Integration relationaler Datenbanken in ein polymorphes, persistentes Objektsystem

Diplomarbeit

Michael Skusa
Universität Hamburg
Fachbereich Informatik

Betreuung:

Prof. Florian Matthes
Technische Universität Hamburg-Harburg
Arbeitsbereich Softwaresysteme

Dr. Ingrid Wetzel
Universität Hamburg
Fachbereich Informatik

9. September 1998

Zusammenfassung

Diese Arbeit zeigt, wie relationale Datenbanken systematisch in polymorphe, persistente Objektsysteme integriert werden. Für die Entwicklung komplexer, datenintensiver Anwendungen, die ihre Daten mit Hilfe objektorientierter Systeme verarbeiten, sie aber in relationalen Datenbanken dauerhaft speichern, müssen konzeptuelle Unterschiede zwischen beiden Systemklassen überwunden werden. Diese Arbeit untersucht, wie sich Dienste relationaler Datenbanken so in polymorphe, persistente Objektsysteme integrieren lassen, daß ein Datenbankzugriff ohne detaillierte Kenntnisse datenbankspezifischer Programmiersprachen oder -schnittstellen möglich ist. Diese Integration wird für ein konkretes polymorphes, persistentes Objektsystem durchgeführt und mit kommerziellen Lösungen verglichen.

Inhaltsverzeichnis

1 Einleitung **4**
- 1.1 Kontext . . . 4
  - 1.1.1 Relationale Datenbanken . . . 5
  - 1.1.2 Objektorientierte Systeme . . . 6
- 1.2 Motivation . . . 7
- 1.3 Ziel der Arbeit . . . 7
- 1.4 Aufbau der Arbeit . . . 8

2 Programmierschnittstellen für relationale Datenbanken **9**
- 2.1 Grundprinzipien . . . 9
  - 2.1.1 Eingebettetes SQL . . . 10
  - 2.1.2 Dynamisches SQL . . . 10
  - 2.1.3 Mengenwertiger Zugriff versus tupelorientierter Zugriff . . . 11
- 2.2 Programmierschnittstellen für objektorientierte Systeme . . . 12
- 2.3 Bewertung . . . 16

3 Anforderungen an den objektorientierten Datenbankzugriff **17**
- 3.1 Höhere Abstraktionen für Inhalte der Datenbank . . . 17
- 3.2 Implizite Erzeugung von SQL-Code . . . 19
- 3.3 Minimierung der Kommunikation mit der Datenbank . . . 19
- 3.4 Isolation . . . 19

4 Kommerzielle Werkzeuge für objektorientierten Datenbankzugriff **21**
- 4.1 Klassifikation objektrelationaler Middleware . . . 21
- 4.2 Kriterien für die Bewertung existierender Systeme . . . 22
  - 4.2.1 Abgebildete Konzepte . . . 22
  - 4.2.2 Einbettung von SQL . . . 23
  - 4.2.3 Isolation gegen parallele Datenbankzugriffe . . . 23
  - 4.2.4 Beschleunigung der Datenbankkommunikation . . . 23
- 4.3 DBTools.h++ und Object Factory . . . 24
- 4.4 Persistence . . . 26
- 4.5 ONTOS*Integrator . . . 29
- 4.6 Vergleich der unterschiedlichen Lösungen . . . 31

5 Entwurf einer objektrelationalen Klassenbibliothek für Tycoon **33**
5.1 Eigenschaften des objektorientierten Systems 33
5.1.1 Objektmodell 34
5.1.2 Polymorphes Typsystem 35
5.1.3 Persistenz 36
5.1.4 Speicherverwaltung 37
5.2 Generierung von Klassen für den Datenbankzugriff 38
5.2.1 Objekte für den direkten Zugriff auf relationale Daten 39
5.2.2 Erzeugung spezifischer Datensatzklassen 43
5.2.3 Metadaten für die Klassengenerierung 44
5.3 Synchronisation und Caching 47
5.3.1 Implizites Caching 47
5.3.2 Synchronisation 48
5.3.3 Eindeutige Identifikation von Objekten und Daten 51
5.3.4 Explizites Caching 52
5.3.5 Lebensdauer eines Datensatzobjektes 53
5.3.6 Caching externer Daten in persistenten Systemen 53
5.3.7 Behandlung von Abhängigkeiten zwischen Datensätzen 54
5.4 Abwicklung der Datenbankzugriffe 54

6 Implementierung der objektrelationalen Klassenbibliothek für Tycoon **57**
6.1 Einzelne Datensätze 57
6.1.1 Änderungsoperationen 58
6.1.2 Operationen auf Fremdschlüsselattributen 59
6.1.3 Objektzustand 60
6.1.4 Datenbankoperationen 64
6.1.5 Zusammenfassung 64
6.2 Datensatzklassen 66
6.2.1 Erzeugung von neuen Datensatzobjekten 66
6.2.2 Cache 67
6.2.3 Zusammenfassung 68
6.3 Die Generierung datenbankspezifischer Klassen 69
6.4 Verbindungen zwischen Datensatzklasse und Datenbanken 77

7 Ergebnisse **80**
7.1 Leistungen der implementierten Lösung 80
7.1.1 Höhere Abstraktionen für Inhalte der Datenbank 80
7.1.2 Implizite Erzeugung von SQL-Anfragen 82
7.1.3 Minimierung der Kommunikation mit der Datenbank 82
7.1.4 Isolation 83
7.1.5 Vergleich mit kommerziellen Lösungen 83
7.2 Praktischer Einsatz objektrelationaler Klassen 84

8 Ausblick **92**
8.1 Erweiterungsmöglichkeiten 92
8.1.1 Klassengenerierung 92
8.1.2 Evolutionäre Systementwicklung 93

8.1.3 Optimierung der Cacheverwaltung 94
8.1.4 Minimierung der Zahl und Dauer von Datenbanksperren 95
8.1.5 Handhabung von Objektbeziehungen 96
8.1.6 Mehrbenutzerfähigkeit 96
8.2 Aktuelle Entwicklungen 97
8.2.1 Objektrelationale Datenbanken 97
8.2.2 Datenbankzugriff mit Java 99
8.3 Zusammenfassung 102

Literaturverzeichnis **103**

Kapitel 1

Einleitung

Moderne Anwendungsprogramme sind in der Lage, große Datenmengen mit hoher Geschwindigkeit zu verarbeiten und über Datennetze wie das Internet einem großen Anwenderkreis verfügbar zu machen. Viele dieser Daten müssen über lange Zeiträume vorgehalten werden und dürfen nicht etwa nach dem Ende eines Anwendungsprogrammlaufs vernichtet werden. Daten, deren Lebensdauer die Laufzeit eines Anwendungsprogramms übersteigt, wie z.B. Personal- oder Verkaufsdaten eines Unternehmens oder etwa medizinische Daten, werden als *persistent* bezeichnet.
Die zur Implementierung von Anwendungsprogrammen verwendeten Entwicklungs- und Speichersysteme unterscheiden sich nicht nur hinsichtlich der Verfahren, mit denen sie Persistenz gewährleisten, sondern auch in der Art und Weise, in der sie Daten repräsentieren und nutzbar machen. In dieser Arbeit werden zwei Klassen von Systemen betrachtet, die komplexe Daten verwalten und dauerhaft speichern — relationale Datenbanken und polymorphe, persistente, objektorientierte Systeme (Abschnitt 1.1). Insbesondere hinsichtlich der Repräsentation und der Manipulation von Daten unterscheiden sich diese Systeme erheblich. Jede der beiden Systemklassen hat gegenüber der jeweils anderen spezifische Vor- und Nachteile. Abschnitt 1.2 zeigt auf, weswegen eine Integration dieser beiden Systemklassen sinnvoll erscheint.
Durch die Integration relationaler Datenbanken in persistente Objektsysteme wird der Anwendungentwickler von der Notwendigkeit befreit, sich mit zwei unterschiedlichen Daten- und Objektmodellen befassen zu müssen. Er greift über das Objektsystem transparent auf eine relationale Datenbank zu (Abschnitt 1.3).
Die einzelnen Schritte, mit denen die gewünschte Integration der Datenbanken erreicht wird, werden im Abschnitt 1.4 zusammengefaßt.

1.1 Kontext

Relationale Datenbanken werden für die zentrale Bereitstellung, langfristige Speicherung und Integritätssicherung von Datenbeständen eingesetzt. Objektorientierte Systeme erlauben Entwicklern den Entwurf und die Implementierung von Anwendungen auf einem hohen Abstraktionsniveau. Dieser Abschnitt stellt wesentliche Eigenschaften dieser Systemklassen vor.

1.1.1 Relationale Datenbanken

Eine dieser beiden Klassen von Systemen ist die der relationalen Datenbanken [Dat95]. Seit der Vorstellung des relationalen Datenmodells (vgl. [Cod70], als Überblick [LS87, Kapitel 1.4]) wurde eine Reihe von Produkten entwickelt, die die Verwaltung großer Mengen strukturierter Daten ermöglichen. Informationen zu aktuellen relationalen Datenbanksystemen sind in großer Zahl im *World Wide Web* verfügbar, z.B. [Ora, Inf, Syb, MSS, ADA, DB2].
Hauptzweck relationaler Datenbanksysteme ist die langfristige Speicherung und Verwaltung großer Datenbestände in strukturierter Form. Das Datenbanksystem sichert die Dauerhaftigkeit der Daten, überwacht die Einhaltung von Integritätsbedingungen und stellt sicher, daß auch beim gleichzeitigen Zugriff vieler Benutzer die Konsistenz des Datenbestandes erhalten bleibt.
Relationale Datenbanksysteme verwalten alle Daten in Form von Tabellen. Jede Tabelle besteht aus mehreren Spalten, wobei jede dieser Spalten Daten eines bestimmten Typs enthalten darf. Die zu speichernden Daten werden in Form von Zeilen dieser Tabellen dargestellt, wobei jede Zeile maximal einen Wert für jede Spalte der Tabelle enthalten darf. Ein Datenbankschema stellt eine endliche Menge von Tabellen dar, die zu einem gemeinsamen Anwendungszweck entworfen wurden. Ein Datenbankschema wird auf der Basis eines abstrakten Datenmodells entworfen.
Ein Standardverfahren zur Beschreibung komplexer Daten und der Beziehungen zwischen ihnen ist die Darstellung durch *Entity-Relationship*-Modelle (ER-Modelle, [Che76]). Ein ER-Modell besteht aus *Entities*, Objekten der zu modellierenden Welt, und den *Relationships*, Beziehungen zwischen den Objekten. Alle Objekte desselben Typs (z.B. Angestellte einer Firma) werden durch einen Satz von Attributen beschrieben, die sich im einfachsten Fall direkt auf Spalten einer Tabelle abbilden lassen. Beziehungen zwischen Objekten verschiedener Typen werden in relationalen Datenbanken durch sogenannte Fremdschlüssel repräsentiert. Das sind spezielle Attribute, deren Werte Objekte eines anderen Typs, d.h. einer anderen Tabelle, bezeichnen.
Das relationale Datenmodell hat sich für viele Anwendungen als hinreichend mächtig zur Verwaltung großer Datenbestände erwiesen und aktuelle Datenbanksysteme verfügen über eine große Zahl von Funktionen und Hilfsprogrammen, um den effizienten Zugriff auf diese Daten zu unterstützen. Außerdem sind viele dieser Datenbanksysteme in der Lage, ihre Datenbestände einer großen Zahl von Benutzern gleichzeitig zur Verfügung zu stellen. Die Verwendung spezieller Sperr- und Transaktionsmechanismen verhindert dabei, daß die Datenbank durch den parallelen Zugriff mehrerer Benutzer in einen inkonsistenten Zustand gerät.
Relationale Datenbanken sind auf die effiziente Verwaltung von Daten spezialisiert. Sie stellen Mechanismen zur Verfügung, diese Daten zu lesen und sie in konsistenter Weise zu verändern. Wann diese Daten gelesen werden und in welcher Weise sie verändert werden, ist Sache des jeweiligen Anwendungsprogramms. Durch die Verwendung einer relationalen Datenbank werden die Daten, auf denen eine Anwendung arbeitet, strikt von den Operationen getrennt, die auf diesen Daten stattfinden. Diese Trennung ist in zweierlei Hinsicht sinnvoll.

1. Der Anwendungsprogrammierer braucht sich nicht um die Persistenz seiner Daten zu kümmern. Diese wird durch das Datenbanksystem garantiert. Die Anwendungsdaten werden an einer zentralen Stelle gespeichert, wodurch sichergestellt wird, daß alle Anwendungen denselben, konsistenten Datenbestand nutzen können.

2. Durch die Unabhängigkeit von konkreten Anwendungsprogrammen kann die Entwicklungsarbeit für relationale Datenbanksysteme ganz auf die Optimierung der Speicher-, Zugriffs- und Synchronisationsalgorithmen konzentriert werden, durch die die Integrität, Konsistenz, Dauerhaftigkeit und jederzeitige schnelle Verfügbarkeit des Datenbestandes garantiert wird.

1.1.2 Objektorientierte Systeme

Die andere Gattung von Systemen, die in dieser Arbeit eine wichtige Rolle spielt, ist die der persistenten, objektorientierten Systeme. Anders als bei relationalen Datenbanken, die eine strikte Trennung der persistenten Daten von der Anwendungslogik implizieren, besteht bei objektorientierten Systemen eine starke Kopplung zwischen Daten und den Operationen, die auf ihnen ausgeführt werden. Objektorientierte Systeme verwalten komplexe Objekte, die nicht nur einfache Daten enthalten, sondern auch Operationen, also ein *Verhalten* kapseln.
Beziehungen zwischen verschiedenen Objekten der Welt, die in einer relationalen Datenbank mit Hilfe von zusätzlichen Tabellen oder Fremdschlüsseln repräsentiert werden (assoziativ), werden in objektorientierten Systemen direkt dargestellt (referentiell). Zu assoziativen und referentiellen Selektoren vgl. Abschnitt 1.3.2 in [LS87].
Durch Vererbung können bestehende Klassen von Objekten um neue Eigenschaften erweitert werden. Ein so spezialisiertes Objekt kann trotzdem weiterhin überall dort verwendet werden, wo Objekte der Superklasse erwartet werden. Diese Eigenschaft objektorientierter Systeme, durch die ein Objekt in "vielen Gestalten", also sowohl als Objekt einer spezialisierten Klasse als auch als Objekt einer Superklasse auftreten kann, bezeichnet man als *Polymorphismus* [CW85, S. 475-479].
Für den Entwurf und die Implementierung von Anwendungsprogrammen bieten objektorientierte Systeme eine intuitive Übertragbarkeit von Objekten der realen Welt auf Objekte einer spezifischen objektorientierten Programmiersprache, eine erhöhte Wiederverwendbarkeit von implementierten Programmteilen und eine im Vergleich zur Entwurfsphase wesentlich kürzere Implementierungsphase, da sich viele Elemente eines Entwurfsmodells direkt im objektorientierten System darstellen lassen.
Für relationale Datenbanken hingegen sind je nach der Komplexität des Datenmodells Transformationen nötig, die das Datenmodell für das Datenbanksystem handhabbar machen. Außerdem läßt sich das Verhalten eines Objekts aus der zu modellierenden Welt in einer relationalen Datenbank nicht oder nur mit großen Einschränkungen darstellen.
Während sich relationale Datenbanken bereits als Standardtechnik zur Realisierung großer, datenintensiver Informationssysteme etabliert haben, setzen sich objektorientierte Systeme erst allmählich durch. Relationale Datenbanken konnten sich unter anderem deshalb schneller durchsetzen, weil das Konzept der Trennung von Anwendungsdaten und Anwendungslogik schon aus nicht-objektorientierten Programmiersprachen bekannt ist. Programme in diesen Sprachen zerfallen in Teile, in denen Datenstrukturen definiert werden, und in andere Teile, die auf diesen Datenstrukturen Operationen ausführen. Die Auslagerung der Datenstrukturen in eine externe Datenbank hat auf die Verarbeitungslogik eines solchen Anwendungsprogramms keine Auswirkungen. Objektorientierte Programmierung verlangt vom Anwendungsentwickler eine völlig andere Sichtweise des Entwicklungsprozesses. Er manipuliert nicht länger Datenstrukturen, sondern er entwirft eine Anwendung als Menge von Objekten, die über definierte Schnittstellen miteinander kommunizieren. Ob diese Objekte dabei Daten manipulieren oder wie sie das tun, ist von sekundärer Bedeutung. Vielmehr besitzt jedes Objekt ein individu-

elles Verhalten, das allein von seinem momentanen Zustand und den Nachrichten beeinflußt wird, die es empfängt. Obwohl diese Sichtweise eine intuitive und direkte Assoziaton des Anwendungsprogramms mit Objekten und Gegebenheiten der realen Welt ermöglicht, setzt sie sich erst langsam durch, da sie mit der historisch bedingten Trennung von Algorithmen und Datenstrukturen bricht [1].
Eine bedeutende Beschleunigung in der Durchsetzung objektorientierter Technik zeichnet sich durch die Verbreitung der von der Firma SUN entwickelten objektorientierten Programmiersprache JAVA ab [GJS96]. Übersetzte Java-Programme können ohne Änderungen auf jeder Rechner- oder Betriebssystemplattform ausgeführt werden, für die eine virtuelle Java-Maschine existiert. Virtuelle Java-Maschinen, die übersetzten Java-Code ausführen können, existieren für nahezu alle kommerziell relevanten Betriebssysteme und Rechnerarchitekturen. Daher eignet sich Java für die plattformunabhängige Programmentwicklung, die für die Arbeit in heterogenen Systemen wie dem Internet ein wichtiger Produktivitätsfaktor ist.

1.2 Motivation

Anwendungsentwickler, die den Schritt zur objektorientierten Programmentwicklung vollziehen wollen, sehen sich mit dem folgenden Problem konfrontiert: Ihnen steht ein leistungsfähiges, objektorientiertes System mit mächtigen Abstraktionsmechanismen zur Verfügung, wie Vererbung, Polymorphismus und die Kapselung von Daten und Operationen.. Die Datenbestände, auf denen zukünftige Anwendungen arbeiten sollen, liegen aber in Form von relationalen Datenbanken vor. Diese enthalten oft sehr große, vielfach unternehmensweit bedeutsame Daten, die über einen langen Zeitraum gewachsen sind und die nicht einfach in ein objektorientiertes System übertragen werden können. Denn ebenso wie die Daten, sind auch die Anwendungsprogramme, die mit diesen Daten arbeiten, historisch gewachsen und sollen vielfach noch weiterverwendet werden. Außerdem ist gerade die Unabhängigkeit von konkreten Anwendungen ein wichtiger Vorteil der relationalen Datenbanken, der auch für zukünftige Anwendungsprogramme von Bedeutung ist.
D.h. der Anwendungsentwickler steht vor der Herausforderung, seine Anwendungsprogramme objektorientiert zu entwickeln, dabei aber Daten einer relationalen Datenbank zu manipulieren. Dabei dürfen die Investitionen in das Datenbankschema und die darauf basierenden historisch gewachsenen Anwendungen nicht gefährdet werden, z.B. durch Änderungen oder Ergänzungen des Datenbankschemas. Selbst wenn das zugrundeliegende Datenmodell speziell für die konkrete Anwendung neu entworfen wird, kann es erforderlich sein, daß die Daten durch Hilfsprogramme der relationalen Datenbank manipulierbar und für den Benutzer der Datenbank verständlich bleiben.

1.3 Ziel der Arbeit

Für einen Anwendungsentwickler ergeben sich aus heutiger Sicht zwei wichtige Anforderungen an seine Programme:

1. Die Anwendung soll objektorientiert entworfen und implementiert werden.
2. Existierende Datenbestände sollen von der Anwendung weiterverwendet werden.

[1] Einen Vergleich der objektorientierten Sichtweise mit anderen Methoden des Programmentwurfs bietet z.B. das Kapitel 2.1 in [Boo94]

Liegen die bereits vorhandenen Anwendungsdaten in Form einer relationalen Datenbank vor, so steht man vor einem sogenannten *Impedance Mismatch*, d.h. die Konzepte der relationalen Theorie lassen sich nur bedingt auf Konzepte der Anwendungsprogrammiersprache abbilden [Cat94, S.122 ff]. Viele Vorteile der Objektorientierung gehen bei der Verwendung einer relationalen Datenbank verloren, weil es für einige wichtige objektorientierte Konzepte keine Entsprechung in der relationalen Theorie gibt.
Ziel dieser Arbeit ist es daher, am Beispiel eines konkreten objektorientierten Entwicklungssystems zu zeigen, wie sich Daten aus relationalen Datenbanken in der objektorientierten Welt weiterverwenden lassen und wie sich die Aussagekraft der relationalen Daten sogar vergrößern läßt. Beziehungen zwischen Objekten in der modellierten Welt, die in der Datenbank hinter Fremdschlüsseln und Hilfstabellen verschwinden, sollen auf intuitiv zu erfassende Objektreferenzen abgebildet werden (*Semantic Key Swizzling*, vgl. [KJA93, S. 526]). Änderungen an einzelnen Objekten sollen für den Anwender möglich sein, ohne daß ihm die Existenz der relationalen Datenbank bewußt sein muß, in der letztlich alle Daten gespeichert werden.

1.4 Aufbau der Arbeit

Am Anfang dieser Arbeit steht eine Bestandsaufnahme der bereits existierenden Datenbankschnittstellen des verwendeten objektorientierten Systems. Die wesentlichen Eigenschaften dieser Schnittstellen werden kurz vorgestellt und an Beispielen demonstriert. Anhand dieser Beispiele wird deutlich, daß ein Datenbankzugriff bereits mit den vorhandenen Mitteln möglich ist, daß diese Zugriffsmethode aber noch nicht den Ansprüchen genügt, die an eine komfortable Benutzung einer Datenbank gestellt werden. Welche Anforderungen wir im Rahmen dieser Arbeit an eine leistungsfähige objektorientierte Datenbankschnittstelle stellen, ist Thema des Kapitels 3.
Aus der Menge der verfügbaren kommerziellen Werkzeuge, die eine Verbindung von relationaler mit objektorientierter Technik ermöglichen, werden einige typische Vertreter ausgewählt. Im Kapitel 4 wird untersucht, inwiefern diese Werkzeuge die in Kapitel 3 gestellten Anforderungen erfüllen und mit welchen Mitteln dies erreicht wird.
Auf der Grundlage der in Kapitel 3 formulierten Anforderungen und den in Kapitel 4 untersuchten Lösungen wird dann eine Klassenbibliothek für ein konkretes objektorientiertes System entworfen, für das noch keine vergleichbare Lösung existiert (Kapitel 5). Den Schwerpunkt dieses Kapitels bilden diejenigen Klassen und Methoden, die direkt zur Programmierung großer Datenbankanwendungen eingesetzt werden.
Der so entstandene Entwurf wird in eine Implementierung umgesetzt, deren Details in Kapitel 6 näher erläutert werden. Das verwendete objektorientierte System unterstützt parametrischen Polymorphismus, was die Implementierung und die Nutzung datenbankspezifischer Klassen zum Teil erheblich erleichtert. Typumwandlungen zur Laufzeit, wie in anderen objektorientierten Sprachen (z.B. Java [GJS96]) werden dadurch überflüssig. Typfehler können bereits bei der Übersetzung eines Programms erkannt werden. Das Persistenzkonzept des objektorientierten Systems erlaubt ferner die Sicherung des Systemzustandes zu beliebigen Zeitpunkten. Bei einem erneuten Start des Systems zu einem späteren Zeitpunkt kann der gespeicherte Zustand wiederhergestellt werden. In Kapitel 7 wird überprüft, inwieweit die entstandene Klassenbibliothek die anfangs geforderten Eigenschaften aufweist und welche Problemstellungen und Fragen noch offengeblieben sind. Die Arbeit schließt mit einem Ausblick auf aktuelle und zukünftige Entwicklungen auf diesem Forschungsgebiet (Kapitel 8).

Kapitel 2

Programmierschnittstellen für relationale Datenbanken

Sollen in einem Anwendungsprogramm Daten verarbeitet werden, die in relationalen Datenbanken gespeichert sind, so ist die Existenz geeigneter Programmierschnittstellen (*Application Programming Interfaces*, APIs) notwendig, mit deren Hilfe Daten mit dem Datenbanksystem ausgetauscht werden können. Zu den meisten existierenden relationalen Datenbanksystemen gibt es solche Schnittstellen für eine Vielzahl von Programmiersprachen. In den folgenden Abschnitten werden wesentliche Eigenschaften dieser Schnittstellen vorgestellt und ihre Entwicklungsgeschichte bis zu den heute verfügbaren Schnittstellen für objektorientierte Programmiersprachen nachvollzogen.

- Abschnitt 2.1 erläutert den prinzipiellen Ablauf der Kommunikation zwischen Anwendungsprogramm und relationaler Datenbank unabhängig von einer konkreten Programmiersprache.
- Die Eigenschaften einer objektorientierten Schnittstelle für den Datenbankzugriff werden im Abschnitt 2.2 näher erläutert. Als Beispiel wird die TYCOON-SQL-Schnittstelle vorgestellt, welche die für diese Arbeit erforderliche Basisfunktionalität bereitstellt.
- Abschnitt 2.3 schließlich zeigt auf, welche Einschränkungen für den objektorientierten Entwickler durch den Zugriff auf eine relationale Datenbank entstehen. Diese sind durch die strukturellen Unterschiede zwischen dem relationalen und dem objektorientierten Modell begründet.

2.1 Grundprinzipien

Zwei wesentliche Fragen stellen sich bei jeder Art von Kommunikation zwischen Anwendungsprogramm und Datenbank

- Wie wird dem Datenbanksystem mitgeteilt, welche Daten verarbeitet werden sollen?
- Wie werden die Daten selbst mit der Datenbank ausgetauscht?

Das erste dieser beiden Probleme wird durch alle gängigen APIs für relationale Datenbanken in derselben Weise gelöst. Sämtliche Operationen auf den Daten einer relationalen Daten-

bank werden mit Hilfe der standardisierten Anfragesprache SQL (*Structured Query Language*, [DD97]) abgewickelt. Das bedeutet, daß alle Programmierschnittstellen für relationale Datenbanken, so unterschiedlich sie auch sein mögen, letztendlich diese spezielle Sprache in irgendeiner Form unterstützen müssen. Jede dieser Programmierschnittstellen, verfügt daher über eine Möglichkeit direkt oder indirekt SQL-Befehle an das Datenbanksystem zu versenden.
Das zweite Problem, der Austausch der eigentlich interessierenden Daten, wird folgendermaßen gelöst: Das Anwendungsprogramm reserviert Datenstrukturen, die für die Aufnahme der Daten aus der Datenbank geeignet sein müssen. Die Orte, an denen diese Datenstrukturen im Speicher des Rechners liegen, auf dem das Anwendungsprogramm abgearbeitet wird, werden der Programmierschnittstelle des Datenbanksystems bekannt gegeben. Werden nun Daten aus der Datenbank oder in die Datenbank übertragen, so werden dafür diese Speicherbereiche benutzt, die zwischen Anwendung und Programmierschnittstelle vereinbart worden sind. Die beiden wichtigsten praktischen Lösungen für diese Aufgaben [1], *eingebettetes SQL* (Abschnitt 2.1.1) und *dynamisches SQL* (Abschnitt 2.1.2) werden im folgenden kurz vorgestellt. Die Diskrepanzen, die zwischen den Zugriffsmechanismen der Datenbank und denen der Programmiersprache bestehen, sowie den typischen Ansatz zum Umgang mit diesen Unterschieden, behandelt Abschnitt 2.1.3.

2.1.1 Eingebettetes SQL

Bei diesem Ansatz werden SQL-Befehle direkt in den Programmtext des Anwendungsprogramms eingebettet. D.h. der Anwendungsprogrammierer kann SQL-Befehle in seinem Programm verwenden, als wären sie Teil der Programmiersprache (z.B. C), in der die Anwendung entwickelt wird. Da der Compiler der jeweiligen Programmiersprache nicht mit diesem eingebetteten SQL-Code umgehen kann, ist vor der Übersetzung des Anwendungsprogramms ein zusätzlicher Verarbeitungsschritt nötig. Ein spezieller Präprozessor übersetzt in diesem Schritt den eingebetteten SQL-Code in Aufrufe von Funktionen, die in Funktionsbibliotheken des Datenbanksystems enthalten sind. Das Ergebnis dieses Verarbeitungsschritts ist ein Programmtext, der ausschließlich Befehle und Funktionen der Anwendungsprogrammiersprache enthält und sich damit vom Compiler übersetzen läßt.
Die Verwendung dieses Ansatzes ist insbesondere dann vorteilhaft, wenn die im Anwendungsprogramm verwendeten SQL-Befehle bereits zum Zeitpunkt der Programmübersetzung bekannt sind und sich ihre Struktur während des Programmablaufs nicht ändert. Für die Programmierung von Anwendungen, bei denen die Struktur eines SQL-Befehls erst zur Laufzeit des Programms festgelegt wird, ist dieser Ansatz nicht geeignet. Ein Beispiel für letzteres sind alle Anwendungen, die einem Benutzer interaktiven, wahlfreien Zugriff auf beliebige Datenbankinhalte ermöglichen.

2.1.2 Dynamisches SQL

Steht zum Zeitpunkt der Übersetzung eines Anwendungsprogramms noch nicht fest, welche SQL-Befehle zur Laufzeit ausgeführt werden müssen oder ist die Benutzung eines Präprozessors für eingebettetes SQL nicht möglich oder erwünscht, so ist ein Zugriff auf eine Datenbank über sogenanntes *dynamisches SQL* nötig.
Bei diesem Ansatz verwendet der Anwendungsprogrammierer Funktionen, die ihm in Form

[1] Teile des Abschnitts 2.1 werden voraussichtlich in [MSht] erscheinen

einer Bibliothek des Datenbankherstellers zur Verfügung gestellt werden. Die standardisierte Datenbankschnittstelle ODBC (*Open Database Connectivity*, [Gei96]) ist als eine solche Bibliothek implementiert und wird von nahezu jedem kommerziellen Datenbankhersteller angeboten. In der Literatur werden solche Schnittstellen auch als *Call Interfaces* oder *Call Level Interfaces* bezeichnet (vgl. [Sip96]). Mit Hilfe ihrer Funktionen werden SQL-Befehle als Zeichenketten (Strings) an die Datenbank gesendet und ausgeführt (vgl. in Abbildung 2.1 die Funktionen `SQLPrepare`, `SQLExecute` und `SQLExecDirect`).

Die Ergebnisse einer SQL-Anfrage erhält die Anwendung, indem sie im Speicher des Rech-

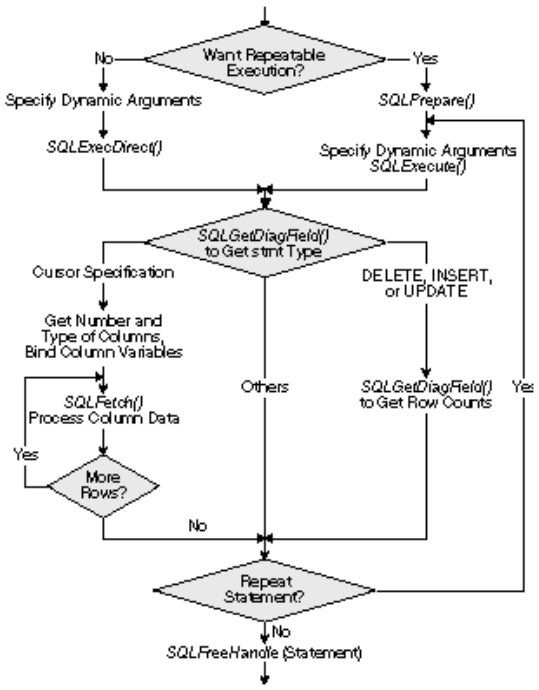

Abbildung 2.1: Ausführung von SQL-Befehlen über ein *Call Level Interface* (aus [Sip96])

ners Bereiche reserviert, in denen das Datenbanksystem die Ergebnisse der Anfrage ablegen kann. Den genauen Ort dieser Speicherbereiche teilt die Anwendung dem Datenbanksystem mit, indem sie Zeiger auf diese Speicherbereiche an eigens dafür vorgesehene Funktionen des *Call Level Interface* übergibt (in Abbildung 2.1 der Punkt *Bind Column Variables*).
Dieser Ansatz erlaubt im Vergleich zu eingebettetem SQL die Erstellung flexiblerer Anwendungen. Allerdings wird der Anwendungsprogrammierer wesentlich stärker als bei der Verwendung von eingebettetem SQL mit den Problemen der dynamischen Speicherverwaltung belastet.

2.1.3 Mengenwertiger Zugriff versus tupelorientierter Zugriff

Die oben beschriebenen Schnittstellen zur Programmierung von Datenbankanwendungen stellen nicht nur Funktionalität zur Abarbeitung von einfachen SQL-Befehlen zur Verfügung, sie müssen auch mit dem sogenannten *Impedance Mismatch* [Cat94, S.122 ff] zwischen dem Datenmodell des Datenbanksystems und dem Datenmodell der Anwendungsprogrammiersprache

zurechtkommen. Der in diesem Kontext relevante Unterschied - der *Mismatch* - zwischen den beiden Modellen besteht in der Handhabung großer Datenmengen.
Während die meisten Programmiersprachen Mechanismen anbieten, mit denen einzelne Datensätze direkt verändert werden können, besitzen sie kein Sprachkonstrukt für Relationen und deren Manipulation. Relationale Datenbanken hingegen operieren immer auf Mengen von Datensätzen, die durch eine deklarative Sprache (SQL) beschrieben werden. Operationen auf einzelnen Datensätzen sind nur indirekt möglich, indem man eine Menge so weit einschränkt, daß sie nur noch einen Datensatz enthält.
Um die Kluft zwischen diesen beiden Datenmodellen zu überbrücken, bedient man sich des Konzepts des *Cursors*. Ein Cursor ist eine Art Zeiger, der eine Relation sequentiell durchläuft und jeweils Verweise auf genau einen Datensatz dieser Relation liefert. Schickt ein Anwendungsprogramm also eine SQL-Anfrage an ein Datenbanksystem, so bestimmt das Datenbanksystem die Ergebnismenge für diese Anfrage und öffnet einen Cursor, der auf das erste Element der Ergebnismenge verweist. Das Anwendungsprogramm kann nun den Datensatz lesen und ihn weiterverarbeiten. Um die nächsten Elemente der Ergebnismenge zu erhalten, schaltet das Anwendungsprogramm den Cursor weiter. Die Sequenz "Weiterschalten des Cursors" und "Lesen des aktuellen Datensatzes" wird so lange wiederholt, bis die vollständige Ergebnismenge abgearbeitet ist (vgl. in Abbildung 2.1 die mit dem Befehl `SQLFetch` beginnende Schleife). Ein Cursor läuft in der Regel nur vorwärts. Ein Rücksprung zu den bereits abgearbeiteten Datensätzen ist nicht vorgesehen. Für einen erneuten Durchlauf durch eine Menge von Datensätzen muß die zugehörige SQL-Anfrage ein weiteres Mal ausgeführt werden, wodurch ein neuer Cursor erzeugt wird.
In einigen Programmiersprachen existieren Schnittstellen, die dieses Cursor-Konzept auf allgemeinere Konzepte wie z.B. Ströme (*Streams* oder *Reader*) abbilden. Dadurch erscheint dem Programmierer die Datenbank als "Datenspeicher" mit ähnlichen Zugriffsmethoden, wie er sie auch von gewöhnlichen Dateien oder Ein-/Ausgabeströmen kennt (vgl. z.B. die in Kapitel 4 vorgestellten DBTools.h++).

2.2 Programmierschnittstellen für objektorientierte Systeme

Die im vorigen Abschnitt beschriebenen Programmierschnittstellen für relationale Datenbanken wurden zuerst für imperative Programmiersprachen, wie COBOL oder C, entwickelt. Die Verbreitung objektorientierter Programmiersprachen setzte erst später ein. Die ersten Datenbank-APIs für objektorientierte Systeme basierten daher und basieren zum Großteil auch noch auf den APIs für nicht-objektorientierte Sprachen.
Als Beispiel für eine solche Schnittstelle soll im folgenden das Tycoon-SQL-Interface [Sku98] dienen, das Benutzern des objektorientierten Tycoon-Systems den Zugriff auf relationale Datenbanken unter Rückgriff auf Funktionsbibliotheken des Datenbankherstellers gestattet. Die Tycoon-Schnittstelle zur Nutzung relationaler Datenbanken integriert die gemeinsame Funktionalität der beiden weit verbreiteten Standard-APIs ODBC und OCI. ODBC (Open Data Base Connectivity, [Gei96]) ist ein von der Firma Microsoft entscheidend geprägter Standard für den Austausch von Daten mit relationalen Datenbanken. ODBC stellt die erste kommerziell verfügbare Implementierung des *Call Level Interface*-Standards der SQL Access Group dar [Sip96]. Eine große Zahl von Datenbankherstellern unterstützt diesen Standard durch die Lieferung von Bibliotheken, die die Funktionen dieser einheitlichen Schnittstelle implementieren. Das Oracle Call Interface (OCI, [OCI97]) ist eine Bibliothek der Firma

ORACLE, die einen vergleichbaren Funktionsumfang aufweist und wegen der marktbeherrschenden Stellung der Firma ORACLE als Hersteller relationaler Datenbanksysteme gesondert berücksichtigt wurde.

Zugriff auf externe Funktionen

Die Basis für die TYCOON-Schnittstelle zu relationalen Datenbanken ist die Fähigkeit des TYCOON-Systems, Funktionsbibliotheken fremder Herkunft über spezielle Objekte zugänglich zu machen. Im TYCOON-System wird dazu ein Objekt erzeugt, dessen Methoden den Funktionen der externen Bibliothek entsprechen. Bei jedem Aufruf einer solchen Methode, wird die entsprechende Funktion der externen Bibliothek ausgeführt. Da die externe Funktionsbibliothek nicht für die Zusammenarbeit mit TYCOON konzipiert und erstellt wurde, sind bereits an dieser Stelle einige Probleme zu lösen.
So werden die Parameter der externen Funktionen mit Hilfe von Typen beschrieben, die in der Implementierungssprache der Bibliothek bekannt sind. Diese Typen müssen in den Methoden des korrespondierenden TYCOON-Objekts durch vergleichbare TYCOON-Typen ersetzt werden.
Ein weiteres Problem stellt sich bei Funktionen, die nicht mit Wertparametern, sondern Variablenparametern (*Call-By-Reference*) arbeiten. Da TYCOON über kein Konzept für die Behandlung solcher Parameter verfügt, müssen diese künstlich erzeugt werden. Dazu wird im Hauptspeicher des Rechners dynamisch Speicherplatz angefordert, der groß genug ist, um den Parameterwert aufzunehmen. Ein Zeiger auf diesen Speicherbereich wird durch einen einfachen TYCOON-Datentyp (`Int`) repräsentiert und kann dann direkt an die Funktionen der externen Bibliothek übergeben werden.
Die wichtigste Leistung der TYCOON-SQL-Schnittstelle besteht nun darin, dafür zu sorgen, daß die externen Datenbankfunktionen ausschließlich mit zulässigen Parametern aufgerufen werden und das System nicht durch ungültige Parameterwerte oder Zeiger in einen unsicheren Zustand gerät. Darüberhinaus werden die gemeinsamen Konzepte der beiden Schnittstellen ODBC und OCI auf TYCOON-Objekte abgebildet, die die Komplexität der externen Funktionen vor dem Anwendungsprogrammierer verbergen und einen direkten Zugriff auf die externen Funktionen überflüssig machen.

Auf- und Abbau einer Datenbank-Verbindung

Für die Herstellung einer Datenbankverbindung ist der `SQLDriverManager` zuständig. Dieses Objekt verwaltet die im konkreten Objektspeicher verfügbaren "Treiber", die zur Herstellung von Datenbankverbindungen genutzt werden können. Wird die `connect`-Methode des `SQLDriverManager` aufgerufen, so bestimmt dieser anhand der übergebenen Parameter einen Treiber, der eine entsprechende Datenbankverbindung aufbauen kann und reicht die für den Verbindungsaufbau relevanten Daten an diesen weiter. Momentan sind Treiber für jeweils eine Implementierung des ORACLE CALL INTERFACE und des ODBC-Standards verfügbar.
Der Treiber versucht nun, eine Verbindung zur relationalen Datenbank herzustellen. Dazu benötigt er Daten über den Ort bzw. den Namen der Datenbank, den Namen des Datenbankbenutzers und dessen Datenbankpaßwort. Diese Daten werden in Form einer `URL` (*Universal Resource Locator*), wie sie im *WWW* üblich ist, an den Treiber übergeben. Ist der Verbindungsaufbau erfolgreich, so liefert der Treiber ein Objekt der Klasse `SQLConnection`. Neben Methoden zum Öffnen und Schließen der Datenbankverbindung verfügt eine `SQLConnection`

auch über Methoden, um das Transaktionsverhalten der Datenbankverbindung zu beeinflussen. So kann hier eingestellt werden, ob eine Transaktion explizit durch Aufruf entsprechender `commit`- oder `rollback`-Methoden beendet wird oder ob jede Ausführung eines SQL-Befehls eine eigene Transaktion darstellt.

Ausführen von einfachen SQL-Befehlen

Zum Absetzen von SQL-Befehlen dient eine eigene Klasse `SQLStatement`, die einen umfangreichen Satz der dafür erforderlichen Methoden bereitstellt. Objekte dieser Klasse werden durch Aufruf der `newStatement`-Methode von Objekten der Klasse `SQLConnection` erzeugt. Ein SQL-Befehl wird dann als Zeichenkette an eine der `execute`-Methoden übergeben. Handelt es sich dabei um einen Befehl der *Data Definition Language* (DDL) oder der *Data Manipulation Language* (DML), beides Teilmengen von SQL, so wird er unmittelbar ausgeführt und die Zahl der betroffenen Tabellenzeilen bestimmt. DDL-Befehle dienen zum Erzeugen, Ändern und Löschen neuer Datenbanktabellen und *Views* (Sichten), DML-Befehle dienen zum Einfügen, Ändern und Löschen von Einträgen in Tabellen. Enthält der SQL-Befehl eine Anfrage, also einen SQL-Befehl, der keine Daten verändert, sondern eine Menge von Daten aus Tabellen liest, so wird ein Objekt der Klasse `SQLCursor` erzeugt. Ein `SQLCursor` entspricht in etwa einem Zeiger, der immer auf genau ein Element der Ergebnismenge einer Anfrage zeigt. Mit seiner `next`-Methode, läßt sich der `SQLCursor` zum jeweils nächsten Datensatz fortschalten. Über seine `get`-Methoden lassen sich die Werte des aktuellen Datensatzes ausgeben. Dazu muß als Parameter die Nummer der interessierenden Spalte angegeben werden. Ein `SQLCursor` steht in einer 1:1-Beziehung zu dem `SQLStatement`, durch das er erzeugt wurde. Sobald über das `SQLStatement`-Objekt ein neuer SQL-Befehl abgesetzt wird, wird der alte `SQLCursor` ungültig und es wird ggf. ein neuer `SQLCursor` erzeugt. Alle `SQLStatements` werden bei ihrer Erzeugung von dem erzeugenden `SQLConnection`-Objekt registriert. So wird sichergestellt, daß die von ihnen belegten Ressourcen, wie dynamisch angeforderter Speicher, beim Schließen einer Datenbankverbindung wieder freigegeben werden.

