Universität Paderborn

Fachbereich 17 - Mathematik/Informatik

Arbeitsgruppe Softwaretechnik

Warburger Straße 100

D-33098 Paderborn

Entwicklung eines Generators für eine objektorientierte Zugriffsschicht auf einer relationalen Datenbank

Diplomarbeit

für den integrierten Studiengang Informatik

im Rahmen des Hauptstudiums II

Felix Wolf

Horner Hellweg 72

33100 Paderborn

vorgelegt bei

Prof. Dr. Schäfer

und

Prof. Dr. Engels

Paderborn, im Oktober 2001

Ich versichere, diese Arbeit ohne fremde Hilfe und ohne Verwendung von weiteren, hier in den Referenzen nicht aufgelisteten, Quellen angefertigt zu haben, und daß diese Arbeit in dieser oder einer ähnlichen Form noch keiner Prüfungsbehörde vorgelegen hat. Sie wurde auch nicht als Teil einer anderen Prüfung angenommen. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Ort, Datum	Unterschrift

DANKE

Ich bedanke mich bei meinen Eltern, die mir die nötige Erziehung gegeben und mich bei dem Informatik-Studium immer unterstützt haben.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Neue Entwicklungen wie E-Commerce, Telearbeit oder die Globalisierung von Unternehmen stellen auch neue Anforderungen an vorhandene Systeme.

Die vorhandenen Informationssysteme sind meist ungeeignet, die neuen Anforderungen gewährleisten zu können. Durch das einfach gehaltene Datenmodell, das den relationalen Datenbanksystemen zugrundeliegt, sind die relationalen Datenbanksysteme nicht in der Lage, komplexe Datenstrukturen, wie zum Beispiel eine hierarchische Datenstruktur, effizient abzuspeichern. Diese komplexen Daten werden aber von den oben genannten Anwendungsbereichen vorausgesetzt.

Im Gegensatz dazu bietet sich das objektorientierte Datenmodell an, die neuen Entwicklungen umzusetzen, da seine Definition die Verwaltung solcher komplexer Strukturen anbietet.

Da die vorhandenen Informationssysteme bereits seit mehreren Jahren im Einsatz sind und in dieser Zeit auch erweitert wurden, ist es heute sehr schwer möglich, diese Systeme zu überschauen bzw. zu verstehen. Im Laufe der Zeit haben unterschiedliche Personen Änderungen an den Systemen vorgenommen. In den meisten Fällen sind die Änderungen undokumentiert und die Personen, die man noch fragen könnte, haben das Unternehmen bereits verlassen.

Dadurch wird die Einführung eines neuen Systems sehr erschwert, wenn nicht sogar unmöglich gemacht, da der Zeitaufwand für die anstehende Analyse des bestehenden Systems und die anschließende Datenübernahme von dem alten System in das neue System immens ist.

Hierzu werden insbesondere Konzepte und Techniken benötigt, mit deren Hilfe relationale in objektorientierte Systeme überführt (migriert) bzw. integriert werden können.

Ein erster Schritt in diese Richtung sind objektorientierte Zugriffschichten für relationale Datenbanken. Um möglichst flexibel und unabhängig von den Betriebssystemen und der jeweiligen Datenbank zu sein, soll die Zugriffschicht in Java implementiert werden und der Datenbankzugriff über JDBC erfolgen. Die

objektorientierte Zugriffschicht soll mit UML modellierbar sein und einen lesenden wie auch schreibenden Zugriff auf die relationale Datenbank ermöglichen.

1.2. Ziel der Arbeit

Diese Diplomarbeit befaßt sich mit der Erstellung eines Generators für den objektorientierten Zugriff auf eine relationale Datenbank. Als Basis der Implementierung diente das Projekt *FUJABA* [FUJ] und die Implementierung verwendet bei der Generierung dessen UML Meta-Schema. FUJABA verfügt ohne diese Arbeit über keine Modellierungsmöglichkeit von relationalen Datenbanken wie ein ER[1]-Diagramm, in dem man die Struktur von den relationalen Tabellen beschreiben und ändern kann.

Die Arbeit umfaßt neben der Generierung die Möglichkeit, auf eine zunächst einmal beliebige relationale Datenbank zuzugreifen und aus ihr die Schema-Informationen auszulesen. Die Informationen können in einer Darstellung ähnlich zu dem ER-Diagramm angezeigt werden.

Von dem Diagramm bzw. den relationalen Schemata, die auf dem Diagramm beschrieben werden, kann ein initiales konzeptionelles UML-Schema erzeugt werden. Über den Generator wird auf Basis des relationalen Schemas und des konzeptionellen Schemas die objektorientierte Zugriffsschicht generiert. Der Generator besorgt sich die dafür notwendigen Informationen aus beiden Diagrammarten.

Der generierte Code ist ein ausführbarer “Server”, der es gestattet, auf eine relationale Datenbank objektorientiert zuzugreifen. Die Diplomarbeit hatte nicht zur Aufgabe, die beiden oben beschriebenen Schemata zueinander *konsistent* zu halten. Falls nach der Erzeugung des konzeptionellen Schemas eine Änderung auf einem der beiden Schemata durchgeführt wird, sind die beiden Schemata nicht mehr konform zueinander. Sie sagen nicht mehr das gleiche aus.

Eine beliebige Datenbank muß dahingehend eingeschränkt werden, daß sie über einen JDBC2.0-Treiber[JDB] verfügt und die Datenbank gestattet, die Struktur-Informationen von ihr auszulesen.

Die Diplomarbeit bildet die Grundlage für das Projekt *REDDMOM*[RED]. Das Projekt erweitert diese Ausarbeitung um Modellierungskonzepte, die sich mit der Spezifikation von verteilten Datenbanken und dem Zugriff auf Multimedia-Objekte befassen.

Das Projekt REDDMOM ist desweiteren eine Weiterentwicklung des Projektes *VARLET [VAR],* das das *Reengineering* von relationalen Datenbanken zum Schwerpunkt hatte. Der Begriff Reengineering kann in die Begriffe *Reverse-Engineering* und *Forward-Engineering* unterteilt werden.

[1] ER: Entity Relationship

Unter dem ,,Reverse-Engineering einer relationalen Datenbank" vesteht man zunächst das Ermitteln der vorhandenen Struktur der Datenbank. Durch Analyseverfahren werden bestehende, aber vergessene Eigenschaften der Datenbank wiedergewonnen. Die wiedergewonnenen Erkenntnisse reichern die Struktur der Datenbank an. Abgeschlossen wird der Reverse-Engineering Prozeß durch die Überführung der vorhandenen, angereicherten Struktur in eine neue Struktur. Beide Strukturen müssen aber das gleiche aussagen bzw. repräsentieren. Diese Diplomarbeit umfaßte also auch die teilweise Implementierung eines Reverse-Engineering Prozesses, der in Zukunft noch erweitert wird.

Das Projekt VARLET beschränkte sich in dem Reverse-Engineering-Prozeß auf eine Datenbank. Das Projekt REDDMOM wird die in VARLET entwickelten Verfahren auf mehrere, verteilte Datenbanken ausbauen und neue Verfahren, die sich speziell auf verteilte Datenbanken beziehen, entwickeln.

Unter Forward Engineering versteht man die Erstellung neuer Systeme, die ihren Ursprung in der neuen Struktur, die aus dem Reverse-Engineering gewonnen wurde, haben.

Die Erzeugung des oben erwähnten initialen Schemas ist eine Neu-Implementierung des Algorithmus, der in dem Projekt *VARLET[VAR]* verwendet wurde. Auch das Erkennen der oben erwähnten Inkonsistenz war Bestandteil des Projektes in Form einer Diplomarbeit [Wad98].

In der weiteren Entwicklung von REDDMOM werden nach und nach die Konzepte von VARLET übernommen, somit auch die Konsistenz-Erhaltung aus der oben erwähnten Diplomarbeit.

Für nähere Informationen wird hier auf die Internet-Seiten der drei Projekte verwiesen [FUJ, RED, VAR].

1.3. Gliederung der Arbeit

Das Kapitel 2 gibt einen Überblick über die von dem Generator verwendeten Grundlagen. In dem Kapitel 3 werden mit dieser Arbeit verwandte Ansätze vorgestellt. Das 4. Kapitel befaßt sich mit der konzeptionellen Umgebung, auf die der Generator aufsetzt und in dem Kapitel 5 wird der Generator selbst vorgestellt. Abgerundet wird die Diplomarbeit mit dem Kapitel 6, in dem die Arbeit zusammengefaßt und ein Ausblick auf zukünftige Arbeiten gegeben wird.

2. Grundlagen

Dieses Kapitel gibt einen Überblick über die Bereiche, die von dieser Arbeit berührt werden. Es werden Begriffe und Definitionen aus diesen Bereichen erläutert. Die hier angesprochenen Theorien haben Auswirkungen auf die Realisierung des Generators. An geeigneten Stellen wird ein Ausblick auf weitere Kapitel dieser Diplomarbeit gegeben.

2.1. Relationales Datenmodell vs. Objektmodell

Der in dieser Arbeit vorgestellte Generator soll den Zugriff auf eine relationale Datenbank über eine objektorientierte Zugriffsschicht ermöglichen. Es wird also die mengenorientierte relationale Welt mit der auf ein Objekt fixierten objektorientierten Welt verbunden. Dabei müssen die Eigenschaften aus beiden Welten beachtet werden. Beiden Welten liegt ein eigenes Modell zugrunde. Auf relationaler Seite wird das Modell als *relationales Datenmodell* bezeichnet. Das objektorientierte wird dementsprechend *Objekt-Modell* genannt. Die Modelle sind unabhängig voneinander entstanden, und es wurde bei der Definition des einen Modells nicht auf das andere Rücksicht genommen, sodaß bei der Zusammenführung der beiden Welten Konflikte entstehen können und in der Realität auch vorkommen.

Für den internen Zugriff auf die relationale Datenbank wurde das *JDBC* [JDB] -Paket verwendet. In diesem Kapitel soll eine kurze Vorstellung dieses Paketes genügen. Im Laufe der Diplomarbeit werden weitere Eigenschaften des JDBC-Paketes vorgestellt.

2.1.1. Relationales Datenmodell

Jede *Relation* erhält im *relationalen Datenmodell* einen *Relationennamen. Die* Relation wird in Tabellenform dargestellt. Die Spalten der Tabelle werden über die Attribute der Relation definiert. Die Attribute legen die Struktur der Relation fest. Die Struktur wird als *Relationenschema* bezeichnet. Wie in Abbildung 2.1

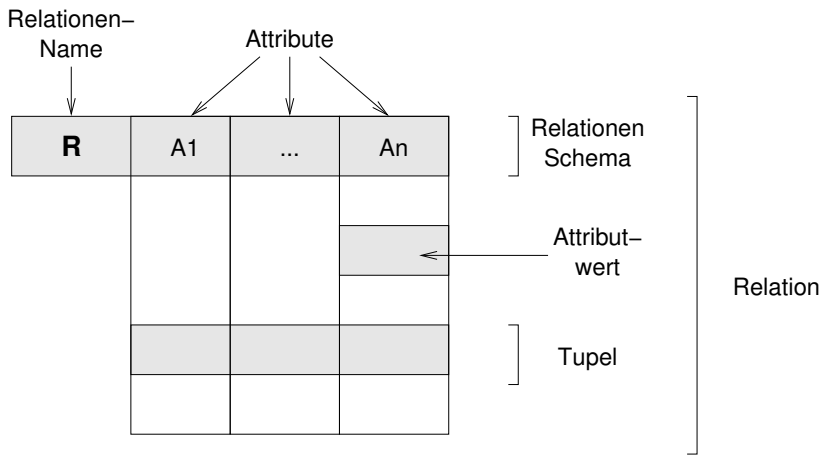


Abbildung 2.1.: Das Relationale Modell [Sch98]

zu sehen ist, entsprechen die Zeilen der Tabelle den *Tupeln* der Relation, die die gespeicherten Datensätze der Relation darstellen. Jedes Attribut in einem Tupel erhält einen *Attributwert*. Da die Tupel einer Relation eine Menge bilden, besteht keine feste Reihenfolge zwischen ihnen. Das Relationenschema und die Tupel bilden zusammen die Relation.

Der Primärschlüssel eines Relationenschemas wird über ein oder mehrere Attribute des Schemas gebildet. Es ist nicht möglich, zwei Tupel in einer Relation zu speichern, deren Primärschlüssel gleich sind. Alle Attribute eines Relationenschemas, die nicht in der Attributmenge des Primärschlüssels enthalten sind, werden als *Nichtschlüsselattribute* (NSA) bezeichnet.

Es bestehen zwischen den einzelnen Attributen eines Relationenschemas *funktionale Abhängigkeiten*. Der Primärschlüssel ist eine besondere Art von funktionaler Abhängigkeit, da sich aus den Werten des Primärschlüssels die Werte der Nichtschlüsselattribute bestimmen lassen. Daneben kann es eine weitere funktionale Abhängigkeit geben, die in der Theorie ungewollt ist, aber nicht verhindert und in der Praxis unter Umständen auch bewußt verwendet wird. Es ist die funktionale Abhängigkeit zwischen Attributmengen eines relationalen Schemas, wobei die Attributmenge, von der die andere Menge abhängt, nicht der Primärschlüssel des Schemas ist. Dabei ergibt sich aus der Betrachtung aller Tupel eines relationalen Schemas, daß die Werte der einen Attributmenge aus den Werten der anderen Menge geschlossen werden können. Im Normalfall ist diese Art der funktionalen Abhängigkeit ungewollt, da durch die mehrfache Haltung der gleichen Daten unnötig Platz verbraucht wird und die Gefahr der Inkonsistenz entsteht. Falls die funktionale Abhängigkeit durch ein Programm und nicht durch den Datenbank-Server errechnet wird, kann es zu einer Redundanz der Daten kommen, wenn sich die Funktion des Programms ändert.

Es stehen in der relationalen Modellierung Mittel zur Verfügung, diese funktionale

Abhängigkeit aus den relationalen Schemata zu entfernen. Die Änderung der Schemata wird als *Normalisierung* der Relationenschemata bezeichnet. Der Grad der Normalisierung wird in sechs *Normalformen* unterteilt, in die Normalformen 1 bis 5 und in die *Boyce-Codd-Normalform* (BCN).

Die Normalformen bauen aufeinander auf, sodaß ein in dritter Normalform vorliegendes Relationenschema auch die Kriterien für die erste und die zweite Normalform erfüllt.

Die erste Normalform schließt Kollektionstypen wie Set, Bag oder List innerhalb eines Relationenschemas aus. Damit kann ein Relationenschema nur atomare Attribute enthalten.

Die zweite Normalform verbietet die partielle Abhängigkeit eines Nichtschlüsselattributs von einem Schlüssel; das Nichtschlüsselattribut darf nicht von einer echten Teilmenge der Attributmenge des Primärschlüssels abhängen.

Die dritte Normalform verbietet die transitive Abhängigkeit eines Nichtschlüsselattributes von einem Primärschlüssel. Dabei versteht man unter der transitiven Abhängigkeit, daß es in einer Relation eine Attributmenge gibt, die von anderen Attributmenge, die nicht den Primärschlüssel der Relation definiert, abhängt.

Die BCN ist eine Verfeinerung der dritten Normalform. Während sich die dritte Normalform nur auf die Nichtschlüsselattribute beschränkt, dürfen bei der BCN auch in den Schlüsselattributen keine transitiven Abhängigkeiten auftreten.

Die vierte und die fünfte Normalform haben keine besondere praktische Bedeutung, sodaß an dieser Stelle auf die Literatur verwiesen wird [EN94].

Neben der funktionalen Abhängigkeit über den Primärschlüssel eines Relationenschemas gibt es die *Inklusionsabhängigkeit* bzw. *Inclusion Dependency* zwischen Relationenschemata. Hierbei sind alle Primärschlüssel-Attribute des einen Relationenschemas in der Attributmenge des anderen Schemas enthalten. Diese spezielle Teilmenge wird als *Fremdschlüssel* bezeichnet. Die allgemeine Einführung der beiden Abhängigkeitsbegriffe ist für die Arbeit ausreichend. Für eine genauere Definition sei an dieser Stelle auf [EN94] verwiesen.

2.1.2. Das Objekt-Orientierte Modell

Die *Objektorientierung* kann in die folgenden Bereiche eingeteilt werden:

- *Objektorientierte Konzepte*:
  Die reine Definition der Elemente, die in der Objektorientierung vorkommen, wie “Objekt”, “Klasse”, “Nachricht” oder “dynamisches Binden”. Diese Elemente werden auch als objektorientierte Sprachelemente bezeichnet. Die objektorientierten Konzepte definieren zusammen das später noch näher vorgestellte *Objekt-Modell*.

- *Objektorientierte Systeme*:
  Die Systeme bieten objektorientierte Sprachelemente an. Ein solches objektorientiertes System ist zum Beispiel die Programmiersprache Java[JAV].

- *Objektorientierte Modellierung*:
  Unabhängig von einer Programmiersprache wird eine Methode, ein Problem oder ein komplettes System analysiert und ein Design entworfen. Die Modellierung erfolgt in einer eigenen Sprache, die die objektorientierten Konzepte abstrakt beschreibt. Die Sprachen werden als *Modellierungssprachen* bezeichnet. Zu diesen Sprachen zählt zum Beispiel die *Unified Modeling Language* (UML)[UMLa].

- *Objektorientierte Software-Entwicklung*:
  In diesen Bereich fällt die Organisation und die Strukturierung eines Software-Systems. Hierzu zählt zum Beispiel der *Unified Process[UMLa].*

Das *Objekt-Modell* setzt sich aus den objektorientierten Sprachelementen zusammen. Ein *Objekt* besitzt eine *Struktur* und ein *Verhalten*. Die *Variablen* bzw. *Attribute* eines Objektes definieren seinen Zustand und die *Methoden* sein Verhalten. Jedes Objekt besitzt zu seiner Lebzeit in der Systemwelt eine eindeutige *Objekt-Identität*. Diese Identität ist nicht änderbar, sodaß das Objekt immer über die Identität eindeutig identifiziert werden kann. Es kann zur Laufzeit nicht mehrere Objekte mit der gleichen Identität geben.

Zwei Objekte sind gleich, wenn sie sich in ihrer Struktur und in ihrem Verhalten gleichen. Objekte werden auch als Instanzen einer *Klasse* bezeichnet. Die Klasse beschreibt die Struktur und das Verhalten aller Objekte der Klasse.

Die *Attribute* eines Objektes nehmen die Daten des Objektes auf. Über den *Attributnamen* ist das Attribut in dem Objekt eindeutig bestimmt. Der *Datentyp* des Attributes gibt an, welche Werte das Attribut enthalten darf. Es wird der Wertebereich des Attributes festgelegt. Man spricht von einer *Referenz*, wenn der Datentyp eines Attributes selbst wieder ein Objekt ist. Ein Objekt mit einer Referenz als Attribut wird *komplexes Objekt* genannt.

Eine Referenz wird aus Sicht der Klasse als *Assoziation* bezeichnet. Sie baut eine Beziehung zu einer anderen Klasse auf. Eine Assoziation besitzt einen *Namen* und eine *Kardinalität*. Typische Kardinalitäten sind 1:1, 1:n oder n:m. Die Kardinalität drückt das Beziehungsverhältnis zwischen den beiden Klassen aus. Eine Diplomarbeit hat zum Beispiel n Seiten. Jede Seite aus der Arbeit gehört aber genau zu der einen Arbeit und nicht noch zu einer zweiten. In diesem Beispiel hätte man also eine 1:n-Beziehung zwischen einer Diplomarbeit und ihren Seiten.

Spezialfälle der Assoziation sind die *Aggregation*, die *Composition* und die *Vererbung*. Alle drei Assoziationsarten drücken eine Enthaltensein-Beziehung aus.

Obwohl bei der Aggregation und der Composition ein Bezug wie bei der Assoziation zwischen zwei Objekten besteht, wird das eine Objekt dem anderen Objekt untergeordnet. Das übergeordnete Objekt wird als Ganzes (whole) und das untergeordnete als Teil (part) bezeichnet. Das Ganze setzt sich aus Teilen zusammen. Im Gegensatz zu der Aggregation wird bei der Composition verhindert, daß ein Teil ohne sein Ganzes existieren kann. Außerdem muß das Ganze zu jeder Zeit seiner Existenz einen Teil aufweisen.

Die Vererbung wird auch *is-a-Beziehung* genannt. Die Kardinalität ist bei dieser Beziehung 1:1. Dabei erbt die eine Klasse die Struktur und das Verhalten der anderen Klasse. Die Klasse, von der die Struktur und das Verhalten geerbt wurde, bezeichnet man als *Oberklasse*. Die andere Klasse wird als *Unterklasse* bezeichnet.

Ein Objekt besitzt eine *Schnittstelle*, in der seine Methoden bzw. Operationen öffentlich gemacht werden. Der Zugriff auf ein Objekt, und damit auch der Zugriff auf die Attribute eines Objektes, erfolgt nur über die in seiner Schnittstelle definierten Operationen. Eine Schnittstelle definiert den Kontext, in dem das Objekt verwendet werden kann. Ein Objekt kann mehr als eine Schnittstelle haben; eine Schnittstelle ist unabhängig von der konkreten Implementierung des Objektes.

Unified Modeling Language (UML)[UMLa]

Generell wird die Entwicklung einer Applikation, also einer Software, als *Software-Entwicklungsprozeß* bezeichnet. Der Software-Entwicklungsprozeß kann als *Wasserfall-Modell* definiert werden. Dabei besteht das Modell aus den in Abbildung 2.2 zu sehenden Schichten. Der Prozeß umfaßt also alle notwendigen Arbeitsschritte von der Analyse bis zur Wartung.

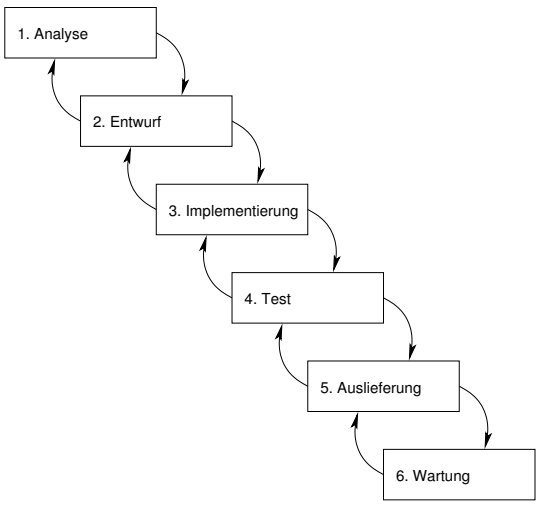


Abbildung 2.2.: Der Software-Entwicklungs-ProzeSS als Wasserfall-Modell

Die *UML* ist eine standardisierte graphische Notation für objektorientierte Modelle. Sie besteht aus einer Anzahl unterschiedlicher Diagramme, die spezielle Aufgaben bei der Entwicklung eines Software-Systems übernehmen. Die einzelnen UML-Diagramme lassen sich in drei Modellgruppen aufteilen:

- *Analyse-Modell*:
  Die Modellkategorie unterstützt die Analysephase des Software-Entwicklungsprozesses. Der Entwickler verschafft sich einen Überblick über die Anforderungen des Systems. Danach kann er die Funktionalität und den Aufwand des Systems ermitteln bzw. abschätzen. Zu dieser Kategorie zählt das *Use-Case-Diagramm.*

- *Statisches Modell*:
  Hier wird die Struktur des Systems entworfen. Auf dem *Klassen-Diagramm* wird das Konzept des Systems erstellt. Das *Package-Diagramm* sorgt für die logische Struktur.

- *Dynamisches Modell*:
  Diese Modellart geht auf die Objekt-Zustände und die Interaktion zwischen den Objekten ein. Die einzelnen Diagramme bieten graphische Notationen an, um die Lebensdauer und die Aktivität von Objekten anzuzeigen. Die Diagramme bieten auch die Möglichkeit, den Nachrichtenaustausch, das Zusammenspiel und die parallelen Abläufe zwischen den Objekten darzustellen. Zu dieser Modellart zählen das *Sequenz-*, das *Kollaborations-*, das *Zustands-* sowie das *Aktivitäts-Diagramm.*

Eine Einordnung der reinen UML in den Software Entwicklungsprozeß beschränkt sich auf die Punkte 1 und 2. Die Software-Entwicklung mittels UML wird heutzutage durch *CASE-Tools*[1] wie Rational Rose [Ros], Together J [Tog] oder FUJABA [FUJ] unterstützt. Diese Programme erlauben, aus dem UML-Design entsprechenden Code einer Programmiersprache zu generieren. FUJABA bietet mit Mr. DOBS[2] eine Unterstützung beim Debuggen an. Der Einsatz von UML mit Hilfe von Case-Tools wirkt sich auf die Punkte 1 bis 4 des Software Entwicklungsprozesses aus.

Der Generator, der in dieser Diplomarbeit vorgestellt wird, erzeugt eine objektorientierte Zugriffsschicht auf eine relationale Datenbank. Dabei entsteht eine Klassenstruktur, die den "nativen" Zugriff auf die Datenbank-Tabellen kapselt. Hierfür ist das UML Klassen-Diagramm entscheidend. Die Diplomarbeit beschränkt sich auf die Erzeugung der Struktur- und Verhaltenbeschreibung der Objekte.

[1] CASE: Computer Aided Software Engineering
[2] DOBS: Dynamic Object Browsing System

Die vollständige Spezifikation mit der graphischen Notation von UML findet man unter [UMLb, UMLa].

2.1.3. Vergleich der Modelle

Zusammenfassend ist festzuhalten, daß das relationale Modell wertbasiert ist. Datentupel mit genau gleichen Werten innerhalb einer Relation werden nicht unterschieden bzw. können nicht vorkommen. Ein Tupel hat keine aus der objektorientierten Programmierung bekannte Identität. Die Identität wird über den Primärschlüssel simuliert. Da die Attribute eines Primärschlüssels auch veränderbar sein können, ergibt sich das Problem, daß der Primärschlüssel wiederum Bestandteil einer Inklusionsabhängigkeit sein kann und die Fremdschlüssel aus anderen Relationen noch auf den alten Wert des Schlüssels verweisen.

Eine moderne Datenbank erlaubt hier Einstellungsmöglichkeiten bei der Definition der Inklusionsabhängigkeit, was in diesem konkreten Fall passieren soll. Unter anderem kann es sein, daß die Datenbank bei der Änderung des Primärschlüssels versucht, auch die Tupel aus anderen Relationen zu ändern, die auf den alten Wert des Primärschlüssels verweisen. Da in der objektorientierten Zugriffsschicht Kopien der relationalen Tupel aus der Datenbank angelegt werden und sich somit auch Kopien der im Hintergrund veränderten Tupel befinden, muß die objektorientierte Schicht wissen, daß die Tupel geändert wurden.

Das noch später in Kapitel 2.1.4 vorgestellte JDBC-Paket[JDB] bietet nicht die Möglichkeit, solche Einstellungen einer Inklusionsabhängigkeit auszulesen, sodaß die eventuell im Hintergrund ablaufenden Änderungen von Tupeln ohne weiteres Zutun nicht bekannt sind. Die Eigenschaft müßte in das in Kapitel 4.1.1 noch näher vorgestellte relationale Datenbank-Schema nachträglich als Anreicherung der Informationen eingetragen werden.

Auf die Implementierung dieser speziellen Anreicherungsmöglichkeit wurde in dieser Diplomarbeit verzichtet.

Bei Neuanlage eines Datenbank-Systems kann das Problem mit dem änderbaren Primärschlüssel ausgeschlossen werden, indem man unveränderbare Identifikatoren für jedes relationale Schema vergibt. Dazu wird jedem relationalen Schema während des Datenbank-Entwurfs ein Primärschlüssel zugewiesen.

Es muß festgelegt werden, ob der Benutzer den Wert des Identifikators bei Neuanlage eines Tupels einmal vergibt oder das Datenbank-Management-System ihn selbst verwaltet.

Für die Eigenverwaltung des Primärschlüssels durch das Management-System bieten aktuelle Datenbanken Attributeinstellungen wie *autoincrement* an. Diese Einstellung sorgt dafür, daß bei Neuanlage eines Tupels das Primärschlüssel-Attribut den nächsten noch nicht vergebenen Wert bekommt. Dabei muß der

Datentyp des autoincrement-Attributes den nächsten noch nicht vergebenen Wert wohldefinieren, wie es zum Beispiel bei numerischen Werten mit x=x+1 der Fall ist. Außerdem ist auf dieses Feld nur lesend zugreifbar.

Die Neuanlage eines Tupels einer Relation, dessen Primärschlüssel ein autoincrement-Attribut ist, hat zur Folge, daß der Wert des autoincrement-Attributes zunächst in der objektorientierten Zugriffsschicht unbekannt ist, da der Wert intern von dem Datenbank-System vergeben wurde. Das Tupel muß nach Anlage erneut ausgelesen werden, um den Wert des autoincrement-Attributes zu erhalten. Das erneute Auslesen solcher Tupel ist in dieser Diplomarbeit ermöglicht worden.

Der Einsatz eines Generators für neuangelegte Systeme ist eine zu große Einschränkung. Wie bereits in Kapitel 1.2 erwähnt, basiert die Diplomarbeit auf den Konzepten des VARLET-Projektes.

Dort war es möglich, zu einem bestehenden Datenbank-System ein objektorientiertes konzeptionelles Schema zu entwerfen. An dem konzeptionellen Schema konnten Umstrukturierungen vorgenommen werden, die, wenn notwendig, auch auf die relationale Basis übertragen wurden. Da relationale Datenbanken auch die Definition von relationalen Schemata ohne Primärschlüssel erlauben, wurde bei der Erzeugung des konzeptionellen Schemas die Strategie angewendet, alle Attribute des relationalen Schemas als Primärschlüssel anzusehen.

Die Richtigkeit dieser Regel kann aus der Definition des relationalen Datenmodells entnommen werden, die besagt, daß es keine zwei Tupel in einem Schema geben kann, die genau gleich sind. Der hier vorgestellte Generator verfolgt das gleiche Prinzip, sodaß auch auf Relationen ohne Primärschlüssel objektorientiert zugegriffen werden kann. Dabei muß aber die Einschränkung gemacht werden, daß das relationale Schema nur über Attribute verfügen darf, die von dem Datenbank-Management-System auch zum Suchen verwendet werden können. Relationale Attribute vom Typ *Blob* zum Beispiel können nicht von jedem Datenbank-Management-System als Suchkriterium ausgewertet werden.

Der zweite Unterschied zwischen den beiden Modellen ist, daß das relationale Modell keine Vererbung besitzt. In der Objekt-Orientierung ist es einfach, Dinge aus der realen Welt in sein Modell zu übernehmen. Die Klassen Auto, Fahrrad und Flugzeug präsentieren die gleichnamigen Gegenstände aus der Realität. Diese Klassen haben etwas gemeinsam. Es sind Fahrzeuge und sie besitzen alle fahrzeugspezifische Eigenschaften wie etwa die Anzahl an zurückgelegten Kilometern. Darüberhinaus besitzt jede Klasse spezielle Eigenschaften. Das Auto hat zum Beispiel einen Motor, das Fahrrad hat stattdessen Pedale und das Flugzeug besitzt einen Düsenantrieb. Die Abbildung 2.3 zeigt die Vererbungsbeziehung in UML-Notation.

Bei der Darstellung in dem relationalen Modell tut man sich schwerer. Das Modell bietet nur die Möglichkeit, die oben dargestellte Struktur auf Tabellen und Inklu-

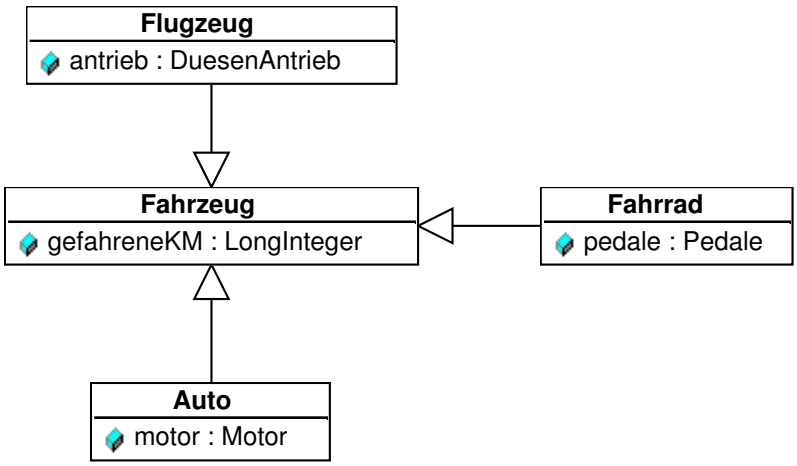


Abbildung 2.3.: UML-Beispiel der Vererbung

sionsabhängigkeiten abzubilden. Dabei bieten sich dem Modellierer zwei Möglichkeiten, die kombinierbar sein können:

1. Jedes Objekt bzw. die Klasse des Objekts erhält in der Datenbank seine eigene Tabelle. Für das obige Beispiel würden vier Tabellen benötigt. Da jedes Auto auch ein Fahrzeug ist, müßte es zu jedem Auto-Tupel auch ein Fahrzeug-Tupel geben. Neben den Fahrzeug-Tupeln, die Autos sind, gibt es dann noch für jedes Fahrrad und jedes Flugzeug weitere Tupel in dem Fahrzeug-Schema. Diese besondere Beziehung kann erreicht werden, indem

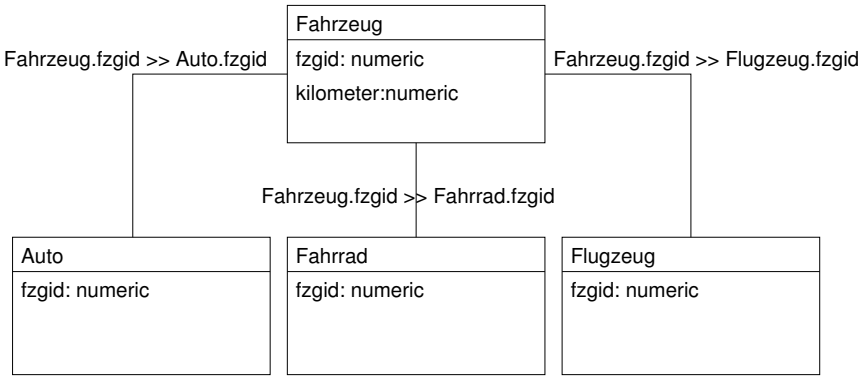


Abbildung 2.4.: Jede Klasse bekommt eigene Tabelle

eine spezielle Inklusionsabhängigkeit über Primärschlüssel zwischen der Tabelle Auto und der Tabelle Fahrzeug definiert wird. Bei dieser besonderen Inklusionsabhängigkeit ist der Fahrzeug-Fremdschlüssel in der Tabelle Auto auch gleichzeitig Primärschlüssel der Tabelle Auto.

2. Die gesamte Klassenhierarchie wird zu einer Tabelle Fahrzeug zusammengefaßt und es werden vier *Varianten* zu jeder Klasse auf dieser Tabelle definiert. Dabei enthalten die Varianten nur die Spalten der Tabelle, die das durch die Variante präsentierte Objekt auch wirklich nur besitzen darf. Die Variante Fahrzeug enthält die Eigenschaften, die alle Fahrzeuge aufweisen. Die Variante Fahrrad enthält alle speziellen Eigenschaften eines Fahrrades

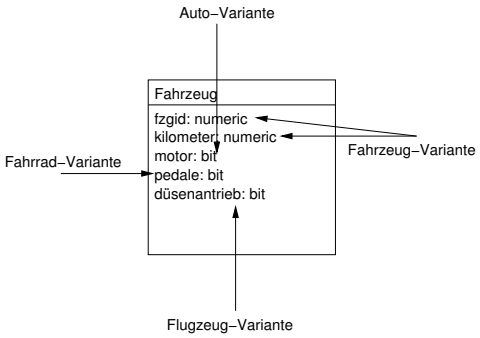


Abbildung 2.5.: Klassen als Tabellenvarianten

und die des Fahrzeuges, aber nicht die eines Autos wie zum Beispiel den Motor. Bei dieser Art des Tabellendesigns geht man davon aus, daß bei einem Fahrrad-Tupel die Werte des Fahrzeuges und die des Fahrrades gefüllt, die anderen hingegen, die entweder ein Auto oder ein Flugzeug beschreiben, leer sind. Dieses Konzept wird auch als *Exclusion-Dependency* bezeichnet.

Beide Möglichkeiten besitzen ihre Vor- und Nachteile. Die erste Möglichkeit bildet die Realität zwar besser ab und man kann die objektorientierte Darstellung wiederfinden. Die Objektorientierung verlangt aber, daß bei der Erzeugung eines Autos auch der Fahrzeugteil des Autos erzeugt wird. Das hätte zur Folge, daß auf relationaler Seite für jedes Auto-Objekt das Auto-Tupel mit dem dazu gehörigen Fahrzeug-Tupel verknüpft wird. Die Verknüpfung wird relational auch als *join* bezeichnet. Ein join zwischen zwei Relationen ist in der Laufzeit relativ teuer. Bei großen Systemen ist dieses Laufzeitverhalten nicht mehr tollerierbar.

Bei der zweiten Variante werden die joins vermieden, da alle benötigten Attribute eines Objektes sich in einer Tabelle befinden. Der Nachteil an dieser Lösung ist, daß die so geformten Tabellen sehr groß und damit unübersichtlich werden können.

Wie noch in Kapitel 4.1.1 zu sehen ist, besitzen relationale Datenbanken in ihrer Verwendung Einschränkungen. Hierzu zählt auch die maximale Anzahl an Attributen innerhalb eines relationalen Schemas. Es kann also bei alleiniger Verwendung der zweiten Variante passieren, daß diese maximale Anzahl, die von Datenbank zu Datenbank dazu noch unterschiedlich sein kann, überschritten wird.

Spätestens dann muß auf die erste Variante ausgewichen und ein join in Kauf genommen werden.

Beide Design-Entscheidungen werden durch das in Kapitel 4 vorgestellte *Mapping* und damit auch von dem Generator unterstützt. Für weitere Informationen zu der relationalen Modellierung und der objektorientierten Modellierung wird hier noch einmal auf [UMLb, UMLa, EN94] verwiesen.

