DIPLOMARBEIT

Werkzeuge zur Optimierung erweiterter Entity–Relationship Modelle und deren Abbildung in ein relationales Datenbankschema

Monika Schneider

UNIVERSITÄT DORTMUND

Lehrstuhl Software–Technologie
44221 Dortmund

Dortmund, den 15. August 1994

An dieser Stelle danke ich allen, die mich bei der Erstellung dieser Diplomarbeit unterstützt haben. Für die Betreuung danke ich Prof. Dr. Ernst–Erich Doberkat. Mein Dank für wertvolle Anregungen und konstruktive Kritik gebührt Dr. Volker Gruhn, Claus Pahl und Michael Zielonka. Bedanken möchte ich mich auch bei meinen LEU–Kollegen, die mich bei der Integration meiner Werkzeuge in LEU unterstützt haben. Last not least danke ich meinem Verlobten Thomas für seine Geduld und moralische Unterstützung.

Dortmund, im August 1994 Monika Schneider

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Das Entity–Relationship Modell ist eine weit verbreitete Methode zur graphischen Beschreibung von Datenbankschemata. Graphische Editoren, die eine Datenmodellierung mit Entity–Relationship–Diagrammen ermöglichen, unterstützen den Anwender bei der Spezifikation von Informationssystemen. Um diese Systeme aber auch benutzen zu können, müssen sie in eine Datenbank übertragen werden. In dieser Diplomarbeit wird ein Werkzeug vorgestellt, das den Übergang vom konzeptionellen Datenbankentwurf zur praktischen Anwendung automatisiert. Der Entwurf in Form eines erweiterten Entity–Relationship Modells wird in eine relationale Datenbank transferiert. Dabei wird auch die Schemaevolution unterstützt. Ein geänderter Entwurf kann wiederholt generiert werden. Vorhandene Daten werden so weit wie möglich übernommen.
Wann immer von Datenbankentwurf gesprochen wird, stellt sich auch die Frage nach Performance bei der Ausführung. Es gibt Verfahren, die das Datenmodell in der Anwendung beobachten [EHH+90]. Der Benutzer kann aus ihren Ergebnissen Rückschlüsse auf mögliche Verbesserungen ziehen. In der industriellen Praxis werden Techniken zur Steigerung der Performance während des Betriebs von Datenbanken angewandt [ORA90c]. In beiden Fällen befindet sich das Datenmodell schon in der Ausführung. Das zweite Werkzeug, das in dieser Arbeit entwickelt wird, setzt in der Entwurfsphase an. Es prüft das Schema und schlägt Veränderungen vor, durch die die Performance bei der Ausführung gesteigert wird. Zusätzlich wird getestet, ob der Entwurf vollständig ist. Dabei werden sowohl die Syntax als auch ein Teil der Semantik betrachtet.
Im schriftlichen Teil dieser Arbeit werden die Grundlagen und die Konzepte für die beiden

Werkzeuge vorgestellt. Der praktische Teil besteht aus ihrer Implementierung im Rahmen der LION–Entwicklungs–Umgebung LEU der Firma LION GmbH in Bochum.

1.2 Gliederung der Arbeit

Im **zweiten Kapitel** wird die LION–Entwicklungs–Umgebung vorgestellt. Zunächst werden die Architektur und die einzelnen Komponenten beschrieben. Dann wird gezeigt, wie die Optimierung und die Generierung in LEU eingeordnet werden. Schließlich wird anhand einer einfachen Applikation gezeigt, wie man mit LEU arbeitet.
Das **dritte Kapitel** besteht aus einer Sammlung von Grundlagen, Definitionen und Notationen, auf die sich die Arbeit in den folgenden Kapiteln abstützt. Das Entity–Relationship Modell, das LEU–ER–Modell und das Relationen–Modell werden vorgestellt. Die Definitionen der verschiedenen Normalformen, die zur Untersuchung relationaler Datenbankschemata entwickelt wurden, werden angegeben. Außerdem wird das Transaktionskonzept der Datenbank ORACLE, die von LEU als Informationssystem genutzt wird, erläutert.
Im **vierten Kapitel** wird eine Einordnung dieser Diplomarbeit in verwandte Systeme vorgenommen.
Die **Kapitel fünf und sechs** beschäftigen sich mit der Generierung und der Optimierung. Die Anforderungen und Vorgaben an diese beiden Werkzeuge werden spezifiziert. Für jedes Werkzeug wird ein Konzept vorgestellt, das die Grundlage für die Implementierung im praktischen Teil der Diplomarbeit bildet. Anschließend wird der zugehörige Entwurf beschrieben. Zuletzt wird anhand eines Beispiels gezeigt, wie die Werkzeuge arbeiten. Für die Entwicklung der Optimierung ist es wichtig zu wissen, wie die Generierung arbeitet. Deshalb wird sie an zweiter Stelle vorgestellt, obwohl sie auch schon vor der Generierung aufgerufen werden kann.
Im **siebten Kapitel** wird die Testphase beschrieben. Sie besteht aus den Funktions– Integrations– und Laufzeittests für Generierung und Optimierung.
Das **achte Kapitel** enthält Schlußbemerkungen. Einige Daten zur Implementierung werden angegeben. Die bisherige Erfahrung mit dem praktischen Einsatz beider Werkzeuge wird dargestellt. Anschließend wird die Arbeit noch einmal zusammengefaßt. Zuletzt wird ein Ausblick auf zukünftige Arbeiten an den Werkzeugen gegeben.
Der **Anhang A** enthält die Testbericht für die Generierung und die Optimierung.

Kapitel 2

Die Lion–Entwicklungsumgebung

Die **L**ION–**E**ntwicklungs–**U**mgebung **LEU** wurde von der LION GmbH zur Entwicklung von bau– und wohnungswirtschaftlicher Software initiiert. Die Grundidee von LEU ist, "den Entwurf statischer Informationssystemaspekte mit dem Entwurf von dynamischen Aspekten zu verbinden und damit das systematische Management von Vorgängen zu unterstützen" [Gru93a]. Die statischen Aspekte eines Informationssystems werden in Form von **Datenmodellen** beschrieben. Auf diesen Datenmodellen basieren **Abläufe**, die die Dynamik des Systems repräsentieren. Datenmodelle und Abläufe sind Teile von **Geschäftsprozessen** [ES92].
Im folgenden Abschnitt 2.1 werden die Architektur und die Komponenten von LEU vorgestellt. In 2.2 werden die Optimierung und die Generierung in LEU eingeordnet. Schließlich zeigt 2.3 ein Beispiel für eine mit LEU entwickelte **Applikation**. Unter einer Applikation versteht man in diesem Zusammenhang eine vollständige mit LEU modellierte Anwendung für ein bestimmtes Aufgabengebiet.
Weitere Informationen zu LEU findet man in [Gru94, DGZ94, GLD+93, Gru93a].

2.1 Die Architektur von LEU

Die Abbildung 2.1 zeigt die Architektur von LEU. Auf oberster Ebene steht die **Ablaufsteuerung**. Sie stellt eine sinnvolle Aufrufreihenfolge der einzelnen LEU–Komponenten sicher. Über die Ablaufsteuerung wird auf die LEU–Komponenten für **Modellierung**, **Ausführung** und **Analyse** zugegriffen. Auf unterster Ebene steht das **graphische Entwurfswerkzeug** Motif und das **Data–Dictionary**, welches zunächst in der relationalen Datenbank ORACLE abgelegt wird. Mit Motif wurde eine einheitliche graphische

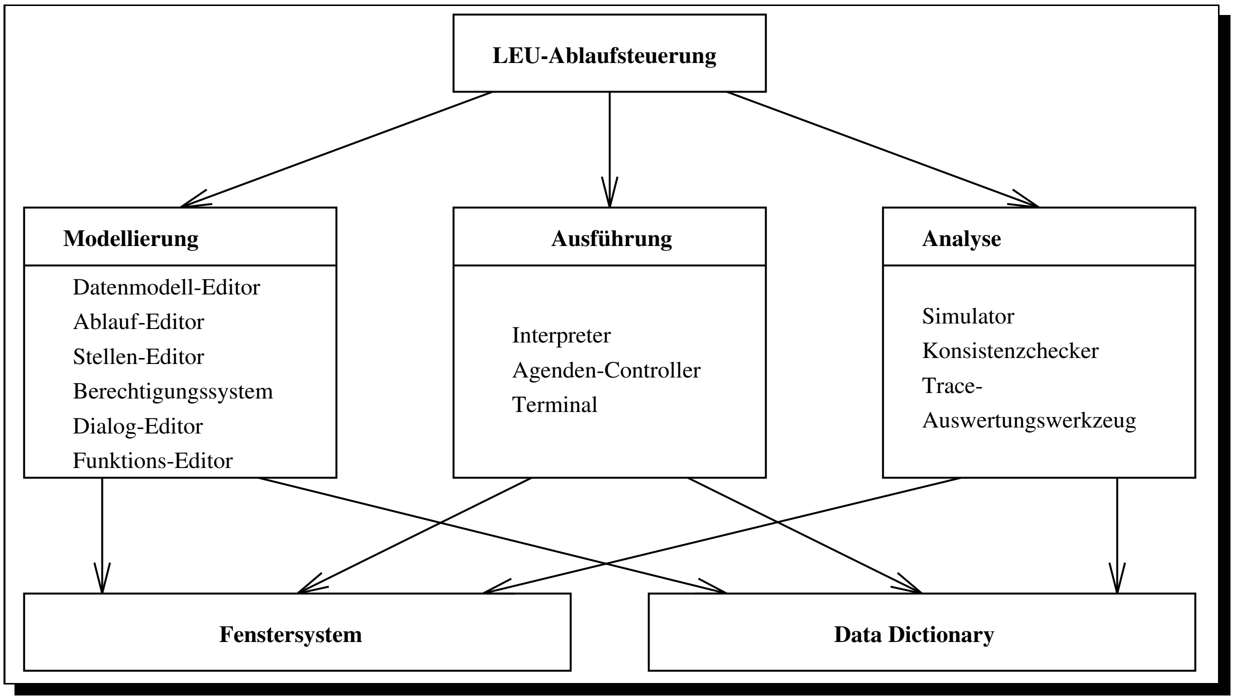


Abbildung 2.1: Architektur der LION–Entwicklungs–Umgebung

Oberfläche für alle Komponenten realisiert. Das Data–Dictionary stellt das Informationssystem dar, in dem alle Komponenten ihre Daten ablegen.

2.1.1 LEU–Komponenten zur Modellierung

Mit den Modellierungskomponenten werden alle Bestandteile eines Geschäftsprozesses beschrieben. Die zentralen Bestandteile sind das **Datenmodell**, das **Stellenmodell** und das **Ablaufmodell**. Zu ihrer Definition steht in LEU je ein Editor zur Verfügung. Darüber hinaus existieren Editoren für **Berechtigungen**, **Dialoge** und **Funktionen**. Im folgenden werden die Editoren der Modellierungskomponenten vorgestellt:

Der Datenmodell–Editor. Hierbei handelt es sich um einen graphischen Editor, mit dessen Hilfe sich Schemata definieren lassen, die an das Entity–Relationship–Modell [Che76] angelehnt sind. Diese Schemata enthalten Objekttypen und Verbindungen, welche die statische Informationsbasis eines Geschäftsprozesses bilden. Zum Datenmodell–Editor gehören noch zwei weitere Editoren: mit dem **Feld–Editor** werden die Attribute der Objekttypen definiert; der **Feldtyp–Editor** dient zur Modellierung verschiedener Wertebereiche für die Felder. Die mit diesem Editor erstellten Schemata werden im folgenden als **LEU–ER–Modelle** bezeichnet.

Der Ablauf–Editor. Mit diesem Editor werden die dynamischen Aspekte eines Geschäftsprozesses modelliert: die Abläufe. Ein Ablauf besteht aus Aktivitäten, ihrer Anordnung und dem Objektaustausch zwischen ihnen. Einige Aktivitäten können automatisch ablaufen, andere benötigen die Mitwirkung von Personen. Die Anordnung der Aktivitäten innerhalb eines Ablaufes kann seriell oder parallel sein. Zur Beschreibung der Abläufe werden FUNSOFT–Netze [EG91] benutzt. Ein FUNSOFT–Netz ist ein abstraktes Petri–Netz [Rei86].

Der Stellen–Editor. Jeder Geschäftsprozeß ist in eine Unternehmensorganisation eingebettet. Sie besteht aus Stellen und deren Beziehungen zueinander. Mit dem Stellen–Editor wird diese Struktur beschrieben.

Das Berechtigungssystem. Über das Berechtigungssystem werden die Personen, die LEU benutzen, erfaßt. Für die verschiedenen Arbeiten innerhalb von LEU brauchen die Benutzer Berechtigungen. Es gibt drei Möglichkeiten, Berechtigungen zu definieren. Zum einen können sie direkt an die Personen vergeben werden. Zum zweiten können sie an die im Stellen–Editor definierten Stellen gehängt werden. Die Personen werden den verschiedenen Stellen zugeordnet und bekommen so die entsprechenden Berechtigungen. Die dritte Möglichkeit bietet der **Rollen–Editor**. Mit ihm werden Rollen mit bestimmten Berechtigungen definiert. Diese Rollen werden wiederum an Stellen oder direkt an Personen vergeben.

Der Dialog–Editor. Für die Objekttypen und Verbindungen, die mit dem Datenmodell–Editor definiert werden, müssen Daten erfaßt werden. Eine Möglichkeit dazu bieten sogenannte Standard–Dialogfenster. Mit einem Standard–Dialogfenster wird genau ein Objekttyp oder eine Verbindung bearbeitet. Diese Fenster werden bei der Datenmodellierung automatisch erzeugt. Der Dialog–Editor bietet darüber hinaus die Möglichkeit, Nichtstandard–Dialogfenster zu definieren. Diese Fenster können so modelliert werden, daß beliebige Teile verschiedener Objekttypen und Verbindungen gemeinsam bearbeitet werden können.

Der Funktions–Editor. An die automatischen Aktivitäten des Ablaufmodells können Funktionen angebunden werden. Diese Funktionen werden ausgeführt, sobald die entsprechenden Aktivitäten angestoßen werden. Mit dem Funktions–Editor können C–Funktionen ediert werden. Zum Testen ruft der Editor einen Debugger auf.

2.1.2 LEU–Komponenten zur Analyse

Die eine Aufgabe der Analysekomponenten besteht darin, die Integration der verschiedenen Bestandteile eines Geschäftsprozesses sicherzustellen. Die andere Aufgabe ist die Simulation der Abläufe. Hierzu wird ein FUNSOFT–Netzsimulator aufgerufen. Dieser Simulator interpretiert die Abläufe und visualisiert sie graphisch innerhalb einer Animationskomponente. Der Modellierer hat die Möglichkeit, aktiv in die Simulation einzugreifen. Die Ergebnisse einer Simulation werden protokolliert. Eine Komponente zur statistischen Auswertung der Simulation setzt auf diesen Protokollen auf. Die Auswertung gibt Aufschluß über Wartezeiten von Objekten auf Weiterverarbeitung, Warteschlangenlängen, die Häufigkeit des Transportes von Objekten, u.a. [Gru93b]. Darüber hinaus kann der kritische Pfad des Ablaufes berechnet werden.

2.1.3 LEU–Komponenten zur Ausführung

Diese Komponenten unterstützen die Ausführung eines Geschäftsprozesses. Bei der Ausführung werden alle Teilmodelle (Datenmodell, Ablaufmodell, Stellenmodell) berücksichtigt. Der **Interpreter** koordiniert die Integration dieser Teilmodelle. Die automatischen Aktivitäten des Ablaufes werden direkt ausgeführt. Die manuellen Aktivitäten müssen von Personen ausgeführt werden. Zu diesem Zweck wird für jeden Benutzer von LEU ein **Terminal** gestartet. Es beinhaltet die Agenda des Benutzers, die aus allen Aktivitäten besteht, die er ausführen darf. Für die Verteilung der Aktivitäten auf die Agenden der Benutzer ist der **Agenden–Controller** zuständig. Wenn eine manuelle Aktivität gestartet wird, schreibt der Controller sie in die Agenden aller Benutzer, die zur Ausführung berechtigt sind.

2.2 Einordnung von Optimierung und Generierung

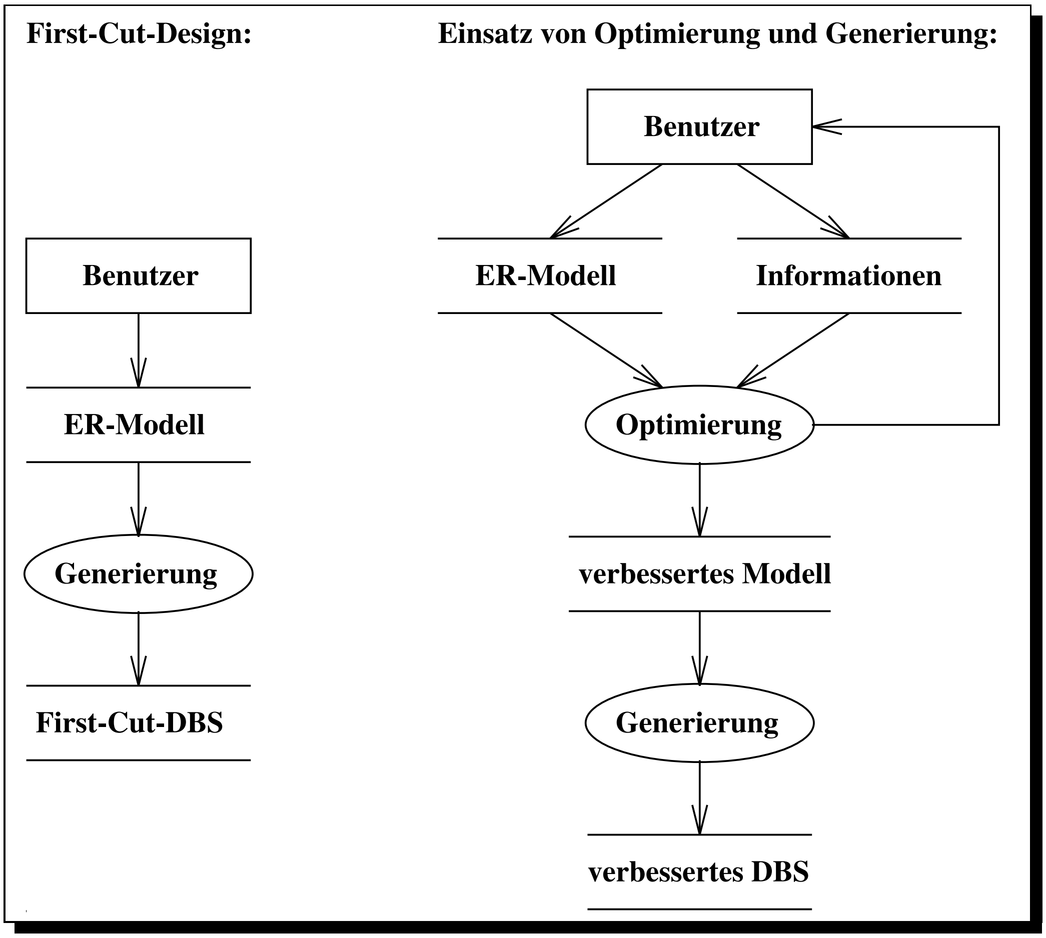


Abbildung 2.2: Vom Benutzer zum Datenbankschema

Die Abbildung 2.2 zeigt auf der linken Seite den einfachen Weg eines LEU–ER–Modells vom Benutzer zur Datenbank. Der Benutzer erstellt mit Hilfe des Datenmodell–Editors und des Feld–Editors ein Modell, das von einer einfachen Generierung in ein relationales First–Cut–Datenbankschema übersetzt wird.
Die rechte Seite der Abbildung 2.2 zeigt, wie die Optimierung und die Generierung in diesen Ablauf integriert werden. Der Benutzer erstellt ein Modell und gibt Informationen über Zugriffspfade, Anzahl der Aufrufe, etc. an. Die Optimierung prüft dies Modell nach

bestimmten Kriterien auf Konsistenz und Effizienz. Die Ergebnisse dieser Untersuchung teilt sie dem Benutzer in Form von Warnungen und Vorschlägen mit. Der Benutzer kann sein Modell ändern und so diesen Kreislauf wieder anstoßen, solange bis er den Zustand akzeptiert. Dann übernimmt die Generierung das fertige Modell. Sie übersetzt es in ein relationales Datenbankschema und installiert es in der Datenbank.

2.3 Eine Applikation

In diesem Abschnitt wird eine einfache LEU–Applikation vorgestellt. Die zahlreichen Abbildungen stehen zur besseren Übersicht zusammen am Ende des Abschnittes. Als Beispiel wird ein Projekt für eine Bibliothek angelegt. Dies Beispiel kann natürlich nur einen Teil der Funktionalität von LEU zeigen.
Als erstes erstellt der Anwendungsentwickler im Datenmodell–Editor das LEU–ER–Modell, das der Applikation zugrunde liegt. Die Abbildung 2.3 zeigt das Modell für eine Bibliothek. Es besteht aus sechs Objekttypen (*Bibliothek, Raum, Regal, Leser, Adresse, Buch*) und fünf Relationen (u.a. *besucht, wohnt in, Standort*). Die Relation *Standort* hat beispielsweise die Kardinalität *1:n*. Das heißt, daß ein Buch nur in einem Regal stehen kann, ein Regal aber mehrere Bücher enthalten darf. Die Relation *Standort* ist außerdem mit *muß:muß* gekennzeichnet, während *besucht* als *kann:kann* ausgezeichnet ist. Das bedeutet, daß über die Verbindung *Standort* ein Buch immer einem Regal zugeordnet sein muß, während ein Leser die Bibliothek besuchen kann. Ausführliche Erläuterungen zum Entity–Relationship Modell folgen im Abschnitt 3.1. Das LEU–spezifische ER–Modell wird in 3.2 erläutert.
Im Feld–Editor werden die Attribute der Objekttypen spezifiziert. Die Abbildung 2.4 zeigt die acht Felder des Objekttypen *Buch*. Als Feldtypen stehen elementare Typen wie Integer, Real, Boolean, Datum oder Strings verschiedener Länge, aber auch komplexere Typen wie Referenz, Aufzählung, Text oder gänzlich selbst zu definierende Typen zur Verfügung.
Wenn das Datenmodell vollständig spezifiziert ist, stößt der Entwickler die Generierung an. Automatisch wird das Modell in der Datenbank installiert. Dann werden die Standarddialog–Fenster erzeugt. Die Abbildung 2.5 zeigt ein solches Fenster für den Objekttyp *Buch*. Über die Dialog–Fenster werden später alle Daten des Modells erfaßt.
Nun ist das Informationssystem, auf das sich die Geschäftsprozesse der Applikation stützen können, abgeschlossen. Als nächstes modelliert der Entwickler die Abläufe. Als Beispiel zeigt die Abbildungen 2.6 den Ablauf–Editor mit dem FUNSOFT–Netz für das *Ausleihen*. Die Kreise stellen Objektspeicher dar, die hier Kanäle heißen. Die Beschriftungen über

den Kanälen zeigen, mit welchen Objekttypen aus dem Datenmodell die einzelnen Kanäle markiert werden können. Die Rechtecke repräsentieren Aktivitäten und werden Instanzen genannt. An die Instanz *ausgeben* ist eine Funktion zur Berechnung des Preises, den der Leser zahlen muß, angebunden.
Diese C–Funktion wurde im Funktions–Editor (Abbildung 2.7) implementiert. Um eine einheitliche Deklaration zu gewährleisten, können die Parameter nur über eine eigene Maske eingegeben werden. Der Rumpf der Funktion kann aber im unteren Fenster des Editors frei ediert werden.
Schließlich muß noch die organisatorische Struktur, in der sich das Projekt befinden soll, modelliert werden. Zunächst definiert der Entwicker die Struktur der Bibliothek im Stellen–Editor. Die Abbildung 2.8 zeigt eine leitende Stelle, die *Verwaltung*. Sie hat die drei untergebene Stellen *Einkauf*, *Buchhaltung* und *Verleih*. Der Buchhaltung untersteht wiederum die Stelle für *Spenden*. Falls eine Stelle einen Leiter hat, wird sein Name mit angegeben. *Herr Wever* steht beispielsweise dem *Einkauf* vor.
Ein Benutzer muß, um mit einer LEU–Applikation arbeiten zu können, eine Rolle besitzen. Die Abbildung 2.9 zeigt die Bearbeitung der Rolle *Projektleiter*. Der Hauptteil des Editors besteht aus einer Reihe von Knöpfen, über die Berechtigungen ein– und ausgeschaltet werden können.
Zuletzt werden in der Benutzerverwaltung die Anwender erfaßt und die Rollen vergeben. Außerdem kann jeder Benutzer selbst später sein Paßwort ändern. Die Abbildung 2.10 zeigt den Benutzer *Schneider*. Frau Schneider hat unter anderem die vorher definierte Rolle des Projektleiters der Bibliothek inne. Zusätzlich besitzt sie noch verschiedene Rollen in anderen Projekten.
Damit ist die Arbeit des Anwendungsentwicklers abgeschlossen.