Wiederverwendbare SQL-Befehle

Solange einem `SQLStatement`-Objekt kein neuer SQL-Befehl übergeben wird, läßt sich der alte SQL-Befehl beliebig oft wiederholen, vorausgesetzt das führt nicht zur Verletzung von Integritätsbedingungen innerhalb der Datenbank (z.B. doppelte Primärschlüsselwerte). Oftmals soll ein SQL-Befehl aber nicht in identischer, sondern nur in ähnlicher Form wiederholt werden. D.h. die Operation (z.B. Einfügen einer Zeile) bleibt die gleiche und auch die Tabelle, auf die sich die Operation bezieht, bleibt gleich. Lediglich die Parameter des SQL-Befehls ändern sich. Parameter sind in diesem Zusammenhang die konkreten Tabellenwerte, die durch den SQL-Befehl manipuliert werden. `SQLStatement` erlaubt es, einen SQL-Befehl so vorzubereiten, daß er nur einmal an die Datenbank gesendet werden muß und bei folgenden Aufrufen nur noch die aktuellen Werte der Parameter mit der Datenbank ausgetauscht werden müssen. Hierfür muß der Datenbank mitgeteilt werden, welchen Namen und welchen Typ die Parameterwerte haben sollen. Die Abbildung der Parameterwerte und ihrer TYCOON-Typen auf Datenstrukturen und Typen, die die Funktionsbibliothek des gerade verwendeten Treibers versteht, übernimmt `SQLStatement` soweit wie möglich. Tabelle 2.1 zeigt, zu welchen SQL-Datentypen (SQL-92,

Beschreibung	SQL-Typen	TYCOON-Typ
ganze Zahlen	INTEGER,SMALLINT, NUMERIC(p,0),DECIMAL(p,0)	Int
reelle Zahlen	REAL, DOUBLE PRECISION, FLOAT(q), NUMERIC(p,q), DECIMAL(p,q); $q > 0$	Real
Datums- und Zeitangaben	TIME(q), DATE, TIMESTAMP(q)	Date
Zeichenketten	CHAR(n), VARCHAR(n)	String
BLOBs	LONG RAW,LONG BYTE	File

Die Parameter der SQL-Typen haben folgende Bedeutung:

p – Maximale Anzahl von Stellen der Zahlendarstellung numerischer Werte

q – Maximale Zahl von Nachkommastellen numerischer Werte bzw. der Sekundenangabe bei Zeitangaben

n – Maximale Länge von Zeichenketten

Tabelle 2.1: Abbildung von SQL-Typen auf TYCOON-Typen

vgl. [MU97, S.121 ff]) derzeit eine Abbildung auf TYCOON-Typen existiert [2]. Die aktuellen Parameterwerte werden mit den `set`-Methoden des `SQLStatement`s festgelegt. Zu jedem der zulässigen TYCOON-Parametertypen existiert eine eigene `set`-Methode. Stimmt der Typ des Parameterwerts nicht mit dem zuvor definierten Typ des Parameters überein, wird die Zuweisung mit einer Fehlermeldung (in TYCOON eine `Exception`) abgebrochen. War das Setzen der Parameter erfolgreich, kann der so vorbereitete SQL-Befehl ausgeführt werden.

Persistenz

Wie die meisten Objekte innerhalb eines TYCOON-Objektspeichers können auch Datenbankverbindungen und die von ihnen abhängenden Objekte persistent gespeichert werden, so daß sie bei einem Neustart des Objektspeichers zu einem späteren Zeitpunkt scheinbar unverändert wieder verfügbar sind. Um dieses Verhalten zu gewährleisten, ist ein nicht unerheblicher Verwaltungsaufwand nötig. So muß bei einem Neustart zunächst die Datenbankverbindung wieder hergestellt werden, die beim Sichern des Objektspeichers geöffnet war. Alle über diese Datenbankverbindung erzegten `SQLStatement`s werden reinitialisiert, so daß sie sofort nach dem Neustart weiterverwendet werden können, der letzte abgesetzte oder vorbereitete SQL-Befehl bleibt dabei erhalten. Einzig die erzeugten `SQLCursor` können nach einem Neustart nicht weiterverwendet werden. Das ist deshalb nicht wünschenswert, weil sich die Daten innerhalb der verwendeten Datenbanktabellen zwischen dem Sichern des Objektspeichers und seinem Neustart erheblich ändern können. Somit ist es sehr wahrscheinlich, daß

[2] Die Abbildung auf TYCOON-Typen wurde so gewählt, daß der Wertebereich des SQL-Typs innerhalb des Wertebereichs des korrespondierenden TYCOON-Typs liegt. Übersteigt der Wertebereich eines SQL-Ganzzahltypen den Wertebereich des TYCOON-Typs `Int`, so wird der SQL-Typ auf den TYCOON-Typ `Real` abgebildet. Für die Verwaltung binärer Daten beliebiger Länge (Binary Large Objects, BLOBs) existiert kein standardisierter SQL-Typ. Tabelle 2.1 zeigt dort beispielhaft die Datentypen, die von den Datenbanksystemen ORACLE [Ora] und ADABAS D [ADA] zur Speicherung von BLOBs verwendet werden.

ein `SQLCursor`, würde man dieselbe SQL-Anfrage nach dem Neustart noch einmal ausführen, andere Ergebnisse liefern würde als zum Zeitpunkt der Systemsicherung. Um also zu verhindern, daß nach einem Neustart mit veralteten Anfrageergebnissen weitergearbeitet wird, wird der Zustand von `SQLCursor`-Objekten nicht gespeichert.

2.3 Bewertung

Schnittstellen, wie die vorigen Abschnitt vorgestellte TYCOON-SQL-Schnittstelle, haben vor dem Hintergrund eines Objektsystems mit Fähigkeiten wie Persistenz, Vererbung, Polymorphismus und Funktionen höherer Ordnung einige schwerwiegende Nachteile.

Niedriger Abstraktionsgrad

Datenbanken verwalten Tabellen. Tabellen enthalten Datensätze. Durch das Konzept des Cursors werden die Datenmengen in einzeln zu behandelnde Datensätze zerlegt. Eine einheitliche Behandlung einer größeren Menge von Datensätzen muß mit den Mitteln der Anwendungsprogrammiersprache explizit realisiert werden. Die Aufspaltung reicht noch weiter. Eine in der Datenbank als Einheit zu erkennende Tabellenzeile wird auf Attribute eines Cursors reduziert, die nicht als Einheit, sondern nur noch einzeln über separate Lesemethoden des `SQLCursor`s zugreifbar sind. Beziehungen zwischen Tabellen des Datenbankschemas, wie sie durch die Definition von Primär- und Fremdschlüsselattributen darstellbar sind, werden von den vorgestellten Schnittstellen nicht abgebildet.

Erfordernis von SQL-Sprachkenntnissen

Da der Zugriff auf relationale Datenbanken aus Anwendungsprogrammen bei den vorgestellten APIs unvermeidlich über die Sprache SQL führt, muß der Anwendungsentwickler nicht nur die Programmiersprache der Anwendung beherrschen, sondern auch noch die Anfragesprache der Datenbank.

Kommunikationsbedarf mit der Datenbank

Für jede Änderung innerhalb der Datenbank müssen SQL-Befehle oder Teile davon mit der Datenbank ausgetauscht werden. Werden Daten also häufig bearbeitet, führt das zu einem hohen Kommunikationsaufkommen zwischen Anwendung und Datenbanksystem. Dadurch wird der Kommunikationsweg zwischen Anwendungssystem und Datenbankserver belastet, der meistens mit anderen Benutzern geteilt werden muß. Durch Wartezeiten, die sich aus hohem Verkehrsaufkommen in Kommunikationsnetzen ergeben, sinkt die Verarbeitungsgeschwindigkeit des Anwendungsprogramms und damit die Akzeptanz beim Anwender. Es liegt also im Interesse des Anwendungsprogrammierers, die Kommunikation mit dem Datenbanksystem zu minimieren.

Kapitel 3

Anforderungen an den objektorientierten Datenbankzugriff

Aus den im Kapitel 2 genannten Nachteilen einfacher Datenbankschnittstellen ergeben sich direkt die Anforderungen an höher entwickelte Schnittstellen. Diese werden in den folgenden Abschnitten näher erläutert.

- Kapselung der Datenbankstrukturen durch objektorientierte Abstraktionen (Abschnitt 3.1).
- Implizite Erzeugung von SQL. Datenbankzugriffe sollen ermöglicht werden, ohne daß der Anwendungsprogrammierer über tiefergehende Kenntnis der Sprache SQL verfügen muß (Abschnitt 3.2).
- Minimierung der Kommunikation mit der Datenbank (Abschnitt 3.3).
- Grundlegende transaktionale Eigenschaften relationaler Datenbanken, insbesondere die Isolation von Datenbankoperationen gegen parallele Datenmanipulationen durch andere Datenbankbenutzer, sollen erhalten bleiben (Abschnitt 3.4).

3.1 Höhere Abstraktionen für Inhalte der Datenbank

Typische Elemente relationaler Datenbanken müssen auch im objektorientierten System repräsentiert werden, bevor sich höhere Abstraktionen auf dieser Repräsentation aufsetzen lassen. Dazu gehören

- Tabellen,
- Datensätze und
- Integritätsbedingungen.

Ein Datensatz entspricht dabei einem Eintrag, also einer Zeile, einer Tabelle. Datensätze können neu erzeugt oder aus der Tabelle gelesen, geändert und auch aus der Tabelle gelöscht werden. Unter dem Begriff Integritätsbedingungen sind in diesem Zusammenhang Bedingungen zu verstehen, die die Wahl bestimmter Attributwerte einschränken. Diese Bedingungen

werden innerhalb der relationalen Datenbank in einem eigenen Datenbankschema, dem sogenannten *Data Dictionary* gespeichert. Das Datenbanksystem überwacht die Einhaltung dieser Integritätsbedingungen. Im Kontext dieser Arbeit sind zwei Arten von Integritätsbedingungen von zentraler Bedeutung:

1. **Primärschlüsselintegrität**
   Der Primärschlüssel einer Tabelle besteht aus einem oder mehreren Attributen der Tabelle und muß für alle Einträge in der Tabelle eindeutig sein, d.h. keine zwei Zeilen einer Tabelle dürfen dieselben Primärschlüsselwerte aufweisen. Der Primärschlüssel dient also der eindeutigen Identifizierung eines Tabelleneintrags. Die Existenz dieses Schlüssels wird sich bei der Abbildung von Tabelleneinträgen auf Objekte als notwendig erweisen. Nur so läßt sich garantieren, daß sich Objekte im objektorientierten System eindeutig Einträgen in Datenbanktabellen zuordnen lassen.

2. **Referentielle Integritätsbedingungen**
   Bei der Implementierung eines Datenmodells in einer relationalen Datenbank werden Beziehungen zwischen Objekten des Datenmodells auf Fremdschlüssel abgebildet. Ein Fremdschlüssel ist dabei ein Attribut oder eine Menge von Attributen, deren Werte einen Eintrag in einer anderen Tabelle identifizieren. Ein Fremdschlüssel verweist deshalb in der Regel auf den Primärschlüssel der referenzierten Tabelle. Eine referentielle Integritätsbedingung beschreibt explizit diese Beziehung zwischen zwei Tabellen. Eine solche Bedingung verlangt, daß ein Fremdschlüssel einer Tabelle nur auf Einträge der referenzierten Tabelle verweisen darf, die tatsächlich existieren.
   Enthalte beispielsweise eine Tabelle Daten über Angestellte eines Unternehmens, darunter auch ein Attribut, das die Abteilung festlegt, in der ein Angestellter arbeitet. Die Daten über Abteilungen sind in einer anderen Tabelle gepeichert, auf die sich die Angestelltentabelle bezieht. Eine referentielle Integritätsbedingung für dieses Beispiel würde lauten “Ein Angestellter darf nur in einer Abteilung arbeiten, die auch in der Firma vorhanden ist”. Wenn also das Abteilungsattribut eines neuen Angestellten auf eine Abteilung verweist, für die in der Abteilungstabelle kein korrespondierender Eintrag besteht, so weist das Datenbanksystem die Einfügung des Datensatzes zurück.

Durch Auswertung von Metadaten, also Daten über die Struktur und die Abhängigkeiten zwischen den einzelnen Tabellen, können Informationen über Beziehungen gewonnen werden, die zwischen Objekten eines Datenmodells bestehen. Diese Beziehungen lassen sich in einem objektorientierten System direkt darstellen. Das relationale Datenbanksystem benötigt dazu die oben beschriebenen Fremdschlüsselattribute. Ist also die Abbildung einzelner Datensätze auf Objekte gelungen, können im nächsten Schritt zusätzlich die gefundenen Beziehungen zwischen Tabellen in die zugehörigen Objekte integriert werden. Für den objektorientierten Programmierer wird damit eine höhere Abstraktion erreicht, weil er sich nicht mehr um das manuelle Setzen von Fremdschlüsselattributen zu kümmern braucht. Eine Beziehung wird direkt durch Angabe der an ihr beteiligten Objekte dargestellt. Das Setzen der nötigen Fremdschlüsselwerte in den zugrundeliegenden Datenbanktabellen sollte dann die Datenbankschnittstelle übernehmen.

3.2 Implizite Erzeugung von SQL-Code

Der Anwendungsprogrammierer soll sich nicht mit der Formulierung von SQL-Anfragen beschäftigen müssen, sondern die Objekte, welche den Inhalt der Datenbank repräsentieren, sollen selber den nötigen SQL-Code generieren, der die entsprechenden Datenbankoperationen veranlaßt. Typische Operationen auf Datensätzen, wie das Erzeugen, Ändern oder Löschen, erfordern auch typische SQL-Befehle. Da sich mit Hilfe von Metadaten die Struktur einer Datenbanktabelle ermitteln läßt, lassen sich auch im voraus standardisierte SQL-Befehle generieren, die neue Zeilen einfügen oder bestehende Zeilen lesen oder verändern. SQL-Anfragen liefern in der Regel nicht nur einen, sondern gleich eine größere Menge von Datensätzen. Deshalb müssen im objektorientierten System Möglichkeiten geschaffen werden, solche Ergebnismengen zu handhaben.

3.3 Minimierung der Kommunikation mit der Datenbank

Nicht jede Veränderung eines Datensatzes sollte zu einer Kommunikation mit dem Datenbanksystem führen. Vielmehr sollen alle Änderungen an Daten zwischengespeichert und zu wohl definierten Zeitpunkten und dann gesammelt an das Datenbanksystem geschickt werden. Die Datenbankschnittstelle muß also intern über alle verwendeten Datensätze Buch führen und sollte dann z.B. am Ende einer Datenbanktransaktion nur die Daten an die Datenbank senden, die auch tatsächlich verändert wurden. Der Vorteil der Zwischenspeicherung der Datensätze besteht im wesentlich schnelleren Zugriff auf Objekte, die sich bereits im Speicher des Rechners befinden und nicht mehr aus der Datenbank gelesen werden müssen. Für einen Datenbankzugriff ist in der Regel die Kommunikation über ein Netzwerk mit dem Datenbankserver nötig. Der Datenbankserver muß eine SQL-Anfrage auswerten, die Ergebnisdaten von seinen Massenspeichermedien lesen und diese dann zurück über das Netzwerk senden. Befinden sich die Datensätze aber bereits auf dem Client-Rechner, so ist lediglich ein Zugriff auf den Speicher des Client-Rechners notwendig. Letzteres ist eine Operation, die um einige Zehnerpotenzen schneller abläuft als die Kommunikation mit einem Datenbankserver. Den Vorteil des schnelleren Zugriffs auf einmal gelesene Daten erkauft man sich allerdings mit einem erhöhten Aufwand zur Erhaltung der *Cache*-Konsistenz. Dieser Aufwand ist begründet durch Effekte, die beim parallelen Datenbankzugriff durch mehrere Benutzer entstehen können und im Abschnitt 3.4 beschrieben werden.

3.4 Isolation

Datenbanksysteme sind mehrbenutzerfähig. Dadurch ergibt sich eine Problemstellung, die unter dem Oberbegriff Isolation zusammengefaßt wird. Während ein Benutzer Daten verändert, muß verhindert werden, daß gleichzeitig ein anderer Benutzer dieselben Daten verändern kann. Datenbanksysteme lösen dieses Problem durch die Verwendung von Sperren. Sobald ein Benutzer auf Daten zugreift, werden diese Daten mit Sperren belegt, die verhindern, daß parallel weitere Benutzer dieselben Daten manipulieren. Beendet ein Benutzer seine Datenbanktransaktion, so werden alle Sperren, die im Verlauf dieser Transaktion angefordert wurden, wieder freigegeben. Erst danach können andere Benutzer wieder auf diese Daten zugreifen (zu Sperrverfahren vgl. Kapitel 4.4.2 in [LS87]). Wenn nun ein objektorientiertes Anwendungsprogramm auf Inhalte relationaler Datenbanken zugreift, so verhält es sich aus

der Sicht des Datenbanksystems wie jeder andere Datenbankbenutzer auch. Es liest Daten und nimmt gegebenenfalls Änderungen an diesen Daten vor.
Durch die im Abschnitt 3.3 erwähnte Zwischenspeicherung von Daten innerhalb der objektorientierten Anwendung ergibt sich nun ein neues Problem. Änderungen, die an den Inhalten der Datenbank vorgenommen werden, werden nicht sofort auf der Datenbank ausgeführt, sondern verzögert. Diese verspätete Ausführung von Datenbankzugriffen ist dem Benutzer des Anwendungsprogramms aber nicht zwangsläufig bewußt. Das Anwendungsprogramm bzw. die verwendete Datenbankschnittstelle muß daher sicherstellen, daß der Ausschnitt der Datenbank, der gerade im objektorientierten System sichtbar ist, nicht durch Datenbankzugriffe anderer Datenbankbenutzer manipuliert werden kann, bevor alle vorgenommenen Änderungen auch tatsächlich in der Datenbank ausgeführt worden sind. Auch wenn eine leistungsfähige Datenbank-Programmierschnittstelle dem Anwendungsprogrammierer und dem Anwender den Eindruck vermitteln kann, daß die Inhalte der Datenbank unmittelbar verändert werden, so muß diese Programmierschnittstelle dennoch die Tatsache berücksichtigen, daß sie mit *Kopien* der Daten arbeitet. Werden keine besonderen Vorkehrungen getroffen, so könnten sich die Ausgangsdaten in der Datenbank ändern, ohne daß das objektorientierte System dies erkennt. Erst beim Versuch, verzögerte Änderungsoperationen auf der Datenbank auszuführen, könnten sich Fehler durch zwischenzeitlich veränderte Datenbankinhalte ergeben, deren Ursachen für das Anwendungsprogramm bzw. seinen Benutzer dann nicht mehr nachvollziehbar sind.

Kapitel 4

Kommerzielle Werkzeuge für objektorientierten Datenbankzugriff

Das Problem des relationalen Datenbankzugriffs aus objektorientierten Systemen heraus ist noch ein vergleichsweise junges Forschungsgebiet. Dennoch sind für verschiedene objektorientierte Programmiersprachen und Entwicklungsumgebungen bereits diverse Produkte auf dem Markt, die eine Integration von relationalen Datenbanken in eine objektorientierte Umgebung leisten sollen. Diese Art von Produkten wird unter dem Begriff *objektrelationale Middleware* zusammengefaßt. Das bedeutet, die *Middleware* stellt eine zusätzliche Instanz zwischen der relationalen Datenbank auf der einen Seite und dem objektorientierten System auf der anderen Seite dar. Aufgabe dieser Instanz ist die Überbrückung der konzeptuellen Unterschiede zwischen den beiden beteiligten Systemen.
Drei typische Vertreter dieser Gattung werden in diesem Kapitel einander gegenübergestellt. Sie unterscheiden sich insbesondere hinsichtlich der Entwurfsmethode, die sie bevorzugt unterstützen. Abschnitt 4.1 stellt dafür ein Klassifikationsschema vor. Die Eigenschaften, hinsichtlich derer die unterschiedlichen Produkte miteinander verglichen werden, sind Gegenstand des Abschnittes 4.2. Die drei untersuchten Systeme werden daraufhin jeweils in einem eigenen Abschnitt vorgestellt. Ein direkter Vergleich der drei Systeme (Abschnitt 4.6) schließt das Kapitel ab.

4.1 Klassifikation objektrelationaler Middleware

Jedes der in diesem Kapitel untersuchten Middlewareprodukte wurde für die Unterstützung einer bestimmten Entwurfsmethode entwickelt (vgl. [Dem96]).
Die erste und aus der objektorientierten Sicht naheliegende Methode ist das sogenannte *Forward Engineering*. Das bedeutet, der Entwickler entwirft für sein Anwendungsprogramm ein Objektmodell und die objektrelationale *Middleware* generiert zu diesem Objektmodell halb- oder vollautomatisch ein zugehöriges Datenbankschema. Dieser Ansatz hat den Vorteil, daß sich die Strukturen der relationalen Datenbank optimal an das Anwendungsprogramm anpassen lassen. Das Produkt PERSISTENCE ist dieser Klasse zuzuordnen (vgl. Abschnitt 4.4).
Ist bereits ein Datenbankschema vorhanden, kann es in aller Regel nicht problemlos verändert werden, um den Anforderungen einer neuen objektorientierten Anwendung zu genügen. Da bereits andere Anwendungen im Einsatz sein können, deren Stabilität durch eine Veränderung des Datenbankschemas beeinträchtigt werden könnte, muß sich eine neue objektorien-

tierte Anwendung an das vorhandene Datenbankschema anpassen. Ziel ist dabei, aus den in der Datenbank enthaltenen Daten über ein Datenbankschema (Metadaten) "rückwärts" das dem Datenbankschema zugrundeliegende Datenmodell zu rekonstruieren. Auch für die Unterstützung dieses zweiten Ansatzes, des sogenannten *Reverse Engineering*, ist entsprechende *Middleware* verfügbar. Als Beispiel für ein solches Produkt fungieren hier zwei Entwicklungen der Firma ROGUE WAVE - DBTOOLS.H++ und OBJECT FACTORY (Abschnitt 4.3).
Die dritte kommerzielle Lösung, die hier betrachtet wird, stammt von der Firma ONTOS (Abschnitt 4.5) und kann sowohl für das *Forward Engineering* als auch das *Reverse Engineering* eingesetzt werden.
Für diese Arbeit sind vorwiegend solche Lösungen für objektrelationale *Middleware* interessant, die einem objektorientierten Anwendungsprogramm ein relationales Datenbankschema zugänglich machen, wobei das Schema nicht zwingend zusammen mit dem Anwendungsprogramm entwickelt worden sein muß. Das heißt, die *Middleware* unterstützt das *Reverse Engineering* und erzeugt für das objektorientierte System eine Repräsentation der relationalen Daten. Für das objektorientierte TYCOON-System wird anschließend eine solche *Middleware* entwickelt. *Middleware*-Produkte für das *Forward Engineering*, wie z.B. PERSISTENCE werden hier nur dann berücksichtigt, wenn sie Teillösungen für Probleme enthalten, die auch für das *Reverse Engineering* von Bedeutung sind.

4.2 Kriterien für die Bewertung existierender Systeme

Die Kriterien, nach denen die in dieser Arbeit betrachteten Systeme untersucht werden, orientieren sich im wesentlichen an den Anforderungen und Fragestellungen, die im Kapitel 3 formuliert worden sind.

- In welcher Form werden Konzepte der relationalen Datenbank auf objektorientierte Strukturen übertragen (Abschnitt 4.2.1)?
- Sind SQL-Kenntnisse zur Nutzung der *Middleware* hilfreich oder gar erforderlich (Abschnitt 4.2.2)?
- Berücksichtigt die *Middleware* die besonderen Probleme, die mit der Mehrbenutzerfähigkeit des Datenbanksystems verbunden sind (Abschnitt 4.2.3)?
- Werden von der *Middleware* Maßnahmen ergriffen, um den Kommunikationsaufwand zwischen Anwedung und Datenbank zu minimieren (Abschnitt 4.2.4)?

4.2.1 Abgebildete Konzepte

Für die betrachteten Middlewareprodukte ist zu untersuchen, in welcher Art und Weise Inhalte der relationalen Datenbank auf Strukturen eines objektorientierten Systems abgebildet werden. Dies schließt insbesondere folgende Fragestellungen mit ein.

- Welche Konzepte der relationalen Datenbank werden im objektorientierten System ab- oder nachgebildet? Beispiele sind hier Tabellen, Datensätze, Schlüssel, referentielle Integritätsbedingungen.
- Werden diese Konzepte direkt durch spezielle Klassen repräsentiert, die der Anwendungsprogrammierer verwenden muß, oder werden sie in komplexere Klassen integriert, die stärker von den technischen Details einer relationalen Datenbank abstrahieren?

- Muß das Datenbankschema, also die Menge der von einer Anwendung benutzten Datenbanktabellen, besondere Voraussetzungen erfüllen, damit ein Anwendungsprogramm über die *Middleware* darauf zugreifen kann?

4.2.2 Einbettung von SQL

Zugriffe auf relationale Datenbanken finden unabhängig von der verwendeten Programmiersprache in der Regel auf der Basis der Anfragesprache SQL statt. Ein weiterer Indikator für die Qualität der Abstraktionen, die eine Programmierschittstelle bereitstellt, ist daher die Frage, ob und in welchem Umfang SQL-Kenntnisse zur Verwendung der *Middleware* nötig sind.

- Muß der Anwendungsentwickler die Sprache SQL beherrschen, um auf Inhalte einer relationalen Datenbank zuzugreifen, oder ist ein Zugriff auch ohne SQL-Kenntnisse möglich?
- Falls der Anwendungsentwickler der Sprache SQL mächtig ist, kann er sie direkt für Anfragen verwenden?
- Dürfen Anfragen an die Datenbank beliebig komplex sein und wenn ja, in welcher Form werden die Ergebnisse verfügbar gemacht?

4.2.3 Isolation gegen parallele Datenbankzugriffe

Moderne relationale Datenbanken sind mehrbenutzerfähig. Das Datenbanksystem sorgt durch die Vergabe von Sperren dafür, daß dieselben Daten von höchstens einem Datenbankbenutzer zur Zeit verändert werden können. Diese Sperren werden erst am Ende einer Transaktion wieder freigegeben, also nachdem ein Benutzer einen Satz von Änderungsoperationen erfolgreich ausgeführt hat und diese Änderungen persistent und damit für alle anderen Datenbankbenutzer sichtbar gemacht werden sollen. Wird über eine objektrelationale *Middleware* auf die Datenbank zugegriffen, so ist zu untersuchen, ob diese Sperrmechanismen von der *Middleware* genutzt werden können oder ob die *Middleware* eigene Konzepte zur Sicherung der Konsistenz der Daten implementiert. Dabei stellen sich unter anderem die folgenden Fragen:

- Ist die betrachtete *Middleware* mehrbenutzerfähig?
- Wie wird sichergestellt, daß sich Datenbankinhalte nicht unbemerkt von der objektorientierten Anwendung verändern können?
- Wie reagiert die Anwendung, falls eine Isolation gegen parallele Datenbankzugriffe nicht möglich oder nicht erwünscht ist?

4.2.4 Beschleunigung der Datenbankkommunikation

Die Kommunikation mit einer entfernten Datenbank läßt sich beschleunigen, wenn die verwendete *Middleware* einen eigenen Zwischenspeicher (*Cache*) zur Speicherung von Daten bereithält. Unter Rückgriff auf diesen Cache können Anforderungen von Datensätzen unter Umständen erheblich schneller beantwortet werden, als wenn für jede Anforderung ein neuer Datenbankzugriff erforderlich ist. Auch die Kommunikation in die Gegenrichtung läßt sich beschleunigen, wenn Änderungsoperationen nicht direkt auf der Datenbank, sondern zunächst im Cache der *Middleware* gespeichert und zu einem späteren Zeitpunkt auf der Datenbank

ausgeführt werden. Ein Cache kann also signifikante Auswirkungen auf die Verarbeitungsgeschwindigkeit einer Datenbankanwendung haben.

4.3 DBTools.h++ und Object Factory

DBTools.h++ und Object Factory sind Produkte der Firma Rogue Wave. Für die Programmiersprache C++ wird mit DBTools.h++ [RW*96] eine Klassenbibliothek zur Verfügung gestellt, die auf Standardbibliotheken von Datenbankherstellern aufsetzt und damit die Basiskommunikation mit relationalen Datenbanken zur Verfügung stellt. Die bereits im Abschnitt 2.2 vorgestellt Schnittstelle des Tycoon-Systems zu relationalen Datenbanken setzt ebenfalls auf Bibliotheken der Datenbankhersteller auf und bietet einen vergleichbaren Funktionsumfang. Das Produkt Object Factory ist ein Code-Generator für eine schnelle, teilautomatisierte Anwendungsentwicklung (*Rapid Application Development*). Object Factory erlaubt es dem Programmierer, über eine grafische Oberfläche interaktiv Eigenschaften von Objekten der zukünftigen Anwendung zu spezifizieren und dann Programmcode gemäß dieser Spezifikation zu generieren (vgl. [OF197]). Für Objekte, die in Beziehung zu Inhalten einer relationalen Datenbank stehen, werden Klassen erzeugt, die die Funktionalität von DBTools.h++ verwenden.
Der Haupteinsatzzweck dieser beiden Produkte ist die Entwicklung von Anwendungssoftware für relationale Datenbanken. Dabei wird davon ausgegangen, daß ein Datenbankschema nicht allein von einer Anwendung benutzt wird, sondern von vielen verschiedenen, auch nicht objektorientierten Anwendungen. Das Datenbankschema ist daher in der Regel nicht speziell für die neue Anwendung entworfen worden und darf auch nicht für diese Anwendung verändert werden, da dies die Stabilität der bereits vorhandenen Anwendungen gefährden könnte. Das Datenbankschema wird also als gegeben akzeptiert und nicht den Erfordernissen einer neuen objektorientierten Anwendung untergeordnet (vgl. [OF197, S.2]).

Abgebildete Konzepte

Tabellen: Objekte der Klasse `RWDBTable` stellen Methoden zur Verfügung, mit denen sich Zeilen einer Tabelle auslesen und verändern lassen. Für die einzelnen Zugriffsoperationen kann ein `RWDBTable`-Objekt spezielle weitere Objekte der Klassen `RWDBSelector`, `RWDBUpdater`, `RWDBInserter` oder `RWDBDeleter` erzeugen. Jedes dieser neuen Objekte kapselt einen analogen SQL-Befehl. Diese Objekte verhalten sich ähnlich wie Standardein- und ausgabeströme oder können solche erzeugen, d.h. Datenbankinhalte können mit der in C++ üblichen Leseoperation (>>) und der korrespondierenden Schreiboperation (<<) zwischen Anwendung und Datenbank ausgetauscht werden.

Datensätze, Tabellenzeilen: In den DBTools.h++ können die Werte einer Tabellenzeile in einem Objekt der Klasse `DBRow` zusammengefaßt werden. `DBRow` repräsentiert eine Menge von Spaltenwerten. Eine Kapselung von Werten einer Zeile in einem Objekt einer eigenen, tabellenspezifischen Klasse ist an dieser Stelle nicht möglich. Eine solche höhere Abstraktion wird durch Object Factory erreicht. Für eine gegebene Tabelle kann dort ein Paar von Klassen generiert werden, das speziell an den Zugriff auf diese Tabelle angepaßt ist. Eine sogenannte *Domain Class* enthält dabei ausschließlich die Daten einzelner Tabelleneinträge, während in einer sogenannten *Interface Class* der

nötige DBTools.h++ Code enthalten ist, der die Kommunikation mit der Datenbank realisiert [OF397, S.10].

Spaltenwerte: Der Wert einer Spalte wird durch ein Objekt der Klasse `RWDBValue` repräsentiert und verfügt über Methoden, mit deren Hilfe sich der Wert als Objekt einer gewünschten Ergebnisklasse (z.B. `int` oder `string`) ausgeben oder manipulieren läßt. Für die Behandlung datenbankspezifischer Typen, die keine direkte Entsprechung in der objektorientierten Sprache besitzen, sind spezielle Klassen verfügbar, z.B. zur Manipulation von Datumsangaben oder für große, unstrukturierte Datenmengen (*Binary Large Objects*, BLOBs).

Ergebnistabellen und Joins: Ergebnisse komplexer Anfragen oder Joins zwischen Tabellen können ähnlich wie eine einzelne Tabelle durch eine eigene, von Object Factory generierte Klasse repräsentiert werden. Anders als bei einzelnen Tabellen können mit diesen Klassen Daten nur gelesen und nicht verändert werden. Das läßt darauf schließen, daß in den erzeugten Klassen außer der SQL-Anfrage selber keine zusätzlichen Daten über die Art der Beziehung zwischen den verwendeten Tabellen zur Verfügung stehen. Klassen für komplexe Anfragen verhalten sich damit wie *Views* in SQL. Durch Projektion und Aggregation entsteht eine Sicht der relationalen Daten, die in den meisten Fällen keine Rückschlüsse mehr auf die präzise Struktur der verwendeten Tabellen zuläßt.

Fremdschlüssel-Primärschlüssel-Beziehungen: Beziehungen zwischen Tabellen werden in einer relationalen Datenbank dadurch definiert, daß ein Fremdschlüsselattribut einer Tabelle als Wert einen Primärschlüssel der referenzierten Tabelle enthält. Entlang solcher Beziehungen kann man mit Hilfe spezieller Hilfsfunktionen navigieren, die von Object Factory bereitgestellt werden. Einer solchen Funktion wird das *Domain*-Objekt, das den Fremdschlüssel enthält, und das Datenbank-*Interface*-Objekt der referenzierten Tabelle übergeben. Über Methoden des Datenbank-*Interface*-Objekts kann man dann auf das referenzierte Objekt zugreifen. Für das Löschen, Ändern oder Auswählen der referenzierten Objekte müssen separate Navigationsfunktionen verwendet werden [OF397, S.11f].

Einbettung von SQL

Ein Zugriff auf Inhalte relationaler Datenbanken ist ohne SQL-Kenntnisse möglich. Typische Operationen auf einzelnen Datensätzen, wie Erzeugen, Löschen, Ändern, werden auf Methoden, bzw. Funktionen von DBTools.h++ abgebildet. Joins können im Join Designer der Object Factory grafisch und unter Zuhilfenahme der Fremdschlüssel-Primärschlüssel-Beziehungen der Datenbank erzeugt werden. Auch dies ist prinzipiell ohne SQL-Kenntnisse möglich. Nichtsdestotrotz muß der Anwendungsprogrammierer Kenntnisse über die Struktur seiner Datenbank besitzen und mit den Basiskonzepten wie Tabelle, Schlüssel, Anfrage, Join vertraut sein, um sinnvolle Klassen generieren zu können. Die grafischen Tools der Object Factory generieren Aufrufe von DBTools.h++ -Funktionen. Anwendungsentwickler, die über die entsprechenden SQL-Kenntnisse verfügen, können diese Funktionen für komplexe Anfragen auch direkt verwenden.

Isolation gegen parallele Datenbankzugriffe

Durch die DBTools.h++ werden keine besonderen Vorkehrungen getroffen, um eine Isolation von Änderungsoperationen gegen parallel laufende Datenbanktransaktionen zu gewährleisten. Ein aus der Datenbank gelesenes Objekt kann vom Anwendungsprogramm manipuliert und zu einem späteren Zeitpunkt in die Datenbank zurückgeschrieben werden. Der Rückschreibevorgang wird nicht durch einen internen Cache der DBTools.h++ verzögert, sondern unmittelbar ausgeführt, sobald die Anwendung die Daten zurückschreibt. Tritt dabei ein Fehler auf, der durch eine parallele Änderung eines anderen Datenbankbenutzers entstanden ist, muß die Anwendung diesen Fehler selber behandeln. DBTools.h++ und Object Factory unterstützen die Entwicklung von Anwendungsprogrammen, die direkt auf relationale Datenbanken zugreifen. Die entstehenden Anwendungen entsprechen damit einer sogenannten *Two-Tier*-Architektur, d.h. für den Betrieb der Anwendung sind nur zwei Komponenten nötig, der Datenbankserver als erste Komponente und das Anwendungsprogramm auf der anderen Seite. Andere Architekturen fügen zwischen diesen beiden Instanzen noch weitere Komponenten ein, um z.B. häufig genutzte Daten mehrerer verschiedener Anwendungsprogramme an einer zentralen Stelle zwischenzuspeichern. Mit DBTools.h++ und Object Factory erzeugte Anwendungen greifen auf die Datenbank als unabhängige, selbständige Klienten zu.

Beschleunigung der Datenbankkommunikation

Wie schon im vorigen Abschnitt angedeutet, verfügen die von der Object Factory erzeugten Klassen über keine eigenen Mechanismen zur zentralen Zwischenspeicherung von Datenbankinhalten. Gewisse Geschwindigkeitssteigerungen bei der Kommunikation mit der Datenbank sind dennoch möglich. So können Datenbankoperationen asynchron ausgeführt werden. Das bedeutet, die Operation wird vom Anwendungsprogramm angestoßen, das Resultat muß aber nicht abgewartet werden. Die Anwendung kann also weiterlaufen, während noch Datenbankoperationen vom Datenbanksystem bearbeitet werden. Eine weitere Möglichkeit zur Verkürzung der Antwortzeiten der Anwendung ist die Ausnutzung von Fähigkeiten zur parallelen Ausführung mehrerer Programmteile (*Multithreading*). Dadurch können Datenbankoperationen parallel abgearbeitet werden. Sowohl die asynchrone als auch die parallele Ausführung von Datenbankoperationen wird allerdings nicht direkt von den DBTools.h++ unterstützt. Vielmehr lassen sich diese Fähigkeiten nur dann nutzen, wenn sie von der Funktionsbibliothek des Datenbankherstellers, auf der die DBTools.h++ aufsetzen, unterstützt werden. Da die von Object Factory erzeugten Klassen auf den Fähigkeiten der DBTools.h++ aufsetzen, werden auch ihre Möglichkeiten zur Zugriffsbeschleunigung durch die Eigenschaften der verwendeten Funktionsbibliothek des Datenbankherstellers begrenzt.

4.4 Persistence

Die im Abschnitt 4.3 beschriebenen Produkte der Firma Rogue Wave sind in erster Linie dazu bestimmt, dem Anwendungprogrammierer den Zugriff auf bestehende relationale Datenbanken zu erleichtern oder gar erst zu ermöglichen. Im Vordergrund steht dort die Abbildung der Strukturen der Datenbank auf Klassen der objektorientierten Programmiersprache, also das *Reverse Engineering* (vgl. Abschnitt 4.1). Dort wird versucht, aus einem bereits implementierten Datenbankschema "rückwärts" auf das zugrundeliegende Datenmodell zu schließen.
Die Firma Persistence verfolgt mit ihren Produkten eine andere Strategie. Auch hier

ist zwar eine Weiterbenutzung vorhandener Datenbanken möglich. Haupteinsatzzweck des PERSISTENCE-Produkts ist aber das *Forward Engineering*, d.h. der Anwendungsentwickler beschreibt das Objektmodell seiner Anwendung und ein *Object Builder* erzeugt dann aus dieser Beschreibung die dazugehörigen Klassen und das dazu passende relationale Datenbankschema [Dem97]. Obwohl sich also die Schwerpunktsetzung der PERSISTENCE-Produkte von der in dieser Arbeit angestrebten Lösung unterscheidet, lohnt sich eine genauere Betrachtung. Denn auch beim *Forward Engineering* finden im Anwendungsprogramm Datenbankzugriffe statt, die letztendlich auf SQL-Anweisungen abgebildet werden müssen und mit Datenbankzugriffen anderer Anwendungen koordiniert werden müssen. Interessante Eigenschaften von PERSISTENCE sind in diesem Zusammenhang die Verwaltung von Transaktionen und Sperren und das Zwischenspeichern häufig genutzter Daten. Diese Aufgaben werden von einem sogenannten *Object Manager* übernommen, der als mittlere Komponente einer *Three-Tier*-Architektur zwischen Anwendungsprogramm und Datenbank geschaltet wird.

Abgebildete Konzepte

Da PERSISTENCE primär für das *Forward Engineering* bestimmt ist, kann hier nur die Frage untersucht werden, welche Konzepte der objektorientierten Sprache in der relationalen Datenbank abgebildet werden. Allerdings orientiert sich der Abbildungsprozeß sehr stark am relationalen Datenmodell.

Klassen: Die Klassen des Objektmodells werden soweit möglich direkt auf einzelne Tabellen in der Datenbank abgebildet. Die Attribute einer Klasse dürfen nicht beliebig komplex sein, sondern unterliegen gewissen Einschränkungen, um eine möglichst einfache Abbildung auf Tabellen zu gewährleisten.

Vererbung: Vererbung wird horizontal aufgelöst, d.h. für jede konkrete Klasse wird eine Tabelle in der Datenbank angelegt, die sämtliche Attribute dieser Klasse enthält. Darin sind auch die Attribute enthalten, die von etwaigen Superklassen geerbt wurden.

Beziehungen zwischen Objekten: In der in [Dem97] untersuchten PERSISTENCE-Version sind lediglich binäre 1:1- oder 1:N-Beziehungen abbildbar. Für komplexere Beziehungen, wie N:M- oder ternäre Beziehungen müssen separate Klassen definiert werden.

Mengen von Objekten: Mengen von Objekten, wie sie als Ergebnis von SQL-Anfragen auftreten können, werden durch Kollektionsklassen repräsentiert. Für jede Klasse wird eine korrespondierende Kollektionsklasse erzeugt. Eine Kollektion wird z.B. dann geliefert, wenn man die `querySQLWhere`-Methode einer Klasse aufruft. Diese Methode wird mit der `where`-Klausel einer SQL-Anfrage parametrisiert.

Änderungsoperationen: Jede aus dem Objektmodell erzeugte konkrete Klasse erbt von einer Superklasse namens `PersistenceObject`, die u.a. eine Methode zur Ausführung von Änderungsoperationen in der Datenbank bereitstellt.

Einbettung von SQL

Auch bei PERSISTENCE sind SQL-Kenntnisse nicht zwingend erforderlich, wenn sich der Entwickler mit dem erzeugten Klassen- und Datenbankschema zufriedengibt. Soll eine Menge von Objekten derselben Klasse bestimmt werden, so ist dies durch Angabe von Teilen einer

SQL-Anfrage (`where`-Klausel) möglich. Eine Möglichkeit zur Verwendung beliebig komplexer Anfragen ist nicht vorgesehen. Beziehungen, die in der Datenbank durch die Definition von Fremdschlüsseln realisiert werden müssen, werden in PERSISTENCE als direkte Objektbeziehungen implementiert. Diese Form der Abbildung, bei der eindeutige Identifikatoren einer Datenbank oder auch eines Objektspeichers auf direkte Objektreferenzen abgebildet werden, ist in der Literatur auch als *Semantic Key Swizzling* bekannt (vgl. [Cat94, S.166 ff], [KJA93, S.526]). Eine Navigation von einem Objekt zum anderen entlang von Beziehungen ist damit möglich. Durch den Objektcache von PERSISTENCE erfordern solche Objektnavigationen in vielen Fällen keine Kommunikation mit der Datenbank. Die zur Navigation nötigen Anfragen werden so weit wie möglich aus dem Cache befriedigt.

Isolation gegen parallele Datenbankzugriffe

Um Inkonsistenzen durch parallele Änderungsoperationen anderer Benutzer zu verhindern, werden dem Anwender von PERSISTENCE-Klassen unterschiedliche Arten von Transaktionen zur Verfügung gestellt. So kann festgelegt werden, ob bei lesenden Zugriffen bereits Sperren für die zu lesenden Objekte angefordert werden und wann Schreibzugriffe in der Datenbank ausgeführt werden. Schreibende Zugriffe können sofort oder gesammelt, am Ende einer Transaktion ausgeführt werden.
Außerdem ist ein optimistisches Sperren möglich, d.h. solange Objekte nur gelesen und verändert werden, werden keine Datenbanksperren erzeugt. Erst wenn die Veränderungen zurück in die Datenbank geschrieben werden, wird überprüft, ob die zu ändernden Objekte zwischenzeitlich durch andere Datenbankbenutzer manipuliert wurden. Dieses Sperrverhalten heißt optimistisch, weil die Grundannahme ist, daß die optimistisch "gesperrten" Datenbankinhalte in den meisten Fällen nicht durch parallele Prozesse verändert werden und somit eine Blockierung der Datenbank durch unnötige Sperranforderungen in den meisten Fällen nicht nötig ist (vgl. Abschnitt 4.4.3.2 in [LS87]). Auf diese Art und Weise verkürzen sich die Antwortzeiten der Datenbank und damit auch indirekt die Antwortzeiten des Anwendungsprogramms.
Im folgenden Abschnitt werden die wesentlichen Eigenschaften des Cache-Protokolls von PERSISTENCE vorgestellt. Durch einen gemeinsamen Cache für mehrere Anwendungen kann PERSISTENCE den parallelen Zugriff auf Objekte selber kontrollieren, ohne daß dafür zusätzliche Kommunikation mit der Datenbank erforderlich wäre.