2.1.4. Das JDBC-Paket

Als objektorientiertes System bzw. Programmiersprache wurde Java[JAV] eingesetzt. In dieser Sprache wird auch der Code der objektorientierten Zugriffsschicht generiert. Als Datenbank-Zugriffs-Schnittstelle wird *JDBC*[JDB] verwendet, das standardmäßig mit Java ausgeliefert wird. Das JDBC-Paket ist zunächst eine Ansammlung von Verwaltungsobjekten und Datentyp-Definitionen, die allgemein und datenbankunabhängig für den Datenbank-Zugriff verwendbar sind. Zu den Verwaltungsobjekten zählt zum Beispiel die Klasse `java.sql.DriverManager` und als Datentyp sei hier der `java.sql.Blob` genannt. Außerdem stellt das Paket Schnittstellenbeschreibungen für den Resourcenzugriff in Form von Java-Interfaces zur Verfügung, die von einem speziellen Datenbank-Hersteller implementiert werden müssen.

Implementierungen des JDBC-Paketes werden normalerweise direkt vom Datenbank-Hersteller und in Form eines Java-Archivs angeboten. Eine Liste vorhandener Treiber findet man unter [DRI].

Zu den wichtigsten Schnittstellen zählen die in Abbildung 2.6 dargestellten `Driver`, `Connection`, `Statement` und `ResultSet`. Über die vier Interfaces wird der Auf-

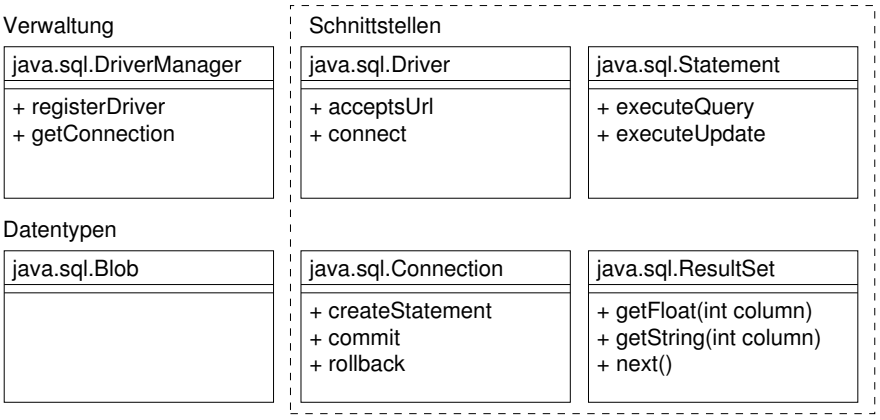


Abbildung 2.6.: Einige JDBC-Komponenten

bau einer Datenbank-Verbindung sowie die Kommunikation mit dem Datenbank-

Server für Datenbank-Anfragen und -Manipulationen definiert. Es gibt noch weitere Interfaces wie das Prepared- oder das Callable-Statement, die die Kommunikationsmöglichkeit zu dem Datenbank-Server erweitern.

Die Diplomarbeit beschränkt sich auf die Vorstellung von JDBC auf Grundlage der vier oben genannten Schnittstellen.

Die vollständige Spezifikation und Dokumentation ist unter [JDB] zu finden.

Kommunikation mit dem Datenbank-Server

Zunächst muß ein Datenbank-Treiber (`Driver`) bei dem Treibermanager (`DriverManager`) über die Methode `registerDriver` angemeldet werden. Dabei wird er in eine Liste beim Treibermanager eingetragen, in der alle dem Manager bekannten Treiber enthalten sind. Wie in Abbildung 2.7 zu sehen ist, sorgt die Methode `getConnection` in der Klasse `DriverManager` für den Verbindungsaufbau zu einem Datenbank-Server. Als Parameter wird ihr unter anderem eine eindeutige Server-Adresse in Form einer *URL*[3] übergeben, an der sich der Datenbank-Server befinden muß. Die übergebene URL hat ein spezielles Format, in dem die Datenbank und damit der für den Verbindungsaufbau notwendige Treiber verschlüsselt ist. Der Treibermanager läßt von jedem in seiner Liste befindlichen Treiber die URL mittels `acceptsURL` überprüfen, ob die URL für den Treiber bestimmt ist oder nicht. Es wird mit dem Treiber, der die angegebene URL als erstes akzeptiert, mittels der Nachricht `connect` eine Verbindung aufgebaut. Bei erfolgreicher Anmeldung erhält man ein `Connection`-Objekt, mit dessen Hilfe mit dem Datenbank-Server kommuniziert werden kann.

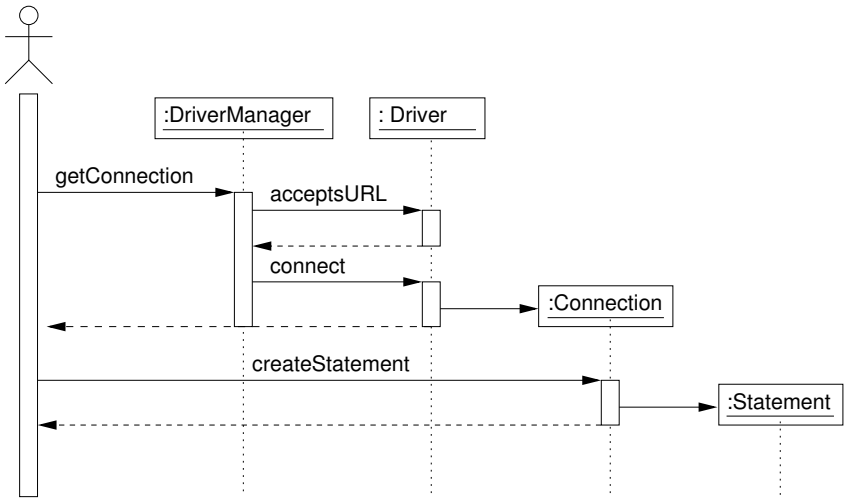


Abbildung 2.7.: UML Sequenzdiagramm zur Erzeugung eines Statements

[3] URL: Unified Resource Locator

In der JDBC-Spezifikation wird empfohlen, die oben erwähnte URL in dem speziellen Format

jdbc:¡Datenbank-Name¿:¡Protokoll-Art¿:

beginnen zu lassen. Das ist aber jedem Treiber-Hersteller selbst überlassen, und das Format wird auch nicht von dem `DriverManager` kontrolliert.

Um einen Datenbankbefehl jeglicher Form absetzen zu können, benötigt man ein `Statement`-Objekt, das mit dem Aufruf der Methode `createStatement` auf dem `Connection`-Objekt erzeugt wird. Bei der Ausführung von Datenbank-Befehlen unterscheidet die `Statement`-Schnittstelle zwischen Datenbank-Anfragen (lesend) und -Manipulationen (schreibend). Für die erste Anforderung steht die Methode `executeQuery` zur Verfügung und für die zweite `executeUpdate`.

Die Ergebnisse bei einer lesenden Anfrage werden in Form eines `ResultSets` zurückgegeben. Das `ResultSet` umfaßt alle Datensätze der gestellten Anfrage und seine Schnittstelle definiert unter anderem die Navigationsmöglichkeiten innerhalb dieser Menge. Man kann zum Beispiel auf den “ersten” bzw. “letzten” Datensatz (first bzw. last) der Menge des `ResultSets` springen oder von dem gerade aktuellen Satz auf den “nächsten” (next). Für das Auslesen der Datensatzinhalte bietet die Schnittstelle des `ResultSets` Zugriffsmethoden wie `getInt` für numerische oder `getString` für Texte an. Für komplexere Datenmengen gibt es weitere Zugriffsmethoden wie `getBlob` bzw. `getBinaryStream` für Binärdaten und `getClob` bzw. `getInputStream` für Textdaten.

2.2. Transaktionen

Unter einer *Transaktion* T versteht man zunächst eine Folge von Aktionen, die mit einer *abort*- oder *commit*-Aktion abgeschlossen wird. Bei einer Datenbank-Transaktion kann die Menge an Aktionen auf

$$A = \{\, read,\, write,\, abort,\, commit \,\}$$

festgelegt werden. Im folgenden werden die einzelnen Aktionen mit ihren Anfangsbuchstaben abgekürzt. Die *read*- und die *write*-Aktionen erhalten als zusätzlichen Parameter das Objekt, auf welches sie sich beziehen. Eine Beispieltransaktion sieht wie folgt aus:

$$T = r(X)\, r(Y)\, w(X)\, c$$

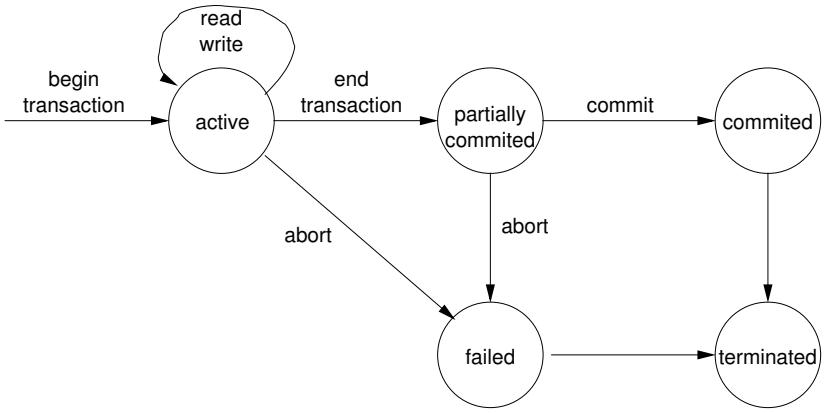


Abbildung 2.8.: Die Transaktion als Automat

Diese Transaktion liest zunächst die Daten von X und Y, schreibt dann X und führt anschließend *commit* aus.

Die Abbildung 2.8 stellt die Transaktion als Automat dar. Nach Erhalt der Nachricht *begin transaction* befindet sich die Transaktion im Zustand `active`. Sie verweilt in diesem Zustand solange sie *read*- bzw. *write*-Nachrichten erhält. Bei Erhalt einer *abort*-Nachricht wechselt die Transaktion in den Zustand `failed`, und die Transaktion wird abgebrochen. Alle in der Transaktion gemachten Schreibaktionen werden widerrufen. Die Transaktion wird wohldefiniert beendet, indem Sie nach dem Zurücksetzen aller gemachten Änderungen in den Zustand `terminated` übergeht.

Wenn sich die Transaktion im Zustand `active` befindet und sie als abschließende Aktion eine *end transaction*-Nachricht erhält, wechselt sie zunächst in den Zwischenzustand `partially commited`. In diesem Zustand überprüft das Datenbank-Managementsystem die von der Transaktion ausgeführten Datenänderungen auf *Konsistenz*. Bei einem *Konsistenz*-Verstoß wird die Transaktion mittels einem `abort` wiederum verworfen. Im anderen Fall wird die Transaktion erfolgreich abgeschlossen. In beiden Fällen terminiert die Transaktion.

2.2.1. Konsistenz

Der Zustand einer Datenbank umfaßt alle relationalen Schemata in ihr und deren Tupel bzw. Ausprägungen. Der Zustand einer Datenbank kann durch die Grundoperationen

- Einfügen (insert)
- Verändern (modify)

- Löschen (delete)

geändert werden. Die Datenbank selbst darf nur erlaubte Zustände (valid states) aufnehmen bzw. besitzen. Ein erlaubter Zustand wird durch das Datenmodell der Datenbank definiert. Die Einhaltung des Datenmodells ist die Aufgabe des Datenbank-Managementsystems. Es tritt ein unerlaubter Zustand auf, sobald ein Tupel eines beliebigen relationalen Schemas aus der Datenbank gegen die Gültigkeitsregeln im Datenmodell verstößt.

Ein erlaubter Zustand wird als konsistenter Zustand bezeichnet. Dadurch, daß kein Tupel gegen die Gültigkeits- bzw. *Konsistenzregeln* in einem konsistenten Zustand verstößt, kann jedes einzelne Tupel selbst wieder als konsistent bezeichnet werden.

2.2.2. ACID-Eigenschaften

Transaktionen auf Datenbanken haben spezielle Eigenschaften, die als *ACID - Eigenschaften* bezeichnet werden. ACID steht für:

- Atomicity
  Die Transaktion wird entweder vollständig oder überhaupt nicht ausgeführt.
- Consistency preservation
  Von einem konsistenten Zustand führt die Transaktion in einen anderen konsistenten Zustand.
- Isolation
  Geänderte Daten in noch laufenden Transaktionen sind von anderen Transaktionen nicht erreichbar. Solange eine Transaktion nicht commited wurde, sind ihre Änderungen nach außen nicht sichtbar.
- Durability bzw. permanency
  Wenn eine Transaktion erfolgreich beendet wurde, dürfen ihre Änderungen durch einen später auftretenden Fehler nicht verloren gehen.

Diese Eigenschaften haben Auswirkungen auf den Code, der generiert werden soll. Eine Zugriffsmethode eines nicht transaktionsfähigen Objektes kann zu jeder Zeit ausgeführt werden, und der geänderte Zustand eines Objektes wird für die Außenwelt sofort sichtbar. Ein normales Objekt besitzt auch nicht die Eigenschaften des persistenten Abspeicherns. Das von der Generierung verwirklichte Transaktionskonzept, das die oben genannten Punkte beachtet, wird im Kapitel 5.3.1 vorgestellt. Das von der objektorientierten Zugriffsschicht verwendete Transaktionskonzept ersetzt nicht das der Datenbank; es wirkt sich nur auf die Zugriffsschicht aus. Zum tatsächlichen Abspeichern wird das durch das JDBC-Paket gekapselte Konzept der Datenbank genommen.

2.2.3. Schedules

In Datenbanksystemen ist es in der Praxis gerade aus Geschwindigkeitsgründen erforderlich, Transaktionen parallel ausführen zu können. Die Transaktionen

$$T_1 = r_1(X)\, r_1(Y)\, w_1(X)\, c_1$$
$$T_2 = r_2(X)\, w_2(X)\, c_2$$

sollen im folgenden gleichzeitig ausgeführt werden. Dabei gibt der Index an der jeweiligen Aktion an, zu welcher Transaktion sie gehört. Die Ablaufreihenfolge der einzelnen Aktionen wird als *Schedule* bezeichnet. Ein mögliches Schedule der beiden oben angegebenen Transaktionen könnte wie folgt sein:

$$S = r_1(X)\, r_2(X)\, w_2(X)\, c_2\, r_1(Y)\, w_1(X)\, c_1$$

Unter einem *vollständigen Schedule* versteht man die Folge von Aktionen mehrerer Transaktionen, wobei die Aktionen einer Transaktion in der gleichen Reihenfolge auftreten müssen wie in der Transaktion selbst. Desweiteren müssen alle Aktionen einer Transaktion in dem Schedule enthalten sein. Es dürfen also keine prinzipiell äquivalenten Aktionen aus unterschiedlichen Transaktionen zu einer zusammengfaßt werden.

Zwei Aktionen stehen in *Konflikt*, wenn zumindest eine davon eine *write*-Aktion ist und sich beide auf das selbe Objekt beziehen. Man nennt ein Schedule *seriell*, wenn alle Aktionen einer Transaktion ohne Unterbrechung durch Aktionen anderer Transaktionen ausgeführt werden. *Rücksetzbar* ist ein Schedule, wenn jede Transaktion erst dann *commit* ausführt, wenn alle Transaktionen, von denen sie gelesen hat, *commit* ausgeführt haben.

Das Transaktionskonzept, daß die generierte Zugriffsschicht verfolgt, wird in Kapitel 5.3.1 noch näher erläutert. Für eine genauere Definition des Transaktionsbegriffes wird auf [EN94] verwiesen.

2.3. Konzepte bei der Gestaltung von Datenbank-Systemen

Bei dem Zugriff auf einen Datenbank-Server stellt sich die Frage, wie man das am geschicktesten lösen kann. Da der Datenbank-Begriff an sich schon sehr alt ist und in der Software-Entwicklung eine bedeutende Rolle einnimmt, sind im Laufe der Zeit Techniken entwickelt worden, die helfen sollen, den Zugriff auf den Datenbank-Server zu vereinfachen bzw. zu steuern. In der Zwischenzeit haben

sich diese Techniken bzw. Architekturen als richtig bzw. gut erwiesen oder aber auch als falsch.

Um dem in dieser Diplomarbeit vorgestellten Generator den nötigen Rahmen zu geben, soll in diesem Kapitel eine Auswahl der bekanntesten Konzepte vorgestellt werden. Die Auswahl erhebt nicht den Anspruch auf Vollständigkeit. Die hier vorgestellten Architekturen gelten heute auch als Design-Pattern bzw. Anti-Pattern.

2.3.1. Layered Architektur

Bei der Erstellung einer Applikation muß mit einer langen Lebenszeit derselben gerechnet werden. In dieser Zeit erlebt die Applikation ihre eigene Evolution. Sie wird ständig um neue Funktionen erweitert, und aufkommende neue Technologien müssen so schnell wie möglich in die Applikation integriert werden. Eine Änderung sollte bzw. muß sich so lokal wie nur möglich auf das System auswirken, um Kosten zu sparen. Um das zu gewährleisten, versucht man, die Applikation in Schichten (Layer) zu partitionieren.

Jede Schicht übernimmt eine bestimmte Aufgabe, und die Schichten bilden eine Hierarchie. Dabei gilt, daß eine Schicht nur von den unter ihr liegenden Schichten abhängt. Die in Abbildung 2.9 gezeigte "*Standard 4 Layer Architektur*" besteht

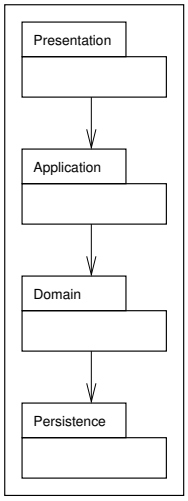


Abbildung 2.9.: Standard 4 Layer Architektur

aus vier Schichten.

Die *Presentation*-Schicht definiert das User-Interface der Applikation. Eine Änderung an ihr wirkt sich nur auf die Präsentation aus. Sie verändert nicht die Logik der Applikation.

Die *Domain*-Schicht definiert die Logik der Applikation.

Die *Application*-Schicht ist eine Mapper-Schicht, mit deren Hilfe die Logik in der Presentation-Schicht dargestellt wird. Dadurch ist die Logik unabhängig von der Darstellung. Die bekannte *Model-View-Controller* Architektur richtet sich nach diesem Layer-Prinzip.

Die *Persistence*-Schicht ist für den Datenbankzugriff verantwortlich. Eine an ihr vorgenommene Änderung wird sich sehr wahrscheinlich auf die gesamte Applikation auswirken, da die Schicht die Basis der Applikation bildet und in den anderen Schichten benutzt wird.

Die Partitionierung der Applikation versucht man so vorzunehmen, daß die meisten Änderungen so weit oben wie möglich durchgeführt werden. So wirken sie sich am wenigsten auf das Gesamtsystem aus.

2.3.2. Two Tier Architektur

Die nachfolgenden Architekturen bzw. Design-Patterns beziehen sich auf die speziellen Umgebungsbedingungen, die bei dem Zugriff auf eine Datenbank vorliegen. In der Regel liegt die Datenbank auf einem eigenen Rechner (Host), und es wird von anderen Rechnern darauf zugegriffen. Der Datenbank-Rechner wird als *DBMS Host* und die zugreifenden Rechner werden als *Clients* bezeichnet. Die Clients greifen über einen *Server* auf den DBMS Host zu. Man spricht von einer *Client/Server-Umgebung*. Die in Abbildung 2.10 gezeigte *Two Tier Architektur* wird auch als Fat Client bezeichnet, was erahnen läßt, daß es sich hierbei um ein Negativ-Beispiel (Anti-Pattern) in der Datenbankentwicklung handelt. Diese

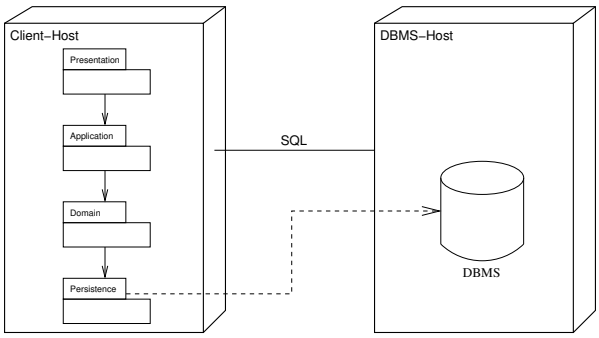


Abbildung 2.10.: Die Two-Tier-Architektur

Architektur zeichnet sich dadurch aus, daß der Client aus allen zuvor vorgestellten Partitionen besteht, und seine Persistenz-Schicht über das Netzwerk auf den Datenbank-Server zugreift. Da der Client die Daten der Datenbank in seiner Persistenz-Schicht selbst verwalten muß, muß er auch über die nötigen Speicher-Resourcen verfügen. Da es sich in der Regel um mehrere Clients handelt, muß

jeder Client diese Resourcen zur Verfügung stellen. Bei einem sehr großen Datenaufkommen, das über das Netzwerk ausgetauscht wird, entsteht eine unnötige Resourcenverschwendung auf Seite des Clients und ein unnötig hoher Netzwerkverkehr zwischen den Clients und dem Server, da jeder Client individuell seine Daten abfragt.

2.3.3. Three Tier Architektur

Bei der *Three Tier Architektur* schaltet man einen weiteren Rechner zwischen den *DBMS Host* und die Clients, der die Serveraufgaben bei dem Datenbankzugriff übernimmt. Er ist der einzige Rechner, der auf den DBMS Host zugreift.

Alle Clients stellen ihre Anfragen und Anforderungen an den Server. Dadurch, daß die Clients nicht mehr direkt auf den DBMS-Host zugreifen, benötigen sie nicht mehr das Wissen der Persistenz-Schicht. Die Persistenz-Schicht muß nur noch auf dem Server gehalten werden. Die Abbildung 2.11 zeigt die *Three Tier Distributed Object Architektur.*

Hier ist neben der Persistenz-Schicht auch die Domain-Schicht der Applikation auf den *Applikations-Server* ausgelagert worden. Die Applikations-Schichten der Clients sind dahingehend angepaßt worden, daß sie nicht mehr direkt die Domain-Schicht ansprechen, sondern *Services* auf Seiten des Servers aufrufen. Die Clients

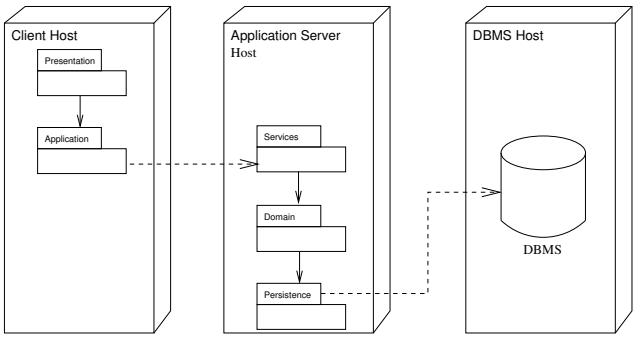


Abbildung 2.11.: Three-Tier Distributed Object Architektur

erhalten so nur noch die Ergebnisse, die sie auch wirklich haben wollen.

Würde die Domain-Schicht nicht ausgelagert werden, müßte *jeder* Client von einer Änderung informiert werden, weil sonst seine Logik inkonsistent zum tatsächlichen Datenbestand werden würde. Er erhielte Informationen, auf die er gar nicht gewartet hat bzw. die er gar nicht wissen wollte. Diese Lösung würde also unnötig das Netzwerk belasten. Durch die Auslagerung der Domain- und der Persistenz-Schicht können auf Seiten der Clients Resourcen gespart werden. Durch dieses ,,PUSH-Prinzip" kann das Netzwerk entlastet werden.

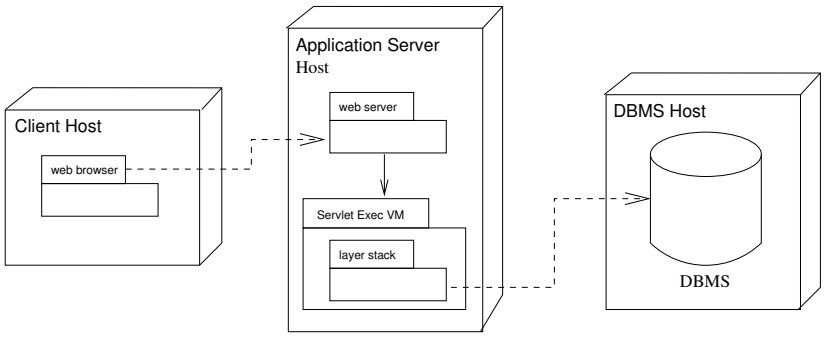


Abbildung 2.12.: Three-Tier Undistributed Object Architecture

Die in Abbildung 2.12 gezeigte *Three Tier Undistributed Object Architecture* ist die Architektur, die heutzutage bei Internet-Applikationen angewendet wird. Auf dem Applikations-Server läuft ein Internet-Server, der von den Clients mittels eines herkömmlichen Internet-Browsers angesprochen wird. Für moderne Programmiersprachen wie Java werden Schnittstellen wie die *Java Server Pages API* angeboten, die die Kommunikation mit dem Internet-Server definieren. Die Schnittstelle kapselt die Logik der Applikation und leitet Nachrichten, die sie vom Internet-Server erhält, an die Logik weiter.

2.3.4. Folgerung

Dadurch, daß nur noch der Server auf den DBMS Host zugreift, könnte er die enthaltenen Daten *cachen*. Dabei versteht man unter einem *Cache*, daß eine Kopie der eigentlichen Daten angelegt und diese Kopie bei Verlangen zurückgegeben wird. Die Kopie der Daten wird in der Regel bei dem ersten Zugriff auf die Daten erstellt.

Dieses Prinzip wendet man an, wenn der Zugriff auf die echten Daten teuer in der Zeit ist. Es ist schneller, die Daten aus seinem Hauptspeicher zu ermitteln, als sie über ein Netzwerk anzufragen. Der Nachteil dieses Prinzips ist der Mehraufwand bei einer Änderung der Daten. Der Cache ist für den Benutzer transparent. Der Benutzer geht bei einem Änderungswunsch der Daten von den echten Daten aus. Der Server muß hier die Echtdaten zum einen und seinen Cache zum anderen aktualisieren. Die Struktur des Servers wird dadurch komplexer.

Die vorgestellten Architekturen für die Gestaltung einer Applikation mit Datenbankzugriff zeigen, daß die Persistenz-Schicht zu der Basis einer Applikation bzw. eines Systems gehört. Sie befindet sich ganz tief in der Logik der Applikation.

Der Generator orientiert sich an diesen Techniken, indem er eine Zugriffsschicht generiert, die auf der Layer-Technik beruht. Er generiert also die Persistenz-Schicht, die unabhängig von der darüber befindlichen Domain-Schicht ist.

Da die vorgestellten Architekturen alle spezielle Schnittstellen für ihre Anforderungen definieren, die von der Logik, also der Domain, abhängen, ist die persistente Zugriffsschicht unabhängig von der später eingesetzten Architektur. Es kann also nach der Generierung der Zugriffsschicht noch entschieden werden, ob die Applikation zum Beispiel auf der Three Tier Distributed Object Architektur basiert oder aber eine Internet-Applikation ist.

3. Verwandte Arbeiten und Konzepte

Im Bezug auf Datenbanken gibt es sehr viele Projekte und Produkte, die in dieser Arbeit nicht alle aufgezählt werden können. Gerade im Bezug auf die Schnittstelle der hier vorgestellten objektorientierten Zugriffsschicht sind bereits sehr mächtige Spezifikationen entwickelt worden, die das Verhalten persistenter Objekte mit einem Transaktionskonzept definieren. Zu diesen Spezifikationen zählen zum Beispiel die *Enterprise Java Beans* (EJB) und die *Java Transaction Service* (JTS) Spezifikation, die im *J2EE SDK [J2E]* enthalten sind. Eine weitere in diesem Zusammenhang zu nennende Spezifikation ist die *Java Data Objects* (JDO) Spezifikation, die inzwischen eine offizielle Schnittstellenbeschreibung des *ODMG-Standards* [ODM] für Java ist.

Der ODMG-Standard definiert den Zugriff auf objektorientierte Datenbanken. Da hier eine objektorientierte Zugriffsschicht auf eine relationale Datenbank vorgestellt wird, liegt die Idee nahe, sich bei der Schnittstellendefinition an den vorgestellten Spezifikationen, und hier gerade an JDO, zu orientieren. Durch den Einsatz von JDBC ist aber die objektorientierte Zugriffsschicht auch auf die Leistungsfähigkeit von JDBC beschränkt.

Der ODMG-Standard definiert unter anderem auch das Ausführen von Transaktionen über mehrere Datenbanken. Dabei wird die große Transaktion in kleinere, nicht datenbankübergreifende Transaktionen für jede Datenbank aufgeteilt. Jede an der Transaktion beteiligte Datenbank muß gewährleisten, die auf ihr ausgeführten Transaktionen so lange in dem `partially commited` Zustand aus der Abbildung 2.8 zu belassen, bis alle Teiltransaktionen korrekt ausgeführt werden konnten. Falls eine Transaktion auf einer Datenbank fehlschlägt, müssen die anderen Datenbanken durch die Definition der Transaktion noch in der Lage sein, ihre korrekt durchgeführten Transaktionen zu verwerfen. Das Halten der Transaktion in dem `partially commited` Zustand und der von außen kommenden Bestätigung des commit wird als *Two-Phase-Commit* bezeichnet. Das JDBC-Paket bietet keine Schnittstelle an, ein Two-Phase-Commit zu steuern. Eine daraus folgende Nachimplementierung ist für diese Diplomarbeit zu aufwendig.

Ein weiterer Bereich, in dem immer wieder neue Lösungen in der Datenbank-

Entwicklung angeboten werden, ist der der verschiedenen auf dem Markt erhältlichen *IDEs*[1] wie der *Borland JBuilder* [Bor]. Im Gegensatz zu den Case-Tools steht bei dieser Art der Programm-Entwicklung nicht die interne Logik oder das System im Vordergrund, sondern die graphische Seite, also die der Applikation.

Die IDEs sind normalerweise mit einem sehr mächtigen Werkzeug ausgerüstet, mit dem man sehr komfortabel graphische Darstellungen erstellen kann. Das Werkzeug wird im allgemeinen auch *GUI-Builder*[2] genannt. Auch Datenbankabfragen und deren Ergebnisse können so sehr komfortabel und bequem erstellt werden. Mit einigen wie dem oben erwähnten JBuilder können auch neue relationale Schemata erzeugt werden. Um die Integrierung des neuen Schemas in die Logik des Systems muß sich der Entwickler aber selber kümmern.

Im Gegensatz dazu erstellt der Generator von REDDMOM eine Datenbank-Zugriffsschicht auf Logik-Seite. Sicherlich interessant ist die Vorstellung, die Logik so aufzubauen, daß sie zum Beispiel graphisch im JBuilder verwendet werden kann. Dies ist aber nicht Gegenstand dieser Arbeit.

Zu dieser Arbeit verwandte Projekte sind zum Beispiel ObjectDRIVER[OBJ], ONTOS[ONT], JDX[JDX] und sicherlich viele andere mehr. In dem Kapitel 3.1 soll aus diesen Projekten ObjectDRIVER näher vorgestellt werden.

Wie bereits in Kapitel 1.2 erwähnt, dient das Projekt VARLET als Grundlage dieser Diplomarbeit. In Kapitel 3.2 wird erklärt, warum man sich gegen eine Weiterführung von VARLET entschieden hat und die Algorithmen aus dem Projekt lieber über eine Neuimplementierung in das Projekt REDDMOM übernimmt.

3.1. ObjectDRIVER

ObjectDRIVER [OBJ] ist zu dem oben erwähnten ODMG-Standard kompatibel. Der ODMG-Standard definiert in seiner Spezifikation eine Abfragesprache mit dem Namen OQL[3]. Relationale Datenbanken verwenden gewöhnlich die Abfragesprache SQL. Beide Sprachen ähneln sich zwar, besitzen aber in ihrer Grammatik Unterschiede. ObjectDRIVER bietet auf objektorientierter Seite als Abfragesprache OQL an. Diese OQL-Statements werden für den Benutzer transparent in SQL-Statements umgeformt und an die Datenbank weitergeleitet. Dadurch entsteht bei der Benutzung der Zugriffsschicht durchgängig der Eindruck, daß man auf einer objektorientierten Datenbank arbeitet.

In der Abbildung 3.1 wird ersichtlich, wie ObjectDRIVER funktioniert. Alles über dem ObjectDRIVER findet objektorientiert statt. Anfragen an die Daten-

[1] IDE: Integrated Development Environment
[2] GUI: Graphical User Interface
[3] OQL: Object Query Language

bank werden in OQL gestellt. Über das später noch näher vorgestellte `Correspondence/Mapping`-Schema kann er auf die relationale Datenbank zugreifen. Aus der Abbildung wird ersichtlich, daß er auch den Zugriff auf datenbankfremde Resourcen wie XML- und HTML-Dateien ermöglicht.

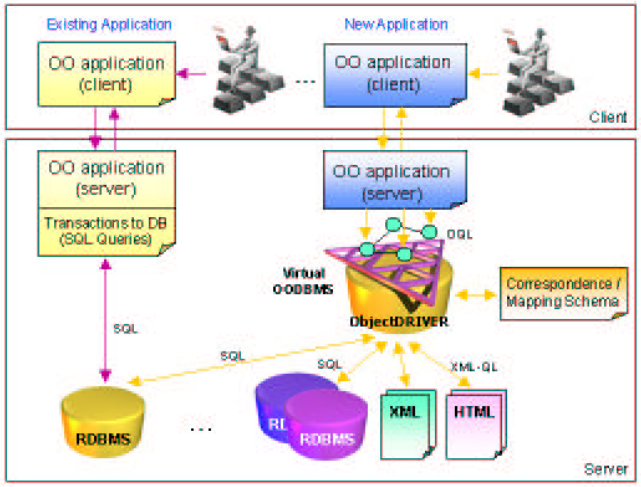


Abbildung 3.1.: ObjectDRIVER Architektur[OBJ]

Beim Einsatz des ObjectDRIVER wird davon ausgegangen, daß das Schema der relationalen Datenbank sowie die objektorientierte Zugriffsschicht fertig erstellt wurden. Die Tabellenstruktur ist auf der Datenbank vorhanden, und es existieren compilierfähige Klassen in einer Programmiersprache wie Java oder C++, die die Logik der Zugriffsschicht definieren, nicht aber die Persistenz. Die Persistenz wird durch einen zweiten Kompiliervorgang mit Hilfe von ObjectDRIVER erzeugt. Dadurch soll die Persistenz für den Benutzer der Objekte transparent gemacht werden. Um ObjectDRIVER einzusetzen, muß ihm in Konfigurationsdateien mitgeteilt werden, wie die beiden Seiten aussehen und was von der objektorientierten Seite der relationalen Seite entspricht. Man baut so manuell ein sogenanntes Mapping auf. Die, um genauer zu sein, zwei Konfigurationsdateien sind reine Textdateien und können mit einem normalen Text-Editor erstellt werden. Die über die Dateien erstellte Konfiguration wird von ObjectDRIVER als "*Definition eines Korrespondenz Schemas*" bezeichnet. Die erste Datei ist zuständig für die Deklaration des relationalen Schemas und wird von ObjectDRIVER als Schema-Deklaration benannt. Die zweite definiert die objektorientierte Zugriffsschicht. Von ObjectDRIVER werden die dort definierten Klassen als *virtuelle Objekt-Klassen* bezeichnet. Außerdem wird in der zweiten Datei die oben erwähnte Korrespondenz zwischen den beiden Welten definiert. Im folgenden werden die beiden Dateien näher beschrieben.

3.1.1. Schema-Deklaration

In der Schema-Deklaration wird das relationale Schema beschrieben. Dabei folgt ObjectDRIVER seiner eigenen Grammatik, die hier kurz und nur auszugsweise vorgestellt wird. Eine vollständige Dokumentation ist unter [OBJ] erhältlich.

Eine Konfigurationsdatei für die Schema-Deklaration hat den folgenden Aufbau:

1. Name des Schemas
2. Verweis auf das zugrundeliegende DBMS
3. Tabellen-Deklaration des Schemas

Dabei sind der Name des Schemas und der Verweis auf das DBMS Schlüsselworte, die der Benutzer in einer globalen Einstellungsdatei des ObjectDRIVERS angegeben hat. In dieser globalen Einstellungsdatei wird den Schlüsselworten die tatsächliche Datenbank zugewiesen. Die dort angegebene Datenbank kann für den notwendigen Verbindungsaufbau und für die spätere Kommunikation spezielle Einstellungen verlangen, die auch in der globalen Einstellungsdatei definiert werden. Dadurch, daß man diese Einstellungen nur einmal konfigurieren muß, kann man die Schema-Deklaration auf das wesentliche beschränken. Sie wird übersichtlicher. Ein Ausschnitt aus einer solchen Schema-Deklaration könnte wie folgt aussehen:

```
define schema Company {
  relationalDbms Oracle;
  define table EMP {
     EMPNO        long          keyPart,
     ENAME        string(20)    notNull,
     FNAME        string(20),
     JOB          string(20),
     MGR          long,
     SAL          double,
     COM          double,
     DEPTN        long,
     primary Key(EMPNO),
  };
  define table PERSON {
     PNUM         long          keyPart,
     PNAME        string(20),
     FNAME        string(20),
     SSNUM        string(15)    unique,
     POSITION     string(10),
     PHONE        string(12),
```

```
      CAR           long
  };
...
};
```

In der Beispiel-Deklaration wird ein Schema `Company` definiert, das auf einem DBMS mit dem Namen `Oracle` zu finden ist. `Company` und `Oracle` sind die Schlüsselworte, die der Benutzer in der globalen Einstellungsdatei angegeben hat. Dort hat er auch die spezifischen Einstellungen für das DBMS `Oracle` vorgenommen. Desweiteren werden in der Schema-Deklaration die Tabellen `EMP` und `PERSON` definiert. Die Tabelle `EMP` hat die Attribute `EMPNO`, `ENAME`, `FNAME`, `JOB`, `MGR`, `SAL`, `COM` und `DEPTN`. Dabei ist das Attribut `EMPNO` der Primärschlüssel dieser Tabelle. Das Schlüsselwort `notNull` gibt an, daß das Feld `ENAME` immer einen wohldefinierten Wert besitzen muß. Das Schlüsselwort `unique` sorgt dafür, daß ein beliebiger Wert der Spalte `SSNUM` in dem Schema `PERSON` tupelweit nur ein einziges Mal vergeben werden darf. Das Schema PERSON wird bei der später vorgestellten Korrespondenz-Definition benötigt. Sie soll hier nicht weiter vorgestellt werden.

