zugriffsplan für den Kontext aufstellen, mit dem Ziel, zum Laden der Daten von Objekten einer Klasse nur eine Anfrage an den Server abzusetzen. Löst man beispielsweise eine (1:n)-Beziehung zu Objekten auf, die im gleichen Kontext noch in einer (1:1)-Beziehung zu weiteren Objekten stehen, so lassen sich beide Beziehungen mit einer Anfrage auflösen, indem die Daten mit einem Verbund verknüpft werden. Abb. 5 macht anhand eines Beispiels aus unserer Anwendungswelt deutlich, wie man mit nur einer Anfrage n Bestellpositionen und die zugehörigen Artikel laden kann. Ohne Kontextinformation ist für jeden Artikel eine eigene Anfrage notwendig. Man kann leicht weitere Szenarien finden, in denen es sich lohnt, Anfragen vorzuplanen. Wir werden später auf eine asynchrone Anbindung eingehen, bei der auch die Reihenfolge der Anfrageauswertung interessant ist. Es bedarf allerdings noch weiterer Erfahrung, um Regeln für mögliche Optimierungen aufzustellen.

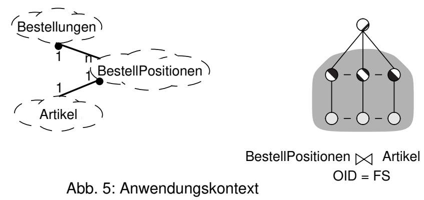

Abb. 5: Anwendungskontext

Es liegt nahe, *innerhalb* eines Kontextes alle persistenten Beziehungen sofort in Smalltalk-Referenzen umzusetzen und damit die Vorteile des *eager swizzling* auszunutzen. Beziehungen, die den Kontext *verlassen*, werden erst bei Bedarf aufgelöst. Damit hat man die Vorteile beider Verfahren kombiniert: Innerhalb eines korrekt spezifizierten Kontextes fällt kein zusätzlicher Aufwand für die Verarbeitung an, es werden keine nicht benötigten Objekte geladen und die Zugriffsstrategie kann optimiert werden. Dennoch lohnt es sich auch in Kontexten unter Umständen, eine *lazy*-Strategie einzusetzen, da man dadurch Freiheiten für die Datenbeschaffung in der Zugriffsschicht gewinnt. Dazu führt man eine zweistufige Umsetzung durch: Sobald die Anwendung den Kontext betritt, wird die Datenbeschaffung angestoßen, die Umsetzung der Referenzen und die Generierung der Objekte finden aber erst bei Bedarf statt. Dabei ist der Zugriffsschicht bekannt, daß sie sich innerhalb eines Kontexts befindet und die benötigten Daten bereits angefordert wurden.

Die Unabhängigkeit von Client und Server erlaubt uns nun, bei der Auswertung von Kontexten parallel zu arbeiten. Voraussetzung ist allerdings eine asynchrone Schnittstelle zum DBS, die uns im Projekt in Form der verwendeten ODBC-Schnittstelle vorliegt. Durch die Kontextspezifikation sind alle Anfragen und die Auswertungsreihenfolge bekannt. Indem die Objektgenerierung aus einer empfangenen Datenmenge im Client mit der Abarbeitung der nächsten Anfrage auf dem Server überlappt, können beide parallel arbeiten.

Eine zweistufige Umsetzung erlaubt auch, die Verarbeitung in der Applikationslogik mit der Anfrageauswertung zu überlappen. Sobald die ersten Ergebnisdaten in Objekte umgewandelt sind, kann die Applikation weiterarbeiten. Bei der nächsten Objektanforderung wird die Applikation kürzer verzögert, da der Server inzwischen in der Anfrageverarbeitung fortgeschritten ist und die Daten eventuell schon vorhanden sind. Solange der Server ausreichend schnell antwortet und die Anwendung den Kontext nicht verläßt, wirkt sich lediglich die Generierung der Objekte leistungsmindernd aus.