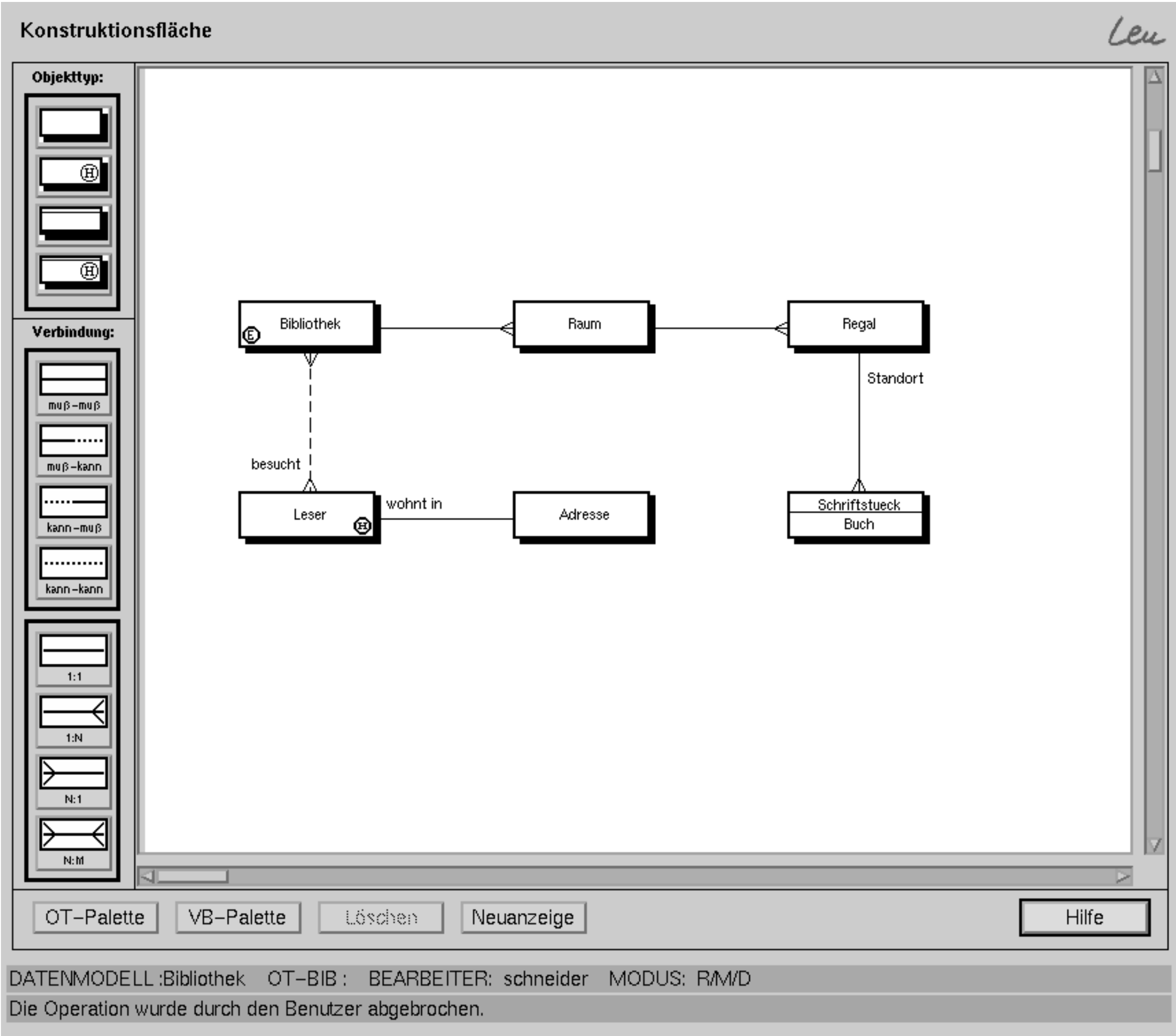


Abbildung 2.3: Datenmodell–Editor

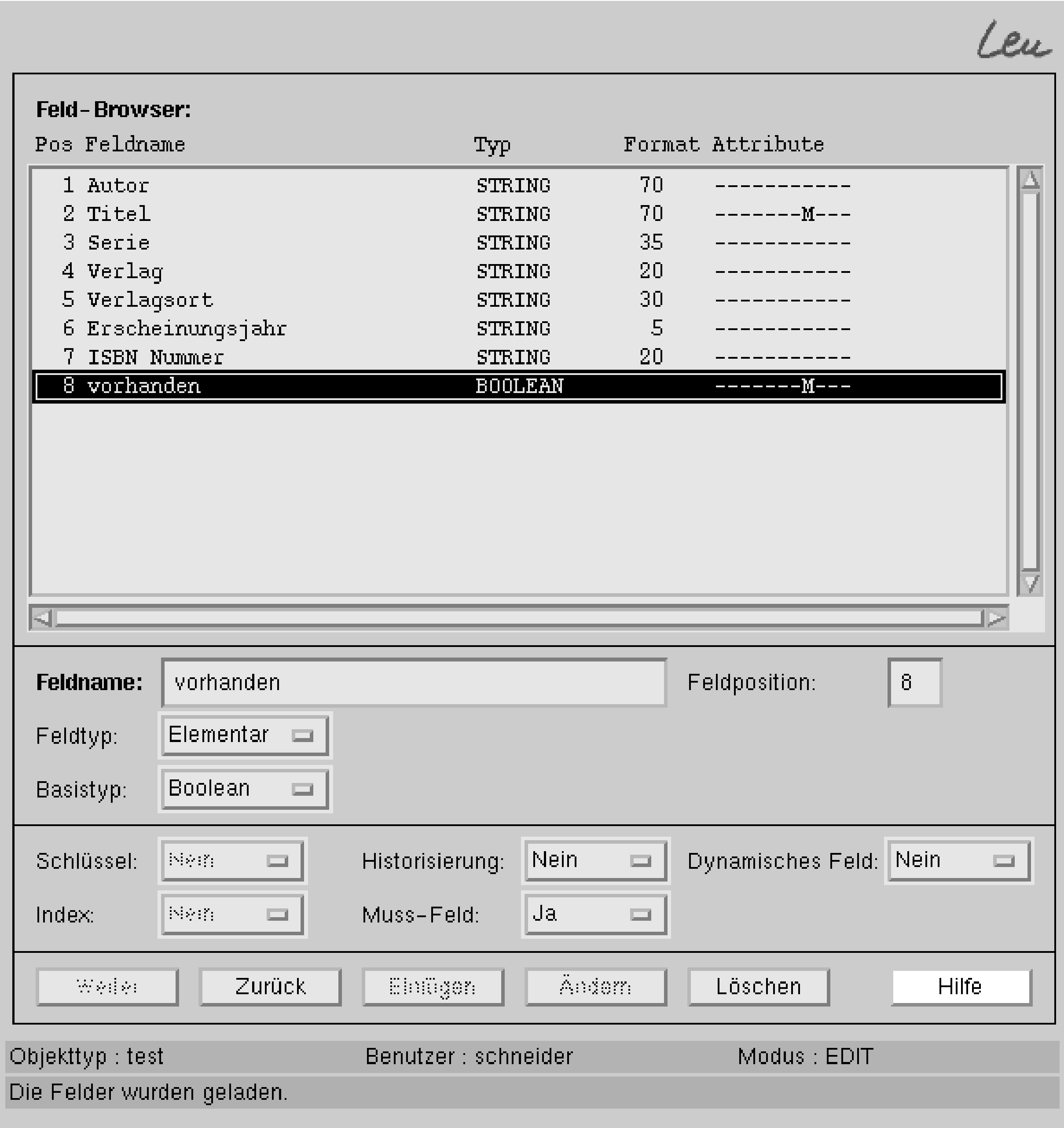


Abbildung 2.4: Feld–Editor

Abbildung 2.5: Standarddialog–Fenster

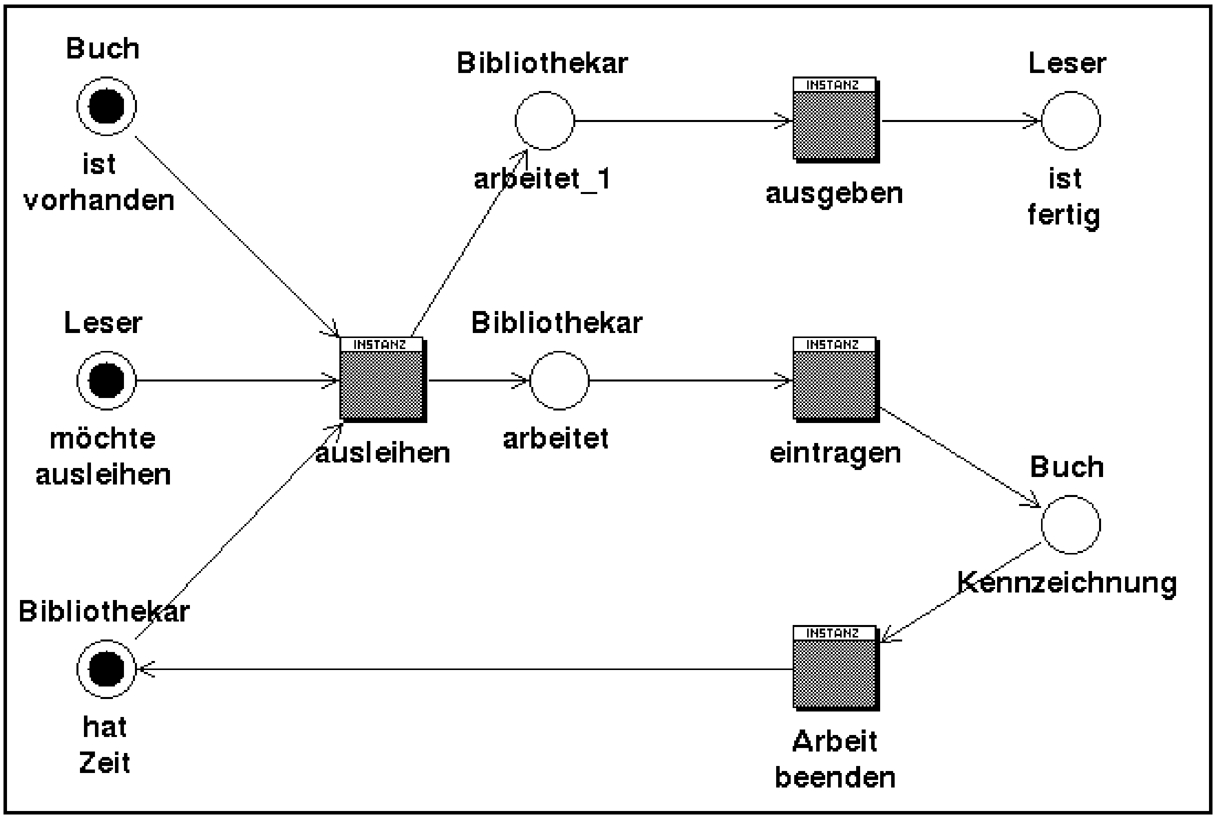


Abbildung 2.6: Ablauf–Editor

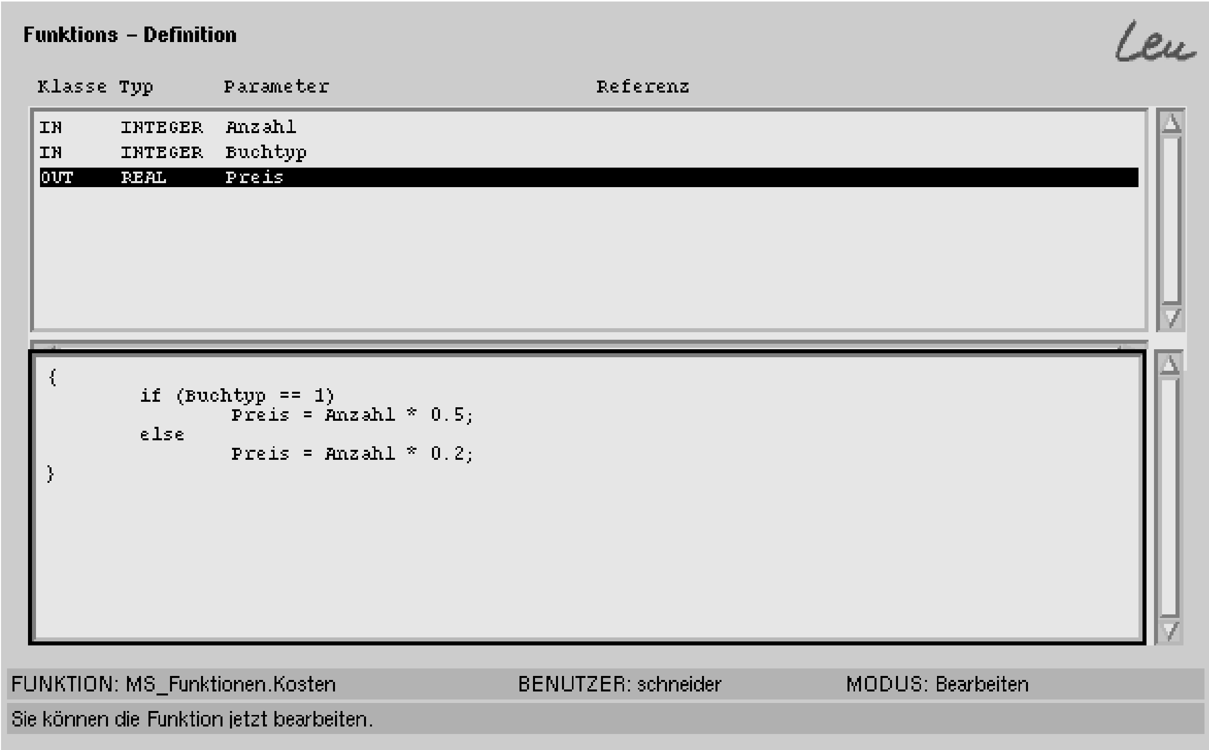


Abbildung 2.7: Funktions–Editor

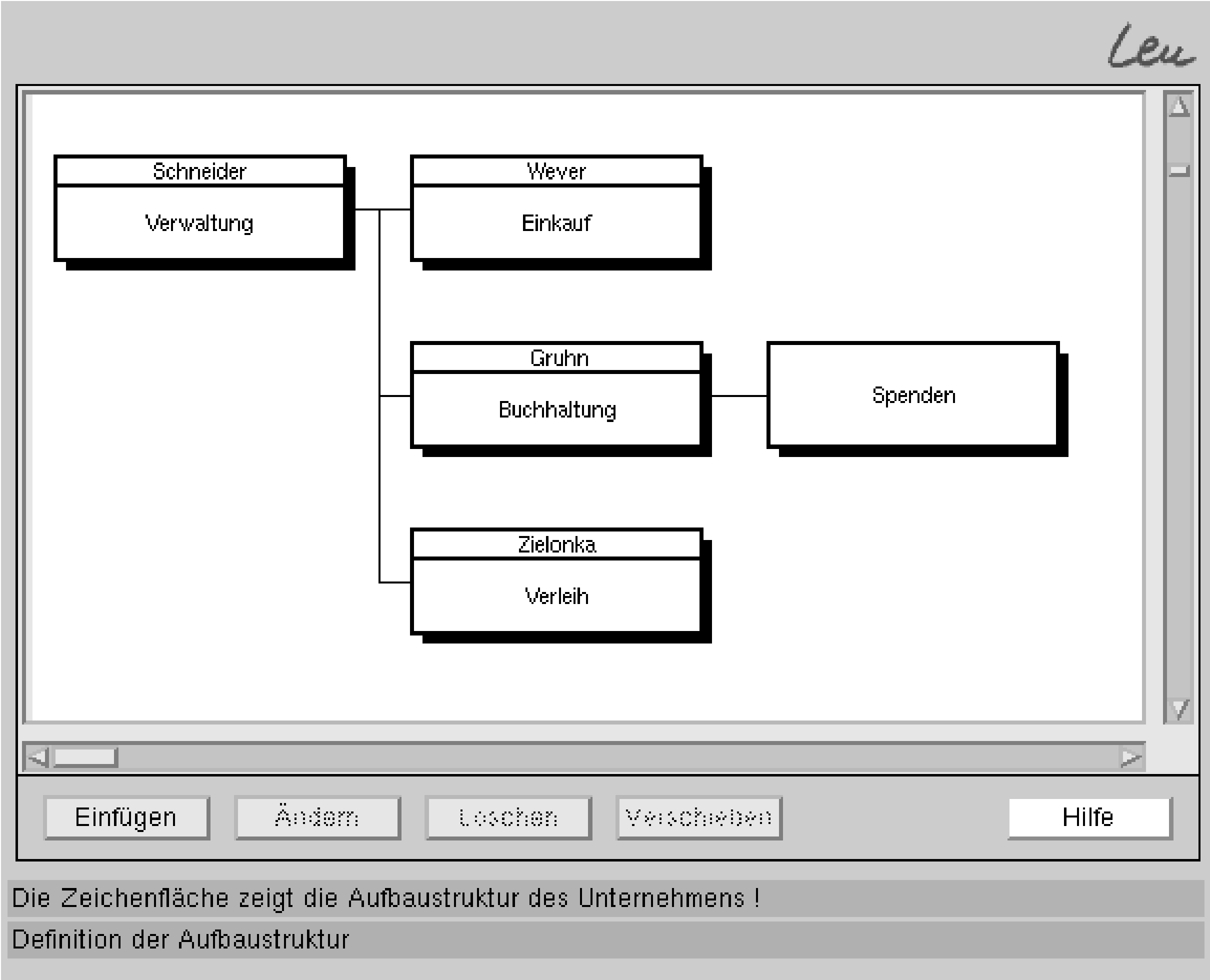


Abbildung 2.8: Stellen–Editor

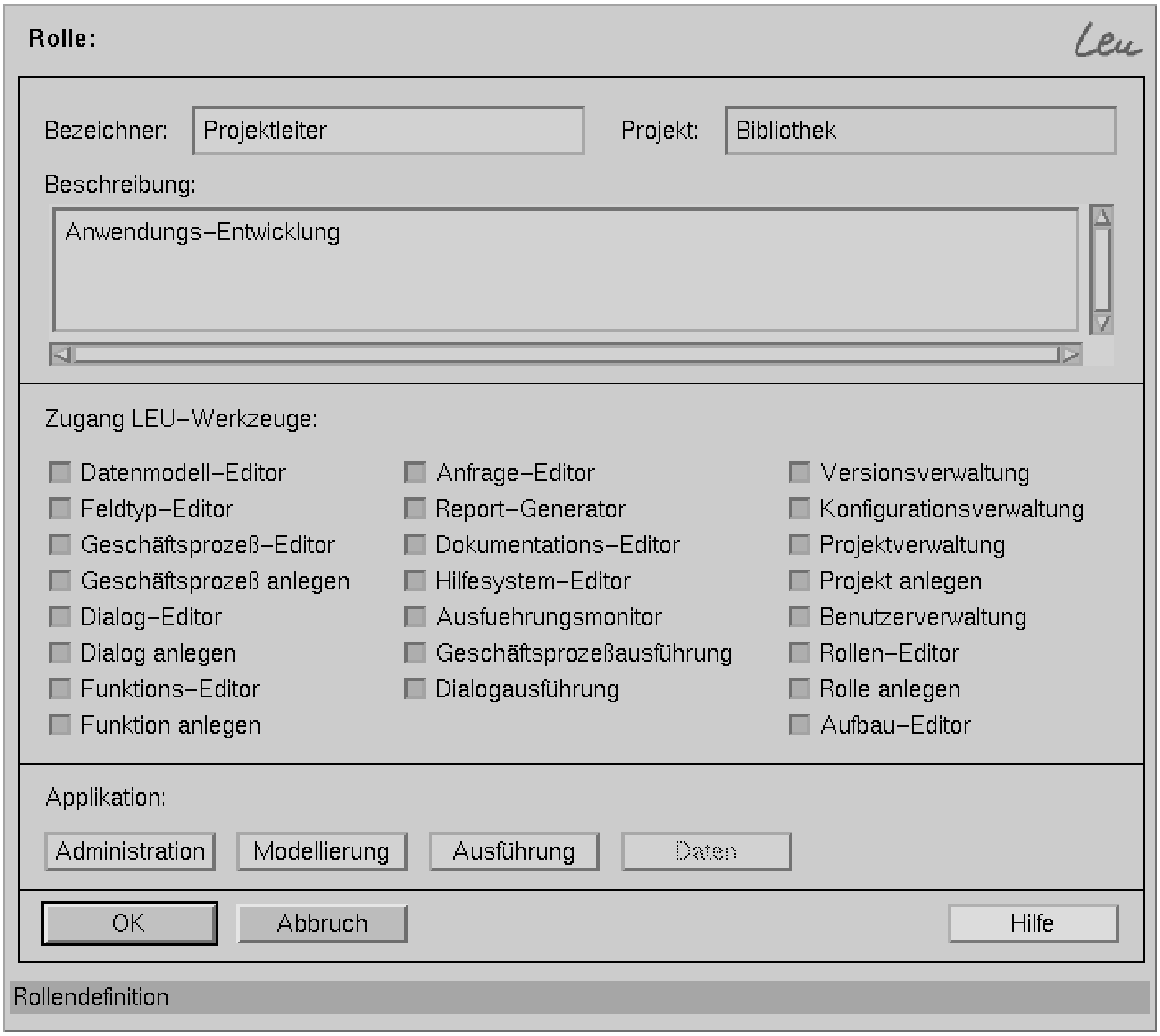


Abbildung 2.9: Rollen–Editor

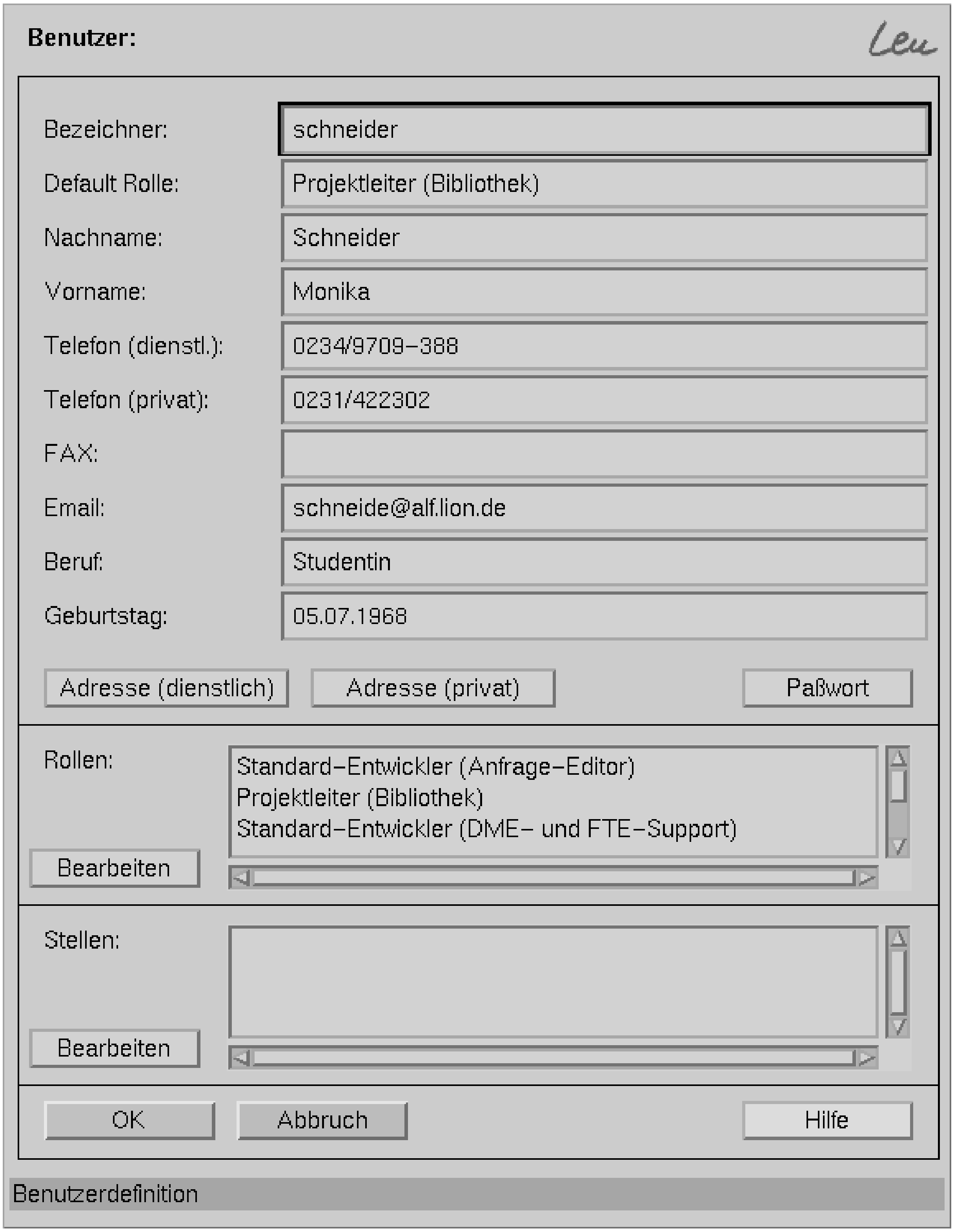


Abbildung 2.10: Benutzerverwaltung

Kapitel 3

Grundlagen

In diesem Kapitel werden einige Grundlagen erläutert, auf denen der Entwurf von Optimierung und Generierung in den folgenden Kapiteln basiert. Zunächst werden das Entity–Relationship Modell (3.1) und das daran angelehnte LEU–ER–Modell (3.2) vorgestellt. Beide Modelle werden zum konzeptionellen Datenbankentwurf verwandt. Mit ihnen werden Ausschnitte aus der "realen Welt" beschrieben. Um ein solches Modell in Betrieb zu nehmen, also beispielsweise Daten zu erfassen, muß es in einem Datenbank–Management–System (DBMS) implementiert werden. Die Grundlage vieler moderner DBMS ist das Relationen–Modell (3.3). Deshalb wird ein Entity–Relationship Modell (und auch ein LEU–ER–Modell) in zwei Schritten auf ein DBMS abgebildet. Zuerst wird das ER–Modell in das Relationen–Modell übersetzt, welches anschließend in dem DBMS installiert wird.
Ein methodisches Hilfsmittel für den Datenbankentwurf ist die Normalisierung. Sie dient zur Verbesserung des erstellten Modells und bietet damit einen möglichen Ansatz für die Optimierung, die dasselbe Ziel verfolgt. Bei der Normalisierung wird ein Modell so umorganisiert, daß es am Ende den Normalformen (3.4) entspricht.
Das DBMS, auf dem LEU aufsetzt, ist die relationale Datenbank ORACLE. Alle Zugriffe auf die Datenbank sind über ihr Transaktionskonzept geregelt. Dies Konzept muß besonders von der Generierung berücksichtigt werden, da sie das LEU–ER–Modell automatisch in der Datenbank implementiert. Abschnitt 3.5 beschreibt das Transaktionskonzept von ORACLE.

3.1 Das Entity–Relationship Modell

Das **Entity–Relationship Modell** wurde 1976 von Peter Pin–Shan Chen vorgestellt [Che76]. Die vier Konzepte des Entity–Relationship Modells sind Entitäten, Relationen,

Attribute und Werte.
Eine **Entität** ist ein eindeutig identifizierbares **Objekt**. Es kann eine Person, ein Gegenstand, ein Ereignis aber auch ein abstrakter Begriff sein. Mehrere Entitäten mit gleichen Merkmalen werden zu einer **Entitätenmenge** bzw. einem **Objekttypen** zusammengefaßt. Eine **Relation** setzt Entitäten in Beziehung zueinander. Zum Beispiel verbindet die Relation *Standort* die Objekttypen *Buch* und *Regal*. Relationen werden zu **Relationenmengen** zusammengefaßt. Sowohl Objekttypen als auch Relationenmengen können **Attribute** besitzten. Ein Attribut ist ein Merkmal oder eine Eigenschaft, wie *Titel*, *Nummer* oder *Inhalt*. Der Typ eines Attributes wird durch eine **Wertemenge** bestimmt. Die Wertemenge *Integer* enthält beispielsweise als **Werte** alle natürlichen Zahlen, und kann als Typ für das Attribut *Nummer* fungieren.
Für jeden Objekttypen wird ein **Schlüssel** bestimmt, der aus einem oder mehreren Attributen besteht. **Schlüssel–Attribute** sind alle diejenigen Attribute, über deren Werte ein Objekt eindeutig identifiziert werden kann. Eine sinnvolle Untermenge aller Schlüssel–Attribute wird als **Primärschlüssel** bezeichnet und dient zur Identifizierung der Objekte innerhalb des Modells. Der Primärschlüssel kann auch künstlich erzeugt werden, indem eigens für ihn ein Attribut hinzugefügt wird.

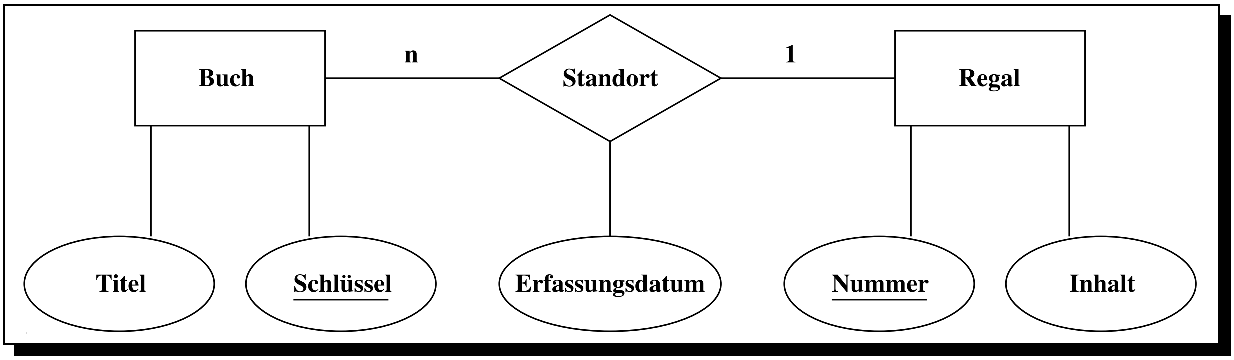


Abbildung 3.1: Das Schema eines Entity–Relationship Modells

Ein Entity–Relationship Modell wird durch ein Schema dargestellt. Es besteht aus Rechtecken für die Objekttypen, Rauten für die Relationen und Ovalen für die Attribute. Die Attribute des Primärschlüssels werden unterstrichen. Die Abbildung 3.1 zeigt ein solches Schema. Die Daten des Modells werden in Tabellen abgelegt. Dazu wird für jeden Objekttypen eine Tabelle erstellt. Die Attribute des Objekttypen bilden die Spalten. Die

Zeilen, die jeweils ein Objekt definieren, werden auch **Tupel** genannt. Für jede Relation wird ebenfalls eine Tabelle erzeugt, die die eigenen Attribute und die Primärschlüssel der verbundenen Objekttypen enthält. In Abbildung 3.2 stehen die drei Tabellen des Schemas mit einigen Beispieldaten.

Buch	
Titel	Schlüssel
Der Rosenmord	1863
Es	8733
Metamorphose	5404
⋮	⋮

Regal	
Nummer	Inhalt
3	Krimi
15	Science Fiction
24	Horror
⋮	⋮

Standort		
Schlüssel	Nummer	Erfassungsdatum
1863	3	1987
8733	24	1990
5404	15	1992
⋮	⋮	⋮

Abbildung 3.2: Die Tabellen eines Entity–Relationship Modells

Das ER–Modell wurde mehrfach erweitert [Cod79]:

Surrogate. Eine Erweiterung ist die Einführung der Surrogate [Cad76, HOT76]. Jeder Objekttyp bekommt ein zusätzliches Attribut für das Surrogat. Dies Attribut fungiert als Primärschlüssel des Objekttypen. Die Werte für die Surrogate werden so vergeben, daß sie nicht nur innerhalb eines Objekttypen, sondern innerhalb des ganzen Schemas eindeutig sind.

Funktionale Beziehungen. Jeder Relationenmenge werden Kardinalitäten zugeordnet. Durch die Bezeichnung *1* oder *n* wird die Anzahl der Objekte eines Objekttypen festgelegt, die an einer Relation beteiligt sein dürfen. So legt die Kardinalität *n:1* für die Relationenmenge *Standort* aus Abbildung 3.1 fest, daß jedes Buch nur in einem Regal steht, und daß ein Regal mehrere Bücher enthält.

Generalisierung. Mit Hilfe der Generalisierung (bzw. Spezialisierung) werden Unter– bzw. Oberbegriffe gebildet [SS77]. Dabei vererbt der Oberbegriff alle seine Attribute an den Unterbegriff. Der Unterbegriff besitzt seinerseits zusätzliche Attribute, durch die er sich von dem Oberbegriff unterscheidet.

Detaillierte Informationen zum Entity–Relationship Modell können in [Che77, Des90, EN89] nachgelesen werden.

3.2 Das LEU–ER–Modell

Das LEU–ER–Modell lehnt sich zwar an das erweiterte Entity–Relationship Modell an, es gibt aber verschiedene Änderungen. Die Abbildung 3.3 zeigt eine erweiterte Version des ER–Modells aus Abbildung 3.1 so, wie sie in LEU dargestellt wird.

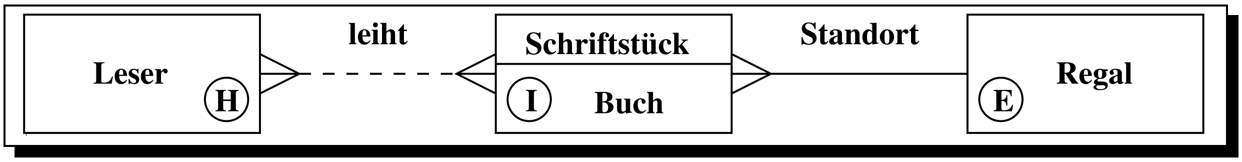


Abbildung 3.3: Das Schema eines LEU–ER–Modells

Verbindungen. Die Relationenmengen werden in LEU Verbindungen genannt. Es gibt ausschließlich binäre Verbindungen. Sie werden deshalb auch nicht durch eine Raute repräsentiert, sondern einfach durch eine beschriftete Gerade. Die Kardinalität der Verbindungen wird ebenfalls graphisch dargestellt. Eine Gerade, die direkt an einem Objekttypen endet, hat die Stelligkeit *1*. Ein "Krähenfuß" am Ende einer Geraden symbolisiert die Stelligkeit *n*. Die **Restriktion** einer Verbindung besagt, ob eine Verbindung zu einem Objekt des angebundenen Objekttypen hergestellt werden *kann* oder *muß*. Eine *kann*–Beziehung wird durch eine gestrichelte Linie dargestellt. Eine durchgezogene Linie steht für eine *muß*–Beziehung. Die Verbindung *Standort* aus Abbildung 3.3 fordert demnach, daß ein Buch in einem Regal stehen *muß*. Dagegen besagt die Verbindung *leiht*, daß ein Leser ein Buch ausleihen *kann*. Verbindungen können keine eigenen Felder besitzen. Sie werden aber wie im ER–Modell durch die Primärschlüssel der beiden verbundenen Objekttypen dargestellt.

Objekttypen. Objekttypen werden grundsätzlich durch ein Rechteck repräsentiert. Darüber hinaus gibt es verschiedene Erweiterungen dieser Darstellung. Es gibt zwei Arten von Objekttypen: **lokale** und **globale**. Ein lokaler Objekttyp ist nur innerhalb des Modells bekannt, in dem er definiert wird. Soll er auch von anderen Modellen benutzt werden können, muß er exportiert werden. Er ist dann nicht mehr lokal, sondern steht global zur Verfügung. Dies wird durch ein eingekreistes E in der linken unteren Ecke des entsprechenden Rechteckes symbolisiert. Das exportierende Modell wird auch **Heimat–Datenmodell** des Objekttypen genannt. Ein globaler exportierter Objekttyp kann in jedes andere Modell importiert werden, was durch ein eingekreistes I angezeigt wird. Durch die globalen Objekttypen ist also eine **Schemaintegration** zumindest auf Objekttyp–Ebene möglich.

Felder. Die Attribute der Objekttypen werden in LEU als Felder bezeichnet. Sie werden der Übersicht halber nicht in der Graphik des LEU–ER–Modells dargestellt.

Feldtypen. Für die Werte der Felder gibt es eine Anzahl vordefinierter Feldtypen:

- Aufzählung: die Elemente, aus denen der Aufzählungstyp besteht, sind maximal 255–stellige darstellbare Zeichenketten. Es gibt lokale Aufzählungstypen, die nur die Werte eines Feldes bestimmen, und globale, die unter einem Bezeichner abgelegt werden und für jedes beliebige Feld benutzt werden können.

- Elementar: dies ist kein eigener Feldtyp, sondern gliedert sich in acht Basistypen:

  - Boolean := {0,1},
  - Integer := die Menge der maximal fünfzehnstelligen ganzen Zahlen,
  - Real := die Menge der rationalen Zahlen mit maximal 16 Stellen, davon maximal 15 Nachkommastellen,
  - String_n := die Menge aller darstellbaren Zeichenketten der Länge 1, 3, 5, 10, 20, 30, 35 oder 70,
  - Text := die Menge aller darstellbaren Zeichenketten,
  - Time := die Menge aller Zeiten im Format *hhmmss*,
  - Date := die Menge aller Daten im Format *ddmmyyyy*.

Jeder Basistyp kann direkt als Feldtyp benutzt werden. Zusätzlich können globale Basistypen definiert werden, indem beliebigen Bezeichnern Basistypen zugeordnet werden. Jeder Basistyp kann auf diese Weise unter mehreren Bezeichnern global zur Verfügung stehen.

- Referenz: hier wird eine Referenz auf einen Objekttypen oder auf ein Feld eines Objekttypen abgelegt. An dieser Stelle wird also die Definition **verschachtelter Strukturen** innerhalb von Objekttypen ermöglicht.
- Liste: lokale, globale und elementare Feldtypen können eine Dynamik enthalten. Das heißt, es wird eine Liste mit Elementen des entsprechenden Typs angelegt. Diese Liste ist der eigentliche Feldtyp, und ihre Elemente bilden zusammen den Wert eines Feld–Eintrages. Mit Hilfe der Liste ist es möglich, tupelwertige Felder zu definieren.

Die Erweiterungen gegenüber dem klassischen ER–Modell bestehen in den **tupelwertigen** und **objektwertigen Feldern** (Liste und Referenz).

Vererbung. In LEU wird die Generalisierung als Vererbung bezeichnet. Es handelt sich um eine **Einfachvererbung**. Ein Objekttyp kann als Sohn eines anderen Objekttypen definiert werden. Der Sohn erbt alle Felder seines Vaters. Darüber hinaus kann er eigene Felder besitzen, die ihn vom Vater unterscheiden. Ein Objekttyp darf mehrere Söhne aber nur einen Vater haben. Die Objekte des Vaters können unabhängig von seinen Söhnen existieren. Wenn ein Objekt zu einem Sohn angelegt wird, wird eines der Objekte des Vaters übernommen. Auf diese Weise kann ein Vater seine Objekte an mehrere Söhne gleichzeitig vererben. In dem Beispiel in Abbildung 3.3 ist *Buch* ein Sohn von *Schriftstück*.

Historisierung. Eine andere Erweiterung des LEU–ER–Modells ist die Historisierung. Für jedes Feld eines Objekttypen kann der Benutzer bestimmen, ob es historisiert werden soll. Das bedeutet, daß bei einer Änderung der Feldwerte die alten Werte nicht einfach überschrieben, sondern in einem eigens dafür angelegten Objekttypen gesichert werden. Auf diese Weise kann die Veränderung der Daten über einen beliebigen Zeitraum nachvollzogen werden.

Schlüssel. Es gibt zwei Arten von Schlüsseln für jeden Objekttypen: einen benutzerdefinierten und einen von LEU vergebenen. Für jeden Objekttypen kann der Benutzer einen Schlüssel definieren. Das bedeutet allerdings nur, daß die Kombination der Werte aller

Schlüsselfelder eindeutig sein muß. Zusätzlich wird jedem Objektypen von LEU automatisch ein zusätzliches Feld für ein Surrogat zugeordnet. Das Surrogat ist der Primärschlüssel des Objekttypen und dient somit zur systemweit eindeutigen Identifizierung seiner Objekte.

Schemaevolution. Alle Konzepte des LEU–ER–Modells können dynamisch geändert werden. Es gibt lediglich einige Einschränkungen hinsichtlich der Konsistenz des Schemas:

- Ein globaler exportierter Objekttyp darf nicht gelöscht werden, solange er noch in mindestens einem anderen Modell importiert wird.
- Ein Vater–Objekttyp darf nicht gelöscht werden.
- Ein referenzierter Objekttyp darf nicht gelöscht werden.
- Ein Objekttyp, der ein referenziertes Feld enthält, darf nicht gelöscht werden.

3.3 Das Relationen–Modell

Die Grundlage vieler Datenbank Management Systeme bildet das Relationen–Modell. Es wurde 1970 von Codd vorgestellt [Cod70]. Die vier Konzepte des Relationen–Modells sind Relationen, Tupel, Attribute und Domänen.
Eine **Relation** besteht aus einer Menge von logisch zusammenhängenden Informationen. Sie ist aufgebaut wie eine Tabelle und wird auch als solche dargestellt. Die Abbildung 3.4 zeigt die Relation *Buch*.

Buch		
Titel	Schlüssel	Kategorie
Der Rosenmord	1863	Krimi
Es	8733	Horror
Metamorphose	5404	NULL
⋮	⋮	⋮

Abbildung 3.4: Eine Relation eines Relationen–Modells

Die Spalten der Tabelle stellen die **Attribute** der Relation dar, die wie im Entity–Relationship Modell Merkmale oder Eigenschaften beschreiben. Die Relation *Buch* hat die drei Attribute *Titel*, *Schlüssel* und *Kategorie*. Der Wertebereich eines Attributes wird als **Domäne** bezeichnet. Eine Domäne ist modellweit definiert und somit von allen Relationen zugreifbar. Die Zeilen der Relation heißen **Tupel**. Sie enthalten die Daten der Relation. Ein Tupel besteht also aus je einem Eintrag zu jedem Attribut, wie zum Beispiel *(Der Rosenmord, 1863, Krimi)*. Wenn ein Wert noch nicht bekannt ist, kann stattdessen ein **Null–Wert** eingetragen werden. Zum Beispiel ist in der Abbildung 3.4 die *Kategorie* des Buches *Metamorphose* nicht bekannt.

Jede Relation besitzt einen oder auch mehrere eindeutige **Schlüssel**. Ein Schlüssel setzt sich aus den Attributen der Relation zusammen, die gemeinsam jedes Tupel eindeutig identifizieren. Einer der Schlüssel wird als **Primärschlüssel** ausgewählt. Ein Primärschlüssel kann in einer anderen Relation als **Fremdschlüssel** eingeführt werden. Die Werte eines Fremdschlüssels müssen aus der gleichen Domäne gewählt werden.

Damit eine Tabelle als Relation bezeichnet werden kann, müssen vier Bedingungen erfüllt sein:

1. Es gibt keine doppelten Tupel.
2. Die Reihenfolge der Tupel ist nicht relevant.
3. Die Reihenfolge der Attribute ist nicht relevant.
4. Die Werte aller Attribute sind atomar.

Außerdem gibt es zwei Integritätsregeln, die immer beachtet werden müssen, wenn Daten der Relation eingetragen, geändert oder gelöscht werden:

1. Kein Attribut eines Primärschlüssels darf einen NULL–Wert enthalten. (**Entitäten–Integrität**)
2. Die Werte eines Fremdschlüssels müssen immer als Werte des zugehörigen Primärschlüssels existieren. (**Referenzielle Integrität**)

Detailliertere Beschreibungen zum Relationen–Modell sind in den meisten Büchern über Datenbank Management Systeme zu finden, zum Beispiel in [Dat90, Des90, EN89, Ullm88].

3.4 Normalformen

Normalformen sind gute Kriterien, um den Umfang von Redundanzen und Anomalien innerhalb eines Datenmodells zu beschreiben. Bevor jedoch die Normalformen definiert werden können, müssen einige Begriffe geklärt werden:

Definition : Gegeben sei eine Relation R. Ein Attribut Y ist **funktional abhängig** von einem Attribut X, wenn zu jedem Wert aus X genau ein Wert aus Y existiert. Man sagt auch: X bestimmt Y.

Definition : Ein **Schlüssel(kandidat)** einer Relation ist eine minimale Attributmenge, von der alle anderen Attribute funktional abhängen.

Definition : Ein **Nichtschlüsselattribut** einer Relation ist ein Attribut, das zu keinem Schlüsselkandidaten gehört.

Ursprünglich definierte Codd drei Normalformen (1NF, 2NF, 3NF), die aufeinander aufbauen [Cod72]. Später definierte er eine strengere dritte Normalform, die Boyce/Codd Normalform [Cod74]. Schließlich definierte Fagin eine vierte [Fag77] und später eine fünfte Normalform [Fag79]. In der Regel werden aber nur die ersten drei Normalformen benutzt:

Definition : Eine Relation ist in **erster Normalform (1NF)**, wenn alle Attribute nur atomare Wertebereiche haben.

Definition : Eine Relation ist in **zweiter Normalform (2NF)**, wenn sie in erster Normalform ist, und wenn jedes Nichtschlüsselattribut von jedem Schlüssel voll funktional abhängt.

Definition : Eine Relation ist in **dritter Normalform (3NF)**, wenn sie in zweiter Normalform ist, und wenn kein Nichtschlüsselattribut von einem Schlüssel transitiv abhängt.

Ausführliche Erläuterungen zu den Normalformen sind in [Dat86] zu finden.

3.5 Das Transaktionskonzept von ORACLE

Alle Daten von LEU werden in der relationalen Datenbank ORACLE abgelegt. Der Zugriff auf diese Datenbank ist durch ihr Transaktionskonzept geregelt [ORA90a, Stu93].

Eine **Transaktion** ist eine Folge von Datenbankzugriffen, die als logische Einheit betrachtet werden. Sie überführt eine Datenbank von einem konsistenten Zustand in einen anderen konsistenten Zustand. Wenn eine Transaktion gestartet wird, sind alle Manipulationen, die der Benutzer innerhalb dieser Transaktion durchführt, zunächst nur für ihn sichtbar. Erst

bei der Beendigung einer Transaktion wird festgelegt, was mit den durchgeführten Änderungen passiert. Hierfür sind zwei Operationen aus der Daten–Manipulations–Sprache von besonderer Bedeutung: **Commit** und **Rollback**. Durch ein *Commit* wird eine Transaktion beendet. Alle innerhalb der Transaktion vorgenommenen Manipulationen werden entgültig in die Datenbank übernommen und dadurch auch jedem anderen Benutzer sichtbar gemacht. Ein *Rollback* beendet ebenfalls die laufende Transaktion. Allerdings werden dabei alle innerhalb der Transaktion ausgeführten Operationen wieder verworfen. Nach jedem *Commit* und jedem *Rollback* startet mit dem nächsten Befehl automatisch auch die nächste Transaktion.
Eine Transaktion beginnt

- nach dem Aufbau der Verbindung zur Datenbank,
- nach der Anweisung *Commit* und
- nach der Anweisung *Rollback*.

Eine Transaktion endet

- nach der Anweisung *Commit*,
- nach der Anweisung *Rollback*,
- mit der Abmeldung von der Datenbank (wie *Commit*) und
- mit einem fehlerhaften Abbruch durch das Betriebssystems oder den Benutzerprozeß (wie *Rollback*).

Zu beachten ist weiterhin, daß alle Befehle aus der Daten–Definitions–Sprache (wie *Create, Drop, Rename, Alter* [ORA90b]) innerhalb einer eigenen Transaktion ausgeführt werden, die jeweils nur aus diesem einen Befehl besteht. ORACLE selbst beendet vor einem DDL–Befehl die laufende Transaktion durch ein *Commit* und führt dann erst den Befehl aus. Anschließend wird diese Transaktion wieder beendet.

Kapitel 4

Einordnung in verwandte Systeme

In diesem Kapitel werden zum einen LEU und zum anderen die in dieser Diplomarbeit entwickelten Werkzeuge (Generierung und Optimierung) gegen verwandte Systeme abgegrenzt.
Der Abschnitt 4.1 befaßt sich mit Systemen zur Softwareprozeß–Modellierung. LEU wird in diesen Kontext eingeordnet.
In 4.2 und 4.3 werden zwei Projekte, CADDY und EASYMAP, vorgestellt, die sich mit der Unterstützung des logischen Datenbankentwurfes befassen. Dabei werden zwei Bereiche besonders betrachtet: die Umsetzung eines konzeptionellen Entwurfes in ein Datenbankschema und die Modifikation dieses Schemas im Hinblick auf Performancegewinn bei der Ausführung. Die Generierung und die Optimierung verfolgen dieselben Ziele. Daher lassen sich einige Vergleiche zwischen ihnen und CADDY bzw. EASYMAP ziehen.

4.1 Systeme zur Softwareprozeß–Modellierung

Seit einiger Zeit etabliert sich im Bereich der Softwaretechnologie die Softwareprozeß–Modellierung. Im akademischen Umfeld bildete sich das Forschungsgebiet des Softwareprozeß–Management [Per91, Dow91] und im kommerziellen Umfeld das Geschäftsprozeß–Management [ES92]. In beiden Fällen wird das Kernproblem durch die folgende Frage beschrieben: Wie kann ein Computersystem für die Entwicklung von softwarebasierten Systemen eingesetzt werden?
Es wurden inzwischen einige Ansätze zur Softwareprozeß–Modellierung vorgestellt. Sie setzen auf verschiedenen Paradigmen auf, wie Regeln (Marvel [KFP88], Merlin [PSW92], Oikos [AJ91]), imperativer Programmierung (APPL/A [HSO90], IPSE 2.5 [War89]) oder

Petri–Netzen (Melmac [Gru91], Slang [BFG91]). Viele dieser Systeme (abgesehen von den Petri–Netz–basierten) benutzen eine textuelle Repräsentation und arbeiten auf der Ebene von Programmiersprachen. Der Nachteil dabei ist, daß der Modellierer durch die technischen Probleme des jeweiligen Modellierungs–Formalismus gehemmt wird. Darüber hinaus wird keine getrennte Modellierung der statischen und dynamischen Aspekte eines Softwareprozesses angeboten.
Die Integration statischer und dynamischer Anteile eines Prozesses wird in verschiedenen Systemen realisiert. Beispielsweise werden in DAIDA [JMS+92] verschiedene Sichten im Informationssystementwurf verwandt. In TROLL [HJS93] wird ein Objekt als beobachtbarer Prozeß aufgefaßt und sein Verhalten über der Zeit beschrieben.
Auch in LEU werden statische und dynamische Akpekte eines Softwareprozesses realisiert. Die statischen Anteile werden mit LEU–ER–Modellen beschrieben, die dynamischen Anteile mit FUNSOFT–Netzen. Zusätzlich gibt es Komponenten zur Integration beider Aspekte. Um die LEU–ER–Modelle nicht nur erstellen, sondern auch benutzen zu können, wird in dieser Diplomarbeit die Generierung entwickelt. Damit darüber hinaus der Zugriff auf die generierten Modelle möglichst performant verläuft, wird die Optimierung eingesetzt.

4.2 Die Datenbank–Entwurfsumgebung CADDY

Die Abteilung Datenbanken des Instituts für Programmiersprachen und Informationssysteme der Technischen Universität Braunschweig entwickelte in den Jahren 1987 bis 1992 die Datenbank–Entwurfsumgebung CADDY [EHH+90, EGH+90, HE91, EL91, SNH+87]. Das Ziel war die Unterstützung des Entwurfs von Nichtstandard–Datenbanken für Anwendungen aus Bereichen wie CAD, VLSI, Software Engineering, Geowissenschaft, etc. Die Werkzeuge, ein interaktiver, graphischer Editor, ein Prototyp–Generator und ein Integritätsmonitor, sollten eine komfortable Modellierung erweiterter ER–Modelle, deren Test auf innere Konsistenz und die Überprüfung von Handhabbarkeit und Funktionalität gewährleisten.
Der Editor dient dem graphisch–interaktiven Entwurf von erweiterten ER–Modellen (kurz: EER–Modelle). Während des Designs verhindern automatische Konsistenzprüfungen die Operationen, die in jedem Fall zu einem falschen Schema führen würden. Dazu gehören zum Beispiel Verbindungen ohne Quelle oder Ziel, gleiche Bezeichner, undefinierte Wertebereiche oder das Löschen von Vater–Objekten. Zusätzlich gibt es weitere "globale" Konsistenzprüfungen. Dort werden unter anderem die funktionalen Beziehungen, die Ge-

neralisierung und die Syntax der Integritätsbedingungen kontrolliert. Die Funktionalität des Entwurfs wird dann mit Hilfe des Prototyp–Generators getestet.
Der Generator übersetzt das EER–Schema in ein relationales Schema. Dabei wird jeder Objekttyp (hier: Objektklasse) in eine Relation übersetzt. Diese Relation bekommt ein künstliches Schlüsselattribut für die Surrogate der Objekte. Für jedes atomare Attribut der Objektklasse wird in der Relation eine Spalte angelegt. Listen– und mengenwertige Attribute werden durch eine eigene Relation repräsentiert. Sie enthält ebenfalls ein künstliches Schlüsselattribut für die Surrogate der Listen– bzw. Mengenelemente. Außerdem wird das Surrogat des zugehörigen Objektes, das Datum selbst und ggf. eine Listenposition gespeichert. Attribute können außerdem eine Record–Struktur besitzen. Sie werden gemäß ihrer Struktur in einzelne Attribute aufgespalten und an die Tabelle der Objektklasse angehängt. Die Generalisierung wird mit Hilfe von konstruierten Objektklassen dargestellt. Eine konstruiertes Objekt erbt die Identität von jeweils einem der zugrundeliegenden Objekte. Deshalb hat die Tabelle einer konstruierten Objektklasse neben dem Surrogat und den eigenen Attributen für jede Ursprungsklasse ein Attribut vom Typ Boolean und zusätzlich noch ein Attribut für das Surrogat der Ursprungsklasse. Zum Beispiel hat die konstruierte Objektklasse "Gewässer" die Ursprungsklassen "Meer", "See", "Fluß" und "Kanal". Die Relation des Gewässers sieht dann so aus: Gewässer (Surr_Gewässer, Surr_Ursprungsklasse, Is_Meer, Is_See, Is_Fluß, Is_Kanal, *Attribute*). Von den Booleschen Is–Attributen kann bei jedem Eintrag nur eines den Wert TRUE haben. Attribute können auch objektwertig sein. Das heißt, die Attributwerte einer Klasse (der sogenannten Wurzelrelation) bestehen aus den Objekten einer anderen Klasse. Diese Beziehung wird durch eine Relation dargestellt. Sie besteht aus einem eigenen Surrogat, dem Surrogat der Wurzelrelation, dem Surrogat des Objektes und ggf. einer Listennummer. Die Nummer wird nur benötigt, wenn das Attribut auch noch listenwertig ist. Für jede Beziehungsklasse wird eine Relation mit einem Surrogat, den Surrogaten der beteiligten Objektklassen und den Attributen der Beziehung angelegt. Lediglich die binären funktionalen Beziehungen benötigen keine eigene Relation. Sie werden mit ihren Attributen an die Funktionsursprungsklasse angehängt. Jede Beziehungsklasse kann beliebig viele beteiligte Objektklassen und Attribute besitzen.
Das EER–Modell von CADDY und das LEU–ER–Modell (siehe Kapitel 3.2) weisen einige Unterschiede auf: CADDY–Relationen sind n–är und können Attribute besitzen; LEU–Relationen sind binär und haben keine Attribute. Die CADDY–Generalisierung vererbt die Objekte eines Vaters an genau einen Sohn; die LEU–Generalisierung vererbt an be-

liebig viele Söhne. In CADDY gibt es außer Listen noch Mengen und Multimengen als Wertebereiche.
Trotz dieser Unterschiede lassen sich aus den Methoden des Prototyp–Generators ein paar Anregungen für die Generierung gewinnen:

- Die Tabellen der Objekttypen und Relationen bekommen automatisch ein zusätzliches Attribut für ein systemweit eindeutiges Surrogat.
- Für Felder vom Typ Liste wird eine eigene Relation erzeugt.
- Für die Umsetzung der Generalisierung sind zusätzliche Attribute erforderlich, die die Zuordnung zwischen Vätern und Söhnen definieren. Aufgrund der verschiedenen Auffassungen von Generalisierung weichen die konkreten Realisierungen aber voneinander ab.

Nachdem in CADDY das relationale Datenbankschema erzeugt wurde, füllt ein Testdatengenerator das Schema mit Testdaten. Benutzerdefinierte Anfragen werden in relationale Anfragen (z.B. in SQL oder Quel) übersetzt. Dann können beliebige Transaktionen auf dem relationalen Schema abgewickelt werden. Auf diese Weise wird der Entwurf getestet und kann ggf. verbessert werden. An diesem Punkt setzt in LEU die Optimierung ein. Während in CADDY der Benutzer selbst die Ergebnisse der Testläufe analysieren muß, schlägt ihm in LEU die Optimierung die nötigen Verbesserungen vor (vgl. Kapitel 6).
Auch im Bereich der Datenerfassung geht LEU einen Schritt weiter. CADDY enthält keine Möglichkeit, Daten zu erfassen. Dies geschieht nach Abschluß der Modellierungsphase direkt auf der Datenbank. Das bedeutet aber auch, wenn erst einmal Daten vorhanden sind, kann das Schema nur noch auf Datenbankebene, beispielsweise über die Anfragesprache SQL, geändert werden. Die Generierung in LEU hingegen unterstützt die Schemaevolution in jeder Phase. Auch wenn schon Daten erfaßt wurden, kann das Schema mit Hilfe des graphischen Editors geändert werden. Wenn anschließend die Generierung angestoßen wird, sorgt sie dafür, daß die Tabellen der neuen graphischen Repräsentation angepaßt werden. Dabei werden so viele Daten wie möglich gerettet (vgl. Kapitel 5.7.6).

4.3 EASYMAP, ein Tool für den logischen Datenbankentwurf

Im Rahmen des DATAID Projektes wurde an der Universität Turin das Werkzeug EASYMAP zur Unterstützung des logischen Datenbankentwurfes entwickelt [BLG82, BLG83]. Ein Ziel war die Abbildung eines konzeptionellen Schemas in ein logisches Schema, das als Eingabe für ein Datenbank–Management–System dient. Dabei stand die Performance bei der späteren Anwendung im Vordergrund. Als Eingabe benötigt EASYMAP ein ER–Modell und Anfragen. Darauf aufbauend wird versucht, den Zugriff auf das Datenmodell möglichst performant zu gestalten. Dabei wird wie folgt vorgegangen:
Zuerst wird für jeden Objekttypen und für jede Relation eine logische Zugriffstabelle erstellt. Diese Tabelle enthält Informationen darüber, welche Attribute von welchen Anfragen angesprochen werden. Es wird außerdem gekennzeichnet, ob der Zugriff über eine Relation oder eine Generalisierung stattfindet. Aus der Häufigkeit, in der die Anfragen pro Zeiteinheit auftreten, wird berechnet, wie oft auf einen Objekttypen bzw. eine Relation zugegriffen wird. Mit Hilfe dieser Informationen wird im zweiten Schritt das Datenmodell neu strukturiert. Die Ziele dabei sind, die Anzahl der Zugriffe auf einen Objekttypen zu reduzieren, und die Menge an Daten, die pro I/O–Operation transferiert wird, zu vermindern. Die Umstrukturierung des Modells umfaßt das Splitten von Objekttypen und das Replizieren von Attributen.
Beim Splitten werden die Attribute der Objekttypen im Idealfall so gruppiert, daß jede Anfrage immer nur auf eine der Gruppen zugreifen muß. Die Gruppe, die den Primärschlüssel enthält, bleibt unter dem Namen des alten Objekttypen erhalten. Die anderen Gruppen werden als Sub–Objekttypen definiert und mit dem oberen Objekttypen über eine 1:1–Relation verbunden.
Das Replizieren von Attributen dient zur Verkürzung langer Zugriffspfade. Grundsätzlich gilt dabei folgende Regel: ein Attribut des Objekttypen X wird im Objekttypen Y repliziert, falls dadurch der Zugriff über X überflüssig wird. Natürlich entstehen dadurch auch Nachteile wegen der doppelten Datenhaltung. Deshalb wird im Einzelfall abgewogen, ob sich das Replizieren lohnt.
Für die Optimierung lassen sich aus dem Ansatz von EASYMAP zwei Anregungen gewinnen: die logischen Zugriffstabellen und das Splitten der Objekttypen. Die Optimierung soll während der Modellierungsphase aufgerufen werden. Zu diesem Zeitpunkt gibt es zwar noch keine konkreten Anfragen, aber der Modellierer kann auf einer abstrakteren Ebene

Anfragen in Form von logischen Zugriffspfaden angeben. Es ist in der Optimierung also möglich, ähnlich wie in EASYMAP aufgrund der Zugriffspfade zu entscheiden, ob Objekttypen gesplittet werden sollten. Der Vorteil in LEU ist, daß die Anfragen selbst noch nicht vorhanden sein müssen. Ebenso kann die Optimierung die Komplexität der Pfade analysieren und zu lange Pfade finden. Wie diese beiden Verfahren in die Optimierung integriert werden, wird im Kapitel 6.6 genauer erläutert.

Kapitel 5

Die Generierung

Dies Kapitel stellt die Generierung vor. Zunächst werden im Abschnitt 5.1 die Anforderungen an die Generierung spezifiziert. Die Rahmenbedingungen für die konzeptionelle Arbeit beschreibt Kapitel 5.2. In 5.3 wird die Repräsentation des LEU–ER–Modell im Data–Dictionary dargestellt.
Anschließend werden Konzepte für Teilprobleme der Generierung vorgestellt. Abschnitt 5.4 erläutert die Umsetzung des LEU–ER–Modells in das Relationen–Modell. Die durch Schemaevolution entstehenden Probleme werden in 5.5 behandelt. Abschnitt 5.6 beschreibt das Transaktionsmanagement.
Im Kapitel 5.7 wird schließlich das Konzept für die Generierung vorgestellt. Die dabei auftretenden Probleme werden diskutiert. Der Entwurf folgt in 5.8. Zuletzt wird in 5.9 der Ablauf der Generierung anhand eines Beispiels erläutert. Dabei wird auch gezeigt, wie die Generierung in LEU eingebaut und benutzt wird.