Allgemein ist festzustellen, daß sich die verwendete Sprache in dieser Konfigurationsdatei an den SQL-Standard anlehnt. ObjectDRIVER geht davon aus, daß der Ersteller des Deklaration-Schemas Datenbank-Erfahrungen hat und über SQL-Wissen verfügt. Durch die Anlehnung an den SQL-Standard ist eine schnelle Einarbeitung in diese eher als SQL-Dialekt zu bezeichnende Sprache gewährleistet.

3.1.2. Klassen-Deklaration

Auch bei der Klassen-Deklaration verwendet ObjectDriver eine Sprache, die für einen Benutzer, der bereits Erfahrung mit objektorientierten Sprachen oder mit der objektorientierten Modellierung gesammelt hat, einfach zu erlernen ist. Nachstehend ist eine solche Konfiguration angegeben:

```
class Employee on EMP
type Tuple {
  name            String,
  firstname       String,
  socialnumber    String,
  manager         Employee
}
end;
...
class Salesman inherit Employee type Tuple {
  jobdesc         String,
```

```
   position        String,
   commission      double,
   income          double
   ...
}
end;
```

Die Definition der Klasse wird mit `class` eingeleitet. Den Klassen wird mit dem Befehlswort `type` ein Klassentyp zugeordnet. In der ObjectDRIVER-Dokumentation wird der Typ `Tuple` wie folgt beschrieben:

> The semantics of the Tuple type is an aggregate of elements. Its semantics is similar to the struct construct of the C++ language.

Wie bereits erwähnt, liegt die Logik, die später nach Einsatz des ObjectDRIVER persistent sein soll, in Form von Java- bzw. C++-Klassen vor. In der hier vorgestellten Konfigurationsdatei muß die Struktur der Klassen nachgebildet werden. ObjectDRIVER interessiert nur die Struktur. Um eine Klasse persistent zu machen, wird die Schnittstellenbeschreibung der Klasse nicht benötigt. Der Typ `Tuple` definiert also eine aus dem Objekt-Modell herkömmliche Klasse, ohne die Schnittstellenbeschreibung mit anzugeben. Neben diesem Typ gibt es noch den Typ `Set`. Er wurde in der Dokumentation in diesem Zusammenhang nur erwähnt und in den angegebenen Beispielen, wie eine Konfigurationsdefinition auszusehen hat, nie benutzt. Deswegen soll er hier nicht weiter beachtet werden.

Die Definition der Attribute der Klasse erfolgt in etwa analog zu der von herkömmlichen Programmiersprachen wie Java.

Da ObjectDRIVER ODMG-konform ist, unterstützt er die von dem ODMG-Standard verlangten *Standard-Typen* und *Collection-Typen*. Zu der ersten Kategorie zählen zum Beispiel `float` und `double` sowie `String` aber auch `Date`, `Time` oder `Timestamp`. Zu den Collection-Typen zählen die von dem ODMG-Standard definierten Typen `Array`, `Bag`, `Dictionary`, `Set` oder auch `List`. Diese Datentypen definieren alle Mengen über Objekte. Sie unterscheiden sich vorwiegend in ihrem Mengenverhalten. Zum Beispiel sind `Set` und `Bag` beides Mengen ohne Ordnung. Ihre Elemente liegen unsortiert vor. Während die Menge `Set` keine Duplikate erlaubt, ist es bei dem `Bag` möglich, in ihm mehrmals das selbe Element zu speichern.

Neben diesen Datentypen unterstützt ObjectDRIVER die aus dem Objekt- Modell bekannten komplexen Datentypen. Es kann in der Klassendeklaration von ObjectDRIVER eine Referenz zu einem anderen Objekt angegeben werden.

3.1.3. Korrespondenz

Bisher wurde aus Gründen der Übersicht die noch fehlende Korrespondenz-Definition in der Klassendeklaration weggelassen. Neben der Strukturbeschreibung der Klasse muß eine Zuordnung auf die zugrundeliegende relationale Tabelle, die in der Schema-Deklaration definiert wurde, erfolgen. Die in Kapitel 3.1.2 angegebene Klassen-Deklaration kann wie folgt erweitert werden:

```
class Employee on EMP
type Tupel {
   name             String     on EMP.ENAME,
   firstname        String     on EMP.FNAME,
   socialnumber     String     on PERSON.SSNUM,
   manager          Employee   on EMP.MGR
   ...
   join EMP to PERSON by       (((EMP.ENAME == PERSON.ENAME) &&
                               (EMP.FNAME == PERSON.FNAME)))
}
end;
...
class Salesman inherit Employee type Tupel {
   jobdesc          String     constrained by (EMP.JOB == ‘‘Salesman’’),
   position         String     on PERSON.POSITION,
   commission       double     on EMP.COM,
   income           double     calculatedBy(EMP.SAL + EMP.COM)
   ...
}
end;
```

Dabei wird der Klasse `Employee` die relationale Tabelle `EMP` zugeordnet. Da die Klasse `Salesman` von der Klasse `Employee` erbt und keine weiteren Angaben über eine womöglich andere Tabelle als `EMP` gemacht wurden, ist der Klasse `Salesman` ebenfalls auf relationaler Seite die Tabelle `EMP` zugeordnet. `EMP` wird als Haupttabelle von der Klasse `Employee` bzw. `Salesman` angesehen.

Bei der Korrespondenz-Definition zu den Attributen der Klasse muß zwischen den *atomaren* und den *komplexen* Attributen unterschieden werden.

Atomare Attribute der Klasse haben entsprechende Attribute auf der relationalen Seite, die dasselbe aussagen. Die Attribute `name`, `firstname` und `socialnumber` der Klasse `Employee` sind solche atomaren Attribute, da sie auf relationaler Seite die korrespondierenden Attribute `EMP.ENAME`, `EMP.FNAME` und `PERSON.SSNUM` besitzen. Da sich das Attribut `SSNUM` nicht in der Tabelle `EMP` befindet, muß ObjectDRIVER mitgeteilt werden, wie er an das Attribut herankommt. Dafür ist die in der Klassendefinition von Employee enthaltene `join`-Angabe zuständig.

Bei dem Zugriff auf ein `Employee`-Objekt führt ObjectDRIVER intern den oben angegebenen `join`-Befehl aus, um die Werte aus der Tabelle `PERSON` der Klasse `Employee` zur Verfügung zu stellen.

Wie aus der Definition des Objekt-Modells zu entnehmen ist, ist ein komplexes Attribut eine Referenz auf ein anderes Objekt. Referenzen zwischen zwei persistenten Objekten spiegeln eine Fremdschlüssel-Beziehung auf relationaler Seite zwischen zwei Tabellen wider. Das Attribut `manager` in der Klasse `Employee` ist eine solche Referenz.

Der Unterschied zwischen der Deklaration eines atomaren Attributs und der eines komplexen Attributs wird kaum ersichtlich. Da hier eine Fremdschlüssel-Beziehung zwischen zwei relationalen Tabellen definiert wird, hätte man eine solche auch in der Spezifikationsdarstellung erwartet. In der hier gewählten Darstellung sieht diese Definition aber wie eine gewöhnliche atomare Attribut-Definition aus. Die Auflösung liegt in der internen Behandlung des `manager` Attributs von ObjectDRIVER. ObjectDRIVER ist bekannt, daß der Klasse `Employee` die Tabelle `EMP` als Haupttabelle zugeordnet ist. `EMP` hat als primären Schlüssel das Attribut `EMPNO`. Da der Datentyp des Attributes `manager` auch die Klasse `Employee` ist, weiß ObjectDRIVER, daß es sich hier um eine Referenz auf eine persistente Klasse handelt.

Es handelt sich also auf relationaler Seite um eine Fremdschlüssel-Beziehung. Fremdschlüssel-Beziehungen können nur in eine Richtung, und zwar vom Fremdschlüssel zum Primärschlüssel, definiert werden. Somit weiß ObjectDRIVER bei der Angabe der Klasse `Employee` als Datentyp, daß hier auf relationaler Seite der Primärschlüssel der Tabelle `EMP` gemeint ist. Abgeschlossen wird die Definition des komplexen Attributs `manager` durch die Angabe des Fremdschlüssels `EMP.MGR`.

Durch das obige Verhalten bei der Definition von komplexen Attributen wird der Primärschlüssel für die Klasse prinzipiell überflüssig, da er intern verwaltet wird. ObjectDRIVER bietet die Möglichkeit an, den Primärschlüssel in die objektorientierte Zugriffsschicht gar nicht zu übernehmen. Das ist gerade bei den in Kapitel 2.1 erwähnten autoincrement-Werten praktisch, da sie normalerweise nur für die eindeutige interne Identifizierung benötigt werden, aber ihr Wert nach außen hin eigentlich vollkommen gleichgültig und damit zumeist unbekannt ist. Andererseits gibt es spezielle Primärschlüssel, die auch von der objektorientierten Zugriffsschicht zur Verfügung gestellt werden müssen. Hier sei nur die bekannte Rechnungs-Nummer erwähnt, die gesetzlich in einem Unternehmen eindeutig sein muß und praktisch als Kommunikationsmittel zwischen Unternehmen und Kunden eingesetzt wird. ObjectDRIVER erlaubt das Weglassen dieser Attribute, er verbietet aber nicht deren Definition im Klassen-Schema.

Das `income`-Attribut in dem oben angegebenen Beispiel ist ein sogenannter *calculated slot*. Der Wert des Attributs ergibt sich aus einer Funktion, die sich aus mehreren Attributen der relationalen Seite zusammensetzt. `Income` der Klasse

`Salesman` setzt sich aus dem Grundgehalt eines Employees (`EMP.SAL`) und einer Kommisssion (`EMP.COM`) zusammen.

Weitere Eigenschaften des ObjectDRIVER werden hier nur noch kurz skizziert. Für weitere Informationen wird auf die Dokumentation zu ObjectDRIVER verwiesen [OBJ].

Mit ObjectDRIVER ist es ebenfalls möglich, *bidirektionale Assoziationen* persistent zu definieren. Dabei spricht man von einer bidirektionalen Assoziation, wenn es zu einer Referenz von einem Objekt zu einem anderen Objekt eine Gegenreferenz von dem anderen Objekt zu dem Objekt gibt.

Um Konflikte bei dem Einsatz der Zugriffsschicht zu vermeiden, wird die eine Seite der bidirektionalen Assoziation in ObjectDRIVER als nur lesend definiert. Bei der Definition einer Referenz, die zu einer bidirektionalen Assoziation gehört, muß die Gegenreferenz mit angegeben werden.

Fremdschlüssel-Beziehungen sind auf objektorientierter Seite in der Regel 1:n-Assoziationen. Ein Objekt muß also eine Beziehung zu mehreren Objekten aufbauen können. Dafür stellt ObjectDRIVER die von sich als *Complex Fields* bezeichneten ODMG-Collections wie `Array`, `Set, Bag` oder `List` zur Verfügung. Das Attribut des Objektes, das diese Beziehung definiert, ist dabei zum Beispiel vom Datentyp `Set`. In der dazugehörigen relationalen Korrespondenz-Definition wird ein `join` definiert, der den `Set` mit Werten füllt.

In dem oben aufgeführten Beispiel ist bisher noch nicht die `constrainedBy`-Definition hinter dem Attribut `jobdesc` in der Klasse `Salesman` behandelt worden. Diese Angabe sorgt dafür, daß während der Laufzeit ein `Salesman`-Objekt erzeugt wird, wenn in dem dazugehörigen Datentupel des Relationenschemas `EMP` in dem Feld `EMP.POS` der Wert “Salesman” steht. Da die Klasse `Salesman` von `Employee` erbt, und es durch die Definition der Objektorientierung auch möglich sein muß, von der Klasse `Salesman` zu erben, könnte sich das hier nur kurz vorgestellte Problem ergeben:

Es wird eine weitere Klasse `Officer` eingeführt, die von der besagten Klasse `Salesman` erbt.

```
class Officer inherit Salesman type Tupel {
  jobdesc         String     constrained by (EMP.JOB == ‘‘Officer’’),
...
```

Durch eine zu kleinliche Auslegung auf die Definition des Objekt-Modells ergibt sich der Konflikt, daß ein Objekt vom Typ `Officer` nur erzeugt wird, wenn seine Berufsbezeichnung sowohl “Officer” als auch “Salesman” ist. Ohne weiteres Zutun von ObjectDRIVER ist diese Definition nicht realisierbar. ObjectDRIVER kann diesen Konflikt auflösen, und man kann diese Definition angeben. An dieser Stelle wird aber die Einführung in ObjectDRIVER abgeschlossen, und es wird nur noch auf die Dokumentation zu ObjectDRIVER verwiesen [OBJ].

3.1.4. Folgerung

ObjectDRIVER ist sicherlich ein Werkzeug, mit dem eine objektorientierte Zugriffsschicht auf eine relationale Datenbank erzeugt werden kann. Außerdem bietet er sicherlich auch den Vorteil, daß seine Zugriffsschnittstelle durch ODMG standardisiert ist. Die ODMG-Spezifikation schreibt vor, daß eine ODMG-standardisierte Zugriffsschicht durch Neukompilierung derselben ODMG datenbankunabhängig ist. Dadurch ist man bei Einsatz des ObjectDRIVERs auch nicht auf eine Datenbank festgelegt.

Der wichtigste Gegenaspekt ist die Philosophie von ObjectDRIVER, davon auszugehen, daß sowohl die objektorientierte Struktur als auch die relationale bekannt sind. Der Kernteil von VARLET und somit von REDDMOM ist nicht die Code-Generierung einer objektorientierten Zugriffsschicht, sondern die Analyse des dahinter befindlichen relationalen Schemas, um daraus ein initiales, objektorientiertes Schema zu erzeugen. Dieses von VARLET und REDDMOM angewendete Verfahren wird auch als *Reverse Engineering* bezeichnet. Die Code-Erzeugung ist in diesem Zusammenhang eher als schöner Nebenaspekt zu sehen.

Außerdem setzt ObjectDRIVER viel zu spät an. Schon für die Analyse der Datenbankstruktur wird ein Datenbankzugriff benötigt. ObjectDRIVER geht aber davon aus, daß er für ein vollständiges, also ein komplett analysiertes, System nur eine objektorientierte Zugriffsschicht generieren muß. Der in dieser Arbeit vorgestellte Ansatz bietet aber die Möglichkeit, REDDMOM während der Analyse bereits eine Zugriffsschicht zur Verfügung zu stellen. Diese Zugriffsschicht ist durch die während der Analyse neu gewonnenen Erkenntnisse noch veränderbar.

Ein weiterer Punkt ist die um den Generator befindliche Umgebung. REDDMOM bietet durch FUJABA die Modellierung von Activity-Diagrammen an. Nach Fertigstellung des hier vorgestellten Generators sollen die persistenten Objekte auch in den dynamischen Modellarten von FUJABA verwendet werden können. FUJABA geht bei der Generierung des Codes nach der reinen Lehre der Software-Technik vor: Der Zugriff auf Attribute eines Objektes erfolgt nur über seine Zugriffsmethoden. Das heißt, daß bei der Benutzung eines Objektes auf einem Activity-Diagramm eine spezielle Schnittstelle von dem Objekt erwartet wird. Dadurch, daß ObjectDRIVER diese Schnittstelle bei seiner Persistenz-Definition nicht benötigt, ist es fraglich, inwieweit diese beiden Systeme später zusammen harmonieren.

Eine mögliche Entfernung von der reinen Lehre, nur um ObjectDRIVER einsetzen zu können, kommt für die beiden Projekte REDDMOM und FUJABA nicht in Frage. Da es das Ziel dieser Diplomarbeit ist, REDDMOM um Persistenzfähigkeit zu erweitern, ist es wahrscheinlich einfacher, sich an den Vorgaben von FUJABA zu orientieren, um eine persistente Zugriffsschicht zu generieren. Spezielle Eigenschaften der persistenten Objekte wie etwa das ausschließliche Entgegennehmen

von Änderungen nur innerhalb einer Transaktion können dann in die Generierung der Activity-Diagramme nachgetragen werden.

Desweiteren soll der Aufwand für die Implementierung des Generators erwähnt werden. Für die Zugriffsschicht müssen für ObjectDRIVER zwei Textdateien erstellt werden, die beide unterschiedlichen Grammatiken unterliegen. Beide Grammatiken müssen studiert und bei der Generierung umgesetzt werden. Das Studium darf hierbei nicht unterschätzt werden, da hier gerade der benötigte relationale join für die Definition der objektorientierten Assoziationen sehr komplex ist. In der gleichen Zeit kann man sich ein objektorientiertes Konzept überlegen, auf dem die objektorientierte Zugriffsschicht aufbaut. Der Aufwand in der Generierung an sich ist bei beiden Verfahren gleich.

ObjectDRIVER ist nur für Windows erhältlich. Eine der Stärken von Java ist seine Plattformunabhängigkeit. Dadurch, daß man während der Entwicklung irgend wann einmal ObjectDRIVER zur Erstellung der persistenten Zugriffsschicht ausführen muß, müßte extra dafür eine Windows-Plattform vorhanden sein. Das ist für eine plattformunabhängige Entwicklung eine zu große Einschränkung.

Wie bereits in Kapitel 1.2 erwähnt, basiert der Generator auf dem Metamodell des Projektes FUJABA und verwendet für die Erzeugung des initialen Schemas Praktiken, die in VARLET realisiert wurden. Bei beiden Projekten handelt es sich, kurz gesagt, um GNU-Projekte. Beide Applikationen sind über das Internet frei erhältlich, und jeder darf an der Weiterentwicklung der Projekte teilnehmen. Von beiden Projekten ist der Quellcode erhältlich. Das auf den beiden Projekten aufbauende Projekt REDDMOM soll ebenfalls frei verfügbar sein.

ObjectDRIVER ist kommerziell. Bei der alleinigen Unterstützung von ObjectDRIVER müßte diese Philosophie der Verteilung überdacht und geändert werden, was sicherlich nicht der gravierende Grund sein kann. Es muß dem Interessenten an REDDMOM aber freigestellt sein, die Entscheidung selbst zu treffen, ObjectDRIVER zu kaufen und bei sich zu installieren. Ohne die Installation von ObjectDRIVER wäre REDDMOM ziemlich nutzlos. Die Applikation kann zwar auch ohne die Insttallation von ObjectDRIVER Modellierungen an einer Datenbank vornehmen. Das entscheidende dieser Diplomarbeit ist aber eine lauffähige objektorientierte Zugriffsschicht auf eine relationale Datenbank, und das wäre nur mit einer installierten ObjectDRIVER-Version gewährleistet.

Durch die Erzeugung seiner eigenen Zugriffsschicht ist das Projekt in sich abgeschlossen und nicht von der Politik eines Fremdanbieters abhängig. Eine Unterstützung des ObjectDRIVER und auch der anderen oben genannten Projekte kann danach als Alternative angeboten werden.

3.2. VARLET

Das Projekt REDDMOM wird die in dem Projekt VARLET [Jah99] realisierten Verfahren übernehmen. In der Abbildung 3.2 sieht man eine Übersicht von Varlet, die in diesem Abschnitt vorgestellt wird.

3.2.1. Das System

Bei der Erzeugung einer objektorientierten Zugriffsschicht auf eine relationale Datenbank muß zuerst das relationale Schema der Datenbank analysiert werden. Zu der Analyse des Schemas gehört zunächst das Auslesen der Struktur der relationalen Schemata innerhalb der Datenbank. Ein aus der Datenbankentwicklung bekanntes Entity-Relationship-Diagramm entsteht.

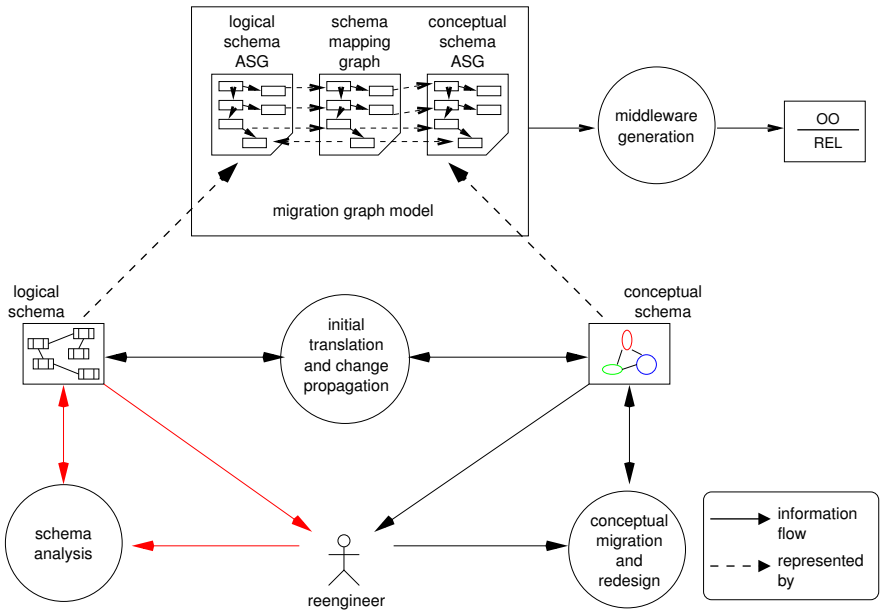


Abbildung 3.2.: Quelle [Jah99], S. 133: "incremental schema migration and generative data migration"

Im Regelfall haben Datenbanken eine sehr lange Lebenszeit. Sie bestehen meist mehrere Jahre, wenn nicht sogar Jahrzehnte. Neben der eigentlichen Benutzung der Datenbanken, nämlich der Speicherung von anfallenden Daten, werden sie auch weiterentwickelt. Dabei kann es passieren, daß nicht alle Änderungen, die eigentlich das Schema berühren müßten, auch wirklich in das Schema eingetragen werden. Eine Fremdschlüssel-Beziehung zwischen zwei Tabellen wird zum Beispiel, wenn sie in der Datenbank definiert würde, als ein sogenannter *Constraint* von der Datenbank verstanden.

Ein solcher Constraint unterliegt Gültigkeitsregeln, die von den betroffenen Daten nicht verletzt werden dürfen. Bei dem Nachtrag eines solchen Constraint kann es passieren, daß Daten innerhalb der Datenbank vorliegen, die gegen diesen Constraint verstoßen. Die Datenbank würde erst die Definition des Constraint erlauben, wenn alle Konflikte innerhalb der Daten aufgehoben sind. Dabei kann es sich um so viele Konflikte handeln, daß ihre Beseitigung einen erheblichen Zeit- und Kostenaufwand verursachen würde. Dadurch, daß die Definition eines Constraints innerhalb der Datenbank keine Erweiterung der Funktionalität auf Seiten der Applikation mit sich bringt, sondern "nur" die Sicherheit der Daten erhöht, bietet es sich an, den Constraint nicht in der Datenbank zu definieren, sondern innerhalb der Applikation zu simulieren. Dabei könnten die gegen den Constraint verstoßenden Daten extra behandelt werden.

Um auf einer solchen Datenbank später eine objektorientierte Zugriffsschicht aufzusetzen, müssen die in der Applikation versteckten Constraints ermittelt werden, damit die Zugriffsschicht äquivalent zu dem bisherigen funktioniert. VARLET bietet Methoden an, solche Constraints ausfindig zu machen. Aus der ausgelesenen Struktur und den erkannten versteckten Constraints entsteht das logische Schema. Intern wird es als abstrakter Syntax-Graph gehalten (ASG).

Für weiterführende Informationen zu der Analyse relationaler Datenbanken wird an dieser Stelle auf [Jah99] verwiesen.

Um eine objektorientierte Zugriffsschicht auf eine relationale Datenbank zu bekommen, benötigt man zunächst eine konzeptionelle Darstellung bzw. ein UML-Klassendiagramm des logischen Schemas. Es muß das konzeptionelle Schema initialisiert werden. Dabei wird ein Regelwerk durchlaufen, das alle Elemente des logischen Schemas in äquivalente konzeptionelle Elemente transformiert.

Bei der Transformation sollen auch Modell-Elemente auf konzeptioneller Seite berücksichtigt werden, die von dem logischen Schema nicht angeboten werden. Die in dem Kapitel 2.1.2 vorgestellte Vererbung auf konzeptioneller Seite ist ein solches Element. Die aus dem Kapitel 2.1.3 bekannte Varianten-Definition auf dem relationalen Schema ist eine Darstellungsform der Vererbung auf logischer Seite.

Diese Varianten können, wie bereits erwähnt, nicht über die Datenbank definiert werden. Es sind Entscheidungen, die außerhalb der Datenbank getroffen werden, und um die das logische Schema angereichert werden muß, damit sie in das konzeptionelle Schema übertragen werden können.

Die Transformation eines relationalen Schemas in ein konzeptionelles Schema muß als initial bezeichnet werden. Bis zu der in Abbildung 3.2 gezeigten Code-Erzeugung ist die Modellierung der objektorientierten Zugriffsschicht ein langwieriger Prozeß. Auch nach der ersten Transformation müssen Änderungen vorgenommen werden, ohne daß die beiden Seiten inkonsistent zueinander werden.

Um beide Seiten konsistent halten zu können, darf die Transformation als solche nicht verloren gehen. Jede einzelne Transformation eines logischen Elements in ein konzeptionelles muß behalten werden bzw. nachvollziehbar sein. Es wird zwischen den beiden Schemata ein Transformations-Graph aufgebaut, der diese Nachvollziehbarkeit gewährleistet. Mit der Konsistenzerhaltung befaßt sich die Diplomarbeit von J. Wadsack [Wad98]. Die Code-Generierung wurde von H. Schalldach [Sch98] realisiert.

Der in der Abbildung 3.2 gezeigte Reengineer kontrolliert den gesamten Prozeß. Er hat jederzeit die Möglichkeit, manuell in den Prozeß einzugreifen. Bei dem zuvor vorgestellten Analyse-Prozeß können Ergebnisse erzielt worden sein, die nicht automatisch behandelbar sind. Auch hier muß der Reengineer manuell eingreifen.

3.2.2. Folgerung

Da bereits bekannt ist, daß das hier verwendete Verfahren zur Erzeugung des konzeptionellen Schemas von VARLET stammt, kann man sich fragen, warum dieses Projekt nicht weitergeführt wird.

VARLET ist vor dem Aufkommen von Java entwickelt worden. Als Entwicklungssprachen wurden C, C++, Progres, tcl/tk und später auch Java eingesetzt. Es war sehr aufwendig, dieses Projekt weiter zu pflegen. Durch den Einsatz von so vielen Sprachen wurde das Testen sehr erschwert. Es stand das Hilfsmittel des *Debuggens* nicht mehr zur Verfügung.

Progres[4] ist ein Spezifikationswerkzeug, das auf Graphen aufsetzt. Intern verwendet dieses Werkzeug eine Datenbank, die knotenbasiert arbeitet. Die Anzahl der Knoten ist durch die Datenbank beschränkt. Dadurch ist die Größe der Projekte von Progres selbst beschränkt.

Weil die Überschreitung der Anzahl der Knoten von Progres nicht abgefangen wird, sondern einfach passiert, gingen damals die bis dahin gemachten Änderungen bei dem anstehenden Absturz verloren. Als Notlösung wurde VARLET in Teilprojekte mit einer gemeinsamen Basis aufgeteilt. Am Ende waren es fünf Teilprojekte. Wenn an der Basiskonfiguration etwas geändert wurde, und das kam häufiger vor, mußten alle fünf Teilprojekte angepaßt und neu generiert werden.

Auf einer damals aktuellen Sun SPARC Ultra2 Maschine mit 1GB Hauptspeicher dauerte die Generierung über eine Stunde. Auf einem heutigen, gewöhnlichen, halbwegs modernen PC dauert die Neukompilierung von REDDMOM nicht einmal 30 Sekunden.

Progres wurde nur für einige Unix-Derivate angeboten, sodaß auch die Entwicklung auf diese Systeme beschränkt war. Über eine Redhat-Linux Version konnte immerhin der PC als Arbeitsplattform herangezogen werden. Windows wurde

[4]Progres: Programmiertes Graph Ersetzungs-System

nicht unterstützt. Da während der Laufzeit von VARLET auch die progresinterne Datenbank benötigt wurde und diese auch nur auf den von Progres unterstützten Systemen lief, war VARLET auf einem Windows-Rechner nicht ausführbar.

Die Entwicklung mit Progres erforderte einen Riesenaufwand an Hauptspeicher. Auf dem oben erwähnten Sun Rechner mit dem 1GB war ein ordentliches Arbeiten möglich. Es ist aber auch vorgekommen, daß Progres alleine dieses eine GB benötigte. Der Einsatz eines PC war damit eigentlich wieder ausgeschlossen, da es damals noch nicht möglich war, einen PC mit einem GB oder mehr auszustatten.

Das Auslesen der Datenbank-Informationen erfolgte mit dem auf C basierenden Lex/Yacc-Parser. Es wurden die Schema-Informationen aus einer Textdatei gelesen. Die Schema-Informationen lagen in irgend einem SQL-Dialekt vor, der von einem Datenbanksystem verwendet wurde. Der Parser mußte immer angepaßt werden, wenn ein neues Datenbank-Schema eingelesen werden sollte bzw. mußte. Eine Datenbank-Verbindung von VARLET aus wurde nie realisiert.

In VARLET wurde ebenfalls eine objektorientierte Java-Zugriffsschicht generiert [Sch98]. Dadurch, daß sich aber die damalige Logik von VARLET sehr stark von der von FUJABA und REDDMOM unterscheidet, war es einfacher, nicht das vorhandene an die neue Umgebung anzupassen, sondern die Generierung neu zu schreiben.

Durch die Entscheidung, FUJABA als Grundlage für das Projekt REDDMOM zu verwenden, stehen REDDMOM sämtliche Funktionalitäten von FUJABA zur Verfügung. VARLET beschränkte sich damals auf die Erzeugung eines OMT-Diagramms. Der hier vorgestellte Generator beschränkt sich zwar auch nur auf die Struktur-Erzeugung eines Klassendiagramms. Es ist aber jetzt schon nach Fertigstellung dieser Diplomarbeit möglich, innerhalb eines Activity-Diagramms von FUJABA lesend auf persistente Objekte zuzugreifen. Der schreibende Zugriff wird dadurch unmöglich, da die Activity-Diagramme derzeit noch kein Spezifikations-Element für Transaktionen anbieten.

Durch die oben genannten Punkte wurde natürlich auch das Vorankommen in dem Projekt sehr beeinträchtigt. Durch die Neuimplementierung der Verfahren von VARLET in REDDMOM ergeben sich die folgenden Vorteile:

Es gibt keine Einschränkungen in der Größe mehr, da man sich von Progres getrennt hat.

Es wird nur noch in Java implementiert.

Da es für Java auch Debugger gibt, wird das Testen seiner Implementierung einfacher. Daneben gibt es noch weitere Applikationen, die die Entwicklung von Java-Programmen unterstützen und auch bei der Entwicklung von REDDMOM eingesetzt werden können. Hierzu zählen zum Beispiel javadoc und OptimizIt. Außerdem erhält man die durch Java gewährleistete Plattformunabhängigkeit.

Zum Parsen eines relationalen Schemas steht die JDBC-API zur Verfügung. Im Gegensatz zu vorher wird hier eine direkte Datenbankverbindung aufgebaut, und die Schema-Informationen werden in Strings gekapselt.

Damit kann aus jeder Datenbank ihr Schema ausgelesen werden, wenn die Datenbank es gestattet.

Die unterschiedlichen Dialekte können durch Konfigurationsmöglichkeiten in REDDMOM gelöst werden. Auch ohne die Konfiguration erhält man aber immerhin schon einmal die Präsentation eines logischen Schemas.

4. Die Datenbank-Schemata

Diese Diplomarbeit umfaßt in ihrer Implementierung nicht nur die Code-Erstellung des Generators. Die Implementierung beinhaltet auch die Erstellung eines konzeptionellen Rahmenwerkes, auf dem der Generator aufsetzt. Das Rahmenwerk deckt die Erzeugung eines relationalen Schemas aus den Informationen einer Datenbank sowie die Erzeugung eines initialen konzeptionellen Schemas aus dem relationalen Schema ab.

Für die Darstellung des konzeptionellen Schemas wird das UML-Metamodell von FUJABA [FUJ] verwendet. Im Rahmen dieser Diplomarbeit wurde das Metamodell von FUJABA um die Darstellungsmöglichkeit von relationalen Schemata erweitert. In den folgenden Abschnitten soll das konzeptionelle Rahmenwerk näher vorgestellt werden.

In dieser Arbeit wird versucht, die Vorstellung des Konzepts, das sich hinter den Schemata befindet, unabhängig von einem speziellen Datenbank-Schema zu halten. Gerade bei der Vorstellung des dynamischen Verhaltens ist es aber sinnvoller, auf ein funktionierendes Beispiel zurückzugreifen. Das in den folgenden Kapiteln immer wiederkehrende Beispiel einer relationalen Datenbank hat die DSD-Datenbank als Grundlage.

Das Projekt DSD [GNZ01] befaßt sich mit der Versionierung und Konfiguration von Dokumenten und speichert seine Daten in der oben erwähnten Datenbank ab. Um auf spezielle Eigenschaften von relationalen Datenbanken im allgemeinen einzugehen, die in der DSD-Datenbank vorher nicht enthalten waren, wurde eine Kopie von der DSD-Datenbank angelegt und die Struktur der Kopie an die Belange dieser Arbeit angepaßt. Die Kopie wird in dieser Arbeit aber der Einfachheit halber als *dsd* bezeichnet. Bei Bedarf wird auf Ausschnitte der Struktur der Datenbank eingegangen. Eine komplette Darstellung des relationalen und des initialen objektorientierten Schemas zu dieser Datenbank befindet sich im Anhang (A.1, A.2).

4.1. Das Relationale Schema

Wie bereits erwähnt, befaßt sich die hier vorgestellte Arbeit bei der Ermittlung des logischen Schemas ausschließlich mit dem Auslesen eines vorhandenen Schemas aus einer existierenden Datenbank. Es muß ein Hilfmittel geschaffen werden, das die Informationen aus dem Datenbank-Schema ermittelt und in eine für den Generator bekannte Form umsetzt.

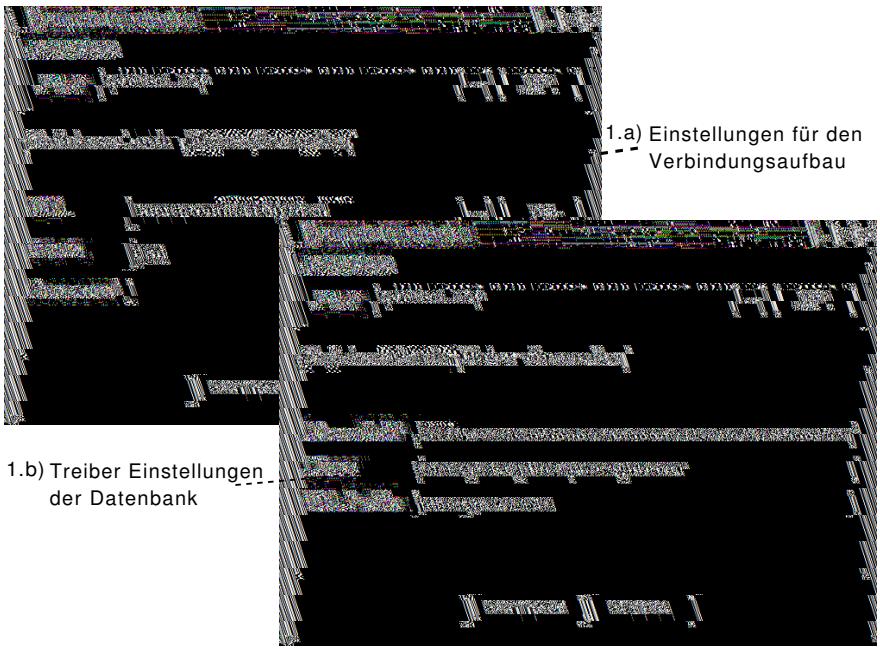


Abbildung 4.1.: Anmeldung bei der Datenbank

Die Darstellung des Datenbank-Schemas wird innerhalb von REDDMOM als EER[1]-Diagramm bezeichnet. Bei der Neuanlage eines solchen Diagramms muß eine Verbindung zu einem Datenbank-Server aufgebaut werden. Die benötigten Angaben für den Verbindungsaufbau sind in Abbildung 4.1 zu sehen. Neben den herkömmlichen Informationen wie der Datenbank-Adresse (`URL`), dem `login` und dem `Passwort` müssen datenbank- bzw. JDBC-Treiber-spezifische Einstellungen vorgenommen werden, die in der Abbildung 4.1 in dem mit 1.b) bezeichneten Dialog zu sehen sind.

Da bei dem Auslesen der Datenbank-Struktur JDBC eingesetzt wird, soll zunächst eine Klärung der Begriffe ,,Datenbank" und ,,Datenbankunabhängigkeit" erfolgen, die im Zusammenhang mit JDBC sehr oft fehlinterpretiert werden.

[1] EER: Extended Entity Relationship

Der Begriff Datenbank

Das Wort "Datenbank" muß in diesem Zusammenhang sehr frei verstanden werden. In der JDBC-Dokumentation steht

> The JDBC API is a Java API for accessing virtually any kind of tabular data. (As a point of interest, JDBC is the trademarked name and is not an acronym; nevertheless, JDBC is often thought of as standing for "Java Database Connectivity.")