Eager swizzling bringt gegenüber einer *lazy*-Strategie nur Vorteile, wenn die Anwendung mit hoher Lokalität arbeitet und häufig Referenzen mehrmals traversiert. Die parallele Auswertung ist günstig, wenn Referenzen selten mehrfach verfolgt werden, da dann die Nachteile der *lazy*-

Strategie nicht ins Gewicht fallen, die Applikation aber seltener und kürzer blockiert ist.

In unserer Anwendungswelt treten häufig Beziehungen auf, an denen sehr viele Objekte teilnehmen. Solche Beziehungen vollständig aufzulösen ist kritisch, da sehr große Datenmengen in den Client eingelagert werden müssen. Da man häufig nur einen kleinen Ausschnitt aus der Objektmenge gleichzeitig bearbeitet, benutzt man die zweistufige Umsetzung, um die Daten paketweise vom Server entgegenzunehmen. Man hält nur die Objekte auf dem Client, die zur Verarbeitung benötigt werden.

Die bisher beschriebenen Techniken hatten eine Einbettung des Datenzugriffs in die Programmiersprache zum Ziel, die dem Anwendungsentwickler verborgen bleibt. Für manche Szenarien ist die Verarbeitung auf dem Client jedoch nicht sinnvoll, da große Datenmengen ausgewertet werden, um die gewünschte Information zu beschaffen. Solche Aufgaben möchte man gerne an den Server delegieren, wenn dieser entsprechende Fähigkeiten besitzt. Die SQL-Schnittstelle unseres RDBMS erlaubt zwar mächtige deskriptive Anfragen, eine entsprechende Einbettung durchbricht jedoch immer die Kapselung der Objekte. Es empfiehlt sich deshalb, solche Optimierungen erst in späten Phasen des Prozesses einzusetzen, wenn Struktur und Verhalten der Objekte relativ stabil sind.

Auswertungen, die sich besser auf dem Server ausführen lassen, werden in der Modellierung identifiziert und explizit modelliert. Dazu führen wir einen neuen Beziehungstyp in das Objektmodell ein, den wir als generischen Link bezeichnen. Ein solcher Link ist gerichtet und stellt eine dynamische Beziehung dar, die normalerweise durch eine Methode realisiert wird. Bei ausreichend stabilem Objektmodell wird er mit einer deskriptiven Beschreibung der Eigenschaften versehen, die man von den Zielobjekten fordert. Ein generischer Link wird erst ausgewertet, wenn man entlang der Beziehung traversiert. Da die Beziehung durch den Zustand der Objekte bestimmt ist, muß sie bei jedem Traversieren erneut überprüft werden. Es könnten nun bereits andere Objekte teilnehmen. Man kann generische Links aus dem gleichen Grund auch nicht explizit erzeugen oder löschen. Die Wartung ist daher aufwendiger als bei einer normalen Beziehung.

Generische Links sind keine deskriptiven Anfragen, wie sie zu Anfang des Abschnitts angesprochen wurden. Sie stehen dem Entwickler nicht als Programmierhilfe zur Verfügung. Er sieht sie lediglich als zusätzliche Beziehung und muß sie adäquat implementieren. Jedoch wird, wie bei Anwendungskontexten, die Beschreibung in der Modellierung festgelegt und später in ein Format übersetzt, das die Zugriffsschicht auf Serveranfragen abbilden kann. Damit kann man die Implementierung gegen eine neue Version austauschen, die den dynamischen Link vom Server auswerten läßt, ohne die Anwendung anpassen zu müssen. Beispielsweise besteht zwischen einem Lieferanten und seinen Lieferungen eine (1:n)-Beziehung. Für manche Anwendungsfälle ist es notwendig nur die Lieferungen der letzten fünf Wochen zu betrachten. Es ist wesentlich günstiger diese Auswertung auf dem Server durchführen zu lassen, inbesondere wenn viele Lieferungen an der Beziehung teilnehmen. Dies geschieht für den Entwickler transparent in einer Methode deren Implementierung vom Code-Generator aus der Definition des generischen Links erzeugt und später anstelle der programmierten Methode verwendet wird.

3.2.3 Client/Server-Umgebung

Bisher sind wir im wesentlichen auf die Anbindung des DBMS an die Programmiersprache eingegangen und haben die Besonderheiten der Betriebsumgebung weitgehend außer acht gelassen. Dieser Abschnitt soll kurz einige Aspekte einführen.