5.1 Anforderungen

Das Ziel der Generierung ist die Umsetzung eines LEU–ER–Modells in eine relationale Datenbank. Dabei kann es sich sowohl um ein neues als auch um ein altes, geändertes Modell handeln. Alle Bestandteile des LEU–ER–Modells müssen so auf die Datenbank abgebildet werden, daß eine spätere Datenerfassung korrekt und konsistent ablaufen kann. Die folgende Liste spezifiziert die Anforderungen an die Generierung:

Modellbezogene Anforderungen:

- Die LEU–Datentypen und die benutzerdefinierten Datentypen müssen auf die Typen der Datenbank abgebildet werden.
- Die Objekttypen mit ihren Feldern müssen abgebildet werden.
- Die Verbindungen müssen abgebildet werden.
- Die Vererbung muß realisiert werden.
- Die Historisierung muß realisiert werden.
- Der benutzerdefinierte Schlüssel muß angelegt werden.
- Die Schemaevolution muß unterstützt werden: alle Änderungen an einem bestehenden Modell müssen in der Datenbank nachgezogen werden.

Datenbezogene Anforderungen:

- Wenn ein Modell geändert wird, das schon Daten enthält, müssen bei der Änderung in der Datenbank so viele Daten wie möglich erhalten bleiben.

Benutzerbezogene Anforderungen:

- Der Benutzer muß über den Stand der Generierung regelmäßig informiert werden.
- Der Benutzer muß während der Generierung noch bestimmen können, ob die Änderungen von Objekttypen tatsächlich durchgeführt werden, oder ob der alte Stand wieder hergestellt wird.

Konsistenzsichernde Anforderungen:

- Die muß–Felder müssen gekennzeichnet werden.
- Inkonsistenzen, die durch den Übergang von einem in sich konsistenten Modell zu einem anderen in sich konsistenten Modell entstehen, müssen vermieden oder beseitigt werden.
- Für den Fall, daß während der Generierung ein Fehler auftritt, muß die Datenbank wieder in den alten konsistenten Zustand versetzt werden.

5.2 Rahmenbedingungen

Da die Generierung im Rahmen von LEU entwickelt wird, existieren verschiedene Rahmenbedingungen, die bei der Entwicklung berücksichtigt werden müssen:

ANSI–SQL: Alle Informationen, die bei der Entwicklung von LEU und von LEU–Applikationen anfallen, werden in einer Datenbank abgelegt. Zur Kommunikation mit dieser Datenbank wird *Embedded SQL* (SQL zur Einbettung in C–Code) benutzt [ORA90b]. Es soll möglich sein, LEU auf beliebige Datenbanken aufzusetzen. Daher darf bei der Implementierung nur SQL nach dem ANSI–Standard [Ame92] benutzt werden. Erweiterungen, die spezielle Datenbanken bieten, sind nicht verwendbar, da sie unter Umständen nicht auf andere Datenbanken portierbar sind.

Transaktionskonzept von Oracle: Die Datenbank, auf die LEU zuerst aufsetzt, ist ORACLE. Deshalb sind alle Zugriffe auf die Datenbank dem Transaktionskonzept von ORACLE (Kapitel 3.5) unterworfen. Dies muß auch bei der Entwicklung der Generierung berücksichtigt werden.

Eins–zu–eins Übersetzung des LEU–ER–Modells in Tabellen: LEU besteht aus einer Vielzahl von Komponenten, die verschiedene Aufgaben erfüllen. Der Datenmodell–Editor dient zur Erstellung der Schemata von LEU–ER–Modellen. Von dort aus wird auch das Modell in die relationale Datenbank übersetzt. Für die Erfassung von Daten wird eine völlig andere Komponente, die Dialog–Ausführung, aufgerufen. Beide Komponenten arbeiten mit den Tabellen, die das LEU–ER–Modell in der Datenbank repräsentieren: der Editor erstellt sie mit Hilfe der Generierung, und die Ausführung füllt sie mit Daten.

Für die Umsetzung des ER–Modells in das Relationen–Modell gibt es in der Literatur Transformationsregeln [FH89, Mei92, Vos87]. Diese Regeln sehen vor, daß für jeden Objekttypen und für jede n:m–Verbindung des ER–Modells eine Tabelle erzeugt wird. Die 1:1– und 1:n–Verbindungen werden umgesetzt, indem ihre Attribute an die Tabelle eines der verbundenen Objekttypen angehängt werden.

Wenn aber einige der Verbindungen keine eigenen Tabellen besitzen, muß noch eine zusätzliche Information gespeichert werden, die angibt, in welcher der Objekttyp–Tabellen die Daten der betroffenen Verbindungen zu finden sind. Dies würde aber bei jeder Datenmanipulation einen Datenbankzugriff mehr bedeuten. Gerade die Anzahl der Datenbankzugriffe muß aber möglichst gering gehalten werden, da sie gegenüber den C–Funktionen einen erheblichen Teil der Verarbeitungszeit verbrauchen.

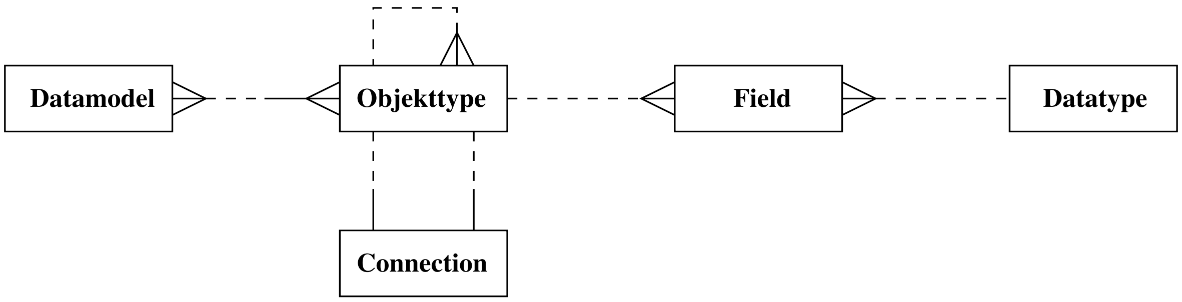
Aus diesem Grund wird in LEU festgelegt, daß das LEU–ER–Modell eins–zu–eins in Tabellen umgesetzt wird. Das heißt, auch für die 1:1– und 1:n–Verbindungen werden eigene Tabellen angelegt. Dadurch ergibt sich eine klare, für alle Komponenten leicht durchschaubare Tabellenstruktur.

Eigene Tabelle für Vererbungs–Beziehung: Die Vererbung wird in LEU durch eine spezielle Verbindung zwischen Vater– und Sohn–Objekttyp realisiert. Diese Verbindung hat die Kardinalität 1:n (ein Vater, mehrere Söhne). Für diese Verbindung treffen dieselben Überlegungen wie für alle anderen 1:n–Verbindungen aus dem vorigen Punkt zu. Deshalb wird festgelegt, daß auch die Vererbungs–Beziehung immer durch eine eigene Tabelle dargestellt wird.

5.3 Repräsentation des LEU–ER–Modells im Data–Dictionary

Mit dem LEU–ER–Modell bietet sich die Möglichkeit zur einheitlichen Verwaltung von Schemata und Inhalten. Darüber hinaus läßt die Unterstützung der Schemaevolution eine gute Wartung der Modelle zu. Deshalb wird das Data–Dictionary von LEU mit Hilfe des LEU–ER–Modells aufgebaut und gepflegt.
Das Data–Dictionary besteht aus den sogenannten **Systemtabellen**. Sie werden zum Zeitpunkt der Entwicklung von LEU angelegt. Diese Tabellen dienen dazu, alle mit LEU modellierten Applikationen (ER–Modelle, Dialoge, Abläufe, etc.) dauerhaft in der Datenbank zu speichern. Ein LEU–ER–Modell ist ein Teil einer solchen Applikation. Der Datenmodell–Editor sichert alle Daten eines erstellten LEU–ER–Modells in seinen Systemtabellen. Dadurch ist es möglich, den Editor zu verlassen, ohne daß das Modell verloren geht. Bei einem erneuten Start des Editors liest dieser die nötigen Informationen aus der Datenbank und kann so die graphische Repräsentation des Modells wieder herstellen.
Die Abbildung 5.1 zeigt den Ausschnitt aus dem Data–Dictionary, der zur vollständigen Sicherung beliebiger LEU–ER–Modelle dient. Er enthält die Objekttypen *Datamodel*, *Objecttype*, *Field*, *Datatype* und *Connection*.
In den folgenden Erläuterungen muß unterschieden werden zwischen dem Datenmodell, das das Data–Dictionary repräsentiert (Abbildung 5.1), und dem *Datamodel*, das ein Objekttyp dieses Data–Dictionaries ist. Dasselbe gilt für Objekttyp und *Objecttype*, Feld und *Field*, Datentyp und *Datatype*, Verbindung und *Connection*. Um Begriffsverwirrungen so weit wie

möglich zu vermeiden, werden die Begriffe, die sich auf Bestandteile des Data–Dictionaries beziehen, normal geschrieben, während für die Namen der Objekttypen und ihrer Felder die *kursive Schrift* gewählt und die englichen Namen beibehalten werden.



Datamodel (Surrogate, State, Ident, Author, GenDate, LockByUser, LastModification)

Objecttype (Surrogate, State, Ident, Author, GenDate, LockByUser, Symbol, Function, SText, HText, HistFlag, CompFlag)

Field (Surrogate, RefSurr, Ident, Aggr, ElementalType, Length, Prec, FieldReference, OTReference, Res, WithIndex, IsFieldKey, Hist, Seqno, ReplaceBehaviour, DefaultValue, OnlineText)

Datatype (Surrogate, RefSurr, Ident, Aggr, ElementalType, Length, Prec, FieldReference, OTReference, ReplaceBehaviour, DefaultValue, OnlineText, LockByUser, State)

Connection (Surrogate, State, DMRef, Ident1, Ident2, Source, Target, SRes, TRes, SDig, TDig)

Abbildung 5.1: Ausschnitt aus dem Data–Dictionary

Jeder Objekttyp des Data–Dictionaries besitzt ein Feld (*Surrogate*), in dem die eindeutigen Bezeichner der Objekte stehen, und das als Schlüssel des Objekttypen verwandt wird. Ebenfalls besitzt fast jeder Objekttyp ein Feld (*Ident*), das einen vom Benutzer angegebenen Bezeichner enthält. Für das *Datamodel* wird außerdem gespeichert, wer (*Author*) es wann (*GenDate*) angelegt hat, wer es gerade bearbeitet (*LockByUser*), welchen Berechtigungsstatus (*State*) es hat und wann es zuletzt geändert wurde (*LastModification*).
Ein *Datamodel* kann natürlich mit mehreren *Objecttypes* verbunden werden. Da *Objecttypes* aber auch exportiert werden dürfen, können sie ihrerseits in mehreren *Datamodels* benutzt werden. Sie müssen aber nur zu einem *Datamodel* gehören. Die Verbindung zwischen

Datamodel und *Objecttype* ist also *n:n/kann:muß*.
Ein *Objecttype* besitzt dieselben Felder wie ein *Datamodel*. Außerdem besitzt er Spalten für Verweise auf eine spezielle Darstellung (*Symbol*) und anzubindende Funktionen (*Function*). Ein kurzer Text (*SText*) für eine Statuszeile im Editor und ein längerer Text (*HText*) für das Hilfesystem können gespeichert werden. Schließlich zeigen zwei Boolean–Felder an, ob der Objekttyp historisiert wird (*HistFlag*) und ob seine Definition vollständig ist (*CompFlag*).
Die Vererbung zwischen *Objecttypes* wird durch eine *1:n/kann:kann*–Verbindung dargestellt. Das bedeutet, ein Vater kann mehrere Söhne haben, und ein Sohn hat höchstens einen Vater.
Ein *Objecttype* kann mehrere *Fields* enthalten. Andererseits existiert ein *Field* nur im Zusammenhang mit genau einem *Objecttype*. Die Verbindung zwischen *Objecttype* und *Field* ist also *1:n/kann:muß*. Für diese Verbindung gibt es keine eigene Tabelle. Stattdessen besitzt der Objekttyp *Field* ein Feld (*RefSurr*), das das Surrogat des zugehörigen *Objecttypes* enthält.
Der *Datatype* eines *Fields* wird mit Hilfe der Attribute *Aggr*, *ElementalType*, *Length*, *Prec*, *FieldReference* und *OTReference* in der *Field*–Tabelle gespeichert. Vier Boolean–Felder geben an, welche Restriktion das *Field* besitzt (*Res*), ob es zum Index gehört (*WithIndex*), ob es zum Schlüssel gehört (*IsFieldKey*) und ob es historisiert wird (*Hist*). *Seqno* gibt die sequentielle Nummer eines *Fields* innerhalb des *Objecttypes* an. Zuletzt werden noch das Schaltverhalten (*ReplaceBehaviour*), ein initialer Wert (*DefaultValue*) und ein Text für die Online–Hilfe (*OnlineText*) gespeichert.
Der *Datatype* des *Fields* wird normalerweise wie beschrieben in der *Field*–Definition abgelegt. Wenn der Typ aber zu komplex ist, um dort in einem Eintrag gespeichert zu werden, wird der Objekttyp *Datatype* benutzt. In seinem Attribut *Refsurr* wird dann auf das Surrogat des *Fields* verwiesen, zu dem der *Datatype* gehört. Die restlichen Attribute sind dieselben wie bei *Field*. Der *Datatype* wird auch zum Speichern von globalen Feldtypen benutzt, die allen Feldern zur Verfügung stehen (vgl. Kapitel 3.2).
Die Verbindung zwischen *Field* und *Datatype* ist *n:1/kann:kann*. Ein *Field* braucht höchstens einen zusätzlichen Eintrag in der *Datatype*–Tabelle, damit sein Typ komplett abgebildet werden kann. Ein globaler *Datatype* kann aber von mehreren *Fields* referenziert werden.
In der *Connection*–Tabelle werden die Verbindungen zwischen den Objekttypen gespeichert, und zwar für jede Seite das Surrogat des angebundenen *Objecttypes* (*Source, Target*), einen Bezeichner (*Ident1, Ident2*), eine Restriktion (*SRes, TRes*) und eine Kardinalität

(*SDig, TDig*). Außerdem wird das Surrogat des *Datamodels*, in dem sich die *Connection* befindet (*DMRef*) und ein Berechtigungsstatus (*State*) abgelegt.
Die beiden Verbindungen zwischen *Objecttype* und *Connection* sind *1:1/kann:muß*. Eine *Connection* kann nur im Zusammenhang mit zwei *Objecttypes* (je einen für Quelle und Ziel) existieren. Ein *Objecttype* muß dagegen keine *Connection* haben.
Eine Beschreibung des LEU–Data–Dictionaries findet man in [GLD+93].

LEU–Datentyp	Typ des Relationen–Modells
Boolean	Number(1)
Integer	Number(n), $0 < n \leq 15$
Real	Number(n,m), $0 < n \leq 16$, $0 \leq m < n$
String_n	Char(n), $n \in \{1, 3, 5, 10, 20, 30, 35, 70\}$
Date	Number(8)
Time	Number(6)
Text	Char(255) und R(Refsurr:Char(40), Seqno:Number(4), Length:Number(7), Content:Long)
Aufzählung	Number(6)
Referenz	Char(40)
Liste(X)	*Typ von X* und R(Refsurr:Char(40), Seqno:Number(6), Element:*Typ von X*)

Abbildung 5.2: Die LEU–Datentypen

5.4 Vom LEU–ER–Modell zum Relationen–Modell

In diesem Abschnitt wird beschrieben, wie das LEU–ER–Modell (3.2) in das Relationen–Modell (3.3) abgebildet wird.
Die Abbildung 5.2 zeigt die LEU–Datentypen und ihre Umsetzung in die Typen des Relationen–Modells. Es wird deutlich, daß nicht alle LEU–Datentypen direkt umgesetzt werden können. Im Relationen–Modell werden die Typen, wenn möglich, durch Number und Char dargestellt. Dies hat den Vorteil, daß eine Konvertierung zwischen den Typen einfach ist, da sie in der Regel von der Datenbank selbständig vorgenommen wird. Für

die beiden Typen *Text* und *Liste*, die nicht durch Number oder Char repräsentiert werden können, werden zusätzliche Tabellen angelegt.
Die einfachen elementaren Typen wie *Boolean*, *Integer*, *Real* und *String* können problemlos umgesetzt werden. Die Typen *Date* und *Time* können als Integer behandelt werden, da sie in ihrer Darstellung nur Zahlen enthalten. Die *Referenz* enthält Zeiger auf Objekte oder Felder. Da jedes Objekt und jedes Feld ein eindeutiges Surrogat besitzt, wird die *Referenz* im Relationen–Modell als Char(40), dem Typ der Surrogate, repräsentiert. Für den Typ *Aufzählung* existiert im LEU–Data–Dictionary eine Tabelle, die die Elemente aller Aufzählungstypen aufnimmt und mit eindeutigen Nummern versieht. Im Relationen–Modell wird dieser Typ daher in Number(6) umgesetzt, so daß die Nummern der Elemente gespeichert werden können. Der Typ *Text* wird durch eine eigene Relation und zusätzlich als Char(255) repräsentiert. Das bedeutet, daß ein Feld vom Typ *Text* im Relationen–Modell den Typ Char(255) besitzt, und darin die ersten 255 Zeichen abgelegt werden. Der gesamte Text wird in der Text–Relation in einem Feld des Typs Long gespeichert. Da auch ein Long–Feld eine bestimmte Länge (in ORACLE 64 Kilobyte) nicht überschreiten kann, werden größere Texte in Blöcke zerlegt und numeriert. Zusätzlich enthält die Relation ein Feld für die Länge des Textes und ein Feld für einen Verweis auf den Eintrag in das Char(255)–Feld. Ähnlich wird der Typ *Liste* umgesetzt. Ein Feld vom Typ *Liste* hat im Relationen–Modell den Typ der Liste. Zum Beispiel wird der Typ *Liste(Integer)* als Integer dargestellt. Dadurch kann in diesem Feld nur ein Element der Liste, der Listenkopf, gespeichert werden. Für die komplette Liste wird eine Listen–Relation erstellt. Sie besteht aus Feldern für die Elemente der Liste, einer Nummer und einem Verweis auf den Listenkopf.
Die nun beschriebene Umsetzung von Objekttypen und Verbindungytypen erfolgt nach Transformationsvorschriften, die in vielen Büchern über Relationale Datenbanken beschrieben werden, zum Beispiel in [FH89, Mei92, Vos87].
Im LEU–ER–Modell (wie auch im ER–Modell) ist ein Feld definiert als die Abbildung eines Objekttypen auf einen Datentyp:

$$A : E \longrightarrow D$$

Ein Objekttyp ist definiert als eine Menge von Feldern:

$$E(A_1 : D_1, \ldots, A_n : D_n)$$

Im Relationen–Modell wird ein Objekttyp als Relation dargestellt. Der Bezeichner des Objekttypen ist der Name der Relation, und die Bezeichner der Felder sind die Namen der Spalten. Die Typen der Spalten entsprechen den relationalen Datentypen, auf die die

Felder abbilden. Die Relation eines Objekttypen wird als Tabelle wie folgt dargestellt:

E		
$A_1 : D_1$	...	$A_n : D_n$

Die Verbindungstypen des ER–Modells sind definiert als eine Menge von Objekttypen und Feldern. Im LEU–ER–Modell ist diese Definition eingeschränkt. Ein Verbindungstyp besteht genau zwischen zwei Objekttypen und besitzt keine eigenen Felder:

$R(E_1, E_2)$

Im Relationen–Modell wird ein Verbindungstyp ebenfalls durch eine Relation repräsentiert. Der Bezeichner des Verbindungstypen ist der Name der Relation. Die Spalten sind die Schlüsselfelder der beiden verbundenen Objekttypen. Da im LEU–ER–Modell alle Objekttypen ein Surrogat als künstlichen Schlüssel besitzen, gibt es in jeder Verbindungs–Relation nur zwei Spalten für die beiden Surrogate der verbundenen Objekttypen. Als Tabelle stellt sich die Relation so dar:

R	
$X_1 : D_1$	$X_2 : D_2$

(X_i ist Schlüsselattribut von E_i)

Auch die 1:1– und 1:n–Verbindungen werden entsprechend den Vorgaben (Kapitel 5.2) in eigene Relationen übersetzt, obwohl sie eigentlich an die Tabellen der verbundenen Objekttypen angehängt werden könnten.

Die Generalisierung wird im ER–Modell als eine spezielle *is a*–Verbindung zwischen zwei Objekttypen verstanden. Im LEU–ER–Modell wird sie Vererbung genannt und ist eine Verbindung zwischen einem Vater–Objekttypen und einem Sohn–Objekttypen. Daher wird sie genau wie eine Verbindung in das Relationen–Modell übersetzt. Der Name der Vererbungs–Relation ist der Bezeichner der Vererbung. Die beiden Felder sind die Schlüsselfelder (Surrogate) der Vater– und Sohn–Objekttypen.

Schließlich gibt es im LEU–ER–Modell noch die Historisierung. Ihr Sinn ist, Folgen von alten Objektwerten aus Objekttypen zu retten. Deshalb wird für einen Objekttypen, der historisiert werden soll, eine zweite Relation mit den zu historisierenden Feldern angelegt. Sie hat genau die gleiche Struktur wie die eigentliche Objekttyp–Relation. Zusätzlich erhält sie ein Feld für eine Zeitangabe, mit der die zu verschiedenen Zeitpunkten historisierten

Daten eines einzelnen Datensatzes unterschieden werden können. Der Name der Tabelle erhält einen Zusatz, damit sie von der Original–Tabelle unterschieden werden kann.
Die Integritätsbedingungen, die im ER–Modell inhärent enthalten sind, werden bei der Datenerfassung sichergestellt. Sie brauchen an dieser Stelle nicht beachtet zu werden.
Die Tabellen des Relationen–Modells können direkt in einer relationalen Datenbank angelegt werden. Sie werden im folgenden auch **Applikationstabellen** genannt, da sie die Objekte der LEU–Applikationen aufnehmen werden.

5.5 Schemaevolution

Der Datenmodell–Editor nimmt bei der Modellierung einige Konsistenzprüfungen vor. Zum Beispiel müssen Bezeichner eindeutig sein, Verbindungen müssen eine Quelle und ein Ziel haben, Zyklen in der Vererbungshierarchie sind nicht modellierbar und Objekttypen oder Felder, die referenziert werden, können nicht gelöscht werden. Diese Tests sind aber objektwert–unabhängig und berücksichtigen immer nur die Definition eines Schemas zu einem bestimmte Zeitpunkt. Wenn das Modell in der Datenbank installiert ist, und Objekte erfaßt werden, können Änderungen des Schemas sowohl Inkonsistenzen als auch Datenverlust hervorrufen. Diese Probleme erkennt der Editor nicht, da sie bei der Schemaevolution entstehen.
Die folgenden Punkte erläutern die Änderungen, die problematisch sind, und stellen Lösungen vor. Für alle Punkte gilt, daß die Probleme nur dann auftreten, wenn schon Tabellen generiert und mit Objekten gefüllt wurden.

1. Die Restriktion eines Feldes wird von *kann* auf *muß* gesetzt. Es kann sein, daß einige Spalten keinen Eintrag für dieses Feld besitzen, da bisher die Datenerfassung freigestellt war. Nun aber muß jede Spalte ein Objekt in diesem Feld stehen haben. Eine einfache Lösung für dies Problem ist, dem Benutzer die Änderung zu verbieten, sobald die betroffene Tabelle Objekte enthält. Dabei würde allerdings die Modellierungsfreiheit des Benutzers eingeschränkt.
   Eine andere Möglichkeit besteht darin, zu testen, ob alle Spalten im betroffenen Feld schon einen Eintrag besitzen. Dann wäre die Restriktion *muß* erfüllt und die Änderung zulässig. Das hieße aber, daß der Benutzer unter Umständen vor der Änderung dafür sorgen müßte, daß alle Objekte eingetragen sind. Auch das ist nicht zumutbar, vor allem, da wahrscheinlich verschiedene Personen für Modellierung und Datenerfassung zuständig sind.

Die dritte Lösung nimmt dem Benutzer einen Teil der Arbeit ab. Bei der Generierung wird diese problematische Änderung der Restriktion erkannt. Die Spalten, die im betroffenen Feld keine Werte enthalten, werden automatisch mit Dummy–Werten aufgefüllt. Die Dummy–Werte müssen zwar im Wertebereich des entsprechenden Feldtypen liegen, sie werden aber so gewählt, daß sie von den anderen Einträgen leicht zu unterscheiden sind. Diese Lösung erlaubt dem Benutzer zum einen, die Änderung durchführen. Zum anderen kann er, wenn er später bei der Datenerfassung Dummy–Werte als Einträge findet, diese durch richtige Werte ersetzen.

2. Ein Objekttyp wird um ein neues Feld mit der Restriktion *muß* erweitert. Die zugehörige Tabelle kann für diese Spalte natürlich noch keine Einträge enthalten, so daß sich dasselbe Problem stellt, wie im vorigen Punkt. Es wird ebenso durch das Eintragen von Dummy–Werten gelöst.

3. Für einen Objekttypen ist bisher kein Schlüssel definiert worden. Nun wird ein neues Feld als Schlüsselfeld hinzugefügt. Dies Feld besitzt entweder die Restriktion *kann* oder *muß*. Ein neues kann–Feld enthält für alle schon vorhandenen Objekte NULL–Werte. Ein neues muß–Feld wird, wie im vorigen Punkt beschrieben, automatisch mit Dummy–Werten gefüllt. In beiden Fällen haben alle Einträge des Schlüssels denselben Wert. Die Forderung, daß ein Schlüssel eindeutig sein muß, ist demnach verletzt. Da ein Eintrag in ein Schlüsselfeld sinnvoll sein sollte, können bei der Generierung auch nicht automatisch verschiedene Einträge erzeugt werden.
   Aus diesen Überlegungen resultiert, daß es verboten wird, nachträglich einen Schlüssel für einen Objekttypen zu definieren, der schon Objekte enthält. Wenn der Benutzer diese Aktion ausführen möchte, wird sie zurückgewiesen.

4. Für einen Objekttypen ist bisher kein Schlüssel definiert worden. Nun wird ein altes Feld zum Schlüsselfeld erklärt. Die Werte dieses Feldes sind aber nicht notwendigerweise eindeutig oder vollständig erfaßt.
   Eine einfache objektwert–unabhängige Lösung ist also, generell zu verbieten, daß ein Feld nachträglich als Schlüsselfeld deklariert wird.
   Eine andere Möglichkeit bietet ein Test, der feststellt, ob das Feld schon die Schlüsseleigenschaft erfüllt. Dieser Test prüft, ob jedes Objekt einen Wert für das zukünftige Schlüsselfeld enthält, und ob diese Werte eindeutig sind. Sind beide Bedingungen erfüllt, wird die Änderung zugelassen.

		nach									
		bool	int	real	string5	string30	text	time	date	Aufz.	Ref.
von	bool	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
	int	$\oslash$	$\surd$	$\surd$	$\surd$	$\surd$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
	real	$\oslash$	$\oslash$	$\surd$	$\surd$	$\surd$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
	string5	$\oslash$	$\oslash$	$\oslash$	$\surd$	$\surd$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
	string30	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\surd$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
	text	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\surd$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
	time	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\surd$	$\oslash$	$\surd$	$\oslash$	$\oslash$	$\oslash$
	date	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\surd$	$\oslash$	$\oslash$	$\surd$	$\oslash$	$\oslash$
	Aufz.	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\surd$	$\oslash$
	Ref.	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\surd$

Abbildung 5.3: Konvertierungen

5. Der Schlüssel eines Objekttypen wird verkleinert, aber nicht ganz gelöscht. Das heißt zum Beispiel, die Länge des Schlüssels wird von drei Feldern auf zwei verkürzt. Dies kann geschehen, indem ein Schlüsselfeld gelöscht wird, oder indem das Schlüssel–Flag eines Feldes von TRUE auf FALSE gesetzt wird. Beim Erfassen der Daten wird nur geprüft, ob die Kombination der Werte aller Schlüsselfelder eindeutig ist. Deshalb sind die Einträge der übrigbleibenden Schlüsselfelder nach der Verkürzung nicht notwendigerweise eindeutig.
   Die Lösung für dies Problem entspricht der einfachen Lösung des vorigen Punktes: diese Operation wird verboten.

6. Der Typ eines Feldes wird derart geändert, daß die Werte nicht in das neue Format konvertiert werden können. Es ist beispielsweise möglich, Werte vom Typ *Integer(5)* in den Typ *String(5)* zu konvertieren. Hingegen ist eine Konvertierung zwischen den Typen *Date* und *Time* undurchführbar. Wenn eine solche Typänderung dennoch modelliert wird, werden die Werte der betroffenen Spalte gelöscht. Die Tabelle in Abbildung 5.3 zeigt, welche Konvertierungen möglich sind ($\surd$) und welche Datenverlust nach sich ziehen ($\oslash$). Dabei stehen *string5* und *string30* als Beispiele für die Konvertierung eines kürzeren in einen längeren String und umgekehrt.
   Neben den in der Tabelle dargestellten Typen gibt es noch den Typ *Liste*. Eine Liste kann den Basistyp Boolean, Integer, Real, String, Date oder Time oder den Typ Aufzählung haben. Wenn der Typ der Liste geändert wird, wird anhand der

Tabelle bestimmt, ob Datenverlust entsteht. Wird ein Feldtyp von Liste auf Basis geändert, so werden der Typ der Liste und der Basistyp anhand der Tabelle verglichen. Aber auch wenn laut Tabelle kein Datenverlust entsteht, ist dennoch ein teilweiser Datenverlust unvermeidbar, da nur das erste Element der Liste behalten wird. Die restlichen Elemente werden gelöscht. Bei einer Umstellung von Basis nach Liste werden ebenfalls der Basistyp und der Typ der Liste verglichen. Datenverlust entsteht nur in den von der Tabelle aufgezeigten Fällen. In allen anderen Fällen werden die Werte des Basistyps als jeweils erste Elemente in die Liste übernommen.

7. Der Typ eines Schlüsselfeldes wird derart geändert, daß Datenverlust entsteht. Wenn es sich um ein kann–Feld handelt, enthält es nach der nächsten Generierung nur noch NULL–Werte. Ein muß–Feld wird mit Dummy–Werten aufgefüllt. In beiden Fällen ist die Eindeutigkeit des Schlüssels nicht mehr gegeben. Deshalb wird auch diese Änderung verboten.

Alle diskutierten Probleme treten schon bei der Modellierung der Felder auf. Es ist also sinnvoll, dem Feld–Editor eine Routine zur Verfügung zu stellen, die alle Änderungen prüft und bewertet. Dadurch können einerseits Inkonsistenzen schon früh verhindert werden, so daß sie bei der Generierung erst gar nicht auftreten. Andererseits kann der Benutzer rechtzeitig vor Datenverlust gewarnt werden.

5.6 Das Transaktionskonzept

Die Aktionen von LEU stützen sich auf das Transaktionskonzept der Datenbank (Kapitel 3.5). Jede LEU–Aktion startet eine eigene Transaktion. Die letzte Operation einer Aktion ist ein *Commit*, wenn alles fehlerfrei abgelaufen ist, oder ein *Rollback*, wenn ein Fehler auftrat. Somit ist mit der Aktion auch die zugehörige Transaktion beendet. Dies Verfahren funktioniert allerdings nicht für Aktionen, die das Anlegen (*Create*) und Löschen (*Drop*) von Tabellen enthalten. Diese beiden Operationen werden von der Datenbank immer mit einem *Commit* begonnen, so daß die Transaktion an dieser Stelle bestätigt und beendet ist. Für die laufende LEU–Aktion beginnt anschließend eine zweite Transaktion. Dadurch kann aber ein *Rollback* nach einem Fehler nur einen Teil der Operationen, nämlich die der letzten begonnenen Transaktion rückgängig machen.

Die Generierung eines Objekttypen ist eine Aktion, die auf der Datenbank als atomare Operation realisiert werden muß, da nur so die Konsistenz des Datenbankschemas und aller seiner Repräsentationen gewährleistet ist. Eine Transaktion ist eine atomare Operation auf

der Datenbank. Da bei der Generierung aber *Create*– und *Drop*–Operationen ausgeführt werden, werden (wie oben beschrieben) mehrere Transaktionen nacheinander gestartet. Das Transaktionskonzept der Datenbank reicht also nicht aus, um die Generierung als atomare Operation ablaufen zu lassen. Deshalb muß für sie ein eigenes Konzept entwickelt werden, das im Fehlerfall nicht nur die letzte Transaktion, sondern alle Transaktionen seit Beginn der Generierungs–Aktion zurücksetzt. Dies Konzept betrifft die Funktionen zum Anlegen neuer Objekttypen (Abschnitt 5.7.5) und zum Anlegen geänderter Objekttypen (Abschnitt 5.7.6). Es sieht wie folgt aus:
Mit jeder dieser drei Funktionen startet wie üblich eine neue Transaktion. Zu Beginn der Funktion werden alle Informationen, die für die Manipulation der Datenbank benötigt werden, gesammelt. Wenn in dieser Phase ein Fehler auftritt, kann die laufende Transaktion wie gewöhnlich mittels *Rollback* zurückgesetzt werden. Tritt kein Fehler auf, beginnt die nächste Phase mit der Manipulation der Datenbank. Dabei werden sowohl *Create*– als auch *Drop*–Operationen ausgeführt. Um diese Operationen, für die ja jeweils eigene Transaktionen gestartet werden, auch wieder zurücksetzen zu können, muß die Generierung selbst entsprechende Vorkehrungen treffen. Deshalb wird zuerst der Name der manipulierten Tabelle zusammen mit der inversen Operation gespeichert. Wenn eine Tabelle gelöscht oder verändert werden soll, wird zusätzlich eine temporäre Kopie angelegt, mit deren Hilfe die ursprüngliche Tabelle wieder hergestellt werden kann. Dann erst wird die Operation selbst ausgeführt. Wenn ein Fehler auftritt, werden anhand der gespeicherten Informationen nacheinander alle Manipulationen rückgängig gemacht. Wenn die Funktion fehlerfrei arbeitet, werden zum Schluß die temporären Tabellen und die gespeicherten Informationen wieder gelöscht.
Mit diesem Konzept besitzt die Generierung ein eigenes Transaktionsmanagement, das alle betroffenen Funktionen auf der Datenbank als atomare Operationen ablaufen läßt.

5.7 Ein Konzept für die Generierung

In der Literatur werden Transformationsregeln zur Abbildung eines ER–Modells auf ein relationales Schema vorgeschlagen [FH89, Mei92, Vos87]. Dieser Ansatz wird im Rahmen der Generierung benutzt (wie in Abschnitt 5.4 beschrieben) und um die Schemaevolution erweitert.
Zwei Anforderungen an die Generierung sind, den Benutzer regelmäßig zu informieren und ihn einige Entscheidungen über den Fortgang selbst treffen zu lassen. Die Generie-

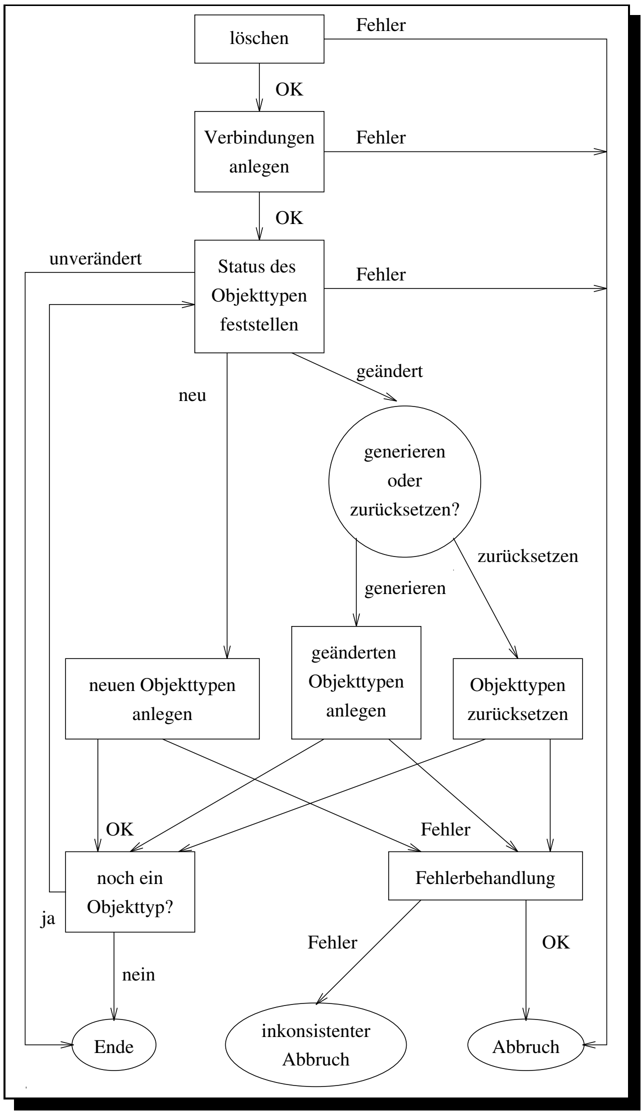


Abbildung 5.4: Steuerung der Generierung

rung ist ein Teil der Datenbankkomponente und als solche nur für die LEU–Komponenten, nicht aber für den Anwender sichtbar. Der Dialog mit dem Benutzer kann nur über den Datenmodell–Editor erfolgen. Um eine Kommunikation zwischen Anwender und Generierung zu ermöglichen, wird also die globale Steuerung der Generierung vom Editor übernommen. Dies ändert aber nichts am Gesamtkonzept der Generierung. Sie besteht aus verschiedenen Funktionen, die nach einem festgelegten Schema aufgerufen werden müssen,

damit sie korrekt zusammenarbeiten. Der Unterschied ist nur, daß diese Aufrufe nicht von einem Hauptprogramm der Generierung sondern vom Editor aus erfolgen.
Die Generierung wird immer für das gerade bearbeitete Datenmodell aufgerufen. Mit dem Aufruf wird ein Prozeß angestoßen, der in Abbildung 5.4 dargestellt ist. Für alle Abbildungen in diesem Kapitel gilt: Funktionen, Algorithmen und Teile aus beiden werden in Rechtecken dargestellt. Ein– und Ausgaben sind oval.
Die folgenden Punkte erläutern die einzelnen Schritte aus Abbildung 5.4. In den darauffolgenden Abschnitten, auf die jeweils verwiesen wird, werden die einzelnen Schritte ausführlich erläutert.

1. Wenn das Datenmodell schon einmal generiert und seitdem geändert wurde, besteht die Möglichkeit, daß Objekttypen oder Verbindungen gelöscht wurden. Beide werden in der Datenbank als Tabellen repräsentiert. Wenn aber Objekttypen oder Verbindungen gelöscht worden sind, müssen auch die dazu gehörenden Tabellen gelöscht werden, da keine Möglichkeit mehr besteht, auf sie zuzugreifen. Deshalb ruft der Editor als erstes eine Routine auf, die überflüssige Tabellen sucht und löscht. Sie wird im Abschnitt 5.7.2 beschrieben.
2. Als nächstes werden die Verbindungen behandelt. Wenn das Modell neu ist, wird für jede Verbindung eine Tabelle angelegt. Sie besteht aus den Schlüsselfeldern der beiden verbundenen Objekttypen. Wurde das Modell schon einmal generiert, werden nur noch Tabellen für Verbindungen angelegt, die seitdem neu hinzugekommen sind. Änderungen brauchen nicht beachtet zu werden, da im Datenmodell–Editor im Zusammenhang mit Verbindungen nur Name, Kardinalität und Restriktion geändert werden können. (Die Repräsentation einer Verbindung wurde in 5.3 beschrieben.) Quelle, Ziel und Surrogat der Verbindung, also die Informationen, die bei der Generierung eine Rolle spielen, bleiben gleich. Aus der Sicht der Generierung kann eine Verbindung nur angelegt und gelöscht werden. Wie die Tabellen genau angelegt werden, wird in Abschnitt 5.7.3 beschrieben.

Jetzt beginnt die Generierung der Objekttypen. Sie wird vom Datenmodell–Editor für jeden Objekttypen einzeln angestoßen. Dabei werden nur die Objekttypen beachtet, die in dem aktuellen Datenmodell beheimatet sind. Globale importierte Objekttypen werden also nicht generiert, da sie ihrerseits in ihrem Heimat–Datenmodell aufgerufen werden. Die folgenden drei Schritte werden für jeden Objekttypen durchgeführt.

3. Zuerst wird der Status des Objekttypen festgestellt. Dieser Test kann drei verschiedene Ergebnisse liefern:

   (a) Der Objekttyp ist neu definiert worden. Er besitzt noch keine Installation in der Datenbank. Ein Objekttyp wird als neu erkannt, wenn das gesamte Modell neu ist, wenn der Objekttyp nach der letzten Generierung angelegt wurde, oder wenn er keine Felder enthält. Im letzten Fall wird er auch in Zukunft neu sein, solange er keine Felder bekommt. Ein Objekttyp ohne Felder wird nicht generiert, da er noch keine Möglichkeit bietet, Objekte zu speichern.

   (b) Der Objekttyp ist alt und unverändert. Er wurde schon einmal generiert. Da er seitdem nicht verändert wurde, ist seine Repräsentation in der Datenbank noch korrekt.

   (c) Der Objekttyp ist alt und wurde geändert. Er wurde schon einmal generiert, und seitdem wurde seine Definition verändert. Der Test kann in diesem Fall folgende Informationen über die Art der Änderungen liefern:

      - Die Änderung erzeugt Datenverlust/keinen Datenverlust.
      - Die Definition mindestens eines Feldes wurde geändert.
      - Es sind neue Felder hinzugekommen.
      - Es sind Felder gelöscht worden.
      - Der Schlüssel wurde verändert.
      - Die Restriktion (kann oder muß) mindestens eines Feldes wurde geändert.
      - Die Historisierung wurde verändert.

      Da mindestens eine der genannten Veränderungen gefunden wurde, besteht eine Diskrepanz zwischen der graphischen Darstellung des Objekttypen und seiner Repräsentation in der Datenbank.

   Im Abschnitt 5.7.4 wird erläutert, wie diese Routine arbeitet.

4. Je nach Ergebnis des Tests stößt der Datenmodell–Editor eine der folgenden Aktionen an:

   (a) Wenn der Objekttyp neu ist, muß er generiert werden. Sofern er Felder besitzt, wird für ihn eine Objekttyp–Tabelle in der Datenbank angelegt. Sie besteht aus je einer Spalte für das Surrogat des Objekttypen und jedes seiner Felder. Felder vom Typ Liste oder Text werden darüber hinaus durch eine eigene Relation repräsentiert. Falls der Objekttyp also derartige Felder besitzt, werden zusätzlich noch Listen– bzw. Text–Tabellen angelegt. Eine Historisierungs–Tabelle wird angelegt, wenn mindestens eins der Felder historisiert werden soll. Schließlich wird eine Vererbungs–Tabelle erstellt, falls der Objekttyp einen Vater hat. Tritt während der Generierung ein Fehler auf, wird mit Hilfe der Fehlerbehandlung (Punkt fünf) der ursprüngliche Zustand wieder hergestellt. Die Generierung neuer Objekttypen wird in Abschnitt 5.7.5 beschrieben.

(b) Im einfachsten Fall ist der Objekttyp alt und unverändert. Dann ist seine Bearbeitung an dieser Stelle beendet. Der Punkt fünf, die Fehlerbehandlung, entfällt ebenfalls.