Zuerst muß man feststellen, daß nicht von einer Datenbank die Rede ist, sondern von Daten, die in tabellarischer Form vorliegen. Hierzu zählen neben der Art Datenbank unter anderem speziell strukturierte Textdateien und auch Datenblätter von Tabellenkalkulationen. Tatsächlich gibt es Ausimplementierungen des JDBC-Paketes, die es erlauben, auf Microsoft [Mic]-Excel-Tabellen zuzugreifen. Eine Zugriffsmöglichkeit auf strukturierte Textdateien bietet zum Beispiel die Ausimplementierung des Projektes *tinySQL [TIN]* an.

Die in Klammern stehende Anmerkung verschärft diesen Sachverhalt dahingehend, daß sie die oft falsch wiedergegebene Darstellung des Begriffs JDBC widerlegt. Dadurch, daß JDBC eben nicht für "Java Database Connectivity" steht, ergibt sich, daß Sun bei der Entwicklung dieses Paketes nie die Absicht hatte, nur Datenbanken über dieses Paket zu unterstützen. Diese Diplomarbeit befaßt sich aber nur mit den relationalen Datenbanken.

Datenbankunabhängigkeit

In der Dokumentation heißt es weiter:

> The JDBC API makes it easy to send SQL statements to relational database systems and supports all dialects of SQL. But the JDBC 2.0 API goes beyond SQL, also making it possible to interact with other kinds of data sources, such as files containing tabular data.

Fälschlicherweise wird oftmals das JDBC-Paket als eine "datenbankunabhängige Zugriffsmöglichkeit" verstanden. Auf der objektorientierten Seite trifft das auch zu, da der Zugriff durch die vorgegebenen Schnittstellen wohldefiniert ist. Datenbankunabhängig geht aber weiter. Die SQL-Aufrufe werden zur Ausführung als String-Parameter an den Datenbank-Server übergeben. Um datenbankunabhängig zu sein, muß der Inhalt dieser Aufrufe von jeder Datenbank verstanden werden. "All dialects of SQL" spricht gegen diese These. Man kann von einem Sybase [SYB] Datenbank-Server zum Beispiel nicht erwarten, daß er neben seinem SQL-Dialekt auch den von Oracle [ORA] versteht.

Außerdem wird in dem Zitat dem Hersteller einer Datenbank die Möglichkeit gegeben, seine eigene, von SQL vollkommen losgelöste Sprache zu verwenden. Diese Freiheit in der Gestaltung der Datenbank-Anfragen ist für den Generator nachteilig, da er genau wissen muß, für welche Datenbank er gerade eine Zugriffsschicht generiert. Daß der Zugriff auf relationale Datenbanken nicht datenbankunabhängig ist, wird im Laufe der Arbeit durch Beispiele noch weiter vertieft.

4.1.1. Auslesen der Datenbank-Struktur

Neben den in Kapitel 2.1.4 vorgestellten Standard-Schnittstellen für die allgemeine Kommunikation mit dem Datenbank-Server stellt das JDBC-Paket noch zwei weitere Schnittstellen zur Verfügung, die das Auslesen der Datenbank- Struktur ermöglichen. Die erste Schnittstelle heißt `java.sql.DatabaseMetaData`. Die zweite Schnittstelle heißt `java.sql.ResultSetMetaData`. Die beiden Schnittstellen werden im folgenden vorgestellt.

java.sql.DatabaseMetaData

Diese Schnittstelle erlaubt das Auslesen allgemeiner Datenbank-Informationen. Insgesamt besitzt die Schnittstelle 151 Methoden-Definitionen. Die meisten dieser Methoden spezifizieren den Zugriff auf Eigenschaften des Datenbank-Servers. Die Eigenschaften-Methoden können in vier Kategorien eingeteilt werden.

1. *Allgemeine Informationen*:
   In diese Kategorie fallen die Methoden, die den Datenbank-Server und den JDBC-Treiber identifizieren. Hierzu zählt zum Beispiel die `getDatabaseProductName()`-Methode, die den offiziellen Namen des vorliegenden Datenbank-Servers zurückgibt. Eine andere Methode ist die `getDriverName()`- Methode, die den offiziellen Namen des verwendeten JDBC-Treibers wiedergibt.

2. *Maximale Werte*:
   In der Definition von Tabellen sowie in der Verwendung der Abfragesprache SQL wird man durch einige Eigenschaften des Datenbank-Servers beschränkt. Zum Beispiel verbietet der Datenbank-Server Sybase Adaptive Server Enterprise in der Version 11.9.2 die Definition einer Tabelle, deren Name länger als 30 Zeichen ist (`getMaxTableNameLength()`). In einer aus SQL bekannten select-Anweisung erlaubt derselbe Server maximal 16 Spalten in einer `Order By` Angabe (`getMaxColumnsInOrderBy`).

3. *SQL- und JDBC-Unterstützung*:
   Auch wenn das JDBC-Paket nicht nur für relationale Datenbanken gedacht ist, orientiert sich seine Schnittstellendefinition an dem SQL92- Standard. Natürlich ist es klar, daß die über einen JDBC-Treiber angesprochenen Systeme, die keine relationale Datenbank als Grundlage haben, diesen Standard nicht vollständig unterstützen. Auch reale relationale Datenbanken unterstützen diesen Standard nicht immer vollständig. Über die in diese Kategorie fallenden Methoden kann der Benutzer erfahren, ob die von ihm verwendete relationale Datenbank zum Beispiel *Stored Procedures* (`supportsStoredProcedures`) unterstützt oder welchen Transaktions-Isolations-Level (`supportsTransactionIsolationLevel`) sie anbietet.
   Bei der Ausimplementierung des JDBC-Paketes ist es dem Treiber-Anbieter offengelassen, alle Möglichkeiten des Datenbankzugriffs, die das JDBC-Paket definiert, zu unterstützen. Zu den Methoden, die die Unterstützung des JDBC -Paketes spezifizieren, zählt zum Beispiel `supportsBatchUpdate()`. Für eine Erklärung des Begriffs *Batch Update* wird hier nur noch auf die JDBC-Spezifikation verwiesen [JDB].

4. *SQL-Eigenschaften und Datenbank-Funktionen:*
   In dem SQL92-Standard wird zum Beispiel der spezielle Wert `null` definiert. Ein Attribut, das den Wert null aufweist, ist in seinem Wert als undefiniert gekennzeichnet. Da `null` somit kein Wert ist, Daten aber sehr oft nach Werten sortiert ausgegeben werden, stellt sich die Frage, wo denn die `null`-Werte erscheinen. Die Methode `nullsAreSortedAtEnd()` gibt an, daß die `null`-Werte von der Datenbank unabhängig von der Sortierreihenfolge immer am Ende ausgegeben werden. Die Methode `nullsAreSortedAtStart()` hingegen verweist darauf, daß die `null`-Werte immer als erstes auftreten. Es gibt insgesamt sechs Methoden, die sich auschließlich mit dem `null`-Wert befassen.

Die Methoden der Klasse `java.sql.DatabaseMetaData`, die nicht in die vier vorgestellten Kategorien fallen, definieren den Zugriff auf die Struktur der Datenbank. Da diese Methoden für die vorliegende Diplomarbeit von besonderer Bedeutung sind, sollen sie im folgenden in ihrer Definition und in ihrer Verwendung näher vorgestellt werden. Die Tabelle 4.1 gibt einen Überblick über die wichtigsten Methoden, die es ermöglichen, die Datenbank-Struktur auszulesen. Bei der Ermittlung des logischen Schemas werden zunächst die vorhandenen relationalen Schemata ausgelesen. Hierfür steht die Methode `getTables` zur Verfügung. Man bekommt ein Objekt vom Typ `ResultSet` als Ergebnis geliefert. Jede Zeile repräsentiert dabei ein relationales Schema aus der Datenbank. Eine genaue Definition der Spalten des `ResultSet`s, das von `getTables` zurückgegeben wird, kann in der JDBC-Spezifikation [JDB] nachgelesen werden.

Methoden-Name	Erläuterung
`ResultSet getTables(...)`	gibt die in der Datenbank bekannten Tabellen-Namen zurück.
`ResultSet getColumns(...)`	liefert die Spalten einer Tabelle, die als ein Parameter übergeben wird.
`ResultSet getPrimaryKeys(...)`	ermittelt den zu einer Tabelle definierten Primärschlüssel.
`ResultSet getCrossReference(...)`	gibt die in einer Tabelle definierten foreign-keys zurück.
`ResultSet getExportedKeys(...)`	gibt die in einer Tabelle definierten foreign-keys zurück.
`ResultSet getImportedKeys(...)`	gibt die Primärschlüssel zurück, die durch die definierten foreign-keys der Tabelle referenziert werden

Tabelle 4.1.: Methoden der Schnittstelle java.sql.DatabaseMetaData

Unter anderem beinhaltet dieses `ResultSet` eine Spalte, die den Namen eines relationalen Schemas darstellt. In dem Metamodell von REDDMOM wird ein relationales Schema einer Datenbank durch eine Instanz vom Typ `SQLTable` präsentiert. Ein Ausschnitt aus dem Metamodell, in dem auch `SQLTable` enthalten ist, ist in Abbildung 4.2 zu sehen.

Beim Einlesen des relationalen Schemas wird zu jeder Zeile des `ResultSets` ein `SQLTable`-Objekt erzeugt, und das Objekt wird nach der Namenspalte des `ResultSets` benannt. Die übrigen in der Tabelle 4.1 angegebenen Methoden bieten über einen Parameter die Möglichkeit an, nur Werte zu einem angegebenen relationalen Schema zurückzugeben. Diese Eigenschaft wird bei dem Einlesen des relationalen Schemas ausgenutzt. Sobald ein neues relationales Schema gelesen und ein `SQLTable`-Objekt erzeugt wurde, wird das `SQLTable`-Objekt durch Aufruf der übrigen Methoden mit Angabe des `SQLTable`-Namens um weitere logische Eigenschaften ergänzt. Man erhält die Attribute des relationalen Schemas (`SQLAttr`), dessen Primärschlüssel (`SQLPrimaryKey`) und die zu dem relationalen Schema definierten Indizes (`SQLIndex`).

Die letzten drei Methoden in der Tabelle 4.1 beziehen sich auf die Definition der Fremdschlüssel innerhalb des Datenbank-Schemas. Um die über die Fremdschlüssel aufgebauten Referenzen in die logische Darstellung von REDDMOM übernehmen zu können, müssen die relationalen Schemata sowie ihre Primärschlüssel eingelesen worden sein und als `SQLTable`- bzw. `SQLPrimaryKey`-Objekte in der Logik von REDDMOM vorliegen.

Die von den drei Methoden gelieferten `ResultSets` enthalten den Namen der Referenz-Tabelle sowie den Namen der Tabelle, auf die referenziert wird. Bei

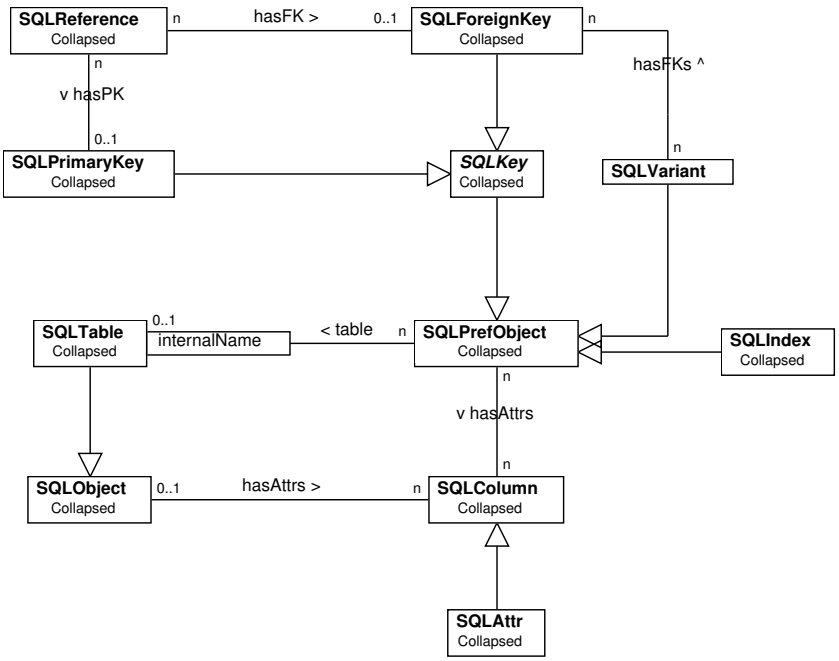


Abbildung 4.2.: Ausschnitt aus dem SQL-Metamodell von REDDMOM

dem Abarbeiten der einzelnen Zeilen der `ResultSet`s werden die beiden in der Logik von REDDMOM vorhandenen `SQLTable`-Objekte ermittelt.

Zwei weitere Spalten der `ResultSet`s definieren ein Attribut-Paar, das an der Definition der Fremdschlüssel-Beziehung beteiligt ist. Dabei ist das erste Attribut Element des Fremdschlüssels der Referenztabelle, und das zweite ist das Attribut aus der referenzierten Tabelle, auf das sich das erste bezieht.

Fremdschlüssel, die über mehrere Attribute aufgebaut werden, werden in den `ResultSet`s über mehrere, der Anzahl der beteiligten Attribute entsprechende Zeilen ausgegeben. Jede Fremdschlüssel-Beziehung besitzt auf relationaler Seite einen eindeutigen Namen, der entweder vom Datenbank-Designer oder vom Datenbank-Server intern vergeben wurde. Dieser Name ist ebenfalls Bestandteil der `ResultSet`s, sodaß eine eindeutige Bestimmung der Fremdschlüssel und die dadurch entstehende Beziehung zwischen zwei Tabellen möglich ist.

Jede gefundene Fremdschlüssel-Beziehung erzeugt ein `SQLReference`-Objekt und ein `SQLForeignKey`-Objekt in der logischen Struktur von REDDMOM. Während des Lesens dieser einen Fremdschlüssel-Beziehung über gegebenenfalls mehrere `ResultSet`-Spalten werden die zu dem Fremdschlüssel gehörenden Tabellen-Attribute ermittelt und dem `SQLForeignKey`-Objekt zugeordnet.

In dem in Abbildung 4.2 gezeigten Auszug aus dem SQL-Metamodell ist es nicht ersichtlich, wozu es die beiden Klassen `SQLObject` und `SQLColumn` gibt. Die hier

vorgestellte Ermittlung des logischen Schemas einer relationalen Datenbank ist aber ebenfalls nur ein Auszug des eigentlichen Verfahrens. Neben den hier vorgestellten sechs Methoden bietet die Schnittstelle `java.sql.DatabaseMetaData` zahlreiche Methoden an, die das logische Schema der Datenbank berühren. Es ist ebenfalls möglich, über die Schnittstelle die in der Datenbank gehaltenen *Stored Procedures* zu ermitteln und deren Signatur auszulesen.

Bei der Erstellung der zu dem Metamodell gehörenden Klassenstruktur wurde darauf geachtet, daß sämtliche Informationen aus den 151 Methoden der Klasse `java.sql.DatabaseMetaData` gehalten werden können. Damit gibt es neben den Klassen `SQLTable` und `SQLAttr` auch die Klassen `SQLStoredProcedure` und `SQLProcedureColumn`, die jeweils von den Klassen `SQLObject` und `SQLColumn` erben.

Die in Abbildung 4.2 gezeigte `SQLVariant`-Klasse ermöglicht die bereits vorgestellte Varianten-Definition auf Relationenschemata. Sie wird in dem Kapitel 4.1.3 noch näher vorgestellt.

`SQLIndex`, `SQLPrimaryKey`, `SQLForeignKey` und weitere in der Abbildung 4.2 nicht enthaltene Klassen besitzen Attribute und beziehen sich immer auf eine Tabelle. Deswegen wurde das in der Abbildung 4.2 zu sehende `SQLPrefObject` eingefügt, das diese gleichen Eigenschaften definiert. Ein relationales Schema, das durch ein `SQLTable`-Objekt repräsentiert wird, kann mehrere foreign-keys und Indizes besitzen. Dadurch entsteht die `table`-Assoziation zwischen den Klassen `SQLPrefObject` und `SQLTable`. Qualifiziert ist sie, um von der `SQLTable`-Seite komfortabler auf eine bestimmte Art von `SQLPrefObjecten` wie zum Beispiel allen `SQLIndexes` zugreifen zu können.

java.sql.ResultSetMetaData

Bei der Erstellung des logischen Schemas werden auch die Attribute der relationalen Schemata übernommen. In Kapitel 2.1.1 wurde ein Attribut eines relationalen Schemas als ein 2-Tupel definiert, welches aus einem Namen und einem Datentypen besteht. Über den Datentypen eines relationalen Attributs wird definiert, welche Werte es enthalten darf, und wie es in einer SQL-Abfrage verwendet werden muß. Der durch den SQL92-Standard definierte `VARCHAR`-Datentyp zum Beispiel erlaubt die Speicherung aller alphanumerischen Zeichen. Zu den alphanumerischen Zeichen gehört auch das Leerzeichen.

Eine SQL-Abfrage hingegen ist eine Aneinanderreihung von Befehlen und Werten, die einer Grammatik gehorcht, und die als Zeichenkette an den Datenbank-Server gesendet wird. In der Grammatik ist das Leerzeichen als Trennzeichen zwischen den einzelnen SQL-Befehlen definiert. Ohne weitere Kennzeichnung eines alphanumerischen Wertes wäre es nicht möglich, ein Leerzeichen in einem `VARCHAR`-Attribut abzuspeichern, da der Datenbank-Server die in dem Wert enthaltenen

Leerzeichen fehlinterpretieren würde. Um das Abspeichern eines alphanumerischen Wertes, der unter Umständen Leerzeichen enthält, zu gewährleisten, wird er durch die Verwendung von Sonderzeichen eingerahmt. Das zumeist verwendete Sonderzeichen ist in diesem Zusammenhang das Anführungszeichen.

Eine objektorientierte Zugriffsschicht auf eine relationale Datenbank kapselt die Werte aus den relationalen Schemata in Objekten. Über die Objekte werden die Daten in der Datenbank manipuliert. Damit die Daten dauerhaft in der Datenbank gespeichert werden können, müssen die Änderungen an den Objekten intern an den Datenbank-Server weitergeleitet werden. Für die Weiterleitung an den Datenbank-Server wird die Abfragesprache SQL verwendet. Damit der Datenbank-Server die SQL-Abfrage versteht, muß die SQL-Abfrage in seiner Grammatik artikuliert, und die in der Abfrage enthaltenen Werte müssen in seinem ihm bekannten Format formatiert worden sein.

Aus den oben erwähnten Gründen muß beim Erstellen des logischen Schemas bekannt sein, von welchem Datentyp die einzelnen Attribute sind, und welche besonderen Formatierungen ihre Daten benötigen, wenn sie später in der objektorientierten Zugriffsschicht in einer SQL-Abfrage benutzt werden.

Die bereits vorgestellte Schnittstelle `java.sql.DatabaseMetaData` bietet in ihrer Definition die Methode `getTypeInfo` an, die allgemeine Eigenschaften zu den relationalen Datentypen des Datenbank-Servers bereitstellt. Über diese Methode kann ermittelt werden, ob der Datentyp `VARCHAR` allgemein den `null`-Wert als Wert zuläßt, oder ob er als `autoincrement`-Datentyp verwendet werden darf. Sie liefert auch das oben erwähnte Trennzeichen für seine Werte. Dabei richtet sich das Ergebnis nach dem Datenbank-Server. Das Ergebnis ist keine konstante Definition des JDBC-Paketes. Die Ergebnisse von zwei verschiedenen Datenbank-Servern zu dem Datentyp `VARCHAR` können unterschiedlich sein.

Die bereits erwähnten Einschränkungen durch den Datenbank-Server, wie zum Beispiel die Länge des Tabellennamens, erstrecken sich auch auf die Datentypen des Datenbank-Servers. Das Ergebnis der Methode `getTypeInfo` gibt hier ebenfalls Auskunft. Hier kann die maximale Länge bzw. die maximale Genauigkeit sowie die maximal mögliche Anzahl an Nachkommastellen zu einem Datentyp des Datenbank-Servers ermittelt werden.

Die Ergebnisse der Methode `getTypeInfo` reichen aber nicht aus. Da die objektorientierte Zugriffsschicht die Daten aus der Datenbank aufnimmt, die Datentypen wie zum Beispiel `VARCHAR` in der Programmiersprache Java aber unbekannt sind, muß eine Abbildung von den Datenbank-Datentypen auf in Java bekannte Datentypen geschaffen werden.

Hierzu stellt das JDBC-Paket die Methode `getMetaData` der Klasse `ResultSet` zur Verfügung. Wie bereits erwähnt ist die Klasse `ResultSet` das JDBC-Mittel, um Tabellen darzustellen und zu durchlaufen. Bisher wurde das `ResultSet` nur verwendet, um allgemeine Informationen von der Datenbank zu erfragen. Auf

die tatsächlichen Daten der Datenbank wurde nicht zugegriffen. Um an die Java-Datentypen für die Repräsentation der Daten auf objektorientierter Seite heranzukommen, muß ein `ResultSet`-Objekt erzeugt werden, das die Struktur der relationalen Schemata aus der Datenbank aufweist. Eine einfache Anfrage in Form des SQL-Statements

```
select * from <table>
```

genügt, um die nötige Struktur zu erhalten.

Über den Aufruf der Methode `getMetaData` auf dem `ResultSet`-Objekt erhält man ein Objekt der Klasse `java.sql.ResultSetMetaData`. Dieses Objekt bietet die Methode `getColumnClassName` an, über die man den vollqualifizierten Java-Datentypen als String erhält.

SQL92 - Standard

Bisher wurde der SQL92-Standard immer als größtmögliche Eigenschaften-Menge von relationalen Datenbanken verstanden. Das ist aber falsch. Der SQL92-Standard bietet zwar sicherlich für die Realisierung einer relationalen Datenbank, wie dem Microsoft SQL-Server oder Postgres, ein gute Grundlage, er ist aber in der praktischen Anwendung eher unvollständig.

Die einfache Anforderung ,,Gib mir das erste Tupel aus einer Menge" verstößt zwar gegen die harte Auslegung der Definition des relationalen Datenmodells, da dort keine Ordnung über die Tupel definiert wird; sie ist aber auch ein in der Praxis immer wieder auftauchendes Problem, wenn es darum geht, die Anforderung dahingehend zu kontrollieren, daß es eine nicht leere Menge dazu gibt, aber die tatsächliche Ausprägung der Menge nicht von Interesse ist.

Über den SQL92-Standard ist das vorgestellte Problem nicht realisierbar. Die beiden oben vorgestellten Datenbanken haben ihre Umsetzung des SQL92-Standards dahingehend erweitert, daß die vorgestellte Anforderung möglich ist. Bei der Datenbank Postgres lautet die Anfrage:

```
select * from <table> limit 1
```

Die entsprechende Anforderung in dem Microsoft SQL-Server Dialekt heißt:

```
select top 1 * from <table>
```

Die datenbankspezifischen Erweiterungen des SQL92-Standards sind über das JDBC-Paket nicht erhältlich und demnach ist eine automatische Erkennung aller Eigenschaften einer Datenbank nicht möglich.

Es wird sich aber im späteren Verlauf der Arbeit noch herausstellen, daß die objektorientierte Zugriffs-Schicht durch die Verwendung als Cache sehr viele Funktionalitäten dem Benutzer zur Verfügung stellt, ohne sie direkt in Datenbank-Anforderungen umsetzen zu müssen. Die oben vorgestellte Anforderung könnte zum Beispiel im Zusammenhang einer Kontrolle verwendet werden, ob in einer Tabelle ein Eintrag enthalten ist oder nicht.

In der Datenbank dsd gibt es zum Beispiel die Tabelle `products`, die im wesentlichen alle Dokumente enthält, die in DSD unterstützt werden. Der Code für die Kontrolle, ob in der besagten Tabelle überhaupt ein Eintrag vorliegt oder nicht, könnte über JDBC auf der Datenbank Postgres wie folgt aussehen:

```
...
Connection connection = ... ;
Statement statement = connection.
    createStatement(,,select * from products limit 1'');
ResultSet result = statement.executeQuery(statement);
if ( result.next() )
{
    ...
}
...
```

Dabei wird der durch die if-Abfrage geschachtelte Quell-Code nur ausgeführt, wenn mindestens ein Eintrag in der Tabelle `products` vorhanden ist.

Die noch später genauer vorgestellte Zugriffs-Schicht für die Datenbank dsd enthält ein global bekanntes Manager-Objekt (`DSDPersistentManager`), über das man zum Beispiel alle DSD-Dokumente erhalten kann (`iteratorOfProducts()`). Neben dieser Methode bietet das Manager-Objekt auch die Methode `sizeOfProducts()` an, über die man die Anzahl der Dokumente erhält. Beide Methoden könnten verwendet werden, um das oben vorgestellte Problem ebenfalls zu lösen. Der folgende Code-Ausschnitt mit der Verwendung der Methode `sizeOfProducts()` bietet die selbe Funktionalität wie der oben verwendete Code-Ausschnitt:

```
...
if ( DSDPersistentManager.get().sizeOfProducts() > 0)
{
    ...
}
...
```

Die Einsparung mehrerer Zeilen in der zweiten Variante soll hier nicht als Vorteil dargestellt werden. Auch die erste Variante kann ohne objektorientierte Zugriffsschicht durch Einführung von Verwaltungs-Objekten und Methoden auf die drei angegebenen Zeilen reduziert werden. Dieses Beispiel macht aber deutlich, daß auch ohne die Verwendung der speziellen Datenbank-Dialekte die Funktionalität der objektorientierten Zugriffs-Schicht nicht unbedingt eingeschränkt ist. In dem oben angegebenen Beispiel würde die objektorientierte Zugriffs-Schicht zuerst zwar alle Dokumente aus der dsd-Datenbank laden und dann erst die Anzahl der Products-Objekte ermitteln und dem Benutzer zurückgeben. Der Datenbank-Manager würde aber auch die Objekte behalten bzw. cashen, sodaß weitere Zugriffe auf die Datenbank zum Erhalt von anderen Products-Objekten überflüssig werden.

Hier ist also nicht die Geschwindigkeit innerhalb eines Methoden-Aufrufes entscheidend, sondern die Frage, wie oft später in der Applikation bzw. dem System noch auf die Dokumente von dsd zugegriffen wird. Durch das dann sofortige Zurückliefern der gespeicherten Objekte werden in der Laufzeit teure Anfragen an den Datenbank-Server vermieden.

Das alleinige Speichern der ResultSets ohne weitere objektorientierte Zugriffsschicht, das als Lösungsalternative für die alleinige Nutzung von JDBC herangezogen werden könnte, würde weitere Anforderungen an die dahinter befindlichen Datenbanken und JDBC-Treiber stellen, die nicht von jeder Datenbank und nicht von jedem Treiber erfüllt werden. Die aufgesetzte objektorientierte Zugriffsschicht stellt also eine Zugriffs-Möglichkeit auf eine Datenbank zur Verfügung, die weniger von der Datenbank voraussetzt als der direkte Einsatz von JDBC.

Die objektorientierte Zugriffsschicht bietet also Alternativen an, die die Belange des Benutzers abdecken, aber nicht die Datenbank tangieren. Damit müssen also nicht alle Eigenschaften einer Datenbank über die objektorientierte Zugriffsschicht abgedeckt werden, sodaß sich die später tatsächlich gebrauchten SQL-Aufrufe auf ein ausgewähltes Minimum reduzieren, das über REDDMOM für jede einzelne Datenbank konfigurierbar und erweiterbar ist.

Falls erkannt wird, daß die hier vorgestellte Zugriffsschicht nicht mehr ausreicht, kann sie durch die Anpassung des Generators und der Konfigurations-Möglichkeiten von REDDMOM erweitert werden. Hierbei sollte aber nicht das unmögliche Absetzen eines „wilden" SQL-Aufrufes herangezogen werden, sondern die Unmöglichkeit der Spezifikation in UML einer alternativen Darstellung.

4.1.2. Relationale Attribute und ihre Datentypen

Von der Methode `getTypeInfo` der Klasse `java.sql.DatabaseMetaData` konnten die maximalen Werte zu einem Datentypen ermittelt werden. Zum Beispiel erlaubt der Datentyp `CHAR` von dem Datenbank-Server Frontbase die Speicherung

von weit über 2 Mrd. Zeichen in einem Attribut. Der Datentyp `CHAR` unterscheidet sich von dem bereits vorgestellten Datentyp `VARCHAR` darin, daß das zu ihm gehörende Attribut in jedem Tupel des relationalen Schemas seine in der Definition angegebene Länge einnimmt - bei der Benutzung der maximalen Länge also 2 Mrd Zeichen pro Tupel. Diese Definition macht natürlich nur Sinn, wenn sichergestellt ist, daß das Attribut in jedem Tupel auch wirklich diesen Platz benötigt. In der Praxis wird ein Attribut vom Typ `CHAR` mit weit weniger Platz auskommen, sodaß es bei der Definition des Attributs möglich sein muß, diese Länge auf einen passenden, normalerweise wesentlich kleineren Wert zu reduzieren.

Bei der Anlage eines Attributs mit einer eingeschränkten Länge wird die Länge von dem Datenbank-Server als Constraint verstanden. Die Länge darf von keinem Tupel überschritten werden. Der Server könnte bei der Einhaltung dieses Constraints zwei Strategien verfolgen. Zum einen könnte er bei Ausführung einer Transaktion die Länge des zu speichernden Wertes mit der eingeschränkten Länge vergleichen und bei einer Überschreitung die Transaktion abbrechen (abort). Zum anderen könnte er aber auch den Überschuß einfach ignorieren und die Transaktion weiterführen.

Die Objekte in der objektorientierten Zugriffsschicht sind reine Java-Objekte und ihre Attribute besitzen Java-Datentypen. Java bietet keine Definition an, die Attribute in ihrem Wertebereich zu beschränken. Somit steht zum Beispiel einem Datenbank-Attribut vom Typ `VARCHAR` mit der beschränkten Länge von 10 Zeichen ein `String`-Attribut aus Java gegenüber, welches die Speicherung von wieder mehr als 2 Mrd. Zeichen erlaubt. Da die Änderung zunächst über das Objekt in der objektorientierten Zugriffsschicht erfolgt, besteht die Gefahr, daß die Länge des String-Attributes die Länge des Datenbank-Attributes übersteigt.

Wie noch in Kapitel 5.3.1 gezeigt wird, besitzen die Objekte der objektorientierten Zugriffsschicht ein eigenes Transaktions-Konzept. Ihre Änderungen werden über eine Transaktion festgehalten und sie können sich zu jeder Zeit in den Status vor Transaktions-Beginn zurückversetzen.

Die erste von dem Server eingeschlagene Variante bei einer Längenüberschreitung würde keine Probleme verursachen. Die objektorientierte Zugriffsschicht erhält die Nachricht, daß sie sich zurücksetzen muß, und führt auf sich den abort der Transaktion aus.

Die zweite Variante verursacht Schwierigkeiten.

Die hier vorgestellte objektorientierte Zugriffsschicht soll einen Cache darstellen, wie er in Kapitel 2.3.3 beschrieben ist. Sie soll also die Daten aus der Datenbank widerspiegeln und bei Verlangen von Daten zuerst die eventuell bereits im Cache befindlichen Daten zurückliefern, ehe sie den zeitintensiven Weg zu der Datenbank einschlägt. Da der Server bei der zweiten Variante unter Umständen die Transaktion erfolgreich beendet, beendet sich auch die objektorientierte Zugriffsschicht erfolgreich. Das Objekt, welches das zu lange Datum beinhaltet, geht

bei einem commit davon aus, daß seine Daten richtig sind und stellt ab sofort diese Daten zur Verfügung. Da das Datum beim Abspeichern durch den Server abgeschnitten wurde, spiegelt es nicht den korrekten Sachverhalt wider, der in der Datenbank vorliegt.

Von einer Transaktion wird auch der Anspruch der Dauerhaftigkeit der Daten erwartet. Bei einer möglichen Neuinitialisierung der objektorientierten Zugriffsschicht würden alle Cache-Objekte im Hauptspeicher gelöscht und durch die kommenden Anfragen neu aufgebaut werden. Dabei würde es passieren, daß ein Objekt entsteht, daß jetzt dem tatsächlichen Inhalt der Datenbank entspricht. Aus Betrachtung des Benutzers wäre dies ein Verstoß gegen die Gewährleistung der Dauerhaftigkeit, da er davon ausgegangen ist, den zu langen Wert des Attributes zu erhalten.

Zur Lösung des Konflikts ergeben sich zwei Möglichkeiten:

1. *Auslesen der Daten nach Transaktions-Ende*: Hier würde sich die objektorientierte Zugriffsschicht nach jedem Transaktionsende erneut aufbauen.

2. *Aufnahme der Längen-Constraints in das logische Schema*: Die Längenangaben der relationalen Attribute werden in das logische Schema mit aufgenommen. Bei der späteren Generierung der Zugriffsschicht werden diese Constraints beachtet und die Möglichkeit der Speicherung längerer Daten in einem Objekt ausgeschlossen.

Der Vorteil der ersten Alternative ist, daß auf eine Sicherung der vor der Transaktion gültigen Daten in der objektorientierten Zugriffsschicht verzichtet werden könnte, da sie ja immer den aktuellen und damit konsistenten Zustand aus der Datenbank ausliest. Der Nachteil ist aber, daß durch diese pessimistische Haltung der Zugriffsschicht der eigentliche Vorteil des Caches nur noch eingeschränkt besteht, da durch die permanente Anfrage der neuen Daten der Netzwerk-Verkehr wieder steigt, der, wie bereits erwähnt, in der Laufzeit teuer ist.

Die zweite Alternative greift auf die Definition des bekannten Algorithmus- Begriffs zurück. Ein Algorithmus besteht dabei aus einer Menge von Elementar-Operationen, die nach einer Vorschrift durchlaufen werden. Außerdem besitzt der Algorithmus Vor-, Zwischen- und Nachbedingungen, die eindeutig bestimmbar sind. Eine Methode ist auch ein Algorithmus. Die schreibende Zugriffsmethode zu einem Attribut hat im allgemeinen die folgende Signatur:

```
public void setAttribute ( <Attribute-Type> newAttributeValue )
```

Die Vorbedingung in der Sprache Java ist hierbei, daß die Variable `newAttributeValue` unbedingt vom Typ `Attribute-Type` sein muß.

Eine herkömmliche `set`-Methode würde in ihrem Methoden-Rumpf dem Attribut des Objektes, für welches sie zuständig ist, den neuen Wert `newAttributeValue` zuweisen. Durch die Erweiterung der Vorbedingung um die Constraint- Überprüfung, ob die Länge des neuen Wertes kleiner oder gleich der Maximal-Länge des Datenbank-Attributes ist, kann der oben erwähnte Konflikt bereits in der objektorientierten Zugriffsschicht ohne erneutes Lesen aus der Datenbank beseitigt werden.

Der Methodenrumpf der `set`-Methode könnte dann wie folgt aussehen:

```
public void setName (String newName ) throws IllegalDataException
{
    if ( newName == null )
    {
        throw new IllegalDataException();
    }
    if ( newName.length() > 40 )
    {
        throw new IllegalDataException();
    }
    ...
}
```

In dem Beispiel besitzt die Klasse, in der diese Methode spezifiziert wird, ein Attribut `name`. Das zu dem Attribut korrespondierende relationale Attribut erlaubt als Wert keine `null`-Werte und ist in seiner Länge auf 40 Zeichen beschränkt.

Die erste Fall-Abfrage der `set`-Methode verhindert das Setzen der `null`-Werte und die zweite Abfrage ist für die Längen-Kontrolle zuständig. Diese Zugriffs-Methode verhindert das Abspeichern ungültiger Werte für das relationale Attribut. Da die Überprüfung dieser Attribut-Constraints den Aufwand von O(1) hat, ist die Constraint-Überprüfung innerhalb der objektorientierten Zugriffsschicht sinnvoller als das neue Auslesen der Datenbank-Informationen.

Für das Auslesen der logischen Struktur bedeutet das, daß auch die definierten Attribut-Längen mit aufgenommen werden müssen und später dem Generator zur Verfügung stehen. Auch diese Werte können für jedes einzelne Attribut der relationalen Schemata von der im Kapitel 4.1.1 vorgestellten Klasse `java.sql.ResultSetMetaData` erhalten werden.

4.1.3. Anreicherung der Struktur

In den Kapiteln 3.2 und 2.1.3 wurde bereits angedeutet, daß das alleinige Auslesen der Datenbank-Struktur gewöhnlich nicht ausreicht. Das aus dem logischen

Schema entstehende konzeptionelle Schema soll keine Einschränkungen in den Eigenschaften des Objektmodells erhalten. Damit muß zum Beispiel auch gewährleistet sein, daß die aus der Objektorientierung bekannte Vererbungsbeziehung auf das relationale Modell abbildbar ist.

Diese Informationen können nicht von der Datenbank erhalten werden, weil sie die Definition einer Vererbung nicht kennt. Auf der anderen Seite wiederum repräsentieren auch die in der Datenbank festgehaltenen Daten eine gewisse Umwelt. Sie stellen Personen, Adressen, etc. dar, je nachdem, wofür die Tabellen und die Daten bestimmt sind.

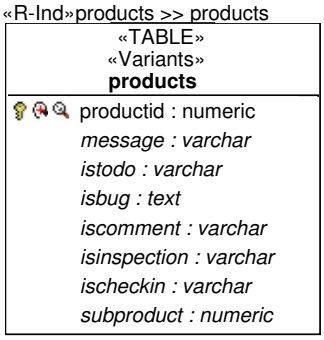


Abbildung 4.3.: Das relationale Schema ,,products”

Die in Kapitel 4.1.1 bereits vorgestellte und in der Abbildung 4.3 gezeigte Tabelle `products` aus der Beispieldatenbank dsd stellt die Dokumente zur Verfügung, die in dem Projekt DSD verwaltet werden können. Auffallend in der Struktur der Tabelle `products` sind die Attribute `istodo`, `isbug`, `iscomment`, `isinspection` sowie `ischeckin`. Alle Attribute erlauben `null`-Werte, was in der Abbildung daran zu erkennen ist, daß sie kurisv dargestellt werden. Der Datentyp `text` des Attributs `isbug` erlaubt wie der Datentyp `varchar` der anderen hier genannten Attribute das Abspeichern von alphanumerischen Zeichen. Die Unterschiede in der Definition dieser beiden Datentypen ist hier nicht wichtig. Die definierte Länge der Attribute ist jeweils 1, und die Attribute sollen praktisch True/False-Flags simulieren. Dabei soll der spezielle Wert `null` für false (ist nicht) und ein beliebiger anderer Wert für true (ist) stehen.