Aus der Sicht der Anwendung sorgt die Zugriffsschicht dafür, daß benötigte Daten stets auf dem Client vorhanden sind. Geladene Objekte werden gepuffert, um den Bedarf möglichst ohne Serverzugriff befriedigen zu können und so Lokalität auszunutzen. Damit der Server bei der Auswertung von Anfragen eine korrekte Sicht auf die Daten bekommt, muß man geänderte Objekte vorher propagieren, was mit großem Aufwand verbunden ist. Die Zugriffsschicht muß das Propagieren daher solange verzögern, bis es tatsächlich erforderlich ist, und nur die Daten schreiben, die von der Auswertung betroffen sind.

Da in unserem Projekt auch der Einstieg mit Fremdwerkzeugen möglich sein soll, können wir keine Cache-Kohärenz-Protokolle oberhalb der SQL-Schnittstelle einsetzen. Daher muß am Ende jeder Transaktion der Inhalt des Puffers an den Server propagiert werden. Lokalität über Transaktionsgrenzen ist nur dann ausnutzbar, wenn man optimistische Synchronisationsprotokolle auf dem Client einsetzt. Diese eignen sich allerdings nur bei hoher Lokalität und sehr niedriger Konfliktwahrscheinlichkeit [GR93]. Daher unterstützt die Zugriffsschicht sowohl eine pessimistische als auch eine optimistische Synchronisation, bzw. aufgrund des oben angesprochenen vorzeitigen Propagierens eine semi-optimistische Strategie [KB96].

Ein weiteres kritisches Problem ist die Unsicherheit des Client im Fehlerfall. Zwischen dem erfolgreichen Ende einer Transaktion und der Rückmeldung für den Benutzer entsteht eine kritische Lücke. Tritt hier ein Fehler auf, wird der Anwender im Zweifel gelassen, ob seine letzte Aktion erfolgreich war oder nicht. Dazu muß ein persistentes Log der erfolgreichen Anwendungsschritte realisiert werden, das bei Bedarf analysiert werden kann.

3.3 Zusammenfassung

Die eingeführte 3-Ebenen-Architektur erlaubt die Umsetzung des vorgeschlagenen Entwurfszyklus bei der Realisierung und ermöglicht die weitgehend getrennte Entwicklung von Benutzerschnittstelle, Applikationslogik und Datenzugriff. In der Zugriffsschicht bündeln sich die technischen Probleme beim Übergang vom Objektmodell der OOPL zum Datenmodell des DBS. Während die statische Anbindung eines DBS relativ einfach ist, muß für eine leistungsfähige dynamische Anbindung Anwendungswissen in die Ansteuerung der Zugriffsschicht einfließen. Die Zugriffsschicht unterstützt mit den oben beschriebenen Techniken die Ausnutzung der Ergebnisse, die man mit der von uns vorgeschlagenen Entwicklungsmethodik gewinnt. Sie verbindet die Anwendungslogik mit dem Datenbanksystem und ermöglicht eine Einbettung in die Programmiersprache, die den Datenzugriff verbirgt. Dadurch sind auch prototypische Implementierungen einfach möglich. Spätere Optimierungen, die sich aus den nicht-funktionalen Anforderungen ergeben, können modelliert und ohne Änderung der Applikationslogik eingesetzt werden. Hierzu bietet die Zugriffsschicht Anwendungskontexte mit flexiblen Umsetzungs- und Zugriffsstrategien an. Der Einsatz generischer Links erlaubt es, Aufgaben an den Server zu delegieren und so dessen Fähigkeiten auszunutzen, ohne die Kapselung frühzeitig zu durchbrechen.