Beschleunigung der Datenbankkommunikation

PERSISTENCE gestattet keinen unmittelbaren Zugriff auf die relationale Datenbank. Vielmehr läuft die gesamte Kommunikation mit der Datenbank über den schon erwähnten *Object Manager* ab. Dieser stellt eine zusätzliche Vermittlungsinstanz zwischen Anwendungen und der Datenbank dar, die insbesondere die Zwischenspeicherung häufig genutzter Daten übernimmt. Mehrere Anwendungen können über denselben *Object Manager* auf die Datenbank zugreifen. Der *Object Manager* verwaltet für alle Anwendungen einen gemeinsamen Objektcache. Fordert ein Anwendungsprogramm Daten aus der Datenbank an, wird zunächst überprüft, ob diese Daten bereits im Cache existieren. Ist das der Fall, findet kein Datenbankzugriff statt und die Daten werden direkt aus dem Cache geliefert [AKK95]. Werden Daten verändert, so erhält jedes Anwendungsprogramm einen eigenen Cache, in dem alle Änderungen zwischengespeichert werden. Beendet eine Anwendung ihre Transaktion, werden alle Änderungen auf

der Datenbank ausgeführt und der gemeinsame Cache auf den aktuellen Stand gebracht. Die privaten Caches aller anderen Anwendungen erhalten eine Nachricht über die Veränderung und können dann gemäß ihres eingestellten transaktionalen Verhaltens darauf reagieren.
PERSISTENCE verwaltet den Zugriff auf die Inhalte der relationalen Datenbank über sogenannte *Smart Pointers*. D.h. ein Anwendungsprogramm, das einen Datensatz bearbeiten möchte, bekommt keine völlig eigenständige Kopie der Daten, sondern nur eine Referenz auf den Datensatz in der Datenbank. Dadurch werden Inkonsistenzen vermieden, die dadurch entstehen könnten, daß verschiedene Teile der Anwendung jeweils auf eigenen Kopien der Originaldaten arbeiten. Änderungen, die in einem Teil der Anwendung vorgenommen werden, werden unmittelbar in allen anderen Teilen der Anwendung sichtbar. Die Speicherung in der Datenbank findet dann statt, wenn das Anwendungsprogramm seine Transaktion beendet [KJA93, S. 527].

4.5 ONTOS*Integrator

Der INTEGRATOR der Firma ONTOS besitzt eine ähnliche Funktionalität wie der *Object Builder* von PERSISTENCE. Seine Einsatzgebiete liegen sowohl im *Forward* als auch im *Reverse Engineering*. Ein interaktives Werkzeug schlägt dem Anwendungsentwickler eine Abbildung eines Datenbankschemas auf objektorientierte Klassen vor. Dieser Vorschlag kann dann bei Bedarf innerhalb des Werkzeugs überarbeitet werden. Anschließend werden gemäß der so definierten Abbildungsvorschrift Klassen erzeugt. Auch der umgekehrte Fall, die Abbildung eines Objektmodells auf ein Datenbankschema, ist möglich.

Abgebildete Konzepte

Tabellen: Einzelne Tabellen können auf eine oder mehrere Klassen abgebildet werden. Zu jeder Spalte der Tabelle kann in der korrespondierenden Klasse ein Attribut angelegt werden. Repräsentiert eine Tabelle eine sogenannte Universalrelation, so kann diese Relation auf mehrere Klassen abgebildet werden, die zueinander in Vererbungsbeziehungen stehen. Eine Universalrelation enthält alle Attribute, die innerhalb einer Vererbungshierarchie auftreten können. Der Typ eines Tabelleneintrags wird anhand des Wertes eines "Typattributs" ermittelt. Diese Datenstruktur wird bei der relationalen Modellierung von Vererbungsbeziehungen genutzt, wenn die Zugriffe auf die Objekte der Hierarchie meistens über dieselben gemeinsamen Attribute erfolgen. Dadurch werden aufwendige Joins in der Datenbank vermieden. Inhalte einer Tabellenzeile werden je nach vorher gewählter Abbildung auf Werte von Attributen einer oder mehrerer Klassen abgebildet.

Fremdschlüssel-Primärschlüssel-Beziehungen: ONTOS bietet dem Entwickler zahlreiche Möglichkeiten, über Fremdschlüssel definierte Beziehungen in die objektorientierte Welt abzubilden.

- Ein Fremdschlüssel repräsentiert eine 1:N-Beziehung; das heißt, die Tabelle, die ein Fremdschlüsselattribut besitzt, enthält N Datensätze, die auf eine Zeile der referenzierten Tabelle verweisen können. Die zur referenzierten Tabelle korrespondierende Klasse erhält dann ein Attribut, das eine Menge von Objekten der referenzierenden Klasse repräsentiert. Die zur referenzierenden Tabelle korrespondierende Klasse erhält ein Attribut das direkt das referenzierte Objekt repräsentiert.

- Ist ein Fremdschlüssel gleichzeitig Primärschlüssel der referenzierenden Klasse, so bietet ONTOS dem Anwendungsentwickler zwei Interpretationsvarianten an. Die erste Abbildungsmöglichkeit ist eine 1:1-Beziehung, d.h. beide Klassen erhalten ein Attribut, das ein Objekt der jeweils anderen Klasse aufnimmt. Die zweite Möglichkeit ist die Herstellung einer Vererbungsbeziehung. Da beide Klassen denselben Primärschlüssel aufweisen, liegt der Schluß nahe, daß ein gleicher Primärschlüsselwert in beiden Tabellen nicht verschiedene, sondern dasselbe Objekt bezeichnet. Existiert ein Eintrag also nur in der vom Fremdschlüssel referenzierten "Elterntabelle", gehört es zur korrespondierenden Elternklasse. Taucht derselbe Eintrag auch in der "Kindtabelle" auf, so gehört er zur Kindklasse, enthält also alle Attribute der Elternklasse und zusätzlich seine eigenen. Eine solche Beziehung zwischen Tabellen bezeichnet man auch als vertikale Zerlegung einer Vererbungshierarchie.
- N:M-Beziehungen werden im relationalen Modell durch separate Beziehungstabellen modelliert. Diese Tabellen enthalten in einer Zeile jeweils Fremdschlüssel auf diejenigen Tabellen, die an der Beziehung partizipieren. ONTOS kann sowohl die Beziehungstabelle als auch die an der Beziehung beteiligten Basistabellen direkt auf getrennte Klassen abbilden. Alternativ dazu kann anstelle der Beziehungsklasse auch ein mengenwertiges Attribut in jede der an der Beziehung beteiligten Basisklassen eingefügt werden, das direkt alle zugehörigen Objekte der jeweils anderen Basistabelle liefert. Dadurch wird das Objektmodell übersichtlicher und seine Anwendung wird vereinfacht.
- Ternäre bzw. n-äre Beziehungen werden im relationalen Modell ebenfalls durch separate Beziehungstabellen dargestellt. Diese werden auf eigene Beziehungsklassen abgebildet. Eine direkte Repräsentation der Beziehung als Attribut in den beteiligten Basisklassen ist nicht möglich, da jedes Objekt einer Basisklasse zu mehreren Objekten in mehreren Klassen in Beziehung stehen kann.

Wie schon an den Abbildungsregeln zu erkennen, unterstützt auch ONTOS die direkte Objekt-zu-Objekt-Navigation. Da die Abbildung aber nicht eindeutig ist, ist in jedem Fall durch den Benutzer eine Auswahl hinsichtlich der Abbildungsvorschrift zu treffen.

Stored Procedures: Moderne relationale Datenbanken bieten die Möglichkeit, komplexe Änderungsoperationen durch sogenannte *Stored Procedures* abzuwickeln. Dies sind in einer prozeduralen Sprache formulierte Operationen, die unter einem eindeutigen Namen in der Datenbank gespeichert werden und sich jederzeit unter Angabe dieses Namens ausführen lassen. ONTOS kann *Stored Procedures* auf Methoden der erzeugten Klassen oder auf freie Funktionen abbilden.

Einbettung von SQL

Das von ONTOS erzeugte Objektmodell und die zugehörigen Klassen sind ohne SQL-Kenntnisse nutzbar. Die für die zu entwickelnde Anwendung relevanten Beziehungen zwischen Objekten und die dafür nötigen SQL-Anfragen werden während des interaktiven Abbildungsprozesses generiert und in einem speziellen Speicher (einem *Repository*) abgelegt. Der Zugriff auf einzelne Objekte findet später durch Angabe ihres Primärschlüssels oder durch eine SQL-Anfrage statt.

Isolation gegen parallele Datenbankzugriffe

ONTOS nutzt weitgehend die Transaktionsmechanismen des verwendeten Datenbanksystems. D.h. sobald eine Zeile einer Datenbank gelesen wird, wird sie in der Datenbank gesperrt und erst am Ende einer Transaktion wieder freigegeben.

Beschleunigung der Datenbankkommunikation

ONTOS stellt jedem Anwendungsprogramm einen eigenen Cache zur Verfügung, in dem bereits aus der Datenbank gelesene Objekte zwischengespeichert werden. Die Navigation zwischen den Objekten wird so weit wie möglich im Cache abgewickelt. Da die zugehörigen Datensätze in der Datenbank gesperrt werden, solange sie sich im Cache befinden, ist eine Veränderung durch parallele Datenbankbenutzer nicht möglich. Der Cache wird geleert, sobald das Anwendungsprogramm seine Transaktion beendet.

4.6 Vergleich der unterschiedlichen Lösungen

	DBTools.h++	Object Factory	Persistence	ONTOS
Forward Engineering	nein	nein	ja	ja
Reverse Engineering	nein	ja	(ja)	ja
Generierung von DB-Klassen	nein	halbautom.	autom.	halbautom.
direkte SQL-Ausführung	ja	DBTools.h++	where-Klausel, kein vollst. SQL	
Caching	nein	nein	ja [1),2)]	ja [2)]
mehrere Clients	nein	nein	ja	nein
Sperrkonzept	kein eigenes, von DB abhängig		ja [3)]	ja [4)]
Abbildung von Beziehungen				
1:1	nein	ja	ja	ja
1:N	nein	Navigationsfunkt.	Methoden der generierten Klassen	
N:M	nein	nein	Verbundklassen	Methoden
n-äre	nein	nein	Verbundklassen	

1) gemeinsamer Cache für alle Clients
2) separater Cache pro Client
3) Versionskontrolle durch gemeinsamen Cache
4) Sperrung gecachter Daten in der Datenbank

Tabelle 4.1: Vergleich der untersuchten objektrelationalen Middleware

Tabelle 4.1 faßt die wesentlichen Eigenschaften der betrachteten Systeme zusammen. Die vielseitigsten Möglichkeiten für den Entwurf objektrelationaler Anwendungen stellt das System von ONTOS bereit. Es unterstützt das *Forward* und das *Reverse Engineering* gleichermaßen und erlaubt neben einer automatischen Abbildung zwischen relationaler und objektorientierter Welt auch eine interaktive Spezifikation der Abbildungsvorschriften. PERSISTENCE wurde hingegen primär für das *Forward Engineering* entwickelt und ist daher in der Unterstützung vorhandener Datenbankschemata weniger flexibel.

Im Gegensatz zu ONTOS verfügt PERSISTENCE über eine leistungsfähigere Cache-Verwaltung. Diese ermöglicht mehreren PERSISTENCE-Anwendern die Nutzung eines gemeinsamen Cache. Zugriffskonflikte können bereits abgefangen werden, bevor ein Datenbankzugriff erfolgt.
Die OBJECT FACTORY von ROGUE WAVE dient dem *Reverse Engineering*. Ihr Hauptziel ist die Unterstützung des objektorientierten Programmierers beim Zugriff auf vorhandene relationale Daten. Leistungssteigerungen des Anwendungsprogramms durch Caching von Datenbankzugriffen werden dem Anwendungsprogrammierer überlassen. Eine Integration von Objektbeziehungen in die Objekte selbst ist nicht vorgesehen.
Die DBTOOLS.H++ stellen für sich allein noch keine echte objektrelationale *Middleware* dar. Sie bilden lediglich vorhandene Datenbankprogrammierschnittstellen auf eine objektorientierte Schnittstelle ab.
Die Funktionalität einer objektrelationalen Klassenbibliothek für TYCOON wird sich hinsichtlich der Abbildung von Datenbankinhalten an den Fähigkeiten eines Klassengenerators orientieren, wie er bei ONTOS oder OBJECT FACTORY zu finden ist. Für das Caching von Datenbankinhalten kommen Mechanismen in Frage, wie sie bei PERSISTENCE und ONTOS existieren. Da TYCOON im Gegensatz zu den anderen hier vorgestellten Systemen über ein eigenes Persistenzkonzept verfügt (vgl. dazu Abschnitt 5.1.3), sind zusätzliche Entwicklungen nötig, um die Konsistenz zwischen den Daten des Datenbanksystems und persistenten Objekten des TYCOON-Systems zu gewährleisten.

Kapitel 5

Entwurf einer objektrelationalen Klassenbibliothek für Tycoon

Für das am Fachbereich Informatik (Arbeitsbereich DBIS) der Universität Hamburg und von der Firma HIGHER-ORDER entwickelte TYCOON 2-System [GMSS97] wird im folgenden ein Subsystem entwickelt, das TYCOON-Anwendungsprogrammen den Zugriff auf relationale Datenbanken gestattet. Ziel ist dabei ein Zugriff, der für den Anwender die wesentlichen Komponenten des Datenbankschemas unmittelbar zugänglich macht und Metainformationen aus der Datenbank zur Implementierung leistungfähiger Zugriffsmechanismen verwendet, die in dieser Form in der relationalen Welt nicht zur Verfügung stehen.
In Abschnitt 5.2 werden grundlegende Eigenschaften von Objekten spezifiziert, die Inhalte der relationalen Datenbank repräsentieren sollen. Außerdem werden die zu ihrer Erzeugung nötigen Maßnahmen diskutiert. Anschließend wird erörtert, in welcher Weise und zu welchen Zeitpunkten diese Objekte mit der Datenbank kommunizieren und wie dabei dem Bedarf nach Isolation, Konsistenz- und Integritätssicherung Rechnung getragen wird (Abschnitt 5.3). Schließlich wird im Abschnitt 5.4 die Kapselung der Datenbankzugriffe via SQL durch eine objektorientierte Abstraktion diskutiert. Vor dem eigentlichen Entwurf ist aber eine Betrachtung einiger besonderer Eigenschaften des TYCOON 2-Systems notwendig, um den Anwendungsbereich abzugrenzen, für den es im Kontext dieser Arbeit eingesetzt wird.

5.1 Eigenschaften des objektorientierten Systems

Das TYCOON 2-System unterscheidet sich von seinem Vorgänger TYCOON [Mat93] vor allem durch eine objektorientierte Programmiersprache, deren Objektmodell in Abschnitt 5.1.1 vorgestellt wird. Um dem veränderten Sprachparadigma Rechnung zu tragen, wurden alle Komponenten des nicht-objektorientierten Vorgängersystems, also Compiler [Wie97], Typüberprüfung [Ern98] und virtuelle Maschine [Wei98] zur Ausführung der TYCOON-Programme, neu entwickelt. Vom ursprünglichen TYCOON-System werden die Konzepte des polymorphen Typsystems (vgl. dazu Abschnitt 5.1.2), Typen und Funktionen höherer Ordnung und die Möglichkeit zur persistenten Speicherung des aktuellen Zustands des Gesamtsystems (vgl. Abschnitt 5.1.3) übernommen. Besonders die orthogonale Integration der Persistenz von Objekten in das System wirft einige besondere Fragestellungen auf, die bei den im Kapitel 4 untersuchten, größtenteils nicht-persistenten Systemen nicht zum Tragen kommen. Für diese Arbeit relevante Fragen der Speicherverwaltung innerhalb des persistenten Objektsystems

beleuchtet Abschnitt 5.1.4. Details des TYCOON 2-Systems werden hier nur in soweit behandelt, als sie für die folgenden Entwurfs- und Implementierungsarbeiten von Bedeutung sind. Für nähere Informationen zu den allgemeinen Konzepten des TYCOON 2-Systems und seines Vorgängers sei auf die umfangreiche Literatur (z.B. [Mat93, GM95, Wah97]) zu diesem Thema verwiesen.
Alle im Rahmen dieser Arbeit durchgeführten Entwurfs- und Implementierungsarbeiten beziehen sich auf das objektorientierte TYCOON 2-System. Dieses wird im weiteren Verlauf nur noch als "das TYCOON-System" oder kurz "TYCOON" bezeichnet. Der Name wird nur dann mit der Versionsnummer 2 versehen, wenn Unterschiede zum ersten, nicht objektorientierten TYCOON-System betont werden sollen.

5.1.1 Objektmodell

Die Basisabstraktion der TYCOON 2 LANGUAGE, im folgenden TL2 genannt, sind Objekte. Jedes Objekt besitzt einen Zustand und ein Verhalten. Der Zustand wird in sogenannten *Slots* repräsentiert, die man sich als Behälter für Werte und Referenzen auf andere Objekte vorstellen kann. Das Verhalten wird durch die Methoden des Objekts bestimmt, die Zugriff auf den Zustand des Objekts haben und diesen verändern können. Jedes Objekt gehört zu einer Klasse. Eine Klasse beschreibt die Struktur und das Verhalten der zu ihr gehörigen Objekte. Auch Klassen sind Objekte.
Die Erzeugung neuer Objekte wird in vielen objektorientierten Programmiersprachen durch spezielle Methoden, sogenannte Konstruktoren realisiert (z.B. C++ [Lou94, S. 390] oder Eiffel [Lou94, S. 399 f]). Konstruktoren werden in diesen Sprachen in derselben Klasse definiert wie das Verhalten der erzeugten Objekte. Da die Erzeugung eines neuen Objekts aber kein Verhalten des zu erzeugenden Objekts ist, sondern vielmehr ein Verhalten der zugehörigen Klasse, wird in TL2 ein anderes Vorgehen gewählt: Da eine Klasse selber ein Objekt ist, kann sie auch ein eigenes Verhalten besitzen. Das Verhalten einer Klasse, zu dem also auch die Erzeugung neuer Objekte gehört, wird in ihrer *Metaklasse* beschrieben.
Beim Zugriff auf relationale Datenbanken müssen Objekte erzeugt werden, die Inhalte der Datenbank repräsentieren. Da jedes dieser Objekte untrennbar mit Einträgen in der Datenbank verbunden ist, müssen Methoden zur Verfügung gestellt werden, die die Aufrechterhaltung dieser Verbindung sicherstellen bzw. sie in konsistenter Weise beenden und wiederherstellen können. Dieses Verhalten ist nicht spezifisch für ein konkretes Objekt, sondern betrifft in vielen Fällen alle Objekte einer Klasse, z.B. beim Beenden einer Datenbanksitzung. Eine zentrale Verwaltung von Objekten, die sich auf Datenbankinhalte beziehen, durch ihre jeweilige Klasse erscheint daher sinnvoll. Verhalten, das alle Objekte einer "Datenbankklasse" betrifft, wie die Zwischenspeicherung für schnellere Zugriffe, wird daher sinnvollerweise in der Metaklasse der Datenbankklasse definiert.
Eine Klasse erfüllt in TYCOON verschiedene Aufgaben. Diese werden im folgenden unter Zuhilfenahme eines Klassifikationsschemas, zusammengefaßt, das in [Wet94, S. 60 f] zur Klärung des Klassenbegriffs eines objektorientierten Datenmodells verwendet wird.

Intensionaler Charakter einer Klasse: Eine Klasse legt durch ihre Definition von Attributen und Methoden die gemeinsame Struktur und das Verhalten für eine Menge von Objekten fest.

Extensionaler Charakter: Da jedes Objekt in TYCOON einer Klasse zugeordnet wird, legen Klassen nicht nur gemeinsame Strukturen von Objekten fest; sie unterteilen damit

die Menge aller in System vorhandenen Objekte auch logisch in Kollektionen gleichartiger Objekte. Die Extension einer Klasse ist damit die Menge aller im System existierenden, zu dieser Klasse gehörigen Objekte. Obwohl die Extension einer Klasse bereits implizit durch die Zuordnung von Objekten zu dieser Klasse gegeben ist, ist im Rahmen dieser Arbeit eine explizite Speicherung aller zu einer Klasse gehörigen Objekte sinnvoll. Die modifizierenden und identifizierenden Aufgaben einer Klasse beinhalten Anforderungen, die für alle Objekte einer Klasse erfüllt werden müssen.

Modifizierender Charakter: Auch eine Klasse besitzt in TYCOON einen Zustand (z.B. eine Liste der von ihr erzeugten Objekte) und ein Verhalten (z.B. Erzeugung neuer Objekte). Eine Klasse ist damit ihrerseits ein Objekt, dessen Struktur und Verhalten spezifiziert werden müssen. Diese Spezifikation findet in TYCOON nicht innerhalb der Klasse selbst statt, sondern sie erfolgt in einer zugehörigen Metaklasse. Durch die Verwendung von Metaklassen kann dadurch in TYCOON auf programmiersprachlicher Ebene zwischen dem modifizierenden Charakter und dem intensionalen Charakter einer Klasse unterschieden werden.

Identifizierende Aufgaben einer Klasse: Jedes Objekt in einem objektorientierten System besitzt eine Identität, die unveränderlich ist. Dem Benutzer des Systems wird daher keine Funktionalität für den Zugriff auf die Implementierung dieser Identität zur Verfügung gestellt. Dennoch ist es in vielen Fällen erforderlich, besonders bei Datenbankanwendungen, Objekte eindeutig zu identifizieren, um sie dann gezielt manipulieren zu können. Diese Identifikation muß anhand von Eigenschaften des gesuchten Objekts erfolgen. Zur eindeutigen Identifikation von Objekten, die Inhalte eines relationalen Datenbankschemas repräsentieren, wird in dieser Arbeit der Primärschlüssel der zugehörigen Datenbanktabelle verwendet. Die Zuordnung von Objekten einer Klasse zu Datensätzen einer Tabelle ist Bestandteil des Verhaltens einer Klasse. Auch dieses Verhalten wird in TYCOON in der Metaklasse einer Klasse festgelegt.

Neben den Klassen und Objekten, die ein Anwendungsentwickler selber erzeugt, verfügt TYCOON über eine Reihe vordefinierter Klassen und Objekte, die innerhalb von Anwendungen genutzt werden können. So sind z.B. der Compiler und die Typüberprüfung über eigene Objekte zu erreichen. Ein Anwendungsprogramm kann damit während seiner Laufzeit im Objektspeicher neue Klassen erzeugen, übersetzen und benutzen.

5.1.2 Polymorphes Typsystem

Das Typsystem von TYCOON unterstützt neben dem in objektorientierten Sprachen üblichen Subtyppolymorphismus auch parametrischen Polymorphismus. Jede Klasse definiert implizit einen Typ, der die für andere Objekte sichtbare Schnittstelle der Objekte dieser Klasse repräsentiert. Sollen nun generische Klassen entwickelt werden, die mit Objekten unterschiedlicher Typen umgehen können, so ist man bei objektorientierten Programmiersprachen ohne parametrischen Polymorphismus auf die folgende Vorgehensweise angewiesen: Die neue Klasse, die z.B. Listen von Objekten verwalten soll, erhält als Elementtyp einen Supertyp aller zu verwaltenden Objekte. Jedes Objekt, das Subtyp dieses Elementtyps ist, kann von Objekten der neuen Klasse verarbeitet werden. Allerdings "sehen" sie dabei nur die Schnittstelle des allgemeinen Elementtyps; ein Zugriff auf Methoden, die erst in Subklassen definiert werden, ist später nicht mehr möglich. Dazu ist eine Typumwandlung zur Laufzeit, ein sogenannter

Downcast, nötig. Dies kann allerdings zur Programmlaufzeit zu Instabilitäten führen, wenn das umgewandelte Objekt ursprünglich gar nicht diesen Typ besaß.
Die Gefahr solcher Laufzeitprobleme wird durch das Typsystem von TL2 vermieden. In TL2 können Klassen mit Typen parametrisiert werden. Das hat den Vorteil, daß generische Klassen geschrieben werden können, die ebenfalls mit Objekten eines allgemeinen Supertyps umgehen können, die aber den konkreten Typ der behandelten Objekte beibehalten. Zur Veranschaulichung ein kleines Beispiel:

> Angenommen, es gibt eine Klasse zur Verwaltung von Listen. Eine Liste von Personen ist dann in einer objektorientierten Programmiersprache ohne parametrischen Polymorphismus eine Liste von Objekten. Entnimmt man nun ein Objekt dieser Liste, muß man es zur Programmlaufzeit explizit in ein Personenobjekt umwandeln, damit man Zugriff auf die Methoden der Personenobjekte erhält. In TL2 parametrisiert man die Liste bei Ihrer Erzeugung mit dem Typ der zu verwaltenden Objekte und erhält damit eine Liste von Personenobjekten. Entnimmt man dieser Liste ein Objekt, so erhält man unmittelbar ein Objekt des Typs Person mit all seinen Methoden und *Slots*.

Der Vorteil dieses Ansatzes ist die Tatsache, daß bereits zum Zeitpunkt der Programmübersetzung sichergestellt werden kann, daß Objekte, die auf diese Liste zugreifen, mit dem korrekten Typ der Listeninhalte arbeiten. Aufrufe nicht vorhandener Methoden lassen sich somit bereits zum Übersetzungszeitpunkt vermeiden.
Alle in TYCOON verfügbaren Klassen zur Verwaltung großer Datenmengen (Listen, Felder, Kollektionen, Wörterbücher) sind mit den Typen der zu verwaltenden Objekte parametrisierbar. In dieser Arbeit entwickelte Methoden, die größere Mengen von Daten zurückliefern, erzeugen sogenannte `Reader`. Objekte der Klasse `Reader` liefern in TYCOON einen Strom von Objekten, der einmal vom Anfang bis zu seinem Ende elementweise durchlaufen werden kann. Dies ist vergleichbar mit Eingabeströmen, wie sie in den meisten Programmiersprachen beim zeichenweisen Lesen von Benutzereingaben eingesetzt werden. Außerdem sind `Reader` in ihrem Verhalten den aus der Datenbankprogrammierung bekannten Cursorn sehr ähnlich. Aus einem `Reader` läßt sich in TYCOON nahezu jede andere Massendatenstruktur erzeugen. Alle Klassen für Massendatenstrukturen verfügen über entsprechende Methoden, die aus einem Reader die jeweilige spezifische Datenstruktur erzeugen.
Der parametrische Polymorphismus erweist sich bei der Spezifikation von Methoden als hilfreich, die zwar für alle datenbankspezifischen Objekte gleich sind und damit nur einmal in einer abstrakten Superklasse definiert werden müssen, die aber den Typ der Objekte für spätere Operationen nicht verändern bzw. durch einen weniger spezifischen Supertyp ersetzen dürfen.

5.1.3 Persistenz

Der Zustand des TYCOON-Systems läßt sich jederzeit speichern. Alle dafür erforderlichen Daten werden in einem gemeinsamen Speicher, dem sogenannten TYCOON-STORE abgelegt. Mit Hilfe dieser Daten stellt das TYCOON-System bei einem Neustart den Zustand zum Zeitpunkt der letzten Speicherung wieder her. Da TYCOON die Nutzung von externen Ressourcen wie relationalen Datenbanken ermöglicht, stellt sich die Frage, inwieweit sich der Zustand solcher externen Ressourcen persistent machen läßt.
Die Verbindung zwischen einer relationalen Datenbank und dem TYCOON-System besteht

höchstens für den Zeitraum, in dem das TYCOON-System aktiv ist. Wird TYCOON beendet, so endet automatisch auch die Verbindung zur Datenbank. Änderungen, die danach an der Datenbank vorgenommen werden, unterliegen nicht mehr der Kontrolle des TYCOON-Systems. Enthält der TYCOON-STORE nun Daten, die auf Datenbankinhalten basieren, so muß bei einem Neustart mit diesem TYCOON-STORE überprüft werden, ob der im Objektspeicher abgelegte Zustand noch mit dem aktuellen Datenbankzustand übereinstimmt.
Ein ähnliches Problem ergibt sich aus dem transaktionalen Verhalten einer Datenbank. Eine Transaktion ist eine Menge von Datenbankoperationen, die zu einer einzigen, atomaren Operation zusammengefaßt werden. Eine Transaktion kann nur dann erfolgreich beendet werden, wenn sich all ihre Teiloperationen erfolgreich ausführen lassen. Wird eine Transaktion abgebrochen, weil sich eine ihrer Teiloperationen nicht ausführen läßt, so wird auch keine der anderen Teiloperationen ausgeführt. Aus der Sicht der Datenbank ist ein laufendes TYCOON-System ein gewöhnlicher Datenbankklient, der Transaktionen ausführt. Wenn nun der Zustand des TYCOON-Systems gespeichert würde, während noch eine Datenbanktransaktion läuft, ergäbe sich ein Problem. Die laufende Datenbanktransaktion könnte nach dem Speichern des TYCOON-Systemzustandes noch scheitern. Dadurch würden alle vorherigen Operationen dieser Transaktion in der Datenbank rückgängig gemacht (Rollback). Das TYCOON-System befände sich nach einem Neustart aber wieder in einem Zustand, der vor dem Abbruch der Transaktion vorgelegen hatte. Dieser Zustand wäre dann nicht mehr mit dem tatsächlichen Datenbankzustand identisch. Es müssen also Vorkehrungen getroffen werden, die solche Inkonsistenzen verhindern.
Für den Umgang mit externen Diensten (Ressourcen), die sich der Kontrolle des persistenten Objektspeichers ganz oder teilweise entziehen, steht in TYCOON eine Klasse `Resource` zur Verfügung. Diese Klasse enthält Methoden, mit deren Hilfe sich externe Ressourcen beim Sichern des TYCOON-Objektspeichers in einen definierten Zustand bringen lassen. Beim Wiederanfahren des Objektspeichers versucht das TYCOON-System, diesen definierten Zustand zu rekonstruieren. Aus der Sicht des Benutzers erscheinen die externen Ressourcen dann persistent wie alle anderen Objekte, die sich innerhalb des Objektspeichers befinden.
Ressourcen werden bei ihrer Erzeugung implizit bei einem zentralen `Resourcemanager` registriert. Wird nun der Objektspeicher gesichert, so wird allen `Resource`-Objekten eine Nachricht darüber geschickt und diese müssen ihrerseits alle Maßnahmen ergreifen, um ihren momentanen Zustand bei einem Neustart wiederherstellen zu können. Wird nun ein solcher Objektspeicher neu gestartet, der bei seiner letzten Sicherung offene Ressourcen enthielt, so wird diesen Ressourcen unmittelbar nach dem Start eine `reopen`-Nachricht geschickt. Die `Resource`-Objekte versuchen dann mit Hilfe der Daten, die sie bei der letzten Sicherung erzeugt haben, ihren Zustand zum Zeitpunkt der Sicherung wiederherzustellen.

5.1.4 Speicherverwaltung

Persistente Objektsysteme stellen, wie im Abschnitt 5.1.3 gezeigt, leistungsfähige Mechanismen zur langfristigen Speicherung komplexer Objekte bereit. Innerhalb eines Anwendungprogramms werden aber nicht nur langfristig zu speichernde Objekte erzeugt. Ein Großteil der in einem Objektspeicher vorhandenen Objekte wird nur kurzzeitig benötigt, z.B. nur während der Ausführung einer Methode. Danach kann der von diesen temporären Objekten belegte Speicher wieder freigegeben werden.
Diese Bereinigung des Speichers von nicht mehr benötigten Objekten wird durch eine sogenannte *Garbage Collection* erreicht. Ein *Garbage Collector* überprüft entweder in regelmäßigen

Abständen oder bei bestimmten Ereignissen (z.B. Überschreiten eines Schwellwertes für die Größe des Objektspeichers), welche Objekte im System nicht mehr erreichbar sind. Ein Objekt ist dann nicht mehr erreichbar, wenn es weder direkt noch indirekt über Methoden oder Attribute anderer erreichbarer Objekte angesprochen werden kann. Der von solchen nicht erreichbaren Objekten belegte Speicher wird während einer Garbage Collection wieder freigegeben.
Der Garbage Collector gibt den Speicher all derjenigen Objekte frei, auf die keine starken Referenzen verweisen. In TYCOON gibt es zusätzlich das Konzept der schwachen Referenzen oder *Smart Pointers*, das über eine eigene, mit einem Objekttyp parametrisierbare Klasse genutzt werden kann (`WeakRef`). Schwache Referenzen werden vom Garbage Collector bei der Entscheidung, ob ein Objekt aus dem Objektspeicher gelöscht wird, nicht berücksichtigt. D.h. der Garbage Collector löscht alle Objekte, auf die entweder überhaupt keine oder nur noch schwache Referenzen bestehen.
Bevor ein nur noch schwach referenziertes Objekt aus dem Objektspeicher gelöscht wird, wird eine spezielle Methode der Klasse `WeakRef` aufgerufen (`finalize`). Dadurch wird es möglich, vor der endgültigen Löschung des Objektes benutzerdefinierte Operationen auszuführen. Dies erweist sich dann als nützlich, wenn z.B. vor dem Löschen des Objekts eine Synchronisation mit externen Diensten stattfinden muß. Bezieht sich ein schwach referenziertes Objekt beispielsweise auf Inhalte einer relationalen Datenbank, kann der Fall auftreten, daß das Objekt zwar nicht mehr stark referenziert wird, sein aktueller Zustand aber noch in die Datenbank übernommen werden muß. Mit Hilfe der von der Klasse `Weakref` bereitgestellten `finalize`-Methode kann automatisch die eventuell noch ausstehende Änderung des Datenbankinhalts veranlaßt werden, bevor der vom Objekt belegte Speicher freigegeben wird.
Die Verwendung schwacher Referenzen gestattet außerdem die Realisierung von Datenstrukturen, die für die Verwaltung von Objekten notwendig sein können (z.B. eine Liste offener Ressourcen, vgl. Abschnitt 5.1.3), die aber eine Garbage Collection dieser Objekte nicht verhindern dürfen, indem sie die betreffenden Objekte stark referenzieren. Ausführliche Betrachtungen über Eigenschaften und Einsatzzwecke schwacher Referenzen findet man z.B. bei Jones und Lins [JL96, S.261 ff].

5.2 Generierung von Klassen für den Datenbankzugriff

Dieser Abschnitt beschäftigt sich mit zwei grundsätzlichen Fragen der zu entwerfenden Datenbankschnittstelle:

- Wie sollen die Objekte beschaffen sein, durch die Datenbankinhalte repräsentiert werden (Abschnitt 5.2.1)?
- Wie werden diese Objekte erzeugt (Abschnitt 5.2.2)?

Für die Klassenerzeugung wird auf Daten über das Datenbankschema zurückgegriffen (Metadaten), die das Datenbanksystem zur Verfügung stellt (Abschnitt 5.2.3). In diesen und allen weiteren Abschnitten werden wir den Entwurfsvorgang und später auch die Implementierung des Programmpakets mit den Mitteln der *Unified Modeling Language* veranschaulichen. Diese Sprache führt die unterschiedlichen Ansätze für den objektorientierten Programm- und Systementwurf von Booch [Boo94], Jacobson [Jac92] und Rumbaugh [RBP+91] zu einer einheitlichen Entwurfmethodik zusammen (Details zu UML z.B. in [UML97a, UML97b]).

5.2.1 Objekte für den direkten Zugriff auf relationale Daten

In einer relationalen Datenbank werden alle Daten in Form von Tabellen gespeichert. Die Spalten einer Tabelle repräsentieren dabei jeweils ein Attribut der zu speichernden Daten. Ein Datensatz entspricht gerade einer Zeile einer solchen Tabelle, die für jede Spalte, also für jedes Attribut, genau einen konkreten Wert enthält. Als Beispiel möge im folgenden eine primitive Tabelle zur Verwaltung von Personendaten dienen, die lediglich den Vornamen, Nachnamen, das Geburtsdatum von Personen und eine Personalnummer enthält. Die "Objekte", die in solch einer Tabelle gespeichert werden, sind Personen — oder präziser, repräsentieren Personen.

Einfache Attribute

Sollen im objektorientierten System Personen repräsentiert werden, wobei wir uns auf die Daten stützen, die in der relationalen Datenbank verfügbar sind, so wird eine Klasse benötigt, zu der Personenobjekte gehören. Die Objekte dieser Klasse sollten mindestens dieselben Attribute aufweisen wie die entsprechenden Datensätze in der Datenbank. Diese Attribute werden als *Slots* der zugehörigen Klasse modelliert.
Im nächsten Schritt gilt es, das Verhalten eines solchen "Datensatzobjekts" zu modellieren. Hier zeigt sich der erste wesentliche Unterschied zwischen dem relationalen und dem objektorientierten Modell. Die "Objekte", die in einer relationalen Datenbank gespeichert werden, besitzen kein eigenes Verhalten. Sie bestehen nur aus Daten, die mit Hilfe des Datenbanksystems manipuliert werden. Allerdings existieren einige typische Datenbankoperationen, die sich sehr wohl dem Verhalten eines einzelnen Objekts zurechnen lassen.

- Das **Verändern** der Werte eines Datensatzes ist z.B. vergleichbar mit dem Setzen von Attributwerten eines Objekts.
- Ein existierendes Objekt kann **gelöscht** werden.
- Objekte können **neu erzeugt** werden.

Während das Verändern von Attributwerten eine Standardoperation in objektorientierten Systemen ist, ist für das Löschen eine eigene Methode erforderlich. Durch den Aufruf dieser `delete`-Methode (vgl. Abbildung 5.1) soll dafür gesorgt werden, daß der zu einem Objekt korrespondierende Datensatz aus der Datenbank gelöscht wird. Ein Löschen des Objektes aus dem Objektspeicher läßt sich damit aber nicht erzwingen. Die Konsequenzen, die sich aus letzterem Effekt ergeben, werden in Abschnitt 5.3.4 erörtert. Eine `undelete-` oder `reuse`-Methode kann sinnvoll sein, um einen versehentlichen Löschvorgang wieder rückgängig zu machen. Da das Objekt nach dem Aufruf der `delete`-Methode weiterhin im Objektspeicher existiert, bietet sich ferner eine Methode an, durch deren Aufruf sich feststellen läßt, ob ein Objekt bereits "gelöscht" wurde.
Um nicht bei jeder noch so kleinen Veränderung des Objektzustandes mit der relationalen Datenbank kommunizieren zu müssen, speichert das Objekt jeweils seinen aktuellen Zustand und schreibt ihn erst beim Aufruf einer `flush`-Methode in die Datenbank. Die hier beschriebenen Eigenschaften eines solchen Datensatzobjekts finden sich in Abbildung 5.1 in den beiden Klassen `Record` und `Person`. Die Klasse `Person` enthält dabei nur die Attribute und Methoden, die spezifisch für Personen sind. Die Methoden zum Speichern, Löschen und Wiederverwenden treten nicht nur bei Personen auf, sondern sie sind bei allen anderen, datenbankspezifischen

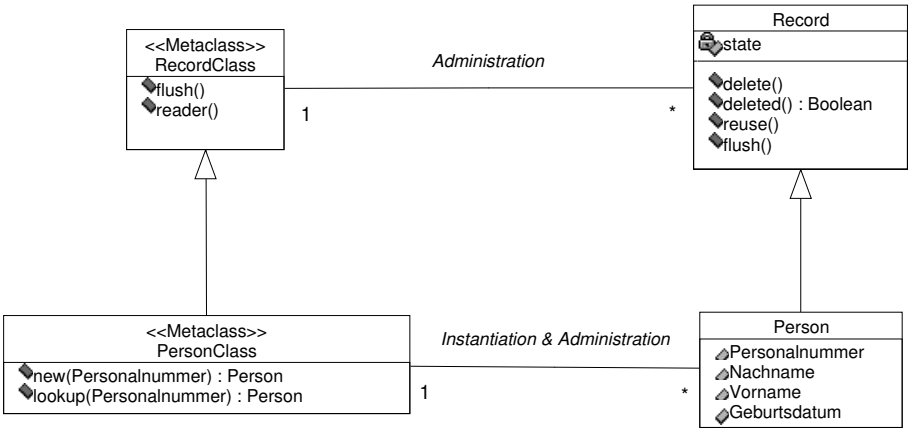

Abbildung 5.1: Basisklassen zur Manipulation von Datensätzen

Objekten ebenfalls anzutreffen. Sie werden daher in einer Superklasse definiert, von der alle Klassen für konkrete Datensätze erben. Die Symbole vor den Attribut- und Methodennamen werden vom eingesetzten Modellierungstool (`Rational Rose`) dazu verwendet, um die Zugriffsberechtigung anzuzeigen. Ein einfacher schräger Balken steht für öffentliche Attribute oder Methoden. Ein Schloß vor dem Balken kennzeichnet private Attribute und Methoden, die nur innerhalb des Objekts zugänglich sind. Ein Schlüssel kennzeichnet in späteren Diagrammen solche Attribute und Methoden, die nur von "befreundeten" Objekten, nicht aber von beliebigen Objekten aus zugreifbar sind oder sein sollen.
Ein Verhalten, das in den bisher beschriebenen Klassen noch nicht enthalten ist, ist die Erzeugung neuer Objekte. Wie im Abschnitt 5.1.1 bereits erwähnt, ist die Erzeugung neuer Objekte nicht ein Verhalten der Objekte selbst, sondern ihrer Klasse. Die nötigen Methoden werden folglich in einer Metaklasse definiert. Abbildung 5.1 zeigt, welche Operationen dabei neben der reinen Objekterzeugung mit der `new`-Methode auftreten können. Da die zu erzeugenden Objekte zwar immer neu aus der Sicht des objektorientierten Systems sind, aber tatsächlich bereits in der Datenbank vorhanden sein können, ist zusätzlich zur `new`-Methode eine zweite Methode zur Objekterzeugung vorgesehen. Diese legt zwar ebenfalls ein neues Objekt im Objektspeicher an, sucht dieses aber zunächst in der Datenbank und initialisiert es mit Werten aus der Datenbank. Da diese beiden Methoden Objekte einer spezifischen Klasse generieren, werden sie auch in einer spezifischen Metaklasse definiert (im Beispiel: `PersonClass`). Außerdem benötigen diese beiden Methoden Parameter, anhand derer sich feststellen läst, ob ein Objekt bereits in der Datenbank existiert, im Beispiel die `Personalnummer` der Person. Diese Parameter sind spezifisch für die zu repräsentierenden Daten. Dies ist ein weiterer Grund, weswegen die Methoden zur Erzeugung neuer Datensatzobjekte Teil einer konkreten Metaklasse und nicht einer abstrakten Superklasse sind. Andere Operationen wie das Speichern der Zustände aller Objekte einer Klasse in der Datenbank (`flush`) oder das Anlegen und Initialisieren aller in der Datenbank auffindbaren Objekte einer Klasse, ist nicht spezifisch für konkrete Klassen und wird daher in einer Superklasse definiert.
Wie die Kommunikation mit Datensatzobjekten innerhalb eines Anwendungsprogramms aussehen kann, demonstriert das Interaktionsdiagramm in Abbildung 5.2. Dort wird ein Objekt `p1` erzeugt, mit Werten versehen und dann in der Datenbank gespeichert. Ein anderes Objekt

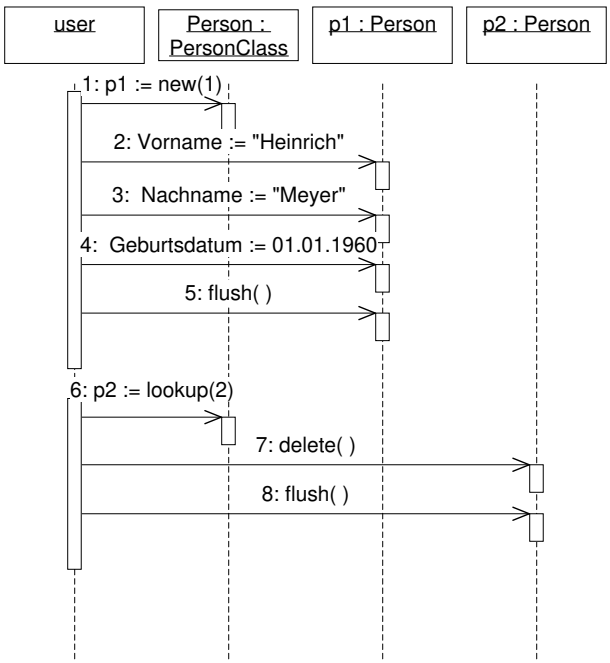

Abbildung 5.2: Manipulation von Datensatzobjekten

`p2` wird mit `lookup` aus der Datenbank gelesen und anschließend in der Datenbank gelöscht. Bei diesem und allen folgenden Sequenzdiagrammen ist folgende Abweichung von den Konventionen der UML zu beachten. Ein dicker senkrechter Balken kennzeichnet nicht die Lebensdauer eines Objektes, sondern die Dauer seiner Aktivität. Das eingesetzte Modellierungstool (RATIONAL ROSE) verfügte zum Zeitpunkt der Modellerstellung über keine Möglichkeit, die Erzeugung und Geamtlebensdauer eines Objekts zu veranschaulichen.