(c) Falls der Objekttyp alt ist und geändert wurde, informiert der Datenmodell–Editor zuerst den Benutzer. Dieser kann nun zwischen zwei Alternativen wählen: entweder stimmt er der erneuten Generierung zu, oder er verwirft alle Änderungen des Objekttypen. Je nach Entscheidung des Anwenders stößt der Editor eine der beiden folgenden Aktionen an:

   i. Soll der Objekttyp erneut generiert werden, muß seine Repräsentation in der Datenbank korrigiert werden. Der Test lieferte Informationen über sieben mögliche Arten von Änderungen, auf die nun jeweils reagiert wird. Der genaue Ablauf wird im Abschnitt 5.7.6 beschrieben. Generell werden überflüssig gewordene Tabellen gelöscht, neue angelegt und alte angepaßt, so daß die relationale Repräsentation des Objekttypen wieder mit der graphischen übereinstimmt. Dabei werden vorhandene Objekte so weit wie möglich gerettet. Wenn ein Fehler auftritt, wird mit Hilfe der Fehlerbehandlung (Punkt fünf) der ursprüngliche Zustand wieder hergestellt.

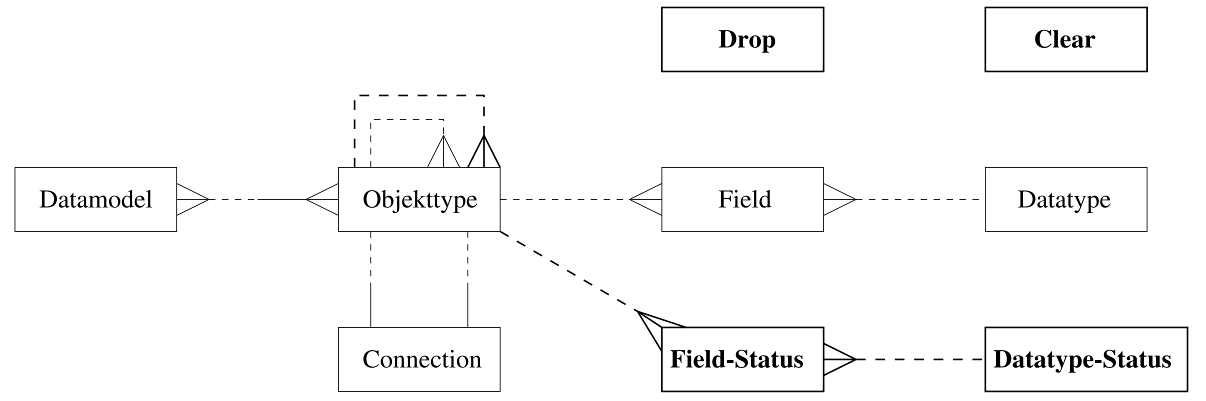
   ii. Wenn die Änderungen verworfen werden sollen, muß die graphische Darstellung des Objekttypen wieder auf den Stand zum Zeitpunkt der letzten Generierung zurückgesetzt werden. Dies geschieht, indem seine Repräsentation im Data–Dictionary korrigiert wird. Seine Tabellen bleiben davon unberührt, da sie sich ja noch im richtigen (alten) Zustand befinden. Wie das Zurücksetzen abläuft, erläutert Abschnitt 5.7.7.

5. Wenn während der Generierung in den Schritten eins bis vier ein Fehler auftritt, wird die gesamte Generierung sofort abgebrochen. Die Funktionen zum Generieren neuer und geänderter Objekttypen brauchen dann eine Fehlerbehandlung, die die Datenbank wieder in einen konsistenten Zustand versetzt. Diese Fehlerbehandlung

arbeitet nach dem in Kapitel 5.6 vorgestellten Transaktionskonzept. Im wesentlichen wird dabei wie folgt vorgegangen: Bei jedem Anlegen oder Löschen einer Tabelle wird das Surrogat des betroffenen Objekttypen, der Name der Tabelle, der Name einer Hilfstabelle (wenn nötig) und die inverse Operation gespeichert. Im Falle eines Fehlers kann die Fehlerbehandlung diese Informationen benutzen, um den alten Datenbankzustand wieder herzustellen. Tritt dabei wiederum ein Fehler auf, kann immer noch der Datenbank–Administrator diese Informationen benutzen, um von Hand einen konsistenten Zustand wieder herzustellen. Die Fehlerbehandlung wird im Abschnitt 5.7.8 ausführlich beschrieben.

5.7.1 Erweiterungen des Data–Dictionaries

Die Generierung braucht für ihre Arbeit neue Systemtabellen. Ihre Aufgabe ist, den alten Status des Modells zum Zeitpunkt der letzten Generierung zu konservieren. Sie werden deshalb zur Unterscheidung von den anderen Systemtabellen in Zukunft als **Statustabellen** bezeichnet. Die Statustabellen ermöglichen einerseits einen Vergleich zwischen dem alten und dem neuen Zustand des Modells. Ein solcher Vergleich ist zum Beispiel bei der Schemaevolution (Kapitel 5.5) erforderlich. Andererseits kann anhand der Statustabellen der alte Zustand des Schemas wieder hergestellt werden (Kapitel 5.7.7).
Die Abbildung 5.5 zeigt den relevanten Ausschnitt des Data–Dictionaries. Die neuen Objekttypen und Verbindungen, die für die Statustabellen angelegt werden, sind fettgedruckt. Um Begriffsverwirrungen so weit wie möglich zu vermeiden, werden wiederum die Begriffe, die sich auf Bestandteile des Data–Dictionaries beziehen, normal geschrieben, während für die Namen der Objekttypen und ihrer Felder die *kursive Schrift* gewählt und die englichen Namen beibehalten werden.
Die Objekttypen *Field–Status* und *Datatype–Status* sind genauso aufgebaut, wie *Field* und *Datatype*. Sie speichern die Definitionen von *Fields* und *Datatypes* zum Zeitpunkt der letzten Generierung. Wenn also die *Fields* nach der Generierung geändert werden, werden diese Änderungen zwar in den Systemtabellen nachgezogen, die Statustabellen bleiben jedoch unberührt. So kann die Generierung einerseits feststellen, was seit dem letzten Aufruf geändert wurde. Andererseits ist sie in der Lage, die Repräsentation der *Objekttypes* wieder auf den Stand, der in den Statustabellen gesichert ist, zurückzusetzen.
Mit der Vererbung der *Objekttypes* verhält es sich ebenso. Eine zweite Verbindungs–Tabelle wird angelegt, die die Vererbung zum Zeitpunkt der letzten Generierung sichert.
Eine andere Aufgabe der Generierung ist, die Tabellen von gelöschten *Objekttypes* und



Felder von Datamodel, Objecttype, Field, Datatype, Connection: siehe Abbildung 5.1

Field-Status (Surrogate, RefSurr, Ident, Aggr, ElementalType, Length, Prec, FieldReference, OTReference, Res, WithIndex, IsFieldKey, Hist, Seqno, ReplaceBehaviour, DefaultValue, OnlineText)

Datatype-Status (Surrogate, RefSurr, Ident, Aggr, ElementalType, Length, Prec, FieldReference, OTReference, ReplaceBehaviour, DefaultValue, OnlineText, LockByUser, State)

Drop (Datamodel, Objecttype, Connection)

Clear (Surrogate, Type, Realname, Helpname)

Abbildung 5.5: Erweiterung des Data–Dictionaries

Connections zu entfernen. Die Information, was gelöscht werden muß, wird in der *Drop*–Tabelle abgelegt. Diese Tabelle besitzt ausnahmsweise kein *Surrogate*. Ihr Schlüssel ist die Kombination aller ihrer Felder. Sie speichert das Surrogat des betroffenen *Datamodels* (im Feld *Datamodel*) und entweder das Surrogat des gelöschten *Objecttypes* (im Feld *Objecttype*) oder der gelöschten *Connection* (im Feld *Connection*).
Schließlich braucht die Fehlerbehandlung noch eine *Clear*–Tabelle. Das Surrogat gehört dem betroffenen *Objecttype*. Außerdem enthält *Clear* die Namen von Tabellen (*Realname*) und Hilfstabellen (*Helpname*) und die Operation, die auf diesen Tabellen ausgeführt werden soll (*Type*). Die genaue Funktion von *Clear* wird im Abschnitt 5.7.8 über die Fehlerbehandlung deutlich.

5.7.2 Löschen alter Daten

Wenn ein Objekttyp aus einem Datenmodell gelöscht wird, trägt die entsprechende Lösch–Routine die beiden Surrogate (von Datenmodell und Objekttyp) in die Drop–Tabelle ein. Ebenso verhält es sich beim Löschen von Verbindungen. Die Generierung liest zu Beginn ihrer Arbeit alle Informationen zu dem behandelten Datenmodell aus der Drop–Tabelle, um die zugehörigen Tabellen aus der Datenbank zu entfernen (Abbildung 5.6). Wurde das Datenmodell noch nie generiert, ist es eigentlich überflüssig, die Drop–Tabelle zu füllen. In diesem Fall sind ja noch keine Applikationstabellen vorhanden. Da die Lösch–Routinen aber nicht wissen können, ob schon generiert wurde, tragen sie ihre Informationen dennoch ein. Dasselbe gilt für Objekttypen, die nach der letzten Generierung angelegt und wieder gelöscht wurden. Beim Auslesen der Drop–Tabelle ist natürlich auch nicht mehr zu entscheiden, ob die Einträge gerechtfertigt sind. Es wird auf jeden Fall versucht, die entsprechende Applikationstabelle zu löschen. Erst die Datenbank wird bei diesem Versuch einen Fehler zurückliefern, der besagt, daß die zu löschende Tabelle nicht existiert. Da dies im Sinne der Generierung kein Fehler ist, wird er einfach ignoriert. Auf diese Weise werden alle wirklich vorhandenen Tabellen auf jeden Fall gelöscht.

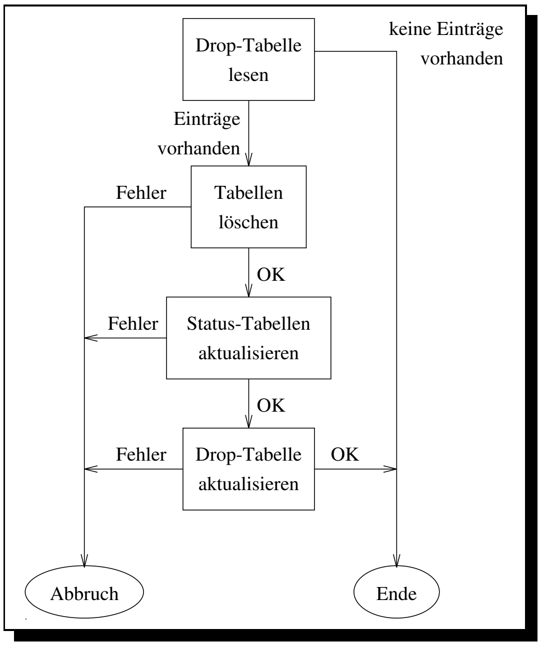


Abbildung 5.6: Löschen alter Daten

Werden Verbindungen gefunden, so reicht es, die entsprechenden Verbindungs–Tabellen (falls vorhanden) zu löschen. Etwas mehr Aufwand ist nötig, wenn Objekttypen gelöscht worden sind, da zu einem Objekttypen mehrere Tabellen existieren können. Zuerst werden alle Felder des Objekttypen überprüft. Werden Felder vom Typ Liste gefunden, so werden die zugehörigen Listen–Tabellen gelöscht. Für Felder vom Typ Text wird jeweils die entsprechende Text–Tabelle entfernt. Als nächstes wird geprüft, ob der Objekttyp historisiert wurde. War dies der Fall, wird die Historisierungs–Tabelle gelöscht. Die Vererbungs–Tabelle wird entfernt, wenn der Objekttyp einen Vater hatte. Die Objekte des Vaters sind davon nicht betroffen. Lediglich die Verbindung zu seinem Sohn wird unterbrochen. Zuletzt wird die Objekttyp–Tabelle selbst gelöscht.

Nun existiert die Repräsentation der behandelten Objekttypen und Verbindungen in der Datenbank nicht mehr. Zumindest die Repräsentation der Objekttypen steht aber immer noch in den Statustabellen. (Verbindungen werden nicht in Statustabellen gesichert, da sie nicht geändert, sondern nur angelegt und gelöscht werden können.) Deshalb werden jetzt alle Informationen über die gelöschen Objekttypen aus den Statustabellen entfernt.

Wenn bis jetzt kein Fehler auftrat, wurden alle Einträge, die die Drop–Tabelle zu dem behandelten Datenmodell liefern konnte, bearbeitet und sind damit überflüssig. Sie werden ebenfalls entfernt.

Falls in irgendeinem Stadium dieser Lösch–Routine ein anderer Fehler auftritt, als der, daß eine Tabelle nicht existiert, bricht die Funktion sofort ab. Es kann nun sein, daß ein Teil der Drop–Tabelle bereits abgearbeitet wurde, so daß einige Tabellen schon gelöscht wurden. Dies ist nicht wieder rückgängig zu machen, da die Datenbank von sich aus kein *Rollback* für das Löschen von Tabellen bietet. Andererseits wurden nur Tabellen gelöscht, die sowieso nicht mehr benötigt werden. Sie wieder herzustellen ist überflüssig. Die Informationen zum Löschen dieser Tabellen stehen aber immer noch in der Drop–Tabelle, da die Funktion abgebrochen wurde, bevor sie entfernt werden konnten. Dies hat zur Folge, daß beim nächsten Aufruf der Lösch–Routine versucht wird, die entsprechenden Tabellen erneut zu entfernen. Die Datenbank wird dabei den Fehler melden, daß die Tabellen nicht existieren. Da dieser Fehler aber immer abgefangen wird, kann die Routine korrekt arbeiten. Die Datenbank ist also in jedem Fall nach dem Verlassen der Lösch–Routine konsistent. Sie enthält höchstens ein paar überflüssige Tabellen und Einträge in die Drop–Tabelle, die die weitere Arbeit nicht beeinflussen.

5.7.3 Installation der Verbindungen

Alle Verbindungen, die im Datenmodell–Editor angelegt werden, werden in die Connection–Tabelle eingetragen. Für die Generierung sind aber nur die Informationen über Surrogat, Datenmodell, Quelle und Ziel der Verbindung relevant. Zuerst werden alle Verbindungen des zu bearbeitenden Datenmodells gelesen (Abbildung 5.7). Für jede gefundene Verbindung wird eine Applikationstabelle angelegt. Der Name einer solchen Tabelle setzt sich zusammen aus dem String "LEU_RE_" und dem Surrogat der Verbindung. Die beiden Felder für Quelle und Ziel bestehen aus dem String "OT_" und dem Surrogat des Quell– beziehungsweise Ziel–Objekttypen. Der Typ beider Felder ist Char(40), da die Surrogate der Applikationsdaten dies Format haben. Die Applikationstabelle für eine Verbindung hat in der Datenbank folgende Struktur:

```
LEU_RE_123456 (OT_987654: CHAR(40),
               OT_909090: CHAR(40))
```

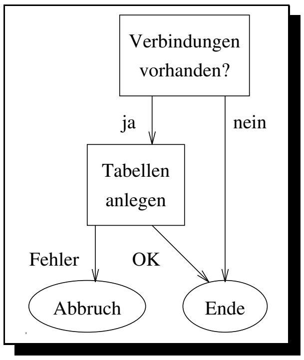


Abbildung 5.7: Generierung von Verbindungen

Da diese Funktion alle Verbindungen des Datenmodells bearbeitet, kann es vorkommen, daß einige der Applikationstabellen aus früheren Generierungen schon existieren. In diesem Fall meldet die Datenbank den Fehler, daß es unter dem angegebenen Namen schon eine Tabelle gibt. Da die Surrogate der Verbindungen datenbankweit eindeutig sind, muß es sich bei der Tabelle um dieselbe handeln, die die Generierung versucht anzulegen. Da die Tabelle also schon existiert, liegt aus der Sicht der Generierung kein Fehler vor, und die Meldung der Datenbank wird ignoriert. Alle Applikationstabellen, die noch nicht existieren, werden korrekt angelegt.

Tritt innerhalb der Funktion ein anderer Fehler auf, wird sie sofort abgebrochen. Tabellen, die bis dahin generiert wurden, brauchen nicht wieder entfernt zu werden. Sie würden beim nächsten erfolgreichen Aufruf dieser Funktion sowieso angelegt, und bis dahin gibt es noch keine Möglichkeit, auf sie zuzugreifen. (Ein Zugriff erfolgt nur aus der LEU–Komponente "Dialog–Ausführung" heraus. Sie kann aber erst für das aktuell bearbeitete Modell gestartet werden, wenn die Generierung fehlerfrei abgeschlossen wurde.) Es ist also zeitsparender, die Tabellen einfach stehen zu lassen und den Fehler, den die Datenbank bei der nächsten Generierung deswegen liefert, zu ignorieren, als die Tabellen wieder zu löschen. Der Zustand der Datenbank ist nach dem Verlassen dieser Routine in jedem Fall konsistent.

5.7.4 Status des Objekttypen

Für jeden Objekttypen des behandelten Datenmodells wird festgestellt, ob er neu, alt und unverändert oder alt und verändert ist (Abbildung 5.8). Ob der Objekttyp neu ist, ist am schnellsten zu prüfen. In der Statustabelle der Felder wird nachgesehen, ob ein Feld zum untersuchten Objekttypen gehört. Die Statustabellen speichern ja die Repräsentation des Modells zum Zeitpunkt der letzten Generierung. Wenn dort kein Eintrag gefunden wird, wurde der zu testende Objekttyp noch nie generiert. Die Funktion kann an dieser Stelle abbrechen, da zur Generierung eines neuen Objekttypen keine weiteren Informationen benötigt werden. Wird jedoch mindestens ein Feld in der Statustabelle gefunden, so ist der Objekttyp schon generiert worden. Er ist also alt. Nun wird festgestellt, ob sich seine Definition geändert hat. Dazu werden sechs verschiedene Tests durchgeführt, deren Ergebnisse an die aufrufende Instanz, den Datenmodell–Editor, zurückgegeben werden. Diese Tests betreffen die Anzahl der Felder, die Felddefinitionen, den Schlüssel, die Historisierung und die Vererbung (vgl. Kapitel 5.5). Zusätzlich wird jeweils festgestellt, ob die Änderung einen Datenverlust hervorruft.
Zuerst wird geprüft, ob Felder gelöscht wurden. Dies läßt sich feststellen, indem in der Statustabelle nach Feldern des Objekttypen gesucht wird, die nicht mehr in der Systemtabelle stehen. Das Löschen von Feldern bedeutet in jedem Fall den Verlust aller in diesen Feldern gespeicherten Daten. Neue Felder werden auf ähnliche Weise gesucht. Sie stehen zwar schon in der Systemtabelle aber noch nicht in der Statustabelle. Das Hinzufügen neuer Felder ist eine Änderung ohne Datenverlust.
Als nächstes werden alle alten noch vorhandenen Felder auf Änderungen ihres Typs geprüft. Dazu wird dieselbe Funktion verwandt, die der Feld–Editor aufruft, um die Änderungen

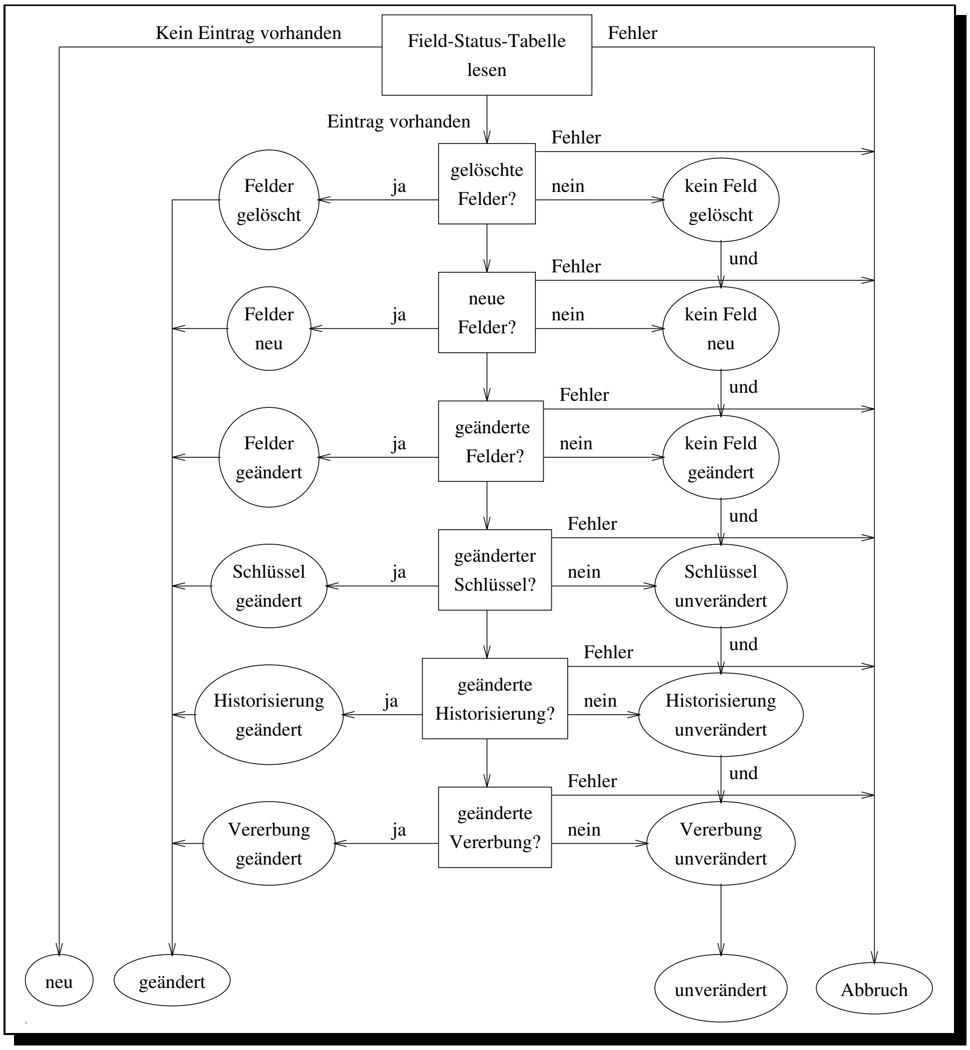


Abbildung 5.8: Status eines Objekttypen feststellen

des Benutzers zu testen (Kapitel 5.5). Der Unterschied ist, daß der Feld–Editor den Test durchführt, um eine konkrete Änderung zu bewerten. An dieser Stelle der Generierung wird hingegen festgestellt, welche Änderungen tatsächlich durchgeführt wurden. Wenn die Funktion für mindestens ein Feld eine Änderung meldet, wurde die Definition des Objekttypen geändert. Diese Änderung ist mit Datenverlust verbunden, wenn bei mindestens einem Feld eine Typänderung mit Datenverlust gefunden wurde.
Nun wird der Schlüssel des Objekttypen getestet. Dazu wird wiederum die Definition des Objekttypen in den Systemtabellen mit jener in den Statustabellen verglichen. Zum einen

werden die Felder gezählt, die neu zum Schlüssel hinzugekommen sind. Es ist aber auch erforderlich, festzustellen, ob es Felder gibt, die nicht länger zum Schlüssel gehören. Denn solange noch keine Objekte erfaßt wurden, ist die Verkürzung des Schlüssels erlaubt, auch wenn der Objekttyp schon generiert wurde. Eine Veränderung des Schlüssels ist immer eine Änderung ohne Datenverlust. Das gilt auch dann, wenn beispielsweise ein Schlüsselfeld gelöscht wurde. Dieser Datenverlust wurde schon festgestellt, als die gelöschten Felder bestimmt wurden.

Die Historisierung wird auf ähnliche Weise getestet wie der Schlüssel. Es werden alle Felder gesucht, die nicht mehr historisiert werden sollen, und alle Felder, deren Historisierung neu beginnt. Falls Felder nicht mehr historisiert werden, ist dies eine Änderung mit Datenverlust, da die betroffenen Felder aus der Historisierungs–Tabelle gelöscht werden.

Zuletzt wird die Vererbung geprüft. Der alte und der neue Vater werden aus der Status– und der Systemtabelle für Vererbung gesucht. Eine Änderung mit Datenverlust liegt vor, wenn die alte Vererbung gelöst oder geändert wurde. In beiden Fällen wurde ein alter Vater gefunden. Ob ein neuer Vater eingetragen wurde, oder ob es keine Vererbung mehr gibt, spielt dabei keine Rolle. Eine Änderung ohne Datenverlust liegt vor, wenn die Vererbung neu ist. In diesem Fall wurde nur ein neuer Vater gefunden.

Wenn keiner der sechs Tests eine Änderung festgestellt hat, so ist der Objekttyp zwar alt, aber seit der letzten Generierung nicht mehr verändert worden. Wird hingegen von mindestens einem der oben beschriebenen Tests eine Änderung angezeigt, muß der Objekttyp von der Generierung als geändert betrachtet werden. In diesem Fall werden die Ergebnisse der einzelnen Tests an die aufrufende LEU–Komponente, den Datenmodell–Editor, zurückgeliefert.

5.7.5 Installation neuer Objekttypen

Die Routine zum Anlegen neuer Objekttypen (Abbildung 5.9) wird nur dann aufgerufen, wenn der Test zuvor ergeben hat, daß der Objekttyp neu ist. Daher kann sie davon ausgehen, daß noch keine der anzulegenden Tabellen vorhanden ist. Die Funktion arbeitet in vier Phasen: Vorbereitung, Tabellen anlegen, Statustabellen aktualisieren und Generierung des Objekttypen beenden.

Phase 1: Vorbereitung. In der ersten Phase (Abbildung 5.10) werden alle Informationen bzgl. des bearbeiteten Objekttypen aus den Systemtabellen gelesen und ausgewertet. Es wird entschieden, welche Applikations–Tabellen angelegt werden müssen, um alle Ob-

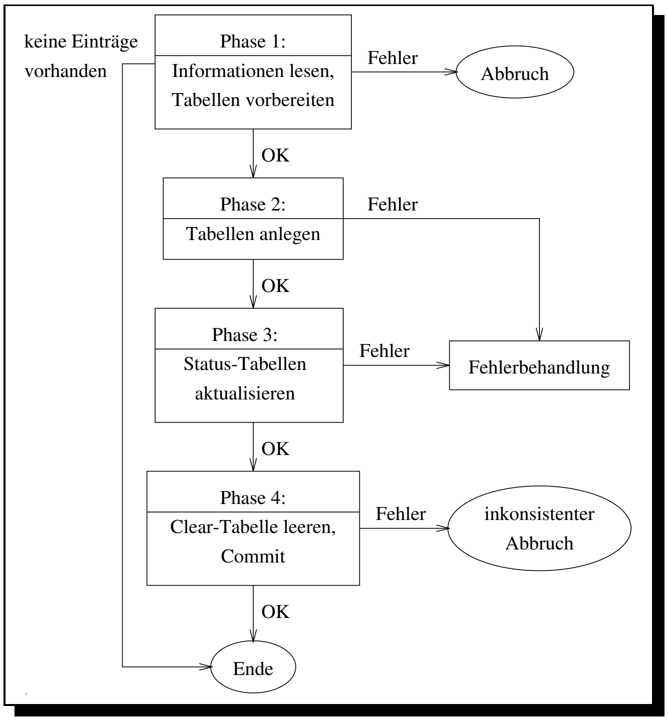


Abbildung 5.9: Generierung neuer Objekttypen

jekte des Objekttypen speichern zu können. Zuerst werden die Felder und ihre Datentypen gelesen. Werden keine Felder gefunden, so ist die Generierung dieses Objekttypen schon beendet. Ein Objekttyp ohne Felder ist nicht in der Lage, Objekte aufzunehmen. Deshalb braucht er auch noch keine Applikations–Tabellen. Wenn aber Felder vorhanden sind, werden sie auf verschiedene Merkmale hin getestet. Für jedes Feld vom Typ Liste wird vermerkt, daß eine Listen–Tabelle anzulegen ist. Ebenso wird für jedes Feld vom Typ Text vermerkt, daß eine Text–Tabelle zu generieren ist. Wenn Felder gefunden werden, die historisiert werden sollen, wird eine Historisierungs–Tabelle vorgemerkt. Und falls es Schlüsselfelder gibt, muß auch noch ein eindeutiger Index auf diese Felder der Objekttyp–Tabelle gelegt werden. Durch den Index wird sichergestellt, daß die Kombination der Werte aller Schlüsselfelder immer eindeutig ist. Nun wird noch festgestellt, ob der Objekttyp einen Vater hat. Ist dies der Fall, wird vermerkt, daß eine Vererbungs–Tabelle zu generieren ist. Diese Tabelle besteht aus genau zwei Spalten: je eine für die Surrogate von Vater und Sohn. Natürlich könnte das Surrogat des Vaters als Fremdschlüssel in die Tabelle des Sohnes aufgenommen werden, um so die Vererbungs–Tabelle ganz zu sparen.

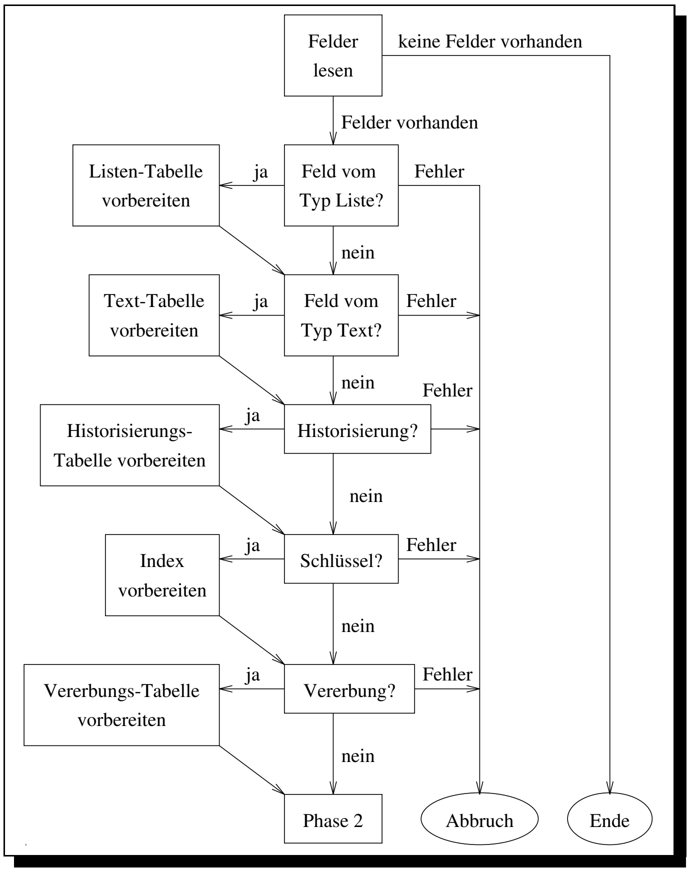


Abbildung 5.10: Generierung neuer Objekttypen: Phase 1

In den Vorgaben für die Generierung ist aber festgelegt worden, daß die Vererbung immer durch eine eigene Tabelle dargestellt wird (Kapitel 5.2).
Wenn während dieser ersten Phase ein Fehler auftritt, wird die Funktion beendet und eine Fehlermeldung wird zurückgegeben. Da bisher nur Daten gelesen wurden, hat sich der Zustand der Datenbank nicht verändert. Die Generierung kann problemlos abgebrochen werden. Wenn kein Fehler auftrat, beginnt die zweite Phase (Abbildung 5.11).

Phase 2: Tabellen anlegen. Es ist nun bekannt, welche Tabellen generiert werden müssen, um den Objekttypen vollständig in der Datenbank zu repräsentieren. Nacheinander werden die Tabellen für den Objekttypen selbst, die Vererbung, die Historisierung, die Listenfelder und die Textfelder angelegt, sofern in der ersten Phase jeweils festge-

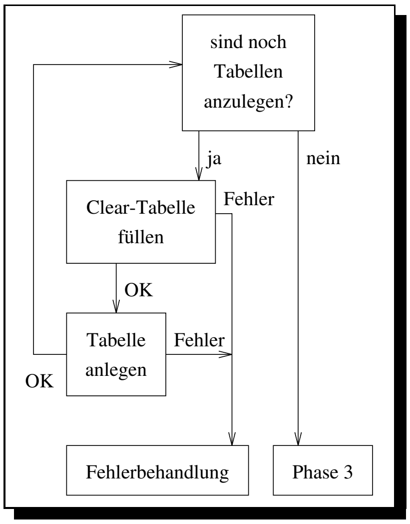


Abbildung 5.11: Generierung neuer Objekttypen: Phase 2

stellt wurde, daß sie gebraucht werden (Abbildung 5.11). Die Objekttyp–Tabelle bekommt den Namen ”LEU_OT_<*Surrogat des Objekttypen*>”. Sie hat ein Feld ”Surrogate” vom Typ Char(40) für die Surrogate der Applikationsobjekte. Die anderen Felder werden mit ”F_<*Surrogat des Feldes*>” bezeichnet und haben den Typ, den der Benutzer für das entsprechende Feld vorgesehen hat. Die Vererbungs–Tabelle heißt ”LEU_IN_<*Surrogat des Objekttypen*>”. Sie hat zwei Felder, ”OT_<*Surrogat des Vaters*>” und ”OT_<*Surrogat des Objekttypen*>”, vom Typ Char(40) für die Surrogate der Väter und Söhne. Die Historisierungs–Tabelle unterscheidet sich von der Objekttyp–Tabelle lediglich durch ihren Namen, ”LEU_OTH_<*Surrogat des Objekttypen*>”, ein Feld ”Date” für den Zeitpunkt der Historisierung und evtl. durch die Anzahl ihrer Felder. Die Listen–Tabellen werden ”LEU_LI_<*Surrogat des Listenfeldes*>” genannt. Sie besitzen ein Feld ”Refsurr” vom Typ Char(40) für das Surrogat des Objektes, zu dem die Liste gehört, ein Feld ”Seqno” vom Typ Number(6) für die sequentiellen Nummern der Listenelemente und ein Feld ”Element” vom Typ der Liste für die Elemente selbst. Die Text–Tabellen heißen ”LEU_TE_<*Surrogat des Textfeldes*>”. Sie haben ebenfalls die Felder ”Refsurr” und ”Seqno”. Außerdem besitzten sie ein Feld ”Length” für die Länge des Textes und ein Feld ”Content”, in dem der Text selbst in maximal 64 Kilobyte großen Fragmenten abgelegt wird. Ein Text, der diese Größe überschreitet, wird in mehreren Fragmenten mit sequentiellen Nummern abgelegt. Die Applikationstabellen für einen Objekttypen haben in der Datenbank also folgende

Struktur:

```
LEU_OT_123456 (Surrogate: CHAR(40),
               F_111111: NUMBER(10),
               F_222222: NUMBER(1),
               F_333333: CHAR(255))

LEU_OTH_123456 (Surrogate: CHAR(40),
                Date: NUMBER(12),
                F_222222: NUMBER(1))

LEU_IN_123456 (OT_987654: CHAR(40),
               OT_123456:CHAR(40))

LEU_TE_333333 (Refsurr: CHAR(40),
               Seqno: NUMBER(4),
               Length: NUMBER(7),
               Content: LONG))

LEU_LI_111111 (Refsurr: CHAR(40),
               Seqno: NUMBER(6),
               Element: NUMBER(10))
```

Das Generieren einer Tabelle ist unabhängig von ihrer Definition und ihrer späteren Verwendung. Es wird deshalb im folgenden exemplarisch für eine beliebige Tabelle beschrieben.
Da das Anlegen einer Tabelle in der Datenbank entgültig ist, muß zunächst sichergestellt werden, daß im Fehlerfall die Tabelle wieder entfernt werden kann (siehe: Transaktionskonzept, Abschnitt 5.6). Die zu diesem Zweck erzeugte Clear–Tabelle wird mit dem Surrogat des Objekttypen, dem Tabellennamen und der zum Anlegen inversen Operation *Drop* gefüllt. Die Spalte für den Namen einer Hilfstabelle bleibt leer, da zum Generieren einer neuen Tabelle keine Hilfstabelle benötigt wird. Wenn beim Füllen der Clear–Tabelle ein Fehler auftritt, wird die Fehlerbehandlung aufgerufen, sofern die behandelte Tabelle nicht gerade die erste ist. Denn alle vorher angelegten Tabellen müssen wieder entfernt werden, damit der Objekttyp nach dem gescheiterten Versuch der Generierung immer noch neu ist. Wenn aber der Eintrag in die Clear–Tabelle erfolgreich war, wird nun die Tabelle in der Datenbank erzeugt. Tritt hierbei ein Fehler auf, wird ebenfalls die Fehlerbehandlung aufgerufen.
Auf diese Weise werden nacheinander alle Tabellen des Objekttypen generiert. Anschließend wird noch ein eindeutiger Index auf die Kombination der Schlüsselfelder in der Objekttyp–Tabelle gelegt. Da die Datenbank beim Eintragen von Daten darauf achtet, daß die Werte der Index–Felder eindeutig sind, wird auf diese Weise der benutzerdefinierte Schlüssel überwacht. Der Index muß nicht in die Clear–Tabelle eingetragen werden, da er

beim Löschen der Objekttyp–Tabelle automatisch mit entfernt wird. (Der SQL–Befehl *drop table* löscht eine Tabelle mitsamt ihren Indizes [ORA90b].) Im Fehlerfall wird wiederum die Fehlerbehandlung aufgerufen. Andernfalls beginnt die dritte Phase der Generierung neuer Objekttypen.

Phase 3: Statustabellen aktualisieren. Die Repräsentation des Objekttypen in der Datenbank ist nun vollständig. Sie muß jetzt noch in den Statustabellen abgelegt werden. Zu diesem Zweck werden einfach alle Informationen zu dem Objekttypen aus den Systemtabellen in die entsprechenden Statustabellen kopiert. Auch hierbei wird die Fehlerbehandlung aufgerufen, sobald ein Fehler auftritt. Solange die Statustabellen nämlich nicht vollständig gefüllt sind, besteht eine Diskrepanz zwischen der Repräsentation des Objekttypen in der Datenbank und seiner Beschreibung in diesen Tabellen.

Phase 4: Funktion beenden. Wenn die Generierung bis zu dieser Stelle fehlerfrei lief, sind die Informationen in der Clear–Tabelle überflüssig. Sie werden wieder entfernt. Nun ist die Generierung des neuen Objekttypen abgeschlossen. Die Datenbank befindet sich wieder in einem konsistenten Zustand. Deshalb wird zuletzt noch ein *Commit* in der Datenbank abgesetzt, um diesen Zustand entgültig zu bestätigen. Wenn bei einer dieser beiden letzten Operationen ein Fehler auftritt, wird die aufrufende LEU–Komponente, der Datenmodell–Editor, über den inkonsistenten Abbruch informiert. Denn dann enthält die Clear–Tabelle noch immer die Daten des behandelten Objekttypen, obwohl seine Generierung korrekt abgeschlossen wurde. Diese Inkonsistenz wird behoben, indem der Editor seinerseits die Fehlerbehandung aufruft.

5.7.6 Installation geänderter Objekttypen

Die Routine zum Anlegen geänderter Objekttypen (Abbildung 5.12) wird nur dann aufgerufen, wenn der Test zuvor ergeben hat, daß der Objekttyp alt ist und geändert wurde, und wenn sich der Benutzer für die erneute Generierung entschieden hat. Diese Routine bekommt die Ergebnisse der Test–Routine (Kapitel 5.7.4) beim Aufruf mitgeliefert, damit sie auf die verschiedenen Änderungen speziell reagieren kann, ohne die Tests selbst noch einmal durchzuführen. Sie arbeitet wie die Generierung neuer Objekttypen in vier Phasen: Vorbereitung, Tabellen ändern, Statustabellen aktualisieren und Generierung des Objekttypen beenden.

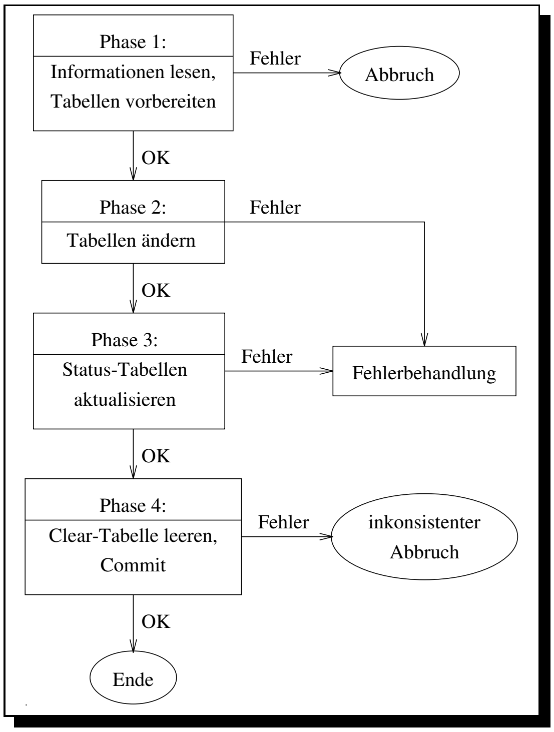


Abbildung 5.12: Generierung geänderter Objekttypen

Phase 1: Vorbereitung. In der ersten Phase (Abbildung 5.13) wird festgestellt, welche Operationen auf welchen Tabellen ausgeführt werden müssen. Zuerst wird die Vererbung behandelt. Die nötigen Informationen liefern die Testergebnisse. Wenn eine Vererbung gelöst oder geändert wurde, wird das Löschen der alten Vererbungs–Tabelle vermerkt. Gibt es eine neue oder geänderte Vererbung, wird eine neue Vererbungs–Tabelle vorbereitet. Als nächstes werden gelöschte Listen– und Text–Felder gesucht. Die Tabellen dieser Felder müssen gelöscht werden.
Die Objekttyp–Tabelle muß geändert werden, wenn die Testergebnisse Änderungen beim Schlüssel, der Anzahl oder der Definition der Felder aufweisen. Für diese Änderung wird in der ersten Version der Implementierung grundsätzlich eine neue Tabelle aufgebaut. Dann werden alle Werte, für die es möglich ist, aus der alten in die neue Tabelle kopiert. Zuletzt wird die alte Tabelle gelöscht. In einer nächsten Version kann dieser Vorgang weiter differenziert werden. Beispielsweise ist es überflüssig, eine neue Tabelle anzulegen, wenn nur ein Feld hinzugefügt werden muß. In diesem Fall könnte die alte Tabelle einfach um ein

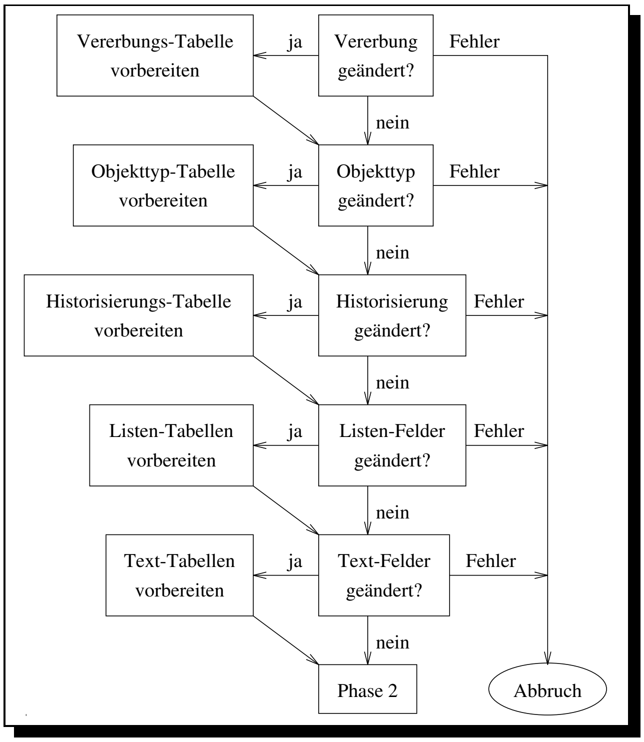


Abbildung 5.13: Generierung geänderter Objekttypen: Phase1

Feld erweitert werden.
Um die neue Objekttyp–Tabelle aufzubauen, werden zunächst die Definitionen aller Felder aus den Systemtabellen gelesen. Die Felder werden nun nacheinander betrachtet. Dabei werden schrittweise verschiedene temporäre Listen aufgebaut, die die Veränderung des Objekttypen darstellen. Sie werden später zur Änderung der Tabellen benötigt. Eine Objekttyp–Liste nimmt alle Felder auf, die zur neuen Definition des Objekttypen gehören. Eine Insert–Liste wird für die Felder angelegt, deren Werte aus der alten in die neue Objekttyp–Tabelle übernommen werden können. Eine Index–Liste enthält alle Schlüssel–Felder, um den eindeutigen Index anzulegen. In die Historisierungs–Liste werden die Felder eingetragen, die zur neuen Historisierungs–Tabelle gehören. Für die Felder, deren Werte aus der alten in die neue Historisierungs–Tabelle übernommen werden können, wird eine Hist_Insert–Liste aufgebaut. Zusätzlich gibt es noch zwei besondere Listen für Felder vom Typ Liste und Text.
Ein Feld wird zuerst auf die Änderung seines Typs getestet. Dazu wird wiederum dieselbe

Funktion aufgerufen, die der Feld–Editor benutzt, um die Änderungen des Benutzers zu bewerten (Kapitel 5.5). Anhand der Tabelle in Abbildung 5.3 wird festgestellt, ob eine Änderung des Feldtyps Datenverlust erzeugt. Die Werte des Feldes müssen in die neue Tabelle übernommen werden, falls es schon existierte und entweder unverändert ist, oder höchstens teilweiser Datenverlust gefunden wurde. Es wird also in die Insert–Liste eingetragen. Falls es historisiert wird, wird es zusätzlich in die Hist_Insert–Liste eingetragen. Einen Spezialfall bilden die Felder mit der Restriktion *muß*. Sie werden in jedem Fall in die Insert–Liste eingetragen. Allerdings werden für sie nicht nur die alten Werte übernommen. Zusätzlich wird vermerkt, daß alle leeren Einträge dieser Felder mit Dummy–Werten aufgefüllt werden müssen. Diese Behandlung gilt aber nur für die Objekttyp–Tabelle. Die Historisierungs–Tabelle enthält keine muß–Felder, denn diese Restriktion soll nur sicherstellen, daß bei der Datenerfassung die entsprechenden Felder immer gefüllt werden. Die Historisierung dient aber zur Sicherung von Objekten bei einer Änderung, so daß eine Überwachung der Restriktion überflüssig ist. Ein muß–Feld wird in der Datenbank als *not null* gekennzeichnet. Dadurch überwacht die Datenbank beim Eintragen von Applikationsdaten automatisch, ob alle diese Felder Einträge enthalten. Wenn das Feld zum Schlüssel gehört, wird es in die Index–Liste eingetragen. Da die Felder aus der Systemtabelle gelesen wurden, gehören alle hier behandelten Felder automatisch zur neuen Objekttyp–Tabelle. Sie werden also in die Objekttyp–Liste eingetragen. Ein Feld, das historisiert wird, wird zusätzlich in die Historisierungs–Liste aufgenommen.
Ein **Feld vom Typ Liste** wird speziell behandelt. Es gibt fünf Arten von Typänderungen:

1. Umstellung von Liste nach Liste ohne Datenverlust: die Listen–Tabelle wird geändert; die alten Werte werden übernommen.