4. Werkzeugunterstützung

Bei der Entwicklung von Software stellt sich immer auch die Frage, welche Werkzeuge die gewählte Entwicklungsmethodik unterstützen können. Im Rahmen unseres Projektes wird in den Prozeßschritten des objektorientierten Entwurfs nach Booch ein kommerzielles CASE-Werkzeug einsetzt. Für die Aufgaben des DB-orientierten Entwurfs ist es allerdings weniger geeignet. Die Abbildung der objektorientierten Strukturen auf relationale Datenbanksysteme wird nur rudimentär, die Erfassung des Verhaltens und der nicht-funktionalen Anforderungen nicht unterstützt. Für diese Bereiche gibt es einige kommerzielle Frameworks, die zum Teil von uns im Rahmen des Projektes evaluiert worden sind [Top95, Wa96]. Die Leistungsbreite solcher Frameworks reicht von der Abbildung der Klassen auf Relationen bis zu einer Unterstützung der Dynamik durch in Zugriffsmethoden gekapselte Datenzugriffe.

Im Rahmen unseres Projektes entwickeln wir ein Werkzeug, das sich nahtlos in die bereits bestehende Werkzeugwelt des Projektpartners integriert, dessen CASE-Daten (die im wesentlichen Struktur und Funktionalität beschreiben) benutzt und sie für einen applikationsspezifischen DB-Entwicklungszyklus bereitstellt (Abb. 6). Aufbauend auf diesen Daten werden mit Hilfes des Werkzeuges die nicht-funktionalen Anforderungen erfaßt, die damit in einheitlich spezifizierter Form vorliegen. Diese Informationen nutzt man für den konzeptuellen DB-Entwurf, die Optimierung des internen Schemas und insbesondere die dynamische Anbindung.

Es bietet sich an, aus den vorliegenden Beschreibungsdaten eine Klassenbibliothek zu generieren, deren Nutzung dem Entwickler die persistenten Objekte bereitstellt. Diese Vorgehensweise wird auch von anderen Werkzeugen verfolgt [AKK95, KJA93]. Die Klassenbibliothek verbindet Applikationslogik und Zugriffsschicht (Abb. 4). Die Objektanforderungen an die Zugriffsschicht sind für den Anwendungsprogrammierer unsichtbar in den Zugriffsmethoden für Attribute der persistenten Objekte gekapselt. Zusammen bilden Klassenbibliothek und Zugriffsschicht einen Objekt-Cache. Die Klassenbibliothek manuell zu implementieren und zu warten, ist aufgrund des Aufwands nicht vertretbar. Die iterative Vorgehensweise bei der Entwicklung hätte zur Folge, daß auch die Bibliothek ständig angepaßt werden muß.

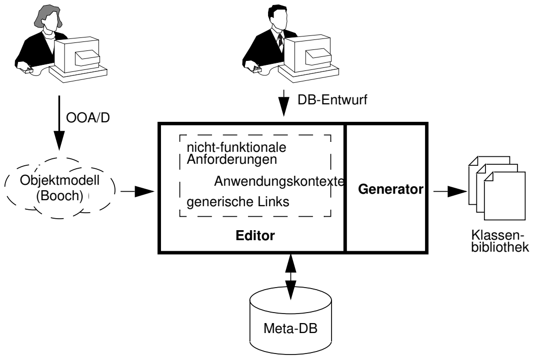

Abb. 6: Werkzeugeinsatz

In Abschnitt 3.2.2 haben wir beschrieben, wie mit Kontexten die Integration von Anwendungswissen zur Leistungssteigerung erreicht wird. Alle Applikationen nutzen zwar das gleiche Objektmodell, unterscheiden sich jedoch stark in ihren Zugriffsmustern. Diese werden in den erweiterten Interaktionsdiagrammen erfaßt (Abschnitt 2.3.3), wodurch die Kontextspezifikation erleichtert wird. Zur Laufzeit werden Anwendungskontexte bei der Zugriffsschicht von der erzeugten Klassenbibliothek angefordert. Die Generierung erlaubt uns, *jede* Applikation mit einer applikationsspezifischen Klassenbibliothek zu versorgen, *ohne* daß die Übersichtlichkeit des Entwurfs leidet. Dadurch können wir die Zugriffsschicht für die Bedürfnisse einzelner Anwendungen optimal parametrisieren. Hiervon versprechen wir uns eine starke Verbesserung des Laufzeitverhaltens..