Beziehungen zwischen Datensätzen

Im vorigen Abschnitt wird beschrieben, wie Spalten einer relationalen Tabelle auf Attribute von Objekten abgebildet werden können. Nun verfügen Tabellen in einem Datenbankschema neben einfachen Attributen zur direkten Beschreibung von Eigenschaften der zu speichernden Objekte meistens über weitere Attribute, die eine besondere Aufgabe haben: Diese Fremdschlüsselattribute beschreiben keine Eigenschaft eines Objekts, sondern sie identifizieren eindeutig ein zweites Objekt, das zum ersten Objekt in einer Beziehung steht.
Zur Erläuterung ein einfaches Beispiel: Jede unserer Personen aus dem vorigen Abschnitt arbeite in einer bestimmten Abteilung eines Unternehmens. Die Abteilung habe einen Namen und eine eindeutige Nummer. Diese Beziehung oder Assoziation läßt sich durch das UML-Klassendiagramm in Abbildung 5.3 grafisch darstellen. Die Assoziation `Arbeit` wird in einer relationalen Datenbank durch die Einführung eines zusätzlichen Attributes `Arbeitsplatz` in der Personen-Tabelle realisiert. Der Wert dieses Attributes identifiziert die Abteilung, in der die betreffende Person arbeitet. Da eine Abteilung durch ihre Nummer eindeutig bestimmt werden kann, diese Nummer also als Primärschlüssel fungiert, werden in der Arbeitsplatz-Spalte der Personen-Tabelle gerade diese Abteilungsnummern verwendet, um die Assoziation

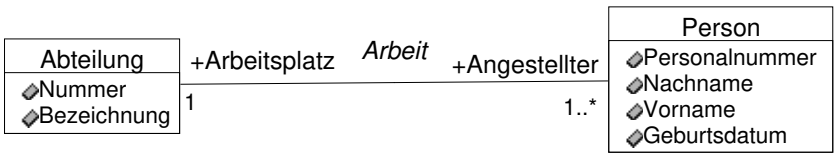

Abbildung 5.3: Assoziation zwischen Datensätzen

einer Person mit einer Abteilung zu repräsentieren. Während in einer relationalen Datenbank Fremdschlüssel eingeführt werden müssen, um Assoziationen zwischen Datensätzen darzustellen, sind in objektorientierten Systemen keine derartigen Hilfskonstruktionen nötig. Dort kann ein Objekt direkte Referenzen auf andere assoziierte Objekte enthalten.
In unserem Beispiel besitzt dann die Klasse Person einen Slot "Arbeitsplatz", dessen Wert ein Objekt der Klasse Abteilung ist. Umgekehrt arbeiten in jeder Abteilung eine Menge von Personen. Diese Menge läßt sich als Slot der Klasse Abteilung darstellen. Abbildung 5.4 zeigt das entsprechend erweiterte Klassendiagramm. Im Beispiel wird eine 1:N-Beziehung verwen-

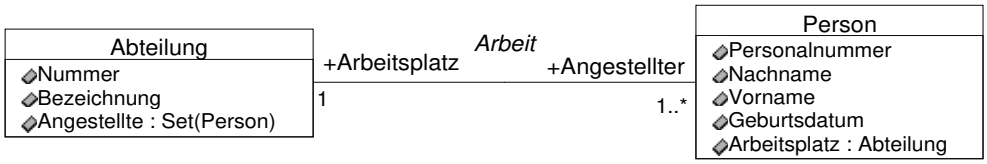

Abbildung 5.4: Implementierung von Assoziationen durch Objektreferenzen

det, wie sie sehr häufig in der Datenmodellierung anzutreffen ist. Jeder Angestellte arbeitet in genau einer Abteilung und in jeder Abteilung arbeiten N verschiedene Angestellte. Weitere Typen von Assoziationen, wie 1:1-, M:N-Beziehungen oder Assoziationen zwischen mehr als zwei Klassen lassen sich ebenfalls sowohl in der objektorientierten als auch in der relationalen Welt repräsentieren.

1:1-Assoziationen werden in relationalen Datenbanken dadurch dargestellt, daß ein Fremdschlüsselattribut einer Tabelle eindeutig sein muß, d.h. keine zwei Datensätze dieser Tabelle dürfen denselben Fremdschlüsselwert besitzen. Auf der objektorientierten Seite besitzen dann beide beteiligten Klassen einen Slot, der genau ein Objekt der jeweils anderen Klasse enthalten kann. 1:1-Assoziationen können in relationalen Datenbanken alternativ auch durch eine einzige Tabelle repräsentiert werden, die alle Attribute der beiden beteiligten Klassen in sich vereinigt. Im objektorientierten System würde zu solch einer Tabelle auch nur eine Klasse generiert werden.

N:M-Assoziationen werden in der Datenbank durch zusätzliche Beziehungstabellen dargestellt, die Fremdschlüsselattribute für die beiden beteiligten Relationen enthalten. Im ersten Ansatz werden für diese Beziehungstabellen im objektorientierten System analog "Beziehungsklassen" generiert, deren Objekte jeweils Referenzen auf Objekte der beteiligten Klassen enthalten. In einem zweiten Schritt könnten anstelle der Beziehungsklasse

Slots in den an der Beziehung teilnehmenden Klassen erzeugt werden, die eine Menge von Objekten der jeweils anderen Klasse enthalten.

Assoziationen zwischen mehr als zwei Klassen repräsentiert man im relationalen Modell in ähnlicher Weise wie N:M-Assoziationen durch separate Beziehungstabellen. Im objektorientierten System werden solche Assoziationen ebenfalls durch eigene Assoziationsklassen dargestellt.

5.2.2 Erzeugung spezifischer Datensatzklassen

Aus den im Abschnitt 5.2.1 vorgestellten Eigenschaften, die eine Datensatzklasse besitzen soll, lassen sich direkt Anforderungen für eine Generierung solcher Klassen ableiten. Zentrale

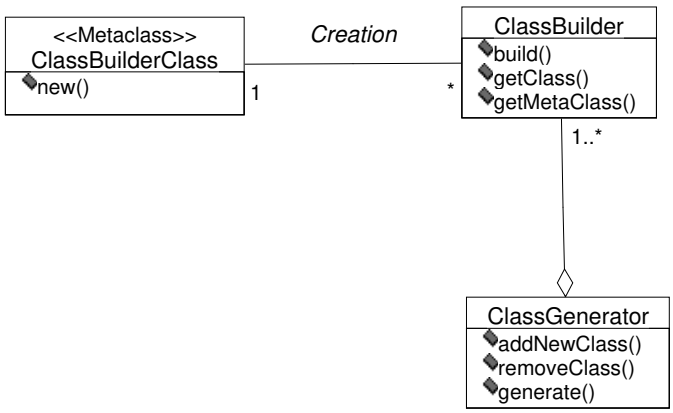

Abbildung 5.5: Zur Generierung von Datensatzklassen erforderliche Basisklassen

Instanz für diesen Prozeß ist ein spezieller Generator (vgl. Abbildung 5.5). Dieser erhält Eingaben, durch welche die neu zu erzeugenden Klassen beschrieben werden. Da in dieser Arbeit Klassen erzeugt werden sollen, die auf Strukturen in einer relationalen Datenbank basieren, muß dem Generator mindestens mitgeteilt werden, zu welchen Tabellen korrespondierende Klassen erzeugt werden sollen. Im ersten Ansatz könnte man sich also für diese Generierung eine Methode vorstellen, die mit dem Namen einer Datenbanktabelle parametrisiert wird und die dann die dazugehörige Klasse erzeugt. Diese einfache Methode ist aber noch nicht hinreichend. Wenn der Generator nur den Namen dieser einen Klasse kennt, kann er die Attribute der Tabelle bereits auf entsprechende objektorientierte Attribute abbilden. Damit stellt er die im Abschnitt 5.2.1 beschriebene Funktionalität für den Zugriff auf einfache Attribute sicher. Mit Hilfe von Metadaten aus der Datenbank kann der Generator eventuell sogar feststellen, ob die erzeugte Klasse in Beziehung zu anderen Tabellen steht. Allerdings verfügt der Generator über keine Daten darüber, ob er diese Beziehungen zu anderen Tabellen auch nachbilden soll, bzw. ob der Programmierer dies wünscht. Um also auch die in Abschnitt 5.2.1 erläuterten Assoziationen in die objektorientierte Welt abzubilden, benötigt der Generator Angaben darüber, welche dieser Assoziationen er abbilden soll. Daher wird für den Generierungsprozeß ein zweistufiger Ansatz gewählt (vgl. Abbildung 5.6).

Im ersten Schritt wird der Generator konfiguriert, d.h. ihm wird mitgeteilt, welche Tabellen

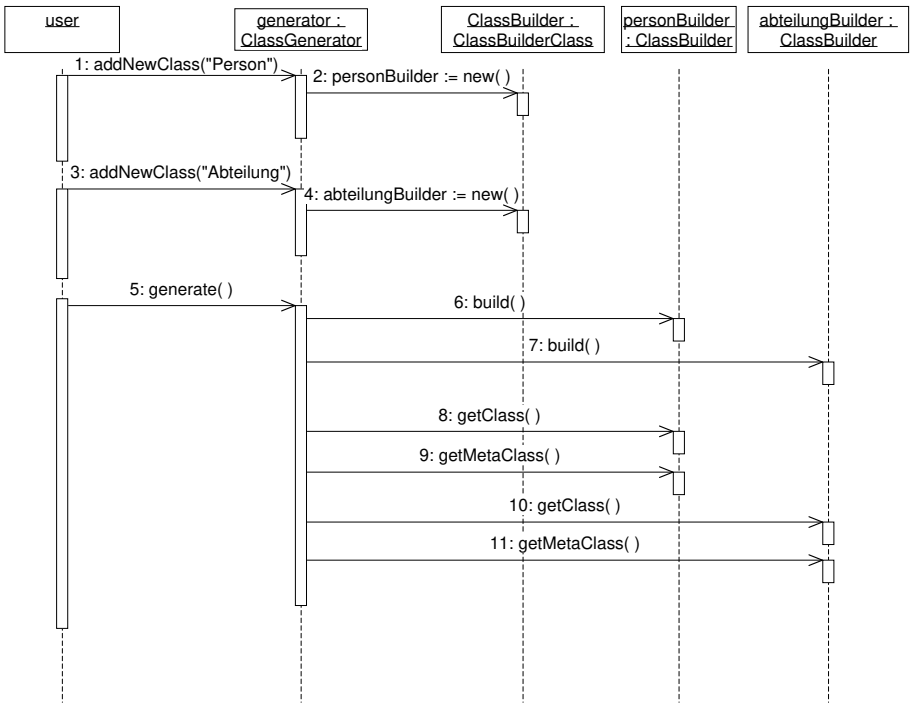

Abbildung 5.6: Generierung von Datensatzklassen

der relationalen Datenbank auf Klassen abgebildet werden sollen. Hierfür enthält der Generator eine `addNewClass`-Methode. Beim Aufruf dieser Methode erzeugt sich der Generator ein spezielles Objekt, einen sogenannten `ClassBuilder`, der genau für diese Tabelle eine zugehörige Klasse erzeugen kann (vgl. in Abbildung 5.6 die Schritte 1–4). Erst wenn alle Tabellen, für die eine Klassengenerierung gewünscht ist, beim Generator registriert sind, wird der eigentliche Generatorprozeß durch Aufruf seiner `generate`-Methode gestartet (Schritt 5 in Abbildung 5.6). Nun werden zu allen registrierten Tabellen entsprechende Klassen durch Aufruf der `build`-Methoden der jeweiligen `ClassBuilder` erzeugt. Stößt ein `ClassBuilder` dabei auf eine Beziehung zu einer anderen Klasse, so kann er über eine Referenz auf den Klassengenerator feststellen, ob die assoziierte Klasse ebenfalls dem Klassengenerator bekannt ist. Ist dies der Fall, so wird die Klasse um Methoden und Slots zur Behandlung dieser Assoziation erweitert. Ist die assoziierte Klasse nicht beim Generator registriert, so werden keine entsprechenden "Assoziationsmethoden" oder -attribute erzeugt und lediglich das Fremdschlüsselattribut aus der relationalen Datenbank wird unverändert abgebildet. Ist die Klassengenerierung abgeschlossen, so kann auf die neuen Datensatzklassen und ihre Metaklassen über die `getClass`- und die `getMetaClass`-Methode des jeweiligen classBuilders zugegriffen werden.

5.2.3 Metadaten für die Klassengenerierung

Während der Erzeugung von Datensatzklassen zu Datenbankrelationen wird auf Daten *über* die abzubildenden Strukturen zurückgegriffen. Da die bisher beschriebenen Methoden der Generatorklasse als Eingabe lediglich den Namen einer abzubildenden Tabelle als Parameter

übergeben bekommen, stellt sich die Frage, woher all die Daten kommen, mit deren Hilfe beispielsweise die Namen und Datentypen von Slots und Methoden bestimmt werden. Für die Implementierung von Assoziationen müssen ebenfalls Daten zur Verfügung stehen, die Rückschlüsse auf Beziehungen zwischen Datenbanktabellen erlauben. All diese sogenannten Metadaten speichert ein relationales Datenbanksystem in einem eigenen Datenbankschema, dem *Data Dictionary* (vgl. Kapitel 2.5 in DBH87). Die meisten Programmierschnittstellen für relationale Datenbanken stellen ihren Benutzern spezielle Funktionen oder Methoden zur Verfügung, mit deren Hilfe sich Daten aus dem *Data Dictionary* extrahieren lassen. Für das TYCOON-System wurde zu diesem Zweck die Klasse `SQLConnection` der bereits vorhandenen TYCOON-SQL-Schnittstelle [Sku98] um eine Methode `describeTable` erweitert. Diese liefert die wichtigsten Metadaten zu einer Datenbanktabelle. `describeTable` generiert ein Objekt der Klasse `SQLTableDescription`, das unter anderem folgende Metadaten liefert:

- die Namen aller Spalten der Tabelle
- eine Beschreibung des SQL-Datentyps jeder Spalte und der korrespondierenden TYCOON-Klasse
- eine Beschreibung aller Schlüssel einer Tabelle, darunter
  - die Attribute des Primärschlüssels, anhand dessen ein Datensatz eindeutig identifiziert werden kann,
  - weitere eindeutige Schlüssel, die für die Erkennung von 1:1-Assoziationen herangezogen werden können,
  - Fremdschlüssel und ihre Attribute, die Hinweise auf Assoziationen liefern können.

In relationalen Datenbanken existiert in der Regel kein Mechanismus zur expliziten Auszeichnung von Schlüsseln. Aber nahezu jedes moderne relationale Datenbanksystem verfügt über die Möglichkeit, Integritätsbedingungen für Tabellen zu definieren. Das Datenbanksystem überwacht die Einhaltung dieser Integritätsbedingungen. Beispiele für solche Bedingungen sind (vgl. [MU97, S. 126-132]):

- Ein Attributwert darf in einer Tabelle höchstens einmal vorkommen (`unique`).
- Ein Attribut muß einen Wert enthalten (`not null`).
- Eine Menge von Attributen bildet den Primärschlüssel, d.h. für die einzelnen Attribute gilt `not null` und die Kombination der Attribute muß `unique` sein.
- Ein Attributwert muß einen existierenden Datensatz in einer anderen Tabelle identifizieren (`references`).

Ein sorgfältig implementiertes Datenbankschema enthält solche Bedingungen, um die Integrität der gespeicherten Daten sicherzustellen. Da das Datenbanksystem über die Einhaltung dieser Bedingungen wachen muß, werden sie im *Data Dictionary* gespeichert. Aus diesen Daten wiederum lassen sich Informationen für die Klassengenerierung gewinnen.

Primärschlüssel: Die Attributwerte des Primärschlüssels dienen in der Datenbank dazu, einen Datensatz eindeutig zu charakterisieren. Diese Attributwerte können also als Parameter einer `lookup`-Methode verwendet werden, um zu einem Datensatz ein korrespondierendes Objekt zu erzeugen.

Referentielle Integritätsbedingungen: Eine referentielle Integritätsbedingung sorgt dafür, daß Attribute nur solche Werte annehmen können, zu denen in anderen Tabellen korrespondierende Datensätze existieren. Zu der anderen Tabelle besteht demnach eine Beziehung.

Kombinationen von Integritätsbedingungen: Müssen die Werte eines Fremdschlüssels tabellenweit eindeutig (`unique`) sein, so deutet das auf eine 1:1-Beziehung zur referenzierten Tabelle hin. Andernfalls läßt sich nur auf eine 1:N-Beziehung schließen, da dann mehrere Einträge der Tabelle auf denselben Datensatz in der referenzierten Tabelle verweisen können. Referentielle Integritätsbedingungen dürfen nur Verweise auf eindeutige Attribute der Zieltabelle enthalten. Daraus ergibt sich, daß nicht etwa eine N:M-Beziehung gefunden wurde.
Während sich Daten über 1:1- oder 1:N-Beziehungen direkt aus den Integritätsbedingungen zweier Tabellen ableiten lassen, ist für komplexere Beziehungen ein indirektes Vorgehen nötig. N:M-Beziehungen oder k-näre Beziehungen ($k > 2$) werden in relationalen Datenbanken durch separate Tabellen nachgebildet. Besteht der Primärschlüssel einer Tabelle ausschließlich aus Fremdschlüsseln, so deutet dies darauf hin, daß diese Tabelle zur Darstellung einer solchen komplexen Assoziation dient. Eine Instanz einer komplexen Assoziation wird gerade durch die Primärschlüsselwerte der an der Assoziation beteiligten Objekte identifiziert.

Die oben genannten Rückschlüsse auf das dem Datenbankschema zugrundeliegende Datenmodell lassen sich nur dann ziehen, wenn die entsprechenden Integritätsbedingungen auch bei der Implementierung des Datenbankschemas formuliert worden sind. Wurde dies versäumt, so besteht für den Generator keine Chance, Assoziationen zwischen Tabellen automatisch objektorientiert nachzubilden. Klassen für den Zugriff auf die einzelnen Attribute einer Relation können zwar dennoch erzeugt werden, die Semantik eines Fremdschlüsselattributs ist aber nicht zu erkennen. Das Attribut darf dann beliebige Werte annehmen und weder die Datenbank, noch das Datensatzobjekt kann die Gültigkeit eines Fremdschlüsselwertes überprüfen. Bei einer halbautomatischen Klassengenerierung könnte der Anwender dieses Problem umgehen, indem er den Klassengenerator mit zusätzlichen Hilfsdaten über solche Beziehungen parametrisiert, die nicht aus den Metadaten der Datenbank hervorgehen.
Sind referentielle Integritätsbedingungen definiert, so kann deren objektorientierte Umsetzung, das Verständnis und die Handhabung des ursprünglichen Datenmodells verbessern. Beim direkten Zugriff auf die Datenbank, muß der Anwender bei jeder Änderungsoperation prüfen, ob alle Integritätsbedingungen eingehalten werden. Beim Einfügen neuer Daten muß er z.B. dafür sorgen, daß alle Fremdschlüssel auf tatsächlich existierende Datensätze verweisen. Wird aber anstelle der Fremdschlüsselattribute in einem Objekt direkt das durch den Fremdschlüssel bezeichnete Objekt benutzt, so werden beim Speichern in der Datenbank die Fremdschlüsselattribute implizit korrekt gesetzt, nämlich auf den Primärschlüsselwert des referenzierten Objekts.
Die Umsetzung von referentiellen Integritätsbedingungen auf Objektbeziehungen befreit den Anwendungsentwickler damit von separaten Konsistenzüberprüfungen, die er bei direkter Datenbankprogrammierung einsetzen müßte, um Fehlermeldungen seitens der Datenbank zu vermeiden.

5.3 Synchronisation und Caching

Im Abschnitt 5.2 wird beschrieben, über welche grundlegenden Eigenschaften die Objekte verfügen sollen, die die Datensätze einer relationalen Datenbank und ihre Beziehungen untereinander repräsentieren. Um nicht bei jedem Zugriff auf die Attributwerte eines Datensatzes mit der Datenbank kommunizieren zu müssen, werden die Attributwerte zu definierten Zeitpunkten aus der Datenbank gelesen und dann innerhalb des Objekts gespeichert (vgl. Abschnitt 5.3.1).
Das bedeutet aber, daß diejenigen Daten, die ursprünglich ausschließlich innerhalb der Datenbank verwaltet worden sind, nun parallel dazu innerhalb eines Objektes in einem Objektspeicher existieren können. Dadurch, daß im objektorientierten System somit eine Kopie der Daten erzeugt wird, die in einer relationalen Datenbank bereits enthalten sind, entsteht Redundanz. Es stellt sich die Frage, wie diese beiden Instanzen der Daten miteinander synchronisiert werden können (Abschnitt 5.3.2).
Dabei ergibt sich die Notwendigkeit, eine eindeutigen Zuordnung eines Datensatzobjekts zu einem Datensatz in der Datenbank zu treffen (Abschnitte 5.3.3 und 5.3.4). Es müssen Vorkehrungen getroffen werden, die verhindern, daß sich diese beiden parallel existierenden Repräsentationen derselben Daten unabhängig voneinander ändern können. Ändert sich der Zustand der Datenbank, so muß diese Änderung im objektorientierten System nachvollzogen werden und umgekehrt.
Da in dieser Arbeit ein objektorientiertes System zum Einsatz kommt, das außerdem über ein eigenes Persistenzkonzept verfügt, ergeben sich einige besonder Fragestellungen hinsichtlich der Lebensdauer (Abschnitt 5.3.5) und des Verhaltens persistenter Objekte, die von externen Diensten, wie hier einer relationalen Datenbank, abhängen (Abschnitt 5.3.6).
Weiterhin sind bei der Synchronisation von objektorientiertem System und relationaler Datenbank Abhängigkeiten zwischen Objekten zu berücksichtigen, die sich aus referentiellen Integritätsbedingungen bzw. Objektbeziehungen ergeben (Abschnitt 5.3.7).

5.3.1 Implizites Caching

Um Daten aus einer relationalen Datenbank zu lesen, muß eine SQL-Anfrage an die Datenbank gestellt werden. Das Anfrageergebnis wird in Form eines sogenannten *Cursors* übertragen. Dies ist ein Objekt, das den Zugriff auf die Inhalte jeweils einer Zeile der Ergebnistabelle gestattet. Die Daten einer solchen Zeile werden dann vom anfragenden Programm gelesen. Danach wird der Cursor auf die nächste Ergebniszeile positioniert. Dies wird so lange wiederholt, bis alle Daten der Ergebnistabelle gelesen wurden. Der Ablauf für beispielsweise den lesenden Zugriff auf ein Attribut eines Datensatzobjekts wäre demnach der folgende:

1. Abschicken der Anfrage an den Datenbankserver.
2. Schalten des als Antwort erhaltenen Cursors auf die erste (einzige) Zeile des Anfrageergebnisses.
3. Lesen des interessierenden Attributwertes.
4. Schließen des Cursors.
5. Rückgabe des Attributwertes.

Bei diesem Ablauf ergeben sich noch keine Synchronisationsprobleme mit der Datenbank, weil bei jedem erneuten Zugriff dieser Ablauf wiederholt wird und damit im Anwendungsprogramm zu jeder Zeit der aktuelle Datenbankzustand sichtbar ist. Bei jedem Attributzugriff ist also eine Kommunikation mit dem Datenbankserver nötig. Da sich dieser häufig auf einem anderen Rechner befindet, vergeht eine gewisse Zeit, bis die Anfrage den Server über ein Netzwerk erreicht hat. Das Datenbanksystem bearbeitet die Anfrage und sendet dem Anwendungsprogramm seine Bereitschaft, die Ergebnisse zu übertragen. Die Ergebnisse kann das Anwendungsprogramm nun Zeile für Zeile anfordern. Auch wenn nur eine Zeile übertragen werden soll, sind somit mindestens zwei Interaktionen zwischen dem Anwendungsprogramm und dem Datenbankserver nötig, eine für das Abschicken der Anfrage, die zweite für die Übertragung des Ergebnisses.
Wenn ein Anwendungsprogramm nun sehr häufig in dieser Weise auf Datenbankinhalte zugreift, so wird für jeden Zugriff auf ein Attribut eines Datensatzes das oben beschriebene Verfahren durchlaufen. Dadurch entsteht eine erhebliche Anfragelast an den Datenbankserver, eine hohe Belastung des Kommunikationsnetzwerks und damit letztendlich ein Geschwindigkeitsverlust beim Ablauf des Anwendungsprogramms durch die entstehenden Wartezeiten auf Antworten des Datenbankservers.
Aus diesem Grund wird in dieser Arbeit ein Zugriffsmodell gewählt, das die Zahl der Datenbankzugriffe reduziert, allerdings einigen administrativen Aufwand auf der Clientseite, also auf der Seite des Anwendungsprogramms, erfordert. Auf die Attributwerte eines Objektes wird nicht über Methoden zugegriffen, die die oben beschriebene Kommunikation mit der Datenbank bei jedem Attributzugriff ausführen, sondern bei der Erzeugung eines Datensatzobjektes werden die Werte all seiner Attribute mit einem einzigen Datenbankzugriff gelesen und im Objekt gespeichert. Lesende oder schreibende Zugriffe auf diese Attribute verändern den internen Zustand des Datensatzobjekts, bewirken aber keine weitere Kommunikation mit der Datenbank. Der gesamte Zustand des Datensatzobjektes wird in einem einzigen SQL-Befehl auf einmal zu definierten Zeitpunkten in die Datenbank geschrieben (vgl. dazu den Abschnitt 5.3.2).
Dadurch, daß die Werte eines Datensatzes innerhalb eines Objektes für weitere Zugriffe zwischengespeichert werden, wird eine Art Cache-Speicher realisiert. Da dies mehr ein Seiteneffekt der Implementierung der Datensatzobjekte ist, als ein vollständiger Cache mit eigener Speicherverwaltung, sei dieser Mechanismus mit *implizitem Caching* bezeichnet. Der dadurch erreichten Reduktion des Kommunikationsbedarfs zwischen Datenbank und objektorientierter Anwendung steht nun der Bedarf nach Synchronisation zwischen dem Datensatz in der Datenbank und dem Datensatzobjekt im objektorientierten System gegenüber.

5.3.2 Synchronisation

Wenn nicht jeder Zugriff auf ein Attribut eines Datensatzobjekts unmittelbar zu einer Kommunikation mit der Datenbank führt, so müssen Zeitpunkte definiert werden, zu denen der Zustand des Datensatzobjektes im objektorientierten System mit dem des Datensatzes in der relationalen Datenbank abgeglichen wird. Zum einen kann der Benutzer selber ein Schreiben des aktuellen Objektzustandes in die Datenbank veranlassen, indem er die `flush`-Methode des Datensatzobjekts aufruft. Zum anderen werden immer dann die Inhalte aller Datensatzobjekte, die vom Anwender verändert worden sind in der Datenbank gespeichert, wenn folgende Ereignisse eintreten:

- Die laufende Datenbanktransaktion wird durch den Benutzer beendet.

- Die Verbindung zur Datenbank wird geschlossen.
- Der Systemzustand wird gespeichert (vgl. Abschnitt 5.1.3 über Persistenz).

Beim Speichern des aktuellen Objektzustandes in der Datenbank ist es sinnvoll, nur diejenigen Objekte zu speichern, die tatsächlich verändert wurden. Ein Schreiben von Objektinhalten, die vom Anwendungsprogramm überhaupt nicht verändert worden sind, erhöht den Kommunikationsaufwand zwischen Anwendung und Datenbankserver unnötig. Dazu müssen die Datensatzobjekte Daten darüber speichern, ob sich ihr Zustand seit der letzten Kommunikation mit der Datenbank verändert hat. Hierzu verfügt jedes Datensatzobjekt über ein privates Attribut `state` (vgl. Abb. 5.1) anhand dessen festgestellt werden kann, welche Operationen in der Datenbank nötig sind, um den Objektzustand auf die Datenbank abzubilden. Wurde das Objekt nicht verändert, findet auch kein Datenbankzugriff statt.
Bei dieser externen Form der Synchronisation in Form eines Abgleichs zwischen Inhalten eines Objektsystems und einer Datenbank sind auch Effekte der internen Synchronisation des Mehrbenutzerzugriffs der Datenbank zu berücksichtigen. Solange das objektorientierte System allein Änderungen auf Datenbankinhalten ausführt, läuft der Abgleich zwischen Datensatzobjekten und der Datenbank reibungslos. Moderne Datenbanksysteme zeichnen sich aber gerade durch ihre Mehrbenutzerfähigkeit aus. Es muß also davon ausgegegangen werden, daß sich die Daten innerhalb der Datenbank durch den Einfluß fremder Anwendungen ändern können, ohne daß das objektorientierte System, das jetzt auf Kopien dieser Daten arbeitet, diese Änderungen nachvollzieht. Eine relationale Datenbank verfügt als typische Server-Anwendung über keine Möglichkeit, alle angeschlossenen Client-Anwendungen über solche Veränderungen zu informieren. Der Server bedient lediglich die Anfragen dieser Anwendungen, kann aber selbst keine Ereignisse auf dem Rechner der jeweiligen Anwendung anstoßen. Das Datenbanksystem kann durch die Vergabe von Sperren verhindern, daß mehrere Benutzer gleichzeitig dieselben Daten verändern. Diese Sperren zur datenbankseitigen Synchronisation paralleler Zugriffe werden aber nach Abschluß einer Datenbanktransaktion wieder freigegeben. Danach stehen dann keine Informationen mehr zur Verfügung, an denen sich erkennen läßt, von welcher Anwendung ein Datensatz zuletzt und in welchem Zustand gelesen worden ist.

Pessimistisches Sperren

Die konservative Methode, um dieses Problem zu lösen, ist das sogenannte pessimistische Sperren. Das bedeutet, ein Anwendungsprogramm sperrt einen Datensatz in der Datenbank unmittelbar nach dem ersten Zugriff und beendet seine Transaktion — und gibt damit die Sperre frei — erst dann, wenn keine weiteren Zugriffe auf Datensatzobjekte mehr stattfinden sollen. Dies kann aber schlimmstenfalls bis zum Ende des Anwendungsprogramms dauern. Läuft das Anwendungsprogramm nun sehr lange (z.B. ein WWW-Server-Prozeß), dann werden einige Datensätze u.U. für sehr lange Zeit nicht freigegeben. In dieser Zeit könnte kein anderer Datenbankbenutzer auf diese Daten zugreifen, was langfristig zu einer Blockierung der Datenbank für andere Benutzer führen würde. Pessimistisches Sperren verhindert zwar die Manipulation der Daten bzw. des *Working Sets* eines Anwendungsprogramms wirksam, ist aber wegen der beschriebenen Auswirkungen auf die Mehrbenutzerfähigkeit des Datenbanksystems nur bei kurz dauernden Transaktionen sinnvoll.

Optimistisches Sperren

Beim optimistischen "Sperren" werden für Datensätze zunächst keine Datenbanksperren angefordert. Alle anderen Datenbanknutzer können die Daten parallel lesen und ändern. Allerdings muß ein Anwendungsprogramm, das einen optimistisch gesperrten Datensatz verändert, überprüfen, ob die beabsichtigte Änderungsoperation noch zulässig ist, sprich der Datensatz zwischenzeitlich nicht von anderen Benutzern verändert wurde. Das optimistische Sperren ist in TYCOON nicht uneingeschränkt sinnvoll, weil die Möglichkeiten für Tests auf Veränderungen in der Datenbank begrenzt sind. Ein Vergleich Attribut für Attribut ist aufwendig, weil dazu der aktuelle Zustand aus der Datenbank gelesen werden muß und mit dem Objektzustand zum Zeitpunkt der letzten `flush`-Operation verglichen werden muß. Dazu müßte jedes Datensatzobjekt seine Attributwerte doppelt speichern. Diesen doppelten Speicherbedarf wird man aber gerade bei sehr großen Datensätzen nicht in Kauf nehmen wollen.
Dieses Problem wird bei einigen kommerziellen *Middleware*-Produkten wie PERSISTENCE durch eine Versionierung der Objekte vermieden. Jeder Datensatz erhält von der *Middleware* eine eigene Versionsnummer, die sich ändert, sobald der Datensatz von einem Nutzer der *Middleware* manipuliert wird. Schreibt ein Anwendungsprogramm seine Änderungen in die Datenbank, so unterrichtet die *Middleware* alle anderen Nutzer von dieser Änderung, bzw. die Anwendungen erkennen diese Änderung an der neuen Versionsnummer der Daten. So existiert pro Anwendungsprogramm nur eine Version des Datensatzes und zusätzlich ein Exemplar in der *Middleware* selbst. Der Preis für diesen Komfort ist der, daß dieser Mechanismus nur für solche Anwendungen funktioniert, die über das *Middleware*-Produkt auf die Datenbank zugreifen.

Das Sperrkonzept für Tycoon

Trotz der Nachteile des optimistischen Sperrverfahrens, kommt auch ein rein pessimistisches Sperren für TYCOON nicht in Frage, da dann ein einmal im TYCOON-System angelegtes Datensatz-Objekt so lange gesperrt bleiben würde, bis es wieder aus dem Objektspeicher entfernt wird. Letzteres passiert aber erst dann, wenn das Objekt von keinem anderen Objekt aus mehr erreichbar ist. Der Zeitpunkt für diese in objektorientierten Systemen übliche *Garbage Collection* ist nicht vorhersehbar und der Freigabezeitpunkt der Datenbanksperren damit nicht eindeutig zu bestimmen. Außerdem lassen sich Datenbanksperren nicht für einzelne Objekte freigeben, sondern nur für alle innerhalb einer Transaktion erzeugten Objekte auf einmal.
Der Anwendung muß also eine Möglichkeit geboten werden, die von ihr produzierten Datenbanksperren freizugeben, ohne dabei die Konsistenz zwischen Datensatzobjekt und Datensatz in der Datenbank zu gefährden. Dies wird gewährleistet, indem alle Datenbanksperren zu den am Anfang dieses Abschnitts genannten Zeitpunkten freigegeben werden. Die Sperren werden erst dann wieder angefordert, wenn erneut auf die Datensatzobjekte zugegriffen wird. Dazu muß also ein geeigneter Mechanismus implementiert werden, der diese Sperroperation veranlaßt, sobald ein neuer Zugriff auf ein Datensatzobjekt stattfindet. Ein Datensatzobjekt wird also nicht für die gesamte Dauer seiner Existenz gesperrt, sondern nur für den Zeitraum, in dem es benutzt wird.
Auch bei diesem Vorgehen sind noch langfristige Sperren möglich, wenn das Anwendungsprogramm aus einer einzigen, langlaufenden Transaktion besteht. Hier ist es Aufgabe des Anwendungsprogrammierers, wie bei anderen Datenbankanwendungen auch, zu entscheiden,

welche Operationen sich als sinnvolle logische Einheit zu einer Transaktion zusammenfassen lassen.

5.3.3 Eindeutige Identifikation von Objekten und Daten

In objektorientierten Systemen besitzt jedes Objekt implizit eine Identität, d.h. es ist von allen anderen Objekten im System unterscheidbar. Ein Objekt kann nicht durch Änderung seines Zustandes zu einem anderen Objekt werden. Um festzustellen, ob zwei Objektreferenzen auf dasselbe Objekt verweisen, gibt es daher in objektorientierten Systemen ein Identitätsprädikat. Dieses liefert immer dann den Wahrheitswert `true`, wenn die beiden übergebenen Objektreferenzen dasselbe Objekt bezeichnen, ansonsten `false`.
Hiervon deutlich zu unterscheiden ist der Test auf Gleichheit oder Äquivalenz zweier Objekte. Dieses Prädikat liefert dann den Wert `true`, wenn die beiden Objektreferenzen auf zwei äquivalente Objekte verweisen. Die Definition dieser Äquivalenz hängt vom jeweiligen objektorientierten System ab und kann häufig, wie z.B. in TYCOON, vom Programmierer selbst definiert werden. Eine mögliche Definition für diese Gleichheit wäre die folgende: "Zwei Objekte sind dann gleich, wenn sie derselben Klasse angehören und ihr Zustand, also die Gesamtheit ihrer Attributwerte, gleich sind."
Relationale Datenbanken verfügen über keinen impliziten Identitätsbegriff. D.h. eine Tabelle kann mehrere Zeilen enthalten, die sich anhand ihrer Werte nicht unterscheiden lassen. Dadurch entsteht ein Problem bei der Übertragung der Daten in ein objektorientiertes System. Zwei Zeilen einer Tabelle, die sich innerhalb der Datenbank nicht voneinander unterscheiden lassen, werden auf zwei Datensatzobjekte abgebildet, die nun im objektorientierten System sehr wohl unterschieden werden können. Dies könnte zu der Annahme verführen, daß sich die Attributwerte dieser beiden Objekte unabhängig voneinander ändern lassen. Werden die geänderten Attributwerte nun in der Datenbank gespeichert, so werden beim Speichern von nur einem der beiden Objekte die beiden ursprünglichen Datenbankzeilen gleichzeitig verändert, weil die Datenbank die beiden Zeilen nicht unterscheidet. Es ließen sich also nicht zwei verschiedene Objektzustände in der Datenbank speichern, sondern der Zustand nur eines Objektes würde in zwei Datenbankzeilen gespeichert. Welcher Zustand nach dem Speichern beider Objektzustände in der Datenbank vorliegt, hängt dann von der zeitlichen Reihenfolge ab, in der die beiden Objektzustände in die Datenbank geschrieben wurden. Auch der Vorgang des Speicherns des Objektzustandes ist nicht unproblematisch. So muß festgestellt werden, auf welche Zeile(n) der Datenbanktabelle sich die Änderungsoperation überhaupt beziehen soll. Dazu müßte ein Datensatzobjekt die Werte der ursprünglich aus der Datenbank gelesenen Tabellenzeile zwischenspeichern, um über diese Werte bei späteren Änderungsoperationen in der Datenbank die zu ändernde(n) Zeilen(n) wiederaufzufinden. Dies würde wiederum den Speicherbedarf eines Datensatzobjektes erhöhen, da neben dem aktuellen Objektzustand auch der letzte in der Datenbank gültige Zustand gespeichert werden muß.
Wir fordern daher von einer Datenbanktabelle, die in das objektorientierte System abgebildet werden soll, daß sie einen Primärschlüssel enthalten muß. Ein Primärschlüssel, eine Kombination von Attributen der Tabelle, dient zur eindeutigen Identifizierung der Zeilen einer Tabelle. Durch die Definition eines solchen eindeutigen Schlüssels in der Datenbank ist es nicht mehr möglich, beim Speichern eines Datensatzobjektes versehentlich mehrere Zeilen der Tabelle zu verändern. Bei Änderungsoperationen reicht die Angabe der Primärschlüsselwerte aus, um die zu ändernde Zeile eindeutig zu identifizieren. Jedem Datensatzobjekt läßt sich damit genau eine Tabellenzeile in der Datenbank zuordnen.

Um die Objektidentität, die im objektorientierten System implizit existiert, auf die relationale Datenbank zu übertragen, sind weitere Maßnahmen nötig. In der relationalen Datenbank ist es möglich, die Werte eines Primärschlüssels zu ändern. Setzte man die Begriffe *Primärschlüssel* und *Objektidentität* gleich, so würde das bedeuten, daß sich durch Änderung eines Primärschlüsselwertes, die Identität eines Objekts ändern könnte. Da dies im objektorientierten System nicht möglich ist, wird eine Änderung des Primärschüssels von Datensatzobjekten im objektorientierten System unterbunden. Das geschieht praktisch dadurch, daß auf die Attribute des Primärschlüssels nur lesend und nicht schreibend zugegriffen werden kann. Der Primärschlüssel eines Datensatzobjekts wird bei dessen Erzeugung angegeben und kann danach nicht mehr verändert werden. Durch die Definition eines Primärschlüssels und die Unterbindung seiner Manipulation im objektorientierten System, ist eine eindeutige Abbildung eines Datensatzobjektes auf eine Zeile einer Datenbanktabelle möglich.
Umgekehrt ist aber noch keine eindeutige Abbildung einer Tabellenzeile auf genau ein Datensatzobjekt möglich. Da die Objektidentität in der objektorientierten Welt unabhängig von Primärschlüsselwerten arbeitet, ist es noch immer möglich, zwei Datensatzobjekte zu erzeugen, die auf denselben Datensatz in der Datenbank verweisen. Um auch diesen Fall zu vermeiden und eine eineindeutige Abbildung zwischen Datensatzobjekt und relationaler Tabellenzeile herzustellen, wird ein explizites Caching eingeführt.

5.3.4 Explizites Caching

Die Erzeung neuer Datensatzobjekte ist ein Verhalten der zugehörigen Klasse und wird daher in der entsprechenden Metaklasse definiert (vgl. Abschnitt 5.2.1). Es ist naheliegend, demselben Objekt, das die Datensatzobjekte erzeugt, auch die Sicherung der Korrespondenz zwischen Objektidentität und Primärschlüsselwerten zu übertragen. Zu diesem Zweck übernimmt eine Datensatzklasse auch die Verwaltung der von ihr erzeugten Objekte, d.h. sie speichert Referenzen auf alle Objekte, die durch Aufruf ihrer Methoden erzeugt wurden. Um nun zu verhindern, daß im objektorientierten System mehrere Datensatzobjekte existieren, die sich auf denselben Datensatz in der Datenbank beziehen, greift die Datensatzklasse auf diese Menge ihrer bereits vorhandenen Datensatzobjekte, ihren "Cache" zurück. Immer, wenn ein neues Datensatzobjekt erzeugt werden soll, überprüft die Datensatzklasse, ob dieses Objekt bereits im Objektspeicher, also auch in ihrem Cache vorhanden ist. Um ein solches Objekt im Cache zu finden, wird das Äquivalenzprädikat für Datensatzobjekte in folgender Weise definiert:

> Zwei Datensatzobjekte sind äquivalent, wenn sie der gleichen Klasse angehören und ihre Primärschlüsselwerte gleich sind.

Wird also versucht, ein neues Datensatzobjekt zu erzeugen und es existiert bereits ein Objekt mit denselben Primärschlüsselwerten, so wird kein neues Objekt erzeugt, sondern das bereits vorhandene Objekt zurückgegeben.
Dieser Mechanismus ist ebenfalls dann von Bedeutung, wenn Datensätze durch Aufruf der `delete`-Methode des zugehörigen Datensatzobjektes aus der Datenbank gelöscht werden. Der Datensatz ist dann zwar nicht mehr in der Datenbank vorhanden, das Datensatz*objekt* existiert aber immer noch. Wird nun ein neuer Datensatz in die Datenbank eingefügt, der die gleichen Primärschlüsselwerte besitzt wie ein vorher gelöschter Datensatz, so wird das "alte" Datensatzobjekt wiederverwendet und kein neues Objekt angelegt.

5.3.5 Lebensdauer eines Datensatzobjektes

Wenn ein einmal angelegtes Datensatzobjekt auch nach der Löschung aus der Datenbank im Objektspeicher weiterexistiert und sogar wiederverwendet werden kann, stellen sich zwei Fragen:

1. Woran erkennt der Benutzer von Datensatzobjekten, daß diese "gelöscht" sind?
2. Wann wird das Objekt effektiv aus dem Objektspeicher entfernt?