2. Umstellung von Basistyp nach Liste ohne Datenverlust: die Listen–Tabelle wird neu erzeugt; die alten Werte werden als jeweils erste Elemente in die Tabelle übernommen.

3. Umstellung von Liste nach Liste mit Datenverlust: die Listen–Tabelle wird geändert.

4. Umstellung von Basistyp nach Liste mit Datenverlust oder neues Listenfeld: die Listen–Tabelle wird neu erzeugt; bei muß–Feldern werden die Dummy–Einträge aus der Objekttyp–Tabelle übernommen.

5. Umstellung von Liste nach Basistyp oder Listenfeld gelöscht: die Listen–Tabelle wird gelöscht.

Die gelöschten Listenfelder wurden weiter oben schon erfaßt. Sie können an dieser Stelle nicht mehr gefunden werden, da sie nicht in den Systemtabellen, sondern nur noch in den Statustabellen stehen. Eine Behandlung von muß–Feldern ist an dieser Stelle nicht mehr erforderlich, da das erste Element jeder Liste auch in die Objekttyp–Tabelle eingetragen wird. Die Restriktion muß ist also bei der Behandlung dieser Tabelle schon erfüllt. Aus demselben Grund reicht es im fünften Fall, die Listen–Tabelle zu löschen. Wenn durch die Umstellung Datenverlust auftrat, wurde für die Entfernung des übrigbleibenden ersten Elementes aus der Objekttyp–Tabelle schon gesorgt, indem das betreffende Feld nicht in die Insert–Liste aufgenommen wurde. Wenn ein teilweiser Datenverlust auftrat, wurde das Feld in die Liste eingetragen, so daß das erste Element gerettet wird. Nur die restlichen Elemente werden mit der Listen–Tabelle gelöscht.
Ein **Feld vom Typ Text** wird ebenfalls besonders behandelt. Hier gibt es zwei Fälle:

1. Umstellung von <nicht Text> nach Text oder neues Text–Feld: die Text–Tabelle wird neu erzeugt; bei muß–Feldern werden die Dummy–Einträge aus der Objekttyp–Tabelle übernommen.

2. Umstellung von Text nach <nicht Text> oder Text–Feld gelöscht: die Text–Tabelle wird gelöscht.

Die gelöschten Text–Felder wurden wie die Listen–Felder oben schon bearbeitet, da sie hier nicht mehr gefunden werden. Die Restriktion muß ist bereits erfüllt. Die ersten 255 Zeichen jedes Textes werden auch in der Objekttyp–Tabelle gehalten, so daß dort schon Dummy–Werte eingetragen wurden. Da der Typ Text zu allen anderen Typen inkompatibel ist, entfallen alle Überlegungen bzgl. der Übernahme von Daten.
Wenn während dieser ersten Phase ein Fehler auftritt, wird die Funktion beendet und eine Fehlermeldung wird zurückgegeben. Da bisher nur Daten gelesen wurden, hat sich der Zustand der Datenbank nicht verändert. Die Generierung kann problemlos abgebrochen werden. Wenn kein Fehler auftrat, beginnt die zweite Phase.

Einleitende Erläuterungen zu Phase 2. Es gibt drei Operationen, die im folgenden auf den verschiedenen Tabellen ausgeführt werden müssen: löschen, anlegen und ändern (Abbildung 5.14). Da die Generierung geänderter Objekttypen auf der Datenbank eine atomare Aktion sein soll (siehe: Transaktionskonzept, Abschnitt 5.6), muß bei einem Fehler innerhalb dieser drei Operationen die Fehlerbehandlung aufgerufen werden. Deshalb ist ihre Ausführung etwas komplexer.

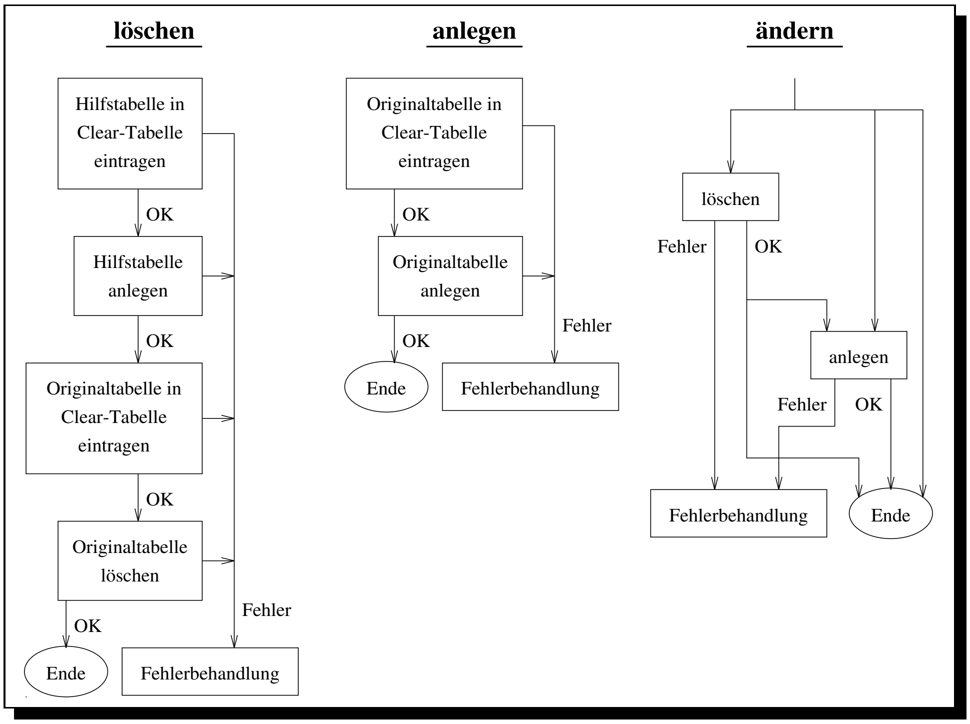


Abbildung 5.14: Tabellen löschen, anlegen und ändern

Das **Löschen einer Tabelle** verläuft wie folgt: Eine Tabelle, die gelöscht wird, muß im Fehlerfall wieder hergestellt werden können. Deshalb wird eine Hilfstabelle erzeugt, die die Struktur und die Daten der zu löschenden Tabelle übernimmt. Aber auch diese Hilfstabelle muß nach einem Fehler oder nach Beendigung der Generierung wieder entfernt werden. Für derartige Informationen gibt es die Clear–Tabelle. Also wird zuerst der Name der Hilfstabelle in die Clear–Tabelle eingetragen. Dabei werden die Spalten für das Surrogat des behandelten Objekttypen, den Namen der Hilfstabelle und die inverse Operation (hier: Drop) gefüllt. Dann wird die Hilfstabelle generiert. Es ist nun möglich, daß die Hilfstabelle noch existiert, und die Datenbank einen Fehler meldet. Diese Situation entsteht dadurch, daß das spätere Entfernen der Hilfstabellen aus der Datenbank nicht mehr überwacht wird. Wenn also eine Hilfstabelle nach einer früheren Generierung nicht gelöscht werden konnte, blieb sie einfach in der Datenbank stehen. Eine überflüssige Tabelle, die von niemandem mehr angesprochen wird, belegt zwar ein wenig Platz, aber es lohnt sich nicht, deswegen die ganze Generierung abzubrechen und den betroffenen Objekttypen wieder auf seinen alten Stand zu setzen. Wenn also die Hilfstabelle noch existiert, wird sie einfach gelöscht

und noch einmal angelegt. Wenn dabei allerdings ein anderer Fehler auftritt, wird die Generierung abgebrochen und die Fehlerbehandlung aufgerufen. Konnte die Hilfstabelle angelegt werden, muß als nächstes die eigentliche Tabelle gelöscht werden. Aber auch das geht nicht ohne Vorbereitung für die Fehlerbehandlung. Wiederum wird die Clear–Tabelle benutzt. Diesmal wird das Surrogate des Objekttypen, der Name der Originaltabelle, der Name der Hilfstabelle und die inverse Operation (hier: Create) eingetragen. Nun wird die Tabelle gelöscht. Tritt hierbei ein Fehler auf, wird die Generierung ebenfalls abgebrochen und die Fehlerbehandlung aufgerufen.
Das **Anlegen einer Tabelle** verläuft wie der entsprechende Vorgang bei der Generierung neuer Objekttypen: zuerst wird die Clear–Tabelle mit dem Surrogat des Objekttypen, dem Tabellennamen und der inversen Operation (hier: Drop) gefüllt. Dann wird die Tabelle generiert. Tritt ein Fehler auf, wird die Generierung abgebrochen und die Fehlerbehandlung aufgerufen.
Das **Ändern einer Tabelle** kann verschiedene Bedeutungen haben: nur löschen, nur anlegen oder löschen und anlegen.
Im folgenden ist mit "löschen", "anlegen" und "ändern" jeweils der entsprechende oben beschriebene Vorgang gemeint.

Phase 2: Tabellen ändern. Nun wird ein kompletter Durchgang durch die zweite Phase (Abbildung 5.15) beschrieben. Die Operationen werden aber tatsächlich nur dann ausgeführt, wenn sie schon in der ersten Phase vorgemerkt wurden.
Wenn die Objekttyp–Liste keine Felder für eine neue Tabelle enthält, werden alle noch existierenden Tabellen des Objekttypen gelöscht, da ein Objekttyp ohne Felder keine Objekte aufnehmen kann. Er ist für die nächste Generierung damit wieder als neu zu betrachten. Wenn Felder vorhanden sind, wird zuerst die alte Vererbungs–Tabelle gelöscht und die neue angelegt. Dasselbe geschieht mit der Objekttyp–Tabelle. Wenn die Insert–Liste Felder enthält, müssen nun die alten Objekte in die neue Tabelle eingetragen werden. Dazu gibt es zwei Fälle. Wenn es keine muß–Felder gibt, die noch mit Dummy–Werten aufgefüllt werden müssen, werden die Objekte aus der dafür angelegten Hilfstabelle importiert. Gibt es muß–Felder, so wird eine Muß–Tabelle auf die gleich Weise angelegt, wie zuvor die Hilfstabelle. In dieser Tabelle werden alle zukünftigen muß–Felder, die noch leere Einträge haben, mit Dummy–Werten aufgefüllt. Anschließend werden alle Objekte aus dieser Tabelle in die neue Objekttyp–Tabelle kopiert.
Aus folgenden Gründen ist es erforderlich, eine Muß–Tabelle anzulegen: in der neuen

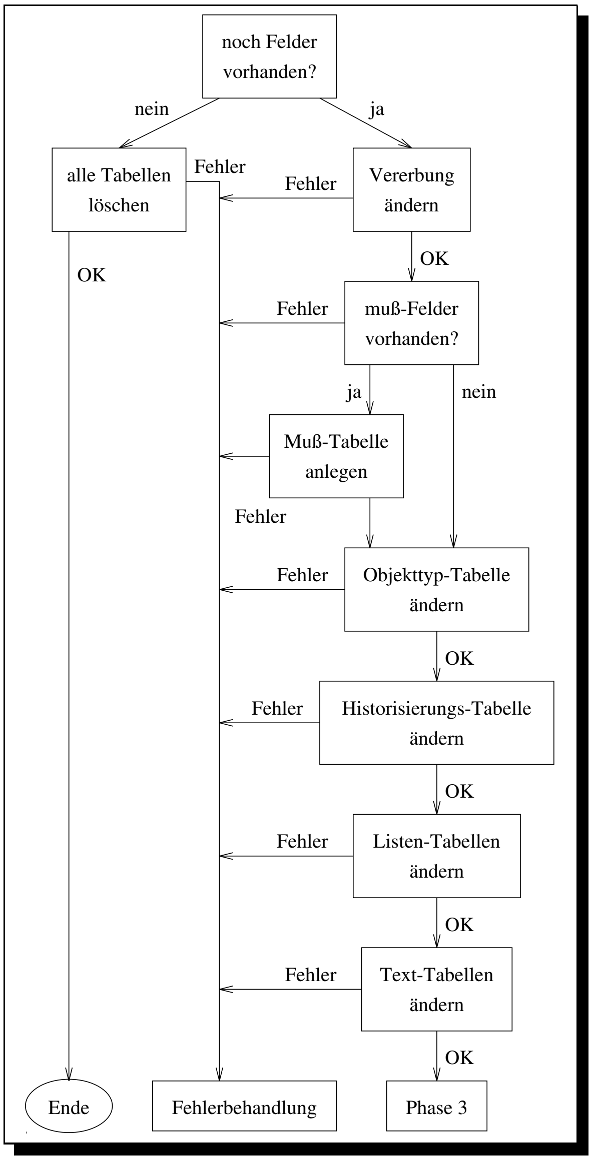


Abbildung 5.15: Generierung geänderter Objekttypen: Phase 2

Objekttyp–Tabelle sind die neuen muß–Felder schon als *not null* gekennzeichnet. Wenn es also noch alte Objekte mit leeren Einträgen in diesen Feldern in der Hilfstabelle gibt, weist die Datenbank das Kopieren dieser Felder zurück. Es ist also nicht möglich, erst einmal die alten Objekte zu kopieren und dann die Dummy–Werte nachzutragen. In der Muß–Tabelle kann genau dies getan werden, da dort die neuen muß–Felder noch nicht gekennzeichnet

sind. Die Hilfstabelle darf ihrerseits nicht mit Dummy–Werten gefüllt werden, da sie möglicherweise noch von der Fehlerbehandlung benutzt wird, um die alte Tabelle wieder herzustellen. Sie darf also nicht verändert werden.
Als nächstes wird die Historisierungs–Tabelle gelöscht und neu generiert, sofern die Historisierungs–Liste Felder enthält. Wenn auch die Hist_Insert–Liste gefüllt ist, werden die alten Objekte aus der Hilfstabelle in die neue Historisierungs–Tabelle kopiert. Wenn die Historisierungs–Liste keine Felder enthält, wird an dieser Stelle zusätzlich getestet, ob es eine alte Historisierungs–Tabelle gibt. Wenn eine gefunden wird, wird sie gelöscht. Die Tabellen für die Felder vom Typ Liste und Text werden je nach Umstellung so behandelt, wie oben beschrieben. Zuletzt wird noch der eindeutige Index auf die Schlüsselfelder der Objekttyp–Tabelle gelegt.

Phase 3: Statustabellen aktualisieren. Die relationale Repräsentation des Objekttypen ist nun auf dem neuen Stand, der aus den Systemtabellen gelesen wurde. Jetzt müssen noch die Statustabellen aktualisiert werden. Dazu werden alle Informationen über den Objekttypen aus den Statustabellen gelöscht. Anschließend werden die entsprechenden Informationen aus den Systemtabellen in die Statustabellen kopiert. Tritt während dieser dritten Phase ein Fehler auf, wird die Generierung abgebrochen und die Fehlerbehandlung aufgerufen. Andernfalls stimmen die graphische und die relationale Repräsentation des Objekttypen wieder überein.

Phase 4: Funktion beenden. Die vierte Phase (Abbildung 5.16) soll die Generierung des Objekttypen beenden. Zuerst muß aber noch die Datenbank aufgeräumt werden. Alle Hilfstabellen werden gelöscht, da sie ja nicht mehr benötigt werden. Allerdings werden dabei, wie oben schon erwähnt, alle Datenbankfehler ignoriert. Wenn also eine dieser Tabellen aus irgendeinem Grund nicht entfernt werden kann, bleibt sie in der Objektebank stehen. Sie wird dann bei der nächsten Generierung des Objekttypen automatisch gelöscht. Da nun auch keine Fehlerbehandlung mehr erforderlich ist, werden alle Einträge des Objekttypen aus der Clear–Tabelle gelöscht. Damit ist die erneute Generierung des geänderten Objekttypen abgeschlossen. Die Datenbank befindet sich wieder in einem konsistenten Zustand. Um diesen Zustand entgültig zu bestätigen, wird zuletzt noch ein *Commit* in der Datenbank abgesetzt. Wenn bei einer dieser beiden letzten Operationen ein Fehler auftritt, wird die aufrufende Komponente, der Datenmodell–Editor, über den inkonsistenten Abbruch der Funktion informiert. Denn dann enthält die Clear–Tabelle noch immer die Informationen des behandelten Objekttypen, obwohl seine Generierung korrekt abgeschlossen wurde.

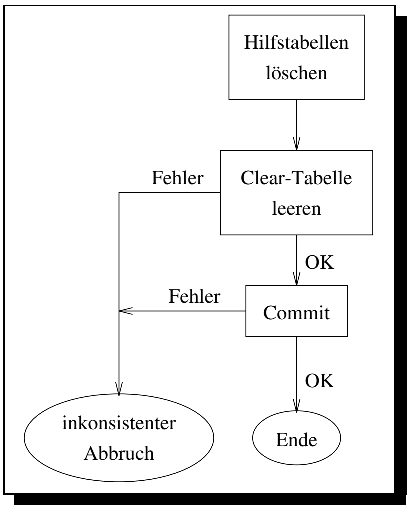


Abbildung 5.16: Generierung geänderter Objekttypen: Phase 4

Diese Inkonsistenz wird behoben, indem der Editor seinerseits die Fehlerbehandung aufruft.

5.7.7 Zurücksetzen geänderter Objekttypen

Die Routine zum Zurücksetzen geänderter Objekttypen (Abbildung 5.17) wird nur dann aufgerufen, wenn der Test zuvor ergeben hat, daß der Objekttyp alt ist und geändert wurde, und wenn der Benutzer entschieden hat, die Änderungen wieder zu verwerfen. Alle Änderungen, die seit der letzten Generierung des Objekttypen vorgenommen wurden, sind in den Systemtabellen abgelegt worden. Sie beschreiben seine aktuelle Darstellung. Die Repräsentation des Objekttypen auf relationaler Ebene in der Datenbank wird in den Statustabellen festgehalten. Diese Funktion soll nun die graphische Repräsentation auf den Stand der relationalen Darstellung zurücksetzen.
Zunächst muß festgestellt werden, ob der Objekttyp überhaupt zurückgesetzt werden darf. Vier Fälle müssen getestet werden:

1. Der Objekttyp A enthält ein Feld vom Typ *Referenz auf einen Objekttypen B*. Der Typ des Feldes wird auf einen beliebigen anderen Typen geändert. Dann wird der Objekttyp B, auf den die Referenz verweist, gelöscht. Nun soll der Objekttyp A zurückgesetzt werden. Dadurch würde aber das Feld wieder den Objekttypen B referenzieren. Da dieser gelöscht wurde, entstünde eine Inkonsistenz. Es darf also

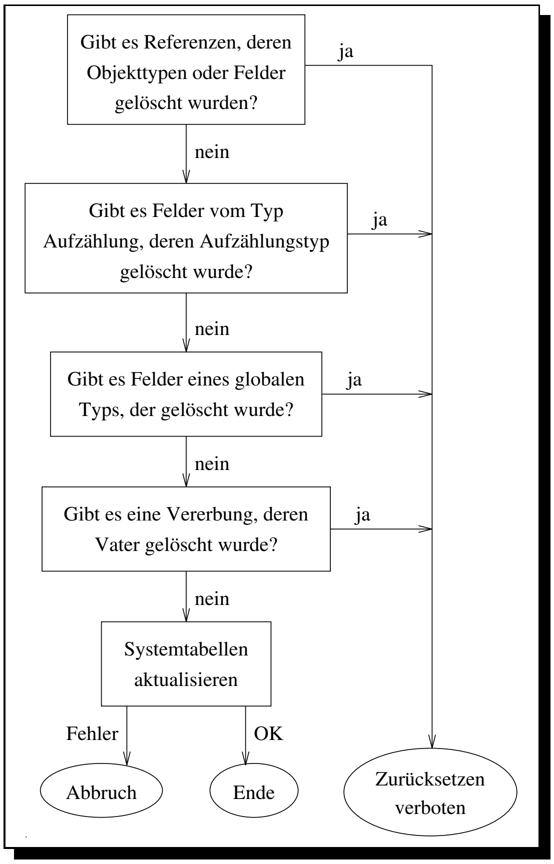


Abbildung 5.17: Zurücksetzen von Objekttypen

nicht zurückgesetzt werden.

2. Der Objekttyp enthält ein Feld vom Typ *Aufzählung*. Der Typ des Feldes wird auf einen beliebigen anderen Typen geändert. Dann wird der Aufzählungstyp gelöscht. Analog zum vorigen Fall ist auch hier ein Zurücksetzen nicht möglich.

3. Der Objekttyp enthält ein Feld vom Typ eines globalen Feldtypen. Der Typ des Feldes wird auf einen beliebigen anderen Typen geändert. Dann wird der globale Feldtyp gelöscht. Analog zum vorigen Fall ist auch hier ein Zurücksetzen nicht möglich.

4. Der Objekttyp A ist ein Sohn des Objekttypen B. Die Vererbung wird gelöst und der Objekttyp B wird gelöscht. Durch das Zurücksetzen des Objekttypen A würde eine Vererbung wieder hergestellt, deren Vater nicht mehr existiert. Auch hier ist ein Zurücksetzen demnach nicht möglich.

Wenn keiner der vier Fälle eingetreten ist, kann der Objekttyp zurückgesetzt werden. Dies geschieht, indem zunächst alle Informationen des Objekttypen aus den Systemtabellen gelöscht werden. Anschließend werden seine Informationen aus den Statustabellen in die Systemtabellen kopiert. Wenn dabei an irgendeiner Stelle ein Fehler auftritt, bricht die Funktion sofort ab und informiert die aufrufende Instanz, den Datenmodell–Editor. Das Transaktionskonzept von LEU sieht vor, daß jedes Werkzeug, das einen Fehler von einer Datenbankfunktion bekommt, die laufende Transaktion mittels *Rollback* zurücksetzt. Die Generierung braucht also für diese Funktion keine eigene Fehlerbehandlung. Sie verwendet nur Datenbankoperationen zum Löschen und Einfügen von Objekten, und diese Operationen werden vom Editor aus durch das *Rollback* zurückgesetzt.

5.7.8 Fehlerbehandlung

Die Fehlerbehandlung (Abbildung 5.18) übernimmt das Zurücksetzen aller Manipulationen, die die aufrufende Funktion in der Datenbank vorgenommen hat. Sie realisiert damit das *Rollback* des Transaktionskonzeptes der Generierung (vgl. Kapitel 5.6). Die Fehlerbehandlung läuft in zwei Stufen ab. Die **erste Stufe** wird entweder aus der Generierung neuer Objekttypen oder aus der Generierung geänderter Objekttypen aufgerufen. Sie liest alle Informationen zu dem behandelten Objekttypen aus der Clear–Tabelle. Es gibt drei Arten von Einträgen:

- den Typ "Drop" in Kombination mit dem Namen einer Originaltabelle,
- den Typ "Drop" in Kombination mit dem Namen einer Hilfstabelle und
- den Typ "Create" in Kombination mit beiden Tabellennamen.

Zuerst werden alle Einträge behandelt, die eine Originaltabelle und den Typ "Drop" aufweisen. Damit werden die Applikationstabellen gelöscht. Wenn der Aufruf von der Generierung neuer Objekttypen kam, ist jetzt der alte Zustand schon wieder hergestellt. Kam der Aufruf von der Generierung geänderter Objekttypen, so sind die neu erzeugten und die geänderten Tabellen nun wieder gelöscht worden. Die geänderten Tabellen müssen aber in ihrem alten Zustand wieder aufgebaut werden. Deshalb werden als nächstes die Einträge vom Typ "Create" behandelt. Die alten Tabellen werden wieder angelegt, indem die Hilfstabellen, die ja den alten Zustand gespeichert haben, kopiert werden. Anschließend werden die Hilfstabellen nicht mehr benötigt. Sie werden durch die Abarbeitung der restlichen Einträge vom Typ "Drop" gelöscht. Zuletzt werden alle Einträge zu dem behandelten

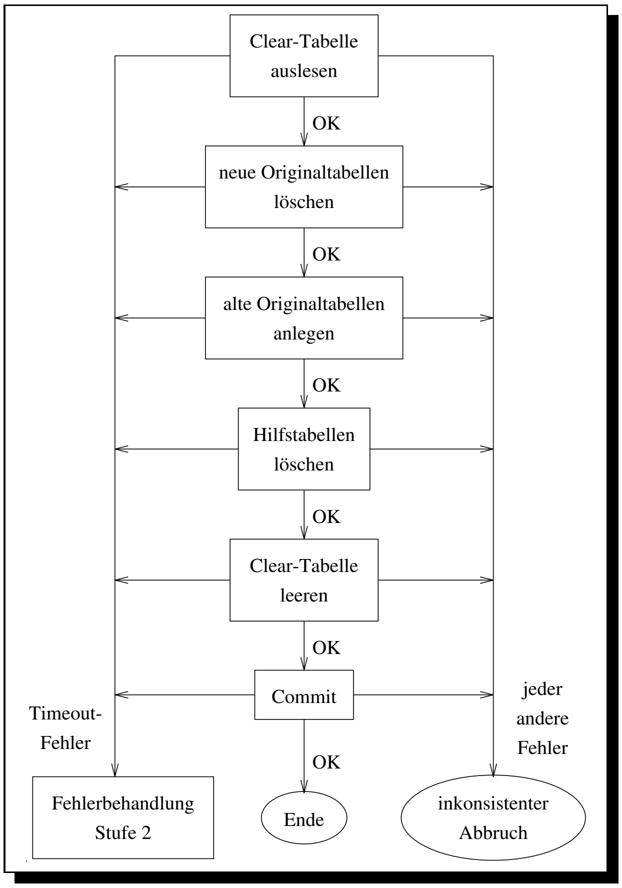


Abbildung 5.18: Fehlerbehandlung

Objekttypen wieder aus der Clear–Tabelle gelöscht. Da damit wieder ein konsistenter Datenbankzustand erreicht ist, wird die laufende Transaktion durch ein *Commit* beendet. Natürlich kann auch während dieser ersten Stufe der Fehlerbehandlung selbst ein Fehler auftreten. Dann wird die Funktion sofort abgebrochen. Je nach Fehler werden zwei verschiedene Meldungen an die aufrufende Komponente, den Datenmodell–Editor, weitergegeben. Wenn die Datenbank den Fehler **Timeout** meldet, wird dies auch dem Editor mitgeteilt. Ein Timeout bedeutet, daß ein anderer Benutzer die zu bearbeitende Tabelle gesperrt hat. Dies geschieht zum Beispiel, wenn die Fehlerbehandlung eine Tabelle löschen möchte, die ein anderer LEU–Prozeß gerade liest. Bei allen anderen Fehlern meldet die Fehlerbehandlung dem Editor, daß die Datenbank inkonsistent ist.

Wenn die erste Stufe der Fehlerbehandlung fehlerfrei lief, ist die Datenbank in einem konsistenten Zustand. Der zuletzt bearbeitete Objekttyp ist auf den alten Stand zurückgesetzt worden. Dennoch bekommt der Benutzer vom Editor die Meldung, daß die Generierung des entsprechenden Objekttypen fehlgeschlagen ist. Eine korrekte Fehlerbehandlung bedeutet ja nur, daß der Fehler in der Generierung keine Auswirkungen auf die Konsistenz der Datenbank hatte.
Wird die erste Stufe mit einer inkonsistenten Datenbank beendet, so informiert der Editor den Benutzer. Da die Fehlerbehandlung noch nicht abgeschlossen war, enthält die Clear–Tabelle immer noch alle Informationen, um einen konsistenten Zustand wieder herzustellen. Der Benutzer kann also den Datenbankadministrator benachrichtigen, so daß dieser die nötigen Operationen von Hand ausführt.
Die **zweite Stufe** der Fehlerbehandlung beginnt, wenn in der ersten Stufe ein Timeout aufgetreten ist. Der Editor informiert zuerst den Benutzer, daß er auf den Zugriff auf die Datenbank warten muß. Dann stößt er eine zweite Version der Fehlerbehandlung an. Diese arbeitet im Prinzip genauso wie die die erste, sie reagiert nur anders auf den Timeout: Wenn die Datenbank einen Timeout meldet, wird die Aktion, die diesen Fehler hervorgerufen hat, wieder ausgeführt. Die geschieht so lange, bis entweder ein anderer oder gar kein Fehler von der Datenbank gemeldet wird. Wenn diese zweite Stufe der Fehlerbehandlung fehlerfrei endet, befindet sich die Datenbank wiederum in einem alten konsistenten Zustand. Dem Benutzer wird mitgeteilt, daß die Generierung des letzten Objekttypen fehlgeschlagen ist. Tritt ein anderer Fehler auf, so wird ein inkonsistenter Datenbankzustand gemeldet, den der Administrator von Hand beseitigen muß.

5.8 Entwurf der Generierung

Die Abbildung 5.19 zeigt die Modulhierarchie der Generierung. Im folgenden werden die Funktionen der einzelnen Module beschrieben:

Die Importschnittstelle. Diese Schnittstelle besteht aus den LEU–Modulen, die von der Generierung benutzt werden. Jeder Kasten steht für eine ganze Hierarchie von Modulen, die unter einem Bezeichner zusammengefaßt werden. **DSB** verkapselt die Datenstrukturen und die abstrakten Datentypen von LEU. **MEM** steuert die Speicherverwaltung von LEU. **DB** verkapselt die Datenbank und enthält damit sämtliche Funktionen, die Zugriff auf die Datenbank haben.

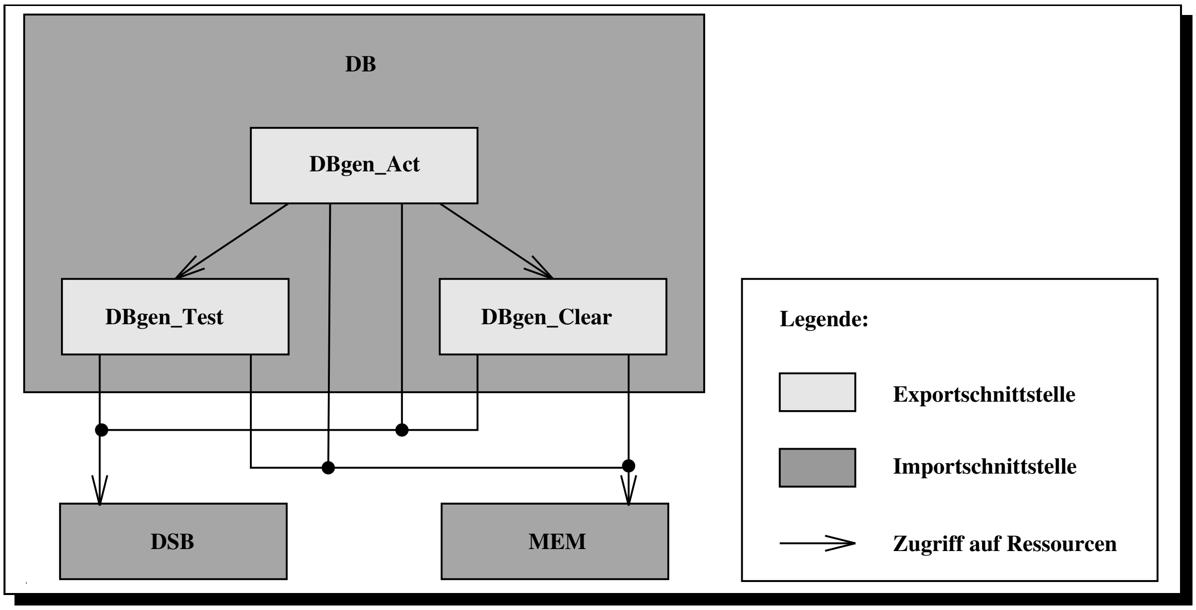


Abbildung 5.19: Modulhierarchie der Generierung

Die Exportschnittstelle. Wie schon im Kapitel 5.7 erläutert, ist die Generierung ein Bestandteil der Datenbank–Komponente von LEU. Deshalb befinden sich alle ihre Module innerhalb von DB. **DBgen_Act** realisiert die eigentliche Generierung. Dies Modul beinhaltet alle Funktionen, die zum Anlegen, Ändern und Löschen von Objekttypen und Verbindungen in der Datenbank benötigt werden (vgl. Kapitel 5.7.2 bis 5.7.7). **DBgen_Test** enthält Funktionen, die die Konsistenz bei der Schemaevolution sicherstellen. Sie werden bei Änderungen des Datenmodells vom Editor aufgerufen um zu prüfen, ob diese Änderungen erlaubt sind (vgl. Kapitel 5.5). Außerdem werden sie während der Generierung von DBgen_Act benötigt, um die tatsächlich vorgenommenen Änderungen festzustellen. **DBgen_Clear** realisiert die Fehlerbehandlung (vgl. Kapitel 5.7.8). Es enthält Funktionen für beide Stufen der Fehlerbehandlung und wird daher sowohl von DBgen_Act als auch vom Datenmodell–Editor aufgerufen.

5.9 Ein Beispiel

In diesem Abschnitt wird der Ablauf der Generierung anhand eines kleinen Beispielmodells beschrieben. Dies Modell erhebt keinerlei Anspruch auf Vollständigkeit. Zu Beginn werden im Datenmodell– und Feld–Editor zwei Objekttypen *Regal (Nummer, Inhalt)* und *Buch*

(Schlüssel, Titel, Autoren) und eine Verbindung *Standort*, die die Objekttypen verbindet, angelegt:

Die Objekttypen, ihre Felder und die Verbindung werden in die Systemtabellen eingetragen. Dabei erhalten sie automatisch je ein eindeutiges Surrogat zugeordnet. Die folgenden Tabellen zeigen Ausschnitte aus den Systemtabellen. Dabei wird der Feldtyp stark vereinfacht durch die Attribute "Typ" und "Liste" in der Field–Tabelle dargestellt. Die Statustabellen sind noch leer.

Objecttype		
Surrogate	Ident	Hist_Flag
100	Regal	true
200	Buch	false

Field							
Surrogate	Refsurr	Ident	Typ	Liste	Res	IsFieldKey	Hist
110	100	Nummer	String(1)	false	muß	false	false
111	100	Inhalt	String(30)	false	kann	false	true
202	200	Schlüssel	Integer(7)	false	muß	true	false
203	200	Titel	String(70)	false	kann	false	false
204	200	Autoren	String(30)	true	kann	false	false

Connection							
Surrogate	Ident	Source	Target	SRes	TRes	SDig	TDig
300	Standort	200	100	muß	muß	n	1

Erste Generierung. Die Generierung wird mit diesem neuen Modell zum ersten Mal angestoßen. Die Funktion zum Löschen alter Tabellen (Kapitel 5.7.2) findet keine Einträge für dies Modell in der Drop–Tabelle. Ihre Arbeit ist damit schon beendet. Dann wird die

Generierung der Verbindungen (Kapitel 5.7.3) aufgerufen. Sie erzeugt die Tabelle für die Verbindung *Standort*:

```
LEU_RE_300 (OT_200: CHAR(40),
            OT_100: CHAR(40))
```

Jetzt beginnt die Generierung der Objekttypen. Zuerst wird der Status des *Regals* bestimmt (Kapitel 5.7.4). Diese Routine findet keine Einträge zum *Regal* in den Statustabellen (Kapitel 5.7.1) und liefert als Ergebnis, daß der Objekttyp neu ist. Daraufhin wird die Generierung neuer Objekttypen (Kapitel 5.7.5) aufgerufen. Sie generiert eine Objekttyp– und eine Historisierungs–Tabelle. Beide Tabellen erhalten das für den Benutzer unsichtbare Surrogat. Es ist ein muß–Feld, da es als Primärschlüssel für die Applikationsdaten verwandt wird. Das Feld *Bezeichner* wurde vom Benutzer als muß–Feld definiert. Die Historisierungs–Tabelle enthält nur das Surrogat, den Zeitpunkt und das historisierte Feld.

```
LEU_OT_100 (Surrogate: CHAR(40) NOT NULL,
            F_110: CHAR(1) NOT NULL,
            F_111: CHAR(30))

LEU_OTH_100 (Surrogate: CHAR(40) NOT NULL,
             Date: NUMBER(12),
             F_111: CHAR(30))
```

Damit ist die Generierung von *Regal* abgeschlossen. Der Status des zweiten Objekttypen *Buch* wird bestimmt. Er ist ebenfalls neu, so daß die Generierung neuer Objekttypen noch einmal angestoßen wird. Sie erzeugt eine Objekttyp– und eine Listen–Tabelle. Das Feld *Schlüssel* wurde als Schlüsselfeld definiert. Deshalb wird noch ein eindeutiger Index auf dieses Feld gelegt.

```
LEU_OT_200 (Surrogate: CHAR(40) NOT NULL,
            F_202: NUMBER(7) NOT NULL,
            F_203: CHAR(70),
            F_204: CHAR(30))

LEU_LI_204 (Refsurr: CHAR(40),
            Seqno: NUMBER(6),
            Element: CHAR(30))
```

Die Generierung des Datenmodells ist abgeschlossen. Die Statustabellen enthalten zu diesem Zeitpunkt dieselben Informationen wie die Systemtabellen (Kapitel 5.3).

Datenerfassung. Der Anwender erfaßt nun die ersten Objekte. Zur besseren Lesbarkeit werden die Namen der Objekttypen und Felder mit in die Tabellen geschrieben.

LEU_OT_100 (Regal)		
Surrogate	Nummer F_110	Inhalt F_111
804	k	Krimi
805	p	
806	h	Horror
807	s	Science–Fiction

LEU_RE_300 (Standort)	
Buch OT_200	Regal OT_100
901	806
902	804
903	804

LEU_OT_200 (Buch)			
Surrogate	Schlüssel F_202	Titel F_203	Autoren F_204
901	1830765	Es	King
902	5739254	Der Rosenmord	Peters
903	9973622	Alibi	Christie
904	3627291	Metamorphose	

LEU_LI_204 (Autoren)		
Refsurr	Seqno	Element
901	1	King
902	1	Peters
903	1	Christie

Schemaevolution. Nun nimmt der Benutzer einige Änderungen am Modell vor. Für das Buch führt er ein neues Feld *Beschreibung* vom Typ Text ein. Die Regale sollen ab sofort nicht mehr mit Buchstaben sondern mit Zahlen bezeichnet werden. Deshalb ändert er den Typ des Feldes *Nummer* in Integer(3). Da aber die Buchstaben aus den Feldern nicht in Zahlern konvertiert werden können, warnt der Feld–Editor vor Datenverlust. Der Benutzer bestätigt die Änderung dennoch. Zuletzt sollen die Bücher als Schlüssel nicht mehr lange Zahlen sondern kürzere Buchstaben–Zahlen–Kombinationen bekommen. Der Typ von *Schlüssel* wird also von Integer(7) in String(5) geändert. Dies würde aber einen Datenverlust bedeuten, da die alten Werte wegen der Verkürzung des Feldtyps nicht übernommen werden können. Da *Schlüssel* aber als Schlüsselfeld definiert wurde, ist diese Änderung verboten (Kapitel 5.5). Der Benutzer entscheidet sich daraufhin für eine Umstellung auf String(7). Dies ist erlaubt, da die Werte problemlos von Integer in einen String gleicher Länge übernommen werden können.