Ohne Berücksichtigung der Attribut-Namen ist aus der Struktur nicht zu entnehmen, daß das Setzen eines dieser Attribute auf einen Wert ungleich `null` die Eigenschaften, die die anderen Attribute darstellen, ausschließt. Das Dokument vom Typ `Bug` (`isbug = true`) kann nicht auch gleichzeitig ein Kommentar (`iscomment = true`) sein. Man hat also eine Tabelle für einige Dokumente, und die Art des Dokuments kann aus der Spalte gefolgert werden, die ungleich `null` ist.

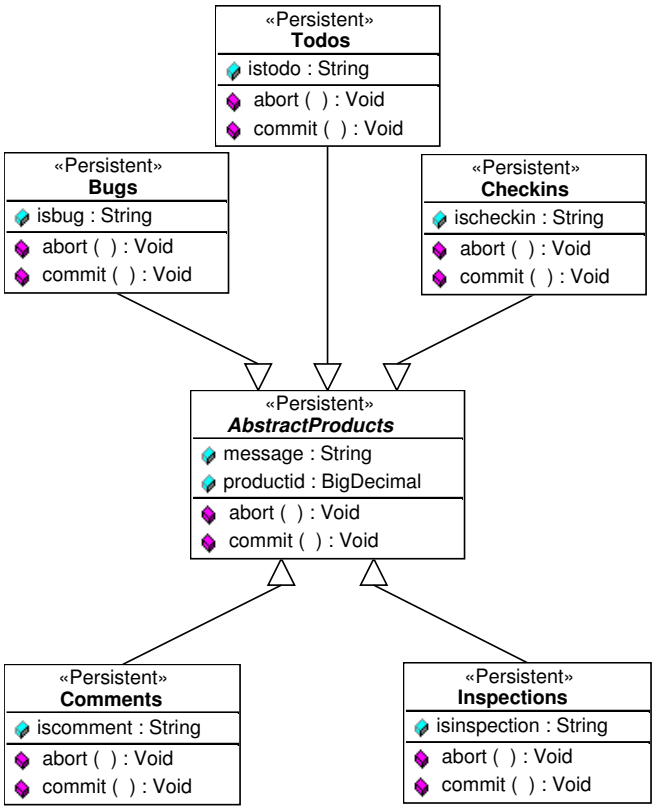


Abbildung 4.4.: Klassenhierarchie ,,products”

Da sich die Attribute gegenseitig ausschließen, würde sich auf objektorientierter Seite eine Vererbungsbeziehung anbieten. Dabei würde es eine Klasse `Products` geben, von der die unterschiedlichen Dokument-Arten erben.

Die Abbildung 4.4 zeigt die gewünschte Klassenhierarchie mit der noch dazukommenden speziellen Eigenschaft, daß die Vaterklasse aller Dokumente `abstract` ist und demnach nicht `Products`, sondern `AbstractProducts` heißt. Damit muß ein Dokument in DSD als `Bug`, `Checkin` oder `Comment` definiert werden, da die Erzeugung eines abstrakten Objektes nicht möglich ist. Da in diesem Beispiel vorausgesetzt wird, daß die einzelnen Dokumentarten sich gegenseitig ausschließen, müssen auch die in dem relationalen Schema `products` enthaltenen Tupel diesen Sachverhalt widerspiegeln. Es darf kein Tupel geben, das in mehr als einer der genannten Spalten einen Wert ungleich `null` enthält.

Das VARLET-Projekt, von dem in dieser Diplomarbeit einige Verfahren übernommen wurden, beinhaltet auch ein Analyse-Werkzeug, das die hier vorgestellte Eigenschaft der Attribute automatisch überprüft und erkennt. Das Analyse-Werkzeug erkennt anhand aller Ausprägungen der `products`-Tabelle, daß das Setzen eines der hier genannten Attribute das Setzen der anderen Attribute ausschließt.

So werden die aus dem Kapitel 2.1.3 bereits bekannten Varianten, die durch Exclusion-Dependencies definiert werden, ermittelt. Die Tabelle 4.2 beinhaltet

Nicht null Attribut	**Daraus folgende null-Attribute**
`istodo`	`isbug,iscomment,isinspection,ischeckin`
`isbug`	`istodo,iscomment,isinspection,ischeckin`
`iscomment`	`istodo,isbug,isinspection,ischeckin`
`isinspection`	`istodo,isbug,iscomment,ischeckin`
`ischeckin`	`istodo,isbug,iscomment,isinspection`

Tabelle 4.2.: Exclusion Dependencies der Tabelle ,,products”

alle in der Tabelle `products` enthaltenen Exclusion Dependencies. Für jede in dem relationalen Schema enthaltene Exclusion-Dependency wird eine Variante angelegt, die einen Verweis auf das zu der Exclusion-Dependency gehörende `Nicht-Null` Attribut hat. Das `Nicht-Null` Attribut ist Element der Variante. So entstehen in diesem Beispiel fünf Varianten zu der Tabelle `products`. Bei der Definition einer Variante zu einer Tabelle muß die Variante alle Attribute beinhalten, die keine `null`-Werte erlauben, oder über die keine Angabe gemacht werden kann, ob sie `null` sind oder nicht.

Die Variante muß also alle Attribute beinhalten, die nicht an der Exclusion-Dependency beteiligt sind. Die Abbildung 4.5 zeigt die vollständige Definition der Variante `Bugs` der Tabelle `products`, die die Attribute `productid`, `message`,

`subproductid` und `isbug` enthält, aber nicht die durch das Attribut `isbug` ausgeschlossenen Attribute `istodo`, `iscomment`, `isinspection` und `ischeckin`.

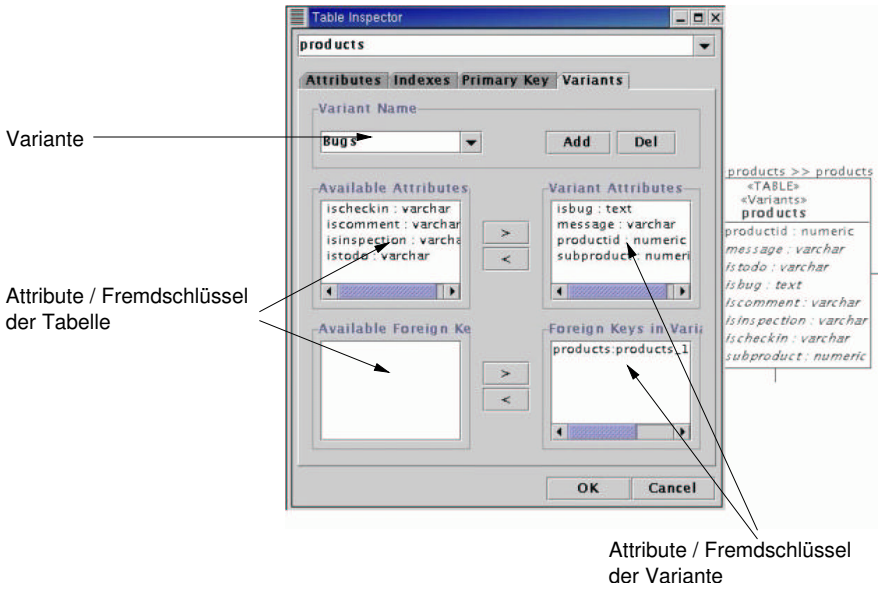


Abbildung 4.5.: Varianten-Definition am Beispiel von ,,products"

Die durch eine Variante ausgeschlossenen Attribute können Bestandteil eines in der Datenbank definierten Fremdschlüssels zu einer Inklusionsabhängigkeit sein. Da zumindest ein Attribut von dem Fremdschlüssel in der Varianten-Definition ausgeschlossen wurde, kann der Fremdschlüssel nicht mehr Bestandteil dieser Variante sein und muß ebenfalls ausgeschlossen werden. Erst, wenn in einer Variante alle Attribute eines Fremdschlüssels enthalten sind, darf auch der Fremdschlüssel in die Variante aufgenommen werden. Da das Attribut `subproductid` als einziges Attribut den in Abbildung 4.5 gezeigten Fremdschlüssel definiert, und das Attribut in der Variante `Bugs` enthalten ist, kann auch der Fremdschlüssel in der Variante enthalten sein. Es wäre aber auch möglich, ihn bei der Definition der Variante `Bugs` auszuschließen, was aber Auswirkungen auf das erzeugte initiale Schema hätte und hier nicht weiter erläutert werden soll.

Derzeit ist das Analyse-Werkzeug von VARLET noch nicht in REDDMOM enthalten, sodaß nur eine manuelle Ermittlung der Varianten möglich ist. Das Werkzeug wird aber im weiteren Bestehen von REDDMOM noch integriert.

Es erscheint merkwürdig, daß in dem hier vorgestellten Beispiel die Attribute von dem Typ `varchar(1)` sind und nicht von einem Datentypen, der dem aus Java bekannten `java.lang.Boolean` entspricht. Außerdem bräuchten die Attribute der Exclusion-Dependencies in diesem Fall überhaupt nicht in das objektorientierte Schema übernommen zu werden, da die Klassentypen der Objekte bereits aussagen, welche Dokumentart sie repräsentieren. Es wurden aber bisher nur die

Verfahren von VARLET übernommen und nicht erweitert bzw. ergänzt. Derzeit können die von VARLET bereitgestellten Analyse-Möglichkeiten nur echte Exclusion-Dependencies erkennen, sodaß die Attribute `null`-Werte zur Erkennung aufweisen müssen.

Ein Datentyp einer Datenbank, der dem `java.lang.Boolean` entsprechen würde, könnte aber explizit den `null`-Wert als gültigen Wert verbieten und nur zwei Werte (`true/false`) erlauben. Der während dieser Diplomarbeit vorwiegend eingesetzte Datenbank-Server Sybase Adaptive Server 11.9.2 bietet dafür den Datentypen `bit` an, der nur die Werte 0 (`false`) und 1 (`true`) erlaubt. Die Analyse zur Erkennung der Varianten könnte demnach dahingehend erweitert werden, daß bei dieser speziellen Art des Datentypen nicht nach `null`-Werten gesucht wird, sondern nach 0. Außerdem müßten die Attribute dieser besonderen Exclusion-Dependency nicht in das konzeptionelle Schema übernommen werden, da die aus den Varianten entstehenden Klassen bereits die Eigenschaften des Attributes darstellen.

4.2. Konfiguration von REDDMOM

Das Projekt REDDMOM hat unter anderem zum Ziel, die Entwicklung verteilter Datenbanken zu unterstützen. Diese Diplomarbeit beschränkt sich im Titel auf eine Datenbank. Es wurde aber in der Implementierung erkannt, daß die Datenbanken in ihren Eigenschaften zwar ähnlich sind, aber bei detailierterer Betrachtung so stark auseinandergehen, daß eine spätere Unterstützung von mehreren Datenbanken in einem System sehr erschwert würde, wenn nicht frühzeitig Konflikte zwischen Datenbanken erkannt und behoben werden: Gleichnamige Datentypen sagen unterschiedliches aus, die Anmeldungsart bei der Datenbank ist nicht immer gleich, spezielle Datentypen benötigen eine spezielle Behandlung, usw.

Diese speziellen Eigenschaften müssen natürlich hinterher Bestandteil der objektorientierten Zugriffsschicht sein. Auf der anderen Seite soll die objektorientierte Zugriffsschicht so wenig wie möglich von der Datenbank abhängen. Ein Großteil der Zugriffsschicht muß datenbankunabhängig und immer gleich einsetzbar sein, damit eine Überprüfung der Korrektheit der Zugriffsschicht erleichtert wird. Die Verwendung und der innere Ablauf der Zugriffsschicht darf nicht von der Datenbank beeinflußt werden, sondern immer gleich und damit datenbankunabhängig testbar sein muß. Der datenbankabhängige Teil muß so spät wie möglich zum Einsatz kommen und austauschbar sein, damit die allgemeine Datenbankunabhängigkeit der Zugriffsschicht überhaupt erfüllbar ist.

Deshalb hat sich diese Diplomarbeit zum Ziel gesetzt, eine nach außen hin datenbankunabhängige Zugriffsschicht zu erzeugen, die den datenbankabhängigen Teil so stark kapselt, daß er durch eine andere datenbankspezifische Definition

für das selbe System theoretisch austauschbar ist. Derzeit ist es nicht möglich, über REDDMOM eine Datenbank auf einem Datenbank-Server anzulegen. Bei der Generierung der Zugriffsschicht wird aber so getan, als ob es zu jedem, in REDDMOM bekannten, Datenbank-System eine reale Datenbank mit der Struktur gibt, für die gerade die Zugriffsschicht erzeugt wird. Durch manuelles Erzeugen der in REDDMOM vorgegebenen Struktur der Datenbank, könnte die Zugriffsschicht auch auf dieser Datenbank eingesetzt werden.

Dadurch, daß die Datenbanken unterschiedliche Eigenschaften aufweisen, aber im wesentlichen das selbe leisten, müssen die Eigenschaften in REDDMOM konfigurierbar und aufeinander abbildbar sein. Die Leistungsfähigkeit wird hier mehr von dem System vorgegeben als von der Datenbank.

Eine Bankapplikation muß über einen hohen Sicherheitsgrad verfügen, der nur durch ein entsprechendes Transaktionssystem der Datenbank erfüllt werden kann. Bei der Adress-Verwaltung einer Privatperson ist dieser Sicherheitsgrad eher nicht von belang, sodaß auch transaktionsschwächere Datenbanken eingesetzt werden können.

Die Abbildung 4.6 zeigt einen Ausschnitt aus der konfigurierbaren Logik von REDDMOM. Der Ausschnitt befaßt sich mit der Konfiguration der Abbildung der Datentypen aus den unterschiedlichen Datenbanken.

Wenn die Applikation REDDMOM gestartet wird, wird unter anderem auch das Singleton-Objekt `SQLSettings` initialisiert, und es lädt aus einer Konfigurationsdatei alle bereits bekannten Datenbank-Einstellungen. Die Konfigurationsdatei liegt in einem XML-Format vor. Dessen Struktur ist aber für die Vorstellung der Konfiguration unwesentlich. Die Einstellungen zu einem Datenbankserver werden bekannt, nachdem zum ersten Mal erfolgreich eine Verbindung zu einer Datenbank des Servers aufgebaut wurde und die Meta-Informationen gelesen werden konnten.

Jedes bekannte Datenbank-Management-System ist intern ein `SQLDatabase`- Objekt. Zum Beispiel gibt es ein `SQLDatabase`-Objekt für den bereits öfter erwähnten Sybase Adaptive Server 11.9.2 und eines für die Datenbank MySQL. Bei dem Auslesen der Meta-Informationen des Datenbank-Management-Systems erzeugt jeder von dem Management-System unterstützte Datentyp ein `SQLDatatype`-Objekt. Alle so erhaltenen Datentypen werden über die Assoziation `datatypes` bei dem zu dem Management-System gehörenden `SQLDatabase`-Objekt eingetragen. Da jedem in REDDMOM bekannten Management-System ein eigenes `SQLDatabase`-Objekt zugeordnet ist, kann es nicht vorkommen, daß sich eventuell gleichnamige Datentypen aus unterschiedlichen Datenbanken gegenseitig überschreiben.

Der später vorgestellte Zugriff auf die Datenbank soll datenbankunabhängig sein. Der Entwickler einer Applikation bzw. eines Systems soll die Möglichkeit erhalten, sich nicht auf eine Datenbank festzulegen. Er soll die Eigenschaften des

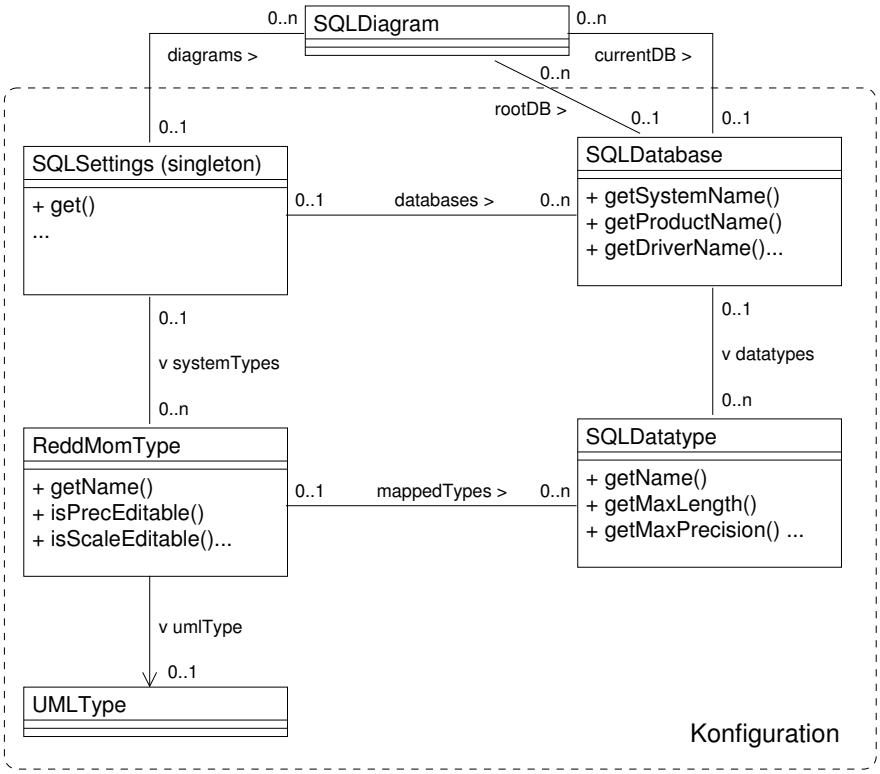


Abbildung 4.6.: Logik zum Abspeichern der datenbankspezifischen Einstellungen

Datenbank-Systems festlegen und REDDMOM soll erkennen, welche Datenbanken diese Eigenschaften erfüllen und welche nicht.

Da sich diese Diplomarbeit nur mit dem Auslesen der logischen Struktur einer konkreten Datenbank befaßt, muß es also möglich sein, diese so weit wie nur möglich auf eine andere Datenbank abzubilden. Entstehende Konflikte sollen während der Entwicklung des logischen Schemas sichtbar sein. Da das auf einem EER-Diagramm in REDDMOM zu sehende logische Schema die Struktur einer real existierenden Datenbank darstellt, und es später in der weiteren Entwicklung von REDDMOM auch möglich sein soll, diese Struktur über REDDMOM direkt zu ändern, muß die Entwicklungsmöglichkeit auf die konkrete Datenbank eingeschränkt werden.

Das EER-Diagramm wird in der Logik von REDDMOM durch ein `SQLDiagram`-Objekt repräsentiert. Über die `rootDB`-Assoziation wird die Datenbank festgelegt, die die Einschränkungen in der Entwicklung vorgibt.

Wenn der Sybase Adaptive Server 11.9.2 als `rootDB` gesetzt ist, wird verhindert, daß zum Beispiel ein `VARCHAR`-Attribut die Länge 255 übersteigt, weil das der Maximalwert für dieses Management-System ist. Falls die Datenbank Frontbase gewählt wurde, wird bei der Definition eines `VARCHAR`-Attributes mit zum Beispiel 300 Zeichen das mit dieser Länge unmögliche Übernehmen auf das Management-System Sybase Adaptive Server 11.9.2 kenntlich gemacht. Bei so einem Konflikt kann das Management-System entweder aus der Entwicklung genommen werden, oder es wird die Leistungsfähigkeit der Applikation für das System eingeschränkt, das den Konflikt verursacht hat. In dem obigen Beispiel würde das bedeuten, daß die Länge des Attributes in dem System 300 Zeichen bleibt - mit Ausnahme des besagten Servers, wo die Länge dann nur noch 255 Zeichen ist. Um die Darstellung des Systems aus Sicht von unterschiedlichen Datenbanken zu ermöglichen, steht die `currentDB`-Assoziation aus Abbildung 4.6 zur Verfügung.

Wenn der Sybase Server als aktuelle Datenbank ausgewählt wurde, wird der oben beschriebene Konflikt durch Hervorhebung des Attributs in REDDMOM kenntlich gemacht. Die Verwaltungsmöglichkeit unterschiedlicher Längen zu einem Attribut und die Möglichkeit, Datenbanken von der Entwicklung auszuschließen, wurden bisher nicht realisiert.

Es müssen also die Datentypen der einen Datenbank auf entsprechende Datentypen aus einer anderen Datenbank abgebildet werden können. Um dies zu realisieren, wird die in Abbildung 4.6 gezeigte Klasse `ReddMomType` eingefügt. Diese Klasse ist eine Konfigurations-Klasse, und ihre Objekte können in REDDMOM frei konfiguriert werden. Vorwiegend besitzt diese Klasse einen Namen, über den die Objekte bei der Konfiguration identifiziert werden können. Über die Assoziation `mappedTypes` kann jedem realen Datentypen genau ein Konfigurations-Datentyp zugeordnet werden.

Dabei muß nicht jedem Datentyp aus einer Datenbank ein eigener Konfigurations-Datentyp zugeordnet werden. Es können auch mehrere Datentypen über einen Konfigurations-Datentyp zusammengefaßt werden. Bei der Konfiguration ist es nicht entscheidend, ob die Datentypen identisch sind, sondern ob sie generell gleiche Werte abspeichern können. Die bereits vorgestellten Datentypen `TEXT`, `CHAR` und `VARCHAR` sind alles relationale Datentypen, die Textinformationen aufnehmen können und die später auf den Java-Datentypen `java.lang.String` abgebildet werden.

Für die spätere Zugriffsschicht ist nicht der Datentyp der Datenbank interessant, sondern der Datentyp, der ihr in Java für den Zugriff auf das Attribut zur Verfügung steht. Für die Zuordnung des Java-Datentypen besitzt die Konfigurations-Klasse `ReddMomType` die Referenz auf `UMLType`, die in dem Meta-Modell von FUJABA generell einen Java-Typen definiert. Ein Typ kann in diesem Zusammenhang ein Basistyp wie `int`, `char`, `float` usw. sein aber auch eine Klasse wie `java.lang.String`, `java.math.BigDecimal` usw.

Dadurch, daß jedem in REDDMOM bekannten Datentyp ein Konfigurations-Datentyp zugeordnet wird, werden die Datentypen aus verschiedenen Datenbanken über den Konfigurations-Datentyp gruppiert. So kann zu einer ausgelesenen Struktur aus einer Datenbank eine Darstellung für eine andere Datenbank erstellt werden, indem zu den Datentypen der einen Datenbank über die Konfigurations-Datentypen entsprechende Datentypen der anderen Datenbank ausgesucht werden.

Die daraus resultierende Darstellung soll als initial verstanden werden. Die hier vorgestellte Konfiguration soll zunächst nur die vorhandene Struktur in eine Struktur für eine andere Datenbank transformieren. Die Transformation soll nicht als optimal verstanden werden. Dadurch, daß mehrere Datentypen aus einer Datenbank auf einen Konfigurations-Datentyp abgebildet werden, passiert es bei der Transformation auf diese Datenbank, daß aus den Datentypen einer ausgewählt werden muß. Bei der automatischen Auswahl soll eine möglichst hohe Trefferquote entstehen. Die transformierten Datentypen sollen weitestgehend mit den originalen Datentypen übereinstimmen und im Einzelfall soll es möglich sein, von dem Resultat der Transformation abzuweichen und sich aus der Datentyp-Gruppe einen anderen Datentypen auszusuchen.

Derzeit ist die Kennzeichnung eines Datentypen aus solch einer Gruppe so möglich, daß er bei der Transformation als Standard-Definition verwendet werden soll.

Bei den Datentypen `TEXT`, `CHAR` und `VARCHAR` könnte dies zum Beispiel `VARCHAR` sein, und es ist sicherlich plausibel, daß diese Konfiguration bei einer Transformation eine Darstellung erzeugt, wo kaum die Textattribute noch angepaßt werden müssen.

Anders sieht das hingegen bei dem Datentyp `NUMERIC` aus. Sybase Adaptive Server 11.9.2 erzeugt bei einem `identity`-Attribut, was im wesentlichen ein `NUMERIC`-

Attribut mit der bereits bekannten `autoincrement`-Einstellung ist, Werte vom Typ `java.math.BigDecimal`.

Wie man es erwartet hat, erhält man also Objekte, mit denen man rechnen kann. Man erhält Zahlen. Ein gewöhnlich definiertes `NUMERIC`-Attribut hingegen, das auch Bestandteil eines Index ist, erscheint plötzlich als `java.lang.String`, mit dem bekannterweise nicht gerechnet werden kann. Der Konflikt entsteht hier, daß das `NUMERIC`-Attribut und das `VARCHAR`-Attribut in der Konfiguration unterschiedlichen Gruppen angehören. Bei `NUMERIC`-Attributen will man generell in der objektorientierten Zugriffsschicht die numerischen Fähigkeiten ausnutzen und mit Objekten wie zum Beispiel `java.math.BigDecimal` arbeiten, mit denen man addieren, subtrahieren, usw. kann. `VARCHAR`-Attribute sollen aber bloß Zeichen darstellen und hier genügt ein `java.lang.String`.

Dieses Problem ist durch weitere Konfiguration behebbar. Diese Konfiguration gehört aber mehr zum Generator, sodaß im Kapitel 5 auf dieses Problem noch einmal eingegangen wird.

Die Möglichkeit der Abweichung von der Standard-Konfiguration wurde bisher noch nicht implementiert.

4.3. Erstellung des konzeptionellen Schemas

Wie bereits erwähnt, wird als konzeptionelles Schema das UML Meta-Schema von FUJABA verwendet. Auf eine genaue Beschreibung dieses Schemas wird hier verzichtet, und es wird dafür nur auf [FNT98, FUJ] verwiesen.

Für diese Arbeit ist es entscheidender, wie aus dem logischen Schema das konzeptionelle Schema erstellt werden kann. Dafür wurden die in dem VARLET-Projekt als Progres-Spezifikationen vorliegenden *Triple-Graph-Grammatik-Regeln* (TGG-Regeln) in dieser Arbeit neu implementiert.

Für die Neuimplementierung wurde der in FUJABA bereits existierende TGG-Editor verwendet. Dieser Editor ist im Rahmen der Diplomarbeit [Müh00] entstanden. Seine Funktionalität soll hier nicht weiter vorgestellt werden sondern nur das reine Prinzip der TGG-Regeln.

Für eine vollständige Beschreibung der Regeln wird auf [Jah99, VAR] verwiesen. Sie sollen in dieser Arbeit nur kurz vorgestellt werden, um die Verbindung zwischen dem logischen Schema und dem konzeptionellen Schema besser vorstellbar zu machen.

Diese Verbindung kann zur Konsistenzerhaltung der beiden Schemata genutzt werden, und die Konsistenzerhaltung wurde in VARLET im Rahmen der Diplomarbeit [Wad98] implementiert.

Die Konsistenzerhaltung ist nicht Bestandteil dieser Arbeit. Sie wird aber im Laufe der Entwicklung von REDDMOM nachgeholt. Die Regeln sollen hier aber auch erwähnt werden, weil der in Kapitel 5 vorgestellte Generator die Verbindung zwischen den beiden Schemata verwendet, um sich alle Informationen für die Generierung zusammenzusuchen.

Beispiel einer Triple-Graph-Grammatik

Die Erzeugung des initialen konzeptionellen Schemas besteht insgesamt aus neun Triple-Graph-Grammatik-Regeln, von denen jede mehr oder weniger komplex ist. Bei der Aufgabenstellung, ein logisches, relationales Schema in ein konzeptionelles, objektorientiertes Schema zu transformieren, versucht man zunächst, zu den relationalen Elementen entsprechende objektorientierte Elemente zu finden. Es ist naheliegend, eine relationale Tabelle auf eine objektorientierte Klasse abzubilden, da sowohl die Struktur der relationalen Tabelle als auch die Struktur der Klasse durch Attribute beschrieben wird.

Eine der neun Regeln ist für die Transformation einer Tabelle in eine Klasse zuständig. Sie ist einfach in ihrem Aufbau und für die Erklärung der TGG-Regeln ausreichend. Wie in Abbildung 4.7 zu sehen, teilen die vertikalen, gestrichelten Linien die Regel in drei Teile auf - in einen linken und einen rechten Teil und die Mitte. Die zu sehenden Kästen werden in der Definition der Triple-Graph-Grammatiken als Knoten bezeichnet und entsprechen in dem REDDMOM/FUJABA zugrundeliegenden Objektmodell einem Objekt.

Die Linien heißen in der Regel Kanten und definieren in der Objektorientierung eine Referenz zwischen zwei Objekten. Alle während der Laufzeit erzeugten Objekte spannen über ihre Referenzen zu anderen Objekten einen aus der Triple-Graph-Grammatik-Welt bekannten *Graphen* auf.

Eine Übersicht über den Transformationsgraphen, der durch die Transformationsregeln aufgespannt wird, ist im Anhang unter A.3 dargestellt.

Die horizontal gestrichelte Linie trennt die ,,nicht zu erzeugenden Knoten und Kanten" von den ,,zu erzeugenden Knoten und Kanten" besser ab. Dabei sind alle ,,zu erzeugenden Knoten und Kanten" mit einem ,,create" gekennzeichnet. Die ,,nicht zu erzeugenden Knoten und Kanten" über der horizontalen Linie gelten auch als Vorbedingung für diese Regel.

Diese eine TGG-Regel definiert gleich drei Transformationsregeln.

Jede Seite bekommt eine eigene Transformationsregel. Dabei gilt durch die Definition der TGG-Regeln, daß bei der Betrachtung einer bestimmten Seite der Regel die ,,zu erzeugenden Knoten und Kanten" nicht erzeugt, sondern mit in die Vorbedingung aufgenommen werden. Hier macht die Mitte eine Ausnahme und definiert die Vorbedingung genau anders herum. Die linke und die rechte

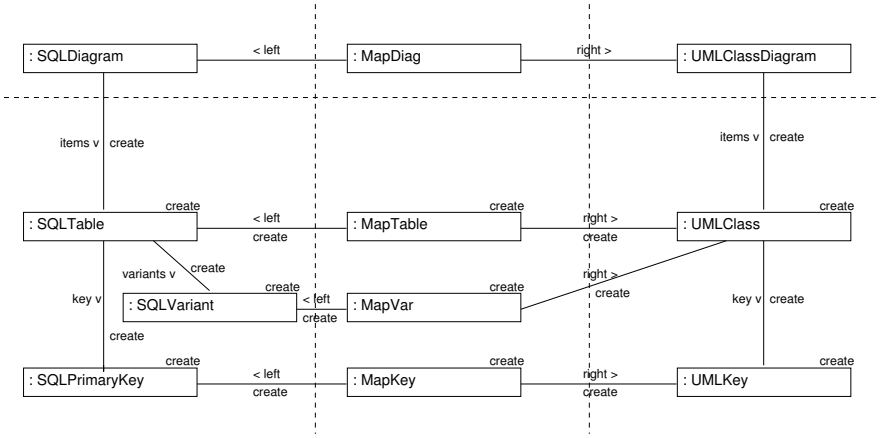


Abbildung 4.7.: TGG-Regel: Relationen ¡-¿ Klassen

Seite gehören mit zu der Vorbedingung und die Mitte wird weiterhin erzeugt. Die gestrichelte Linie in der Abbildung 4.8 trennt die Vorbedingung der linken Transformationsregel von der Erzeugung ab.

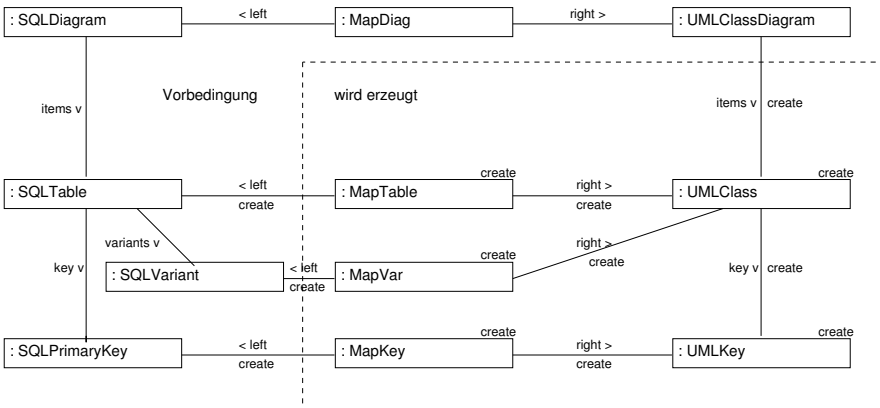


Abbildung 4.8.: Linke Transformationsregel

Bei der Durchführung einer Transformationsregel wird innerhalb des Graphen nach einer Konstellation gesucht, die der Vorbedingung der Regel entspricht. Bei Ausführung der in Abbildung 4.8 zu sehenden linken Transformationsregel wird der durch die vorhandenen Objekte aufgespannte Graph dahingehend untersucht, ob es nicht ein `SQLDiagram` gibt, das über einen `MapDiag`-Knoten mit einem `UMLClassDiagram` verbunden ist. Außerdem muß sich auf dem Diagramm ein `SQLTable`-Objekt befinden, das eine Referenz zu einem `SQLPrimaryKey`- und einem `SQLVariant`-Objekt hat. Erst wenn die Vorbedingung erfüllt ist, wird die Transformationsregel ausgeführt. Es wird ein Objekt vom Typ `UMLClass` und eines vom Typ `UMLKey`, zuzüglich der neu anzulegenden Map-Knoten wie zum Beispiel `MapVar`, erzeugt, und die in Abbildung 4.8 zu sehenden Referenzen wer-

den zwischen den einzelnen Objekten gezogen. Die Map-Knoten werden für die spätere Konsistenzerhaltung benötigt. Falls auf der konzeptionellen rechten Seite beispielsweise der Name der Klasse geändert wird, soll auch der Name der Tabelle geändert werden.

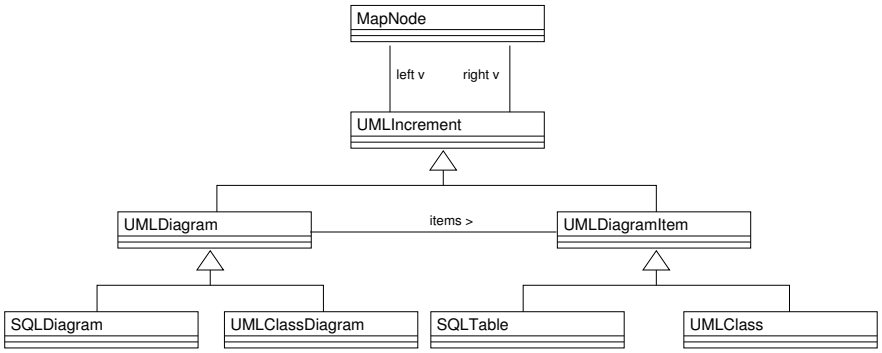


Abbildung 4.9.: REDDMOM/FUJABA Klassendiagramm mit TGG-Knoten

Durch das obige Beispiel kann wegen seiner Einfachheit die Frage aufkommen, wieso es diese Map-Knoten geben muß. Man könnte ja auch die logische Seite direkt mit der konzeptionellen Seite verbinden, und man könnte auch Änderungen auf der einen Seite auf die andere übertragen. Außerdem würde Speicherplatz gespart werden, da die mittleren Knoten wegfallen würden.

Durch die Einführung der Map-Knoten werden aber die linke und die rechte Seite voneinander unabhängig. Als Nachbarn kennen sie nur den Map-Knoten. Aus der in Abbildung 4.9 zu sehenden Klassenstruktur ist zu erkennen, daß zwar die SQL-Klassen und die UML-Klassen indirekt von `UMLIncrement` erben, sich aber gegenseitig nicht kennen. Durch die Klasse `UMLIncrement` erben sie auch die beiden Referenzen zu der `MapNode`-Klasse, von der die in den Transformationsregeln dargestellten Map-Knoten erben. Die Referenzen werden benötigt, damit die Objekte der Klassen in der Transformationsregel verwendet werden können.

Durch die Definition weiterer Transformationsregeln ist es möglich, eine der beiden Seiten auf ein ganz anderes System konsistent abzubilden, ohne die Struktur der Seite zu ändern. Die parallell zu dieser Diplomarbeit entstehende Arbeit [Wag01], die sich mit der Unterstützung von objektorientierten Datenbanken innerhalb von REDDMOM befaßt, verwendet auch Transformationsregeln, um das logische, diesmal objektorientierte, Schema auf ein konzeptionelles objektorientiertes Schema abzubilden.

Auch wenn die Transformationsregeln wegen der Abbildung von einem objektorientierten Schema auf ein objektorientiertes Schema wesentlich einfacher sind, zeigt das, daß es zu der konzeptionellen rechten Seite unterschiedliche Transformationen auf die linke Seite gibt, ohne daß die Struktur der rechten Seite geändert werden muß. Bei einer direkten Verbindung der beiden Seiten wäre das

ausgeschlossen, da eine weitere Abbildung auf ein anderes System unter Berücksichtigung derselben Strategie die Logik verändern würde.

Außerdem ist die hier vorgestellte Regel eine von neun. Die anderen Regeln haben zwar eine andere Aufgabe als die hier vorgestellte, an ihnen nehmen aber auch vorwiegend die oben gezeigten Objekte teil. Zum Beispiel ist das `SQLVariant`-Objekt in der oben aufgeführten Transformationsregel der Ausgangspunkt für weitere Transformationen, weitere Varianten des oben dargestellten `SQLTable`-Objekts in eine Klassenhierarchie zu transformieren. Da dort auch wieder Beziehungen zwischen den Objekten und jetzt ganz anderen Elementen auf der anderen Seite gezogen werden, steht einem Objekt eine komplexe Menge gegenüber, die bei einer Änderung des Objektes konsistent geändert werden muß.

Durch Weglassen der Map-Knoten wird eine hohe Eigenverwaltung an das Objekt gestellt, da es alle Auswirkungen kennen muß, um die Änderungen in die andere Darstellung zu übernehmen. Die eingeführten Map-Knoten bringen etwas mehr Struktur in die Komplexität, da man schon durch die Betrachtung der Objekte, die man von einem einzigen Map-Knoten erreichen kann, auf die Transformationsregel schließen kann, die diesen einen Knoten erzeugt hat.