Die erste Frage wird Aufgrund des Objektzustandes, der im Attribut `state` (vgl. Abb. 5.1) gespeichert wird, entschieden. Dieser Zustand wird bei gelöschten Datensätzen auf den Wert `Deleted` gesetzt. In diesem Zustand können noch die Attributwerte des Objekts gelesen, aber nicht mehr manipuliert werden. Wird dennoch eine Manipulation versucht, wird diese mit einer Fehlermeldung zurückgewiesen. Allerdings kann das Objekt diesen Zustand durch Aufruf seiner `reuse`-Methode wieder verlassen. Diese Methode wird im übrigen auch der Cache der zugehörigen Klasse verwenden, um das Objekt zu reaktivieren. [1]
Die zweite Frage, die Bestimmung des Zeitpunktes, zu dem ein Datensatzobjekt aus dem Objektspeicher entfernt wird, läßt sich mit Hilfe des *Garbage Collectors* des Objektspeichers beantworten. Dieser löscht zu in der Regel nicht exakt vorherbestimmbaren Zeitpunkten all die Objekte aus dem Speicher, die nicht mehr erreichbar sind. Ein Objekt ist dann nicht mehr erreichbar, wenn es weder direkt noch indirekt über Methoden oder Attribute anderer erreichbarer Objekte angesprochen werden kann. Enthält z.B. eine freie Variable ein Objekt und wird der Inhalt dieser Variablen mit einem zweiten Objekt überschrieben, so ist das erste Objekt nicht mehr erreichbar. Wenn nun also ein Datensatzobjekt von keinem anderen Objekt mehr referenziert wird, kann es wie alle anderen Objekte aus dem Objektspeicher gelöscht werden. Im Abschnitt 5.3.4 wurde ein Cache-Konzept entwickelt, daß das Eintreten dieses Ereignisses gerade auszuschließen scheint. Denn selbst, wenn ein Datensatzobjekt von keinem anderen Objekt mehr referenziert wird, so wird es zumindest noch vom Cache seiner Klasse referenziert und damit nicht aus dem Objektspeicher gelöscht. Für diesen Fall bieten einige objektorientierte Systeme sogenannte schwache Referenzen oder *Smart Pointers* an (vgl. Abschnitt 5.1.4). Der Garbage Collector löscht nur Objekte, auf die entweder überhaupt keine oder nur noch schwache Referenzen bestehen. Der Cache einer Datensatzklasse muß demzufolge schwache Referenzen auf Datensatzobjekte verwalten, damit der Garbage Collector diese aus dem Objektspeicher entfernen kann. Die `finalize`-Methode der schwachen Referenz auf das Datensatzobjekt, die während der Garbage Collection aufgerufen wird, sendet dem Datensatzobjekt eine `flush`-Nachricht. Falls der Zustand des aus dem Cache zu entfernenden Objektes noch eine Datenbankoperation erfordert, so wird diese hierdurch veranlaßt.

5.3.6 Caching externer Daten in persistenten Systemen

Das in dieser Arbeit verwendete TYCOON-System unterstützt Persistenz, d.h. der momentane Zustand des Objektspeichers läßt sich zu nahezu jedem beliebigen Zeitpunkt sichern. Wird das System neu gestartet, so versucht es, den letzten gespeicherten Zustand wiederherzustellen. Dies führt zu Komplikationen, wenn der Zustand zum Zeitpunkt der Speicherung von externen Ressourcen abhängt, wie z.B. einer relationalen Datenbank. Zwar kann das objektorientierte System unter Umständen die Verbindung zur Datenbank wiederherstellen, die

[1] Das vollständige Zustandsdiagramm eines Datensatzobjekts wird in Abschnitt 6.1.3 vorgestellt.

Inhalte der Datenbank können sich zwischenzeitlich aber verändert haben. Da auch der Cache von Datensatzklassen persistent gespeichert wurde, muß dieser beim Neustart mit der Datenbank synchronisiert werden. Der Cache sucht also alle bei ihm registrierten Datensätze in der Datenbank und liest von dort deren aktuellen Zustand. Dieser kann sich vom Zustand zum Zeitpunkt der Sicherung des Objektspeichers unterscheiden. Würde das objektorientierte System bei seinem Neustart der Datenbank seinen mittlerweile veralteten Datenbestand "aufzwingen", so wäre der Nutzen der Datenbank als gemeinsamer Speicher für verschiedene Anwendungen stark eingeschränkt, weil das Objektsystem alle Änderungen überschreiben würde, die außerhalb seiner Laufzeit stattgefunden haben.
Der nach einem Neustart sichtbare Zustand des Objektspeichers ist also bezogen auf Datensatzobjekte, die auf Datenbankinhalte verweisen, nicht zwingend mit dem Zustand zum Zeitpunkt der Objektspeichersicherung identisch, spiegelt aber den korrekten Zustand der Datenbank wieder. Dies stellt einen Kompromiß zwischen dem Persistenzbegriff des Objektspeichers und dem der Datenbank dar. Die harte Forderung nach vollständiger Wiederherstellung des letzten gespeicherten Systemzustandes wird zugunsten der Aktualität der verwendeten Daten aufgegeben.
Bei Anwendungen, in denen allein das objektorientierte System den maßgeblichen Zustand der Daten bestimmen soll, erscheint es ohnehin sinnvoller, die Daten ausschließlich im Objektspeicher selbst zu halten und sie nicht zusätzlich in eine externe Datenbank auszulagern, die sich prinzipiell der Kontrolle eines objektorientierten Systems entzieht, sobald die Verbindung zwischen den beiden unterbrochen wird.

5.3.7 Behandlung von Abhängigkeiten zwischen Datensätzen

Wird der Zustand eines Datensatzobjekts in der Datenbank gespeichert, so muß gewährleistet werden, daß dabei keine in der Datenbank definierten referentiellen Integritätsbedingungen verletzt werden. D.h. wenn im Objektspeicher neue Datensatzobjekte angelegt werden und Beziehungen zu solchen Objekten hergestellt oder zwischen bestehenden Objekten verändert werden sollen, so ist die Reihenfolge, in der `flush`-Operationen ausgeführt werden, nicht beliebig. Beispielsweise kann ein Datensatz, der in einer Fremdschlüsselbeziehung zu einem anderen Datensatz steht, erst dann in der Datenbank gespeichert werden kann, wenn der referenzierte Datensatz bereits in der Datenbank vorhanden ist. Beim Löschen von Datensätzen tritt der umgekehrte Fall auf. Bevor ein Datensatz gelöscht werden kann, müssen alle anderen Datensätze, die sich auf diesen Datensatz beziehen, entweder ebenfalls vorher gelöscht oder ihre Referenzen auf andere Datensätze gesetzt werden.
Da solche Operationen im objektorientierten System durch den Cache erst verzögert ausgeführt werden, muß gewährleistet werden, daß diese trotzdem in einer konsistenten Reihenfolge nachvollzogen werden. Da diese Vorgänge ohne Eingriff des Benutzers und ohne seine Kenntnis ablaufen sollen, werden sie erst im Kapitel 6 näher beschrieben. Die öffentlichen Schnittstellen der in diesem Kapitel erläuterten Klassen sind von diesen Überlegungen nicht betroffen.

5.4 Abwicklung der Datenbankzugriffe

In den Abschnitten 5.2 und 5.3 werden zum einen Klassen vorgestellt, deren Objekte zur Erzeugung von Datensatzklassen nötig sind, zum anderen werden wesentliche Eigenschaften

der zu erzeugenden Datensatzklassen erläutert. Die Objekte der erzeugten Klassen sollen dabei Datensätze aus einer relationalen Datenbank repräsentieren. Auch wird angedeutet, in welcher Art und Weise und zu welchem Zeitpunkt die Inhalte eines Datensatzobjektes mit Inhalten der Datenbank synchronisiert werden sollen. In diesem Abschnitt wird erläutert, wie nun die eigentliche Kommunikation mit der Datenbank abgewickelt wird.
Hierbei gilt es, eine Brücke zwischen zwei unterschiedlichen Abstraktionsebenen zu schlagen. Die niedrigere dieser beiden Ebenen wird durch die Schnittstelle definiert, über die im jeweiligen System der Zugriff auf die Datenbank stattfindet. In imperativen Sprachen wäre das z.B. eine Funktionsbibliothek nach dem ODBC-Standard, in objektorientierten Sprachen eine Klassenbibliothek wie z.B. in Java die Klassenbibliothek JDBC [HCF97] oder in TYCOON die TYCOON-SQL-Schnittstelle (vgl. Abschnitt 2.2, nähere Details siehe [Sku98]). Über diese Schnittstellen lassen sich SQL-Anfragen an die Datenbank senden und gegebenenfalls die Ergebnisse der Anfrage gewinnen. Die zweite Abstraktionsebene stellen die Datensatzklassen dar, deren Attribute und Methoden mit Inhalten und Operationen der Datenbank korrespondieren sollen. Auf dieser zweiten, höheren Abstraktionsebene tritt die Sprache SQL nicht mehr direkt in Erscheinung. Daraus ergibt sich die Notwendigkeit einer vermittelnden dritten Schicht, deren Objekte in der Lage sind, Werte und Operationen eines Datensatzobjektes in der hohen Abstraktionsschicht auf SQL-Befehle und -Anfragen der darunterliegenden niedrigen Abstraktionsschicht abzubilden.
Hierfür wird eine Klasse verwendet, deren Objekte sämtliche Standardzugriffe auf eine Tabelle einer relationalen Datenbank kapseln. Zu den in diesem Kontext relevanten Standardzugriffen gehören das Einfügen, Ändern, Löschen und Lesen von einzelnen Datensätzen, sowie das Lesen größerer Mengen von Daten, die ein gemeinsames Prädikat erfüllen. Die SQL-Befehle, die zur Ausführung dieser Operationen nötig sind, lassen sich direkt aus den Metadaten zu einer Datenbanktabelle generieren. Diese Metadaten werden von der Funktions- oder Klassenbibliothek für den Datenbankzugriff via SQL geliefert. Ein solches Tabellenobjekt bereitet

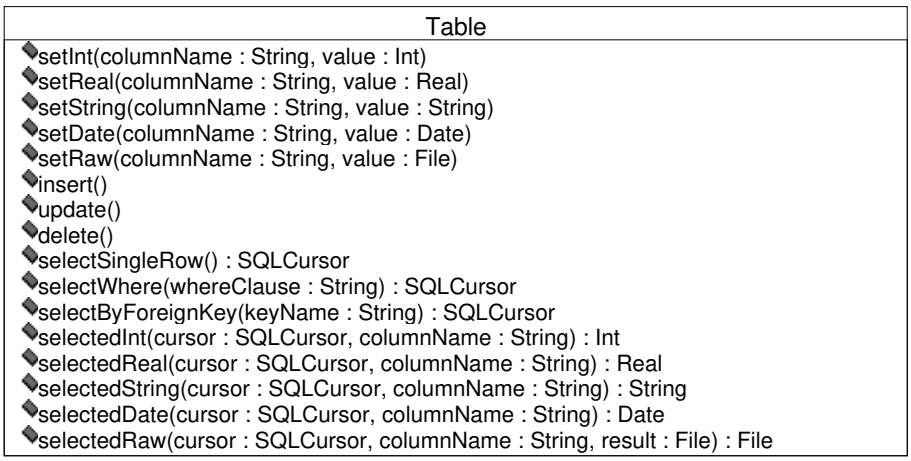

Abbildung 5.7: Tabellenklasse zur Kapselung von Standard-Datenbankoperationen

bei seiner Erzeugung die SQL-Befehle vor, die für die Standarddatenbankzugriffe auf eine konkrete Tabelle nötig sind. Abbildung 5.7 zeigt die öffentliche Schnittstelle eines solchen Tabellenobjekts. Mit Hilfe seiner `set`-Methoden lassen sich die Parameter der vorbereite-

ten SQL-Befehle mit den Werten einer konkreten Tabellenzeile versehen. Durch Aufruf einer speziellen Methode wird dann die gewünschte Datenbankoperation ausgeführt. Diese Lösung hat gegenüber zur Programmlaufzeit generiertem SQL-Code den Vorteil, daß der SQL-Code nur einmal bei der Initialisierung des Tabellenobjekts an die Datenbank übermittelt wird. Bei jeder weiteren Kommunikation mit der Datenbank werden nur noch die eigentlichen Daten übertragen. Der Kommunikationsaufwand bei späteren Datenbankzugriffen ist dadurch geringer.

Kapitel 6

Implementierung der objektrelationalen Klassenbibliothek für Tycoon

Dieses Kapitel erläutert im Detail das Zusammenspiel der im Kapitel 5 entworfenen Komponenten und zeigt, wie diese konkret in der Sprache TL2 implementiert werden. Dazu wird zunächst erörtert, in welcher Form konkrete Datensatzobjekte realisiert werden (Abschnitt 6.1). Die Verwaltung aller Objekte einer Datensatzklasse ist ein Verhalten der Datensatzklasse, welches in ihrer Metaklasse beschrieben wird. Die Implementierung dieses Verhaltens ist Gegenstand des Abschnitts 6.2. Aus den gemeinsamen Strukturen von Datensatzobjekten und -klassen können dann Vorschriften für die automatische Generierung der zugehörigen Klassen und Metaklassen abgeleitet werden. Diese Vorschriften wiederum beeinflussen die Implementierung des eigentlichen Klassengenerators (Abschnitt 6.3). Wurden Datensatzklassen erfolgreich erzeugt, so können sie mit einer relationalen Datenbank verbunden werden. Die wesentlichen Charakteristika dieser "objektrelationalen" Datenbankverbindung werden im Abschnitt 6.4 vorgestellt.

6.1 Einzelne Datensätze

Die grundlegenden Eigenschaften eines Datensatzobjekts wurden bereits in Abschnitt 5.2.1 vorgestellt. Ein solches Objekt enthält

- für jedes Attribut einer Datenbankrelation einen korrespondierenden Slot,
- Methoden zum Löschen und Wiederverwenden des Objekts,
- Methoden zur Navigation zu assoziierten Objekten (Abschnitt 6.1.2),
- einen Slot `state`, anhand dessen entschieden wird, welche Datenbankoperationen nötig sind, um den augenblicklichen Zustand des Objekts in der Datenbank zu speichern (vgl. Abschnitt 6.1.3),
- eine Methode, die die nötigen Datenbankoperationen zur Speicherung des aktuellen Objektzustandes in der Datenbank ausführt (vgl. Abschnitt 6.1.4).

In dieser Liste fehlen Methoden zum Ändern von Attributwerten. Diese erscheinen auf den ersten Blick auch nicht notwendig, da sich Attributwerte direkt mit der in TYCOON üblichen Zuweisungsoperation verändern lassen. Eine Zuweisung stellt in TYCOON ebenfalls den Aufruf einer bestimmten Methode dar. Wie die von TYCOON bereitgestellten "Zuweisungsmethoden" an die Erfordernisse der objektrelationalen Klassenbibliothek angepaßt werden können, zeigt der Abschnitt 6.1.1.

6.1.1 Änderungsoperationen

Der Vorname bzw. das Attribut `vorname` einer Person `p` ließe sich in einem Objekt unserer Beispielklasse folgendermaßen verändern (vgl. Abbildung 5.2):

```
p.vorname := "Heinrich"
```

Das Attribut hat nun einen neuen Wert, der irgendwann auch in der Datenbank gespeichert werden soll. Da der Zeitpunkt dieser Speicherung nicht vorgegeben ist, muß sich das Objekt zunächst merken, welche Veränderungen an ihm vorgenommen wurden. Dies geschieht durch den Slot `state`, dessen Wert bei einer Veränderung von Attributwerten ebenfalls geändert werden muß. Bis jetzt hat der Benutzer aber lediglich einen Attributwert manipuliert. Da `state` ein anderer Slot ist, bliebe er unverändert. Um nun den Benutzer des Objekts nicht mit der manuellen Änderung des `state`-Attributes zu belasten, von dessen Existenz er ohnehin nichts bemerken sollte, muß eine Möglichkeit geschaffen werden, das `state`-Attribut implizit bei jeder Änderungsoperation auf den korrekten Wert zu setzen. Dies geschieht mit Hilfe einer privaten Methode `_modify`, die bei jeder Änderung eines Attributwertes implizit aufgerufen wird. Um nun den impliziten Aufruf dieser Methode zu gewährleisten, kann nicht mehr der normale Zuweisungsoperator verwendet werden, sondern für jedes Attribut wäre eine separate `set`-Methode nötig, die das Attribut auf den neuen Wert setzt und die `_modify`-Methode aufruft. Daß sich die Programmierschnittstelle des Objekts für den Anwendungsprogrammierer dennoch nicht ändern muß, ist der Tatsache zu verdanken, daß auch Infixoperationen, wie die Zuweisung, in TYCOON ganz gewöhnliche Methoden eines Objekts darstellen. Diese können wie alle anderen Methoden auch überschrieben werden. Die obige Infixschreibweise dient nur der besseren Lesbarkeit des Programmcodes. Die Zuweisung in konventioneller Schreibweise sähe für das Beispiel folgendermaßen aus

```
p."vorname:="("Heinrich")
```

Diese Methode läßt sich nun z.B. in folgender Weise redefinieren

```
"vorname:="(value :String):Void
{
  _modify(),
  _vorname := value
}
```

Nun wird bei jeder Zuweisung zunächst durch Aufruf von `_modify` eine Änderung des `state`-Attributs ausgelöst und dann der Slot `_vorname` auf den neuen Wert gesetzt. Ein Benutzer des Personenobjekts wird die Zuweisung in Infixnotation genauso weiterbenutzen wie vorher und nicht bemerken, daß dabei weitere Methoden aufgerufen werden und der neue Wert an einen privaten Slot zugewiesen wird. Private Methoden und Slots werden im folgenden immer durch einen vorangestellten Unterstrich gekennzeichnet. Daß die tatsächliche Zuweisung

nun an einen privaten für den Benutzer nicht zugänglichen Slot erfolgt, hat nicht nur den Grund, ungewollte Rekursionen in der Zuweisungsmethode zu vermeiden. Wie im Abschnitt 5.3.3 erörtert, muß für Attribute des Primärschlüssels der schreibende Zugriff unterbunden werden. Dies wird dadurch erreicht, daß alle Attribute als private, von außen nicht zugreifbare Slots implementiert werden. Für alle diese Slots werden dann Lesemethoden bereitgestellt. Schreibmethoden werden nur für diejenigen Slots angeboten, deren korrespondierendes Attribut in der Datenbank nicht Teil des Primärschlüssels ist. Auch das Lesen eines Slotwertes ist nichts anderes als ein Aufruf einer Methode, die mit dem Namen des Slots identisch ist. Analog zur Zuweisungsmethode läßt sich also auch die Lesemethode eines Slots überschreiben.

```
vorname():String
{
  _lock(),
  _vorname
}
```

Der Rückgabewert eines Methodenaufrufs ist in TYCOON immer der Wert der letzten Anweisung, also hier der Inhalt des privaten Slots `_vorname`. Die Methode `_lock` wird bei Leseoperationen aufgerufen, um eine Sperrung des jeweiligen Datensatzes für andere Benutzer der Datenbank zu veranlassen. Bei Schreiboperationen wird sie bereits durch die Methode `_modify` aufgerufen. Durch das Überschreiben der Standard-Lese- und Schreibmethoden sind nun zwei wichtige Effekte erreicht.

1. Der Benutzer des Datensatzobjektes kann ausschließlich solche Attribute ändern, die nicht zum Primärschlüssel gehören.

2. Bei jeder Änderung wird implizit der Wert des privaten `state`-Attributs aktualisiert.

Diese Effekte werden erreicht, ohne daß sich die für den Benutzer sichtbare Schnittstelle des Datensatzobjekts ändert. Lediglich für Schlüsselattribute wird ihr Funktionsumfang eingeschränkt. Die in diesem Abschnitt vorgestellten Methoden zum Lesen und Ändern von Attributwerten sind spezifisch für einzelne Tabellen und deren Datensätze. Bei der Erzeugung einer neuen Datensatzklasse müssen also jeweils diese an die Tabellenstruktur angepaßten Methoden und die zugehörigen privaten Slots generiert werden. Die `_modify`-Methode, die von den datensatzspezifischen Änderungsmethoden aufgerufen wird, bewirkt eine Änderung des `state`-Attributs. Die Implementierung dieser Methode hängt wie das `state`-Attribut selbst (vgl. Abschnitt 6.1.3) nicht von der Struktur einer konkreten Datenbanktabelle ab. Sie kann daher in einer abstrakten Superklasse der konkreten Datensatzklasse implementiert werden. Gleiches gilt für die `_lock`-Methode, die sicherstellt, daß ein Datensatz in der Datenbank gesperrt wird, sobald auf das korrespondierende Objekt zugegriffen wird.

6.1.2 Operationen auf Fremdschlüsselattributen

Enthält die zu einer Datensatzklasse korrespondierende Tabelle Fremdschlüssel, die auf eine andere Tabelle verweisen, zu der ebenfalls eine Datensatzklasse existiert, so werden dem Benutzer weitere Methoden zur Verfügung gestellt, mit denen sich direkt das referenzierte Objekt erhalten oder durch ein anderes ersetzen läßt.
Enthält die Beispieltabelle `Person` einen Fremdschlüssel `Arbeitsplatz`, durch den ein Datensatz der Tabelle `Abteilung` identifiziert wird und existiert auch eine zugehörige Klasse

`Abteilung`, so liefert die Methode `arbeitsplatz` nicht etwa den Wert des Fremdschlüsselattributes in der Datenbank, sondern eine Referenz auf das vollständige Objekt der Klasse `Abteilung`. Analog existiert eine Methode, um den Personendatensatz mit einer neuen Abteilung zu assoziieren. Steht ein Datensatz zu einer Menge von anderen Datensätzen in Beziehung, enthält er eine Methode, die gerade die Menge dieser Objekte liefert. So würde die Beispielklasse `Abteilung` eine Methode `angestellte` enthalten, die eine Menge von Objekten der Klasse `Person` liefert.
Da diese Methoden zur Verwaltung von Assoziationen aus den referentiellen Integritätsbedingungen der entsprechenden Datenbanktabellen abgeleitet werden, müssen diese Bedingungen beim Speichern der Daten eingehalten werden. Das bedeutet z.B., daß ein Datensatz erst dann aus der Datenbank gelöscht werden darf, wenn er von keinem anderen Datensatz mehr über Fremdschlüssel referenziert wird. Die Kinddatensätze müssen also vorher gelöscht oder ihre Referenzen auf andere Datensätze "umgebogen" werden. Beim Einfügen neuer Datensätze, die sich untereinander referenzieren, müssen hingegen zunächst diejenigen Datensätze in die Datenbank eingefügt werden, auf die sich dann spätere "Kinddatensätze" beziehen können. Für diese Zwecke enthält jede Datensatzklasse die Methoden `flushParentRecords` und `flushChildRecords`, die während des `flush`-Vorganges abhängig vom Zustand des Datensatzes und seiner Beziehung zu anderen Datensätzen aufgerufen werden. Beim Löschen von Datensätzen wird zunächst `flushChildRecords`, beim Einfügen und Ändern von Daten wird `flushParentRecords` aufgerufen, bevor der Zustand des aktuellen Datensatzes selbst in die Datenbank übertragen wird. Hat der Benutzer bei seinen Änderungen alle Integritätsbedingungen der Datenbank eingehalten, also z.B. beim Löschen eines Datensatzes auch alle seine "Kindobjekte" gelöscht, bzw. zum Löschen markiert, so garantieren diese beiden Methoden, daß die betroffenen Datensätze auch in einer Reihenfolge in der Datenbank geändert werden, die die Integritätsbedingungen einhält.

6.1.3 Objektzustand

Nicht nur durch Änderungen von Slotwerten, auch durch Aufruf der `delete`-, `reuse`- oder der `flush`-Methode ändert sich der durch das `state`-Attribut repräsentierte Zustand des Objekts. Während nach der Änderung eines oder mehrerer Attributwerte in der Datenbank ein `update`-Befehl ausgeführt werden muß, ist nach dem Aufruf der `delete`-Methode in der Datenbank ein entsprechender `delete`-Befehl auszuführen. Welche weiteren Zustände bei der Manipulation eines Datensatzes auftreten können, zeigt Abbildung 6.1 Die einzelnen Zustände haben dabei folgende Bedeutung:

- **ToInsert**
  Das Objekt ist neu, d.h. es existierte bisher weder im Cache noch in der Datenbank. Beim Aufruf der `flush`-Methode muß der Datenbank ein `insert`-Befehl geschickt werden.

- **ToUpdate**
  Das Objekt existiert bereits und der Wert eines oder mehrerer seiner Attribute wurde verändert. Beim Aufruf der `flush`-Methode muß der Datenbank ein `update`-Befehl geschickt werden.

- **ToDelete**
  Das Objekt wurde zum Löschen vorgemerkt. Bei Aufruf der `flush`-Methode wird ein `delete`-Befehl an die Datenbank geschickt.

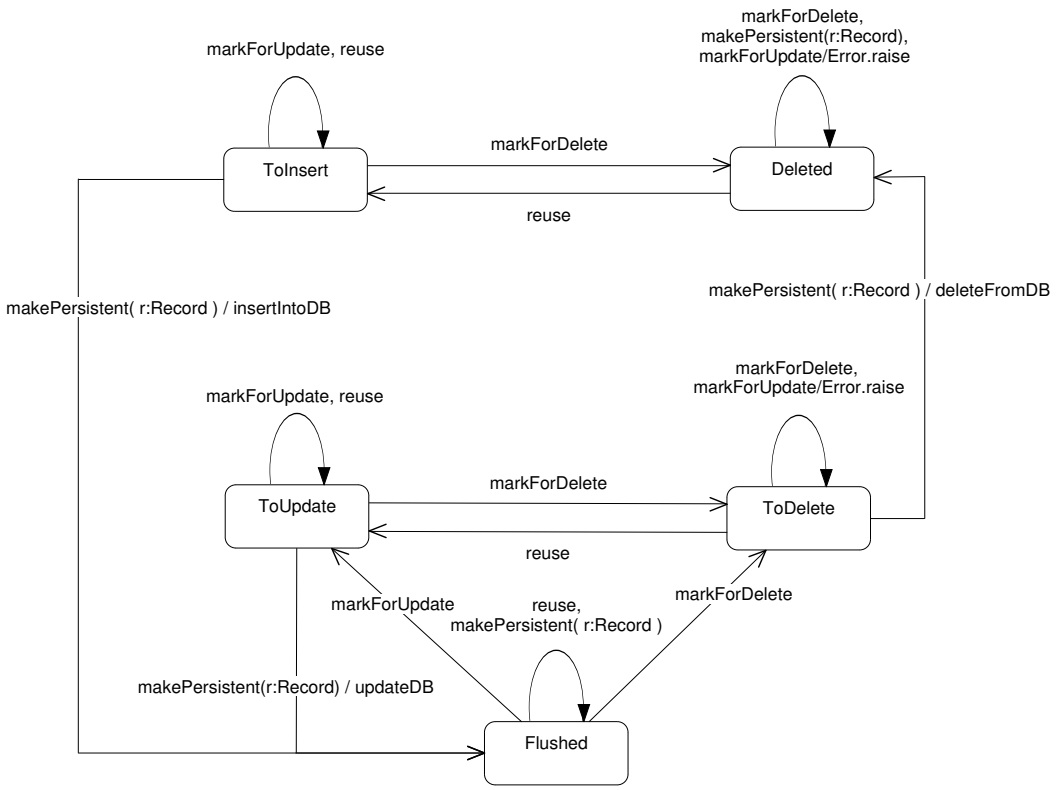

Abbildung 6.1: Zustände eines Datensatzobjekts

- **Deleted**
  Die `flush`-Methode wurde aufgerufen und das Objekt wurde in der Datenbank gelöscht.
- **Flushed**
  Der Inhalt des Datensatzobjekts wurde in der Datenbank gespeichert und das Objekt ist seitdem nicht mehr verändert worden.

Die Zustandsübergänge sind mit den Ereignissen benannt, die den jeweiligen Übergang auslösen können. In jedem Zustand kann eines der folgenden Ereignisse auftreten:

- **makePersistent**
  Der augenblickliche Objektzustand wird auf die Datenbank übertragen.
- **reuse**
  Das Objekt soll wiederverwedet werden. Dieses Ereignis tritt dann ein, wenn das Objekt zuvor aus der Datenbank gelöscht oder zum Löschen vorgemerkt wurde und der Zustand `ToDelete` oder `Deleted` wieder verlassen werden soll.
- **markForUpdate, markForDelete**
  Diese Ereignisse signalisieren diejenige Datenbankoperation, die zur momentanen Änderung des Datensatzobjekts äquivalent wäre. Beim Ändern von Attributen tritt das Ereignis `markForUpdate` auf, beim Löschen das Ereignis `markForDelete`.

Ein Objekt kann zwar aus der Datenbank gelöscht werden, es verschwindet damit aber nicht zwangsläufig aus dem Objektspeicher (vgl. Abschnitt 5.3.4). Daher reagiert das Objekt in den Zuständen `ToDelete` und `Deleted` mit einer Fehlermeldung, falls in diesen Zuständen versucht wird, Objektinhalte zu ändern. Diese Zustände müssen erst durch Eintreten des Ereignisses `reuse` verlassen werden, bevor solche Zugriffe wieder erlaubt werden.
Der Zustandsautomat der Abbildung 6.1 wird unter Zuhilfenahme des Entwurfsmusters *State* implementiert, wie es z.B. bei Gamma et al. [GHJV96] beschrieben ist. Jeder Zustand wird dabei durch ein Objekt einer speziellen Zustandsklasse repräsentiert. Alle Zustandsklassen erben von einer gemeinsamen Superklasse, in der gerade die Ereignisse als Methoden definiert sind, die in jedem Zustand auftreten können. Die Subklassen implementieren dann jeweils das für einen konkreten Zustand spezifische Verhalten beim Eintreten der jeweiligen Ereignisse. Abbildung 6.2 zeigt das Klassendiagramm für unseren konkreten Fall. Tritt ein Ereignis ein,

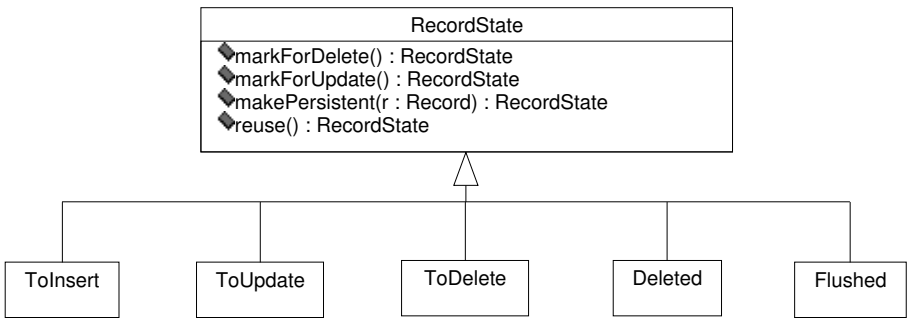

Abbildung 6.2: Klassen zur Implementierung des *State Patterns*

wird die entsprechende Methode aufgerufen. Diese führt die zustandsspezifischen Änderungen aus und liefert als Rückgabewert den Folgezustand. Bei der Implementierung dieses Entwurfsmusters wird noch ein weiteres Muster namens `Singleton` verwendet. Dieses stellt sicher, daß nicht für jeden Datensatz eigene Zustandsobjekte erzeugt werden müssen, sondern jedes Zustandsobjekt nur einmal im Objektspeicher vorhanden zu sein braucht. In TYCOON wird dieser Effekt dadurch erreicht, daß eine spezielle Metaklasse namens `SingletonClass` mit derjenigen Klasse parametrisiert wird, von der nur maximal ein Objekt existieren soll. Dieses Objekt ist dann über den Methodenaufruf <Klassenname>`.instance` systemweit zugänglich. Wird nun also an das Zustandsobjekt `Flushed.instance`, die Nachricht `markForUpdate` geschickt, so liefert es als Resultat das Zustandsobjekt `MarkedForUpdate.instance`. Eine Besonderheit ist beim Ereignis `makePersistent` zu beachten. Tritt dieses Ereignis in den Zuständen `ToInsert`, `ToUpdate` oder `ToDelete` auf, so ist eine Datenbankoperation auszuführen, für die die Attributwerte des jeweiligen Datensatzes benötigt werden, bei dem dieses Ereignis auftritt. Daher wird dieser Datensatz als Parameter an die `makePersistent`-Methode seines aktuellen Zustandsobjekts übergeben. Diese wiederum veranlaßt dann das Datensatzobjekt zu der nötigen Datenbankoperation. Abbildung 6.3 veranschaulicht dieses Zusammenspiel. Die Implementierung des Zustandsautomaten mit Hilfe des Entwurfsmusters *State* und der bei den Zustandsübergängen auftretenden Aktionen ist unabhängig von einer konkreten Datensatzklasse. Daher werden die entsprechenden Slots (`state`) und Methoden (`updateDB`, `insertIntoDB`, `deleteFromDB`) in einer abstrakten Klasse implementiert, von der alle konkreten Datensatzklassen erben.

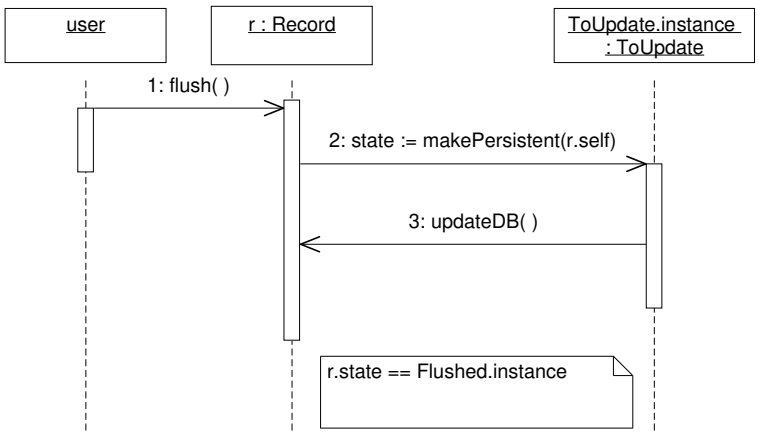

Abbildung 6.3: Ablauf eines `Flush`-Vorganges für einen Datensatz

Neben der bisher beschriebenen Zustandsvariable `state` existiert in jedem Datensatzobjekt noch eine weitere Zustandsvariable `locked`. Diese gibt an, ob der zu einem Datensatzobjekt korrespondierende Datensatz in der Datenbank für den Zugriff durch andere Datenbankbenutzer gesperrt ist. Um zu gewährleisten, daß während eines Datenbankzugriffs über eine TYCOON-Anwendung keine anderen Datenbankbenutzer parallel die gleichen Datensätze verändern können, wird jeder Datensatz, zu dem ein Datensatzobjekt erzeugt wird, in der Datenbank gesperrt. Dadurch kann bis zum Ende der jeweiligen Datenbanktransaktion nur das TYCOON-System auf die Inhalte dieses Datensatzes zugreifen. Die Zustandsvariable `locked` besitzt dann den booleschen Wert `true`. Beendet nun der Benutzer des TYCOON-Systems seine Datenbanktransaktion, so werden im Falle eines *Commit* alle von ihm vorgenommenen Änderungen an Datensatzobjekten in die Datenbank geschrieben und dort persistent gemacht. Im Falle eines *Rollback* werden die Änderungen verworfen. Am Ende der Transaktion werden aber in beiden Fällen die durch die Erzeugung der Datensatzobjekte angeforderten Datenbanksperren freigegeben und die `locked`-Attribute der betroffenen Datensatzobjekte auf `false` gesetzt.
Wird nun auf die Datensatzobjekte erneut lesend oder schreibend zugegriffen, so kann ihr Zustand seit dem Ende der letzten Datenbanktransaktion von anderen Datenbanknutzern verändert worden sein. Um die Konsistenz zwischen den Inhalten der Datensatzobjekte und den Inhalten der Datenbank wiederherzustellen, überprüft jedes Datensatzobjekt bei einem Lese- oder Schreibzugriff auf eines seiner Attribute den Zustand des `locked`-Attributs. Hat letzteres den Wert `false`, so erfolgte auf dieses Objekt seit dem Ende der letzten Transaktion kein Zugriff aus dem TYCOON-System heraus und der Inhalt der Datenbank kann sich zwischenzeitlich durch andere parallele Datenbanksitzungen geändert haben. In diesem Fall wird der aktuelle Inhalt der Datenbank in das Objekt übertragen. Dabei wird der Datensatz implizit in der Datenbank gesperrt und das `locked`-Attribut auf `true` gesetzt. Besitzt das `locked`-Attribut bereits den Wert `true`, ist kein Datenbankzugriff mehr erforderlich. Der Zustand des Datensatzobjekts ist dann mit dem des zugehörigen Datensatzes identisch bzw. enthält ausschließlich Änderungen, die während der laufenden Transaktion vom TYCOON-System selbst vorgenommen wurden.

6.1.4 Datenbankoperationen

Die Aktionen, die beim Übertragen des Objektzustandes in die Datenbank auftreten, wie `updateDB` oder `deleteFromDB`, lassen sich zwar als Methoden einer für alle Datensatzklassen gemeinsamen Superklasse definieren, dennoch lassen sie sich nicht vollständig innerhalb dieser Superklasse implementieren. Der logische Ablauf der Datenbankoperation kann bereits in der abstrakten Superklasse realisiert werden, da dies immer der gleiche Ablauf ist.

1. Parsen eines parametrisierten SQL-Befehls durch die Datenbank
2. Setzen der Parameter des SQL-Befehls auf die Werte der Attribute des Datensatzobjekts
3. Ausführen des SQL-Befehls

Da für jeden dieser Schritte Informationen erforderlich sind, die von der Struktur einer konkreten Datenbanktabelle abhängen, delegieren die Methoden für die einzelnen Datenbankoperationen die tabellenspezifischen Teile der Operation an andere Objekte bzw. an Subklassen. Das Parsen des SQL-Befehls wird von einem Tabellenobjekt, wie es in Abschnitt 5.4 beschrieben wird, vorgenommen. Das Tabellenobjekt wird nach der Herstellung einer Datenbankverbindung erzeugt und stellt seinen Satz von vorbereiteten SQL-Befehlen allen Anwendungen zur Verfügung. Die Klasse eines Datensatzobjekts, die auch die Cache-Verwaltung übernimmt, enthält eine Referenz auf dieses Tabellenobjekt. Jedes Datensatzobjekt wiederum kennt TYCOON seine eigene Klasse und kann daher auf deren Slots und damit auch auf das Tabellenobjekt zugreifen.
Das Setzen der Parameter des SQL-Befehls wird durch eine private Methode namens `_setParameters` übernommen. Sie wird in der abstrakten Datensatzklasse als `deferred` deklariert, d.h. sie wird zwar innerhalb der abstrakten Klasse bereits verwendet, ihre Implementierung wird aber in eine konkrete Subklasse "verschoben", sodaß in der abstrakten Klasse lediglich ihre Methodensignatur bekannt ist. Die Implementierung dieser Methode in einer konkreten Subklasse enthält dann diejenigen Aufrufe, die zum Setzen der für diese spezielle Klasse erforderlichen SQL-Parameter nötig sind.
Die Ausführung des SQL-Befehls wird wieder durch Senden einer Nachricht an das Tabellenobjekt der Datensatzklasse veranlaßt. Den nötigen Zugang zu diesem Objekt erhält ein Datenatzobjekt wiederum über einen Slot seiner eigenen Klasse.

6.1.5 Zusammenfassung

Jeder Datensatz einer relationalen Datenbank wird durch ein Objekt einer an die jeweilige Datenbanktabelle angepaßten Klasse repräsentiert. Zu jedem Attribut der Datenbanktabelle besitzt ein Datensatzobjekt einen privaten Slot, in dem der aktuelle Wert des Attributs gespeichert wird. Für Attribute des Primärschlüssels existieren Methoden zum Lesen von Werten dieser Slots, für alle anderen Attribute werden zusätzlich auch Schreibmethoden generiert. Methoden zur Manipulation von Fremdschlüsselwerten werden durch Methoden zur Manipulation von Objektreferenzen ersetzt bzw. ergänzt.
Die tabellenspezifischen Teile von Standard-Datenbankoperationen (Einfügen, Ändern, Löschen, Auswählen) werden durch private Methoden implementiert (`_setParameters`, `_getcurrentValues`). Der Zustand eines Datensatzes wird in einem privaten Slot `state` gespeichert. Die Zustandsübergänge und die Art der daraus resultierenden Datenbankoperationen,

sind nicht tabellenspezifisch, sondern hängen lediglich von der Art der vorhergehenden Datensatzmanipulationen ab. Daher werden das `state`-Attribut und die bei Zustandsübergängen auftretenden Aktionen unabhängig von konkreten Datenbanktabellen als Slots und Methoden einer abstrakten Klasse `Record` implementiert, von der alle Klassen für konkrete Datensatzobjekte erben.Die Aufgabe des Klassengenerators beschränkt sich also darauf, diejenigen Slots und Methoden zu generieren, die spezifisch für eine bestimmte Datenbanktabelle sind. Tabellenunabhängiges Verhalten erbt die generierte Klasse von der abstrakten Daetnsatzklasse.
Die Zustände eines Datensatzes werden mit Hilfe des Entwurfsmusters *State* implementiert. Das zustandsabhängige Verhalten von Datensatzobjekten läßt sich dadurch an zukünftige Entwicklungen anpassen, ohne daß größere Änderungen an den Datensatzklassen selber nötig werden. Mit Hilfe einer Zustandsvariable `locked` wird überprüft, ob ein Datensatzobjekt in

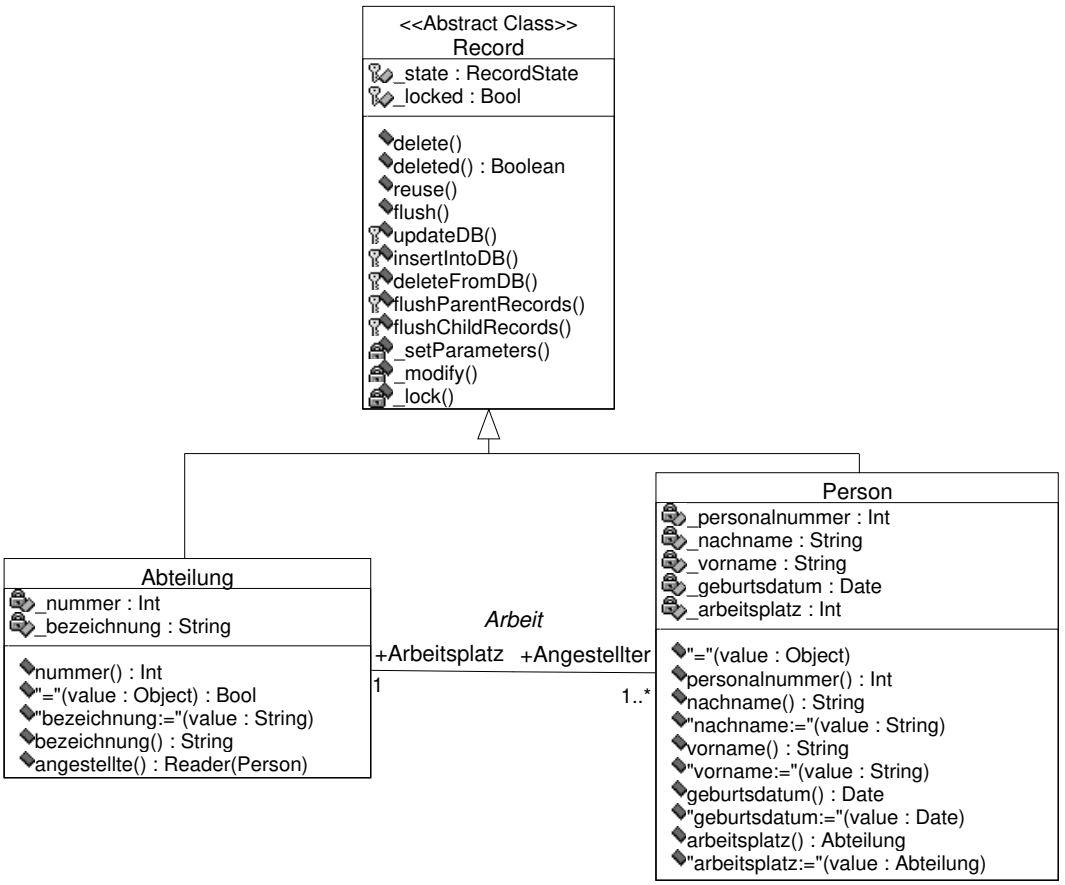

Abbildung 6.4: Implementierung von Datensatzklassen

der Datenbank gegen parallele Zugriffe anderer Datenbanknutzer gesperrt ist oder ob eine Sperre erst noch angefordert werden muß.
Abbildung 6.4 zeigt die abstrakte Datensatzklasse `Record`, ihre Attribute und Methoden und exemplarisch zwei datenbankspezifische Subklassen mit den tabellenspezifischen Attributen, Methoden und den Methoden zur Abbildung von Fremdschlüsselbeziehungen. Methoden, die als *geschützt* (*protected*) deklariert sind, erkennbar durch ein Schlüsselsymbol vor dem Namen,

werden durch die Objekte der Klasse `RecordState` benutzt, sollen aber nicht von anderen Benutzern aufgerufen werden. Da TYCOON nur zwischen den Zugriffsberechtigungen *öffentlich* (*public*, Zugriff durch beliebige Objekte erlaubt) und *privat* (*private*,Zugriff nur durch das Objekt selbst) unterscheiden kann, werden diese geschützten Methoden in TYCOON als öffentlich deklariert, obwohl sie nicht für den öffentlichen Gebrauch bestimmt sind..