Zweite Generierung. Die Generierung des Modells wird zum zweiten Mal angestoßen. Es sind immer noch keine Tabellen zu löschen. Die Generierung für Verbindungen (Kapitel 5.7.3) versucht, die *Standort*–Verbindung anzulegen. Da sie aber schon existiert, ändert sich hier nichts. Dann wird der Status des Objekttypen *Regal* getestet (Kapitel 5.7.4). Da ein Feld geändert wurde, wird die Generierung für geänderte Objekttypen (Kapitel 5.7.6) aufgerufen. Sie ändert die Objekttyp–Tabelle entsprechend. Da die Umstellung des Feldtyps von *Nummer* Datenverlust erzeugt und es sich um ein muß–Feld handelt, werden alle Einträge mit Dummy–Werten (Kapitel 5.5) aufgefüllt. Die *Regal*–Tabelle sieht nun so aus:

```
LEU_OT_100 (Surrogate: CHAR(40) NOT NULL,
            F_110: NUMBER(3) NOT NULL,
            F_111: CHAR(30))
```

LEU_OT_100 (Regal)		
Surrogate	Nummer F_110	Inhalt F_111
804	0	Krimi
805	0	
806	0	Horror
807	0	Science–Fiction

Nun wird der Status des Objekttypen *Buch* bestimmt. Es wird ein geändertes und ein neues Feld gefunden, so daß wiederum die Generierung für geänderte Objekttypen aufgerufen wird. Sie ändert den Typ des Feldes *Schlüssel* und fügt ein neues Feld für die *Beschreibung* in die Tabelle ein. Da es sich dabei um ein Text–Feld handelt, wird außerdem eine Text–Tabelle erzeugt.

```
LEU_OT_200 (Surrogate: CHAR(40) NOT NULL,
            F_202: CHAR(7) NOT NULL,
            F_203: CHAR(70),
            F_204: CHAR(30),
            F_205: CHAR(255))

LEU_TE_205 (Refsurr: CHAR(40),
            Seqno: NUMBER(4),
```

Length: NUMBER(7),
Content: LONG)

LEU_OT_200 (Buch)				
Surrogate	Schlüssel F_202	Titel F_203	Autoren F_204	Beschreibung F_205
901	1830765	Es	King	
902	5739254	Der Rosenmord	Peters	
903	9973622	Alibi	Christie	
904	3627291	Metamorphose		

Damit ist die zweite Generierung des Modells beendet. Der Benutzer kann nun weitere Objekte erfassen und die Dummy–Werte durch richtige Werte ersetzen. Außerdem kann er das Modell beliebig oft ändern und neu generieren.

Kapitel 6

Die Optimierung

Dies Kapitel stellt die Optimierung vor. Im Abschnitt 6.1 werden zunächst die Anforderungen an die Optimierung spezifiziert. Die Rahmenbedingungen für die konzeptionelle Arbeit beschreibt Kapitel 6.2.
Anschließend werden Konzepte für Teilprobleme der Optimierung vorgestellt. Abschnitt 6.3 behandelt die Normalisierung. Die Zugriffspfade werden in 6.4 erläutert. Kapitel 6.5 diskutiert Maßnahmen zur Leistungsverbesserung und ihre Einsatzmöglichkeiten innerhalb der Optimierung.
Im Kapitel 6.6 wird das Konzept für die Optimierung vorgestellt. Der Entwurf folgt in 6.7. Zuletzt wird in 6.8 der Ablauf der Optimierung anhand eines Beispiels erläutert.

6.1 Anforderungen

Die Optimierung hat zwei Ziele. Zum einen prüft sie das Schema des LEU–ER–Modells auf Vollständigkeit. Zum anderen versucht sie, die Performance für den späteren Zugriff auf die Tabellen des Modells zu verbessern. Die folgende Liste spezifiziert die Anforderungen an die Optimierung:

- Das LEU–ER–Modell muß auf eine vollständige Definition hin überprüft werden.
- Das LEU–ER–Modell muß, soweit möglich, auf eine semantisch korrekte Definition hin überprüft werden.
- Das LEU–ER–Modell muß dahingehend überprüft werden, ob die Anfragen und Manipulationen performant ausgeführt werden können.

- Die Optimierung muß iterativ und interaktiv benutzbar sein. Das heißt, sie soll zum einen für ein Datenmodell beliebig oft aufrufbar sein, zum anderen sollen einzelne Teile der Optimierung getrennt aufgerufen werden können.
- Die Ergebnisse müssen dem Benutzer dargestellt werden.

6.2 Rahmenbedingungen

Da die Optimierung im Rahmen von LEU entwickelt wird, existieren verschiedene Rahmenbedingungen, die bei der Entwicklung berücksichtigt werden müssen:

ANSI–SQL: Alle Informationen, die bei der Entwicklung von LEU und von LEU–Applikationen anfallen, werden in einer Datenbank abgelegt. Zur Kommunikation mit dieser Datenbank wird *Embedded SQL* (SQL zur Einbettung in C–Code) benutzt [ORA90b]. Es soll möglich sein, LEU auf beliebige Datenbanken aufzusetzen. Daher darf bei der Implementierung nur SQL nach dem ANSI–Standard [Ame92] benutzt werden. Erweiterungen, die spezielle Datenbanken bieten, sind nicht verwendbar, da sie unter Umständen nicht auf andere Datenbanken portierbar sind.

Keine selbständige Veränderung des Modells: Alle Mängel, die die Optimierung an einem Datenmodell feststellt, müssen an den Benutzer in Form von Vorschlägen und Warnungen weitergegeben werden. Das Modell darf nicht von der Optimierung selbst verändert werden. Der Benutzer soll die volle Kontrolle über den Entwurf seines Schemas besitzen.

Möglichst früh und schnell: Für den praktischen Einsatz in LEU ist es wichtig, daß der Benutzer nicht zu viele zusätzliche Angaben (wie Testdaten, Tabellengrößen oder Anfragen) machen muß, und daß er nach dem Aufruf der Optimierung nicht zu lange auf das Ergebnis wartet. Die Optimierung soll direkt nach der Erstellung eines LEU–ER–Modells aufgerufen werden. Zu diesem Zeitpunkt steht nur das Schema zur Verfügung. Die zusätzliche Erfassung von Zugriffspfaden ist möglich. Mit diesen Informationen sollte die Optimierung auskommen.

6.3 Normalisierung

Im folgenden wird gezeigt, daß eine Normalisierung während der Optimierung nicht erforderlich ist, da sich das generierte relationale Datenmodell im wesentlichen schon in der

dritten Normalform befindet. (Eine Erläuterung der Normalformen steht in Kapitel 3.4.)
Das Schema, das den folgenden Ausführungen zugrunde liegt, ist das Relationale Datenmodell, das von der Generierung aus dem LEU–ER–Modell erzeugt wurde. Die einzige Integritätsbedingung, die ein LEU–ER–Modell enthält, ist der Schlüssel eines Objekttypen. Weitere Integritätsbedingungen, insbesondere funktionale Abhängigkeiten, können für das LEU–ER–Modell nicht definiert werden und müssen daher auch nicht berücksichtigt werden.
Die Generierung verletzt bei ihrer Umsetzung des LEU–ER–Modells nur an einer Stelle die dritte Normalform: bei Feldern vom Typ Liste oder Text werden redundante Daten gespeichert, und zwar das erste Listenelement und die ersten 255 Zeichen des Textes (vgl. Abschnitt 5.4). Diese Redundanz wird toleriert, da die Normalisierung nur eins von mehreren Kriterien für die Qualität eines ER–Modells ist. Ein anderes Kriterium ist die Performance. Und aus Performancegründen ist an dieser Stelle eine redundante Datenhaltung sinnvoll.
Die erste Normalform fordert atomare Wertebereiche für alle Felder. Im Kapitel 5.4 wurde beschrieben, wie die LEU–Datentypen in die Typen der relationalen Datenbank umgesetzt werden. Die meisten Datentypen wurden auf Zahlen (Number) oder Zeichenketten (Char) abgebildet und erfüllen damit die erste Normalform. Lediglich die Typen *Text* und *Liste* sind nicht atomar. Sie werden durch ein atomares Feld innerhalb der zugehörigen Objekttyp–Tabelle und durch eine eigene Tabelle dargestellt. Diese Tabelle enthält wiederum nur Felder mit atomaren Typen. Auch die Text– und Listen–Felder genügen also der ersten Normalform.
Die zweite Normalform setzt die erste voraus und verlangt, daß jedes Nichtschlüsselfelder von jedem Schlüssel voll funktional abhängt. Es gibt, je nach Betrachtung, zwei verschiedene Arten von Schlüsseln im LEU–ER–Modell. Einerseits definiert der Benutzer für jeden Objekttypen einen Schlüssel. Andererseits fügt die Generierung jedem Objekttypen ein Surrogat hinzu, das innerhalb der Datenbank als Schlüssel fungiert. Ein Objekttyp besitzt also aus der Sicht des Benutzers einen anderen Schlüssel als aus der Sicht der Datenbank. Es kann aber gezeigt werden, daß die Art der Betrachtung keine Rolle spielt, da die zweite Normalform in jedem Fall erfüllt ist. Das Surrogat besteht aus genau einem Feld. In diesem Fall ist der Schlüssel also atomar. Es ist demnach nicht möglich, daß ein anderes Feld nur von einem Teil des Schlüssels abhängt, da dieser Schlüssel nicht teilbar ist. Der benutzerdefinierte Schlüssel kann aus mehreren Feldern bestehen. Es ist im Datenmodell–Editor aber nicht möglich, funktionale Abhängigkeiten explizit zu definieren.

Der Benutzer muß also bei der Modellierung davon ausgehen, daß alle Nichtschlüsselfelder von dem einen definierten Schlüssel voll funktional abhängen. Die zweite Normalform ist somit erfüllt.
Die dritte Normalform fordert zusätzlich zur zweiten, daß kein Nichtschlüsselfeld von einem Schlüssel transitiv abhängt. Eine transitive Abhängigkeit kann in einem Modell nur vorkommen, wenn mehrere funktionale Abhängigkeiten modelliert werden können. Wie oben ausgeführt, kann im Datenmodell–Editor aber nur eine einzige Art funktionaler Abhängigkeit modelliert werden: ein Schlüssel wird definiert und alle anderen Felder hängen vom gesamten Schlüssel ab. Ebenso existiert auf der relationalen Ebene nur die Abhängigkeit aller Felder vom Surrogat. Eine transitive Abhängigkeit ist nicht modellierbar, so daß die dritte Normalform erfüllt ist.

6.4 Zugriffspfade

Wie performant ein Datenmodell ist, zeigt sich in der Geschwindigkeit, mit der Anfragen und Manipulationen bearbeitet werden. Die Optimierung sollte demnach wissen, welche Anfragen an das zu verbessernde LEU–ER–Modell gestellt werden. Zum Zeitpunkt der Modellierung sind aber noch keine Anfragen erfaßt. Dennoch kann man davon ausgehen, daß der Modellierer beim Entwurf seines Schemas weiß, wie die späteren Anfragen an sein Modell aussehen werden. Denn bei der Datenmodellierung wird ja ein Ausschnitt aus der realen Welt nachgebildet. Dieser Ausschnitt umfaßt nicht nur Objekte und ihre Verbindungen, sondern auch ihre Beobachtung und Veränderung. Der Benutzer ist also in der Lage, die Anfragen, die er für sein Modell erwartet, auf logischer Ebene als Zugriffspfade anzugeben.
Ein Zugriffspfad besteht hier aus drei Informationen:

1. einer Liste aller Felder, die gesucht oder manipuliert werden,

2. einer Liste aller Felder, an die Suchbedingungen geknüpft sind und

3. einer Zahl, die angibt, wie oft dieser Pfad pro Zeiteinheit benutzt wird.

Welche konkrete Zeiteinheit gewählt wird, ist dabei nicht relevant. Sie dient nur als Maß für die Häufigkeit, in der die einzelnen Pfade benutzt werden. Je öfter ein Pfad benutzt wird (also je höher seine Zahl ist), desto mehr Zeit wird verbraucht, wenn er nicht performant bearbeitet werden kann. Deshalb wird die Optimierung die Pfade, die am häufigsten benutzt werden, besonders berücksichtigen.

Eine mögliche Anfrage an eine Bibliotheks–Datenbank ist:

"Welchem Leser hat der Bibliothekar X das Buch Y geliehen?"

In Form eines Zugriffspfades sieht diese Anfrage wie folgt aus:

1. gesuchte Felder: Kundennummer des Lesers,
2. Suchbedingungen: der Name des Bibliothekars ist X und der Titel des Buches ist Y,
3. Anzahl der Aufrufe pro Zeiteinheit: 18.

6.5 Leistungsverbesserungen

Es gibt verschiedene Möglichkeiten, den Zugriff auf Datenbank–Tabellen performanter zu gestalten. Im folgenden werden verschiedene Ansätze aus der Literatur vorgestellt [FH89, RM90, ORA90b, ORA90c]. Es wird diskutiert, ob die entsprechenden Maßnahmen innerhalb der Optimierung angewandt werden können.

Prototyping. Die Prototyp–Generierung [BKK+92] ist eine Methode, die von Werkzeugen für den konzeptionellen Datenbankentwurf genutzt wird. (CADDY benutzt Prototyping zum Beispiel für die Überprüfung des erstellten EER–Modells, wie in 4.2 beschrieben wurde.) Dabei wird die konzeptionelle Spezifikation in eine Prototyp–Implementierung umgesetzt. Mit Hilfe von Testdaten werden Experimente durchgeführt, die Aufschluß über die Funktionalität geben. Aufgrund der Ergebnisse kann die Spezifikation verbessert werden. Diese Methode ist aber relativ zeitaufwendig. Für ihren Einsatz in der Optimierung müßte zunächst das LEU–ER–Modell generiert und mit Testdaten gefüllt werden. Dann müßten konkrete Anfragen spezifiziert werden, mit denen die Performance des Modells getestet wird. Schließlich müßten die zum Test generierten Tabellen, Daten und Anfragen wieder gelöscht werden. Deshalb wird bei der Optimierung darauf verzichtet. Sie stützt sich nur auf das LEU–ER–Modell und die Zugriffspfade.

Indizes. Ein Index ist ein dynamischer Mechanismus um den Zugriff auf Datenbanktabellen zu beschleunigen. Er wird für eine Spalte oder eine Spaltenkombination einer Tabelle definiert. Wenn eine Anfrage eine solche Spalte benutzt, muß nicht die ganze Tabelle nach den passenden Zeilen durchsucht werden. Vielmehr wird über den Index direkt

auf die richtigen Zeilen zugegriffen. Operationen, die nur etwa zehn bis fünfzehn Prozent der Zeilen einer Tabelle abfragen, können so erheblich beschleunigt werden.
Es gibt aber auch Probleme beim Einsatz von Indizes. Nur die Anfragen profitieren von ihnen. Die Operationen zum Einfügen und Löschen werden mit zunehmender Anzahl von Indizes immer langsamer, da sie nicht nur die eigentliche Tabelle sondern auch die Indizes aktualisieren müssen. Deshalb sollten Tabellen, die oft geändert werden, nicht zu viele Indizes haben. Außerdem lohnt sich ein Index nicht für kleine Tabellen. Die Suche durch die Tabelle dauert unter Umständen ebenso lange wie über den Index.
Ob es sinnvoll ist, eine Tabelle mit einem Index zu versehen, stellt sich immer erst während ihres Gebrauches heraus. Üblicherweise überwacht ein Datenbankadministrator die Operationen, die auf einer Tabelle ausgeführt werden. Dann entscheidet er, wo ein Index nötig ist. Wenn sich im Laufe der Zeit die Operationen ändern, kann er alte Indizes löschen und neue anlegen, die den aktuellen Anforderungen genügen.
Es ist also sinnvoll, über den Einsatz von Indizes erst im laufenden Betrieb zu entscheiden. Für die Optimierung kommt deshalb der Einsatz von Indizes zur Leistungsverbesserung nicht in Frage.

Cluster. Clustering ist eine Speichertechnik, mit deren Hilfe die Performance von Anfragen gesteigert werden kann. Mehrere Tabellen oder auch nur einzelne Tabellenspalten können in einem Speicherbereich (Cluster) abgelegt werden. Ihre Daten liegen dann physikalisch nah beieinander, so daß Anfragen schneller bearbeitet werden können. Dies lohnt sich vor allem bei großen Tabellen, die häufig selektiert und selten manipuliert werden. Das Clustering sollte wie die Indizierung idealerweise im laufenden Betrieb überwacht und gegebenenfalls angepaßt werden. Deshalb ist die Optimierung nicht das richtige Werkzeug, um darüber zu entscheiden.

Redundante Objekte. Um zu vermeiden, daß oft durchgeführte Anfragen ständig auf mehrere Tabellen zugreifen müssen, können bestimmte Spalten dupliziert werden. Zum Beispiel könnte eine Anfrage über die Objekttypen *Buch* und *Regal* zu den Feldern des Buches immer auch die Nummer des Regals suchen, in dem das Buch steht. Dann ist es sinnvoll, den Objekttypen Buch um ein Feld für diese Nummer zu erweitern. Die entsprechende Anfrage braucht dann nur noch auf eine Tabelle zuzugreifen.
Dies Vorgehen hat natürlich auch Nachteile. Redundante Objekte erfordern einen höheren Aufwand bei der Pflege. Jede Änderung der Nummer muß ja in beiden Tabellen nachvollzogen werden.

Aus diesem Grund kommen redundante Objekte als Tuningmaßnahme für die Optimierung nicht in Frage. Ähnlich wie bei der eins–zu–eins Umsetzung von 1:1–Verbindungen (siehe Kapitel 5.2) wäre der Aufwand hierfür zu hoch.

Anfragen. Einen Ansatzpunkt zur Leistungsverbesserung bieten die Anfragen selbst. Je nach Formulierung kann dieselbe Anfrage mehr oder weniger Zeit verbrauchen. Eine Anfrage könnte zum Beispiel zuerst das kartesische Produkt aus zwei Tabellen bilden und anschließend das Ergebnis anhand der Werte einer Spalte filtern. Performanter ist es, zuerst die eine Tabelle zu filtern und dann nur noch den relevanten Teil dieser Tabelle mittels des kartesischen Produktes mit der anderen Tabelle zu verknüpfen. Viele Datenbanken besitzen ein Werkzeug, das Anfragen automatisch optimiert, so daß sie dem Benutzer diese Arbeit abnehmen können.
Die Leistungsverbesserung durch Anfrageoptimierung ist ein Punkt, der nicht in der Modellierungs– sondern in der Ausführungsphase berücksichtigt werden sollte. Deshalb spielt er zum Zeitpunkt der Optimierung noch keine Rolle.

Tabellen splitten. Anhand der Anfragen, die an die Tabellen einer relationalen Datenbank gestellt werden, läßt sich beobachten, wie häufig welche Felder der einzelnen Tabellen angesprochen werden. Dies Wissen kann benutzt werden, um das Schema so zu strukturieren, daß zumindest einige Anfragen performanter bearbeitet werden. Wenn ein Teil der Felder einer Tabelle sehr häufig angefragt wird, während die restlichen Felder eher selten benötigt werden, ist es aus Gründen der Performance sinnvoll, die Tabelle zu splitten. Die oft und die wenig benutzten Felder kommen jeweils in eine eigene Tabelle. So werden die beiden Tabellen kleiner und viele Anfragen können schneller bearbeitet werden. Benachteiligt sind allerdings jene Anfragen, die auf alle Felder der ehemaligen Tabelle zugreifen müssen. Ihre Bearbeitung dauert nun länger.
Diese Art des Tunings kann die Optimierung leisten. Sie hat zwar keine konkreten Anfragen zur Verfügung, kann aber alle nötigen Informationen aus den Zugriffspfaden gewinnen. Sie kann diese Pfade analysieren und daraufhin entscheiden, welche Tabellen bzw. Objekttypen gesplittet werden sollen.

Redefinition des Modells. Hier gibt es mehrere Möglichkeiten. Zum einen kann es sinnvoll sein, **lange Text–Felder** umzustrukturieren. Wenn zum Beispiel nicht immer der gesamte Text für eine Anfrage relevant ist, kann der Text in einen kurzen Abriß und eine komplette längere Version unterteilt werden. Die lange Version wird in einer eigenen

Tabelle gespeichert, auf die dann seltener zugegriffen werden muß. Dies wird schon von der Generierung realisiert (siehe 5.4), so daß die Optimierung sich nicht mehr damit befassen muß.

Die zweite Möglichkeit besteht in der Redefinition der **Fremdschlüssel**. Dies Vorgehen empfiehlt sich vor allem, wenn Verbindungen zwischen Objekttypen umgesetzt werden, indem sie an die Tabelle eines der Objekttypen angehängt werden. Dadurch ist der Primärschlüssel des einen Objekttypen in der Tabelle des anderen als Fremdschlüssel vorhanden. Eine Anfrage braucht dann unter Umständen nur auf eine einzige Tabelle zuzugreifen. Wenn eine Anfrage aber die Verbindung zwischen zwei Objekttypen häufiger über ein anderes Feld als über den Primärschlüssel herstellt, sollte der alte Fremdschlüssel durch das häufiger selektierte Feld ersetzt werden. Für die Optimierung kommt aber auch dies Vorgehen nicht in Frage, da alle Verbindungen eigene Tabellen besitzen. Fremdschlüssel werden nur in den Verbindungstabellen selbst benutzt und dürfen dort auch nicht mehr verändert werden.

Eine weitere Möglichkeit bietet die Umstrukturierung der **Verbindungen**. Wie beim Splitten von Tabellen können auch hier die Anfragen als Informationsquellen dienen. Die Performance einer Anfrage fällt in der Regel mit der Anzahl der Tabellen, die sie benutzt. Anfragen, die über sehr viele Tabellen laufen müssen, um ihre Ergebnisse zusammenzusuchen, können beschleunigt werden, indem zusätzliche Verbindungen in das Modell eingetragen werden. Wo vorher ein Umweg über viele Tabellen nötig war, reicht dann vielleicht die Suche über eine Verbindung aus. Die zukünftigen Anfragen liegen der Optimierung in Form von Zugriffspfaden vor. Sie kann also lange Anfragen identifizieren. Die Entscheidung über eine Veränderung des Modells durch zusätzliche Verbindungen kann natürlich nicht allein auf syntaktischer Ebene getroffen werden. Auch semantische Überlegungen müssen mit einbezogen werden. Dies kann nur der Modellierer selbst leisten. Die Optimierung kann ihn aber zumindest auf lange unperformante Pfade hinweisen.

Schließlich gibt es noch die Redefinition von **Tabellen**. Das bedeutet in diesem Kontext, daß das Schema nach Tabellen durchsucht wird, die eigentlich nicht benötigt werden und daher gelöscht werden können. Es handelt sich dabei um Tabellen, die keine neuen Informationen enthalten oder von keiner Anfrage benutzt werden. Tabellen ohne neue Informationen kann natürlich nur der Benutzer finden, da dazu Kenntnisse über die Semantik des Modells erforderlich sind. Die Suche nach Tabellen ohne Anfragen kann die Optimierung aber übernehmen. Anfragen liegen der Optimierung ja in Form von Zugriffspfaden vor.

Leistungsverbesserung während der Optimierung. Wie die obigen Ausführungen zeigen, gibt es aus dem Katalog der üblichen Leistungsverbesserungen drei Maßnahmen, die die Optimierung durchführen kann. Sie sucht lange Zugriffspfade und teilt dem Benutzer mit, daß er eine alternative Modellierung wählen sollte. Sie findet Objekttypen, die auf keinem Zugriffspfad liegen und daher gelöscht werden sollten. Und sie stellt fest, welche Objekttypen gesplittet werden sollten. Diese drei Ansätze werden im folgenden innerhalb der Optimierung realisiert.

6.6 Ein Konzept für die Optimierung

Die Optimierung wird vom Datenmodell–Editor aus aufgerufen. Sie ist aber — anders als die Generierung — nicht von ihm abhängig. Sie besitzt eine eigene Steuerung und eine Oberfläche bestehend aus Fenstern und Masken zur Kommunikation mit dem Benutzer. Alle Angaben innerhalb der Optimierung beziehen sich auf das aktuell im Datenmodell–Editor geladene Datenmodell.
Nach dem Aufruf der Optimierung öffnet sich ein Fenster mit drei Punkten zur Auswahl:

Zugriffspfad bearbeiten: Hier können beliebig viele Zugriffspfade zu dem aktuellen Datenmodell erfaßt und nachbearbeitet werden werden. Zuerst wird ein Fenster geöffnet, in dem ein bereits vorhandener Zugriffspfad ausgewählt, oder ein neuer angelegt werden kann. Anschließend wird der Zugriffspfad–Editor geöffnet. Dort können alle Informationen, die zu einem Pfad gehören, eingetragen werden, also die selektierten Felder, die Felder mit einer Suchbedingung und die Häufigkeit, mit der der Pfad benutzt wird. Für einen schon vorhandenen Pfad können diese Angaben beliebig geändert werden.

Analyse starten: Dieser Punkt ruft die eigentliche Optimierung auf. Es gibt zehn verschiedene Mängel, auf die ein Datenmodell getestet wird:

1. Objekttypen, die keine Felder besitzen,
2. Objekttypen, die keine Verbindungen besitzen,
3. Objekttypen, die keinen Schlüssel besitzen,
4. Schlüsselfelder, die die Restriktion *kann* haben,
5. Objekttypen, die auf keinem Zugriffspfad liegen,

6. Objekttypen, die zu viele Felder der Typen *Text*, *Liste*, *Referenz* oder *Aufzählung* besitzen,
7. Objekttypen, deren Objekte mehr als einen Block belegen,
8. Objekttypen, die zusammengefaßt werden sollten,
9. Zugriffspfade, die zu lang für eine performante Bearbeitung sind und
10. Objekttypen, die gesplittet werden sollten.

Für jeden Test wird eine Routine aufgerufen, die ihn durchführt und seine Ergebnisse in der Datenbank speichert. Da es möglich ist, daß die Optimierung für das aktuelle Datenmodell schon einmal aufgerufen wurde, entfernt jede Routine ihre alten Daten, bevor sie die neuen ablegt. In den folgenden Abschnitten 6.6.2 bis 6.6.11 werden die zehn Routinen und die Gründe für ihren Aufruf näher erläutert.

Nachdem der letzte Test beendet ist, werden alle Ergebnisse wieder aus der Datenbank gelesen und in mehreren Fenstern dargestellt. Zunächst gibt ein Hauptfenster eine Übersicht über die zehn verschiedenen Arten von Mängeln, die gefunden werden können. Von dort aus kann der Benutzer für jeden angezeigten Mangel ein weiteres Fenster öffnen. Dort werden die Bezeichner der Objekttypen, Felder und Zugriffspfade aufgelistet, die den entsprechenden Mangel besitzen. Der Benutzer hat nun die Möglichkeit, sein Schema im Datenmodell–Editor entsprechend der Angaben in diesen Ergebnis–Fenstern zu ändern. Anschließend kann er von jedem Ergebnis–Fenster aus genau die Routine noch einmal angestoßen, welche die Ergebnisse dieses Fensters produziert hat. Wenn der Benutzer alle Mängel behoben hat, wird diese Routine keine neuen Ergebnisse liefern und ihr Fenster wird geschlossen.

Analyse bearbeiten: Dieser Punkt kann ausgeführt werden, wenn für das aktuelle Datenmodell schon einmal eine Optimierung gestartet wurde. Ihre Ergebnisse werden aus der Datenbank gelesen. Sie werden, wie im vorigen Punkt beschrieben, mit Hilfe der Ergebnis–Fenster dargestellt. Der Benutzer hat dieselben Bearbeitungsmöglichkeiten.

6.6.1 Erweiterungen des Data–Dictionaries

Die Optimierung braucht für ihre Arbeit vier neue Systemtabellen. Sie sind in Abbildung 6.1 dargestellt.

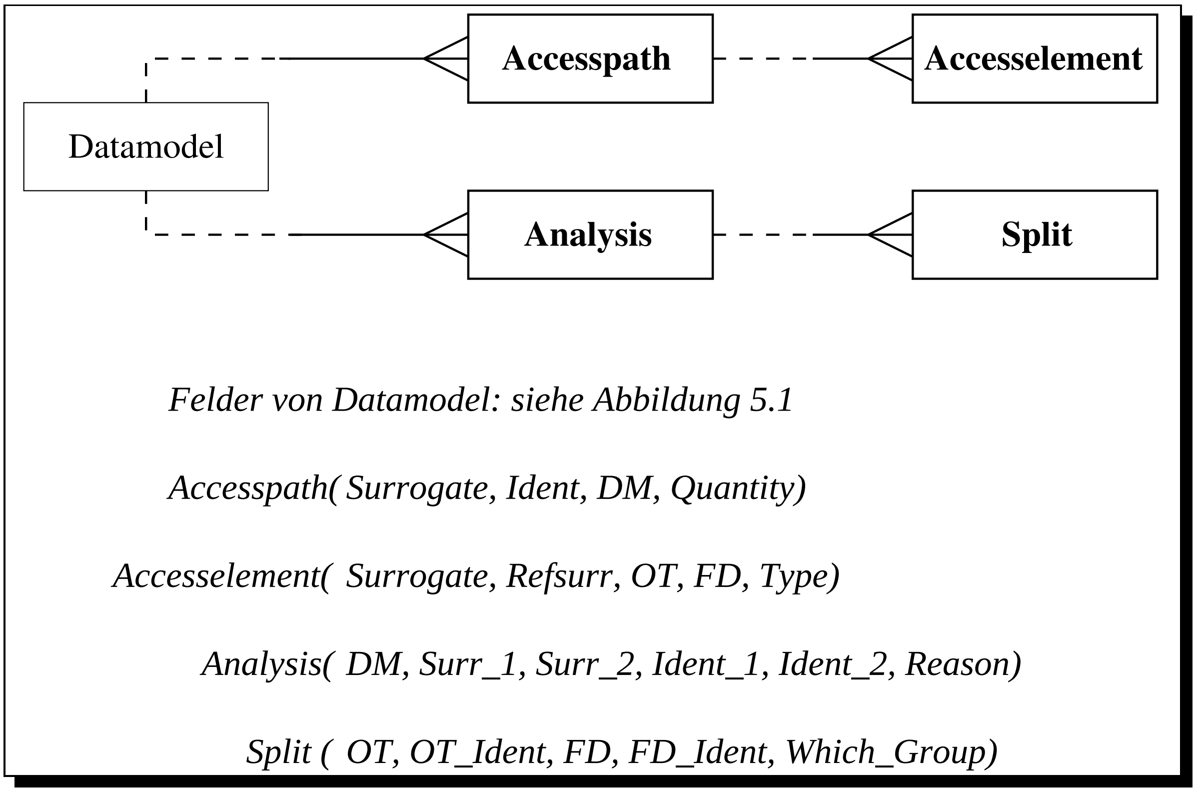


Abbildung 6.1: Erweiterung des Data–Dictionaries

Die Objekttypen *Accesspath* und *Accesselement* dienen zum Speichern der Zugriffspfade. Ein *Accesspath* besteht aus einem Bezeichner (*Ident*) und der Häufigkeit (*Quantity*), in der er innerhalb einer Zeiteinheit auftritt. Die Verbindung zum *Datamodel* ist durch das Feld *DM* realisiert. Ein *Accesselement* enthält die Surrogate eines Feldes (*FD*) und des zugehörigen Objekttypen (*OT*) und einen Typ (*Type*), der angibt, ob es gesucht wird oder an eine Bedingung geknüpft ist. Die Verbindung zwischen *Accesspath* und *Accesselement* wird durch das zusätzliche Feld *Refsurr* im *Accesselement* realisiert. Das Feld enthält das Surrogat des zugehörigen *Accesspath*.

Der Objekttyp *Analysis* erfasst die Ergebnisse der Optimierung. Dazu speichert er den Grund (*Reason*), aus dem das Ergebnis überhaupt aufgeführt wird. Als Gründe kommen die zehn Mängel in Frage, die die Optimierung in einem Datenmodell sucht. Je nach dem, was gefunden wird, müssen ein oder zwei Surrogate (*Surr_1*, *Surr_2*) mit den dazugehörenden Bezeichnern (*Ident_1*, *Ident_2*) erfasst werden. Sie enthalten Angaben über Objekttypen, Felder oder Zugriffspfade, die zu ändern sind. Die Verbindung zum *Datamodel* ist durch das Feld *DM* realisiert.
Für ein besonderen Ergebnis der Optimierung, nämlich das Splitten eines einzelnen Objekttypen in zwei neue, gibt es noch den Objekttypen *Split*. Er speichert den Namen (*OT_Ident*) des Objecttypen und sein Surrogat (*OT*). Die Felder *FD* und *FD_Ident* enthalten die Felder des Objekttypen. In *Which_Group* steht für jedes Feld, in welchen der beiden neuen Objekttypen es gehört.

6.6.2 Objekttypen ohne Felder

Jedes Datenmodell dient in LEU als Informationsbasis für eine Applikation. Objekttypen, die keine Felder besitzen, können aber keine Objekte speichern. Sie sind deshalb nutzlos. Diese Routine durchsucht ein Datenmodell nach Objekttypen ohne Felder. Sie betrachtet dabei aber nur Objekttypen, die im aktuellen Datenmodell beheimatet sind, also lokale und globale exportierte (vgl. Kapitel 3.2). Globale importierte Objekttypen dürfen nur in ihrem Heimatmodell geändert werden. Die gefundenen Objekttypen werden in der Analysis–Tabelle gespeichert mit dem Vermerk, daß sie leer sind.

6.6.3 Objekttypen ohne Verbindung

Ein Datenmodell, das ja einen Ausschnitt aus der realen Welt repräsentiert, sollte auf semantischer Ebene eine Einheit bilden. Ein Objekttyp, der mit keinem anderen in Verbindung steht, gehört deshalb nicht in das entsprechende Modell. Diese Funktion speichert alle Objekttypen, die keine Verbindungen besitzen, in der Analysis–Tabelle.

6.6.4 Objekttypen ohne Schlüssel

Ein Objekt muß innerhalb seines Objekttypen eindeutig identifizierbar sein. Zu diesem Zweck gibt es die Schlüssel. In LEU gibt es zwei Arten: benutzerdefinierte Schlüssel und vom System vergebene Surrogate (vgl. Kapitel 3.2). Innerhalb der Datenbank wird ein Objekt über sein Surrogat identifiziert. Dies ist für den Benutzer aber nicht sichtbar.

Er identifiziert die Objekte über den von ihm definierten Schlüssel. Deshalb sollte jeder Objekttyp einen solchen Schlüssel besitzen. Diese Routine sucht nach Objektttypen ohne Schlüssel. Sie betrachtet dabei nur Objekttypen, die im aktuellen Datenmodell beheimatet sind. Die gefundenen Objekttypen werden in der Analysis–Tabelle mit dem Kennzeichen "ohne Schlüssel" abgelegt.

6.6.5 Schlüsselfelder mit der Restriktion "kann"

Im Feld–Editor, wo die Felder der Objekttypen definiert werden, werden die Schlüsseleigenschaft und die Restriktion des Feldes unabhängig voneinander gesetzt. Dadurch ist es möglich, ein Schlüsselfeld mit der Restriktion *kann* zu definieren. Im Sinne der ER–Modellierung sollte ein Schlüsselfeld aber keine NULL–Werte enthalten. Deshalb sucht diese Funktion die Objekttypen mit kann–Schlüsselfeldern und speichert sie in der Analysis–Tabelle. Zusätzlich wird auch der Objekttyp, zu dem das Feld gehört, gespeichert, damit der Benutzer später nicht alle Objekttypen nach dem Feld durchsuchen muß. Auch hier werden nur Objekttypen, die im aktuellen Datenmodell beheimatet sind, betrachtet.

6.6.6 Objekttypen ohne Zugriffspfad

Um Werte in den Objekttypen speichern und diese Werte manipulieren zu können, muß es Operationen geben, die auf die Objekttypen zugreifen. Diese Operationen sind in diesem Stadium der Modellierung in Form von Zugriffspfaden gespeichert. Ein Objekttyp, der auf keinem Zugriffspfad liegt, wird demnach niemals benutzt. Er ist innerhalb des Datenmodells überflüssig. Deshalb wird getestet, ob jeder Objekttyp auf mindestens einem der zum Datenmodell gehörenden Zugriffspfade liegt. Falls Objekttypen gefunden werden, die in keinem Pfad vorkommen, werden sie in die Analysis–Tabelle eingetragen.

6.6.7 Objekttypen mit zuvielen speziellen Feldern

Es gibt in LEU eine Reihe von Feldtypen, die nicht komplett auf einen atomaren Datentypen abgebildet werden können (siehe 3.2). Dazu gehören die Typen *Text*, *Liste*, *Referenz* und *Aufzählung*. Für Felder vom Typ *Text* und *Liste* wird wie beschrieben eine eigene Tabelle angelegt. Die *Referenz* enthält zwar nur ein Surrogat und ist damit atomar, aber sie verweist mit diesem Surrogat auf ein anderes Feld oder einen ganzen Objekttypen. Beide stehen in einer fremden Tabelle. Eine Anfrage, die ein Referenz–Feld benutzt, muß also immer über eine Tabelle mehr suchen. Ebenso verhält es sich mit Feldern vom Typ

Aufzählung. Das Feld selbst ist atomar. Es enthält die Nummer eines Elementes des Aufzählungstypen. Das Element steht in einer fremden Tabelle, die alle Aufzählungselemente speichert. Auch hier muß demnach jede Anfrage eine Tabelle mehr berücksichtigen. Felder, die einen der vier genannten Typen besitzen, verlangsamen die Ausführung der Anfragen. Denn für jedes dieser Felder muß eine Tabelle mehr verarbeitet werden. Zusätzlich werden die Anfragen dadurch komplexer. Daher wird eine obere Grenze definiert, die bestimmt, wieviele dieser speziellen Felder ein Objekttyp höchstens besitzen sollte. Diese Grenze ist allerdings je nach verwandter Datenbank und Hardware verschieden. Sie muß jeweils für die aktuelle Konfiguration festgelegt werden.
Objekttypen, deren Anzahl an speziellen Feldern die obere Grenze überschreitet, werden in die Analysis–Tabelle eingetragen. Wiederum werden nur Objekttypen betrachtet, die im aktuellen Datenmodell beheimatet sind.

6.6.8 Zu große Objekttypen

Jeder Objekttyp ist in der Datenbank als Tabelle abgelegt. Die Datenbank speichert die Objekte ihrer Tabellen in Blöcken, deren Größe von der Datenbank und der verwandten Hardware abhängt. Wieviele Blöcke eine Tabelle belegt, hängt von der Anzahl, der Definition und dem Füllgrad ihrer Felder und von der Menge ihrer Objekte ab. Je mehr Blöcke eine Tabelle belegt, desto länger kann die Bearbeitung von Operationen auf dieser Tabelle dauern. Denn es sind nicht immer alle Blöcke einer Tabelle im Hauptspeicher präsent, so daß unter Umständen mehrfach Blöcke nachgeladen werden müssen.
Aufgrund dieser Überlegungen ist es besser, kleinere Tabellen anzulegen. Zu große Objekttypen sollten in zwei kleinere aufgeteilt werden. Allerdings muß dabei auch berücksichtigt werden, daß durch das Aufteilen einige Datenbankanfragen nun nicht mehr auf eine sondern auf mehrere Tabellen zugreifen müssen, was wiederum ein Performanceverlust sein kann.
Es ist während der Modellierungsphase schwer vorauszusagen, wieviel Speicherplatz die Tabelle eines Objekttypen tatsächlich belegen wird. Anhand der Datentypen läßt sich aber der maximale Speicherverbrauch eines Objektes bestimmen. (Die Umsetzung der LEU–Datentypen in die Typen der Datenbank wurde im Kapitel 5.4 beschrieben.) Die folgende Liste gibt an, wieviel Speicher jeder Datentyp verbraucht. Dabei wird für den Datenbank–Typ *Char* angenommen, daß er ein Byte pro Zeichen benötigt. Da in LEU die Typen, die nach *Number* umgesetzt werden, die Kapazität von einem Byte nicht überschreiten, und ein Byte die kleinste Speichereinheit ist, wird für sie jeweils ein Byte angenommen.

Boolean :	1 Byte
Integer :	1 Byte
Real :	1 Byte
Time :	1 Byte
Date :	1 Byte
String_n :	n Bytes
Text :	255 Bytes
Referenz :	1 Byte
Aufzählung :	1 Byte
Liste :	≪*Anzahl des Listentyps*≫ Bytes

Der maximale Speicherbedarf eines Objektes ergibt sich nun durch die Addition des Speicherplatzes aller Datentypen. Außerdem verbraucht die Datenbank ORACLE etwa zehn Prozent jedes Blockes für organisatorische Zwecke. Wenn ein einziges Objekt soviel Speicherplatz benötigt, daß es nicht mehr in die restlichen neunzig Prozent eines Blockes paßt, ist der Objekttyp sicherlich für eine performante Bearbeitung zu groß. Diese Routine sucht solche großen Objekttypen, die im aktuellen Datenmodell beheimatet sind, und schreibt sie in die Analysis–Tabelle.

6.6.9 1:1–Verbindungen

Die Transformationsregeln zur Abbildung eines ER–Modells in ein relationales Schema [FH89, Mei92, Vos87] sehen vor, daß 1:1–Verbindungen umgesetzt werden, indem sie mit einem der verbundenen Objekttypen zusammengefaßt werden. Der Grund hierfür liegt in der Performance bei der Datensuche und –manipulation. Durch das Zusammenfassen ist an den entsprechenden Zugriffen jeweils eine Tabelle weniger beteiligt, was sich in der Regel positiv auf die Ausführungsgeschwindigkeit der Datenbankoperationen auswirkt.
Dies Vorgehen wurde aber für die Generierung schon ausgeschlossen (siehe Kapitel 5.2). Deshalb werden die 1:1–Verbindungen von der Optimierung behandelt. Sie können umgangen werden, wenn die beiden verbundenen Objekttypen zusammengefaßt werden. Zunächst werden alle 1:1–Verbindungen eines Datenmodells gesucht. Ob die verbundenen Objekttypen aber zufammengefaßt werden können, hängt aber noch von zwei Tests ab.
Der erste Test betrifft die globalen Objekttypen. Das Datenmodell kann globale Objekttypen, die in fremden Modellen definiert wurden, importieren. Ein importierter Objekttyp kann aber nur gelesen und nicht verändert werden. Wenn also die gefundene Verbindung

an einem importierten Objekttypen hängt, muß sie bestehen bleiben.
Der zweite Test betrifft die Größe des Objekttypen, der nach dem Zusammenfassen entsteht. Wie im vorigen Abschnitt beschrieben wurde, sollten die Objekte die Kapazität eines Blockes nicht überschreiten. Deshalb wird auch hier der Speicherbedarf der beiden verbundenen Objekttypen bestimmt. Wenn beide zusammen die maximal zulässige Größe überschreiten, werden sie nicht zusammengefaßt.
Wenn keiner der beiden Tests Einwände gegen ein Zusammenfassen findet, werden die beiden betroffenen Objekttypen in der Analysis–Tabelle vermerkt. Dennoch sollte sich der Benutzer genau überlegen, ob es semantisch sinnvoll ist, dieser Empfehlung nachzukommen und die Objekttypen zusammenzufassen. Wenn es sich nämlich um eine kann–Verbindung handelt, besteht die Möglichkeit, daß für bestimmte Einträge jeweils nur der eine Teil der Felder gefüllt wird. Der andere Teil enthält NULL–Werte. Für die Datenmodellierung wird aber im allgemeinen empfohlen, möglichst wenige NULL–Werte zuzulassen.

6.6.10 Lange Zugriffspfade

Je mehr Tabellen eine Anfrage verarbeiten muß, desto mehr Zeit benötigt sie. Darauf sollte schon während der Erstellung des Datenmodells geachtet werden. Wenn bestimmte Informationen nur auf großen Umwegen gewonnen werden können, ist das Modell möglicherweise schlecht strukturiert. Dies läßt sich natürlich nicht automatisch feststellen. Diese Funktion kann aber Anfragen (in Form von Zugriffspfaden) aufspüren, die für eine performante Bearbeitung zu lang sind. Was *zu lang* ist, hängt natürlich vom eingesetzten System ab. Deshalb wird an dieser Stelle eine Variable eingeführt, die zwar mit einem initialen Wert vorbelegt ist, später aber vom Datenbankadministrator neu gesetzt werden sollte.
Für jeden Zugriffspfad des Modells wird nun getestet, ob er die durch die Variable festgelegte Grenze überschreitet. Die Pfade, für die das zutrifft, werden in die Analysis–Tabelle eingetragen.

6.6.11 Objekttypen splitten

Anhand der Zugriffspfade läßt sich feststellen, wie auf die einzelnen Objekttypen zugegriffen wird. Aus der Art der Zugriffe können wiederum Rückschlüsse auf eine semantisch sinnvolle Modellierung gezogen werden. Diese Routine betrachtet nacheinander jeden Objekttypen des Datenmodells und die Zugriffspfade, die ihn benutzen. Sie prüft dabei zwei Fragen:

1. Können die Felder des Objekttypen in zwei Gruppen geteilt werden, die jeweils von verschiedenen Zugriffspfaden angesprochen werden?