Für eine vollständige Vorstellung der in dieser Arbeit nachimplementierten neun Triple-Graph-Grammatiken sei hier auf [Jah99] verwiesen.

5. Der Generator

In diesem Kapitel wird der Generator und damit das Konzept zur Erzeugung der objektorientierten Zugriffsschicht vorgestellt. Aus den vorigen Kapiteln ist bekannt, daß ein logisches Schema über Transformationsregeln in ein konzeptionelles Schema überführt wird. Das konzeptionelle Schema soll dabei die nach außen sichtbare objektorientierte Schnittstelle der Zugriffsschicht definieren. Der Zugriff muß datenbankunabhängig sein, das heißt, daß der Anwender der Zugriffsschicht Eigenschaften des logischen Schemas und damit auch die dahinter befindliche Datenbank nicht kennen muß.

Das konzeptionelle Schema liegt aber auch in einer UML-Notation vor, die durch das Meta-Modell von FUJABA eingeschränkt ist. In dem UML-Meta-Modell von FUJABA wird keine OCL[1] unterstützt, mit der man zum Beispiel die vorgestellten Datenbank-Einschränkungen wie die Länge eines Attributes nachbilden könnte. Zusätzlich gibt es spezielle Eigenschaften, die aus der Datenbankwelt kommen, für die Code-Generierung wichtig sind und nach UML nicht ohne weiteres übertragbar sind. In diesem Zusammenhang sei das `autoincrement`-Attribut genannt, das von der Datenbank automatisch hochgezählt wird und in der Zugriffsschicht nur noch einen lesenden Zugriff auf das Attribut erlauben darf.

Diese Diplomarbeit umfaßt nicht die Erweiterung des Meta-Modells von FUJABA um OCL.

Eine Erweiterung um OCL ist sicherlich erstrebenswert. Mit der Einbeziehung des logischen Schemas während der Code-Generierung ist die OCL in diesem Zusammenhang nicht notwendig, da in dem logischen Schema alle notwendigen datenbankspezifischen Eigenschaften für die Zugriffsschicht vorliegen.

Die Abbildung 5.1 gibt einen allgemeinen Überblick über die Umgebung, in der sich der Generator befindet. In dem Überblick werden die Bereiche aufgezeigt, auf die bei der Generierung zugegriffen werden.

Bei dem `Konfigurations`-Element handelt es sich um die allgemeinen Informationen, die aus den in Kapitel 4.1.1 vorgestellten Methoden der Klassen `java.sql.DatabaseMetaData` und `java.sql.ResultSetMetaData` gewonnen werden und um die in Kapitel 4.2 vorgestellte Konfiguration von REDDMOM.

[1] OCL: Object Constraint Language

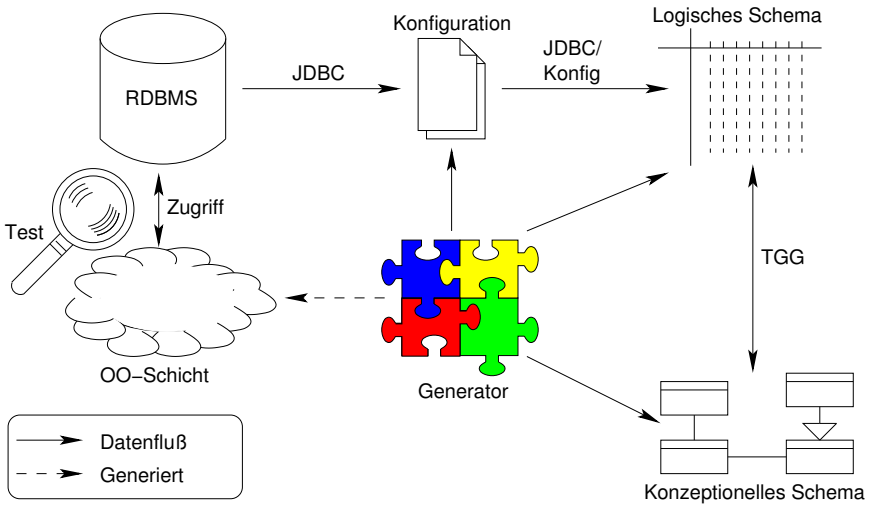


Abbildung 5.1.: Generator Übersicht

Das logische Schema ist das Schema, welches aus der Datenbank ausgelesen wurde. Das konzeptionelle Schema ist das durch die in Kapitel 4.3 vorgestellten Transformationsregeln entstandene initiale UML-Klassendiagramm.

Nach der Generierung möchte man die objektorientierte Zugriffsschicht benutzen. Bei der Benutzung muß sichergestellt sein, daß die Zugriffsschicht korrekt arbeitet. Im Rahmen dieser Diplomarbeit ist das Testwerkzeug Mr. DOLittle[2] entstanden, mit dem man über die Zugriffsschicht auf eine von der Struktur her passende Datenbank zugreifen kann.

Die entstehende objektorientierte Zugriffsschicht ist sehr komplex[3]. In den nachfolgenden Kapiteln wird die Zugriffsschicht anhand der Anforderungen, die an sie gestellt werden, vorgestellt. Eine vollständige Darstellung über die Struktur der Zugriffsschicht ist in Abbildung 5.2 zu sehen.

5.1. Einschränkungen durch JDBC

Aus den Kapiteln 2.1.4 und 4.1.1 ist bekannt, daß das JDBC-Paket nicht nur den Zugriff auf Datenbanken sondern auch auf beispielsweise Textdateien in tabellarischer Form erlaubt.

Textdateien befinden sich normalerweise auf einem Filesystem eines Rechners, daß von seinem Betriebssystem verwaltet wird. Bei dem Schreibauftrag einer Datei schreiben moderne Betriebssysteme den Inhalt nicht sofort auf ihr Filesystem,

[2]Mr. DOLittle: Mr. DOBS and a Little bit more
[3]Zugriffsschicht für die Datenbank dsd: 109 Klassen, 28434 Zeilen.

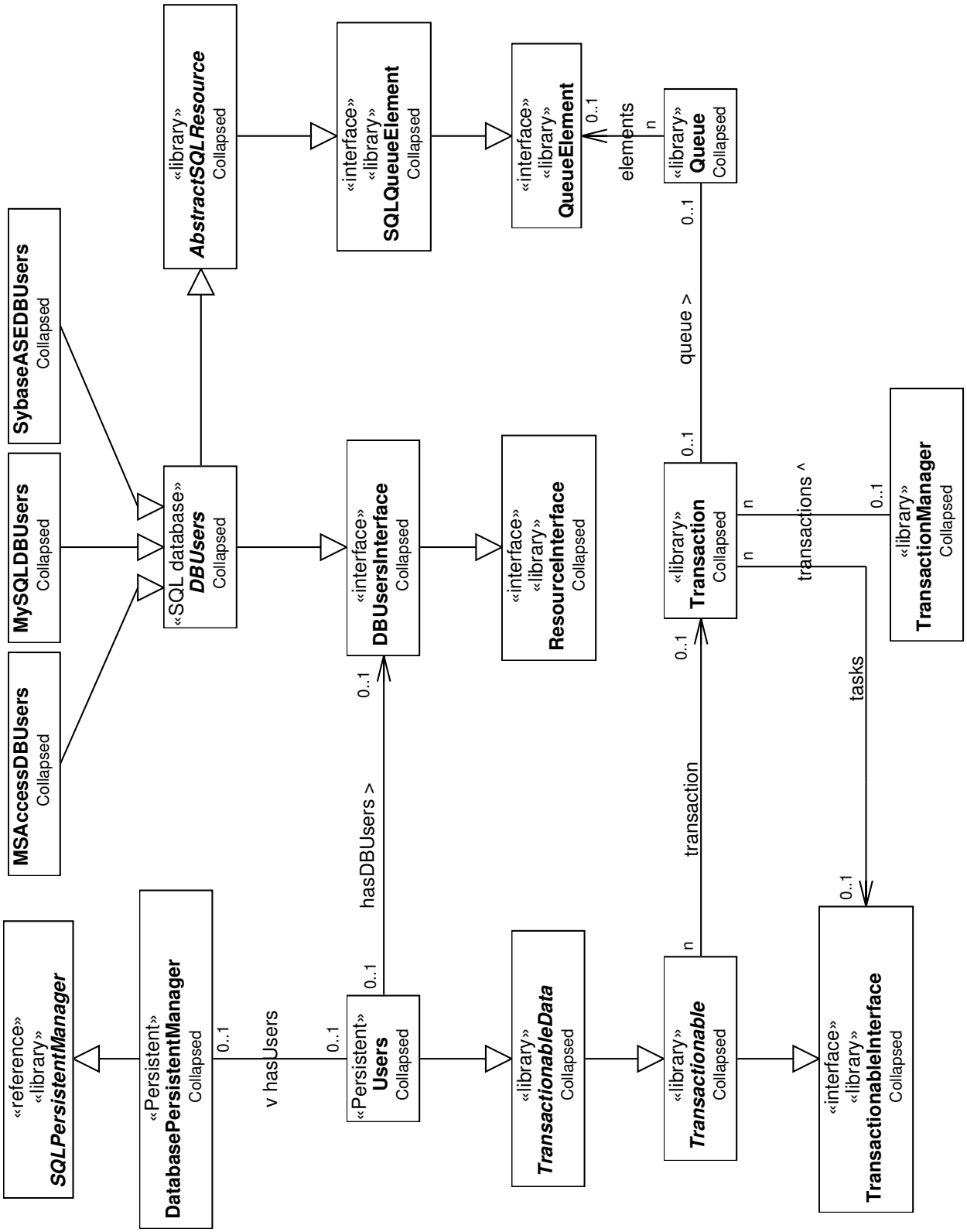


Abbildung 5.2.: Generierte Zugriffsschicht am Beispiel users

sondern halten es zunächst in ihrem Hauptspeicher und speichern es später persistent ab.

Bei einem möglichen Betriebssystemabsturz gehen die noch im Hauptspeicher befindlichen Daten verloren. Der eigentliche Vorteil des Nicht-Sofort-Speicherns hält nicht die ACID - Bedingungen Atomicity und Consistency ein. Beide Bedingungen werden verletzt, wenn nur halbfertig geschrieben werden konnte. Datenbanken umgehen dieses Problem, indem sie ihr eigenes Dateisystem mit sich bringen und es auch selbst verwalten.

Die Erkennung solcher nicht transaktionsfähiger Systeme beim Einlesen ihrer Tabellenstruktur würde eine Konfiguration von REDDMOM mit sich bringen, wenn man den Zugriff auf solche Systeme generell untersagen will. Es müßte spätestens beim Erstellen des initialen konzeptionellen Schemas der Programm-Benutzer informiert werden, daß so ein System nicht unterstützt wird.

Auf der anderen Seite aber bietet die Zugriffsmöglichkeit auf ein solches System mittels JDBC auch Vorteile. Die Speicherungsart wird über die objektorientierte Zugriffsschicht vollkommen transparent. Der Benutzer der Schicht muß sich um die Art des Abspeicherns nicht kümmern. Das übernimmt die Zugriffsschicht für ihn.

Gerade bei der derzeit herrschenden Beliebtheit von den strukturierten *XML*[4]-Textformaten mit dazugehöriger *DTD*[5] ist die Vorstellung von einem möglichen JDBC-Treiber für XML-Dateien, die einer Tabellenstruktur entsprechen, sehr interessant. Die XML-Fähigkeit eines Systems könnte so über die objektorientierte Zugriffsschicht erledigt werden.

Deswegen gilt bei der hier vorgestellten Zugriffsschicht, daß die Transaktionssicherheit durch die tatsächliche Resource definiert wird und man zu jeder JDBC-unterstützten Resource eine Zugriffsschicht generieren kann.

5.2. Zustand vor Ausführung der Generierung

Wie in Abbildung 5.3 vereinfacht dargestellt ist, sind die konzeptionelle Klasse `Users` und die logische Tabelle `users` über einen Transformationsgraphen verbunden, der durch die in Kapitel 4.3 vorgestellten Transformationsregeln erzeugt wurde.

Der Stereotyp „`Persistent`" kennzeichnet in REDDMOM eine konzeptionelle Klasse und der Stereotyp „`SQL database`" eine logische.

Neu an der in Abbildung 5.3 gezeigten Darstellung ist das Zwischenschema, das durch das `DBUsers`-Objekt repräsentiert wird. Bisher wurde die Erzeugung des

[4]XML: Extended Markup Language
[5]DTD: Data Definition Language

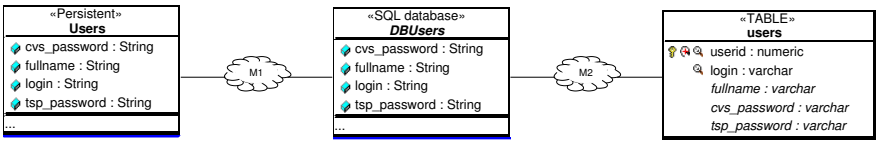


Abbildung 5.3.: Vorgabe der Transformationsregeln

konzeptionellen Schemas so vorgestellt, das es direkt aus dem logischen Schema über die Transformationsregeln in `M1` erzeugt wird. Man hätte also in der Abbildung 5.3 eigentlich nur die Tabelle `users` und die Klasse `Users` erwartet. Die Transformationsregeln M2 erzeugen ein objektorientiertes Zwischenschema, das im wesentlichen eine eins zu eins Darstellung des logischen Schemas ist. Da das Zwischenschema für die Generierung der Zugriffsschicht wichtig ist, soll es an dieser Stelle vorgestellt werden.

In den folgenden Kapiteln dieser Diplomarbeit wird es ersichtlich, daß die objektorientierte Zugriffsschicht über eine Datenklasse verfügen muß, die den Datenbank- Zugriff auf die relationale Tabelle bereitstellt. Der Generator muß also auf ein Objekt zugreifen können, das die zu generierende Datenklasse für die Tabelle `users` darstellt.

Die konzeptionelle Klasse `Users` wird über ein `UMLClass`-Objekt, das Bestandteil des FUJABA Meta-Modells ist, dargestellt. Diese Klasse besitzt unter Umständen eine andere Struktur als die Tabelle, da durch die Transformation die Vererbung mit einbezogen wurde. Außerdem verfügt die Klasse `Users` nicht mehr über die Attribute, die auf relationaler Seite einem Fremdschlüssel angehörten.

Die konzeptionelle Darstellung spiegelt also nicht die tatsächliche Struktur der Tabelle wider, sodaß sie als Generator für die Datenklasse ausscheidet.

Die Tabelle `users` hat als Repräsentationsobjekt in REDDMOM ein `SQLTable`-Objekt. Das `SQLTable`-Objekt ist in erster Linie für die relationale Darstellung der Tabelle `users` vorgesehen und soll im zukünftigen Verlauf des Projektes REDDMOM weitere relationale Funktionalitäten wie das Ändern der Struktur der Tabelle `users` bereitstellen.

Es wäre möglich gewesen, die Generierung der Datenklasse auch über dieses Objekt zu realisieren. Dann hätte aber die relationale Tabelle `users` mehrere Sichten auf sich bereitstellen müssen. Zum einen wäre es eine herkömmliche, relationale Tabelle. Zum anderen müßte sie aber auch über objektorientierte Eigenschaften, wie die Zugriffsmethoden auf Attribute, verfügen, die in der relationalen Welt unbekannt sind.

Bei der Implementierung wurde entschieden, das Objekt auf relationale Funktionalitäten zu beschränken, damit die Darstellung innerhalb von REDDMOM so nahe wie möglich an der Realität bleibt. Da das relationale Datenbank- Management- System über keine objektorientierten Eigenschaften verfügt und somit auch

keinen objektorientierten Zugriff für die Tabelle `users` anbieten kann, kann auch das `SQLTable`-Objekt keinen objektorientierten Code für die Tabelle `users` generieren. Außerdem bleibt so das, um die objektorientierten Eigenschaften befreite, `SQLTable`-Objekt kleiner in seiner Schnittstelle und damit übersichtlicher.

Es wurde das bereits erwähnte Zwischenschema geschaffen, das ausschließlich für die Generierung der Datenklassen verantwortlich ist und über beschränkte objektorientierte Eigenschaften verfügt.

Die zu den Tabellen korrespondierenden Elemente dieses Schemas sind bereits objektorientierte Klassen, die allerdings noch nicht über die Vererbung verfügen. Spätere Assoziationen werden noch als Fremdschlüsselbeziehung dargestellt.

Das Zwischenschema ist im wesentlichen eine Kopie des logischen Schemas. Das Zwischenschema hält die notwendigen Informationen für die tatsächlichen Transformationen in ein konzeptionelles Schema bereit. Die Kopie wird durch eine Vortransformation, die ebenfalls durch Triple-Graph-Grammatiken definiert ist, erzeugt.

Die in Abbildung 5.3 gezeigte Klasse `DBUsers` ist die Datenklasse für die Tabelle `users`. Sie definiert den nativen Zugriff auf die Tabelle `users`.

5.3. ACID-Transaction-Pattern

Transaktionen werden nicht von der Zugriffsschicht für den Benutzer transparent verwaltet. Eine Transaktion wird außerhalb der Zugriffsschicht gestartet und über den Versuch, eine Änderung eines Objektes innerhalb der Zugriffsschicht vorzunehmen, in die Zugriffsschicht hineingetragen.

Der Anwender bzw. die Applikation, der bzw. die auf die Zugriffsschicht zugreift, bestimmt die Änderungsfolge, die in der Transaktion durchgeführt werden soll. Das einzelne Objekt innerhalb der Zugriffsschicht kann nur entscheiden, ob die gewollte Änderung an ihm erlaubt ist oder nicht.

Falls eine Änderung nicht durchgeführt werden darf, muß das Objekt durch die Rückgabe eines Fehlers seine Änderung abbrechen. Da das einzelne Objekt nicht wissen kann, in welchem Zusammenhang seine Änderung durchgeführt werden sollte, darf es nicht die Transaktion abbrechen. Da die Transaktion außerhalb der Zugriffsschicht erzeugt wurde, muß auch außerhalb entschieden werden, wann die Transaktion beendet wird.

Wie bereits in den Kapiteln 2.2 und 2.3 beschrieben, erfolgt der Zugriff auf einen Datenbank-Server von mehreren Stellen gleichzeitig. Dabei werden Transaktionen auch nebeneinander ausgeführt. Da durch diese Diplomarbeit der Zugriff auf die Datenbank gekapselt wird, muß die Zugriffsschicht den von mehreren Stellen gleichzeitig auftretenden Zugriff auf sich erlauben.

Bei der Ausführung einer Transaktion muß gelten, daß die Folge ihrer Anweisungen die Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand überführt. Da der Zustand der Datenbank auch eine objektorientierte Präsentation durch die Zugriffsschicht hat, ergibt sich das Problem, die Präsentation mit dem Zustand der Datenbank konsistent zu halten.

Die von der Datenbank eigentlich mitgelieferte Transaktionsfähigkeit reicht nicht mehr aus, da sie sich nur auf den Zustand der Datenbank erstreckt und nicht die aufgesetzte Zugriffsschicht verändert. Beide Seiten würden inkonsistent, wenn es keine weitere Transaktionsfähigkeit auf Seiten der objektorientierten Zugriffsschicht gäbe.

Da die Manipulation der Daten in der Datenbank über die objektorientierte Zugriffsschicht erfolgt, ist es üblich, die gesamte Transaktion zuerst auf der objektorientierten Seite auszuführen, um die in der Transaktion durchgeführten Änderungen erst am Ende der Transaktion tatsächlich in die Datenbank zu schreiben. Beim Schreiben wird der aus dem Kapitel 2.2 bekannte partially-commited-Zustand der Transaktion simuliert. Falls beim Abspeichern in die Datenbank ein Fehler auftritt, kann die Transaktion auch in der objektorientierten Zugriffsschicht zurückgesetzt werden.

5.3.1. Vorlage

Die Theorie der Transaktion und die Umsetzung in die Realität sind ein sehr komplexes Gebiet, das den Rahmen dieser Diplomarbeit übersteigt. Es liegen bereits Grundkonzepte für die Realisierung von Transaktionen innerhalb einer Applikation vor.

Auf der *Pattern-Languages-of-Programming-'99*-Konferenz wurden vier Design-Pattern vorgestellt [Gra99], von denen jedes eine spezielle Anforderung an die Transaktion erfüllt. Zusammengebracht ergibt sich eine *Pattern-Language*, die eine solide Basis für ein "vollständiges" Transaktionskonzept bildet. An dieser Pattern-Language orientiert sich der Generator bei der Erzeugung der Zugriffsschicht.

Die vier Design-Pattern sind in der Abbildung 5.4 auf der nächsten Seite zu sehen.

Das *ACID - Transaction - Pattern* ist für die Einhaltung der ACID - Eigenschaften einer Transaktion zuständig.

Mit dem *Composite-Transaction* Pattern werden einzelne auf unterschiedlichen Datenbank-Servern laufende ACID - Transaktionen zu einer komplexen Transaktion zusammengefaßt. Die zu einer großen Transaktion zusammengefaßten kleineren Transaktionen werden auch als *Nested-Transactions* bezeichnet.

Das *Two-Phase-Commit* Pattern sorgt für die Atomarität der Composite-Transaktion.

Das *Audit-Trail* Pattern ermöglicht die Historie-Verwaltung über die ACID - Transaktionen.

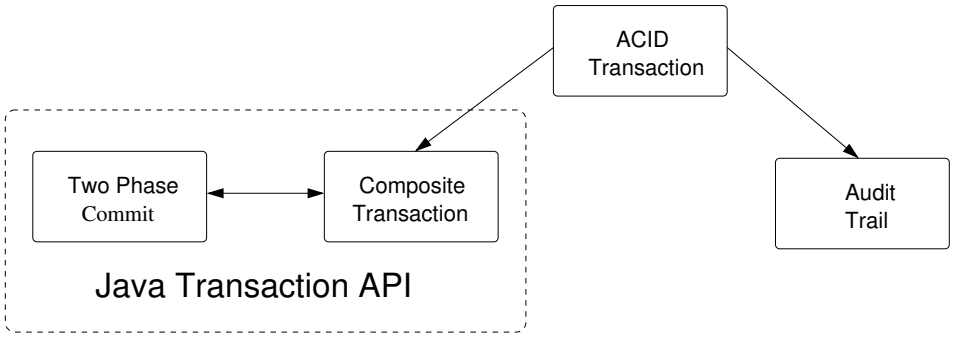


Abbildung 5.4.: Die vier Transaktions-Komponenten

Die *Java Transaction API* bietet in ihrer Schnittstellenbeschreibung Methoden an, die sich in dem Two-Phase-Commit-Pattern und in dem Composite-Transaction-Pattern wiederfinden. Bei der Realisierung dieser Pattern während der Generierung ist es naheliegend, die entstehende Klassenstruktur auf dieser API aufbauen zu lassen.

Da das Two-Phase-Commit Pattern und das Composite-Transaction Pattern sich auf verteilte Datenbanken erstrecken, übersteigt die Realisierung dieser Pattern den Inhalt dieser Arbeit und wird deswegen ausgelassen.

Das Audit-Trail Pattern dient dazu, eine Transaktion in ihrer Ausführung aufzuzeichnen. Dabei werden Zustandsänderungen der Objekte gespeichert, die an der Transaktion beteiligt sind. Es entsteht ein *Journal* zu der Transaktion, das später verwendet werden kann, um den genauen Ablauf der Transaktion zu ermitteln.

Dieses Pattern ist hilfreich, wenn man besondere Sicherheitsanforderungen an Transaktionen stellen will bzw. muß. Die aufgezeichneten Zustände sind dienlich bei der womöglichen Fehlerermittlung oder bei der Ermittlung, wer wann welche Transaktion ausgeführt hat und warum. Dabei werden sämtliche Änderungen erst einmal im Hauptspeicher aufgezeichnet, was im Bezug auf eine Datenbank enorme Ausmaße annehmen kann. Normalerweise wird nur ein kleiner Ausschnitt des Journals im Speicher gehalten und der Rest wird auf ein Filesystem ausgelagert.

Sicherlich ist dieses Pattern ein wesentlicher Bestandteil eines richtigen Transaktionskonzeptes. Eine Verwendung dieser Aufzeichnungsmöglichkeit macht aber nur Sinn, wenn es auch wirklich aus oben genannten Gründen benötigt wird. Der Generator realisiert dieses Pattern nicht.

Der Generator orientiert sich an dem noch nicht näher vorgestellten *ACID - Transaction* Pattern.

Auch wenn drei von den vier Design-Patterns nicht Bestandteil dieser Arbeit sind, ist die erfolgreiche Umsetzung der fehlenden drei Patterns, und hier gerade

das Composite - Transaction - Pattern, sehr stark von der ACID - Fähigkeit der einzelnen Transaktionen abhängig. Die schwächste ACID - Umsetzung der Einzeltransaktionen ist die ACID - Eigenschaft der gesamten Transaktion.

Deswegen soll die generierte Zugriffsschicht die ACID - Eigenschaften so gut wie möglich umsetzen. Da die anderen Pattern das ACID - Transaktion Pattern nur verwenden und nicht in seiner Struktur verändern, können sie nachträglich, auf dem Vorhandenen aufbauend, in weiterführenden Arbeiten realisiert werden.

Das ACID - Transaction - Pattern wird im weiteren Verlauf der Diplomarbeit anhand der vier Transaktionseigenschaften vorgestellt.

Atomarität

In dem in Abbildung 5.5 gezeigten Klassendiagramm sieht man die Struktur, um die geforderte Atomarität einer Transaktion gewährleisten zu können.

In diesem Diagramm wurde bei der Modellierung vorausgesetzt, daß der Transaktions-Manager die Absicht einer Zustandsänderung auf einem Transaktions-Objekt mitbekommt und auf ihm die Methode `startTransaction` ausführt.

Die `startTransaction` veranlaßt das Transaktions-Objekt, seine Daten zu initialisieren. Bei Bedarf werden die Daten aus der Datenbank gelesen. Man setzt voraus, daß die initialisierten Daten den konsistenten Zustand vor Transaktionsbeginn des Objektes widerspiegeln.

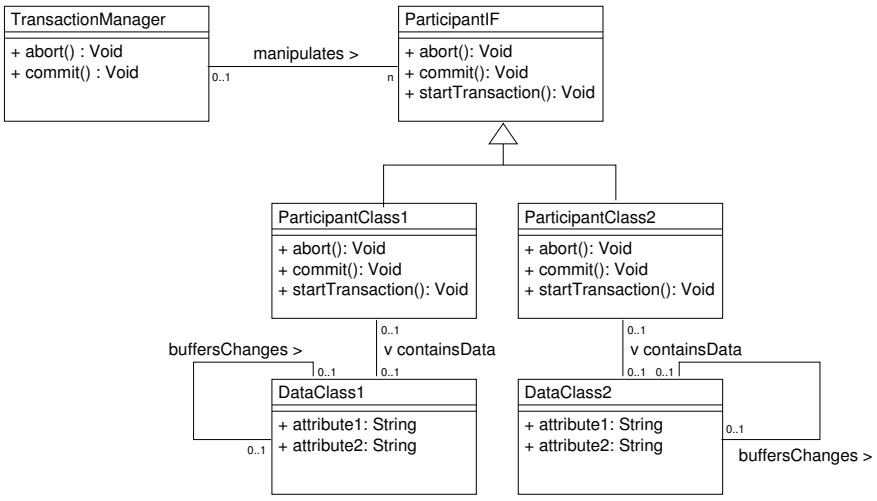


Abbildung 5.5.: Atomarität über Wrapper-Objekte

Die Daten liegen nicht direkt in dem Transaktions-Objekt sondern in einer separaten Datenklasse vor. Zugriffe auf die Daten werden von dem Transaktions-Objekt an das Datenobjekt weitergegeben.

Änderungen werden nicht auf den initialen Datenwerten durchgeführt. Die Änderungen werden auf einem zweiten Datenobjekt gemacht. Dabei ist es sinnvoll, nur die tatsächlichen Änderungen, die in die Datenbank geschrieben werden müssen, festzuhalten. Bei Abschluß der Transaktion mittels `commit` werden die geänderten Daten des zweiten Objektes zuerst in die Datenbank geschrieben.

Wenn die Schreibaktion auf der Datenbank ebenfalls erfolgreich war, werden die Daten des zweiten Datenobjektes in das erste Datenobjekt übernommen, und die gesamte Transaktion wird erfolgreich beendet.

Schlägt die Datenbanktransaktion fehl, kann die Transaktion auf objektorientierter Seite immer noch zurückgesetzt werden, indem die, für die Änderungen vorhandene, Kopie verworfen wird.

Isolation

Das in Abildung 5.6 gezeigte Klassendiagramm ist das um das *Read-/Write-Lock-Design-Pattern* erweiterte Klassendiagramm aus Abbildung 5.5.

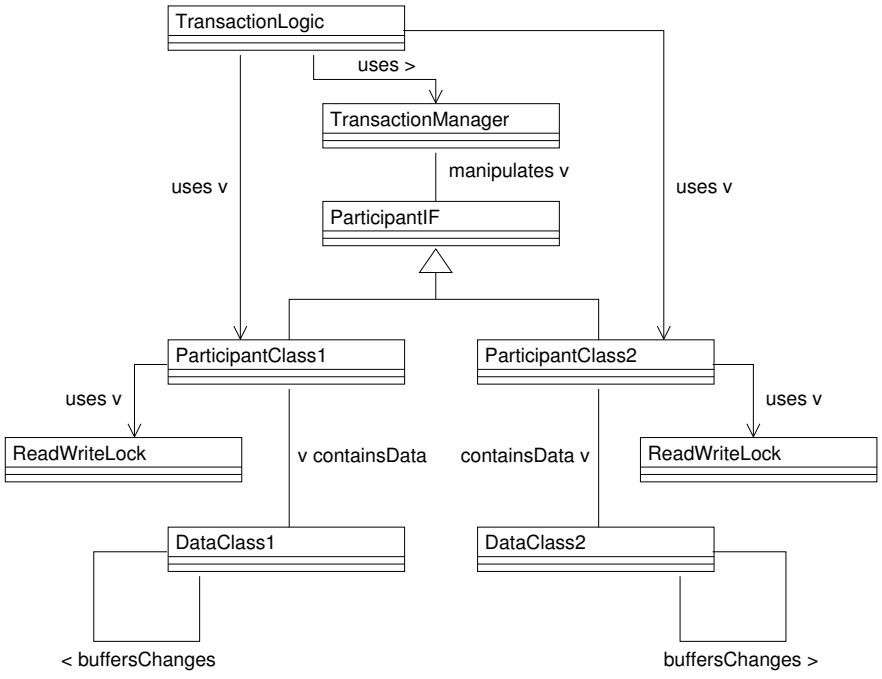


Abbildung 5.6.: Das Read/Write Lock Pattern

Die `ReadWriteLock`-Klasse hat die Aufgabe, ein Transaktions-Objekt zu sperren, wenn das Objekt innerhalb einer Transaktion geändert wird.

Dabei sollen weitere Änderungen an dem Objekt innerhalb derselben Transaktion noch möglich sein. Andere Transaktionen hingegen sollen nicht die Möglichkeit

erhalten, Änderungen an diesem Objekt vorzunehmen, solange die Transaktion noch nicht abgeschlossen ist.

Das Design-Pattern ermöglicht den gleichzeitigen lesenden Zugriff auf das Objekt. Um als isoliert zu gelten, darf das Objekt nicht Daten beim Lesen zurückliefern, die durch eine noch laufende Transaktion erzeugt wurden.

Es muß immer der zuletzt bekannte konsistente Wert aus der Datenbank zurückgeliefert werden.

Mit Hilfe der aus der Abbildung 5.5 übernommenen Struktur kann dieses Verhalten modelliert werden.

Consistency und Durability

Konsistenz kann nicht durch eine besondere Programmiertechnik oder durch ein Design-Pattern erreicht werden. Konsistent kann die generierte Zugriffsschicht nur sein, wenn sie fehlerfrei ist. Dazu müssen ausführliche Tests auf der Zugriffsschicht durchgeführt werden. Neben dem Zeitgewinn durch die Generierung der Zugriffsschicht gegenüber dem manuellen Erstellen liegt hier ein weiterer Vorteil in der Generierung.

Generierter Code läuft oder läuft nicht. Bei der Generierung werden Regeln durchlaufen, die bestimmte Aufgaben bei der Erzeugung des Programm-Codes erfüllen. Eine Regel könnte zum Beispiel der schreibende Zugriff auf ein Attribut eines Transaktions-Objektes sein. Diese Regel wird bei der Generierung sehr oft durchlaufen, und bei größeren Datenbanksystemen sind 1000 eher als Untergrenze anzusehen.

Falls sich in der Generierung der schreibenden Zugriffsmethoden für Attribute ein Fehler verbirgt, wird sich die Zugriffsschicht beim generellen Schreiben innerhalb einer Transaktion immer falsch verhalten, sodaß der Fehler meist sofort ersichtlich wird.

Durch die Änderung einer einzigen Stelle innerhalb der Regel sind bei der nächsten Generierung über 1000 Fehler behoben. Beim manuellen Erzeugen müßten alle 1000 Zugriffsmethoden getestet werden, um sicherzustellen, daß die Zugriffsschicht korrekt funktioniert.

Wie bereits erwähnt, kann man die notwendige Dauerhaftigkeit auf einem Filesystem nicht garantieren. Die Dauerhaftigkeit der Daten, die von der Zugriffsschicht verwaltet werden, ist von dem eingesetzten Resourcen-System abhängig.

5.3.2. Umsetzung

Das in Kapitel 5.3.1 vorgestellte ACID - Transaction - Pattern bildet eine solide Grundlage für die tatsächliche Realisierung der objektorientierten Zugriffsschicht

auf eine relationale Datenbank.

Da Design-Patterns erlauben, aus praktischen Gründen etwas von ihrer Definition abzurücken, und weil die durch die Transformationen entstandenen Schemata die Basis für die Generierung bilden sollen, wird in diesem und in den folgenden Kapiteln die bei der Generierung entstehende Struktur an dem Beispiel der Tabelle `users` aus der dsd-Datenbank beschrieben.

ACID-Transaction-Pattern vs. Vorhandene Logik

Die Datenklasse, die für die Realisierung des ACID - Transaction - Patterns notwendig ist, ist in der vorhandenen Logik leicht zu ermitteln. Die Generierung der Datenklassen übernehmen die Elemente aus dem Zwischenschema, das im Kapitel 5.2 vorgestellt wurde. Nach Ausführung der Transformationsregeln besitzt jede relationale Komponente wie die Tabelle `users` ein Partner-Element auf dem Zwischenschema, das über die benötigten objektorientierten Eigenschaften für den Zugriff auf die Tabelle verfügt.

Die zweite wichtige Komponente des ACID - Transaction - Patterns ist die Klasse, die die Transaktion steuert. Eine Transaktion wird außerhalb der Zugriffsschicht erzeugt. Bei dem Versuch, ein Objekt aus der Zugriffsschicht zu ändern, muß dem Objekt mitgeteilt werden, daß die Änderung innerhalb der außen erzeugten Transaktion durchgeführt werden soll. Das Mitteilen, um welche Transaktion es sich dabei handelt, muß also zu dem Teil der Zugriffsschicht gehören, der nach außen hin öffentlich ist.

Die konzeptionellen Klassen sind dafür gedacht, den Zugriff auf die tatsächliche Datenbank bzw. Resource zu kapseln. Sie sollen spezifische Eigenschaften der verwendeten Datenbank bzw. Resource vor dem Benutzer verbergen, sodaß der Benutzer die spezifischen Eigenschaften nicht kennen muß.

Damit kommen für die Transaktionskomponenten nur die konzeptionellen Klassen der Zugriffsschicht in Frage, da sie die einzigen öffentlichen Elemente der Zugriffsschicht sind.

Bei einer genauen Orientierung an dem Design-Pattern müßte der Generator zwei Java-Klassen für den Zugriff auf die Tabelle `users` generieren. Eine konzeptionelle `Users`-Transaktionsklasse und eine `DBUsers`-Datenklasse. Außerdem müßte die Transaktionsklasse eine direkte Referenz auf die Datenklasse haben.

Die Datenklasse `DBUsers` ist aber eine Klasse, die auf die spezielle Resource ,,relationale Datenbank" zugreift und dort Tupel in der Tabelle `users` anlegt, verändert oder auch löscht. Für diesen Zugriff auf die relationale Datenbank besitzt die `DBUsers`-Klasse neben den allgemeinen Attribut-Zugriffsmethoden noch weitere datenbankspezifische Methoden, die die Kommunikation mit dem Datenbank-Server übernehmen. Die datenbankspezifischen Methoden werden im weiteren Verlauf dieser Diplomarbeit noch näher vorgestellt.

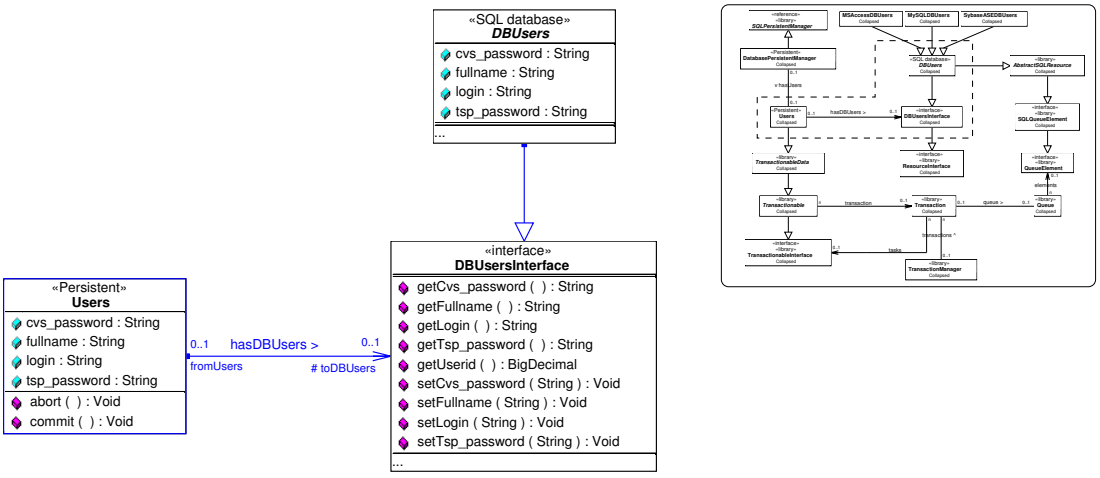


Abbildung 5.7.: Generierte Transaktions- und Datenkomponente

Die `Users`-Klasse muß diese Methoden nicht kennen. Bei einer direkten Referenz auf die Datenklasse `DBUsers` wären der `Users`-Klasse diese Methoden aber bekannt, und sie könnte sie auch anwenden, da die Methoden alle öffentlich sind und dies auch sein müssen.