6.2 Datensatzklassen

Jedes Datensatzobjekt gehört zu einer Klasse. Eine Klasse definiert in der objektorientierten Programmierung die gemeinsamen Eigenschaften der zu ihr gehörigen Objekte. Diese Eigenschaften der einzelnen Datensatzobjekte wurden im Abschnitt 6.1 erläutert. Eine Klasse ist in TYCOON ihrerseits ein Objekt. Das Verhalten einer Klasse beschränkt sich in vielen Fällen auf die Erzeugung von neuen Objekten bzw. Instanzen (Abschnitt 6.2.1).
Eine Klasse kann auch ein komplexeres Verhalten besitzen, das zur korrekten Verwaltung ihrer Instanzen erforderlich ist. In unserem Fall besteht dieses komplexe Verhalten in der Kontrolle der Zwischenspeicherung (*Caching*) von Datensätzen und der Sicherung der Objektidentität (Abschnitt 6.2.2). In TYCOON wird das Verhalten einer Klasse in ihrer Metaklasse definiert. In diesem Abschnitt wird untersucht, welche Teile dieses Verhaltens tabellenunabhängig sind und sich damit in abstrakten Klassen implementieren lassen und welche Teile von konkreten Tabllen abhängen, für die dann angepaßte Subklassen generiert werden müssen.

6.2.1 Erzeugung von neuen Datensatzobjekten

Soll ein neues Datensatzobjekt erzeugt werden, so schickt man der entsprechenden Klasse eine `new`-Nachricht. Die Klasse sorgt dann für die Erzeugung und Initialisierung des neuen Objekts. Wie bereits in Abschnitt 5.3.3 erwähnt, dürfen die Primärschlüsselwerte eines Datensatzes nicht geändert werden, um die Konsistenz zwischen Cache und Datenbank nicht zu gefährden. Da die Datensatzklasse andererseits nicht automatisch eindeutige Schlüssel generieren kann, muß eine Möglichkeit existieren, um den Primärschlüssel zumindest einmalig "manuell" festzulegen. Dies geschieht beim Aufruf der `new`-Methode einer Datensatzklasse. Als Parameter werden dieser Methode die Werte des Primärschlüssels für das neue Objekt übergeben. Dieses behält diese Werte dann bis zu seiner endgültigen Löschung aus dem Objektspeicher.
Existiert in der Datenbank bereits ein Datensatz mit diesem Primärschlüssel, so liefert die `new`-Methode anstelle eines neuen Objektes, ein Objekt mit den bereits vorhandenen Daten. Da eine Datensatzklasse nicht selbständig neue Schlüsselwerte generieren kann, muß der Anwendungsprogrammierer selber dafür sorgen, daß von ihm gewählte oder erzeugte Schlüsselwerte für neue Datensätze nicht mit Schlüsselwerten kollidieren, die andere Datenbankbenutzer während der gleichen Transaktion einfügen können. Kommt es beim Einfügen eines Datensatzes in die Datenbank zu einer solchen Kollision, kann der TYCOON-Cache darauf lediglich mit einer Fehlermeldung reagieren (*duplicate key*) oder den bereits vorhandenen Datensatz mit seinen eigenen Daten überschreiben. Die aktuelle Implementierung benutzt letztere Möglichkeit.
Eine Methode, die ebenfalls zur eindeutigen Identifizierung eines Objekts die Werte des Primärschlüssels benötigt, ist die sogenannte `lookup`-Methode. Diese sucht mit Hilfe des Primärschlüssels nach einem bereits existierenden Objekt. Sie erzeugt also kein neues Datensatzobjekt, sondern liefert bei erfolgreicher Suche das schon vorhandene Objekt zurück.

Da der Primärschlüssel eine Eigenschaft einer konkreten Datenbanktabelle ist, sind die Methoden `new` und `lookup` tabellenspezifisch und müssen damit vom Klassengenerator erzeugt werden.
Weitere tabellenspezifische Methoden sind zur Erzeugung von größeren Objektmengen erforderlich. Wichtig sind hier insbesondere solche Methoden, die alle Objekte liefern, die einen gemeinsamen Fremdschlüsselwert aufweisen. Über diese Methoden läßt sich die Navigation entlang von Assoziationen realisieren, z.B. "alle Angestellten einer Abteilung".

6.2.2 Cache

Ein Objekt, das unter Verwendung seiner Primärschlüsselwerte entweder neu erzeugt oder in der Datenbank gefunden wurde, ist implizit "gecached", da die aktuellen Werte seiner Attribute im Objekt selber und nicht direkt in der Datenbank gespeichert werden. Ein expliziter Cache in Form eines Datentyps zur Massendatenverwaltung (hier: `Set`) ist notwendig, um die Erzeugung zweier verschiedener Objekte mit identischen Primärschlüsselwerten zu verhindern (vgl. Abschnitt 5.3.4). Die wesentlichen Operationen auf dem Cache sind das Einfügen neuer Objekte, das Suchen und die Herausgabe von vorhandenen Objekten. Da der Cache quasi als Behälter für Objekte fungiert, aber die Inhalte der Objekte nicht verändert, kann er unabhängig von konkreten Datenbanktabellen implementiert werden. Er kann also in einer abstrakten Metaklasse definiert werden, von der alle konkreten Metaklassen erben. Hierbei ergeben sich allerdings zwei Probleme:

- In der abstrakten Superklasse sind Eigenschaften wie der Primärschlüssel eines Objekts noch nicht definiert. Sie werden erst in Subklassen festgelegt. Der Primärschlüssel wird aber benutzt, um ein existierendes Objekt aufzufinden.

- Wird ein Datensatzobjekt im Cache gefunden, soll nicht ein Objekt einer abstrakten Datensatzklasse zurückgegeben werden, sondern ein Objekt der konkreten Datensatzklasse mit all ihren tabellenspezifischen Methoden.

Beide Probleme lassen sich mit Mitteln von TYCOON lösen. Um ein Objekt im Cache aufzufinden, muß sein Primärschlüssel mit den Primärschlüsseln aller im Cache befindlichen Objekte verglichen werden. Um diesen Vergleich in einer abstrakten Klasse zu implementieren, wird die "="-Methode genutzt, die jedes TYCOON-Objekt von der TYCOON-Basisklasse `Object` erbt und die in Subklassen überschrieben werden kann. Diese liefert immer dann den Wert `true`, wenn zwei Objekte bezüglich bestimmter Eigenschaften äquivalent sind. Diese Methode wird nun bei der Generierung einer Datensatzklasse so überschrieben, daß zwei Datensatzobjekte immer dann gleich sind, wenn sie zur selben Klasse gehören und ihre Primärschlüsselwerte gleich sind.
Durch diesen Kunstgriff kann bei der Cache-Verwaltung die Suche nach einem Objekt mit einem bestimmten Primärschlüssel mit Hilfe der "="-Methode implementiert werden. Für die Suche nach einem im Cache befindlichen Objekt wird ein temporäres Objekt erzeugt und mit den gegebenen Primärschlüsselwerten initialisiert. Anschließend werden alle Objekte im Cache mit diesem temporären Objekt über die "="-Methode verglichen. Ein Objekt, das dieses Äquivalenzprädikat erfüllt, besitzt demnach denselben Primärschlüssel wie das temporäre Objekt und stellt damit das Ergebnis der Suche dar. Das Äquivalenzprädikat ist für jedes TYCOON-Objekt definiert und kann damit auch in einer abstrakten Metaklasse zur Suche im

Cache verwendet werden. In konkreten Datensatzklassen wird es dann durch den tabellenspezifischen Primärschlüsselvergleich ersetzt.
Für die Beispielklasse `Person` wird folgende "="-Methode erzeugt:

```
"="(x:Object):Bool
{
  x.clazz = Person                          (* Gehört x zur selben Klasse ? *)
  ? {
       let r = _typeCast(x,:Person),
            r.personalnummer = _personalnummer (* Primärschlüssel gleich ? *)
    }
  : { false }
}
```

Damit der Cache nicht nur Objekte der abstrakten Datensatzklasse `Record`, sondern Objekte konkreter Subklassen zurückliefern kann, wird der parametrische Polymorphismus von TYCOON ausgenutzt. Die abstrakte Metaklasse wird dazu mit einem Typparameter versehen, der immer einen Subtyp der abstrakten Recordklasse annehmen muß. In Subklassen wird dann dieser Typparamter auf den Typ einer konkreten Datensatzklasse gesetzt. Dieser Parameter kann in der abstrakten Metaklasse anstelle eines konkreten Typs für Rückgabewerte oder Parameter von Methoden verwendet werden. Subklassen benutzen dann den konkreten Typ.
In Abschnitt 5.3.5 wird die Frage der Lebensdauer eines Datensatzobjektes erörtert. Ein Datensatzobjekt wird demnach erst dann aus dem Cache entfernt, wenn es von keinem anderen Objekt mehr referenziert wird. Der Cache enthält zu diesem Zweck nicht direkte Referenzen auf die gespeicherten Objekte, sondern sogenannte schwache Referenzen. Eine schwache Referenz ist in TYCOON ein Objekt, das neben dem Verweis auf das eigentlich interessierende Objekt eine Methode enthält, die der *Garbage Collector* vor der vollständigen Löschung des Objekts aufruft. Diese Methode wird dann aufgerufen, wenn das betreffende Objekt über keine starke Referenz mehr erreichbar ist. Diese Methode wird benutzt, um den letzten Zustand des Objekts in die Datenbank zu schreiben, falls noch nicht geschehen. Außerdem kann sich diese schwache Referenz selbständig aus dem Cache entfernen. Ein Datensatz, der von keinem Objekt mehr referenziert wird, wird dadurch automatisch bei der nächsten Garbage Collection aus dem Objektspeicher und damit auch aus dem Cache gelöscht.

6.2.3 Zusammenfassung

Abbildung 6.5 faßt die wesentlichen Komponenten der Metaklassen von Datensatzklassen zusammen. Der Cache besteht aus einer Menge von schwachen Referenzen, die auf Objekte der jeweiligen Datensatzklasse verweisen, die momentan im Objektspeicher vorhanden sind. Die Methoden, mit denen Objekte dem Cache hinzugefügt, im Cache gesucht oder wieder entfernt werden, sind privat. Sie werden unter anderem von den `new`- und `lookup`-Methoden der Subklassen verwendet. Durch die Parametrisierung der abstrakten Metaklasse `RecordClass` mit dem Datensatztyp konkreter Subklassen und durch die Redefinition des Äquivalenzprädikats in konkreten Datensatzklassen können die Cache-spezifischen Methoden bereits in dieser abstrakten Klasse implementiert werden.
Die `flush`-Methode sendet an alle Datensatzobjekte im Cache eine `flush`-Nachricht. Die `reader`-Methode bewirkt eine Anfrage, die alle Einträge einer Datenbanktabelle liefert und diese über einen `Reader` (vgl. Abschnitt 5.1.2) lesbar macht. Die `init`-Methode schließlich

dient zur Initialisierung einer Datensatzklasse. Mit ihrer Hilfe wird der Datensatzklasse ein Tabellenobjekt übergeben, mit dessen Hilfe einzelne Datensatzobjekte konkrete Datenbankoperationen auslösen können.
Subklassen dieser abstrakten Metaklasse implementieren dann nur noch die tabellenspezifischen Methoden `new` und `lookup`. Falls eine Tabelle Fremdschlüsselattribute enthält, müssen

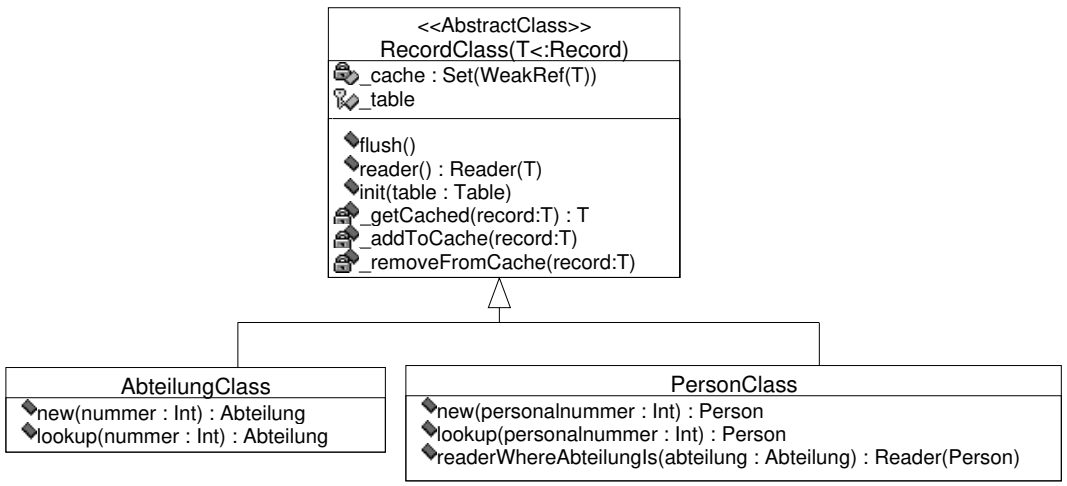

Abbildung 6.5: Metaklassen für Datensatzklassen

vom Klassengenerator zusätzlich Methoden erzeugt werden, die zu einem gegebenen Fremdschlüsselwert bzw. -objekt alle zugehörigen Datensatzobjekte liefern. So liefert die in der Metaklasse `PersonClass` definierte Methode `readerWhereAbteilungIs` (vgl. Abbildung 6.5) einen `Reader` aller Personen, deren Arbeitsplatz die gegebene Abteilung ist.

6.3 Die Generierung datenbankspezifischer Klassen

In den Abschnitten 6.1 und 6.2 wird erläutert, welche Eigenschaften eines Datensatzobjekts und seiner Klasse für alle Datensätze gleich sind und damit in gemeinsamen Superklassen implementiert werden können. Außerdem werden Eigenschaften erwähnt, die nicht für beliebige Datensätze gelten, sondern von der konkreten Struktur des verwendeten Datenbankschemas abhängen. Dieser Abschnitt beschäftigt sich mit der Generierung derjenigen Klassen, die diese datenbankschemaspezifischen Eigenschaften von Datensatzobjekten implementieren.
Zur Veranschaulichung der wesentlichen Generierungsschritte wird im folgenden ein durchgehendes Beispiel eingesetzt. Gegeben seien zwei Klassen `Abteilung` und `Person`, die schon im vorigen Kapitel verwendet wurden (Abbildung 6.6). Beide Klassen besitzen eine Menge von Attributen und stehen zueinander in einer Beziehung `Arbeit` mit den jeweiligen Rollen. Die Attribute `Nummer` und `Personalnummer` mögen jeweils ein Objekt der betreffenden Klasse eindeutig identifizieren. Um nun diese beiden Klassen in einer relationalen Datenbank zu repräsentieren, erzeugt man für jede der beteiligten Klassen eine eigene Tabelle. Die zugehörigen SQL-Befehle haben die folgende Gestalt:

Abbildung 6.6: Assoziation zwischen Datensätzen

```
create table abteilungen (
  nummer      int         primary key,
  bezeichnung varchar(20)
)
;
create table personen (
  personalnummer int         primary key,
  nachname       varchar(20),
  vorname        varchar(20),
  geburtsdatum   date,
  abteilung      int,
  constraint arbeitet_in foreign key (abteilung) references abteilungen
)
;
```

Die Beziehung zwischen den beiden Klassen wird durch die in der Tabelle `personen` definierte referentielle Integritätsbedingung `arbeitet_in` wiedergespiegelt. Diese Bedingung besagt, daß das Attribut `abteilung` einen Fremdschlüssel darstellt, durch den Datensätze der Tabelle `abteilungen` identifiziert werden. Die Datenbank akzeptiert dadurch für dieses Fremdschlüsselattribut nur noch solche Werte, für die in der Tabelle `abteilungen` ein korrespondierender Primärschlüsselwert existiert. Zu jeder Person kann somit angegeben werden, in welcher Abteilung sie arbeitet. In der Tabelle `abteilungen` selbst sind keine Daten über Beziehungen zu anderen Tabellen vorhanden.

Klassengenerator

Zur Erzeugung von Klassen, die an ein Datenbankschema angepaßt sind, wird ein Objekt der Klasse `ClassGenerator` verwendet. Dieses Objekt besitzt Methoden, über die es konfiguriert werden kann (`addNewClass` und `removeClass`) und mit denen die eigentliche Klassenerzeugung angestossen wird (`generate` und `install`). Um nun zu einem konkreten Datenbankschema passende Klassen zu generieren, übergibt man dem Generator jeweils durch Aufruf der `addNewClass`-Methode eine Beschreibung einer Datenbanktabelle und den Namen der neu zu erzeugenden Klasse.
Die Beschreibung der Datenbanktabelle wird in Form eines `SQLTableDescription`-Objekts übergeben. Dieses Objekt enthält Daten über Anzahl, Namen und Typen der einzelnen Tabellenspalten, sowie über Schlüsselattribute einer Tabelle (Abbildung 6.8). Ein `SQLTableDescription`-Objekt kann von den `SQLConnection`-Objekten der TYCOON-SQL-Schnittstelle durch Aufruf ihrer `describeTable`-Methode erzeugt werden. Gehört die Beispieltabelle `per-`

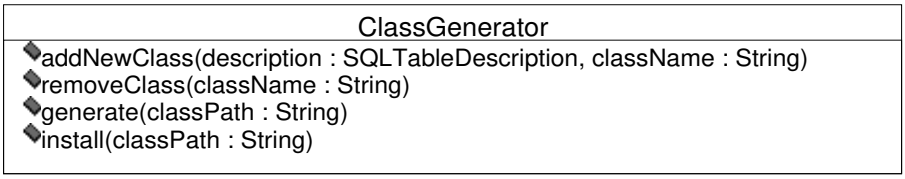

Abbildung 6.7: Schnittstelle des Klassengenerators

sonen einem Benutzer namens "scott", dann wird diese Tabelle durch den folgenden Methodenaufruf beim Klassengenerator registriert:

```
ClassGenerator.instance.addNewClass(
  oracle.describeTable(
                       "scott",      (* Besitzer der Tabelle *)
                       "personen"    (* Name der Tabelle *)
                      ),
  "Person"                           (* Name der zu generierenden Klasse *)
)
```

Der Klassengenerator initialisiert nun einen sogenannten `ClassBuilder`, an den die Tabellenbeschreibung und der Klassenname weitergereicht werden. Der Klassengenerator verwaltet eine Menge aller von ihm erzeugten `ClassBuilder`. Durch Aufruf der `removeClass`-Methode mit dem Namen einer Klasse kann ein `ClassBuilder` wieder aus dieser Menge entfernt werden.
Durch den Aufruf der Methode `generate` wird dann die eigentliche Klassenerzeugung gestartet. Jedem `ClassBuilder` des Klassengenerators wird eine `build`-Nachricht gesendet. Der `ClassBuilder` erzeugt dann zu seiner jeweiligen Tabellenbeschreibung eine zugehörige Klasse und eine Metaklasse. Der Klassengenerator sammelt die so erzeugten Klassenbeschreibungen und speichert sie im Dateisystem des Rechners ab. Der Pfad, über den die neuen Klassenbeschreibungen im Dateisystem zu erreichen sind, wird beim Aufruf der `generate`-Methode als String übergeben. Dieser Ablauf entspricht dem Generierungsvorgang, wie er schon in der Entwurfsphase (Abschnitt 5.2.2, Abbildung 5.6) angedeutet wird. Startet man die Klassengenerierung durch Aufruf der `install`-Methode, so werden zunächst ebenfalls Klassenbeschrei-

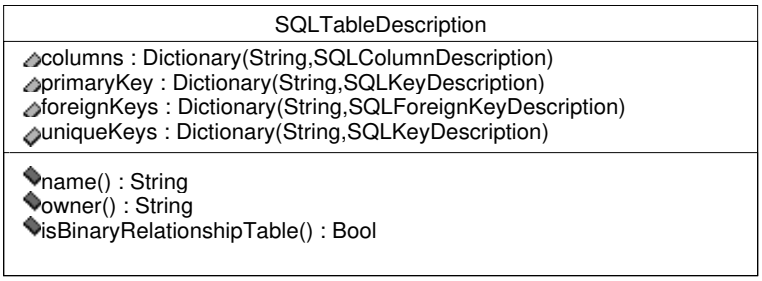

Abbildung 6.8: SQLTableDescription

bungen generiert und abgespeichert. Zusätzlich werden diese Beschreibungen aber wieder in das TYCOON-System geladen und vom Compiler übersetzt, so daß die resultierenden Klassen unmittelbar danach verwendet werden können. Die Klassenbeschreibungen werden im Dateisystem in Form von gewöhnlichem TYCOON-Quellcode abgespeichert.
Das TYCOON-System bietet auch die Möglichkeit, neue Klassen direkt im Objektspeicher zu generieren und zu untersuchen, ohne den Umweg über das Dateisystem zu gehen (Reflektion). Auf dieses Verfahren wurde aber verzichtet, um durch die externe Speicherung eine Inspektion des Quelltextes der erzeugten Klassen zu ermöglichen. Außerdem lassen sich die erzeugten Klassen einfacher an indiviuelle Bedürfnisse anpassen, wenn ein direkter Zugriff auf den Quelltext auch außerhalb des Objektspeichers möglich ist.
Auf der Basis der generierten Datensatzklassen wird dann das eigentliche Anwendungsprogramm entwickelt. Dies kann z.B. durch Delegation datenbankspezifischer Operationen an Datensatzobjekte geschehen oder durch die Erweiterung der Datensatzobjekte entweder in Subklassen oder direkt durch Ergänzung des erzeugten Quellcodes. Die direkte Manipulation des erzeugten Quellcodes ist allerdings nicht zu empfehlen, da dieser bei einer Neugenerierung der Datensatzklassen durch den Klassengenerator überschrieben werden kann. Eine Neugenerierung kann z.B. bei Veränderungen des Datenbankschemas notwendig werden.

Erzeugung von Klassenbescheibungen

Sobald die `build`-Methode eines `ClassBuilder`-Objekts aufgerufen wird, beginnt die eigentliche Klassenerzeugung. Sie besteht aus den folgenden Teilschritten

1. Initialisierung
2. Erzeugung spaltenspezifischer Methoden
3. Übersetzung von eigenen Fremdschlüsseln in Methoden
4. Übersetzung von Fremdschlüsseln anderer Tabellen
5. Integration von Beziehungstabellen

Alle Eigenschaften der zu generierenden Klasse und ihrer Metaklasse werden direkt aus den Daten abgeleitet, die bei der Konfiguration des Klassengenerators angegeben wurden. Weitere Benutzereingriffe sind nicht erforderlich.

Initialisierung

In der Initialisierungsphase werden Basiseigenschaften der neuen Klasse und ihrer Metaklasse festgelegt wie z.B. ihr Name und die Rümpfe einiger Standardmethoden, die jede konkrete Datensatzklasse implementieren muß. Dazu gehören die Methoden `flushParentRecords` und `flushChildRecords`, die für die korrekte Reihenfolge von `flush`-Vorgängen nötig sind, wenn ein Datensatzobjekt an Fremdschlüsselbeziehungen zu anderen Objekten beteiligt ist.
Außerdem wird der Rumpf der "="-Methode konstruiert, die für die Cacheverwaltung und die Identitätssicherung der Datensatzobjekte von essentieller Bedeutung ist (vgl. Abschnitt 6.2.2). Schließlich wird der Rumpf der privaten Methoden `_setParameters` und `_getCurrentValues` erzeugt, die für den Austausch der Attributwerte des Datensatzes mit der Datenbank zuständig sind (vgl. Abschnitt 6.1.4). Für die Metaklasse werden Methodenrümpfe für den primärschlüsselgestützten Zugriff auf einzelne Datensätze erzeugt (vgl. Abschnitt 6.2.1).

Erzeugung spaltenspezifischer Methoden

In diesem Schritt analysiert der `ClassBuilder` die Spaltenbeschreibungen der Datenbanktabelle, auf der die zu generierende Klasse basiert. Diese Beschreibungen erhält er aus dem `SQLTableDescription`-Objekt, das bei seiner Erzeugung übergeben wird. Für jede Spalte der Datenbanktabelle werden der Klassenbeschreibung die folgenden Elemente hinzugefügt:

- ein privater Slot, in dem der Attributwert gespeichert wird,
- eine Methode zum Lesen des Attributwertes,
- eine Methode zum Ändern des Attributwertes,
- ein Methodenaufruf innerhalb der Methode `_setParameters`, durch den der aktuelle Attributwert an die Datenbank gesendet wird,
- ein Methodenaufruf innerhalb der Methode `_getCurrentValues`, über den der aktuelle Attributwert aus der Datenbank gelesen werden kann.

Ist das jeweilige Attribut Teil des Primärschlüssels, so wird keine Methode zum Ändern des Attributwertes erzeugt. Stattdessen werden für Primärschlüsselattribute noch einige andere Elemente erzeugt.

- eine Vergleichsoperation innerhalb der "="-Methode, um festzustellen, ob die Primärschlüsselattribute zweier Objekte gleich sind,
- eine Erweiterung der Methoden der Metaklasse um das aktuelle Primärschlüsselattribut. Da die Primärschlüsselwerte eines Datensatzes nach dessen Erzeugung nicht mehr verändert werden können, werden die Methoden der Metaklasse um entsprechende Parameter erweitert, so daß der Primärschlüssel bei der Erzeugung bzw. beim erstmaligen Zugriff auf einen Datensatz angegeben werden kann.

Diese spaltenspezifischen Methoden werden für jede Datensatzklasse erzeugt. Wird der Generierungsvorgang an dieser Stelle abgebrochen, so lassen sich bereits sämtliche Attribute einer Datenbanktabelle anzeigen und manipulieren, indem man sie wie gewöhnliche öffentliche Slots einer Klasse benutzt. Der Name einer Tabellenspalte dient dabei als Name des korrespondierenden Slots bzw. der korrespondierenden Methoden. Für die Beispielklasse `Person` ergeben sich damit folgende spaltenspezifische Methoden:

```
class Person super Record

public methods

personalnummer():Int              (* Schlüsselattribut - nur lesen         *)

nachname():String                 (* gewöhnliches Attribut - lesen         *)
"nachname:="(value:String):Void   (*                        und schreiben *)

vorname():String
"vorname:="(value:String):Void
```

```
abteilung():Int
"abteilung:="(value:Int):Void

geburtsdatum():Date
"geburtsdatum:="(value:Date):Void
```

Wenn in der `SQLTableDescription` noch weitere Daten enthalten sind, können im nächsten Schritt zusätzliche, komfortablere Methoden generiert werden.

Übersetzung von eigenen Fremdschlüsseln in Methoden

Wenn die Beschreibung einer Datenbanktabelle Fremdschlüsseldefinitionen enthält, so deuten diese auf Beziehungen zu anderen Tabellen hin. Ein konkreter Fremdschlüsselwert identifiziert genau einen Datensatz der referenzierten Tabelle über seinen Primärschlüssel. Der `ClassBuilder` überprüft nun, ob im Klassengenerator ein weiterer `ClassBuilder` registriert ist, der eine Klasse zu derjenigen Tabelle erzeugt, auf die der Fremdschlüssel verweist. Existiert ein solcher `ClassBuilder`, so werden der aktuellen Klasse Methoden zur Navigation über diesen Fremdschüssel hinzugefügt. Eine dieser Methoden ruft die `lookup`-Methode der referenzierten Klasse mit den aktuellen Werten des Fremdschlüssels auf und liefert so ein Objekt der referenzierten Klasse. Dadurch wird eine Objekt-zu-Objekt-Navigation "entlang" eines Fremdschlüssels realisiert. Falls der Fremdschlüssel nicht Teil des eigenen Primärschlüssels ist, wird außerdem eine Methode zum Ändern des Fremdschlüsselwertes erzeugt, die als Parameter ein Objekt der referenzierten Klasse akzeptiert und den Fremdschlüssel auf den Primärschlüsselwert des übergebenen Objekts setzt. Diese beiden Methoden zur Manipulation von Fremdschlüsselwerten direkt über die vom Fremdschlüssel referenzierten Objekte erhalten als Bezeichner den Namen des Fremdschlüssels, wie er im *Data Dictionary* der Datenbank gespeichert ist.
Weiterhin wird die `flushParentRecords`-Methode um einen Aufruf ergänzt, der eine `flush`-Operation des referenzierten Objekts veranlaßt. Die in diesem Kapitel verwendete Beispielklasse `Person` besitzt einen Fremdschlüssel `arbeitet_in`. Daraus erzeugt der Klassengenerator die beiden folgenden Methoden für den Zugriff auf die referenzierte Abteilung:

```
arbeitet_in():Abteilung
{
  _lock,                (* Datensatz sperren, falls nötig *)

  Abteilung[_abteilung] (* In der Klasse Abteilung das Objekt  *)
                        (* mit dem Primärschlüsselwert des     *)
                        (* privaten Slots _abteilung aufsuchen *)
                        (* und zurückgeben.                    *)
}

"arbeitet_in:="(value:Abteilung):Void
{
  _modify,              (* Modifikation/Zustandsänderung signalisieren *)

  value.isNotNil                    (* Wurde eine Abteilung übergeben ?    *)
  ? { _abteilung := value.nummer } (* ja - setze privaten Slot _abteilung *)
```

```
                                    (*      auf Primärschlüsselwert der    *)
                                    (*      übergebenen Abteilung          *)
  : { _abteilung := nil }           (* nein - Abteilung ist leeres Objekt, *)
                                    (*        Datensatz referenziert keine *)
                                    (*        Abteilung                    *)
}
```

Die Metaklasse der aktuellen Klasse erhält schließlich eine Methode, die sämtliche Datensatzobjekte liefert, die den gleichen Datensatz der anderen Klasse referenzieren. Letztere Methode ist besonders dann nützlich, wenn entlang von 1:N-Beziehungen "entgegen" der Fremdschlüsselbeziehung navigiert werden soll. Im Fall der Beispielklasse `Person` enthielte dann die Metaklasse `PersonClass` folgende Methode (zur Übersicht nur die Signatur):

```
readerWhereArbeitet_inIs(abteilung:Abteilung):Reader(Person)
```

Der Name der Methode läßt sich dabei interpretieren als "Erzeuge einen `Reader` aller Personen, deren Fremdschlüssel `arbeitet_in` gerade das Objekt `abteilung` referenziert".

Übersetzung von Fremdschlüsseln anderer Tabellen

Nachdem der `ClassBuilder` die eigene Tabellenbeschreibung auf Schlüssel untersucht hat, die auf andere Tabellen verweisen, überprüft er im nächsten Schritt, ob `ClassBuilder` für andere Datenbanktabellen existieren, die ihrerseits Referenzen auf diese Datenbanktabelle enthalten. Sind solche `ClassBuilder` vorhanden, so können sich ein oder mehrere Datensätze der fremden Tabelle auf Datensätze in der eigenen Tabelle beziehen. Die aktuelle Klasse wird dann um eine Methode erweitert, die alle Objekte der fremden Klasse liefert, die sich auf ein Objekt der aktuellen Klasse beziehen. Dazu wird eine Methode der fremden Metaklasse aufgerufen, die genau diese Menge von Objekten liefert. Ist der Fremdschlüssel in der fremden Klasse ein eindeutiges Attribut, so besteht zwischen den beiden Klassen eine 1:1-Beziehung. In diesem Fall liefert die besagte Methode nicht eine Menge von Objekten, sondern genau das eine Objekt, zu dem die Beziehung besteht.
Auch hier wieder ein Beispiel: Die Tabelle für Personen enthält den schon bekannten Fremdschlüssel `arbeitet_in`, der die Abteilung identifiziert, in der ein Angestellter arbeitet. Der `ClassBuilder` der Klasse `Abteilungen` erzeugt dann zu dem Fremdschlüssel `arbeitet_in` eine Methode, die zu einem Objekt der Klasse `Abteilungen` alle Angestellten liefert, die in dieser Abteilung arbeiten. Der Name dieser neuen Methode wird aus dem Namen der fremden Klasse und dem Fremdschlüssel in der fremden Klasse zusammengesetzt. Da die Navigationsrichtung "entgegen" der Fremdschlüsselrichtung verläuft, wird der Methode außerdem das Präfix `inverse` vorangestellt. Für das Beispiel ergäbe sich dann folgende Methode

```
inversePersonArbeitet_in:Reader(Person)
{ Person.readerWhereArbeitet_inIs(self) }
```

Ein Aufruf dieser inversen Navigationsmethode wird anschließend in die `flushChildRecords`-Methode eingefügt. Dadurch wird sichergestellt, daß beim Löschen eines Datensatzes zunächst ein `Flush` aller Kind-Datensätze ausgelöst wird. So werden die Kind-Datensätze in der Datenbank gelöscht, bzw. ihre Referenzen auf andere Elternobjekte umgesetzt, bevor das Elternobjekt selbst in der Datenbank gelöscht wird.

Die Methode `inversePersonArbeitet_in` implementiert die in der ursprünglichen Definition des Datenmodells (vgl. Abbildung 6.6) beschriebene Rolle `Angestellter` (Kardinalität: 1..*) der Assoziation `Arbeit`. Während sich die Rolle `Arbeitsplatz` direkt aus dem Fremdschlüssel `arbeitet_in` ergibt, fehlt in der Datenbank ein Mechanismus zur Benennung der inversen Beziehung. Der momentan implementierte Generator kann Methoden ausschließlich aufgrund der Daten erzeugen und benennen, die er aus der Datenbank erhält. Daher wirkt der Methodenbezeichner etwas künstlich. Im obigen Fall wäre z.B. eine Umbenennung der Methode `inversePersonArbeitet_in` in `angestellte` sinnvoll. Dazu ist aber das Wissen des Anwenders über die Semantik dieser Beziehung erforderlich, so daß dies dem Generator entweder vor dem Start der Klassenerzeugung mitgeteilt werden müßte oder nachträglich in den erzeugten Klassen manuell geändert werden muß. Falls möglich, wird zu einer inversen `get`-Methode auch eine entsprechende `set`-Methode erzeugt. Da durch die `set`-Methode Fremdschlüsselwerte der fremden Klasse verändert werden, wird die `set`-Methode nur erzeugt, wenn eine Manipulation der fremden Attribute zulässig ist.

Integration von Beziehungstabellen

Neben den bisher beschriebenen 1:1- und 1:N-Beziehungen sind bei der Datenmodellierung häufig auch komplexere Beziehungen anzutreffen, die sich nicht durch einfache Fremdschlüssel modellieren lassen. Zu dieser Klasse gehören unter den binären Beziehungen die N:M-Beziehungen, sowie alle Beziehungen, an denen mehr als zwei Klassen teilnehmen (k-näre Beziehungen, k> 2). Solche Beziehungen werden durch separate Tabellen modelliert. Der Primärschlüssel dieser Beziehungstabellen besteht aus Fremdschlüsselattributen, über die die an der Beziehung partizipierenden Objekte identifiziert werden. Zu solchen Beziehungstabellen werden zunächst eigene Klassen generiert. Dabei werden dieselben Schritte durchlaufen, wie bei allen anderen Datensatzklassen auch.
Bei der Generierung der Klassen, die an der Beziehung teilnehmen, wird aber zusätzlich untersucht, ob die Beziehungstabelle eine binäre-Beziehung modelliert. Dies ist daran zu erkennen, daß der Primärschlüssel der Beziehungsklasse gerade aus genau zwei Fremdschlüsseln auf andere Tabellen besteht. Ist das der Fall, so erhalten die an der Beziehung beteiligten Klassen nach dem bisherigen Verfahren eine Methode, die all diejenigen Objekte der Beziehungsklasse liefert, an der jeweils ein Datensatz beteiligt ist. Der Benutzer erhält dann über die Navigationsmethoden der Beziehungsklasse die Objekte der jeweils anderen an der Beziehung beteiligten Klasse. Wird eine solche Beziehung erkannt, so werden nun in den beiden an der Beziehung beteiligten Klassen Methoden erzeugt, die *direkt* die Menge der Objekte der jeweils anderen Klasse liefern. Eine N:M-Beziehung wird somit durch zwei 1:N-Beziehungen in den beteiligten Klassen repräsentiert.
Als Beispiel für eine solche N:M-Beziehung möge die in Abbildung 6.9 dargestellte Beziehung `Verkauf` dienen. Jeder Artikel kann in mehreren Abteilungen verkauft werden und in jeder Abteilung können verschiedene Artikel verkauft werden. Neben den beiden Tabellen `Artikel` und `Abteilung` wird nun in der relationalen Datenbank eine Tabelle `Verkaeufe` zur Repräsentierung dieser Beziehung angelegt.

```
create table verkaeufe (
  artikel   int, abteilung int,
  primary key (artikel,abteilung),
  constraint fs_artikel foreign key (artikel) references artikel,
  constraint fs_abteilung foreign key (abteilung) references abteilungen )
```

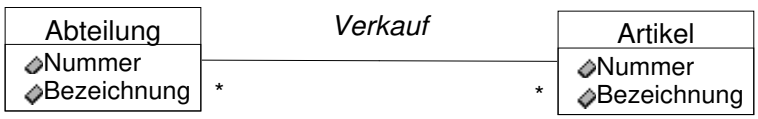

Abbildung 6.9: N:M-Beziehung

Um nun zu herauszufinden, in welchen Abteilungen ein Artikel verkauft wird, erzeugt man sich unter Verwendung der bereits bekannten Methoden einen `Reader` aller Verkaufsobjekte, indem man die Methode `inverseVerkaufFs_artikel():Reader(Verkauf)` aufruft. Für jedes Verkaufsobjekt dieses `Reader`s ist dann die Methode `fs_abteilung():Abteilung` aufzurufen. Da die `ClassBuilder` der Klassen `Artikel` und `Abteilung` die durch die Verkäufe-Tabelle repräsentierte N:M-Beziehung erkennen, werden jeweils Methoden erzeugt, die diesen Weg abkürzen. So erhält z.B. die Klasse `Artikel` eine Methode `verkaufFs_abteilung():Reader(Abteilung)`, die direkt einen `Reader` aller Abteilungen liefert, in denen ein Artikel verkauft wird.

6.4 Verbindungen zwischen Datensatzklasse und Datenbanken

Im Abschnitt 6.3 wird die Erzeugung von Klassen für Datenbankzugriffe erläutert. Nach Ende des Generierungsvorgangs stehen diese Klassen entweder im Dateisystem des Rechners oder bereits als übersetzte Objekte im Objektspeicher des TYCOON-Systems zur Verfügung. Die Objekte dieser Klassen können mit einer Datenbank kommunizieren. Diese Kommunikation kann über eine oder mehrere Datenbankverbindungen ablaufen. Die Datensatzklassen stellen die Verbindung zur Datenbank nicht selber her. Dies hat die folgenden Gründe:

1. Datenbankänderungen laufen in der Regel innerhalb größerer Transaktionen ab. D.h. in einer Datenbankanwendung werden oft mehrere Änderungen vorgenommen, die in einem inhaltlichen Zusammenhang stehen und die entweder vollständig oder gar nicht ausgeführt werden sollen. Um die Mechanismen zur Transaktionsverwaltung nutzen zu können, die von einer relationalen Datenbank zur Verfügung gestellt werden, müssen Änderungen an Datensatzobjekten einer Transaktion zugeordnet werden können. Dies geschieht dadurch, daß jeder Datensatzklasse eine Datenbankverbindung zugeordnet wird. Alle Änderungsoperationen, die über dieselbe Datenbankverbindung ablaufen, gehören damit zur selben Datenbanktransaktion.
   Die in dieser Arbeit verwendeten Programmierschnittstellen erlauben pro offener Datenbankverbindung nur eine einzige laufende Transaktion. Wird die aktuelle Datenbanktransaktion beendet, so startet implizit eine neue Transaktion. Eine Transaktion endet spätestens mit dem Ende der Datenbankverbindung. Transaktionen, die über mehrere verschiedene Datenbankverbindungen ablaufen, sind mit den verwendeten Schnittstellen nicht möglich.

2. Die Datensatzklassen können mit Hilfe einer "Entwicklungsdatenbank" generiert werden und können später ohne Änderung mit jeder beliebigen Datenbank genutzt werden, die ein äquivalentes Datenbankschema enthält. Unter "äquivalentem Datenbankschema" ist

in diesem Zusammenhang ein Schema zu verstehen, das die gleichen Tabellenstrukturen (gleiche Spaltennamen, SQL-Datentypen und Fremdschlüsseldefinitionen) enthält, wie sie bei der Klassengenerierung verwendet wurden. Die Namen der Datenbanktabellen und auch der Name des Datenbankbenutzers, der diese Tabellen erzeugt hat, dürfen sich von denen der Entwicklungsdatenbank unterscheiden.
Dies ist insbesondere dann von Vorteil, wenn Datenbankapplikationen nicht zusammen mit derselben Datenbank installiert werden, mit der sie entwickelt und getestet wurden.

3. Jede Datensatzklasse kann mit einer eigenen Datenbankverbindung assoziiert werden. Dadurch lassen sich Operationen wie das Schreiben des Cache-Inhaltes in die Datenbank parallelisieren. Verschiedene Datensatzklassen können damit sogar Verbindungen zu unterschiedlichen Datenbanken herstellen. Mehrere, auf unterschiedliche physikalische Datenbanken verteilte Datenbankschemata lassen sich dadurch zu einem einzigen logischen Datenbankschema aggregieren.
Allerdings lassen sich im letzteren Fall die Methoden zur Navigation zwischen Datensatzobjekten unter Umständen nur eingeschränkt nutzen, da in der Regel keine datenbankübergreifenden Fremdschlüsseldefinitionen möglich sind. Außerdem sind keine logischen Transaktionen möglich, die mehr als eine Datenbankverbindung nutzen, da den verwendeten Programmierschnittstellen die hierzu nötigen Mittel fehlen.