2. Wird auf einen Teil der Felder des Objekttypen sehr häufig zugegriffen, während der Rest der Felder eher selten benötigt wird?

Die Objekttypen, für die mindestens eine der beiden Fragen positiv beantwortet werden kann, sollten wie folgt gesplittet werden:

1. Trifft die erste Frage zu, wird ein zweiter Objekttyp definiert und mit dem ersten verbunden. Die Felder werden so auf die beiden Objekttypen verteilt, daß die Zugriffspfade jeweils nur einen der Objekttypen ansprechen müssen.

2. Trifft die zweite Frage zu, bleiben die häufig benötigten Felder in dem betroffenen Objekttypen. Für die restlichen Felder wird ein zweiter Objekttyp angelegt. Beide Objekttypen werden miteinander verbunden.

3. Treffen beide Fragen zu, wird ebenfalls ein zweiter Objekttyp definiert und mit dem ersten verbunden. Bei der Aufteilung der Felder ist dem ersten Verfahren der Vorzug zu geben, da es prüft, ob der Objekttyp Daten speichern soll, die eigentlich nichts miteinander zu tun haben.

Die Routine bearbeitet nacheinander jeden Objekttypen des aktuellen Datenmodells wie folgt: Grundsätzlich wird festgelegt, daß der Objekttyp in höchstens zwei neue Objekttypen gesplittet wird. Deshalb werden zunächst zwei Listen angelegt, die im Laufe der Bearbeitung mit den Surrogaten der Felder des betrachteten Objekttypen gefüllt werden. Die Felder, die zu dem ersten neuen Objekttypen gehören sollen, werden in die erste Liste eingetragen, die restlichen Felder in die zweite. Ein Splitten in mehr als zwei Objekttypen wäre nur in einem Fall notwendig, nämlich wenn die beiden entstehenden Objekttypen immer noch im Sinne des in Abschnitt 6.6.8 vorgestellten Tests zu groß wären. Aber selbst dann würde dies bei der nächsten Optimierung von der entsprechenden Funktion festgestellt. Eine Beschränkung auf zwei neue Objekttypen reicht also vollkommen aus.

Test 1. Zuerst wird geprüft, ob die Felder des Objekttypen in zwei Gruppen geteilt werden können, die jeweils von verschiedenen Zugriffspfaden angesprochen werden. Dazu wird jeder Zugriffspfad des Modells wie in Abbildung 6.2 dargestellt behandelt.
Die Felder des ersten gefundenen Pfades werden in die erste Liste eingetragen. Ein Pfad kann natürlich über verschiedene Objekttypen laufen. Hier werden aber nur die Felder

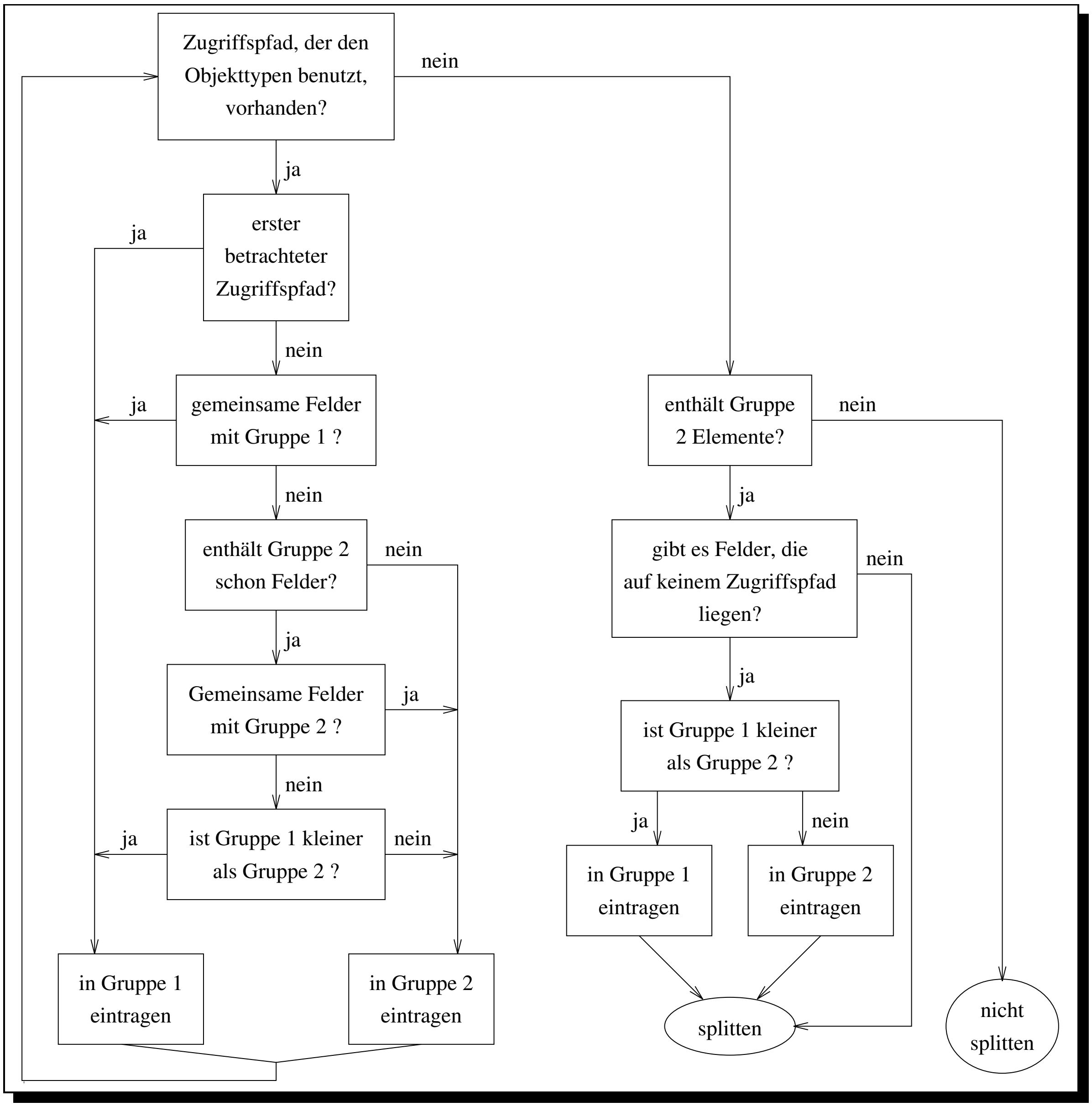


Abbildung 6.2: Splitten von Objekttypen nach Zugriffspfaden

betrachtet, die zu dem aktuellen zu untersuchenden Objekttypen gehören. Für jeden weiteren Pfad wird verglichen, ob er Felder benutzt, die schon in der ersten Liste stehen. Ist dies der Fall, werden seine restlichen Felder ebenfalls an diese Liste angehängt. Andernfalls handelt es sich um einen Pfad, der auf neue Felder zugreift. Wenn die zweite Liste noch

leer ist, werden seine Felder auf jeden Fall dort eingetragen. Ist sie nicht leer, wird zunächst geprüft, ob der Pfad auf Felder zugreift, die schon in der zweiten Liste stehen. Ist dies der Fall, werden seine restlichen Felder ebenfalls an diese Liste angehängt. Andernfalls handelt es sich um einen Pfad, der mit keinem der schon betrachteten Pfade gemeinsame Felder benutzt. Er gehörte also eigentlich in eine dritte Gruppe. Da aber nur zwei Gruppen angelegt werden sollen, werden seine Felder in die kleinere der beiden Listen eingetragen.
Wenn alle Zugriffspfade betrachtet wurden, wird festgestellt, ob der Objekttyp gesplittet werden soll. Ist die zweite Liste leer, überlappen sich die Felder aller betrachteten Zugriffspfade. Der Objekttyp kann nicht gesplittet werden. Ist sie aber gefüllt, sollte der Objekttyp so gesplittet werden, daß die Felder der einen Liste im alten Objekttypen bleiben und für die Felder der anderen Liste ein neuer Objekttyp angelegt wird.
Zuletzt werden die Felder des Objekttypen betrachtet, die auf keinem Zugriffspfad liegen. Sie werden an die kleinere Liste angehängt. Auf diese Weise ist der Objekttyp vollständig partitioniert.

Test 2. Wenn sich nach dem ersten Test herausstellt, daß der Objekttyp nicht gesplittet werden soll, wird ein weiterer Test durchgeführt. Er stellt fest, ob auf einen Teil der Felder des Objekttypen sehr häufig zugegriffen wird, während der Rest der Felder eher selten benötigt wird. Für diesen Test wird zunächst berechnet, wie oft auf die Felder des betrachteten Objekttypen zugegriffen wird.
Ein Zugriffspfad Z_i $(i \in IN)$ ist gewichtet. Das heißt, es ist bekannt, wie oft er pro Zeiteinheit benutzt wird. Das **Gewicht eines Zugriffspfades** wird mit G_{Z_i} bezeichnet. Die Häufigkeit, mit der auf ein Feld F_i pro Zeiteinheit zugegriffen wird, wird als **Gewicht des Feldes** bezeichnet und berechnet durch

$$G_{F_i} = \sum_j G_{Z_j} \ , \ Z_j \text{ greift auf } F_i \text{ zu.}$$

Ein Feld, auf das nicht zugegriffen wird, hat demnach das Gewicht 0. Die n Felder des betrachteten Objekttypen werden nach ihren Gewichten sortiert:

$$F_1, F_2, \ldots, F_n \text{ mit } G_{F_1} \geq G_{F_2} \geq \ldots \geq G_{F_n}$$

Als nächstes wird das **durchschnittliche Gewicht der Felder** berechnet:

$$G_D = \frac{\sum_{i=1}^{n} G_{F_i}}{n}$$

Die Felder F_1 bis F_m, deren Gewicht über dem Durchschnitt liegt, gehören potentiell zu den oft benutzten. Bei den folgenden Berechnungen werden daher nur noch die Felder F_1

bis F_{m+1} betrachtet. (Das Feld F_{m+1} wird für einen Vergleich mit dem Feld F_m benötigt.) Es kann nun der Fall eintreten, daß die Unterschiede zwischen den Gewichten der einzelnen Felder relativ gering sind. Es sollte aber nur dann gesplittet werden, wenn es einen deutlichen Unterschied im Zugriff auf die Felder gibt. Deshalb wird nach einer Lücke in der Reihenfolge der Gewichte gesucht. Dazu werden jeweils die Gewichte des i–ten und des (i+1)–ten Feldes verglichen. Nur wenn das eine mindestens doppelt so groß ist wie das andere, liegt eine Lücke vor:

$$\frac{G_{F_i}}{2*G_{F_{i+1}}} \begin{cases} \geq & 1 \quad \longrightarrow \quad Lücke \quad gefunden \\ < & 1 \quad \longrightarrow \quad keine \quad Lücke \end{cases}$$

Wenn eine Lücke zwischen F_i und F_{i+1} gefunden wurde, sollte der Objekttyp gesplittet werden. Die Felder F_1 bis F_i werden in die erste Liste eingetragen, die Felder F_{i+1} bis F_n kommen in die zweite Liste.
Der oben beschriebene Vorgang wird noch einmal anhand eines Beispiels verdeutlicht. Zu testen ist ein Objekttyp mit sechs Feldern.

1. Sechs Felder eines Objekttypen sind wie folgt gewichtet: $G_{F_1} = 200$, $G_{F_2} = 180$, $G_{F_3} = 150$, $G_{F_4} = 100$, $G_{F_5} = 90$, $G_{F_6} = 48$. Das durchschnittliche Gewicht wird berechnet: $G_D = (200 + 180 + 150 + 100 + 90 + 48)/6 = 128$. Zwischen den ersten vier Feldern wird nach einer Lücke gesucht. Felder 1 und 2: $200/(2 * 180) = 0.56$; Felder 2 und 3: $180/(2 * 150) = 0.6$; Felder 3 und 4: $150/(2 * 100) = 0.75$. Alle Ergebnisse sind kleiner als 1. Es wurde keine Lücke gefunden. Der Objekttyp wird nicht gesplittet.

2. Sechs Felder eines Objekttypen sind wie folgt gewichtet: $G_{F_1} = 200$, $G_{F_2} = 180$, $G_{F_3} = 50$, $G_{F_4} = 30$, $G_{F_5} = 20$, $G_{F_6} = 0$. Das durchschnittliche Gewicht wird berechnet: $G_D = (200 + 180 + 50 + 30 + 20 + 0)/6 = 80$. Zwischen den ersten drei Feldern wird nach einer Lücke gesucht. Felder 1 und 2: $200/(2*180) = 0.56$; Felder 2 und 3: $180/(2*50) = 1.8$. Dies Ergebnis ist größer als 1. Eine Lücke wurde gefunden. Der Objekttyp wird gesplittet.

Ergebnis. Wenn einer der beiden Tests positiv ausfällt, schlägt die Optimierung vor, den Objekttypen zu splitten. Einer der beiden neuen Objekttypen sollte die Felder aus der ersten Liste enthalten, der andere die Felder aus der zweiten Liste. Deshalb werden die Felder der ersten Liste mit der Kennung *1* in die Split–Tabelle eingetragen und die Felder der zweiten Liste mit der Kennung *2*.

6.7 Entwurf der Optimierung

Dies Kapitel beschreibt den Entwurf für die Optimierung. Zunächst wird die Oberfläche, die zur Kommunikation mit dem Benutzer dient, vorgestellt. Anschließend werden die einzelnen Module beschrieben.

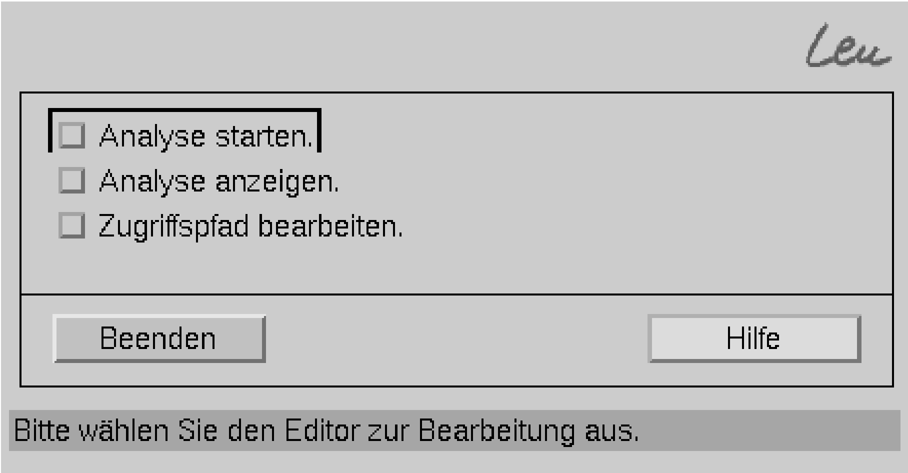


Abbildung 6.3: Startfenster der Optimierung

6.7.1 Die Oberfläche

Die Optimierung wird vom Datenmodell–Editor aus für das aktuelle Datenmodell gestartet. Zuerst öffnet sich das in Abbildung 6.3 gezeigte Startfenster. Es beinhaltet im mittleren Bereich drei Knöpfe zur Bearbeitung von Zugriffspfaden, zum Starten einer neuen Analyse und zum Anzeigen einer alten Analyse. Im unteren Teil des Fensters befindet sich ein Knopf zum Beenden der Optimierung und ein Knopf für den Abruf von Hilfsinformationen zur Handhabung des Fensters.
Zur Bearbeitung von Zugriffspfaden wird zuerst ein Fenster geöffnet, in dem der Name des Pfades bestimmt wird (Abbildung 6.4). Dazu wird entweder einer der im oberen Bereich angezeigten vorhandenen Pfade angeklickt, oder in der darunter liegenden Zeile ein neuer Name eingegeben. Durch Drücken des OK–Knopfes wird der Zugriffspfad–Editor für den gewählten Pfad geöffnet. Mit dem Abbruch–Knopf wird die Bearbeitung des Fensters abgebrochen. Hilfsinformationen zur Handhabung des Fensters werden über den Hilfe–

Knopf abgerufen.

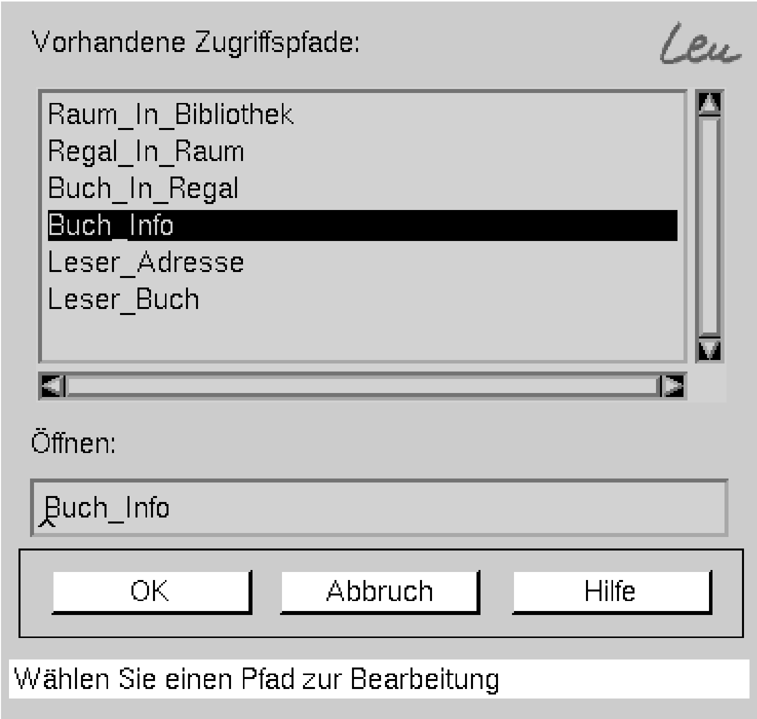


Abbildung 6.4: Auswahl eines Zugriffspfades

Der Zugriffspfad–Editor (Abbildung 6.5) dient zur Bearbeitung eines Zugriffspfades. Im oberen Teil befinden sich zwei Kästen zur Auswahl der Felder, die der Zugriffspfad benötigt. Links werden die Objekttypen des aktuellen Datenmodells angezeigt. Durch Anklicken eines Objekttypen wird der rechte Kasten mit den Feldern des Objekttypen gefüllt. Diese Felder können von dem bearbeiteten Zugriffspfad entweder selektiert oder mit einer Suchbedingung verknüpft werden. Für jeden dieser Fälle gibt es im mittleren Teil des Fensters einen Kasten. Durch einen Doppelklick wird ein Feld in einen dieser Kästen übernommen. Welcher Kasten dies ist, wird durch den Knopf über dem Auswahl–Kasten für Felder festgelegt. Er kann auf "Selektieren" oder "Suchbedingung" geschaltet werden. Anstelle eines Doppelklicks auf ein ausgewähltes Feld kann auch der entsprechende unten im Fenster stehende Knopf gedrückt werden. Die Felder, die in den Selektions– oder Suchbedingungs–Kasten übernommen wurden, können über den Löschen–Knopf wieder entfernt werden. Für die Eingabe der Häufigkeit, mit der der Zugriffspfad benutzt wird, steht unter den

Kästen eine edierbare Zeile zur Verfüfung. Mit dem Speichern–Knopf kann der bearbeitete Zugriffspfad gespeichert werden.
Für die Darstellung der Analyseergebnisse wird zunächst das Hauptfenster aus Abbildung 6.6 geöffnet. Es enthält für jeden möglichen Mangel, den die Optimierung bei der Analyse des Datenmodells gefunden haben könnte, einen Knopf. Die Knöpfe der Mängel, die tatsächlich gefunden wurden, sind sensitiv geschaltet. Durch das Drücken eines Knopfes wird ein spezielles Ergebnis–Fenster geöffnet. Mit dem Beenden–Knopf werden das Hauptfenster und alle geöffneten Ergebnis–Fenster geschlossen. Über den Hilfe–Knopf werden Hilfsinformationen zur Handhabung des Fensters abgerufen.
Es gibt zwei Arten von Ergebnisfenstern. Die Abbildung 6.7 zeigt das Fenster, für die Objekttypen ohne Felder. Es ist repräsentativ für die Fenster der ersten neun Ergebnisse. In dem Kasten werden alle Objekttypen angezeigt, die keine Felder besitzen. Durch Drücken des Analyse–Knopfes wird genau die Routine der Optimierung gestartet, die die Ergebnisse des aktuellen Fensters produziert hat. Über Abbruch wird das Fenster verlassen. Das Fenster aus Abbildung 6.8 zeigt als Ergebnis des zehnten Analyse–Punktes die Objekttypen, die gesplittet werden sollen. Im ersten Kasten werden die Objekttypen selbst dargestellt. Durch Anklicken eines Objekttypen werden die beiden nächsten Kästen mit den Feldern des Objekttypen gefüllt. Die Felder eines Kastens sollen in einen Objekttypen zusammengefaßt werden. Durch Drücken des Analyse–Knopfes wird die Routine gestartet, die zu splittende Objekttypen sucht. Über Abbruch wird das Fenster verlassen.

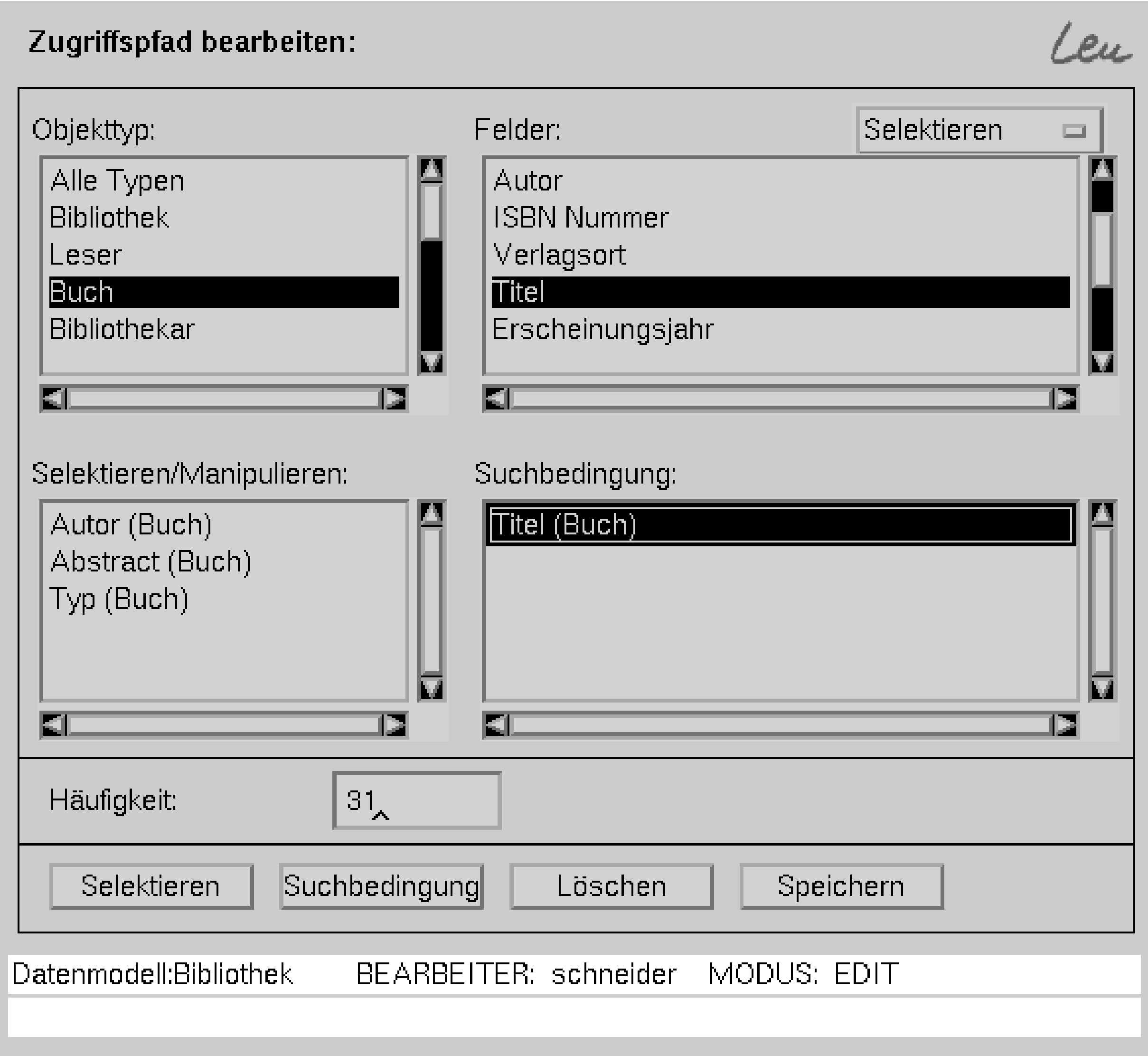


Abbildung 6.5: Zugriffspfad–Editor

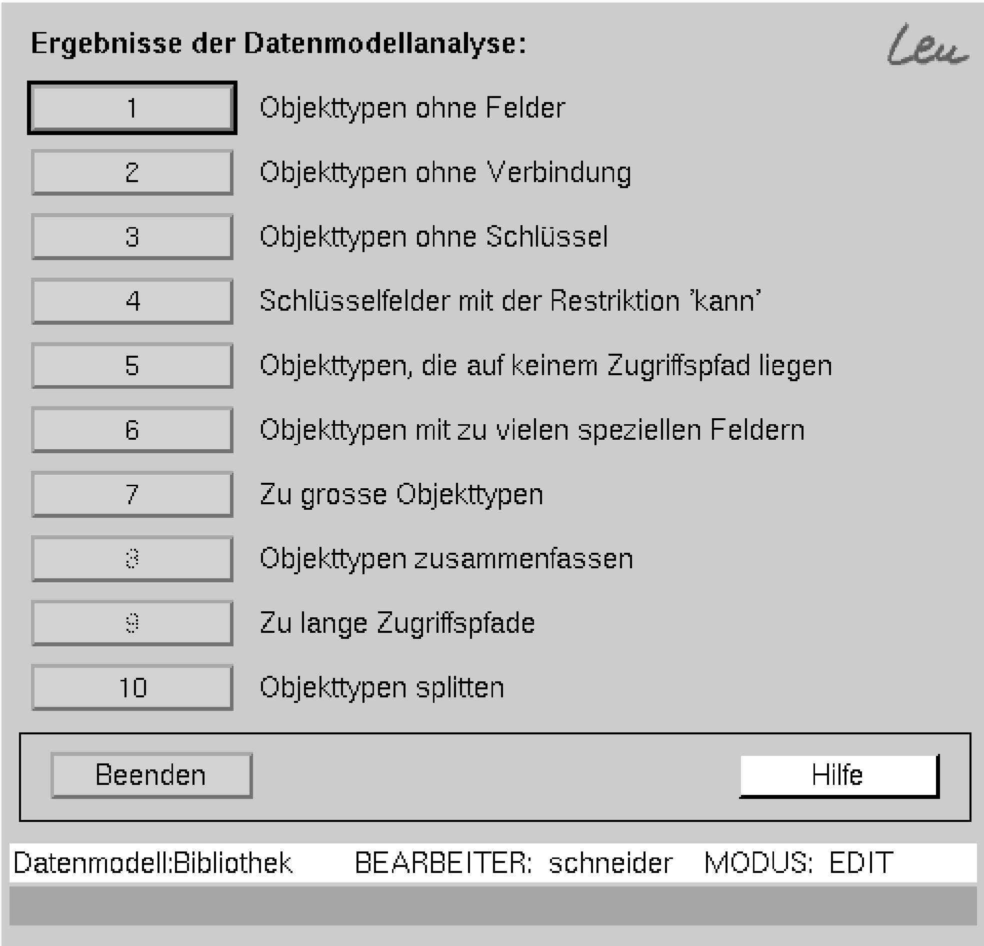


Abbildung 6.6: Hauptfenster für die Ergebnisse

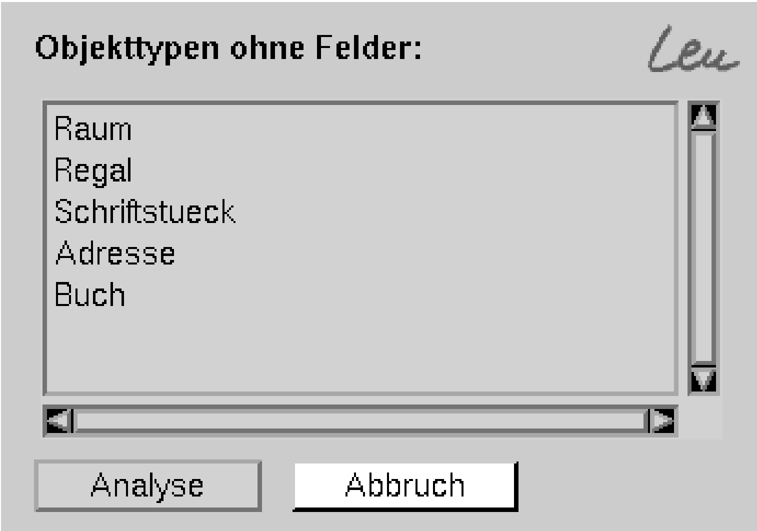


Abbildung 6.7: Ergebnisfenster für Objekttypen ohne Felder

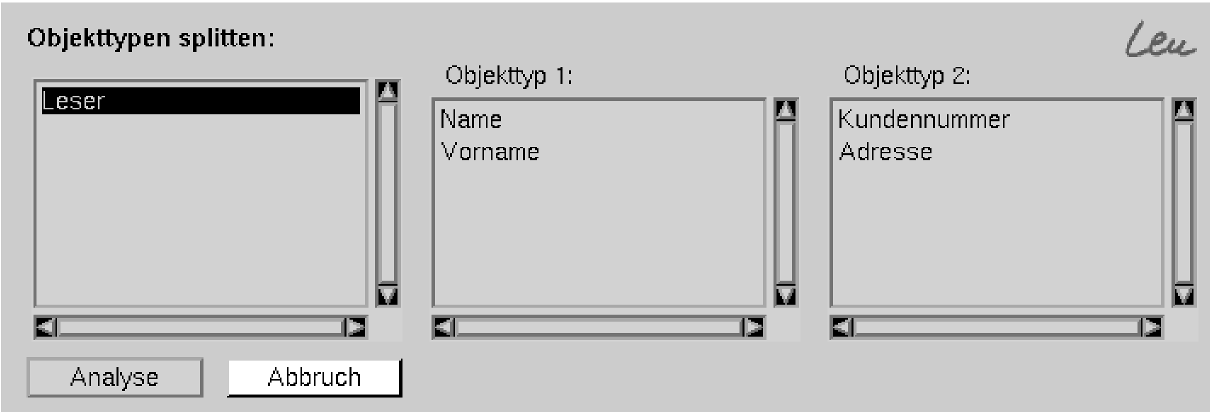


Abbildung 6.8: Ergebnisfenster für zu splittende Objekttypen

6.7.2 Die Module

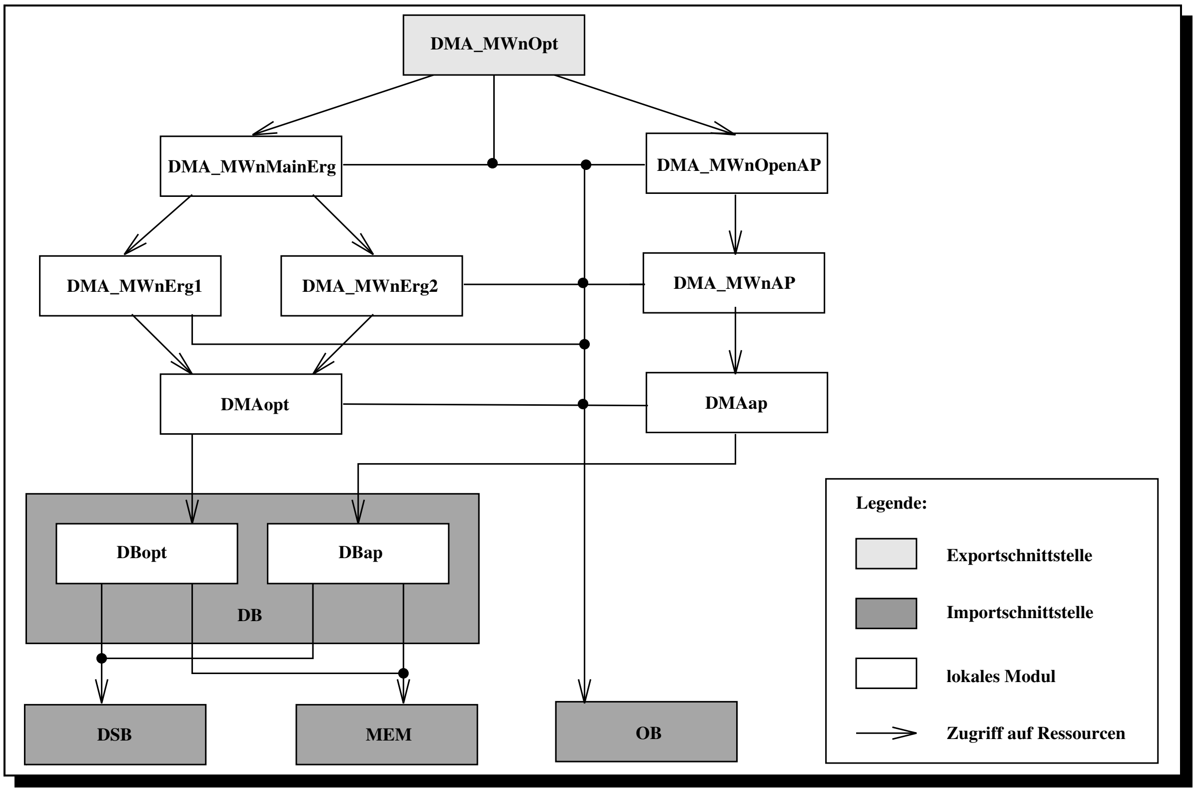


Abbildung 6.9: Modulhierarchie der Optimierung

Die Abbildung 6.9 zeigt die Modulhierarchie der Optimierung. Im folgenden werden die Funktionen der einzelnen Module beschrieben:

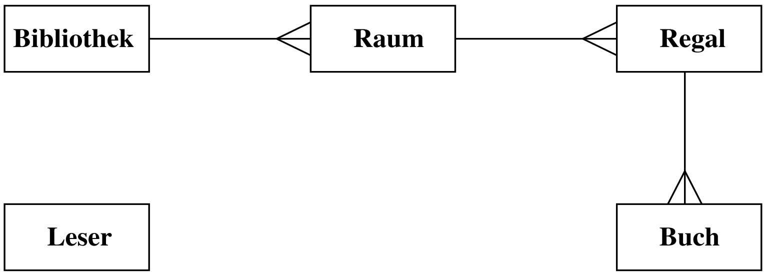
Die Importschnittstelle. Diese Schnittstelle besteht aus den LEU–Modulen, die von der Optimierung benutzt werden. Jeder Kasten steht für eine ganze Hierarchie von Modulen, die unter einem Bezeichner zusammengefaßt werden. **DSB** verkapselt die Datenstrukturen und die abstrakten Datentypen von LEU. **MEM** steuert die Speicherverwaltung von LEU. **OB** enthält Basis–Funktionen zum Aufbau der LEU–Oberflächen. **DB** verkapselt die Datenbank und enthält damit sämtliche Funktionen, die Zugriff auf die Datenbank haben.

Die lokalen Module. Es gibt zwei Arten von lokalen Modulen. Die einen werden in der Programmiersprache C geschrieben. Zu ihnen gehören die Module **DBopt** und **DBap**. Sie sind außerdem ein Teil der DB–Hierarchie. DBopt enthält alle DB–Funktionen, die für den Analyseteil der Optimierung benötigt werden. Ihre Arbeitsweise wurde innerhalb des Konzeptes für die Optimierung (Kapitel 6.6) erläutert. DBap stellt DB–Funktionen zur Erfassung und Manipulation von Zugriffspfaden (Kapitel 6.4) zur Verfügung.
Die zweite Art lokaler Module findet man im oberen Bereich der Hierarchie. Sie werden in der Regelsprache des ISA–Dialog–Managers [ISA92] geschrieben. Mit dieser Sprache wird die Oberfläche von LEU programmiert. Für jede Art Fenster, die die Optimierung besitzt, wird ein Modul angelegt. Diese Module definieren die Elemente (Kästen, Knöpfe, Überschriften, Statuszeilen, etc.) und das Layout eines Fensters. **DMA_MWnOpenAP** erstellt das Fenster für die Auswahl eines Zugriffspfades (Abbildung 6.4). Mit **DMA_MWnAP** wird der Zugriffspfad–Editor (Abbildung 6.5) definiert. **DMA_MWnMainErg** ist für das Hauptfenster der Analyseergebnisse (Abbildung 6.6) zuständig. Die Ergebnisfenster der ersten neun Analysepunkte (Abbildung 6.7) werden mit **DMA_MWnErg1** erstellt. Das Ergebnisfenster des zehnten Punktes (Abbildung 6.8) wird mit **DMA_MWnErg2** definiert.
Auf der mittleren Ebene der lokalen Module findet man **DMAopt** und **DMAap**. Sie bilden die Schnittstelle zwischen den in C und den in Regeln geschriebenen Modulen, da sie beides benutzen. Über sie ist eine Kommunikation und ein Datenaustausch zwischen der Oberfläche und den DB–Modulen möglich. Damit haben diese beiden Module zwei Aufgaben. Die erste besteht darin, Daten, die von DBopt und DBap in Form von C–Strukturen geliefert werden, zu transformieren und zur Darstellung an die Oberflächen–Module weiterzugeben. Die zweite Aufgabe besteht in der Steuerung der Aktionen innerhalb der Oberfläche. Eine solche Aktion ist beispielsweise das Drücken des Beenden–Knopfes. Daraufhin muß das Fenster geschlossen werden. Möglicherweise müssen vorher auch noch Daten in der Datenbank abgelegt werden. Dies alles wird in DMAopt für die Analysefenster und in DMAap für die Zugriffspfadfenster realisiert.

Die Exportschnittstelle. Die Optimierung exportiert nur das Modul **DMA_MWnOpt**. Es ist in der Regelsprache geschrieben und definiert das Startfenster der Optimierung (Abbildung 6.3). Dies Fenster wird vom Datenmodell–Editor aus aufgerufen.

6.8 Ein Beispiel

In diesem Abschnitt wird die Optimierung anhand eines kleinen Beispielmodells beschrieben. Dies Modell erhebt keinerlei Anspruch auf Vollständigkeit.
Zu Beginn werden im Datenmodell– und Feld–Editor die Objekttypen *Bibliothek*, *Raum*, *Regal*, *Buch* und *Leser* mit drei Verbindungen angelegt:



Für die Optimierung ist bzgl. der Definition der Objekttypen nur interessant, welche Felder, Datentypen, Schlüsseleigenschaften und Restriktionen sie haben. Die folgende Tabelle stellt diese Informationen dar:

Definition der Objekttypen				
Objekttyp	Feld	Datentyp	Schlüssel	Restriktion
Bibliothek	Name	String(30)	ja	kann
	Träger	String(70)	nein	kann
Raum	Nummer	Integer(2)	nein	kann
	Thema	String(35)	nein	kann
Regal	Nummer	Integer(3)	nein	kann
	Inhalt	String(35)	nein	kann
Buch	Nummer	Integer(7)	ja	muß
	Autor	Liste(String(30))	nein	kann
	Titel	String(70)	nein	muß
	Abstract	Text	nein	kann
	Typ	Aufzählung	nein	muß
	Standort	Referenz(Regal)	nein	muß

Die folgende Tabelle enthält die Zugriffspfade zu diesem Datenmodell. In der Spalte *Suchen* stehen die Felder, die gesucht werden. Die Spalte *Bedingung* enthält die Felder, an die eine Bedingung geknüpft ist. Die Objekttypen, zu denen die Felder gehören, werden in

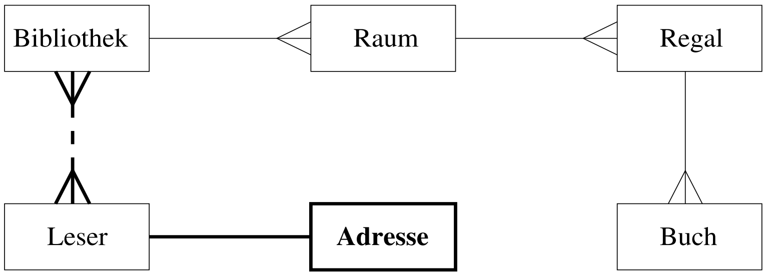
Klammern hinter die Felder geschrieben. Mit *Anzahl* wird die Häufigkeit bezeichnet, mit der der Zugriffspfad pro Zeiteinheit auftritt.