Dadurch entsteht die Gefahr, die objektorientierte Zugriffsschicht falsch zu verwenden. Auch wenn die Zugriffsschicht generiert wird und damit die falsche Anwendung quasi ausgeschlossen ist, muß für die spätere Weiterentwicklung der Zugriffsschicht klar sein, welche Teile der Datenseite der Transaktionsseite bekannt sein müssen und welche nicht.

Deswegen wird eine Schnittstelle `DBUsersInterface` für die `DBUsers` - Klasse eingefügt, die nur die Methoden der `DBUsers` - Klasse öffentlich macht, die die `Users` - Klasse auch wirklich kennen muß. Die Referenz aus dem ACID - Transaction - Pattern wird auf die Schnittstelle umgeleitet. Damit ist die datenbankspezifische Funktionalität der Klasse `DBUsers` gekapselt und der `Users` - Klasse unbekannt.

Dadurch ist eine Falschverwendung innerhalb der Zugriffsschicht an dieser Stelle ausgeschlossen. Die Abbildung 5.7 zeigt die aus dem ACID - Transaction - Pattern übernommene Struktur an dem Beispiel der `user` - Tabelle mit den hier erwähnten Abweichungen von dem Design - Pattern.

Die in der Abbildung enthaltene Legende gibt einen Überblick über die gesamte Struktur, die auf der Seite 5.2 zu sehen ist. Die gestrichelte Linie in der Legende verweist auf den groß dargestellten Ausschnitt in der Abbildung.

Die drei Punkte, mit denen einige der in Abbildung 5.8 zu sehenden Klassen abgeschlossen werden, ist eine für die Vorstellung eines Klassendiagramms sehr nützliche Funktionalität von FUJABA. FUJABA bietet die Möglichkeit, Elemente zu einer Klasse, wie etwa Attribute und Methoden, ein- bzw. auszublenden. Die

drei Punkte deuten darauf hin, daß die entsprechende Klasse noch über weitere nicht sichtbare Elemente verfügt.

So kann man die Darstellung auf das beschränken, was im Text erläutert wird.

Die Transaktionskomponente

Die Abbildung 5.8 gibt einen vollständigen Überblick über die Struktur der Transaktionskomponente. Die Struktur stimmt weitestgehend mit der Struktur des ACID - Transaction - Design - Patterns überein. Die Vorgabe des Design - Patterns wurde an einigen Stellen verfeinert.

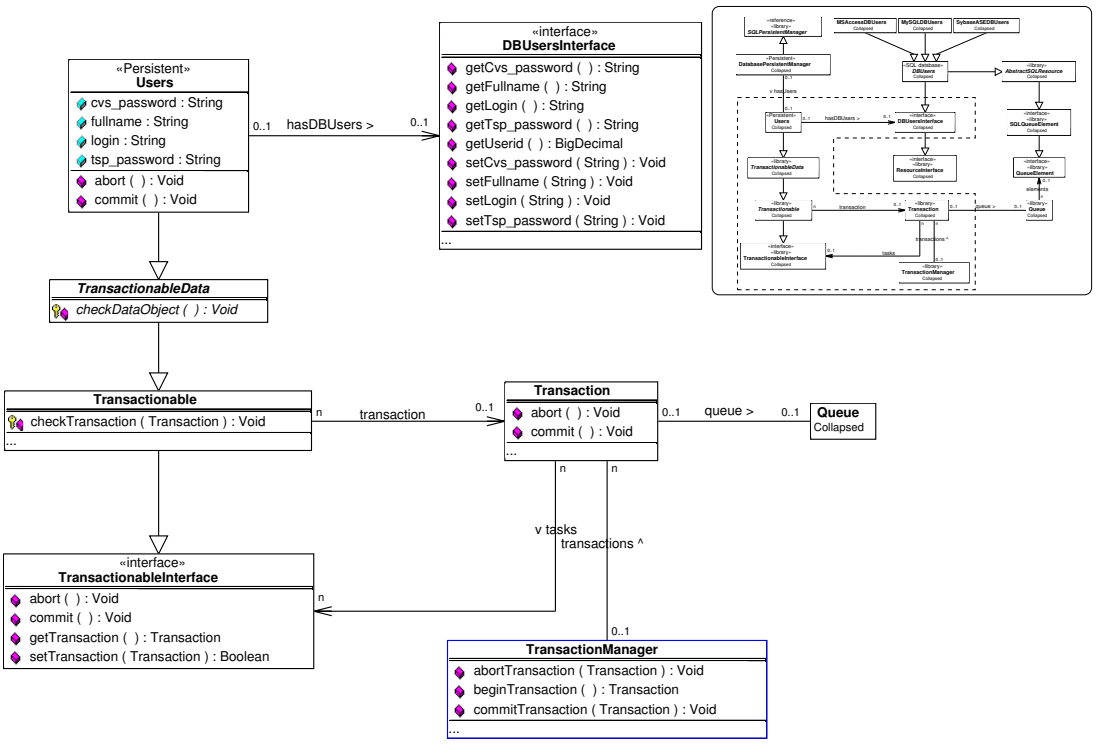


Abbildung 5.8.: Transaktions-Seite am Beispiel Users

Der Stereotyp ,,`library`" kennzeichnet alle Klassen und Schnittstellen, die nicht generiert werden müssen. Bei der Generierung wird auf eine Bibliothek zurückgegriffen, die allgemeingültige Funktionen bereitstellt und im Rahmen dieser Diplomarbeit entwickelt wurde.

Ein Transaktions-Objekt ist bei der folgenden Erläuterung ein Objekt von einer Klasse, die direkt oder indirekt von der in Abbildung 5.8 zu sehenden Klasse `TransactionableInterface` erbt. Objekte von der Klasse `Users` sind zum Beispiel Transaktions-Objekte. Da die meisten Klassen in der Abbildung 5.8 `library`-Klassen darstellen, müssen sie allgemein gültig sein. Mit dem Transaktions- Ob-

jekt wurde ein Begriff geschaffen, der unabhängig von einer konkreten Klasse die Struktur beschreibt.

Die größte Abweichung von der Vorlage aus dem ACID - Transaction - Pattern ist das Vorhandensein der Attribute aus der Tabelle `users` in der Transaktionsklasse `Users`. Das Erscheinen ist eine Vorgabe, die man durch die Ausführung der vorgestellten Transformations-Regeln zum Erzeugen des initialen Schemas erhalten hat.

Dadurch, daß sowohl die logische Klasse `DBUsers` als auch die konzeptionelle Klasse `Users` über Attribute verfügen, die die Daten aus der Tabelle `users` aufnehmen können, ist die Idee naheliegend, über die vorgegebene Struktur die von dem ACID- Transaction- Pattern verlangte Kopie zu realisieren, die für die Rücksetzung des Transaktions - Objektes bei Abbruch der Transaktion benötigt wird.

Wie beim ACID- Transaction- Pattern werden die während einer Transaktion durchgeführten Änderungen beim Datenobjekt gehalten, da sie dort zum Abspeichern in die Datenbank als erstes gebraucht werden. Weil die geänderten Daten bei einer Anfrage wegen dem Isolations - Kriterium noch nicht zurückgegeben werden dürfen und die Anfrage über die Transaktions- Komponente erfolgt, werden die zuletzt konsistenten Werte vor Transaktions- Beginn in dem Transaktions-Objekt gehalten.

Dadurch wird eine zusätzliche Kopien - Haltung auf Seiten der Datenklasse überflüssig. Die aus dem ACID - Transaction - Pattern bekannte Kopie - Referenz auf die Datenklasse muß durch den Generator nicht realisiert werden.

Die restliche Transaktionsumgebung, die in Abbildung 5.8 zu sehen ist, wird in den folgenden Absätzen vorgestellt.

Die Klasse `Users` ist zunächst einmal ein Transaktions - Objekt, das maximal einer Transaktion auf einmal zugeordnet werden kann und die aus der Transaktions - Definition bekannten Methoden `abort` und `commit` bereitstellt. Diese Eigenschaften erbt die Klasse `Users` von der Schnittstelle `TransactionableInterface` und der abstrakten Klasse `Transactionable`.

Die Schnittstelle `TransactionableInterface` definiert die minimalen Eigenschaften eines Transaktions - Objektes. Bisher wurde ein Transaktions - Objekt immer als ein Objekt angesehen, das über eine Datenquelle verfügen muß. Es wird aber auch in der Praxis Fälle geben, wo man von ,,herkömmlichen" Objekten verlangt, daß sie nur unter bestimmten Bedingungen Änderungen erlauben und sich zurücksetzen können. Diese speziellen Objekte können so als Transaktions - Objekte ohne spezielle Datenklasse angesehen werden.

Da die Datenklasse in dieser Zugriffsschicht auch für die Kopieverwaltung benötigt wird, müßten diese Transaktions - Objekte die Kopieverwaltung anders realisieren. Sie könnten aber direkt oder indirekt von der allgemeinen Schnittstelle `TransactionableInterface` erben. Dadurch müßte die Transaktionsverwaltung,

die in der Abbildung 5.8 durch den `TransactionManager` und die `Transaction` dargestellt ist, nicht geändert werden.

Die abstrakte Klasse `Transactionable` implementiert einen Teil der Schnittstelle `TransactionableInterface` aus. Sie definiert vorwiegend die Referenz auf die `Transaction`-Klasse und stellt einige allgemeine Methoden zur Verfügung, die alle Transaktions-Objekte, die von dieser Klasse erben, verwenden können.

Die gerichteten Referenzen `transaction` und `tasks` sind so implementiert, daß sie zusammen eine bidirektionale Assoziation definieren. Wenn dem Transaktions-Objekt ein `Transaction`-Objekt über `setTransaction` zugewiesen wird, trägt es sich in die `tasks`-Menge des `Transaction`-Objektes ein.

Die öffentlichen Zugriffsmethoden `get-` und `setTransaction` der Klasse `Transactionable` werden bereits in der Schnittstelle `TransactionableInterface` definiert. Damit sind dem `Transaction`-Objekt die Zugriffsmethoden auf die `transaction`-Referenz bekannt. Bei der Eintragung eines Transaktions-Objektes in die `task`-Menge setzt das `Transaction`-Objekt die `transaction`-Referenz des Transaktions-Objektes auf sich selbst.

Die abstrakte Klasse `TransactionableData` definiert die Methode `checkDataObject` für Transaktionsklassen, die über eine Datenquelle verfügen. Zusammen mit der `checkTransaction`-Methode aus der `Transactionable`-Klasse stehen den Transaktions-Objekten Mittel zur Verfügung, um ihre Korrektheit zu kontrollieren.

Beide, nur von den Transaktions-Objekten vewendbare, Methoden sind so spezifiziert, daß sie spezielle Exceptions werfen können. Damit können die Transaktions-Objekte der Außenwelt mitteilen, daß etwas mit ihnen nicht stimmt. Die Entscheidung, was in einem solchen Fall passieren soll, muß außerhalb der Zugriffsschicht bestimmt werden, da dort die Transaktion eingeleitet wurde und das einzelne Transaktions-Objekt die Umwelt, in der es zum Beispiel geändert werden sollte, nicht kennt.

Der `TransactionManager` verwaltet die im System laufenden Transaktionen. Über ihn werden Transaktionen gestartet (`beginTransaction`) und beendet (erfolgreich: `commitTransaction`, abbrechen: `abortTransaction`).

In die `queue` des `Transaction`-Objektes werden Objekte abgelegt, die bei einem `commit` zuerst fehlerfrei abgearbeitet werden müssen, bevor die Transaktion erfolgreich beendet werden kann.

Wie bereits erwähnt, führt die hier vorgestellte Zugriffsschicht zunächst alle Änderungen auf sich selbst aus, um die Änderungen dann später, wenn die Transaktion erfolgreich beendet werden soll, gebündelt in die Datenbank zu schreiben. Die `Queue` dient dazu, die angefallenen Änderungen innerhalb einer Transaktion zu sammeln. Auf die `Queue` wird später noch genauer eingegangen.

Falls bei der Abarbeitung dieser `queue` ein Fehler auftritt, besteht immer noch die Möglichkeit, die Transaktion mittels eines `abort` abbrechen zu lassen. Mit Hilfe der `queue` wird so der `partially-commited` Zustand der Transaktion innerhalb der Zugriffsschicht simuliert.

5.4. Attribute / Assoziationen

Aus dem Kapitel 5.3.2 ist bekannt, daß Änderungen an den Daten innerhalb der Datenbank nur über eine Transaktion erfolgen können, die von der konzeptionellen Seite verwaltet wird.

In der Abbildung 5.9 ist das Attribut `login` der Klasse `Users` mit den dazugehörigen Zugriffsmethoden zu sehen. Die von dem Generator behandelte persistente Klasse ist mit dem Stereotyp `REDDMOM` gekennzeichnet.

Die darunter dargestellte, gleichnamige Klasse mit dem Stereotyp `FUJABA` stellt die herkömmlichen Zugriffsschnittstellen für das Attribut `login` dar, die beim Anlegen des Attributs `login` von FUJABA automatisch generiert werden. Auch in FUJABA kann man, wie in Abbildung 5.9 zu sehen, persistente Klassen über einen Stereotypen kennzeichnen. Dies hat aber keine Auswirkung auf die Code-Generierung. Es ist nur ein Hilfsmittel zur Dokumentation.

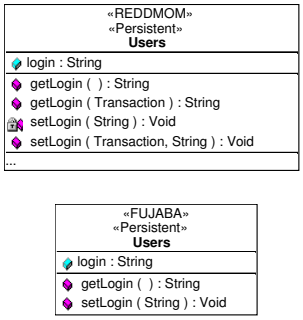


Abbildung 5.9.: Persistente Zugriffsmethoden vs. Fujaba-Zugriffsmethoden

Die konzeptionelle `Users`-Klasse stellt für den Zugriff auf ihr Attribut `login` vier Methoden bereit. Die beiden `get`-Methoden definieren den lesenden und die zwei `set`-Methoden definieren den schreibenden Zugriff auf das Attribut. Alle Methoden sind, mit Ausnahme der einen `set`-Methode mit dem einen `String`-Parameter, öffentlich und von außen damit benutzbar.

lesende Zugriffs-Methoden

Die `getLogin`-Methode ohne Parameter gibt immer den derzeit konsistenten Wert aus der Datenbank zurück, indem sie den Wert des `login`-Attributs des `Users`-

Objektes zurückgibt. Dadurch ist die Eigenschaft der Isolation, die von einem Transaktions-Objekt verlangt wird.

Innerhalb einer Transaktion besteht der Anspruch, den derzeit aktuellen Wert zurückzugeben. Hierfür steht die zweite `getLogin`-Methode bereit, die als Parameter ein `Transaction`-Objekt verlangt. Der `Transaction`-Parameter gibt dabei an, für welche Transaktion der Wert zur Verfügung gestellt werden soll. Es wird zunächst der `Transaction`-Parameter mit der `transaction`-Referenz verglichen.

Falls es sich bei dem Vergleich um ein und das selbe Objekt handelt, wird der während der Transaktion unter Umständen geänderte Wert des Attributs `login` von `DBUsers` zurückgegeben, indem die `get`-Anfrage an das Datenobjekt weitergeleitet wird. Falls die beiden `Transaction`-Objekte unterschiedlich sind, wird der vor Ausführung der Transaktion gültige, konsistente Wert des `login`-Attributs der konzeptionellen Klasse `Users` zurückgegeben.

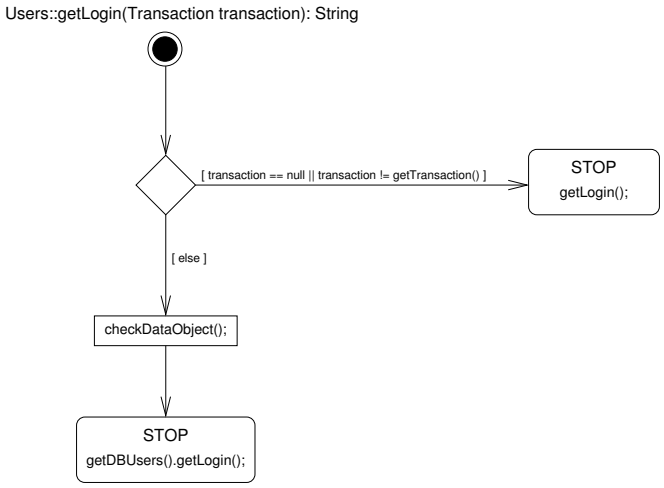


Abbildung 5.10.: getLogin-Methode mit Transaktions-Parameter

Die Zugriffsschicht ist so aufgebaut worden, daß bei ihrer Benutzung davon ausgegangen werden kann, daß bereits nach der Instanziierung eines konzeptionellen Objektes das zu ihm gehörende Datenobjekt vorliegt und seine Datenreferenz gesetzt ist. Deshalb wird bei der Verwendung der Zugriffsschicht ausgeschlossen, daß zu der Lebzeit eines konzeptionellen Objekts seine Datenreferenz `null` ist.

Die Methode `checkDataObject` ist eine Sicherheitsabfrage, um darauf aufmerksam zu machen, daß ein gravierender Fehler in der Zugriffsschicht vorliegt. Diese Methode überprüft die Referenz zu dem Datenobjekt, ob sie wirklich gesetzt ist. Falls nicht, wird die Schreibaktion mit dem Wurf einer Exception abgebrochen.

Die Transaktions-Objekte aus der Zugriffsschicht verwenden ausschließlich die `get`-Methoden mit dem `Transaction`-Parameter. Die get-Methode ohne Parameter dient für die Zusicherung, daß das Transaktions-Objekt immer den konsistenten Wert des Attributs zurückgibt. Das ist gerade dann sinnvoll, wenn in einer

Transaktion die Anforderung besteht, niemals auf einem noch nicht konsistenten Wert zu arbeiten, auch wenn der Wert innerhalb der gleichen Transaktion geändert wurde.

Der Nebeneffekt, der aus der Bereitstellung dieser Methode resultiert, ist die Möglichkeit, Objekte aus der Zugriffsschicht lesend in den Activity-Diagrammen von FUJABA zu verwenden, ohne Anpassungen an FUJABA vornehmen zu müssen.

schreibende Zugriffs-Methoden

Von den `beiden` in der Abbildung 5.9 gezeigten `set`-Methoden ist nur die Methode mit dem `Transaction`-Parameter öffentlich. Damit muß diese Methode benutzt werden, wenn der Wert von dem Attribut `login` geändert werden soll. Wie in der Abbildung 5.11 zu sehen, wird innerhalb dieser set-Methode zuerst der Transaktions-Parameter überprüft und danach das zu dem `User`-Objekt gehörende Datenobjekt.

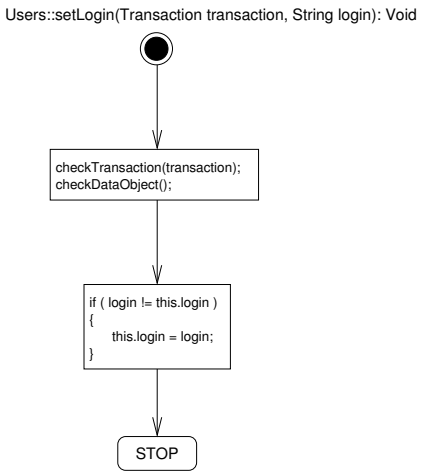


Abbildung 5.11.: setLogin-Methode mit Transaction-Parameter

Die Methode `checkTransaction` wirft eine Exception, wenn der übergebene Parameter `null` ist, oder, wenn das Objekt bereits vorher einem anderen `Transaction`-Objekt zugewiesen wurde. Dabei wird die bereits vorher vorgestellte `transaction`-Referenz überprüft.

Nachdem diese Tests erfolgreich durchgeführt wurden, kann das `login`-Attribut des Datenobjektes mit Aufruf der Methode `setLogin` geändert werden. Auch diese Methode kann noch die Änderung abbrechen, indem es eine Exception wirft. Seine `set`-Methode wird später bei der Vorstellung der Datenseite der Zugriffsschicht näher vorgestellt.

Die zweite, diesmal aber private, `set`-Methode mit nur einem `String`-Parameter wird innerhalb der `commit`-Methode eines `Users`-Objektes verwendet, wenn die Transaktion, an dem das Objekt beteiligt ist, erfolgreich beendet wird. Diese Methode wird benötigt, um den neuen konsistenten Zustand des Datenobjektes zu übernehmen.

Zugriffsmethoden für Assoziationen

Das an den Zugriffsmethoden für Attribute vorgestellte Verfahren gilt auch für die Zugriffsmethoden für die Assoziationen zwischen persistenten Klassen. Diese Diplomarbeit berücksichtigt keine Assoziationen zwischen persistenten und transienten Klassen, die nicht in einer Datenbank beispielsweise abgespeichert werden können.

In der Abbildung 5.12 ist die Assoziation `hasRoles` zwischen `Users` und `Roles` und die zu dieser Assoziation gehörenden Zugriffsmethoden zu sehen.

Die Datenklassen sind passive Klassen. Sie werden von der objektorientierten Zugriffsschicht gesteuert. Sie dienen nur dem Abspeichern in die Datenbank. Für die Datenmanipulation müssen sie das notwendige Datenbank-Statement erzeugen und absetzen. Die Dateninformationen, die für das Statement wichtig sind, werden in den Attributen der Datenklassen gehalten. Damit wissen sie aber immer noch nicht, was sie für eine Datenbank-Anweisung erzeugen müssen. Hierzu dient die Schnittstellenklasse `ResourceInterface`, von der alle DB-Schnittstellen erben. Hierüber kann das `Users`-Objekt dem Datenobjekt mitteilen, was für eine Datenmanipulation am Ende einer Transaktion ausgeführt werden muß.

Generell ist festzuhalten, daß die Zugriffsmethoden einer Assoziation, mit Ausnahme der schreibenden Zugriffsmethoden mit nur einem Parameter, öffentlich sind. Die schreibenden Zugriffsmethoden ohne `Transaction`-Parameter sind nur eingeschränkt sichtbar.

Die Klassen, die an einer Assoziation beteiligt sind, müssen die zu der Assoziation gehörenen Zugriffsmethoden mit nur einem Parameter ihrer Partnerklasse kennen, da die Methoden zwar, wie bei den intern schreibenden Zugriffsmethoden der Attribute, innerhalb der `commit`-Methode verwendet werden, die Assoziation aber, wie in FUJABA, bidirektional realisiert ist, und damit innerhalb der einen schreibenden Methode die korrespondierende Methode der Gegenseite bekannt sein muß.

Wie bereits in den vorigen Kapiteln erwähnt, soll die Zugriffsschicht einen Cache auf die Datenbank darstellen. Die konzeptionellen Objekte müssen dafür also den konsistenten Zustand der Datenbank repräsentieren. Da es aus praktischen Gründen inakzeptabel ist, alle Tupel der Datenbank gleich bei der Initialisierung der Zugriffsschicht objektorientiert darzustellen, verwenden die Objekte der

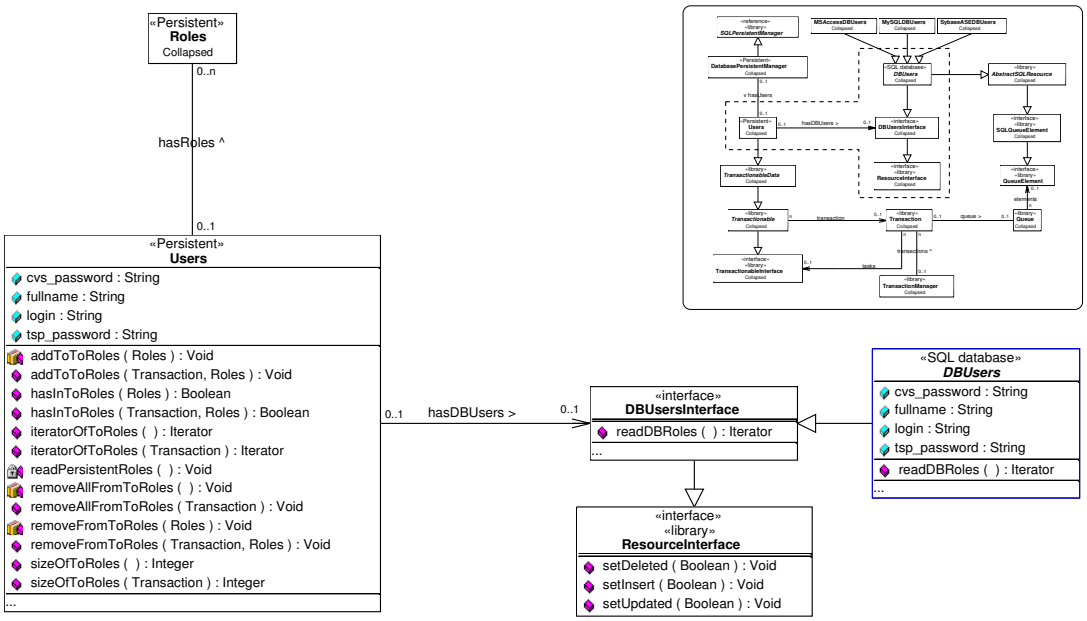


Abbildung 5.12.: Zugriffsmethoden für Assoziationen

Zugriffsschicht bei der Verwaltung ihrer Assoziationen zu anderen Objekten das Prinzip des *smart-pointers.*

Hierbei werden die Objekte, mit denen ein Objekt über eine Assoziation in Beziehung steht, erst bei Bedarf geladen. Sobald die Assoziation `hasRoles` über eine ihrer Zugriffsmethoden angesprochen wird, überprüft das `Users`-Objekt intern, ob es bereits seine `Roles`-Objekte aus der Datenbank geladen hat bzw. schon kennt.

Falls seine `Roles`-Objekte in der objektorientierten Zugriffsschicht noch nicht zur Verfügung stehen, ruft es intern die Methode `readPersistentRoles` auf. Die Methode readPersistentRoles ist in Abbildung 5.13 zu sehen. Innerhalb der `readPersistentRoles`-Methode wird im wesentlichen die Methode `readDBRoles` auf der Datenklasse aufgerufen, die über die Schnittstelle `DBUsersInterface` bekannt ist.

Die Methode der Datenklasse stellt eine Anfrage an die Datenbank, die genau die `roles`-Tupel zurückliefert, die über die konzeptionelle Assoziation von dem ausgehenden `Users`-Objekt erreichbar sind. Die Tupel werden intern ausgewertet, und es werden dabei die Datenobjekte für die später noch zu erzeugenden `Roles`-Objekte instanziiert.

Von der `readDBRoles`-Methode der Datenklasse erhält das `Users`-Objekt einen `Iterator`, in dem sich alle Datenobjekte befinden. Dieser `Iterator` wird von dem `Users`-Objekt abgearbeitet, indem es zu jedem `DBRoles`-Objekt ein `Roles`-Objekt erzeugt und dem `Roles`-Objekt sein `DBRoles`-Objekt zuordnet.

Die `Roles`-Objekte werden in dem, zu der Assoziation gehörenden, Rollenattribut des Objektes `Users` gehalten. Nach diesem einmaligen Auslesen können die kor-

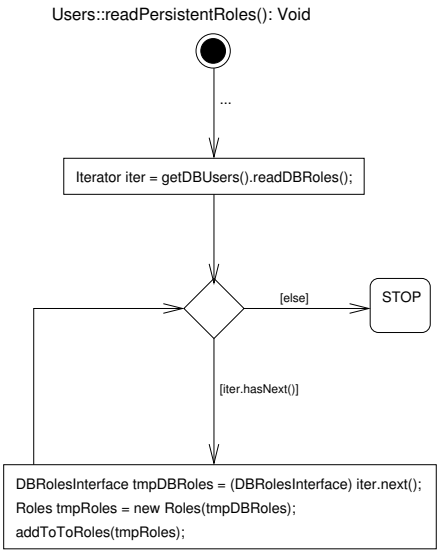


Abbildung 5.13.: readPersistentRoles in der Klasse Users

respondierenden `Roles`-Objekte beim nächsten Zugriff direkt aus dem Speicher verwendet werden.

Bei einer Änderung an einer Assoziation muß die Isolationspflicht des Objektes innerhalb einer Transaktion erweitert werden. Bei dem Hinzufügen eines `Roles`-Objektes über die Methode `addToToRoles` darf das neue Objekt erst nach Beendigung der Transaktion über die Assoziation erreichbar sein. Vorher gilt, wie es die Isolationspflicht der Transaktion vorschreibt, der bis dahin konsistente Zustand.

Das, wie bei den Attributen durchgeführte, Verfahren, die Änderungen erst auf der Datenseite vorzunehmen, würde bedeuten, daß auch zwischen den Klassen `DBUsers` und `DBRoles` eine Assoziation bestehen müßte. Diese Architektur ist zwar möglich und auch realisierbar. Sie würde aber auch für einen erheblichen Aufwand bei einem `commit` sorgen.

Die Elemente aus den beiden Assoziationen müßten bei einem `commit` synchronisiert werden. Es müßten alle hinzugefügten und gelöschten Elemente ermittelt werden. Um dies zu erreichen, müssen die Mengen jeweils einmal durchlaufen und kontrolliert werden, ob die Elemente noch in beiden Mengen vorkommen.

Beim Durchlauf auf konzeptioneller Seite würden alle in der Transaktion gelöschten `Roles` aus der `hasRoles`-Assoziation gelöscht (das Objekt ist auf der Datenseite nicht mehr da). Beim Durchlauf auf logischer Seite werden neue Objekte der Assoziation hinzugefügt (das Objekt ist auf der konzeptionellen Seite noch nicht da). Es würde bei einem `commit` die in Abbildung 5.14 gezeigte Vorbedingung gelten.

Die einfachen Kästen repräsentieren hierbei Objekte. Die Typen der Objekte sind in dem Text innerhalb der Kästen angegeben. Die Objekte, die durch die einfa-

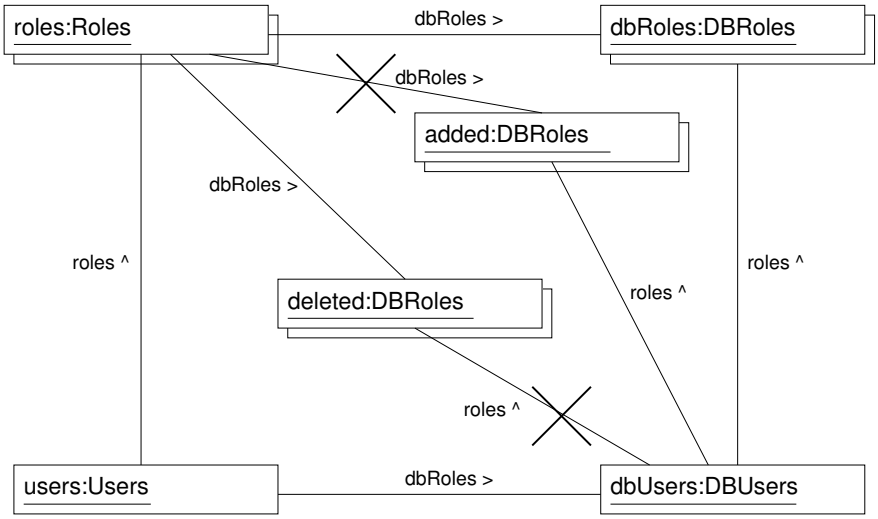


Abbildung 5.14.: commit-Vorbedingung bei Assoziation auf Datenseite

chen Kästen dargestellt sind, müssen mit der, durch eine Linie repräsentierten, Verbindungen dazwischen existieren. Die doppelten Kästen repräsentieren Mengen, die auch leer sein können. Dabei wird eine Menge über eine Verbindung zu einem anderen Objekt oder einer anderen Menge aufgespannt. Die durchgestrichenen Verbindungen dürfen zwischen den Objekten nicht existieren.

Derzeit ist die Datenseite so realisiert, daß sie vollständig von der konzeptionellen Seite gesteuert wird. Wenn ein konzeptionelles Objekt wie `Users` seinem Datenobjekt befiehlt ,,Hole mir ein Datum aus der Datenbank", gehorcht das Datenobjekt und liest Werte aus der Datenbank aus. Das Resultat wird in einem neu erzeugten Datenobjekt an das `Users`-Objekt zurückgegeben, da das Datenobjekt nicht weiß, ob es bereits eine objektorientierte Darstellung zu den gelesenen Tupeln aus der Datenbank gibt.

Die Kontrolle, daß keine Daten doppelt in der objektorientierten Zugriffsschicht vorliegen, liegt auf Seiten der konzeptionellen Objekte.

Bei der oben dargestellten Lösung müßte das Datenobjekt so angepaßt werden, daß es über das Wissen verfügt, welche Daten bereits aus der Datenbank ausgelesen wurden. Die Datenelemente, die über die Daten-Assoziation erreichbar sind, müssen genau denen entsprechen, die auf konzeptioneller Seite über die Assoziation erreichbar sind, damit ein Vergleich, wie in Abbildung 5.14 gezeigt, überhaupt möglich ist.

Deswegen verwaltet das konzeptionelle Objekt seine Assoziationen zu anderen Objekten selbst. Die bereits bekannte abstrakte Klasse `Transactionable` bietet zwei nicht nach außen sichtbare Mengen an, um Änderungen an Beziehungen zu anderen Objekten abspeichern zu können. In der einen Menge werden die aus einer Beziehung gelöschten Objekte gehalten und in der anderen die hinzugefügten.

Anhand der Instanz eines Objektes aus diesen Mengen kann das Objekt auf die Assoziation schließen, die bei einem `commit` von der Änderung betroffen ist.

Da das konzeptionelle Objekt davon ausgehen kann, daß es vor Transaktionsbeginn einen konsistenten Zustand darstellte, muß es während der Transaktion nur die Änderungen an sich protokollieren, um sich bei einem `commit` wieder in den konsistenten Zustand, der in der Datenbank vorliegt, zu bringen.

5.5. Datenbankunabhängigkeit

Bisher wurde die Datenseite als allgemeine Datenbank-Zugriffs-Möglichkeit vorgestellt, und der bisher gezeigte Ausschnitt ist das auch. Bei der Benutzung der Zugriffsschicht muß aber auf eine reale Datenbank zugegriffen werden, wie die Datenbanken MySQL oder Sybase Adaptive Server 11.9.2 es sind.

In der Abbildung 5.15 ist die Datenseite der Zugriffsschicht, die von der konzeptionellen Seite indirekt über die Schnittstelle benutzt wird, zu sehen.

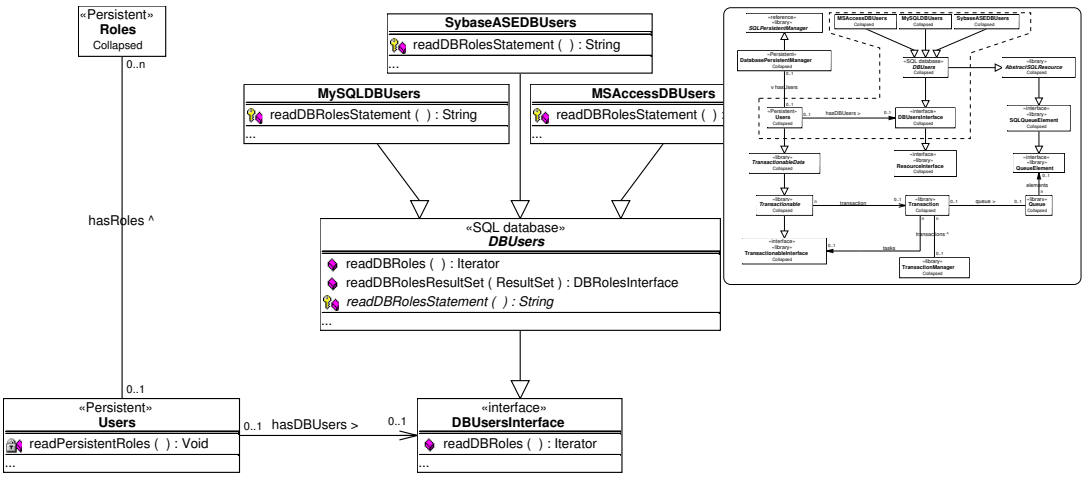


Abbildung 5.15.: Datenbankspezifische Klassenhierarchie am Beispiel Users

Zu jeder, in REDDMOM bekannten, Datenbank wird eine eigene Klasse erzeugt, die die datenbankspezifischen Eigenschaften ihrer Datenbank kapselt. Vorwiegend sind das die für die Kommunikation mit dem Datenbank-Server benötigten SQL-Statements, um eine lesende bzw. schreibende Aktion ausführen zu können.

Die schreibenden Aktionen werden im nächsten Kapitel bei der Vorstellung des Verfahrens bei Transaktions-Ende vorgestellt. Dieses Kapitel erläutert zunächst einmal nur die lesenden Aktionen.

Die `readDBRoles`-Methode der Klasse `DBUsers` wird vom `Users`-Objekt aufgerufen, wenn zum ersten Mal auf seine `hasRoles`-Assoziation zugegriffen wird. Das Lesen

muß sofort erfolgen, da das Users-Objekt bei dem Aufruf `addToToRoles` zum Beispiel ohne die vorhandenen Objekte nicht entscheiden kann, ob es sich wirklich um eine Neuanlage einer Rolle handelt oder nicht. Die lesenden Aktionen werden sofort ausgeführt und werden nicht auf einen späteren Zeitpunkt verschoben.

Wie bereits erwähnt und beschrieben, verfügt die Klasse `DBUsersInterface` über die Methodendefinition `readDBRoles`, die von dem konzeptionellen Objekt beim Auslesen der `hasRoles`-Assoziation aufgerufen wird und von der allgemeinen Datenbank-Klasse `DBUsers` ausimplementiert wurde.