Die Herstellung und Verwaltung von Datenbankverbindungen übernehmen Objekte der Klasse `DB`. Diese werden mit den Daten initialisiert, die nötig sind, um eine Verbindung zu einer Datenbank herzustellen. Dies sind insbesondere der Name und der Ort der Datenbank, der Name und das Paßwort des Datenbankbenutzers. Die wichtigsten Methoden der Klasse `DB` werden im folgenden kurz vorgestellt.

registerRecordClass: Dieser Methode werden eine Datensatzklasse sowie Name und Besitzer der zugehörigen Datenbanktabelle als Parameter übergeben. Diese Methode erzeugt daraufhin ein Objekt der Klasse `Table` (vgl. Abbildung 5.7) und initialisiert die Datensatzklasse mit diesem Objekt. Die Datensatzklasse selber wird einer internen Liste des `DB`-Objekts hinzugefügt.

unregisterRecordClass entfernt eine Datensatzklasse von der internen Liste des `DB`-Objekts. Dabei werden alle noch im Cache dieser Klasse gespeicherten Änderungen in die Datenbank geschrieben und das assoziierte `Table`-Objekt geschlossen.

commitWork schickt jeder registrierten Datensatzklasse eine `flush`-Nachricht, so daß alle noch nicht gesicherten Änderungen an die Datenbank geschickt werden. Treten dabei keine Fehler wie Verletzungen von Integritätsbedingungen auf, wird die `commit`-Methode der Datenbank-Schnittstelle aufgerufen. Danach sind alle Änderungen persistent und alle Datensatzsperren werden freigegeben. Implizit wird eine neue Datenbanktransaktion gestartet.

rollbackWork führt die `rollback`-Methode der Datenbankschnittstelle aus und gibt alle Datensatzsperren frei. Alle Änderungen seit Beginn der aktuellen Datenbanktransaktion werden damit rückgängig gemacht.

close: Diese Methode erbt ein Objekt der Klasse `DB` von der Superklasse `Resource`. Zusätzlich zur ererbten Funktionalität, die am Ende dieser Auflistung näher beschrieben wird, wird die Methode `commitWork` ausgeführt und die Datenbankverbindung geschlossen.

open : Diese ebenfalls von `Resource` geerbte Methode stellt eine vorher geschlossene Datenbankverbindung wieder her. Nach dem Aufbau der Verbindung wird allen registrierten Datensatzklassen eine `refresh`-Nachricht geschickt. Dadurch werden alle Objekte, die sich noch im Cache der jeweiligen Datensatzklasse befinden, in den Zustand gebracht, der aktuell beim Öffnen der Verbindung in der Datenbank vorliegt. Dadurch wird die Konsistenz zwischen Cache und Datenbank wiederhergestellt.

Bei den beiden Methoden `open` und `close` ist schon angedeutet, daß die Klasse `DB` von einer Klasse namens `Resource` erbt. Die Klasse Resource realisiert in TYCOON ein Protokoll für den Umgang mit externen Diensten (Ressourcen), die sich der Kontrolle des persistenten Objektspeichers ganz oder teilweise entziehen (vgl. Abschnitt 5.1.3). Auch relationale Datenbanken sind solch ein externer Dienst.
Ein `DB`-Objekt nutzt diese Klasse in folgender Weise: Wird der TYCOON-Objektspeicher gesichert, so wird ein `commitWork` ausgeführt. Danach befinden sich die Caches der Datensatzklassen und die Datenbank in einem konsistenten Zustand. Die Datenbankverbindung inklusive der aktiven SQL-Befehle und Cursor braucht nicht beendet oder gesondert behandelt werden, da sie ihrerseits `Resource`-Objekte sind. Wird nun der so gesicherte Objektspeicher wieder gestartet, so wird durch Aufruf der `reopen`-Methode durch den TYCOON-`ResourceManager` die Datenbankverbindung wiederhergestellt und allen Datensatzklassen eine `refresh`-Nachricht gesendet. Hat sich der Inhalt der Datenbank geändert, so ändert sich beim Neustart auch der Zustand der noch im Cache befindlichen Datensatzobjekte. Dadurch ist der Gesamtzustand des Systems zwar u.U. nicht mehr mit dem Zustand der letzten Sicherung identisch, aber Cache und Datenbank sind wieder miteinander synchronisiert.
Nach der Herstellung einer Datenbankverbindung können Objekte der Datensatzklassen erzeugt und manipuliert werden. All ihre Methoden bleiben uneingeschränkt nutzbar, solange die Verbindung zur Datenbank besteht. Wird die Datenbankverbindung geschlossen, so kann auf die Attribute der im Cache befindlichen Objekte weiterhin lesend zugegriffen werden. Schreibende Zugriffe oder Zugriffe, die die Auswertung von Datenbankanfragen erfordern, sind in diesem Zustand nicht möglich. Eine Wiederherstellung der Datenbankverbindung hebt diese Restriktionen auf.

Kapitel 7

Ergebnisse

Im Kapitel 3 werden Anforderungen formuliert, die ein objektorientierter Datenbankzugriff erfüllen soll. Inwieweit diese Anforderungen von der für TYCOON realisierten Lösung erfüllt werden, ist Gegenstand dieses Kapitels. Dazu werden zunächst die aufgestellten Anforderungen den entwickelten Lösungen bzw. Lösungsansätzen gegenübergestellt und mit den vorgestellten kommerziellen Lösungen verglichen (Abschnitt 7.1). Anschließend wird anhand eines Beispiels der praktische Einsatz der entwickelten Klassenbibliothek demonstriert (Abschnitt 7.2).

7.1 Leistungen der implementierten Lösung

Die folgenden Abschnitte fassen die wesentlichen Eigenschaften der implementierten Lösung für die im Kapitel 3 formulierten Anforderungen an einen objektorientierten Datenbankzugriff zusammen.

- Höhere Abstraktionen für Inhalte der Datenbank (Abschnitt 7.1.1)
- Implizite Erzeugung von SQL-Anfragen (Abschnitt 7.1.2)
- Minimierung der Kommunikation mit der Datenbank (Abschnitt 7.1.3)
- Isolation (Abschnitt 7.1.4)

Ein Vergleich der für TYCOON realisierten Lösung mit den im Kapitel 4 vorgestellten kommerziellen Werkzeugen (Abschnitt 7.1.5) beschließt diesen Abschnitt.

7.1.1 Höhere Abstraktionen für Inhalte der Datenbank

Für elementare Konzepte der relationalen Datenbank wie Tabellen, Datensätze und Integritätsbedingungen müssen korrespondierende Konzepte in der objektorientierten Welt bereitgestellt werden.

Tabellen

Tabellen finden sich in zwei unterschiedlichen Ausprägungen im TYCOON-System wieder. Die niedrigere Abstraktion stellen die Objekte der Klasse `Table` dar. Diese Klasse bündelt alle Standardoperationen, die für den Zugriff auf eine Datenbanktabelle nötig sind. Dazu gehören

das Einfügen, Ändern und Löschen von Datensätzen, sowie Methoden zur Auswertung von Anfragen. Die nächst höhere Abstraktionsschicht besteht aus den Datensatzklassen deren Verhalten in Subklassen der abstrakten Metaklasse `RecordClass` implementiert wird. In diesen Klassen werden alle Methoden gebündelt, die zur Verwaltung von Datensätzen nötig sind. Hier sind insbesondere die Methoden zur Erzeugung und zum Wiederauffinden von Datensätzen zu nennen. Die Datensatzklassen greifen dabei auf Methoden der oben erwähnten `Table`-Objekte zurück.

Datensätze

Zeilen einer Tabelle werden durch Objekte der Klasse `Record` bzw. deren Subklassen auf TYCOON-Objekte abgebildet. Ein solches Datensatzobjekt enthält mindestens zu jedem Tabellenattribut einen korrespondierenden Slot. Der Typ des Slots wird aus dem in der Datenbank verwendeten SQL-Datentyp abgeleitet. Für die gängigsten SQL-Datentypen, wie Zeichenketten, Zahlenwerte, Datums- und Zeitangaben, aber auch für die zur Speicherung von Bild, Ton und Video verwendeten BLOBs (Binary Large Objects), stehen korrespondierende TYCOON-Typen zur Verfügung.
Die Inhalte dieser Slots können mit den für öffentliche Slots üblichen Methoden gelesen und beschrieben werden. Die generierten Datensatzklassen überschreiben diese Standardmethoden, um vor den eigentlichen Manipulationen konsistenzsichernde Maßnahmen veranlassen zu können (Sperren). Über das `Table`-Objekt, mit dem jede Datensatzklasse assoziiert ist, kann ein Datensatzobjekt die Inhalte seiner Slots mit der Datenbank austauschen. Auch das Löschen eines Datensatzes in der Datenbank wird durch den Aufruf einer Methode des Datensatzobjekts veranlaßt. Da das TYCOON-Objekt selber nicht unmittelbar aus dem Objektspeicher gelöscht wird, sondern sich weiterhin im Cache befindet, ist diese Löschoperation reversibel. Im Gegensatz zum direkten Datenbankzugriff ist die Rücknahme einer Löschoperation ohne einen Abbruch der gesamten Datenbanktransaktion möglich.

Der Primärschlüssel

Eine wichtige Voraussetzung, um ein Datensatzobjekt eindeutig einem Datensatz zuordnen zu können, ist die Existenz eines Primärschlüssels in der Datenbanktabelle. Dieser wird nicht in Form einer eigenen Repräsentation ins TYCOON-System abgebildet. Seine Existenz äußert sich vielmehr in einigen implementatorischen Details der Datensatzobjekte und ihrer Klasse. So dürfen Primärschlüsselwerte eines Datensatzobjekts nicht verändert werden. Schreibmethoden für Primärschlüsselwerte werden daher nicht generiert. Der Primärschlüssel eines Datensatzobjekts kann nur unmittelbar bei der Objektinstantiierung angegeben werden.

Fremdschlüssel

Enthält eine Datenbanktabelle referentielle Integritätsbedingungen, so werden diese vom Klassengenerator in Methoden übersetzt, die eine Navigation zwischen Objekten der beteiligten Klassen zulassen. Eine Klasse `A`, in deren zugehöriger Tabelle ein Fremdschlüssel definiert ist, erhält Methoden, mit denen direkt das durch einen Fremdschlüssel identifizierte Objekt einer Klasse `B` erreicht oder durch ein anderes ersetzt werden kann. Eine Klasse `B`, deren Datensätze durch den Fremdschlüssel referenziert werden, enthält dagegen eine Methode, die zu einem Objekt `b:B` alle Objekte der Klasse `A` liefert, deren Fremdschlüssel sich auf das Objekt `b` beziehen.

7.1.2 Implizite Erzeugung von SQL-Anfragen

Für die Benutzung der vom Klassengenerator erzeugten Datensatzklassen sind keine SQL-Kenntnisse erforderlich. Der SQL-Code, der für die Kommunikation mit der Datenbank notwendig ist, wird durch Methoden der Klasse `Table` gekapselt und bei der Instantiierung eines `Table`-Objekts automatisch erzeugt. Da die Objekte der Klasse `Table` nur einige standardisierte SQL-Befehle zur Datensatzmanipulation und zur Auflösung von Fremdschlüsselbeziehungen zwischen Datensätzen bereitstellen, steht nicht mehr die volle Mächtigkeit der Sprache SQL zur Verfügung.
Diese Mächtigkeit ist allerdings auch nur noch bedingt erforderlich. *Joins*, also Anfragen, die Fremdschlüsselbeziehungen zwischen Tabellen ausnutzen, werden implizit durch die Navigationsmethoden der beteiligten Klassen realisiert. Die Auswahl von Datensatzobjekten einer Klasse kann optional durch Angabe einer SQL-`where`-Klausel eingeschränkt werden. *Joins*, die nicht mit Hilfe von Fremdschlüsselattributen realisiert werden, sind damit allerdings nicht möglich. Auch Anfragen, deren Ergebnis aus Werten mehrerer Tabellen besteht, lassen sich nicht mehr direkt formulieren. Für den vollen Sprachumfang von SQL muß dann auf Programmierschnittstellen niederer Abstraktionsschichten zurückgegriffen werden.

7.1.3 Minimierung der Kommunikation mit der Datenbank

Änderungen an Datensatzobjekten werden zunächst ausschließlich innerhalb des TYCOON-Objektspeichers vorgenommen. Die erforderlichen Datenbankoperationen ergeben sich aus dem aktuellen Zustand des jeweiligen Datensatzobjekts. Diese werden erst dann ausgeführt, wenn eines der folgenden Ereignisse auftritt:

- Ein Datensatzobjekt ist nicht mehr erreichbar und wird aus dem Cache entfernt.
- Die laufende Datenbanktransaktion wird durch ein `commitWork` beendet.
- Die Datenbankverbindung wird geschlossen.
- Der Objektspeicher wird gesichert.
- Der Benutzer löst direkt oder indirekt eine Anfrage aus, deren Ergebnis nur nach Ausführung der noch offenen Datenbankmanipulationen korrekt ist.

Der letzte Punkt bedarf einiger Erläuterungen. Wenn ein Benutzer einer Datensatzklasse eine Methode aufruft, um eine Menge von Datensatzobjekten zu erhalten, so stellt er im Sinne der Datenbankterminologie eine Anfrage. Die hier implementierte Klassenbibliothek verfügt nicht über eigene Mechanismen zur Auswertung von Anfragen, sondern delegiert diese Aufgabe an die darunterliegende relationale Datenbank. Das Datenbanksystem wiederum kann Anfragen nur aufgrund seiner eigenen Datenbasis beantworten. Der Benutzer einer Datensatzklasse ist sich aber in der Regel nicht bewußt, daß seine Änderungen an einzelnen Datensätzen eventuell noch gar nicht in die Datenbank übernommen wurden und so von der Datenbank nicht in die Auswertung der Anfrage miteinbezogen werden können. Aus diesem Grund übertragen vor der Auswertung einer solchen Anfrage alle potentiell betroffenen Objekte ihren aktuellen Zustand in die Datenbank. Das Anfrageergebnis spiegelt danach den aktuellen Zustand der Datenbank inklusive der letzten Änderungen des Benutzers wieder.

7.1.4 Isolation

Relationale Datenbanksysteme sind in der Regel mehrbenutzerfähig. Um zu verhindern, daß andere Datenbankbenutzer gerade die Daten manipulieren, die auch aktuell im TYCOON-System bearbeitet werden, müssen diese für andere Datenbankbenutzer gesperrt werden. Hierzu werden Mechanismen genutzt, die das Datenbanksystem zur Verfügung stellt.
So werden innerhalb einer Transaktion gewöhnlich all die Datensätze für andere Datenbankbenutzer gesperrt, an denen Änderungen vorgenommen werden. Die TYCOON-Klassen verhalten sich noch etwas restriktiver, indem sie Datensätze bereits bei lesendem Zugriff sperren. Die Grundannahme ist, daß ein Datensatz, zu dem im TYCOON-System ein Datensatzobjekt erzeugt wird, nur durch das TYCOON-System selber verändert werden kann. Dazu muß sichergestellt sein, daß sich der korrespondierende Datensatz nicht unabhängig vom Datensatzobjekt verändern kann. Daher wird jeder Datensatz, zu dem ein TYCOON-Datensatzobjekt erzeugt wird, in der Datenbank gesperrt, sobald lesend oder schreibend auf das Objekt zugegriffen wird. Damit ist gewährleistet, daß das Datensatzobjekt entweder dem derzeitigen Zustand der Datenbank entspricht oder nur solche Änderungen beinhaltet, die über das TYCOON-System selbst an diesem Objekt vorgenommen wurden.
Dieses Vorgehen birgt die Gefahr, mit der Zeit einen großen Teil der Datenbank zu sperren, da Objekte erst dann aus dem Cache gelöscht werden, wenn sie nicht mehr erreichbar sind und durch den *Garbage Collector* aus dem Objektspeicher entfernt werden. Solange Datensatzobjekte erreichbar bleiben, blieben sie demnach auch in der Datenbank gesperrt. Um dieses Problem ein wenig zu entschärfen, bleiben Datensatzobjekte bzw. ihre korrespondierenden Datensätze nicht ständig gesperrt. Immer dann, wenn ein Benutzer die laufende Datenbanktransaktion beendet, werden alle Sperren freigegeben. Solange keine Zugriffe auf die im Cache befindlichen Objekte stattfinden, werden keine neuen Datenbanksperren angefordert. Erst, wenn erneut ein Lese- oder Schreibzugriff auf ein Datensatzobjekt stattfindet, wird dieses Objekt wieder in der Datenbank gesperrt. Das Objekt enthält danach die Werte, die zum Zeitpunkt der Zugriffsoperation in der Datenbank vorgelegen haben. Durch dieses Sperrverfahren bleiben Datensätze nur solange gesperrt, wie innerhalb des TYCOON-Systems auf sie zugegriffen wird. Dies vermindert die Gefahr einer ständigen Blockierung der Datenbank durch Datensatzobjekte innerhalb des TYCOON-Objektspeichers. Andererseits muß sich der Benutzer der Datenbankobjekte, wie bei jeder anderen Datenbankanwendung auch, darüber im klaren sein, daß nach dem Ende seiner Datenbanktransaktion auch andere Datenbankbenutzer Zugriff auf die entsprechenden Datensätze erhalten.

7.1.5 Vergleich mit kommerziellen Lösungen

Die Funktionalität der für TYCOON implementierten objektrelationalen Klassenbibliothek vereint zahlreiche Eigenschaften der in Kapitel 4 vorgestellten kommerziellen Produkte. Der Leistungsumfang der Klassengenerierung ist am ehesten mit dem von ONTOS und PERSISTENCE zu vergleichen. Neben der Abbildung von Attributen lassen sich auch 1:1-, 1:N- und N:M-Beziehungen abbilden. ONTOS bietet dem Benutzer eine größere Flexibilität bei der Klassengenerierung, da sich die vom Generator erkannten Beziehungen interaktiv manipulieren lassen, z.B. durch die Wahl aussagekräftigerer Bezeichner von Methoden oder in der Wahl der Umsetzung von Beziehungen. Der Benutzer erhält damit die Möglichkeit, diejenige Semantik wiederherzustellen, die durch die Implementierung eines Datenmodells auf einer relationalen Datenbank verloren geht. In TYCOON und PERSISTENCE laufen diese Prozesse

vollautomatisch ab. Das schränkt in gewisser Weise die Flexibitlität bei der Klassengenerierung ein. Andererseits lassen sich durch den Verzicht auf Benutzereingriffe auch von solchen Benutzern Klassen erzeugen, die nur wenige Vorkenntnisse über den Aufbau und die Semantik des relationalen Datenbankschemas besitzen.
Hinsichtlich der Abstraktion von SQL-Befehlen gelten für alle Produkte ähnliche Einschränkungen. Einfache Anfragen, die Objekte genau einer Klasse liefern, sind in den hohen Abstraktionsschichten von ONTOS, PERSISTENCE, OBJECT FACTORY und TYCOON möglich. Benutzer, die den vollen SQL-Sprachumfang nutzen möchten, müssen auf niedere Schnittstellen, wie JDBC, DBTOOLS.H++, oder TYSQL für TYCOON ausweichen. In der OBJECT FACTORY lassen sich zwar Klassen für die Ergebnisse komplexer SQL-Anfragen generieren; nach der Erzeugung der Klasse sind allerdings keine Manipulationen der Anfragestruktur möglich und ein schreibender Zugriff auf die so erzeugten Anfrageergebisse ist ebenfalls nicht mehr möglich.
Das Cache-Konzept für TYCOON ist mit dem von ONTOS vergleichbar. Jeder Datenbankklient erhält seinen eigenen Cache, wobei im Falle von TYCOON unter einem Klienten jeweils ein laufendes TYCOON-System zu verstehen ist. TYCOON ließe sich durch die Nutzung seiner weiteren Dienste, wie z.B. des TYCOON WEB SERVERS, ähnlich wie PERSISTENCE für den parallelen Zugriff durch mehrere Benutzer aufrüsten. Die damit notwendige Implementierung einer TYCOON-seitigen Transaktions-, Sperr- und Versionsverwaltung hätte aber den Umfang dieser Arbeit gesprengt.
Die Isolation von Datenbankoperationen gegen parallele Zugriffe anderer Datenbankbenutzer findet wie in ONTOS durch die Nutzung von Sperrmechanismen des Datenbanksystems statt.
Ein Konzept, das in keinem der kommerziellen Produkte in vergleichbarer Form existiert, ist die orthogonale Integration von Persistenz, wie sie in TYCOON zu finden ist. Diese erlaubt es, den Zustand des Objektspeichers zu nahezu beliebigen Zeitpunkten zu sichern und die Anwendung zu einem späteren Zeitpunkt in diesem Zustand fortzusetzen. Die Persistenz des TYCOON-Objektspeichers erstreckt sich auch auf den Cache von Datenbankobjekten. Der Benutzer einer Anwendung muß sich den *Working Set* von Daten, den er beim Sichern seiner Anwendung besessen hat, nicht mehr manuell durch Abarbeitung neuer Datenbankanfragen erzeugen. Die Daten befinden sich vielmehr beim Neustart immer noch im Cache und werden vollautomatisch auf den neuesten Stand der Datenbank gebracht. Dies ist ein nicht zu unterschätzender Vorteil bei Anwendungen, die häufig mit immer denselben Daten arbeiten.
Zusammenfassend läßt sich feststellen, daß die für TYCOON implementierte Lösung hinsichtlich der Funktionalität in der Klassenerzeugung und der Mächtigkeit der erreichten Abstraktion durchaus mit den kommerziellen Lösungen vergleichbar ist.

7.2 Praktischer Einsatz objektrelationaler Klassen

In diesem Abschnitt werden an einem einfachen Datenbankschema die wesentlichen Eigenschaften der vom TYCOON-Datensatzklassengenerator bereitgestellten Abstraktionen aufgezeigt und mit der Programmierung äquivalenter Abläufe innerhalb niedrigerer Abstraktionsschichten verglichen. Als Beispiel dient dabei ein Datenmodell, das z.B. der Lagerverwaltung eines Handelsunternehmens entstammen könnte (Abbildung 7.1). Die Objekte, mit denen das Unternehmen in diesem Zusammenhang arbeitet, sind Artikel, Lieferanten und Abteilungen, an die Artikel geliefert werden oder in denen sie verkauft werden. `Verkauf` und `Lieferung`

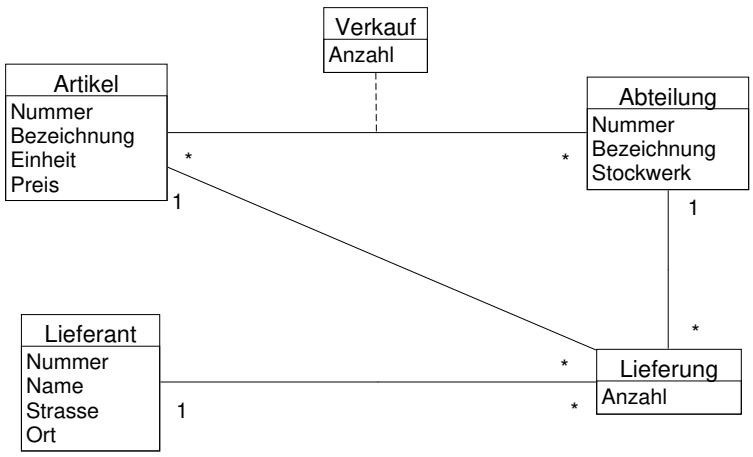

Abbildung 7.1: Beispielmodell

stellen Assoziationen dar. Ein Verkauf ist eine Beziehung zwischen einem Artikel und der Abteilung, in der er verkauft wird. Ein Verkaufsobjekt akkumuliert alle in einer Abteilung zustandegekommenen Verkäufe desselben Artikels in einem Assoziationsattribut `Anzahl`. Gleiches gilt für Lieferungen, wobei hier zusätzlich ein Lieferant an einer Lieferung beteiligt ist. Dieses Modell wird durch die folgenden SQL-Anweisungen in einer relationalen Datenbank implementiert.

```
create table demoartikel(
  nummer number(5)
    constraint ps_demoartikel primary key,
  bezeichnung varchar(20)
    not null
    constraint ea_bezeichnung unique,
  einheit varchar(5)
    not null,
  preis number(7,2)
    not null
)
;
create table demoabteilungen(
  nummer number(5)
    constraint ps_demoabteilungen primary key,
  bezeichnung varchar(20)
    not null
    constraint ea_abteilungsbezeichnung unique,
  stockwerk number(5)
    not null
)
;
```

```
create table demolieferanten(
  nummer number(5)
    constraint ps_demolieferanten primary key,
  name varchar(20)
    not null,
  ort varchar(20)
    not null,
  strasse varchar(20)
    not null
)
;
```

Für jede Basisklasse wird dadurch eine Tabelle mit den entsprechenden Attributen erzeugt. Für die beiden Assoziationen `Verkauf` und `Lieferung` werden eigene Tabellen angelegt, die zum einen Fremdschlüssel für die an der Assoziation teilnehmenden Basistabellen, zum anderen das assoziationsspezifische Attribut `Anzahl` enthalten.

```
create table demolieferungen(
  lieferant number(5)
    constraint geliefert_von references demolieferanten,
  artikel number(5)
    constraint gelieferter_artikel references demoartikel,
  abteilung number(5)
    constraint geliefert_an references demoabteilungen,
  anzahl number(5)
    not null,
  constraint ps_demolieferungen primary key (lieferant,artikel,abteilung)
)
;
create table demoverkaeufe(
  artikel number(5)
    constraint verkaufter_artikel references demoartikel,
  abteilung number(5)
    constraint verkauft_in references demoabteilungen,
  anzahl number(5)
    not null,
  constraint ps_demoverkaeufe primary key (artikel,abteilung)
)
;
```

Klassenerzeugung und -initialisierung

Mit dem Klassengenerator für die Datensatzklassen wird aus den Inhalten des *Data Dictionary* wie in den vorigen Kapiteln beschrieben ein Satz von TYCOON-Klassen generiert. Die generierten Klassen, bzw. deren öffentliche Slots und Methoden zeigt Abbildung 7.2 Die mit einem Schlüssel gekennzeichneten Attribute sind Teil des Primärschlüssels der zugrundliegenden Datenbanktabelle und können nur gelesen, nicht überschrieben werden. Um nun auf Datenbankinhalte zugreifen zu können, muß zunächst eine Datenbankverbindung hergestellt

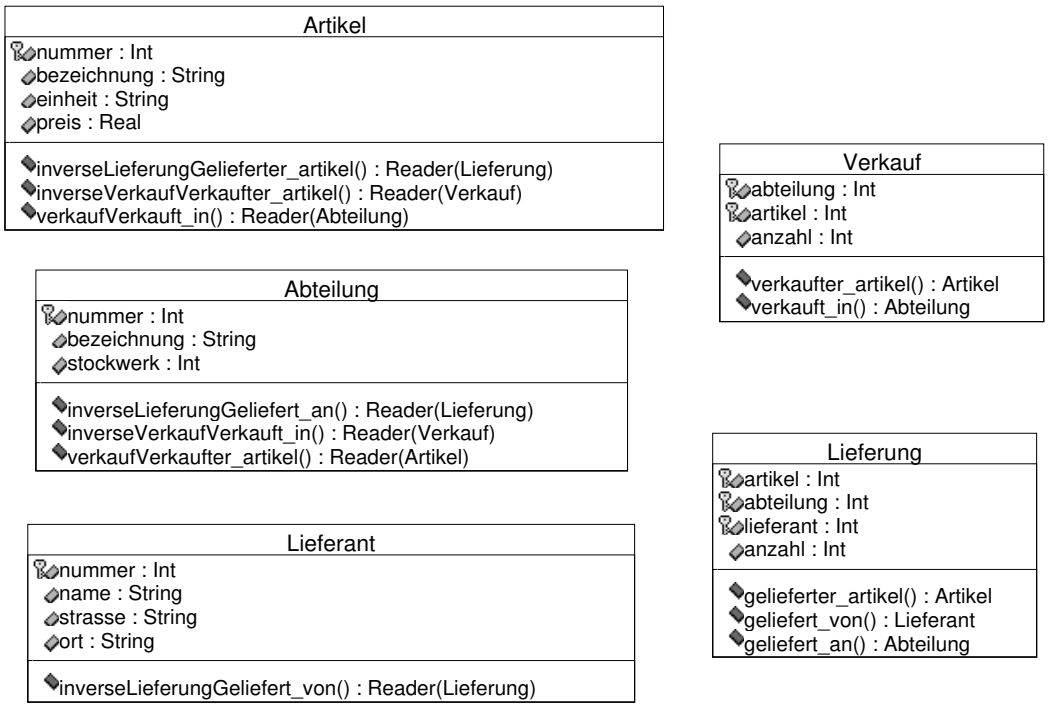

Abbildung 7.2: Automatisch generierte Klassen

werden. Dies geschieht für die Datensatzklassen durch Instantiierung eines Objektes der Klasse `DB`. Anschließend werden die einzelnen Datensatzklassen bei dieser Verbindung registriert. Angenommen die Verbindung bestehe bereits, es existiere also ein Objekt `db` der Klasse `DB`. Dann läuft die Registrierung der Klassen über die folgenden Anweisungen ab.

```
db.registerRecordClass(Artikel,"scott","demoartikel");
db.registerRecordClass(Abteilung,"scott","demoabteilungen");
db.registerRecordClass(Lieferant,"scott","demolieferanten");
db.registerRecordClass(Lieferung,"scott","demolieferungen");
db.registerRecordClass(Verkauf,"scott","demoverkaeufe");
```

Als Parameter werden jeweils die Klasse und die mit ihr zu assoziierende Datenbanktabelle sowie deren Besitzer in der Datenbank angegeben. Nach dieser Registrierung sind die Datensatzklassen nutzbar. Die Assoziation der Datensatzklassen mit der Datenbank bleibt so lange bestehen, bis sie durch eine explizite Deregistrierung wieder aufgehoben wird. Durch das Schließen der Datenbankverbindung geht die Beziehung zwischen Datensatzklasse und Datenbank nicht verloren. Beim erneuten Öffnen der Datenbankverbindung werden die registrierten Datensatzklassen automatisch "reaktiviert". In den folgenden Beispielen wird der objektrelationale Zugriff jeweils einem vergleichbaren relationalen Zugriff gegenübergestellt. Als Beispiel für eine relationale Datenbankschnittstelle wird hier die TYCOON-SQL-Schnittstelle verwendet. Dort wird eine Datenbankverbindung durch die Instantiierung eines Objekts `connection` der Klasse `SQLConnection` hergestellt, das im folgenden auch als bereits gegeben vorausgesetzt wird.

Einfache lesende Zugriffe

Will man in TYCOON über eine SQL-basierte Schnittstelle Datenbankinhalte auslesen, so muß man zunächst einen SQL-Befehl formulieren, der die gewünschten Ergebnisdaten liefert. Über die SQL-Programmierschnittstelle wird dieser Befehl dann als Text an die Datenbank gesendet und dort ausgewertet. Als Ergebnis erhält man einen *Cursor*, mit dem sich das Anfrageergebnis Zeile für Zeile ausgeben läßt. Dieser Ablauf liegt allen gängigen Schnittstellen zugrunde, die für den direkten Datenbankzugriff via SQL verfügbar sind. Bekannte Beispiele sind hier ODBC [Gei96] von MICROSOFT oder das ORACLE CALL INTERFACE [OCI97] für imperative Programmiersprachen. Vergleichbare Schnittstellen für objektorientierte Sprachen sind z.B. die DBTOOLS.H++ für C++ (vgl. Abschnitt 4.3) oder JDBC [HCF97] für Java. Um aus der Beispieldatenbank etwa eine Liste aller vorhandenen Artikel auszugeben, wären die folgenden Aufrufe der TYCOON-SQL-Schnittstelle nötig:

```
let statement:SQLStatement = connection.newStatement(),
let cursor:SQLCursor = statement.executeDirectQuery(
  "select nummer,bezeichnung from demoartikel"
),
while ({cursor.next()}, fun() {
  tycoon.stdout << "Artikelnummer: " << cursor.getInt(0)
                << " Bezeichnung  : " << cursor.getString(1)
                << "\n"
})
```

Zunächst wird mit Hilfe der geöffneten Datenbankverbindung ein Objekt der Klasse `SQLStatement` erzeugt. Diesem wird dann über die Methode `executeDirectQuery` eine SQL-Anfrage geschickt, die die Nummer und den Namen aller in der Tabelle `demoartikel` vorhandenen Artikel liefert. Das Ergebnis ist ein Objekt der Klasse `SQLCursor`. Die folgende `while`-Schleife schreibt so lange Artikelnummer und -name auf den Standardausgabestrom, wie der Cursor neue Datensätze liefert, sprich so lange seine `next`-Methode den Wahrheitswert `true` zurückliefert.
Zu beachten ist hierbei, daß der Programmierer den SQL-Datentyp der Ergebnisspalten kennen muß, damit er eine geeignete `get`-Methode auswählen kann, die ihm das Ergebnis in einem äquivalenten TYCOON-Typ liefert. Außerdem kann eine Ergebnisspalte lediglich über ihre numerische Position im Anfrageergebnis identifiziert werden. D.h. der Anwendungsprogrammierer muß wissen, an welcher Position im Anfrageergebnis sich welche Spalte befindet, um deren Werte korrekt lesen zu können. Ändert sich nun die Anfrage, z.B. durch Auswahl weiterer Tabellenattribute oder Änderung der Attributreihenfolge, so muß auch die Parametrisierung der `get`-Methoden des Cursors geändert werden. Diese Art des Tabellenzugriffs wird daher gerade bei komplexeren Anfragen schnell unübersichtlich und fehleranfällig. Will man nun die gleiche Ausgabe von Artikeldaten über objektrelationale Datensatzklassen erzeugen, so läßt sich dies mit einer einzigen Anweisung lösen.

```
Artikel.reader.do(fun (a:Artikel) {
  tycoon.stdout << "Artikelnummer: " << a.nummer
                << " Bezeichnung  : " << a.bezeichnung
                << "\n"
})
```

Die Methode `Artikel.reader` liefert einen Strom von Objekten der Klasse `Artikel`. Die Methode `do` wendet eine als Parameter zu übergebende Funktion auf alle Objekte dieses `Reader`s an. Im Beispiel ist dies also eine Funktion die als Parameter ein Objekt der Klasse `Artikel` erhält und die Werte seiner Attribute `nummer` und `bezeichnung` auf den Standardausgabestrom schreibt. Diese zweite Version der Anfrage ist wesentlich verständlicher und leichter zu warten, da sie die gesamte Logik der vorigen Anfrage in einem einzigen kurzen Ausdruck bündelt. Sie enthält kein SQL mehr, sondern besteht aus reinem TYCOON-Code, so daß der Programmierer ohne zusätzliche SQL-Kenntnisse den Ausdruck erweitern und an seine Bedürfnisse anpassen kann. Möchte er z.B. zusätzlich den Preis des Artikels mit ausgeben, so braucht er sich keine Gedanken über veränderte Spaltenindizes oder den Datentyp des Preises machen, sondern fügt der Ausgabefunktion einfach die folgende Zeile hinzu.

```
tycoon.stdout << " Preis  : " << a.preis
```

Überlegungen hinsichtlich des korrekten TYCOON-Typs der Attribute sind überflüssig, da die Methoden zum Lesen der Attribute bereits Objeke des korrekten mit der jeweiligen Tabellenspalte korrespondierenden TYCOON-Typs zurückliefern.

Beziehungen zwischen Daten

Eine weitere häufige Form der Anfrage, die beim Umgang mit relationalen Datenbanken anzureffen ist, sind die sogenannten *Joins*. Durch sie werden Daten verschiedener Tabellen, zwischen denen eine semantische Beziehung besteht, über Fremdschlüsselattribute miteinander verbunden. Möchte man beispielsweise eine Liste aller Artikel ausgeben, die an die Abteilung mit der Nummer 1 geliefert wurden, so sind unter Zuhilfenahme der TYCOON-SQL-Schnittstelle folgende Anweisungen nötig:

```
let statement:SQLStatement = connection.newStatement(),
let cursor:SQLCursor = statement.executeDirectQuery(
  "select art.bezeichnung " +
  "from demoartikel art, demolieferungen l " +
  "where art.nummer = l.artikel and l.abteilung = 1"
),
while ({cursor.next()}, fun() {
  tycoon.stdout << "Artikel: " << cursor.getString(0)
                << "\n"
})
```

Der TYCOON-Code zur Erzeugung des Cursors und zur Ausgabe der Ergebnisse ist mit dem des vorangegangenen Beispiels nahezu identisch. Lediglich die an die Datenbank gesendete SQL-Anfrage hat sich verändert. Diese wählt jetzt all die Artikel aus, deren Artikelnummer als Fremdschlüssel in der Tabelle `demolieferungen` auftaucht und bei denen die Lieferung an die Abteilung mit der Nummer 1 erfolgte. Zur Programmierung dieser Anfrage muß der Entwickler also die Struktur der Datenbanktabellen kennen und insbesondere Wissen über Fremdschlüssel in den Tabellen besitzen. Hier ist das Attribut `artikel` der Tabelle `demolieferungen` der relevante Fremdschlüssel, der die Beziehung zwischen der Abteilung und dem Artikel herstellt.
Ein analoges Ergebnis über die objektrelationale Schnittstelle erhält man durch die Verwendung der Navigationsmethoden der betroffenen Klassen. So wurde für die Klasse `Abteilungen`

eine Methode generiert, die alle Lieferungen auswählt, an denen die Abteilung beteiligt ist (`inverseLieferungGeliefert_an`). Jedes Objekt der Klasse `Lieferungen` verfügt über eine Methode, die zu dem Datenbankfremdschlüssel `gelieferter_artikel` korrespondiert und mit der man ein Objekt der Klasse `Artikel` erhält. Der TYCOON-Code zur Anzeige der betreffenden Artikelnamen reduziert sich damit auf den Ausdruck

```
Abteilung[1].inverseLieferungGeliefert_an.do( fun(l:Lieferung) {
  tycoon.stdout << "Artikel: " << l.gelieferter_artikel.bezeichnung
})
```

Die erste Zeile erzeugt einen `Reader` aller Lieferungen, an denen die Abteilung 1 beteiligt ist. Für jede dieser Lieferungen wird die Methode `gelieferter_artikel` aufgerufen. Das Attribut `bezeichnung` des so referenzierten Artikelobjekts wird dann ausgegeben. Diese Anfrage erfordert vom Programmierer keinerlei Kenntnisse über die Struktur und die Abhängigkeiten der Datenbanktabellen. Wenn, wie beim hier verwendeten Datenbankschema, die Fremdschlüssel in der Datenbank sorgfältig benannt wurden, so geht die Bedeutung der automatisch generierten Navigationsmethoden unmittelbar aus ihrem Namen hervor und bedarf keiner manuellen Nachbearbeitung mehr.

Manipulation und Neuerzeugung von Daten

Sollen Werte innerhalb von Datensätzen verändert werden, so bietet die Sprache SQL eine Menge von DML(*Data Manipulation Language*)-Befehlen, an. Soll beispielsweise an den Artikelpreisen in einer Datenbank eine Erhöhung der Mehrwertsteuer von 15 auf 16 Prozent nachvollzogen werden, so leistet das der SQL-Befehl (wiederum eingebettet in TYCOON-Code)

```
let statement:SQLStatement = connection.newStatement(),
statement.executeDirectUpdate(
  "update demoartikel set preis = preis / 1.15 * 1.16"
)
```

Da dieser Befehl direkt in der Datenbank ausgeführt wird, muß diese Änderung an allen Objekten des Anwendungsprogramms, die sich auf diese Daten beziehen, manuell nachvollzogen werden. Bezögen sich die Anwendungsobjekte aber auf objektrelationale Datensatzobjekte, so erzielte die folgende TYCOON-Anweisung denselben Effekt.

```
Artikel.reader.do( fun(artikel:Artikel) {
  artikel.preis := artikel.preis / 1.15 * 1.16
})
```

Hier ist der Gewinn gegenüber der SQL-Lösung nicht so offensichtlich. Neben der Ersparnis gesonderter SQL-Kenntnisse ergeben sich aber zwei weitere wichtige Vorteile. Da der Objektcache dafür sorgt, daß jedes eindeutig identifizierbare Datensatzobjekt nur einmal im Objektspeicher vorhanden ist, verwenden jetzt implizit alle Anwendungsobjekte, die Datensatzobjekte referenzieren, die neu gesetzten Werte. Eine manuelle Nachbearbeitung der Anwendungsobjekte ist nicht erforderlich, solange diese nicht mit eigenen Kopien der Daten arbeiten. Der zweite Vorteil liegt ebenfalls im Cache begründet. Die Änderungen werden erst dann in die Datenbank geschrieben, wenn der Benutzer seine Transaktion oder seine Anwendung abschliesst. Alle weiteren Änderungen, die nach obiger Preisanpassung vorgenommen werden, werden akkumuliert und gesammelt am Transaktionsende in die Datenbank

zurückgeschrieben. Dadurch sinkt das Kommunikationsaufkommen mit der Datenbank unter Umständen erheblich.
Ein anderer DML-Befehl ist `insert`. Mit ihm werden neue Datensätze in die Datenbank eingefügt. Als Beispiel soll hier ein neues Produkt mit einem neuen Lieferanten in die Datenbank eingefügt werden. In reinem SQL leistet dies die Anweisunssequenz

```
insert into demoartikel (nummer,bezeichnung,einheit,preis)
                values (53,'Tycoon','stk',99.95);

insert into demolieferanten (nummer,name,ort,strasse)
                    values (21,'STS','Hamburg-Harburg','Harb. Schlossstr.');

insert into demolieferungen (lieferant,artikel,abteilung,anzahl)
                    values (21,53,9,10);
commit;
```

Hier ist insbesondere dafür zu sorgen, daß die Spaltenwerte den korrekten Spalten zugeordnet werden und daß keine Integritätsbedingungen verletzt werden. Eine Lieferung kann erst eingefügt werden, wenn das Produkt, die belieferte Abteilung und der Lieferant vorhanden sind. Bei den objektrelationalen TYCOON-Befehlen ist hingegen die Zuordnung von Werten zu Attributen deutlicher erkennbar..

```
let artikel:Artikel = Artikel.new(53),
artikel.bezeichnung := "Tycoon",
artikel.einheit     := "stk",
artikel.preis       := 99.95,

let lieferant:Lieferant = Lieferant.new(21),
lieferant.name    := "STS",
lieferant.ort     := "Hamburg",
lieferant.strasse := "Harb. Schlossstr.",

let abteilung:Abteilung =
  Abteilung.readerWhere("bezeichnung = \'Computer\'").read(),

let lieferung = Lieferung.new(
                              abteilung.nummer,
                              lieferant.nummer,
                              artikel.nummer
                             ),
lieferung.anzahl := 10,

db.commitWork;
```

Ein weiterer Vorteil ist hier, daß wiederum nahezu keine Kenntnisse über Fremdschlüsselbeziehungen erforderlich sind. Die Reihenfolge, in der die einzelnen Objekte erzeugt werden, ist beliebig. Der Cache und die von der Klasse `Record` geerbten Methoden `flushParentRecords` und `flushChildRecords` stellen sicher, daß voneinander abhängige Datensätze in der korrekten Reihenfolge in der Datenbank gespeichert werden.

Kapitel 8

Ausblick

Zum Abschluß dieser Arbeit werden Möglichkeiten erörtert, wie die Leistungsfähigkeit und der Bedienkomfort des derzeitigen Systems weiter verbessert werden können (Abschnitt 8.1). Im Anschluß daran wird auf aktuellen Entwicklungen im Bereich objektrelationaler Systeme hingewiesen. Dabei finden insbesondere Entwicklungen Berücksichtigung, die mit der zunehmenden Verbreitung der objektorientierten Programmiersprache JAVA einhergehen (Abschnitt 8.2).

8.1 Erweiterungsmöglichkeiten

Die hier vorgestellte Lösung zur Abbildung von Daten einer relationalen Datenbank auf Objekte eines objektorientierten Systems kann bereits produktiv für die Entwicklung von Datenbankanwendungen eingesetzt werden. Dennoch sind in Teilbereichen noch Verbesserungen möglich, die den praktischen Nutzen des entwickelten Sytems weiter erhöhen können. Diese Verbesserungsmöglichkeiten zielen insbesondere auf

- eine komfortablere Klassengenerierung (Abschnitt 8.1.1),
- Weiterentwicklung bereits laufender Systeme (Abschnitt 8.1.2),
- die Effizienz der Cacheverwaltung (Abschnitt 8.1.3),
- Zeitpunkt und Anzahl von Sperranforderungen in der Datenbank (Abschnitt 8.1.4),
- die Verwaltung von Objektbeziehungen (Abschnitt 8.1.5) und
- die Mehrbenutzerfähigkeit (Abschnitt 8.1.6).