Um die eben beschriebene anwendungsspezifische dynamische Anbindung zu unterstützen, muß unser Werkzeug die formale Spezifikation von Anwendungskontexten und generischen Links ermöglichen. Eine entsprechende Sprachdefinition findet sich in [Zi96]. Da unterschiedliche Anwendungskontexte für verschiedene Applikationen notwendig sind, werden sie in Konfigurationen verwaltet. Die Zusammenhänge zwischen den modellierten Interaktionsdiagrammen und den daraus resultierenden Anwendungskontexten müssen mit Hilfe des Werkzeuges nachvollziehbar sein. Nur dann ist gewährleistet, daß sich Änderungen in der objektorientierten Modellierung auch für die Kontexte nachvollziehen lassen.

Das geplante Werkzeug unterstützt somit den gesamten Makroprozeß, von der Analyse der nicht-funktionalen Anforderungen bis hin zu einer Unterstützung der Implementierung.

5. Zusammenfassung und Ausblick

In dieser Arbeit haben wir die Ergebnisse dargestellt, die wir in einem Projekt mit der „Fa. Markant SW Software und Dienstleistung GmbH“ gewinnen konnten.
Die eingesetzte Software-Entwicklungsmethodik hat sich für datenintensive Anwendungen als unzureichend erwiesen. Im konkreten Projekt fehlte sowohl eine Berücksichtigung der Anforderungen, die sich aus der zentralisierten Datenhaltung ergeben, als auch eine Hilfestellung bei der Überwindung des Überganges von der objektorientierten Programmiersprache zum RDBMS. Durch den erweiterten Mikrozyklus werden nun die vernachlässigten Faktoren in den Entwurf eingebracht. Wir haben dargestellt, welche zusätzlichen Aufgaben in den drei neuen Phasen anfallen, welche Ziele man verfolgt und dazu ein schematisches Vorgehen skizziert.

Im zweiten Teil der Arbeit wurde die Systemarchitektur betrachtet, die wir zur Realisierung datenintensiver Anwendungen für geeignet halten. Die angestrebte Trennung von Applikationslogik und Datenbereitstellung wird durch den zweigeteilten Entwurfszyklus unterstützt. Sie ermöglicht eine weitgehend vom Datenzugriff isolierte Entwicklung der Anwendungen. Am Beispiel unseres Projektes haben wir verdeutlicht, welche Aufgaben einer Zugriffsschicht zukommen, die von den Eigenschaften des zugrundeliegenden DBS abstrahiert. Um ein angemessenes Systemverhalten zu erreichen, haben wir eine weitgehende Flexibilität angestrebt, damit der Datenzugriff auf verschiedenste Bedürfnisse zugeschnitten werden kann. Durch die applikationsspezifische Parametrisierung der Zugriffsschicht hoffen wir, die Fähigkeiten des Datenbanksystems an die Applikationslogik durchreichen zu können, ohne daß der objektorientierte Entwurf darunter leidet.

Die notwendige Werkzeugunterstützung war Thema im letzten Abschnitt. Erst ein generierender Ansatz erlaubt uns letztlich die applikationsspezifische dynamische Anbindung. Ein einheitliches Format zur Dokumentation der in der Analyse gewonnenen Anforderungen, des Anwendungsverhaltens und der Abbildungsentscheidungen ist Vorraussetzung für die Wartbarkeit des entstehenden Systems.

Viele der angestellten Überlegungen waren in erster Linie aufgrund der kaum standardisierten Einbindung existierender DBMS in OOPL notwendig. Insbesondere das Programmiermodell relationaler Systeme ist für die navigierende Verarbeitung in den Programmiersprachen ungeeignet. Solange OODBMS nicht die gleichen Eigenschaften aufweisen können, die für relationale Systeme sprechen, kann die Lösung nicht im Einsatz eines solchen Systems liegen. Vielleicht bieten die augenblicklich aufkommenden objekt-relationalen Systeme einen Ausweg, da sie die Vorteile beider Welten zu vereinigen glauben [St96].