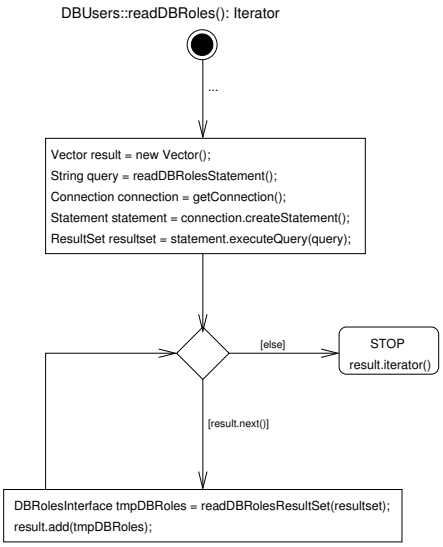


Abbildung 5.16.: Die Methode readDBRoles() aus der Klasse DBUsers

Die Klasse `DBUsers` definiert die abstrakte Methode `readDBRolesStatement`, die innerhalb der `readDBRoles`-Methode aufgerufen wird. Die abstrakte Methode dient dazu, den expliziten Datenbank-Befehl in Form eines `String`s zu erzeugen. Da die Datenbanken unterschiedliche Dialekte haben, kann dieser `String` nicht als allgemein gleich verstanden und deshalb nicht als Konstante angesehen werden. Er wird in der eigentlichen Lese-Methode offen gelassen und erst durch eine, von der `DBUsers`-Klasse erbende, Klasse ausimplementiert.

Da die `DBUsers`-Klasse abstrakt ist, kann es keine konkreten Instanzen von ihr zur Laufzeit geben. Es muß also eine Instanz von einer datenbankspezifischen Klasse erzeugt werden. Es ist also dafür gesorgt, daß die Methode `readDBRolesStatement` ausimplementiert ist und eine für die Datenbank passende SQL-Anfrage erzeugt. Dabei muß darauf geachtet werden, daß nur von einer Datenbank-Art Instanzen erzeugt werden. Bei der Verwendung der Datenbank MySQL dürfen natürlich nur `MySQLDBUsers`-Instanzen erzeugt werden. Diese Frage wird bei der Vorstellung der datenbankspezifischen Datenbank-Manager behandelt.

Die `readDBRolesResultSet`-Methode wird ebenfalls von der `readDBRoles`-Methode

aufgerufen. Sie dient nur internen Zwecken, um das Auslesen eines ResultSets übersichtlicher zu gestalten.

Die Methode `getConnection()` in der Abbildung 5.16 ist eine noch nicht erwähnte Methode aus der in dieser Diplomarbeit entstandenen Bibliothek, die jeder DB-Klasse zur Verfügung steht. Sie dient dazu, sich ein Objekt vom Typ `java.sql.Connection` zu besorgen. Die Klasse `java.sql.Connection` wurde in Kapitel 2.1.4 vorgestellt. Die Methode greift auf einen bisher noch nicht vorgestellten Datenbank-Manager zurück, der alle `java.sql.Connection`-Objekte verwaltet. Der Datenbank-Manager ist ebenfalls in der Bibliothek enthalten und wird im weiteren Verlauf der Diplomarbeit noch vorgestellt.

Die übrigen, in der Abbildung 5.16 zu sehenden, Methoden sind Methoden des JDBC-Pakets, das in Kapitel 2.1.4 vorgestellt wurde, und von den JDBC-Treibern ausimplementiert sein müssen, die innerhalb der objektorientierten Zugriffsschicht verwendet werden sollen. Bei diesen Methoden handelt es sich aber um Basismethoden, die es ermöglichen, überhaupt auf eine Datenbank zuzugreifen. Deshalb kann man davon ausgehen, daß alle für JDBC erhältlichen Treiber die in Abbildung 5.16 gezeigten Methoden ausimplementiert haben.

5.6. Ausführen des commit

Die Abbildung 5.17 zeigt die Struktur der objektorientierten Zugriffsschicht, die erforderlich ist, um eine offene Transaktion abzuschließen. Der fehlerhafte Abschluß, der die objektorientierte Zugriffsschicht veranlaßt, sich in den Zustand vor Transaktionsbeginn zu bringen, erfordert keine weitere Logik, da die Historieverwaltung bei den konzeptionellen Objekten liegt und sie davon ausgehen können, daß im Fehlerfall ihr Zustand in der Datenbank nicht verändert wurde.

Aus Gründen der Übersicht wurde der in der Legende links oben dargestellte `SQLPersistentManager` in der großen Darstellung nach rechts unten verschoben. Ebenfalls aus Gründen der Übersicht wurde in der Gesamtdarstellung der `SQLResourceManager` nicht aufgeführt. Dieser Manager stellt unter anderem die in Kapitel 5.5 vorgestellte `Connection`-Verwaltung bereit und wird zum Abschluß dieses Kapitels noch näher vorgestellt.

Der `TransactionManager` und die `Transaction` sind bereits aus den vorigen Kapiteln bekannt, sodaß sie nicht noch einmal näher vorgestellt werden müssen.

Die ebenfalls schon erwähnte `Queue` dient dazu, alle Änderungen während einer Transaktion aufzusammeln.

Beim Beenden einer Transaktion mittels `commit` wird dem `SQLResourceManager` die `Queue` des `Transaction`-Objektes übergeben, der die `Queue` daraufhin abarbeitet. Ehe er damit anfängt, sorgt er dafür, daß er eine funktionierende Verbindung

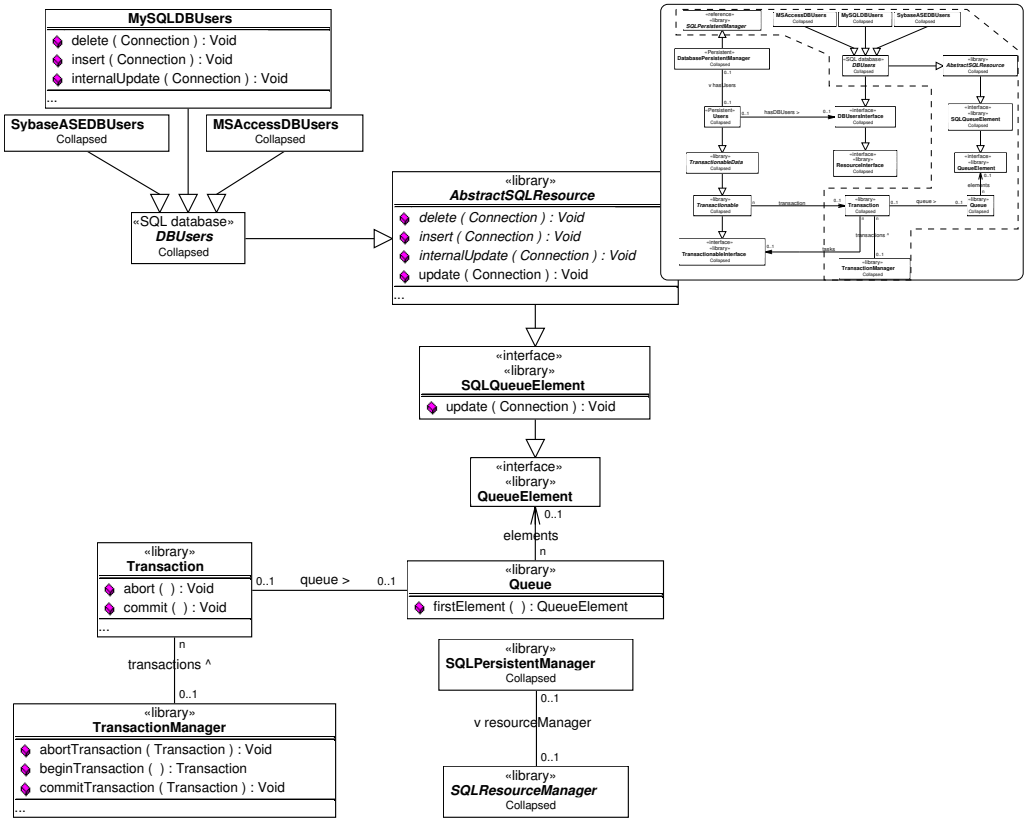


Abbildung 5.17.: Transaktion abschlieSSen

(`Connection`) zu seiner Datenbank hat. Die `Queue` beinhaltet `QueueElemente`, von denen sich der `SQLResourceManager` immer das erste über die Methode `firstElement()` sich geben läßt. Die Methode `firstElement()` löscht daraufhin das erste Element aus der `Queue`, damit kein Element doppelt abgearbeitet wird. Derzeit ist realisiert, daß innerhalb der `Queue` nur gleiche Elemente, die von `QueueElement` erben, enthalten sind.

Der `SQLResourceManager` überprüft das erhaltene `QueueElement` darauf, ob es von der Instanz `SQLQueueElement` ist. Wenn das der Fall ist, ruft er die Methode `update(Connection)` auf dem Element auf. Von dem `SQLQueueElement` erbt die abstrakte Klasse `AbstractSQLResource`, die nicht nur von der `SQLQueueElement`-Schnittstelle sondern auch von der bereits bekannten `ResourceInterface`- Schnittstelle erbt und die drei flag-Attribute `inserted`, `deleted` und `updated` zur Verfügung stellt. Dadurch, daß das konzeptionelle Objekt die Flags über die `ResourceInterface`-Schnittstelle gesetzt hat, kann das Datenobjekt bei der Ausführung der `update`-Methode entscheiden, welche Aktion ausgeführt werden muß. Je nachdem, was getan werden muß, ruft es die abstrakten Methoden `delete`, `insert` oder `internalUpdate` auf. Diese drei Methoden müssen wegen der Datenbanknähe direkt von den datenbankspezifischen Klassen ausimplementiert werden.

5.7. Mr. DOLittle

Nach der Fertigstellung der Zugriffsschnittstelle will man sie auch benutzen. Vor der Benutzung muß sie allerdings getestet werden. Hierfür stellt FUJABA Mr.DOBS bereit, der alle Objekte, die während der Laufzeit einer Anwendung bzw. eines Systems erzeugt wurden, visuell darstellt. Zu einem Objekt kann man seine Attribut-Werte sehen. Aus der ebenfalls visuell dargestellten Methodenschnittstelle des Objekts können Methoden ausgesucht und ausgeführt werden. Mr.DOBS stellt die Objekte mit ihren Assoziationen als Graph dar. Die Objekte erscheinen als Kasten und ihre Referenzen zu anderen Objekten als Linien. Mit Mr.DOBS hat man eine sehr komfortable Testmöglichkeit während der Ausführung des Systems.

Durch die Darstellung der Objekte und Assoziationen als Graph ist Mr.DOBS aber in der Menge sehr begrenzt, wenn es um die Übersichtlichkeit geht. Mr.DOBS bietet, wenn man ein Objekt aus dem Graphen ausgewählt hat, die Funktion ,,Expand Object" an. Diese Methode versucht, alle zu dem ausgewählten Objekt in Beziehung stehenden Objekte zu finden und anzuzeigen. Wenn es sich dabei um eine überschaubare Menge handelt, ist Mr. DOBS vollkommen ausreichend.

Bei einem persistenten Objekt allerdings besteht die Gefahr, daß man plötzlich unzählig viele Objekte erhält, weil das persistente Objekt bei Aufforderung sämtliche ihm bekannten Objekte aus der Datenbank holt. Die Darstellung von Mr.DOBS wird unübersichtlich und nicht beherrschbar.

Außerdem sind die persistenten Objekte keine gewöhnlichen Objekte, die man einfach anlegen kann. Um ein persistentes Objekt erzeugen bzw. von der Zugriffsschnittstelle bekommen zu können, muß man sich wie bei einer herkömmlichen Datenbank bei der Zugriffsschicht anmelden. Außerdem unterstützt Mr.DOBS keine Transaktionen. Sie müssen in Mr.DOBS manuell erzeugt, und eine Transaktion muß einem persistenten Objekt zugewiesen werden, wenn eine Änderung auf dem Objekt ausgeführt werden soll.

Diese manuelle Verwaltung wäre zwar in Mr.DOBS möglich aber unpraktikabel.

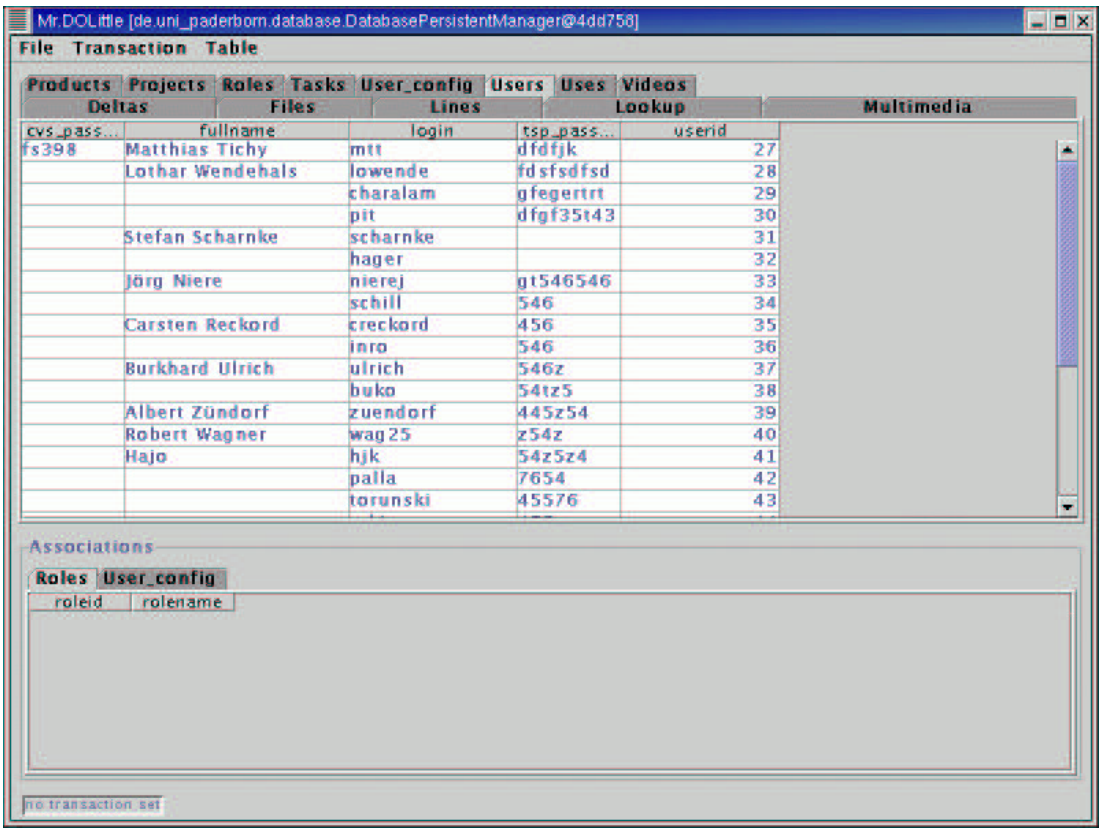


Abbildung 5.18.: Mr.DOLittle

Während dieser Diplomarbeit ist ein Aufsatz auf Mr.DOBS entwickelt worden, der für die Objekte aus der Zugriffsschicht eine überschaubarere Ansicht in Tabellenform bereitstellt und für die Anmeldung bei der Datenbank und für die Verwaltung der Transaktionen ein Menüsystem anbietet. In der Abbildung 5.18 ist die Applikation zu sehen. Aus dem Namen ,,Mr. DOBS and a Little bit more" ist zu entnehmen, daß diese Applikation zunächst nur als Frontend für Mr.DOBS gedacht ist. Es soll die Objekte in tabellarischer Form darstellen. Vorwiegend ist dieses Frontend dazu gedacht, ein Objekt aus der Tabelle an Mr.DOBS weiterzugeben, um es dort wie bisher zu analysieren. Die Abbildung 5.19 zeigt zum

Beispiel ein Users-Objekt aus der objektorientierten Zugriffsschicht und seine zugehörige Rolle in Mr.DOBS.

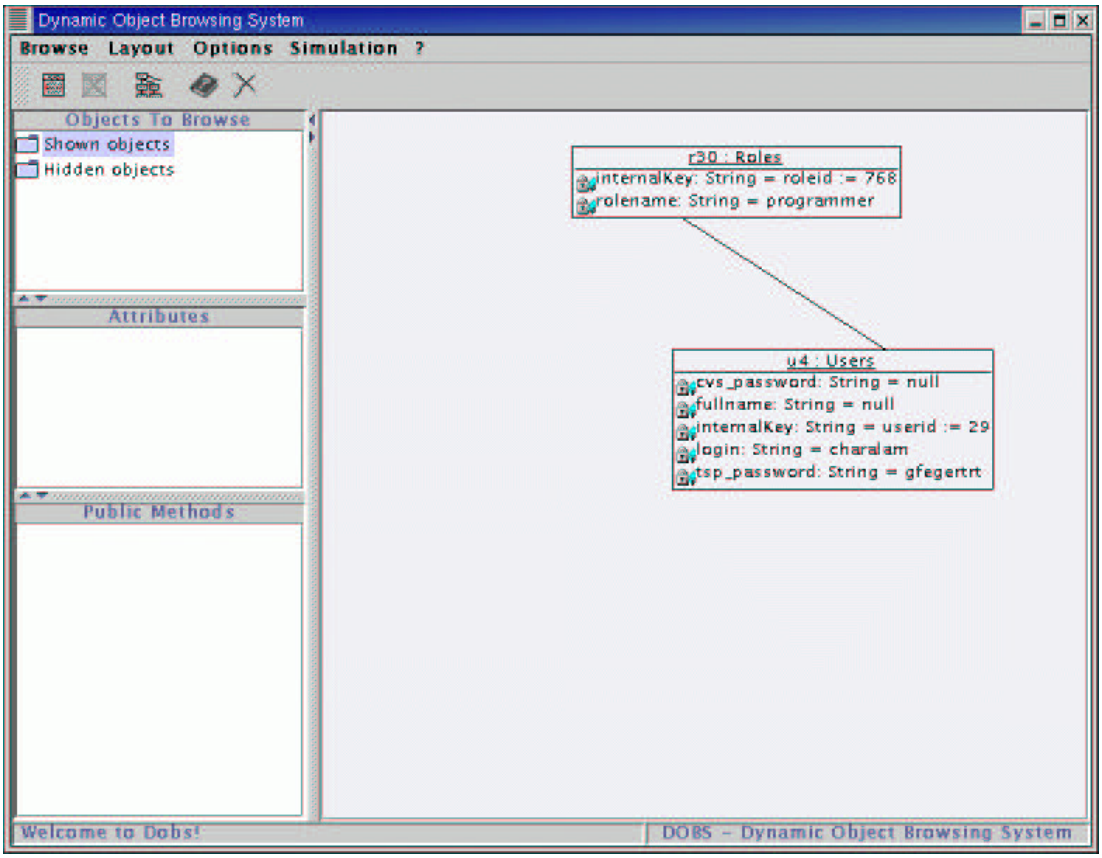


Abbildung 5.19.: Mr. DOBS

Mr.DOLittle sucht beim Start nach speziellen Klassen und Methoden innerhalb der Zugriffsschicht, um sie als Tabellen darstellen zu können. Dadurch kann Mr.DOLittle nur Zugriffsschichten anzeigen, die mit dem hier vorgestellten Generator erzeugt wurden. Die Struktur der hinter der Zugriffsschicht befindlichen Datenbank spielt aber keine Rolle.

6. Zusammenfassung und Ausblick

Diese Diplomarbeit erweitert das Projekt FUJABA um die Fähigkeit, eine Verbindung zu einer relationalen Datenbank aufzubauen, die Struktur der Datenbank auszulesen und als logisches Schema in Form eines EER-Diagramms anzuzeigen. Aus dem erhaltenen Schema kann ein initiales konzeptionelles Schema in Form des FUJABA UML-Diagramms erzeugt werden. Der in dieser Diplomarbeit vorgestellte Generator erzeugt eine objektorientierte Zugriffsschicht, die mit Hilfe der Mr. DOBS Erweiterung Mr. DOLittle getestet werden kann. Die Zugriffsschicht unterstützt bei der Änderung von Daten das aus der Datenbankwelt bekannte ACID - Transaktionskonzept. Die öffentliche Schnittstelle zu der Zugriffsschicht ist vollkommen datenbankunabhängig. Die datenbankabhängigen Einstellungen sind intern in eigenen Paketen für jede Datenbank gekapselt und prinzipiell austauschbar.

Die hier vorgestellte Arbeit ist die Grundlage für das kommende Projekt REDDMOM. Diese Arbeit wird in ihrer Funktionalität um die Einbeziehung von verteilten Datenbanken und um die konzeptionelle Einbeziehung von Multimedia - Objekten erweitert.

In diesem Zusammenhang werden Konzepte aus dem alten VARLET - Projekt durch Neuimplementierung übernommen. Die Konzepte werden hier nicht weiter aufgezählt.

Erweiterungen zu VARLET sind in dem neuen Projekt REDDMOM neben den oben erwähnten die Einbeziehung von objektorientierten Datenbanken[Wag01] und das Internet.

Die objektorientierte Zugriffsschicht kann dahingehend erweitert werden, daß die übrigen drei Design-Pattern der vier vorgestellten aus dem Kapitel 5 mit in die Generierung einbezogen werden. Wegen dem Ziel von REDDMOM, auf verteilte Datenbanken zugreifen zu können, sind hier vor allem das Composite-Transaction- und das Two-Phase Commit Pattern interessant.

Die Funktionalität des EER-Diagramms kann dahingehend erweitert werden, daß vorgenommene Änderungen an dem relationalen Schema in die Datenbank zurückgeschrieben werden. Außerdem können die in der Diplomarbeit erwähnten ausgelassenen Implementierungen nachgeholt werden.

Für die weitere Zukunft kann man sich natürlich vorstellen, Mr.DOLittle um weitere Datenbank-Funktionalitäten zu erweitern, sodaß eine Art Administrationstool entsteht. Dabei könnte die Zugriffsschicht um weitere Funktionalitäten angereichert werden.

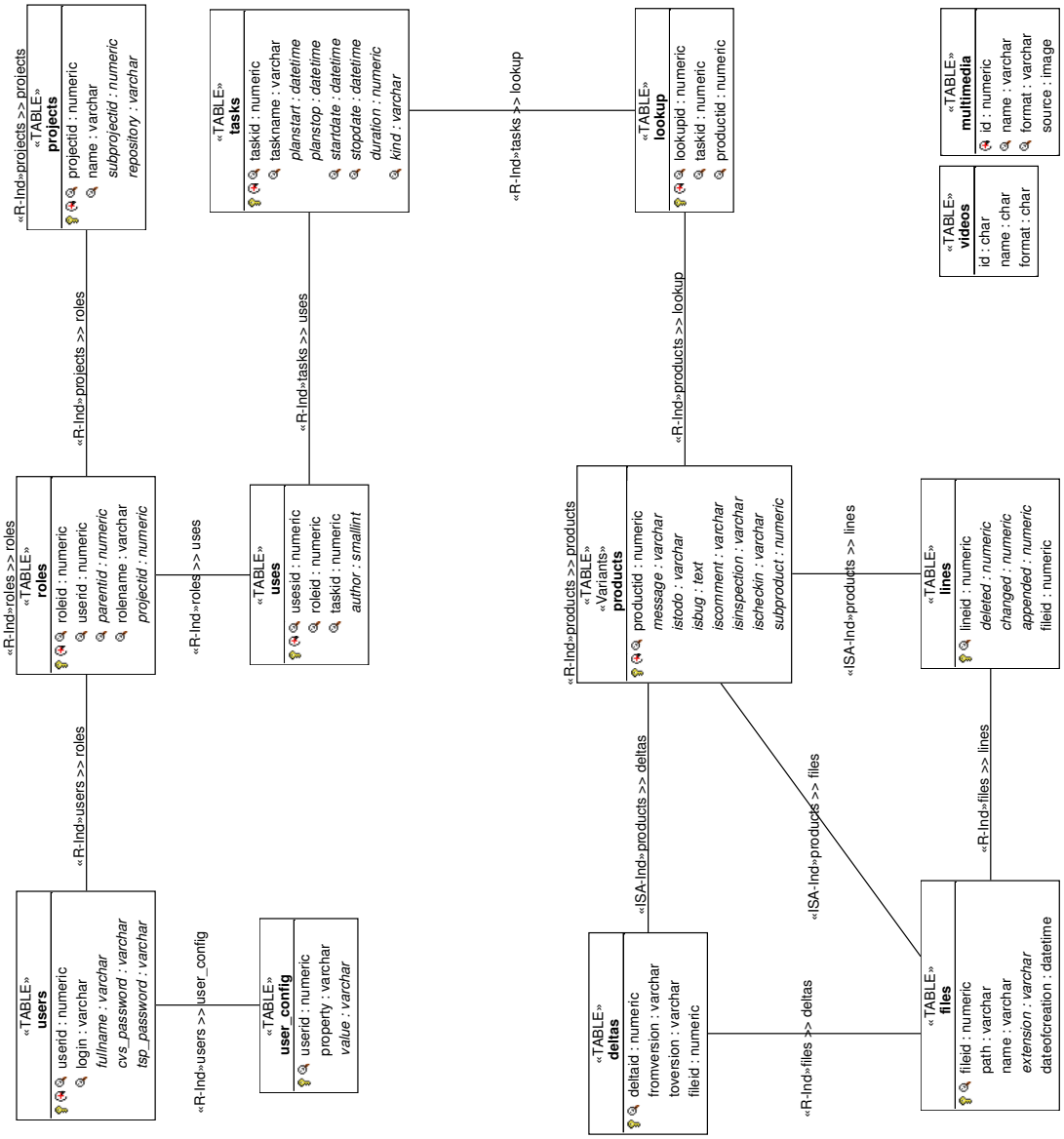
Bisher kann nur die Struktur einer konkreten relationalen Datenbank ausgelesen und in ein konzeptionelles Schema transformiert werden. Es muß auf Dauer aber auch möglich sein, eine bestehende Applikation um ein persistentes Speicherungskonzept zu erweitern, indem auf konzeptioneller Seite einfach eine Klasse als persistent markiert und aus allen persistenten Klassen ein initiales logisches Schema erstellt wird.

Dadurch, daß REDDMOM mit dieser Arbeit ein abgeschlossenes Verfahren besitzt, eine objektorientierte Zugriffsschicht zu einer relationalen Datenbank zu generieren, können jetzt auch Fremdsysteme berücksichtigt werden. Stellvertretend für alle in diesen Bereich fallende Projekte sei hier wieder der aus Kapitel 3.1 bekannte ObjectDRIVER genannt.

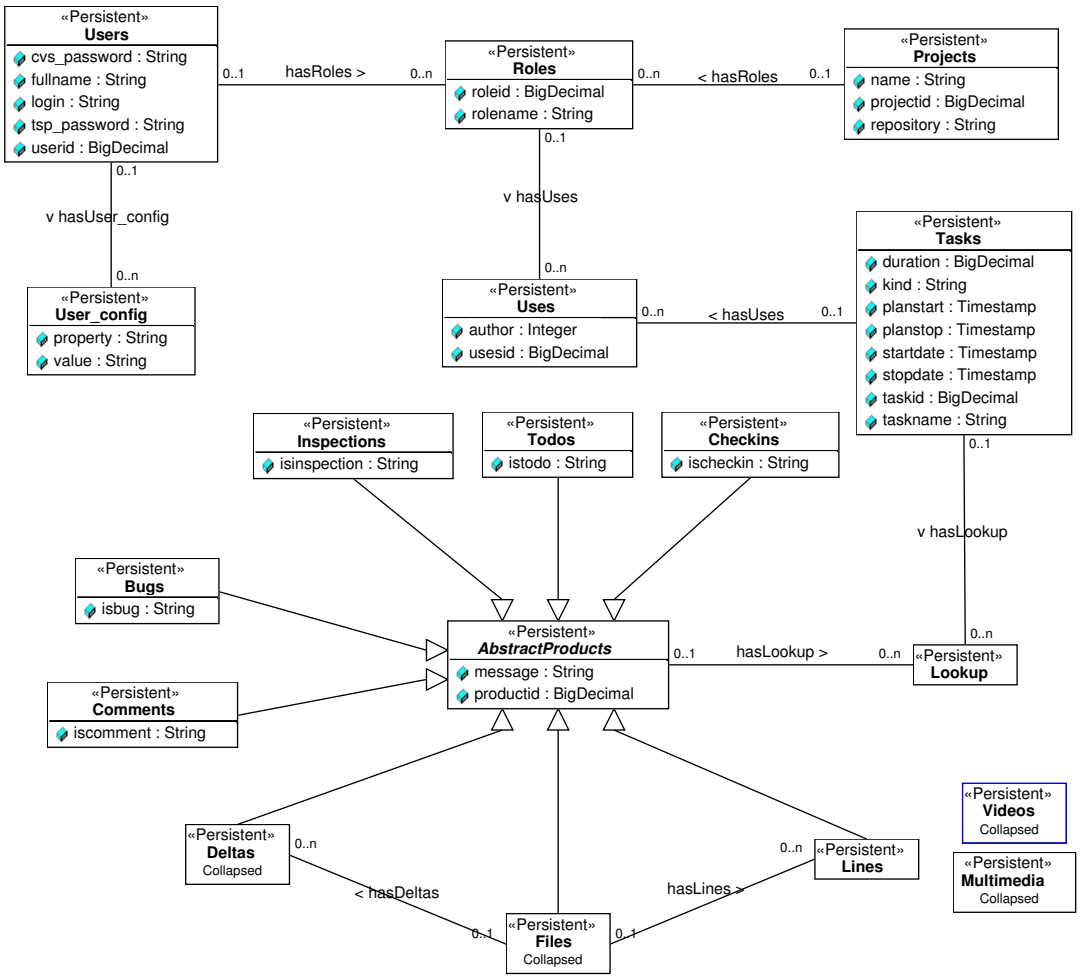
Damit auch die in Kapitel 3 vorgestellten Schnittstellen zum Zugriff auf Datenbanken unterstützt werden, könnten neben dem vorhandenen weitere Generatoren erstellt werden, die sich an diesen Schnittstellen orientieren.

A. Anhang

A.1. DSD - Logisches Schema (EER)



A.2. DSD - Konzeptionelles Schema (UML)



A.3. Übersicht über die Transformationsregeln

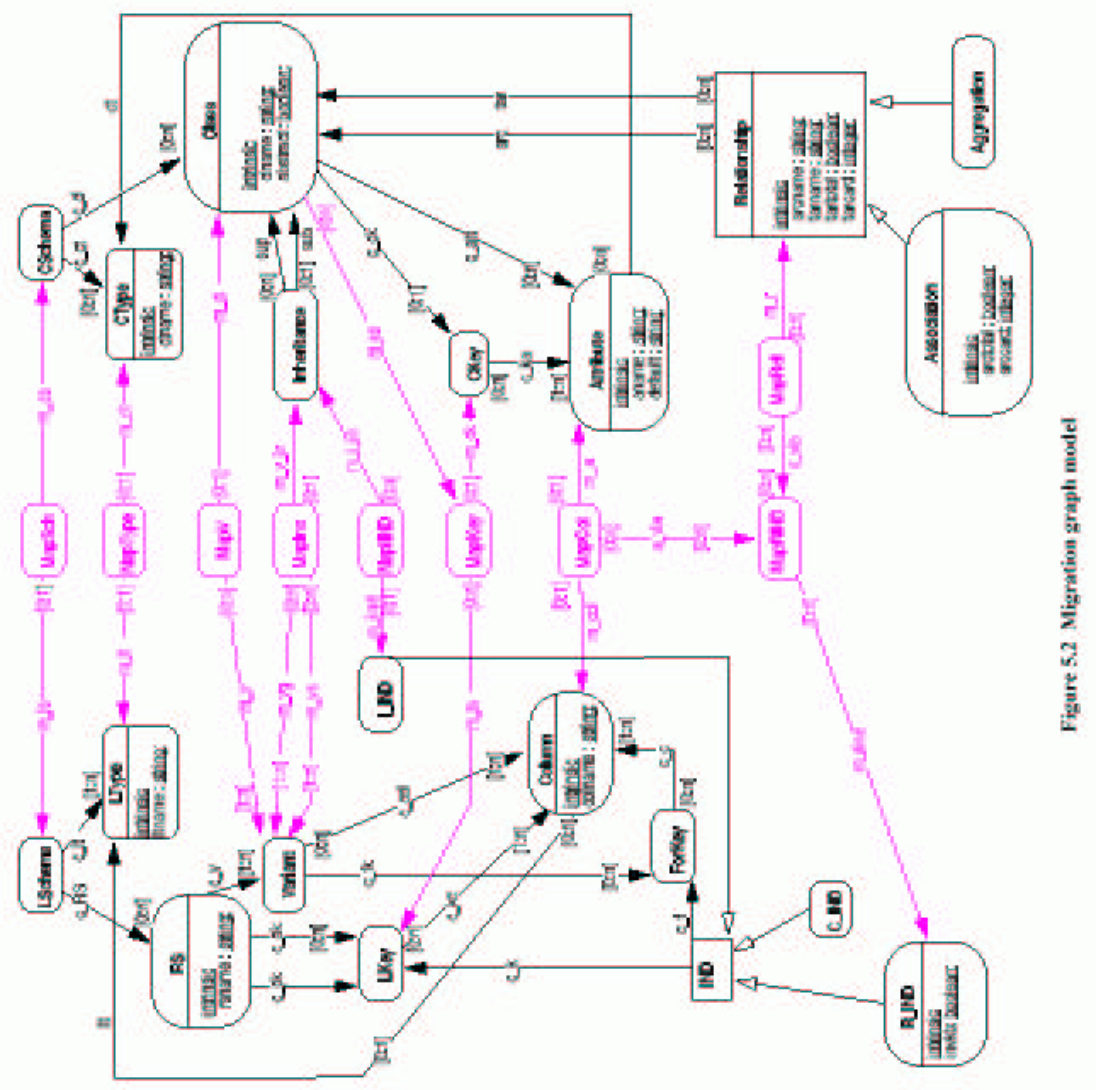


Figure 5.2 Migration graph model

Abbildungsverzeichnis

Literaturverzeichnis

[Bor] Borland, Scotts Valley, California, http://www.borland.com/. 30

[DRI] JDBC^{TM} Treiber-Liste von Sun Microsystems, http://industry.java.sun.com/products/jdbc/drivers. 17

[EN94] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. Benjamin/Cummings, Redwood City, 2nd edition, 1994. 9, 17, 22

[FNT98] T. Fischer, J. Niere, and L. Torunski. Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, July 1998. 69

[FUJ] FUJABA - From Uml to Java And Back Again, http://www.fujaba.de/. 4, 5, 12, 45, 69

[GNZ01] M. Gehrke, J. Niere, and A. Zündorf. Distributed Software Development using Jini. In *Proc. of the 4th International Workshop on Software Engineering over the Internet (SEoI), Toronto, Canada*, 2001. 45

[Gra99] Mark Grand. Transaction Patterns A Collection of Four Transaction Related Patterns. In *PLoP'99 Proceedings*. PLoP'99, 1999. 81

[J2E] Java^{TM} 2 Platform, Enterprise Edition SDK, http://java.sun.com/j2ee/docs.html. 29

[Jah99] J.H. Jahnke. *Management of Uncertainty and Inconsistency in Database Reengineering Processes*. PhD thesis, University of Paderborn, Paderborn, Germany, September 1999. 40, 41, 69, 73, 111

[JAV] Java^{TM} 2 Platform, Standard Edition SDK, http://java.sun.com/. 10, 17

[JDB] JDBC^{TM} Data Access API, http://java.sun.com/products/jdbc/index.html. 4, 7, 13, 17, 18, 49

[JDX] JDX, J-Database Exchange, http://www.softwaretree.com/products/jdx/. 30

[Mic] Microsoft Corporation, Redmond, http://www.microsoft.com/. 47

[Müh00] Jens H. Mühlenhoff. Integration und Konsistenzprüfung von XML Datenbeständen. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, 2000. 69

[OBJ] ObjectDRIVER, Infobjects S.A., 06220 Vallauris - FRANCE, http://www.infobjects.net/default/en/html/Evaluation.html. 30, 31, 32, 37, 111

[ODM] ODMG - Object Data Management Group, The Standard for Storing Objects, http://www.odmg.org/. 29

[ONT] ONTOS, Inc. ONTOS Object Integration Server for Relational Databases 2.0, Sche-ma Mapper User s Guide. OIS - 20-SUN-SMUG-1.0, July 1996. 30

[ORA] Oracle, http://www.oracle.com/. 47

[RED] REDDMOM - Reengineering of distributed (federated) databases for multimedia objectoriented middleware, http://www.uni-paderborn.de/cs/ag-schaefer/Lehre/PG/REDDMOM/index.html. 4, 5

[Ros] Rational. *Rose, the Rational Rose case tool. Online at http://www.rational.com.* 12

[Sch98] Heike Schaldach. Integration von JAVA-Anwendungen mit relationalen Informationssystemen. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, 1998. 8, 42, 43, 111

[SYB] Sybase, http://www.sybase.com/. 47

[TIN] tiny SQL - Project, http://www.jepstone.net/tinySQL/. 47

[Tog] Object International. *TogetherJ, the TogetherJ case tool. Online at http://www.togethersoft.com.* 12

[UMLa] Rational Software Corporation. *UML documentation version 1.3 (1999). Online at http://www.rational.com.* 10, 11, 13, 17

[UMLb] UML - Unified Modeling Language, http://www.omg.org/technology/documents/formal/uml.htm. 13, 17

[VAR] VARLET - Verified Analysis and Reengineering of Legacy database systems using Equivalence Transformations, http://www.uni-paderborn.de/cs/ag-schaefer/Lehre/PG/VARLET/index.html. 4, 5, 69

[Wad98] J.P. Wadsack. Inkrementell Konsistenzerhaltung in der transformationsbasierten Datenbankmigration. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, 1998. 5, 42, 69

[Wag01] Sergej Wagner. Datenbank-Erweiterungen für multimediale Anwendungen, voraussichtlich abgeschlossen Oktober 2001. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, 2001. 72, 105