8.1.1 Klassengenerierung

Abgesehen von der Spezifikation der zu generierenden Klassen und der zugrundeliegenden Datenbanktabellen verläuft die Erzeugung neuer Datensatzklassen bisher vollautomatisch. Insbesondere bei der Benennung von Navigationsmethoden entstehen dabei unter Umständen Methodenbezeichner, denen die wahre Bedeutung im Sinne eines Objektmodells nicht mehr unbedingt anzusehen ist. Hier wären weitere Konfigurationsmöglichkeiten des Klassengenerators denkbar, über die z.B. aussagekräftige Bezeichner vorgegeben werden können, wie sie etwa bei der Bezeichnung von Rollen innerhalb eines Objektmodells zum Einsatz kommen.

Wird ein Datenbankschema benutzt, in dem keine Fremdschlüsselbeziehungen definiert wurden, obwohl diese im konzeptuellen Modell vorhanden waren, so könnte eine Möglichkeit vorgesehen werden, um nachträglich diese Beziehungen zu spezifizieren. In einer letzten Ausbaustufe könnte man schließlich sämtliche Konfigurationsmöglichkeiten über eine grafische Oberfläche nutzbar machen.

8.1.2 Evolutionäre Systementwicklung

Eine Frage, die nicht nur die Erzeugung datenbankspezifischer Klassen betrifft, ist die Erweiterbarkeit vorhandener Klassen in einem persistenten, objektorientierten System. Wenn sich z.B. in einer relationalen Datenbank das Datenbankschema verändert, lassen sich mit dem vorhandenen Klassengenerator neue, an dieses Datenbankschema angepaßte Klassen erzeugen. Bereits bestehende Anwendungen basieren aber auf Klassen, die für ältere Versionen des Datenbankschemas entwickelt wurden. Dadurch ergeben sich mehrere Probleme:

- Die bestehende Anwendung geht von einem veralteten Datenbankschema aus. Dieses Problem stellt sich auch bei anderen Datenbankanwendungen nach einer Schemaveränderung. Neue Attribute von Tabellen werden nicht berücksichtigt. Neue Integritätsbedingungen oder die Löschung von Attributen oder ganzen Tabellen können erhebliche Anpassungen an alten Anwendungsprogrammen erfordern. Diese erforderlichen Anpassungen werden durch einen Klassengenerator zwar beschleunigt, lassen sich aber nicht vollständig automatisieren. Denn die Verfügbarkeit neuer Attribute oder Beziehungen versetzt das Anwendungsprogramm noch nicht in die Lage, diese auch zu nutzen.

- Automatisch erzeugte Klassen wurden unter Umständen um anwendungsspezifische Eigenschaften erweitert. Diese anwendungsspezifischen Eigenschaften gehen bei einer automatischen Neuerzeugung der Klassen verloren. Diesem Problem läßt sich in gewissem Rahmen durch einen Programmierstil begegnen, bei dem nicht die erzeugten Klassen direkt manipuliert werden, sondern die anwendungsspezifische Logik entweder in Subklassen implementiert wird oder Anwendungsobjekte die datenbankspezifische Logik an Objekte der automatisch generierten Klasse delegieren. Durch eine solche Trennung von Anwendungslogik und automatisch generierter datenbankspezifischer Logik können vorhandene Teile einer Datenbankanwendung nicht versehentlich überschrieben werden. Erforderliche Anpassungen an die veränderten Datensatzklassen sind so leichter zu erkennen und auszuführen.

- Die automatisch erzeugten Klassen überschreiben bei Namensgleichheit die Definitionen ihrer Vorgängerklassen. Falls sich noch Objekte der ursprünglichen Klassendefinition im Objektspeicher befinden, stellt sich die Frage wie sich diese Objekte zu der neuen Klasse verhalten. So vielfältig wie die Mglichkeiten zur Behandlung dieses Problems sind auch die neuen Fragestellungen, die sich damit auftun, z.B.

  - Lassen sich Namenskonflikte zwischen Klassen auflösen, so daß jedes Objekt die Implementierung der Klasse nutzen kann, die zum Zeitpunkt seiner Erzeugung gültig war?
  - Sollen/dürfen überhaupt unterschiedliche Versionen derselben Klasse parallel existieren?

- Welche Version einer Klasse wird benutzt, wenn ein neues Objekt erzeugt werden soll?
- Kann ein Objekt einer älteren Klassenversion auch dort benutzt werden, wo ein Objekt einer neueren Klassendefinition erwartet wird ? Dazu müßte eine Art "struktureller Ähnlichkeitstest" implementiert werden.

Auf diesem Gebiet sind noch erhebliche Entwicklungsarbeiten zu leisten.

8.1.3 Optimierung der Cacheverwaltung

Die Cache-Verwaltung, über die jede Datensatzklasse verfügt, erreicht ihren maximalen Nutzen hauptsächlich bei Zugriffen über den Primärschlüssel. D.h. immer dann, wenn direkt der Primärschlüssel angegeben wird, um auf ein existierendes Datensatzobjekt zuzugreifen, kann dieses Objekt aus dem Cache geliefert werden — falls es dort bereits existiert. Auch die Navigation entlang von Objektbeziehungen wird insbesondere dann beschleunigt, wenn für das referenzierte Objekt der Primärschlüssel bekannt ist. Dies ist in aller Regel bei den Fremdschlüsselattributen einer Datenbanktabelle der Fall.
Wird hingegen eine Fremdschlüsselbeziehung in der entgegengesetzten Richtung verwendet, so daß zu einem Ausgangsobjekt `o` alle die Objekte geliefert werden sollen, deren Fremdschlüssel auf das Objekt `o` verweisen, so muß zunächst eine SQL-Anfrage von der Datenbank ausgewertet werden. Dazu wird zunächst der Zustand der im Cache der betreffenden Ergebnisklasse befindlichen Datensatzobjekte in die Datenbank zurückgeschrieben. Für die danach von der Datenbank auf die Anfrage gelieferten Datensätze werden zwar nur dann neue Datensatzobjekte erzeugt, wenn diese nicht bereits im Cache existieren, aber allein durch die Auswertung der Anfrage auf dem Datenbankserver verlängert sich die Antwortzeit solcher Navigationsmethoden.
Diese Einschränkung gilt für alle Methoden, die einen `Reader` von Objekten liefern. Immer wenn mehr als ein Objekt zurückgegeben werden kann, muß zunächst einen Anfrage an die Datenbank gesendet werden, welche die für diesen `Reader` augenblicklich gültige Menge von Datensätzen bestimmt. Die Anfrage wird — überflüssigerweise — auch dann abgeschickt, wenn sich bereits alle Objekte, die die Anfrage erfüllen, im Cache befinden.
Dieses Problem läßt sich in gewissem Umfang durch die Implementierung eines Anfrageprozessors innerhalb des objektorientierten Systems lösen. Dieser könnte die bewußte Anfrage zunächst auf Grundlage der bereits im Cache befindlichen Daten auswerten. Die Anfrage könnte z.B. mit Mitteln der Programmiersprache erfolgen, wie dies vielfach bei objektorientierten Datenbanken üblich ist (vgl. [Loo95]) oder direkt in SQL formuliert werden. Der Anfrageprozessor kann dann die Anfrage auswerten und all diejenigen Objekte aus dem Cache liefern, die die Anfrage erfüllen.
Allerdings läßt sich mit dieser Lösung ein Datenbankzugriff nicht völlig vermeiden. Befinden sich noch nicht alle Datensätze der Datenbank, die eine bestimmte Anfrage erfüllen, im Cache, so liefert die Auswertung der Anfrage allein auf dem Cache eine zu kleine Ergebnismenge. Abhilfe könnte hier eine zweistufige Strategie schaffen. Der Anfrageprozessor liefert zunächst die sich bereits im Cache befindlichen Objekte, von denen die Anfrage erfüllt wird. Anschließend wird an die Datenbank eine stärker eingeschränkte SQL-Anfrage geschickt, die nur noch die Objekte liefert, die nicht bereits im Cache vorhanden waren. Dadurch werden zwei wesentliche Effekte erreicht:

- Die von der Datenbank zu übertragende Ergebnismenge verkleinert sich. Dadurch verringert sich die Übertragungszeit zwischen Datenbank und Objektsystem im Vergleich zur ursprünglichen Anfrage.
- Wenn das Anfrageergebnis als `Reader` zurückgeliefert wird, kann die Auswertung der Datenbankanfrage so lange verzögert werden, bis der letzte Datensatz aus dem Cache gelesen worden ist. Wird der `Reader` gar nicht bis zu seinem Ende gelesen, muß auch die Datenbankanfrage nicht oder nicht vollständig abgearbeitet werden.

Letzterer Punkt ist z.B. bei solchen Anfragen sinnvoll, die eine große Menge von Ergebnissen liefern, von denen aber in der Regel nur einige wenige tatsächlich weiterverwendet werden. Dieser Effekt tritt typischerweise bei der Suche in Volltextdatenbanken auf, die mittlerweile auch auf der Basis relationaler Datenbanken implementiert werden. (vgl. Suchmaschinen im WWW, ConText Cartridge bei Oracle [Con] oder die Text Datablades für Informix [Tex]). Die Suche in einer solchen Datenbank liefert gerade bei unspezifischen Suchbegriffen eine große Menge von Ergebnissen, die nur in seltenen Fällen vom Benutzer vollständig zur Kenntnis genommen oder weiterverarbeitet werden. Wenn also bereits die aus dem Cache erhaltene Antwort auf die Anfrage ausreicht, ist keine weitere Kommunikation mit der Datenbank erforderlich.
Die Geschwindigkeitsvorteile durch eine verzögerte oder vermiedene Datenbankkommunikation werden hier allerdings mit einem erhöhten Aufwand zur Anfrageauswertung im objektorientierten System erkauft. Dabei müssen im objektorientierten System Funktionen implementiert werden, die im Datenbanksystem in ähnlicher oder identischer Form bereits vorhanden sind — ein Aufwand, der nicht für jede objektrelationale Anwendung lohnenswert erscheint.

8.1.4 Minimierung der Zahl und Dauer von Datenbanksperren

Solange Datensätze in der Datenbank gesperrt sind, ist ein schreibender Zugriff für andere Datenbankbenutzer auf dieselben Datensätze nicht möglich. Da dies erheblichen Einfluß auf die Antwortzeiten des Datenbanksystems und damit die Laufzeit der Datenbankanwendungen haben kann, sollten exklusive Sperren immer nur kurz angefordert werden und auch nur dann, wenn die betreffenden Datensätze tatsächlich geändert werden sollen.
Bei der momentanen Implementierung wird eine konservative Strategie verfolgt. Sobald auf einen Datensatz lesend oder schreibend zugegrffen wird, wird er auch in der Datenbank gesperrt. Da der Anwendungsentwickler sich der Existenz der Datenbank nicht bewußt zu sein braucht, wird über sein Verhalten die folgende Annahme getroffen: Der Entwickler erwartet, daß sich zwischen zwei lesenden Zugriffen der Inhalt eines Datensatzes nicht ändert, es sei denn, seine Anwendung hat diese Veränderungen selber vorgenommen. Diese Annahme führt dazu, daß Datensätze so lange gesperrt bleiben, bis die Tycoon-Anwendung beendet oder das Ende einer Transaktion signalisiert wird. Um zu verhindern, daß durch langlaufende Anwendungen oder Transaktionen Datensätze langfristig gesperrt werden, wäre die Implementierung eines optimistischen Sperrkonzepts denkbar.
Dabei wird weder für lesende noch für schreibende Zugriffe Sperren angefordert. Erst dann, wenn ein Datensatz in der Datenbank gespeichert werden soll, wird er für die Dauer des Speichervorganges gesperrt. Dabei muß überprüft werden, ob sich der Datensatz in der Datenbank noch im selben Zustand befindet, in dem er sich bei Beginn der Transaktion befunden hat. Hat sich der Zustand des Datensatzes in der Datenbank mittlerweile geändert, so kann

die Transaktion nicht beendet werden. Änderungen, die innerhalb dieser Transaktion vorgenommen worden sind, müssen dann verworfen werden. Danach kann dann der aktuelle Zustand der Datenbank in die korrespondierenden Datensatzobjekte übernommen und eine neue Transaktion gestartet werden. Für die Erkennung einer Änderung in der Datenbank muß die objektrelationale *Middleware* eine Kopie des Datenbankzustandes zum Zeitpunt des Transaktionsbeginns bereitstellen. Mit dieser Kopie werden die Datenbankinhalte dann verglichen, bevor eine Änderungsoperation ausgeführt wird.
Eine Geschwindigkeitssteigerung des Datenbankzugriffs ist hier nur dann zu erwarten, wenn die betroffenen Datenbanktabellen nur von wenigen Datenbanknutzern parallel manipuliert werden und Schreibzugriffe verhältnismäßig selten sind. Werden dagegen Inhalte einer Tabelle sehr häufig und von vielen Benutzern verändert, so erhöht sich die Zahl der optimistischen Transaktionen, die wegen zwischenzeitlicher Zugriffe anderer Transaktionen scheitern.

8.1.5 Handhabung von Objektbeziehungen

Beziehungen zwischen Datensätzen können über Navigationsmethoden der jeweiligen Objekte nachvollzogen werden. Die Herstellung neuer Objektbeziehungen geschieht durch korrespondierende Zuweisungsmethoden. Ein Objekt, das über einen Fremdschlüssel der zugrundeliegenden Tabelle ein anderes Objekt referenziert, verfügt über eine Zuweisungsmethode, die mit einem neuen Objekt aufgerufen werden kann. Nach Aufruf dieser Methode enthält der Fremdschlüssel den Primärschlüsselwert des neuen Objekts, so daß die zugehörige Navigationsmethode das neue Objekt liefert. Dieser Mechanismus funktioniert bei allen Objekten, bei denen die Navigationsmethode nur höchstens ein referenziertes Objekt liefert.
Liefert die Navigationsmethode mehr als ein Objekt bzw. einen `Reader` von Objekten, so ist die Herstellung einer neuen Beziehung zwischen zwei Objekten nicht direkt möglich. Im Falle einer 1:N-Beziehung verfügen die Objekte der "N-Klasse" über eine Zuweisungsmethode, mit der sich eine Beziehung zu einem Objekt der "1-Klasse" herstellen läßt. Bei N:M-Beziehungen oder k-nären Beziehungen ($k > 2$) muß zur Herstellung einer Beziehung ein neues Objekt einer Beziehungsklasse erzeugt werden, dessen Primärschlüssel sich aus den Primärschlüsselwerten der beteiligten Objekte zusammensetzt. Diese Werte müssen momentan bei der Erzeugung des Beziehungsobjekts explizit angegeben werden.
Einen Verbesserung ist dahingehend möglich, daß bei der Erzeugung eines Beziehungsobjektes nicht die Schlüssel der beteiligten Objekte, sondern die Objekte selbst angegeben werden können. Die Beziehungsklasse muß dann aus den jeweiligen Objekten die Primärschlüsselwerte in der richtigen Reihenfolge extrahieren, damit später ein korrekter Eintrag für die entsprechende Beziehungstabelle erzeugt werden kann.

8.1.6 Mehrbenutzerfähigkeit

Das TYCOON-System besitzt bisher keine eingebaute Benutzerverwaltung, über die ein Zugriff mehrerer TYCOON-Anwender auf denselben Objektspeicher geregelt wird. Mechanismen zum parallelen Ausführen von Programmteilen und zur Synchronisation von Objektzugriffen durch parallele Kontrollfäden (*Threads*) sind aber bereits vorhanden. Zieht man dann noch die Fähigkeiten des TYCOON-Systems zur Kommunikation über Netzwerke hinzu (*Sockets*, TYCOON WEB SERVER), so stehen einer Verwendung des Systems als *Object Manager*, wie er z.B. bei PERSISTENCE (vgl. Abschnitt 4.4) verwendet wird, aus technischer Sicht keine Hindernisse im Wege.

TYCOON könnte mehreren Anwendern einen gemeinsamen Cache von Datenbankobjekten zur Verfügung stellen, durch den nur noch dann Daten aus der Datenbank nachgeladen werden müssen, wenn keiner der Anwender bisher den gewünschten Datensatz verwendet hat. Allerdings besteht auch hier die Möglichkeit, daß jeder Anwender zeitweilig mit eigenen Kopien der im globalen TYCOON-Cache befindlichen Daten arbeitet. Daher wird ein Synchronisationsmechanismus nötig, der Konflikte auflöst, falls mehrere Benutzer in ihrem lokalen Cache Kopien derselben Datensätze aus dem globalen Cache verändern. Möglichkeiten für eine solche Cache-Synchronisation sind z.B.

- Vergabe von Versionsnummern für Objekte, die dann bei Änderungsoperationen inkrementiert werden. Wird beim Zurückschreiben in den globalen Cache eine gleiche oder höhere Versionsnummer gefunden, hat bereits ein anderer Anwender das Objekt verändert.
- Sperren von Datensätzen im globalen Cache gegen weitere Zugriffe, sobald sie von einem Anwender gelesen oder verändert werden.
- Versendung von Nachrichten an alle lokalen Caches, sobald sich ein Objekt im globalen Cache geändert hat. Befindet sich das Objekt noch in einem der benachrichtigten Caches, wird es dort als ungültig markiert bzw. die dort schon vorgenommenen Änderungen werden verworfen.

8.2 Aktuelle Entwicklungen

Bereits in der Einleitung (vgl. Kapitel 1) wird darauf hingewiesen, daß sowohl relationale als auch objektorientierte Systeme in der Anwendungsentwicklung eine weite Verbreitung gefunden haben. Wurden diese unterschiedlichen Systeme lange Zeit unabhängig voneinander entwickelt, so zeichnet sich mittlerweile ein Zusammenwachsen von relationaler und objektorientierter Technik ab. Die Annäherung von relationaler und objektorientierter Technik findet dabei aus zwei Richtungen statt.

- Objektrelationale Datenbanken (Abschnitt 8.2.1) erweitern den Funktionsumfang relationaler Datenbanken um Konzepte aus der objektorientierten Theorie.
- Objektorientierte Programmiersprachen und Datenbanken werden um Pakete erweitert, die den Zugriff auf relationale Datenbanken ermöglichen und die Nutzung der Datenbank als einen persistenten Objektspeicher erlauben.
  Die Verbreitung der objektorientierten Programmiersprache Java hat insbesondere dem Gebiet der objektrelationalen *Middleware* einen neuen Wachstumsschub beschert. Abschnitt 8.2.2 ist daher den Entwicklungen auf dem Gebiet des Java-gestützten Datenbankzugriffs gewidmet.

8.2.1 Objektrelationale Datenbanken

Während relationale Datenbanken eine effiziente Verwaltung von großen, einfach strukturierten Datenmengen ermöglichen, stellen objektorientierte Systeme dem Entwickler mächtige Abstraktionsmechanismen zur Repräsentation der Anwendungswelt zur Verfügung. Objektorientierte Datenbanksysteme gestatten dem Anwendungsentwickler, Objekte der objektorientierten Programmiersprache dauerhaft zu speichern. Ein Paradigmenbruch wie beim Zugriff

auf relationale Datenbanken tritt dabei nicht mehr auf.
Hersteller relationaler Datenbanken erkennen zunehmend die Vorteile objektorientierter Modellierung und versuchen daher, ihre vorhandenen Systeme um die Möglichkeiten des neuen Paradigmas zu erweitern. Anwender können so weiterhin die Vorteile der relationalen Technik nutzen und erhalten die Möglichkeit, bestehende oder neue Anwendungen um objektorientierte Funktionalität zu erweitern (z.B. [Ora]).
Diese neue Klasse von Datenbanken, die sogenannten objektrelationalen Datenbanken, erlaubt die Definition komplexer Datentypen, die nicht mehr der durch Tabellen vorgegebenen flachen Struktur von Datensätzen folgen müssen. Solche benutzerdefinierten Typen dürfen aus nahezu beliebigen vorhandenen Typen zusammengesetzt werden. Datensätze dieser selbstdefinierten Typen werden weiterhin innerhalb einer Tabelle gespeichert. Die Tabelle dient aber nur noch als eine Art Behälter für die neuen Objekte.
Um Objekte eindeutig identifizieren zu können, vergeben objektrelationale Datenbanken automatisch Objektidentifikatoren, wie dies auch in objektorientierten Datenbanken und Programmiersprachen üblich ist. Neben der Möglichkeit, Objektbeziehungen über ausgezeichnete Fremdschlüsselattribute herzustellen, können auch direkte Referenzen auf andere Objekte hergestellt werden. Anstelle eines Joins über mehrere Tabellen kann eine Anfrage dann durch direkte Navigation entlang solcher Objektreferenzen erfolgen.
Zusätzlich zu den einfachen SQL-Datentypen für Texte, Zahlenwerte oder Datumsangaben, enthalten objektrelationale Datenbanksysteme zusätzlich Datentypen für die Verwaltung vergleichsweise unstrukturierter Daten wie Audio-, Bild- oder Videodaten (vgl. [Die98]). Auch Massendatentypen wie Tabellen, Mengen oder Arrays können Bestandteil komplexer Datentypen sein.
Während die Modellierung komplexer Datenstrukturen bereits in kommerzielle Datenbanksysteme integriert ist (Beispiele: [Ora, Inf]), zeichnet sich für die Implementierung des Verhaltens von Objekten noch keine vergleichbar eindeutige Entwicklung ab. Eine Möglichkeit wäre, die proprietäre Anwendungsprogrammiersprache, die viele Datenbankhersteller in ihre Systeme integrieren (z.B. PL/SQL bei ORACLE), um ein Methodenkonzept zu erweitern. Dies hat aber gleich mehrere Nachteile: Zum einen unterscheiden sich die Anwendungsprogrammiersprachen von einem Datenbankhersteller zum anderen, so daß eine Wiederverwendbarkeit des Codes über Herstellergrenzen hinweg nicht möglich ist. Zum anderen ist eine Integration objektorientierter Techniken in eine Sprache der dritten Generation, an der sich die meisten relationalen Anwendungssprachen orientieren, nicht ohne Kompromisse möglich. So müssen alte Anwendungen weiterhin nutzbar bleiben. Neue objektorientierte Anwendungen erfordern aber eine völlig andere Infrastruktur zu ihrer Ausführung (z.B. Versand von Nachrichten, Garbage Collection), die durch die ursprüngliche Datenbankprogrammiersprache nicht gegeben ist. Durch die Erweiterung einer vorhandenen Datenbankprogrammiersprache, entsteht eine neue objektorientierte Sprache, die sich dann sowohl von der bekannten Datenbanksprache, als auch von etablierten objektorientierten Sprachen unterscheidet und dem Anwendungsentwickler dadurch zusätzlichen Einarbeitungs- und Entwicklungsaufwand beschert.
Eine andere Möglichkeit zur Integration von Verhalten in Objekte der Datenbank ist die Nutzung einer etablierten objektorientierten Programmiersprache, deren Programmcode direkt innerhalb des Datenbanksystems ausgeführt werden kann. Die Chance, datenbankunabhängig das Verhalten persistenter Objekte zu implementieren, ist hier zweifellos größer. Hier stellt sich das Problem, daß die Typsysteme von Programmiersprache und Datenbank nicht miteinander kompatibel sind und erst aneinander angeglichen werden müssen. Das könnte durch eine Kapselung der Datenbanktypen innerhalb spezieller Klassen der Programmiersprache

geschehen, deren Schnittstelle datenbankunabhängig spezifiziert ist.
Allerdings ist sowohl bei diesem als auch beim vorigen Ansatz die Integration der objektorientierten Funktionen in das bestehende Zugriffs- und Transaktionskonzept der relationalen Datenbank noch nicht geklärt. Da Methodenaufrufe eines Objekts Änderungen und Seiteneffekte in zahlreichen anderen Objekten nach sich ziehen können, muß z.B. geklärt werden, ob die betroffenen Objekte aufgrund der derzeitigen Sperren und Zugriffsrechte überhaupt manipuliert werden dürfen. Weiterhin stellt sich die Frage, wann solche Veränderungen an transitiv erreichbaren Objekten für andere Benutzer sichtbar werden. Bis zum Ende einer Transaktion muß das Datenbanksystem daher Kopien der Ausgangsobjekte vorhalten, die erst bei erfolgreicher Beendigung einer Transaktion vernichtet werden dürfen. Dabei stellt sich die Frage, wie mit Objektreferenzen zu verfahren ist, die noch auf alte Objektversionen verweisen.
Die Integration von objektorientierten Konzepten in eine relationale Datenbank erfordert also zum Teil erhebliche Änderungen an den Transaktions-, Sperr- und Speicherverwaltung des Datenbanksystems, die immer unter weitgehender Erhaltung der ursprünglichen relationalen Funktionalität stattfinden müssen.
Trotz der oben angedeuteten Schwierigkeiten wird bereits allein durch die Definition komplexer Datentypen eine wesentliche Annäherung des Datenmodells der Datenbank an das Objektmodell einer objektorientierten Programmiersprache erreicht. Der *Impedance Mismatch* zwischen der Programmiersprache des Anwendungsprogramms und der objektrelationalen Datenbank fällt damit bei der Anwendungsentwicklung weitaus weniger ins Gewicht als bei der Verwendung einer rein relationalen Datenbank. Durch die Integration moderner objektorientierter Konzepte in bekannte und etablierte relationale Systeme versprechen sich Datenbankhersteller daher auch einen erheblichen Wachtumsschub für die nähere Zukunft (vgl. [SM96]).

8.2.2 Datenbankzugriff mit Java

Die Programmiersprache Java hat sich innerhalb weniger Jahre durch ihre Unabhängigkeit von spezifischen Rechnerplattformen und Betriebssystemen, ihr objektorientiertes Programmiermodell und nicht zuletzt durch das explosionsartige Wachstum des Internets als Implementationssprache für portable, objektorientierte Anwendungsprogramme etabliert. Mit Hilfe objektorientierter Kommunikationsarchitekturen wie z.B. CORBA (*Common Object Request Broker Architecture*, vgl. [OPR96, OMG]) lassen sich in Java verteilte Anwendungen realisieren, die auch weit entfernten Standorten von Forschungseinrichtungen, Industrieunternehmen oder Privatanwendern eine Zusammenarbeit und eine gemeinsame Nutzung von Ressourcen erlauben. Gerade für unternehmenskritische Anwendungen ist die Integration bestehender Systeme in neue kooperative, verteilte Anwendungssysteme von zentraler Bedeutung. Datenbanken sind typische Vertreter solcher Systeme, die wegen ihrer starken Verflechtung mit bereits vorhandener Anwendungssoftware in der Regel nicht durch neue Systeme ersetzt werden können, sondern in neue Systeme *integriert* werden müssen. Durch die Popularität der Programmiersprache Java sind bereits eine Vielzahl von Produkten verfügbar, die diese Integration leisten sollen. Einige der wichtigsten sollen im folgenden vorgestellt werden.

JDBC und JSQL

Die Klassenbibliothek JDBC (JAVA DATABASE CONNECTIVITY [HCF97]) ist der Versuch der Firma SUN MICROSYSTEMS, für Java einen Standard für den Zugriff auf Datenbanken via SQL zu etablieren. JDBC wird mittlwereile von einer großen Zahl von Herstellern relationaler *und* objektorientierter Datenbanken[1] unterstützt. Die erste Version der JDBC-Klassen ist noch stark von Begrenzungen bestimmt, wie sie auch von den ersten *Call Level Interfaces* bekannt sind, wie z.B. die eingeschränkte Manipulation von Ergebnismengen oder die Unterstützung nur weniger SQL-Standardtypen.
Mittlerweile liegt JDBC in einer zweiten Version vor, deren Funktionsumfang in einigen Bereichen erheblich erweitert worden ist. Zu den wichtigsten Neuerungen gehören:

- wahlfreier Zugriff auf Ergebnismengen (nächste Zeile, vorige Zeile, Ergebniszeile mit gegebenem Zeilenindex),
- Veränderung von Datenbankinhalten direkt in der Ergebnismenge (*"in place update"*),
- Übertragung *mehrerer* Ergebniszeilen während einer Weiterschaltungsoperation der Ergebnismenge,
- konfigurierbarer Isolationsgrad für Änderungsoperationen,
- Aggregation mehrerer SQL-Befehle zu einer einzigen logischen *"Batch"*-Operation, die dann mit einem einzigen Methodenaufruf ausgeführt werden kann (Verringerung der Kommunikation mit der Datenbank),
- verbesserte Unterstützung von Datentypen zur Speicherung großer Binär- und Textdaten (BLOB, CLOB),
- Unterstützung benutzerdefinierter SQL-Typen (SQL 3).

Diese zweite Version des JDBC-Pakets wurde erst im Sommer 1998 vorgestellt, so daß zum Entstehungszeitpunkt dieser Arbeit leider noch keine Implementierungen dieses Pakets für kommerzielle Datenbanken zur Verfügung stehen. Einen Einblick in JDBC 2.0 und einen Ausblick auf künftige Erweiterungen, wie z.B. die direkte Speicherung von Java-Objekten innerhalb einer relationalen Datenbank, findet man in [Klu98].
Trotz der zahlreichen Detailverbesserungen bleibt aber ein wesentlicher Aspekt des *Impedance Mismatches* zwischen relationaler Datenbank und Java erhalten. Attribute eines SQL-Datensatzes müssen immer noch mit speziellen Methoden Wert für Wert gelesen oder manipuliert werden. Der Zugriff auf einen Datensatz als zusammenhängendes, eigenständiges Java-Objekt ist nicht möglich.
Während JDBC eine objektorientierte Variante des dynamischen SQL (vgl. Abschnitt 2.1.2) darstellt, ist JSQL [BBG+97] ein Vorschlag der Firmen ORACLE, IBM, SYBASE und TANDEM für den Datenbankzugriff über eingebettetes SQL (vgl. Abschnitt 2.1.1). Dabei kann SQL-Code direkt innerhalb von Java-Programmen verwendet werden. Durch einen speziellen Präprozessor wird dann das eingebettete SQL in Aufrufe des JDBC-Pakets übersetzt. Vorteil dieser Vorgehensweise ist ein wesentlich schlankerer Programmcode, da für den Anwendungsprogrammierer der mit dynamischem SQL verbundene Verwaltungsaufwand zur Analyse und

[1] Beispielsweise unter http://www.javasoft.com/products/jdbc/jdbc.vendors.html findet sich eine Liste von Herstellern, die JDBC unterstützen.

Manipulation datenbankspezifischer Datenstrukturen entfällt. Die Nachteile des eingebetteten SQL sind auch hier die Notwendigkeit eines Präprozessors und die mangelnde Fähigkeit zur dynamischen SQL-Erzeugung während der Laufzeit des Programms.

Java Blend

JAVA BLEND soll die semantische Lücke schließen, die noch zwischen dem Objektmodell von Java und der von JDBC gebotenen Funktionalität besteht [Jav]. Ähnlich wie die in dieser Arbeit bereits vorgestellten Produkte von PERSISTENCE (Abschnitt 4.4) und ONTOS (Abschnitt 4.5) kann JAVA BLEND entweder zu einem Datenbankschema passende Java-Klassen generieren oder zu Java-Klassen Tabellen in einer Datenbank erzeugen. Wesentliche Eigenschaften von JAVA BLEND sind

- bidirektionale, automatische Abbildung zwischen Klassen und Tabellen (Steuerung der Abbilung durch den Anwendungsentwickler möglich),
- Umsetzung von Fremdschlüsseln in Objektreferenzen und umgekehrt,
- optimistisches und pessimistisches Sperren,
- Caching von Objektmanipulationen und Speicherung in der Datenbank am Transaktionsende,
- ODMG-konforme Abbildungs- und Anfragemechanismen (*Object Data Management Group* [CB97]).

JAVA BLEND nutzt die von JDBC gebotene Funktionalität und ist damit für alle Datenbanksysteme einsetzbar, die auch JDBC unterstützen. In den Vergleich objektrelationaler *Middleware* in Kapitel 4 kann JAVA BLEND leider nicht mehr einbezogen werden, da es erst seit Sommer 1998 verfügbar ist und von den Firmen SUN und BAAN nicht wie viele andere Java-Pakete frei, sondern als kommerzielles Produkt vertrieben wird. Technische Dokumentation und erste Testergebnisse sind daher noch nicht öffentlich verfügbar.

Objektrelationale Middleware für Java

Die Integration relationaler Datenbanken in Java-Applikationen wird mittlerweile von einer Vielzahl an Firmen unterstützt, die bereits vor Einführung von Java Erfahrungen mit objektrelationaler *Middleware*, insbesondere für das *Forward Engineering*, gesammelt haben. Einige interessante Vertreter dieser Klasse seien im folgenden kurz aufgeführt.

Persistence Power Tier for Java stellt die Funktionalität von PERSISTENCE (vgl. Abschnitt 4.4) für Java-Anwendungen zur Verfügung. Die aktuelle Version ist CORBA-kompatibel, d.h. Objekte innerhalb des Objektspeichers können über CORBA-Schnittstellen manipuliert werden [PPT].

O_2 Java Relational Binding ist ein Werkzeug für das *Forward Engineering* von Datenbankanwendungen für Java. Beliebige Java-Klassen werden auf Strukturen relationaler Datenbanken abgebildet. Neben dem Zustand eines Objekts kann auch ausführbarer Bytecode in der Datenbank gespeichert werden, so daß die Datenbank als Java-Objektspeicher genutzt werden kann [O2].

POET - SQL Object Factory dient ebenfalls der dauerhaften Speicherung von Java-Objekten in relationalen Datenbanken. Ziel dieses Produkts ist die Bereitstellung der Programmierschnittstellen und des Verhaltens des objektorientierten Datenbanksystems POET auf der Basis einer relationalen Datenbank. Dabei kann gleichzeitig auf verschiedene Datenbanken zugegriffen werden. Objektreferenzen über Datenbankgrenzen hinweg sind möglich [POF].

Werkzeuge für das *Forward Engineering* von Datenbankanwendungen in Java werden mittlerweile von den meisten Anbietern objektorientierter Datenbanksysteme und Java-Entwicklungsumgebungen angeboten.

8.3 Zusammenfassung

Wie die vorigen Abschnitte dieses Kapitels zeigen, ist eine deutliche Annäherung von relationaler und objektorientierter Technik zu beobachten. Auf der einen Seite erkennen Entwickler relationaler Datenbanken die Vorteile objektorientierter Softwareentwicklung und versuchen, diese in ihren Systemen nutzbar zu machen. Auf der anderen Seite erkennen Hersteller objektorientierter Systeme, daß sie durch die Zugriffsmöglichkeiten auf relationale Systeme auch solche Kunden für sich gewinnen können, die vor einem radikalen Umstieg auf rein objektorientierte Technik zurückschrecken. Es ist zu vermuten, daß in Zukunft Anwendungssysteme entstehen werden, in denen die ehemals strikte Trennung zwischen relationaler und objektorientierter Technik nicht mehr erkennbar ist und die sich jeweils derjenigen Konzepte aus beiden Welten bedienen, die für die konkrete Anwendung den größten Nutzen versprechen.
Relationale Datenbanken und objektorientierte Systeme werden damit Bestandteil eines grösseren Rahmenwerks, dessen Komponenten nicht mehr alternativ oder gar einander ausschließend, sondern in einer für die jeweilige Anwendung sinnvollen Kombination eingesetzt werden.
Obwohl also bereits Systeme existieren, die eine produktiv nutzbare Integration von relationaler und objektorientierter Technik leisten, fehlt diesen Systemen bisher eine gemeinsame theoretische Grundlage. Als ein erster Schritt, diese praktischen Entwicklungen auch theoretisch zu untermauern, kann das "Dritte Manifest" von Date und Darwen [DD98] betrachtet werden. Die beiden Autoren zeigen zunächst die substantiellen Unterschiede zwischen relationalen und objektorientierten Konzepten auf und versuchen dann, auf Basis der relationalen Theorie Eckpfeiler einer neuen, objektrelationalen Theorie zu entwickeln. Das "Dritte Manifest" soll als Vorschlag dienen, auf dessen Grundlage zukünftige objektrelationale Systeme entwickelt werden können. Sollte sich dieser Vorschlag durchsetzen, müßten sich Anwender und Entwickler nicht länger mit den unterschiedlichen Begriffssystemen und den Modellen zweier verschiedener Theorien auseinandersetzen, sondern sie können mit einem einzigen, konsistenten, theoretisch abgesicherten, objektrelationalen Modell arbeiten.

Literaturverzeichnis

[ADA] ADABAS D. http://www.sag.de/produkte/datamanage/adabasd/default.htm.

[AKK95] S. Agarwal, C. Keene, and A.M. Keller. Architecting object applications for high performance with relational databases. Technical report, Persistence Software, San Mateo, CA, USA, 8 1995.

[BBG+97] D. Birsdall, B. Burshteyn, Clossman G., P. Cotton, J. Klein, D. Rosenberg, and F. Zemke. Jsql: Embedded sql for java. Technical report, JSQL Consortium, 6 1997. http://www.oracle.com/nca/java_nca/jsql/html/jsql-ansi-proposal_7.html.

[Boo94] G. Booch. *Object-oriented analysis and design with applications.* Benjamin/Cummings, 2nd edition, 1994.

[Cat94] R.G.G. Cattell. *Object Data Management: Object oriented and extended relational database systems.* Addison-Wesley, 1994.

[CB97] R.G.G. Cattell and D.K. Barry, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann, 5 1997.

[Che76] P. P.-S. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Con] Oracle ConText Cartridge. http://www.oracle.com/st/cartridges/context/.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[Dat95] C.J. Date. *An Introduction to Database Systems.* The systems programming series. Addison-Wesley, 6th edition, 1995.

[DB2] IBM DB/2. http://www.software.ibm.com/data/db2/.

[DD97] C.J. Date and H. Darwen. *A Guide to the SQL Standard.* Addison-Wesley, 4th edition, 1997.

[DD98] C.J. Date and H. Darwen. *Foundation for object/relational databases: The Third Manifesto.* Addison-Wesley, 1998.

[Dem96] B. Demuth. Zweckgemeinschaft - Verschmelzung von objektorientierten und relationalen Systemen. *iX*, 5:140–148, 1996.

[Dem97] B. Demuth. Friedlich vereint - Persistence: Mittler zwischen Objekten und Relationen. *iX*, 7:91–97, 1997.

[Die98] J. Diercks. Am Anfang war das BLOB - Multimedia DBMS. *iX*, pages 94–99, 8 1998.

[Ern98] M. Ernst. *Typüberprüfung in einer polymorphen objektorientierten Programmiersprache - Analyse, Design und Implementierung eines Typprüfers für Tycoon-2.* Studienarbeit, Universität Hamburg, Fachbereich Informatik, 2 1998.

[Gei96] K. Geiger. *Inside ODBC.* Microsoft Press, Unterschleißheim, 1996.

[GHJV96] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Professional Computing Series. Addison-Wesley, 1996.

[GJS96] J. Gosling, B. Joy, and G.L. Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, 1996.

[GM95] A. Gawecki and F. Matthes. Tool: A persistent language integrating subtyping, matching and type quantification. Technical Report FIDE/95/135, FIDE, 1995.

[GMSS97] A. Gawecki, F. Matthes, J.W. Schmidt, and S. Stamer. Persistent object systems: From technology to market. Report, TU Hamburg-Harburg, Higher-Order GmbH, 1997. http://www.sts.tu-harburg.de/papers/1997/GMSS97/gi97.ps.gz.

[HCF97] G. Hamilton, R. Cattell, and M. Fisher. *JDBC Database Access with Java - A Tutorial and Annotated Reference.* The Java Series. Addison-Wesley, 1997.

[Inf] Informix Dynamic Server. http://www.informix.com/informix/products/ids/.

[Jac92] I. Jacobson. *Object-oriented software engineering: a case driven appoach.* Addison-Wesley, 1992.

[Jav] Java Blend. http://java.sun.com/products/java-blend/.

[JL96] R. Jones and R. Lins. *Garbage Collection: Algorithms for automatic dynamic memory management.* Wiley & Sons, 1996.

[KJA93] A. M. Keller, R. Jensen, and S. Agarwal. Persistence software: Bridging object-oriented programming and relational databases. *SIGMOD Record*, 22(2):523–528, 1993.

[Klu98] R. Klute. Kaffeesatz für Datenbanker - Neue Funktionen in JDBC 2.0. *iX*, pages 108–110, 9 1998.

[Loo95] M. E. S. Loomis. *Object Databases: The Essentials.* Addison-Wesley, 1995.

[Lou94] K.C. Louden. *Programmiersprachen - Grundlagen, Konzepte, Entwurf.* Thomson Publishing, 1994.

[LS87] P.C. Lockemann and J.W. Schmidt, editors. *Datenbank-Handbuch.* Informatik-Handbücher. Springer-Verlag, 1987.

[Mat93] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer, 1993.

[MSht] F. Matthes and J.W. Schmidt. *Datenbank-Handbuch*, chapter Kapitel 1. Informatik-Handbücher. Springer-Verlag, 2. edition, unveröffentlicht.

[MSS] Microsoft SQL Server. http://www.microsoft.com/products/prodref/152_ov.htm.

[MU97] G. Matthiessen and M. Unterstein. *Relationale Datenbanken und SQL.* Addison-Wesley, 1997.

[O2] O_2 Java Relational Binding. http://www.o2tech.fr/jrb/wpaper.html.

[OCI97] Oracle Corporation. *Programmer's Guide to the Oracle Call Interface*, 1997. Shipped with the relational database management system Oracle.

[OF197] Application building with object factory. Technical report, Rogue Wave Software Inc., Corvallis, OR, USA, 1 1997.

[OF397] Using object factory to eliminate repetitive coding in database applications. Technical report, Rogue Wave Software Inc., Corvallis, OR, USA, 3 1997.

[OMG] Object Management Group. http://www.omg.org.

[OPR96] R. Otte, P. Patrick, and M. Roy. *Understanding CORBA: The Common Object Request Broker Architecture.* Prentice Hall, 1996.

[Ora] Oracle 8. http://www.oracle.com/st/products/uds/oracle8/html/oracle8.html.

[POF] POET SQL Object Factory. http://www.poet.de/d_prodkt/d_o_sql4/d_o_sql4.htm.

[PPT] Persistence Power Tier for Java. http://www.persistence.com/products/PTejb.html.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[RW*96] Increasing productivity with DBTools.h++. Technical report, Rogue Wave Software Inc., Corvallis, OR, USA, 6 1996.

[Sip96] R. Sippl. SQL Access Groups's Call-Level Interface - an independent interface for database development. *Dr. Dobb's Database Sourcebook*, 01 1996. http://www.ddj.com/ddsbk/1996/1996_01/sippl.htm.

[Sku98] M. Skusa. *The Tycoon SQL Gateway - User's Guide and Reference Manual.* Higher-Order GmbH, Hamburg, Germany, 1 1998.

[SM96] M. Stonebraker and D. Moore. *Object-relatinal DBMSs: The next great wave.* The Morgan Kaufmann series in data management systems. Morgan Kaufmann, 1996.

[Syb] Sybase Adaptive Server. http://www.sybase.com/adaptiveserver/.

[Tex] Informix Text DataBlades.
www.informix.com/informix/products/options/udo/datablade/dbmodule/excalibur1.htm
www.informix.com/informix/products/options/udo/datablade/dbmodule/fulcrum1.htm.

[UML97a] UML notation guide. Technical report, Rational Software Corp., Santa Clara, CA, USA, 1 1997.

[UML97b] UML semantics. Technical report, Rational Software Corp., Santa Clara, CA, USA, 1 1997.

[Wah97] J. Wahlen. Einführung in Tycoon-2. Overhead-Folien, 5 1997. Tycoon-2-Einführungskurs.

[Wei98] M. Weikard. *Entwurf und Implementierung einer portablen multiprozessorfähigen virtuellen Maschine für eine persistente, objektorientierte Programmiersprache*. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, 1998.

[Wet94] I. Wetzel. *Programmieren mit STYLE - Über die systematische Entwicklung von Programmierumgebungen*. Peter Lang, Europäischer Verlag der Wissenschaften, 1994. Dissertation, Fachbereich Informatik, Universität Hamburg.

[Wie97] A. Wienberg. *Bootstrap einer persistenten objektorientierten Programmierumgebung*. Studienarbeit, Universität Hamburg, Fachbereich Informatik, 12 1997.

Erklärung

Die Diplomarbeit wurde von mir selbständig durchgeführt. Dabei habe ich keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.

Hamburg, den 9. September 1998