In jedem Fall ist jedoch eine geschlossene Modellierung aller relevanten Informationen anzustreben. Insbesondere muß in der Zukunft der Bereich der nicht-funktionalen Anforderungen und des Verhaltens im Mehrbenutzerfall von den Werkzeugen besser unterstützt werden. Hier erwarten wir vom Einsatz unseres prototypischen Werkzeuges neue Erkenntnisse, wie derartige Informationen zu spezifizieren sind und welche Möglichkeiten einer frühzeitigen Validierung realistisch sind. Die Eignung der vorgestellten Erweiterung der Entwurfsmethodik nach Booch für andere Vorgehensweisen ist in weiteren Arbeiten noch zu klären.

6. Danksagung

Wir danken Oliver Flege und Theo Härder sowie den anonymen Gutachtern für die konstruktiven Hinweise zur Verbesserung einer frühen Fassung dieser Arbeit. Unseren Partnern bei der „Markant SW Software und Dienstleistung GmbH“ verdanken wir die Einführung in warenwirtschaftliche Fragestellungen.

7. Literatur

[AKK95] S. Agarwal, C. Keene, A. M. Keller: „Architecting Object Applications for High Performance with Relational Databases“, OOPSLA 95, Workshop on Object Database Behaviour, Benchmarks and Performance, Austin, Texas, 1995

[Bo94] G. Booch: „Object-Oriented Analysis and Design with Applications“, The Benjamin/Cummings Publishing Company, Inc., 1994

[BR95] G. Booch, J. Rumbaugh: „Unified Method for Object-Oriented Development“, Documentation Set, Version 0.8, Rational Software Corporation, 1995

[CY91a] P. Coad, E. Yourdon: „Object-oriented analysis“, Yourdon Press, 1991

[CY91b] P. Coad, E. Yourdon: „Object-oriented design“, Yourdon Press, 1991

[GLL96] J. Göers, K.-P. Lisson, H.-G. Linde-Göers: „Experiences in Object-Oriented Modeling of a Real Database Application“, Proc. DEXA 96

[GR93] J. Gray, A. Reuter: „Transaction Processing: Concepts and Techniques“, Morgan Kaufmann Publishers, 1993

[HTW95] W. Hahn, F. Toenniessen, A. Wittkowski: „Eine objektorientierte Zugriffsschicht zu relationalen Datenbanken“, Informatik Spektrum 18, Springer Verlag 1995

[Ja91] P. Jalote: „An integrated approach to software engineering“, Springer, 1991

[KB96] A. M. Keller, J. Basu: „A predicate-based caching scheme for client-server database architectures“, VLDB Journal, Vol. 5, No. 1, 1996

[KGZ93] K. Kilberth, Guido Gryczan, H. Züllighoven: „Objektorientierte Anwendungsentwicklung - Konzepte, Strategien, Erfahrungen“, Vieweg Verlag, 1993

[KJA93] A. M. Keller, R. Jensen, S. Agarwal: „Persistence Software: Bridging Object-Oriented Programming and Relational Databases“, Proc. of the ACM SIGMOD International Conference on the Management of Data, Washington D.C., 1993

[KK93] A. Kemper, D. Kossmann: „Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis“, VLDB Journal, Vol. 4, No. 3, 1995

[Mo92] J. E. B. Moss: „Working with Persistent Objects: To Swizzle or Not to Swizzle“, IEEE Transactions on Sofware Engineering, Vol 18, No. 8, 1992

[Ru91] Rumbaugh et al.: „Object-oriented modeling and design“, Prentice Hall, 1991

[Schr95] D. Schreiber: „Objektorientierte Entwicklung betrieblicher Informationssysteme“, Physica-Verlag, 1995

[So92] I. Sommerville: „Software engineering“, Addison-Wesley, 1992

[St96] M. Stonebraker: „Object-Relational DBMSs: The Next Great Wave“, Morgan Kaufmann Publishers, 1996

[Top95] Toplink 2.0 Entwicklerhandbuch, The Object People Inc., 1995

[Wa96] G. Wanner: „Optimierte Transformation“, Software Entwicklung 2/96, 1996

[Zi96] J. Zimmermann: „Anwendungsspezifische Anbindung relationaler Datenbanksysteme an objektorientierte Programmiersprachen“, Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, 1996