Zugriffspfade			
Bezeichner	Suchen	Bedingung	Anzahl
Raum_In_Bibliothek	Name(Bibliothek)	Nummer(Raum)	3
Regal_In_Raum	Nummer(Raum)	Nummer(Regal)	10
Buch_In_Regal	Standort(Buch)	Nummer(Buch)	56
Buch_Info	Autor(Buch) Abstract(Buch) Typ(Buch)	Titel(Buch)	31

Erste Optimierung. Obwohl die Definition keineswegs vollständig ist, ruft der Benutzer die Optimierung schon einmal auf. Sie liefert folgende Ergebnisse:

Objekttypen ohne Felder (Kap. 6.6.2)	:	Leser
Objekttypen ohne Verbindung (Kap. 6.6.3)	:	Leser
Objekttypen ohne Schlüssel (Kap. 6.6.4)	:	Raum, Regal
kann–Schlüsselfelder (Kap. 6.6.5)	:	Name(Bibliothek)
Objekttypen ohne Zugriffspfad (Kap. 6.6.6)	:	Leser
zuviele spezielle Felder (Kap. 6.6.7)	:	Buch

Der Benutzer ändert das Datenmodell nach den Ergebnissen der Optimierung. Er definiert Felder für den *Leser* und verbindet ihn mit *Bibliothek*. Zusätzlich erstellt er noch einen Objekttypen *Adresse* für den Leser. *Raum* und *Regal* erhalten einen Schlüssel. In *Bibliothek* setzt er die Restriktion des Schlüsselfeldes auf *muß*. Zwei neue Zugriffspfade für den *Leser* werden definiert. Der Benutzer entscheidet, daß das Feld *Standort(Buch)* überflüssig ist, da er diese Information auch über eine Anfrage bekommen kann. Er ändert den Zugriffspfad *Buch_In_Regal* entsprechend. Und um die Anzahl der speziellen Felder von *Buch* noch weiter zu reduzieren, ändert er den Typ von *Autor* und ergänzt das Feld *Co–Autoren*.
Das geänderte Modell mit seinen Zugriffspfaden sieht nun so aus (die Änderungen sind fettgedruckt):



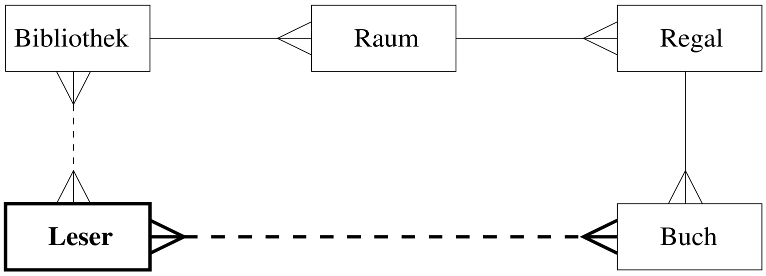
Definition der Objekttypen				
Objekttyp	Feld	Datentyp	Schlüssel	Restriktion
Bibliothek	Name	String(30)	ja	**muß**
	Träger	String(70)	nein	kann
Raum	Nummer	Integer(2)	**ja**	**muß**
	Thema	String(35)	nein	kann
Regal	Nummer	Integer(3)	**ja**	**muß**
	Inhalt	String(35)	nein	kann
Buch	Nummer	Integer(7)	ja	muß
	Autor	**String(30)**	nein	kann
	Co–Autoren	**String(70)**	**nein**	**kann**
	Titel	String(70)	nein	muß
	Abstract	Text	nein	kann
	Typ	Aufzählung	nein	muß
Leser	**Kundennummer**	**Integer(10)**	**ja**	**muß**
	Name	**String(70)**	**nein**	**kann**
Adresse	**Straße**	**String(70)**	**ja**	**muß**
	PLZ	**Integer(5)**	**ja**	**muß**
	Ort	**String(70)**	**nein**	**kann**

Zugriffspfade			
Bezeichner	Suchen	Bedingung	Anzahl
Raum_In_Bibliothek	Name(Bibliothek)	Nummer(Raum)	3
Regal_In_Raum	Nummer(Raum)	Nummer(Regal)	10
Buch_In_Regal	**Nummer(Regal)**	Nummer(Buch)	56
Buch_Info	Autor(Buch) Abstract(Buch) Typ(Buch)	Titel(Buch)	31
Leser_Adresse	**Straße(Adresse)** **PLZ(Adresse)** **Ort(Adresse)**	**Kundennummer(Leser)**	**13**
Leser_Buch	**Nummer(Buch)**	**Kundennummer(Leser)** **Name(Bibliothek)** **Nummer(Raum)** **Nummer(Regal)**	**29**

Zweite Optimierung. Nun ruft der Benutzer die Optimierung wieder auf. Die Mängel, die sie in der vorigen Version des Datenmodells gefunden hatte, sind behoben. Die alten Ergebnisse werden gelöscht. Es werden aber zwei neue Mängel gefunden:

Objekttypen zusammenfassen (Kap. 6.6.9)	:	Leser & Adresse
lange Zugriffspfade (Kap. 6.6.10)	:	Leser_Buch

Um auch diese beiden Mängel zu beheben, ändert der Benutzer das Modell wiederum. Er faßt die Objekttypen *Leser* und *Adresse* zusammen, indem er die Felder der *Adresse* an den *Leser* anhängt. So hat er die 1:1–Verbindung vermieden. Der neue Objekttyp ist nicht zu groß, denn sonst hätte die Optimierung das Zusammenlegen nicht empfohlen. Darüber hinaus enthält er noch nicht einmal NULL–Werte in den neuen Feldern, da es sich um eine muß–Verbindung handelte. Durch diese Änderung muß auch der Zugriffspfad *Leser_Adresse* angepaßt werden. Den Zugriffspfad *Leser_Buch* verkürzt der Benutzer, indem er eine Verbindung von *Leser* nach *Buch* einrichtet.
Nach diesen Änderungen sehen Datenmodell und Zugriffspfade wie folgt aus (die Änderungen sind fettgedruckt):



Definition der Objekttypen				
Objekttyp	Feld	Datentyp	Schlüssel	Restriktion
Bibliothek	Name	String(30)	ja	muß
	Träger	String(70)	nein	kann
Raum	Nummer	Integer(2)	ja	muß
	Thema	String(35)	nein	kann
Regal	Nummer	Integer(3)	ja	muß
	Inhalt	String(35)	nein	kann
Buch	Nummer	Integer(7)	ja	muß
	Autor	String(30)	nein	kann
	Co–Autoren	String(70)	nein	kann
	Titel	String(70)	nein	muß
	Abstract	Text	nein	kann
	Typ	Aufzählung	nein	muß
Leser	Kundennummer	Integer(10)	ja	muß
	Name	String(70)	nein	kann
	Straße	**String(70)**	**nein**	**kann**
	PLZ	**Integer(5)**	**nein**	**kann**
	Ort	**String(70)**	**nein**	**kann**

Zugriffspfade			
Bezeichner	Suchen	Bedingung	Anzahl
Raum_In_Bibliothek	Name(Bibliothek)	Nummer(Raum)	3
Regal_In_Raum	Nummer(Raum)	Nummer(Regal)	10
Buch_In_Regal	Nummer(Regal)	Nummer(Buch)	56
Buch_Info	Autor(Buch) Abstract(Buch) Typ(Buch)	Titel(Buch)	31
Leser_Adresse	**Straße(Leser)** **PLZ(Leser)** **Ort(Leser)**	Kundennummer(Leser)	13
Leser_Buch	Nummer(Buch)	Kundennummer(Leser)	29

Dritte Optimierung. Der Benutzer ruft die Optimierung zum dritten Mal auf. Wiederum sind alle Mängel aus der vorigen Version behoben, so daß die alten Ergebnisse gelöscht werden. Auch neue Mängel werden nicht mehr gefunden. Die Optimierung ist damit beendet.

Kapitel 7

Test von Generierung und Optimierung

Dies Kapitel beschreibt die Testphase für die Generierung und die Optimierung. Zunächst wurden alle implementierten Funktionen separat getestet. Anschließend wurden Integrationstests durchgeführt. Diese Tests betrafen einerseits die Integration der einzelnen Funktionen innerhalb der Generierung und der Optimierung. Andererseits wurde die Integration der Generierung in LEU getestet. Zuletzt wurden einige Laufzeittests durchgeführt.

7.1 Funktions– und Integrationstests

In dieser ersten Phase des Testens wurde für jede implementierte Funktion ein Testplan nach folgendem Muster aufgestellt:

Funktion: *Name der Funktion*
Voraussetzung: *initialer Zustand der Testumgebung*
Anmerkung: *Hinweise zum Verständnis des Tests oder der Notation*
Testfälle:

n. *n–ter Testfall*

Vorbereitung: *Vorbereitungen für diesen Testfall*

Eingabe: *Belegung der Eingabeparameter*

Ausgabe: *Fehlermeldung und Belegung der Ausgabeparameter*

Ergebnis: *Feststellung, ob die Funktion korrekt arbeitet; Auswirkungen*

Nach diesen Testplänen wurden alle Funktionen getestet. Falls ein Fehler gefunden wurde, wurde die Implementierung korrigiert und die Funktion wiederum getestet, bis sie korrekt arbeitete. Der Anhang A enthält die wesentlichen Auszüge aus den Testberichten der Generierung und der Optimierung.
Für die Integrationstests wurde ein ähnlicher Testplan aufgestellt. Jeder Aufruf einer Funktion der Exportschnittstelle wurde nach dem oben beschriebenen Schema getestet.
Nach der Integration in LEU wurde schließlich die Steuerung der Generierung, die (wie im Abschnitt 5.7 beschrieben) vom Datenmodell–Editor aus erfolgt, getestet.

7.2 Laufzeittests

Die Generierung wurde schon in LEU integriert. Wieviel Zeit sie für ein Datenmodell benötigt, hängt natürlich von vielen Parametern ab, z. B. von der Anzahl der Objekttypen, der Felder und der Verbindungen, den verwandten Feldtypen, den Stati der Objekttypen, etc. Die Erfahrung zeigt aber, daß ein Datenmodell in relativ kurzer Zeit generiert werden kann. Die Generierung des *Bibliothek*–Modells dauert zum Beispiel zwischen 10 und 30 Sekunden, sofern keine anderen Prozesse im System laufen.
Die Optimierung ist noch nicht in LEU integriert. Es sind aber einige Tests durchgeführt worden. Die Testsituation ist folgende: Mit dem Datenmodell–Editor wird ein Datenmodell angelegt, in dem die Optimierung alle Mängel, die die Performance betreffen, finden kann, also zu lange Zugriffspfade (6.6.10), zu große Objekttypen (6.6.8), Objekttypen mit zu vielen speziellen Feldern (6.6.7), zu splittende Objekttypen (6.6.11) und Objekttypen mit 1:1–Verbindungen (6.6.9). Dies Datenmodell wird generiert. Pro Objekttyp und pro Verbindung werden 5000 Testdaten eingegeben. Dann werden mit LEU Anfragen an die Datenbank gestellt, die die mangelhaften Objekttypen betreffen. Die Zeit, die diese Anfragen brauchen, wird gemessen. Da diese Anfragen von LEU aus gestellt und gemessen werden, dokumentieren die Testergebnisse die Zeit, die der LEU–Benutzer auf das Ergebnis seiner Anfrage wartet. Für den nächsten Test wird die Optimierung aufgerufen und das Datenmodell entsprechend ihrer Vorschläge verändert. Wiederum wird generiert. Die Anfragen werden, wenn nötig, an das geänderte Modell angepaßt. Dann werden die Anfragen noch einmal gestellt. Wieder wird die Zeit gemessen.
Für jede Möglichkeit zur Performanceverbesserung (bis auf die 1:1–Verbindungen) werden drei Testfälle dokumentiert: einer vor der Optimierung und zwei nach jeweils unterschiedlichen Veränderungen. Um einen möglichst repräsentativen Wert zu erhalten, wird

jede Anfrage zwanzigmal abgesetzt. Die unten angegebenen Ergebnisse sind die durchschnittlichen Zeiten aus je zwanzig Versuchen in Sekunden. Die Genauigkeit liegt bei drei Nachkommastellen, also im Millisekunden–Bereich.
Es folgt ein Ausschnitt aus den durchgeführten Tests:

Zu lange Zugriffspfade:
Anfrage: Suche über x Objekttypen mit Verbindungen.

1. 8 Objekttypen: 8.605
2. 5 Objekttypen: 5.532
3. 2 Objekttypen: 1.725

Zu große Objekttypen:
Anfrage: Suche über fünf Felder eines Objekttypen.

1. Objekttyp mit 35 Feldern: 2.723
2. Objekttyp mit 18 Feldern: 1.231
3. Objekttyp mit 5 Feldern: 0.692

Objekttyp mit zu vielen speziellen Feldern:
Anfrage: Suche über x Felder vom Typ Text, Referenz, Aufzählung.

1. 9 Felder: 2.463
2. 5 Felder: 1.202
3. 2 Felder: 0.737

Objekttyp splitten:
Anfrage: Suche über zehn Felder eines Objekttypen.

1. Objekttyp mit 30 Feldern: 13.024
2. Objekttyp mit 20 Feldern: 2.274
3. Objekttyp mit 10 Feldern: 1.540

1:1–Verbindung:
Anfrage1: Suche über zwei Objekttypen mit einer Verbindung.
Anfrage2: Suche über einen Objekttypen.

1. zweimal 5 Felder mit Verbindungen: 4.111

2. einmal 10 Felder ohne Verbindungen: 0.958

Die Testergebnisse zeigen, daß eine Optimierung in der Regel eine Leistungsverbesserung bei der Ausführung von Anfragen bewirkt. Ein Vergleich der Ergebnisse verschiedener Anfragen zeigt aber auch, daß es Grenzen gibt.
Eine solche Grenze setzt die Größe bzw. der Füllgrad einer Tabelle. Die Tests an 1:1–Verbindungen zeigen, daß das Zusammenfassen der Objekttypen die Anfragen beschleunigt. Ab einer bestimmten Größe des dadurch entstehenden Objekttypen wird sich die Leistung aber wieder verschlechtern. Dies belegt der Test an großen Objekttypen. Die Optimierung versucht, diese Grenze zu respektieren, indem ein Zusammenfassen nur empfohlen wird, wenn die Größe des entstehenden Objekttypen die Grenze, die die Routine für zu große Objekttypen setzt, nicht verletzt (6.6.9).
Auch der umgekehrte Fall kann eintreten: durch das Splitten eines Objekttypen müssen einige Anfragen nun über zwei Objekttypen und eine Verbindung laufen. Diese Anfragen werden wahrscheinlich länger brauchen als vorher. Aber auch dies Problem wird von der Optimierung beachtet, denn auf Kosten weniger Anfragen, die langsamer werden, werden viele Anfragen durch das Splitten beschleunigt (6.6.11), so daß insgesamt von einer Leistungsverbesserung gesprochen werden kann.

Kapitel 8

Schlußbemerkungen

In diesem Kapitel werden einige Angaben zum praktischen Teil der Arbeit aufgeführt. Es wird über Erfahrungen mit dem Einsatz der entwickelten Werkzeuge in LEU berichtet. Schließlich werden eine Zusammenfassung und ein Ausblick auf die weitere Arbeit an der Generierung und der Optimierung gegeben.

8.1 Daten zur Implementierung

Die in dieser Arbeit entworfenen Werkzeuge (Generierung und Optimierung) wurden auf folgendem System implementiert:

NFS–/Datenbank–Server	: Sparc Server 1000, 2 Prozessoren, 256 MB Speicher
Entwicklungs–/Applikations–Server	: Sparc Station 10, 1 Prozessor, 192 MB Speicher
Terminals	: Tektronix X–Terminal, 9 MB Speicher
Betriebssystem	: SunOS 4.1.3 (Solaris 1.1.3)
Graphische Oberfläche	: X11R4, Motif 1.1
Compiler	: Sun ANSI–C Compiler
Debugger	: xdbx, CodeCenter, Purify
Datenbank	: ORACLE 6, ORACLE 7

Die folgende Tabelle gibt einen Überblick über die Größe der erstellten Werkzeuge:

	Generierung	Optimierung
Anzahl Module	3	10
Anzahl Funktionen	27	37
Anzahl Regeln	—	51
Zeilen Code	13000	12700
Kilobyte Code	394	266

Der Programmcode liegt vor bei: Universität Dortmund
Fachbereich Informatik
Lehrstuhl für Softwaretechnologie
44221 Dortmund

8.2 Praxiserfahrung

Sowohl die Generierung als auch die Optimierung wurden in einer ersten Version implementiert. Die Generierung wurde schon in LEU integriert und läuft stabil. Die Optimierung erzielte gute Ergebnisse in den bisherigen Testläufen. Durch den praktischen Einsatz wurden einige Ansätze gefunden, die in einer Weiterentwicklung beider Werkzeuge berücksichtigt werden könnten. Diese Ansätze werden in den folgenden Abschnitten erläutert. Abschließend werden einige Laufzeittests dokumentiert.

8.2.1 Generierung

Die Generierung läuft auf Datenmodellebene. Dadurch kann es im Bereich der globalen Objekttypen zu Schwierigkeiten kommen. Die folgende Situation kann leicht entstehen:
Es gibt zwei Datenmodelle D1 und D2. Im Modell D1 wird ein Objekttyp O1 definiert und als global gekennzeichnet. Das Modell D2 importiert den Objekttypen O1. Außerdem wird in D2 ein weiterer Objekttyp O2 als Sohn von O1 definiert. Für das Datenmodell D2 wird nun die Generierung aufgerufen. Sie generiert nur die Tabelle für den Objekttypen O2, da O1 nicht in diesem Modell beheimatet ist. Anschließend sollen für O2 Objekte erfaßt

werden. Zu diesen Objekten gehören aber auch die Werte für die Felder des Vaters O1. Bei dem Versuch, diese Werte in die Tabelle von O1 einzutragen, meldet die Datenbank einen Fehler. Sie kann die Tabelle von O1 nicht finden, da sie noch nicht generiert wurde. Dies Beispiel zeigt, daß die Generierungen der verschiedenen Datenmodelle nicht unabhängig voneinander sind. Dennoch ist eine globale Generierung über alle Datenmodelle grundsätzlich nicht vorzuziehen. Bei kleinen Änderungen würde das einen zu großen Overhead bedeuten. Daher sollte in Zukunft ein Mechanismus gefunden werden, der Probleme, wie das oben beschriebene, löst. Zur Zeit lassen sich derartige Probleme nur lösen, indem der Benutzer auch noch das andere Datenmodell generiert.

8.2.2 Optimierung

Die Optimierung beschränkt sich bisher darauf, dem Benutzer bestimmte Änderungen vorzuschlagen. Ausführen muß er sie anschließend selbst. Hier können einige Routinen zur Automatisierung ansetzen, die die vorgeschlagenen Änderungen auf Wunsch des Benutzers selbständig durchführen. Im einzelnen ist das bei folgenden Punkten möglich:

Schlüsselfelder mit der Restriktion "kann": Die kann–Felder, die zu einem Schlüssel gehören, sind aufgrund der Ergebnisse der Optimierung bekannt. Eine Funktion kann also automatisch in den Systemtabellen die Restriktion dieser Felder auf *muß* setzen. Dies hat denselben Effekt, als wenn der Benutzer jedes Feld von Hand im Editor bearbeiten würde. Bei der nächsten Generierung werden diese Änderungen auch in der Datenbank umgesetzt. Probleme können dadurch nicht entstehen, da eventuelle leere Einträge von der Generierung automatisch durch Dummy–Werte ersetzt werden.

1:1–Verbindungen: Die Objekttypen, die zusammengelegt werden sollen, sind bekannt. Eine Funktion kann die Felder des einen Objekttypen automatisch an den anderen Objekttypen anhängen, indem sie die Systemtabellen entsprechend manipuliert. Ebenso kann sie den überflüssigen Objekttypen und die Verbindung löschen. Dies Verfahren ist aber nur möglich, wenn die Tabellen noch keine Objekte enthalten. Denn vorhandene Objekte können nicht einfach aus der Tabelle des einen Objekttypen in die des anderen übernommen werden, da nicht klar ist, wie sie zu den dort vorhandenen Objekten in Beziehung stehen.

Objekttypen splitten: Die Optimierung schlägt vor, wie die Felder des zu splittenden Objekttypen aufzuteilen sind. Anhand dieser Informationen kann eine Funktion die

Systemtabellen entsprechend manipulieren. Der Benutzer muß nur noch für den jeweils neuen Objekttypen einen Namen angeben. Das Splitten läßt sich sogar durchführen, wenn schon Objekte vorhanden sind. In diesem Fall darf allerdings nicht auf den nächsten Aufruf der Generierung gewartet werden, da sie nicht in der Lage wäre, die Objekte auf die beiden neuen Tabellen aufzuteilen. Dies muß also direkt geschehen. Zusätzlich muß eine Verbindung zwischen den beiden neuen Objekttypen eingerichtet werden. Für jeden ehemaligen Datensatz, der ja nun aus zwei Datensätzen in zwei Tabellen besteht, wird eine Verbindung eingetragen.

8.3 Zusammenfassung

In dieser Diplomarbeit wurden zwei Werkzeuge entwickelt, die LEU–ER–Modelle verbessern und in ein relationales Datenbanksystem abbilden.
Ein LEU–ER–Modell ist die graphische Repräsentation eines Datenbankschemas. Die Generierung wurde entwickelt, um das Modell in der Datenbank zu installieren. Dazu wurden Transformationsregeln aus der Literatur benutzt. Diese Regeln wurden teilweise erweitert und an LEU–Anforderungen angepaßt. Zum Beispiel erforderte die Umsetzung von Feldern des Typs "Liste" oder "Text" eine besondere Behandlung (vgl. Abschnitt 5.4).
Ein besonderer Aspekt der Generierung ist die Berücksichtigung der Schemaevolution. Es wurden Routinen entwickelt, die Änderungen des Datenmodells im Hinblick auf die Konsistenz des daraus zu generierenden Datenbankschemas prüfen. Eine Konvertierungstabelle wurde definiert, die festlegt, ob die Werte von Feldern, deren Typ geändert wird, bei der nächsten Generierung übernommen werden können.
Es stellte sich heraus, daß das Transaktionskonzept der Datenbank für Teile der Generierung nicht ausreicht. Daraufhin wurde eine Fehlerbehandlung entwickelt, die für kritische Funktionen ein eigenes Transaktionsmanagement realisiert.
Der Einsatz von LEU in der Industrie erfordert Performance bei der Bedienung. Deshalb sollte ein LEU–ER–Modell so definiert sein, daß die Antwortzeiten von Anfragen möglichst kurz sind. Die Optimierung wurde entwickelt, um den Benutzer dabei zu unterstützen. Die Recherche nach Softwareprodukten und Ansätzen in der Literatur, die Anregungen zum Entwurf der Optimierung liefern konnten, gestaltete sich recht schwierig. Die Optimierung wurde bisher unter zwei Gesichtspunkten gesehen: die Verbesserung der semantischen Eigenschaften eines Schemas (wie z. B. in CADDY) und die Verbesserung der Performance im laufenden System (z. B. Anfrageoptimierung). Der Bereich der performanceoptimierten

Modellierung wurde bisher wenig durchdrungen. Auch hier konnten Lösungen entwickelt werden, die insbesondere den erschwerenden Erfordernissen der Schemaevolution gerecht werden.
Zunächst wurden verschiedene Methoden zur Leistungsverbesserung aus der Literatur auf ihre Verwendbarkeit innerhalb der Optimierung geprüft. Es stellte sich heraus, daß viele Methoden erst während der Ausführungsphase sinnvoll angewandt werden können. Die Optimierung setzt aber schon in der Modellierungsphase an. Deshalb wurden nur die Ansätze zum Auffinden langer Zugriffspfade und zum Splitten von Objekttypen übernommen. Die weitere Funktionalität der Optimierung wurde neu entworfen. Es wurden einige Verbesserungen entwickelt, die direkt an spezielle Eigenschaften des LEU–ER–Modells anknüpfen. Dazu gehören das Auffinden von Objekttypen, die zu groß sind oder zu viele spezielle Felder besitzen, oder das Zusammenfassen von Objekttypen mit 1:1–Verbindungen.
Eine zusätzliche Aufgabe der Optimierung ist der Test auf Vollständigkeit eines LEU–ER–Modells. Dazu wurden Routinen entwickelt, die Objekttypen ohne Felder, Schlüssel, Verbindungen oder Zugriffspfade und Schlüsselfelder mit der Restriktion "kann" suchen.
Die Generierung und die Optimierung wurden in dieser Diplomarbeit nicht nur entworfen, sondern auch implementiert. Da beide Werkzeuge innerhalb von LEU eingesetzt werden sollen, war die Performance ein wichtiges Kriterium für die Entwicklung. Verschiedene Tests haben gezeigt, daß beide Werkzeuge eine gutes Laufzeitverhalten besitzen. Durch den praktischen Einsatz in LEU konnten darüber hinaus einige Erfahrungen im Bezug auf mögliche Verbesserungen und Erweiterungen gesammelt werden. So besteht zum Beispiel die Möglichkeit, bestimmte Änderungen, die die Optimierung vorschlägt, automatisch durchzuführen.

8.4 Zukünftige Arbeit

Da die Generierung und die Optimierung innerhalb von LEU eingesetzt werden, bietet sich die Möglichkeit, Verbesserungen und Erweiterungen vorzunehmen. Für die Generierung wird ein Konzept entwickelt, welches die Probleme mit globalen Objekttypen löst (vgl. Abschnitt 8.2.1). Zusätzlich wird der Code an einigen Stellen überarbeitet, so daß eine Verbesserung der Laufzeit erzielt wird. Dies wurde z. B. bei der Generierung geänderter Objekttypen schon erwähnt (vgl. Abschnitt 5.7.6). Die weitere Arbeit an der Optimierung betrifft zunächst die Integration in LEU. Anschließend werden Routinen zur Automatisierung bestimmter Veränderungen des Modells entwickelt (vgl. Abschnitt 8.2.2).

Anhang A

Testberichte

Dies Kapitel enthält einige Testberichte zu den Funktionen der Generierung und der Optimierung. Die verschiedenen Tests einer Funktion bauen aufeinander auf, sofern zur Vorbereitung nicht ausdrücklich gesagt wird, daß die vorigen Eingaben wieder gelöscht werden müssen.

A.1 Test der Generierung

Alle Tests werden auf dem Datenmodell *Bibliothek* mit dem Surrogat *111* durchgeführt.

Funktion: DBgen_TestFieldDefinition
Voraussetzung: Das Datenmodell enthält einen Objekttypen *Buch*.
Anmerkung: Die Felddefinitionen werden der Lesbarkeit wegen in der Form *Def = (Bezeichner, Typ, Länge, Nachkommastellen, Schlüssel)* angegeben.
Testfälle:

1. Die Felddefinition wurde nicht geändert.

   Vorbereitung: Für das *Buch* wird ein Feld *Titel* mit folgender Definition eingetragen: Def = (Titel, String, 70, 0, nein). Es wird generiert.

   Eingabe: NewDef = (Titel, String, 70, 0, nein)

   Ausgabe: NO_ERROR, ImpossibleChange = FALSE, FDChanged = FALSE, DataLoss = 0

   Ergebnis: Die Funktion arbeitet korrekt.

2. Die Felddefinition wurde ohne Datenverlust geändert.

   Vorbereitung: Für das *Buch* wird ein Feld *Preis* mit folgender Definition eingetragen: Def = (Preis, Integer, 3, 0, nein). Es wird generiert.

   Eingabe: NewDef = (Preis, Real, 6, 2, nein)

   Ausgabe: NO_ERROR, ImpossibleChange = FALSE, FDChanged = TRUE, DataLoss = 0

   Ergebnis: Die Funktion arbeitet korrekt.

3. Die Felddefinition wurde mit komplettem Datenverlust geändert.

   Vorbereitung: keine weitere

   Eingabe: NewDef = (Titel, String, 35, 0, nein)

   Ausgabe: NO_ERROR, ImpossibleChange = FALSE, FDChanged = TRUE, DataLoss = 2

   Ergebnis: Die Funktion arbeitet korrekt.

4. Die Felddefinition wurde mit teilweisem Datenverlust geändert.

   Vorbereitung: Für das *Buch* wird ein Feld *Autoren* mit folgender Definition eingetragen: Def = (Autoren, Liste(String), 30, 0, nein). Es wird generiert.

   Eingabe: NewDef = (Autoren, String, 70, 0, nein)

   Ausgabe: NO_ERROR, ImpossibleChange = FALSE, FDChanged = TRUE, DataLoss = 1

   Ergebnis: Die Funktion arbeitet korrekt.

5. Die Änderung der Felddefinition ist verboten.

   Vorbereitung: Für das *Buch* wird ein Feld *ISBN* mit folgender Definition eingetragen: Def = (ISBN, String, 30, 0, ja). Es wird generiert.

   Eingabe: NewDef = (ISBN, String, 20, 0, ja)

   Ausgabe: NO_ERROR, ImpossibleChange = TRUE, FDChanged = TRUE, DataLoss = 2

   Ergebnis: Die Funktion arbeitet korrekt.

Funktion: DBgen_TestFieldType
Der Test verläuft analog zum Test von DBgen_TestFieldDefinition. Statt einer Felddefinition wird hier ein globaler oder lokaler Feldtyp getestet.

Funktion: DBgen_TestNewFieldWithKey
Voraussetzung: Das Datenmodell enthält einen Objekttypen *Buch* mit dem Surrogat *222* und einem Feld *Titel*. Es wird generiert.
Testfälle:

1. Ein neues Feld darf kein Schlüsselfeld sein.

   Vorbereitung: Objekte werden eingegeben. Für das *Buch* wird ein neues Schlüsselfeld *Schlüssel* eingetragen.

   Eingabe: OT_Key = 222

   Ausgabe: NO_ERROR, ImpossibleChange = TRUE

   Ergebnis: Die Funktion arbeitet korrekt. Der Feld–Editor weist daraufhin das Feld *Schlüssel* zurück.

2. Ein neues Feld darf ein Schlüsselfeld sein.

   Vorbereitung: Die Objekte von *Buch* werden gelöscht. Wiederum wird ein neues Schlüsselfeld *Schlüssel* eingetragen.

   Eingabe: OT_Key = 222

   Ausgabe: NO_ERROR, ImpossibleChange = FALSE

   Ergebnis: Die Funktion arbeitet korrekt.

Funktion: DBgen_TestDeleteFieldWithKey
Voraussetzung: Das Datenmodell enthält einen Objekttypen *Buch*. Für *Buch* werden zwei Schlüsselfelder eingetragen: *Titel* mit dem Surrogat *333* und *Schlüssel* mit dem Surrogat *444*. Es wird generiert.
Testfälle:

1. Ein Schlüsselfeld darf nicht gelöscht werden.

   Vorbereitung: Objekte werden eingegeben. Das Schlüsselfeld *Titel* wird gelöscht.

   Eingabe: FD_Key = 333

Ausgabe: NO_ERROR, ImpossibleChange = TRUE

Ergebnis: Die Funktion arbeitet korrekt. Der Feld–Editor weist daraufhin die Löschoperation zurück.

2. Ein Schlüsselfeld darf gelöscht werden.

   Vorbereitung: Die Objekte von *Buch* werden gelöscht. Wiederum wird das Schlüsselfeld *Titel* gelöscht.

   Eingabe: FD_Key = 333

   Ausgabe: NO_ERROR, ImpossibleChange = FALSE

   Ergebnis: Die Funktion arbeitet korrekt.

Funktion: DBgen_TestOT
Voraussetzung: Das Datenmodell ist leer.
Anmerkung: Der Ausgabeparameter *Changed* ist eine Struktur und wird wie folgt angegeben: *Changed = (Status, Vererbung, Felder, Anzahl_Felder, Schlüssel, Historisierung)*. Der *Status* kann vier Werte annehmen: *neu*, *gleich*, *geändert_mit_Datenverlust* und *geändert_ohne_Datenverlust*. Die restlichen Elemente von *Changed* sind Boolean–Werte, die angeben, ob im entsprechenden Bereich eine Änderung gefunden wurde.
Testfälle:

1. Der Objekttyp ist neu.

   Vorbereitung: Ein Objekttyp *Buch* mit dem Surrogat *222* wird angelegt. Ein Feld *Autor* wird eingetragen.

   Eingabe: OT_Key = 222

   Ausgabe: NO_ERROR, Changed = (neu, FALSE, FALSE, FALSE, FALSE, FALSE)

   Ergebnis: Die Funktion arbeitet korrekt.

2. Der Objekttyp ist gleich geblieben.

   Vorbereitung: Es wird generiert. Dadurch existiert nun eine Datenbanktabelle für das *Buch*.

   Eingabe: OT_Key = 222

   Ausgabe: NO_ERROR, Changed = (gleich, FALSE, FALSE, FALSE, FALSE, FALSE)

   Ergebnis: Die Funktion arbeitet korrekt.

3. Der Objekttyp ist ohne Datenverlust geändert worden.

   Vorbereitung: Für das *Buch* wird ein weiteres Feld *Titel* eingetragen, das historisiert werden soll.

   Eingabe: OT_Key = 222

   Ausgabe: NO_ERROR, Changed = (geändert_ohne_Datenverlust, FALSE, FALSE, TRUE, FALSE, TRUE)

   Ergebnis: Die Funktion arbeitet korrekt.

4. Der Objekttyp ist mit Datenverlust geändert worden.

   Vorbereitung: Es wird generiert. Das Feld *Titel* wird wieder gelöscht.

   Eingabe: OT_Key = 222

   Ausgabe: NO_ERROR, Changed = (geändert_mit_Datenverlust, FALSE, FALSE, TRUE, FALSE, TRUE)

   Ergebnis: Die Funktion arbeitet korrekt.

Funktion: DBgen_GenerateConnections
Voraussetzung: Das Datenmodell *111* enthält zwei Objekttypen *Buch* und *Regal*.
Testfälle:

1. Das angegebene Datenmodell existiert nicht.

   Vorbereitung: keine

   Eingabe: DM_Key = 999

   Ausgabe: DB_NOT_EXIST

   Ergebnis: Die Funktion arbeitet korrekt.

2. Eine neue Verbindung wird generiert.

   Vorbereitung: Die Objekttypen *Buch* und *Regal* werden durch die Relation *Standort* verbunden.

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR

   Ergebnis: Eine Datenbanktabelle für *Standort* wurde angelegt. Die Funktion arbeitet korrekt.

3. Eine alte Verbindung wird generiert.

   Vorbereitung: keine weitere

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR

   Ergebnis: Da die Datenbanktabelle für *Standort* schon existiert, ändert sich nichts. Die Funktion arbeitet korrekt.

Funktion: DBgen_GenerateNewOT
Voraussetzung: Das Datenmodell enthält einen Objekttypen *Buch* mit dem Surrogat *222* und einem Feld *Titel*. Es wird generiert.
Anmerkung: Der Eingabeparameter *Changed* ist eine Struktur und wird wie folgt angegeben: *Changed = (Status, Vererbung, Felder, Anzahl_Felder, Schlüssel, Historisierung)*. Der *Status* kann vier Werte annehmen: *neu*, *gleich*, *geändert_mit_Datenverlust* und *geändert_ohne_Datenverlust*. Die restlichen Elemente von *Changed* sind Boolean–Werte, die angeben, ob im entsprechenden Bereich eine Änderung gefunden wurde. Die Belegung von *Changed* wurde vorher von der oben getesteten Funktion DBgen_TestOT vorgenommen.
Testfälle:

1. Der Objekttyp ist nicht neu.

   Vorbereitung: keine

   Eingabe: OT_Key = 222, Changed = (gleich, FALSE, FALSE, FALSE, FALSE, FALSE)

   Ausgabe: DB_WRONG_FUNCTION

   Ergebnis: Die Funktion arbeitet korrekt.

2. Der Objekttyp ist neu.

   Vorbereitung: Ein zweiter Objekttyp *Regal* mit einem Feld *Nummer* wird definiert.

   Eingabe: OT_Key = 222, Changed = (neu, FALSE, FALSE, FALSE, FALSE, FALSE)

   Ausgabe: NO_ERROR

   Ergebnis: Die Datenbanktabelle für *Regal* wurde angelegt. Die Funktion arbeitet korrekt.

Funktion: DBgen_GenerateChangedOT
Voraussetzung: Das Datenmodell ist leer.
Anmerkung: Der Eingabeparameter *Changed* ist eine Struktur und wird wie folgt angegeben: *Changed = (Status, Vererbung, Felder, Anzahl_Felder, Schlüssel, Historisierung).* Der *Status* kann vier Werte annehmen: *neu*, *gleich*, *geändert_mit_Datenverlust* und *geändert_ohne_Datenverlust*. Die restlichen Elemente von *Changed* sind Boolean–Werte, die angeben, ob im entsprechenden Bereich eine Änderung gefunden wurde. Die Belegung von *Changed* wurde vorher von der oben getesteten Funktion DBgen_TestOT vorgenommen.
Testfälle:

1. Der Objekttyp ist nicht geändert.

   Vorbereitung: Ein Objekttyp *Buch* mit dem Surrogat *222* und einem Feld *Titel* wird definiert.

   Eingabe: OT_Key = 222, Changed = (neu, FALSE, FALSE, FALSE, FALSE, FALSE)

   Ausgabe: DB_WRONG_FUNCTION

   Ergebnis: Die Funktion arbeitet korrekt.

2. Der Objekttyp ist geändert.

   Vorbereitung: Es wird generiert. Dann wird für *Buch* ein weiteres Feld *Autor* eingegeben.

   Eingabe: OT_Key = 222, Changed = (geändert_ohne_Datenverlust, FALSE, FALSE, TRUE, FALSE, FALSE)

   Ausgabe: NO_ERROR

   Ergebnis: Die Datenbanktabelle für *Buch* wurde erweitert. Die Funktion arbeitet korrekt.

3. Der Objekttyp ist geändert. Es gibt einen Timeout bei der Ausführung.

   Vorbereitung: Für *Buch* wird ein weiteres Feld *ISBN* eingegeben. Eine zweite Transaktion auf der Datenbank wird gestartet. Die Tabelle von *Buch* wird selektiert. (Dadurch wird die Tabelle gesperrt. Sie kann insbesondere nicht gelöscht werden.)

   Eingabe: OT_Key = 222, Changed = (geändert_ohne_Datenverlust, FALSE, FALSE, TRUE, FALSE, FALSE)

Ausgabe: DB_TIMEOUT

Ergebnis: Die Funktion arbeitet korrekt. (Sie versucht, die Tabelle von *Buch* zu löschen und neu anzulegen. Das Löschen ist aber nicht möglich.)

Funktion: DBgen_ResetOT
Voraussetzung: Das Datenmodell ist leer.
Anmerkung: Der Eingabeparameter *Changed* ist eine Struktur und wird wie folgt angegeben: *Changed = (Status, Vererbung, Felder, Anzahl_Felder, Schlüssel, Historisierung)*. Der *Status* kann vier Werte annehmen: *neu*, *gleich*, *geändert_mit_Datenverlust* und *geändert_ohne_Datenverlust*. Die restlichen Elemente von *Changed* sind Boolean–Werte, die angeben, ob im entsprechenden Bereich eine Änderung gefunden wurde. Die Belegung von *Changed* wurde vorher von der oben getesteten Funktion DBgen_TestOT vorgenommen.
Testfälle:

1. Der Objekttyp ist nicht geändert.

   Vorbereitung: Ein Objekttyp *Buch* mit dem Surrogat *222* und einem Feld *Titel* wird definiert.

   Eingabe: OT_Key = 222, Changed = (neu, FALSE, FALSE, FALSE, FALSE, FALSE)

   Ausgabe: DB_WRONG_FUNCTION

   Ergebnis: Die Funktion arbeitet korrekt.

2. Der Objekttyp ist geändert.

   Vorbereitung: Es wird generiert. Dann wird für *Buch* ein weiteres Feld *Autor* eingegeben.

   Eingabe: OT_Key = 222, Changed = (geändert_ohne_Datenverlust, FALSE, FALSE, TRUE, FALSE, FALSE)

   Ausgabe: NO_ERROR

   Ergebnis: Die Datenbanktabelle für *Buch* wurde nicht verändert. Das Feld *Autor* wurde wieder aus der Definition gelöscht. Die Funktion arbeitet korrekt.

3. Der Objekttyp ist geändert. Ein Zurücksetzen ist nicht möglich.

Vorbereitung: Ein zweiter Objekttyp *Schriftstück* wird angelegt. *Buch* wird als Sohn von *Schriftstück* definiert (Vererbung). Es wird generiert. Dann werden die Vererbung und das *Schriftstück* wieder gelöscht.

Eingabe: OT_Key = 222, Changed = (geändert_mit_Datenverlust, TRUE, FALSE, FALSE, FALSE, FALSE)

Ausgabe: DB_RESET_IMPOSSIBLE

Ergebnis: Die Vererbung wurde nicht wieder hergestellt, da der Vater nicht mehr existiert. Die Funktion arbeitet korrekt.

Funktion: DBgen_DropTables
Voraussetzung: Das Datenmodell *111* enthält einen Objekttypen *Buch* mit einem Feld *Titel*. Es wird generiert.
Testfälle:

1. Ein Aufruf.

   Vorbereitung: Das Buch wird gelöscht.

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR

   Ergebnis: Die Datenbanktabelle von *Buch* wurde gelöscht. Die Funktion arbeitet korrekt.

Funktion: DBgen_ClearGenerate
Voraussetzung: Das Datenmodell *111* enthält einen Objekttypen *Buch* mit einem Feld *Titel* vom Typ *Liste*. Es wird generiert. Bei der Generierung von *Buch* wird ein Fehler erzeugt, und zwar nach dem Anlegen der Objekttyp–Tabelle und vor dem Anlegen der Listen–Tabelle.
Testfälle:

1. Ein Aufruf.

   Vorbereitung: keine

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR

   Ergebnis: Die Objekttyp–Tabelle von *Buch* wurde wieder gelöscht. Die Funktion arbeitet korrekt.

A.2 Test der Optimierung

Alle Tests werden auf dem Datenmodell *Bibliothek* mit dem Surrogat *111* durchgeführt.

Funktion: DBopt_LonelyOT
Voraussetzung: Das Datenmodell ist leer.
Testfälle:

1. Im Datenmodell existieren nur Objekttypen mit Verbindung.

   Vorbereitung: Zwei Objekttypen *Buch* und *Regal* werden angelegt und durch eine Relation *Standort* verbunden.

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR, Found = FALSE

   Ergebnis: Die Funktion arbeitet korrekt.

2. Im Datenmodell existieren Objekttypen ohne Verbindung.

   Vorbereitung: Ein dritter Objekttyp *Leser* wird angelegt.

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR, Found = TRUE

   Ergebnis: Der Objekttyp *Leser* wurde in die Analysis–Tabelle eingetragen. Die Funktion arbeitet korrekt.

Funktion: DBopt_EmptyOT
Der Test verläuft analog zum Test von DBopt_LonelyOT.

Funktion: DBopt_KeyLessOT
Der Test verläuft analog zum Test von DBopt_LonelyOT.

Funktion: DBopt_NullValues
Der Test verläuft analog zum Test von DBopt_LonelyOT.

Funktion: DBopt_OTWithoutAccess
Der Test verläuft analog zum Test von DBopt_LonelyOT.

Funktion: DBopt_SpecialFields
Der Test verläuft analog zum Test von DBopt_LonelyOT.

Funktion: DBopt_OTWithManyBlocks
Der Test verläuft analog zum Test von DBopt_LonelyOT.

Funktion: DBopt_OneToOne
Voraussetzung: Das Datenmodell ist leer.
Testfälle:

1. Im Datenmodell existieren keine 1:1–Verbindungen.

   Vorbereitung: Zwei Objekttypen *Buch* und *Regal* werden angelegt und durch eine n:1–Relation *Standort* verbunden.

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR, Found = FALSE

   Ergebnis: Die Funktion arbeitet korrekt.

2. Im Datenmodell existieren 1:1–Verbindungen zwischen lokalen Objekttypen.

   Vorbereitung: Die Relation *Standort* wird in 1:1 geändert.

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR, Found = TRUE

   Ergebnis: Die Relation *Standort* wurde in die Analysis–Tabelle eingetragen. Die Funktion arbeitet korrekt.

3. Im Datenmodell existieren 1:1–Verbindungen zwischen globalen Objekttypen.

   Vorbereitung: Der Objekttyp *Buch* wird exportiert.

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR, Found = FALSE

   Ergebnis: Die Relation *Standort* wurde wieder aus der Analysis–Tabelle gelöscht. Die Funktion arbeitet korrekt.

Funktion: DBopt_TooLongPathes
Der Test verläuft analog zum Test von DBopt_LonelyOT.

Funktion: DBopt_OTToSplit
Voraussetzung: Das Datenmodell enthält den Objekttypen *Leser* mit den Feldern *Name*, *Vorname*, *Straße* und *Ort*.
Testfälle:

1. Im Datenmodell existieren Objekttypen, die gesplittet werden sollten, da ihre Felder in zwei Gruppen geteilt werden können, die jeweils von verschiedenen Zugriffspfaden angesprochen werden.

   Vorbereitung: Ein Zugriffspfad *LeserName* mit dem Gewicht *8* wird angelegt. Er benutzt die Felder *Name* und *Vorname*. Ein Zugriffspfad *LeserAdresse* mit dem Gewicht *1* wird angelegt. Er benutzt die Felder *Straße* und *Ort*.

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR, Found = TRUE

   Ergebnis: Der Objekttyp *Leser* wurde in die Analysis– und in die Split–Tabelle eingetragen. Er wird gesplittet in (*Name*, *Vorname*) und (*Straße*, *Ort*). Die Funktion arbeitet korrekt.

2. Im Datenmodell existieren Objekttypen, die gesplittet werden sollten, da einige Felder sehr häufig und andere Felder sehr selten zugegriffen werden.

   Vorbereitung: Ein dritter Zugriffspfad *LeserGesamt* mit dem Gewicht *2* wird angelegt. Er benutzt alle Felder des *Lesers*.

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR, Found = TRUE

   Ergebnis: Der Objekttyp *Leser* wurde aus der Analysis– und der Split–Tabelle gelöscht und in beide wieder eingetragen. Er wird gesplittet in (*Name*, *Vorname*) und (*Straße*, *Ort*). Die Funktion arbeitet korrekt.

3. Im Datenmodell existieren keine zu splittenden Objekttypen.

   Vorbereitung: Das Gewicht des Zugriffspfades *LeserName* wird auf *3* reduziert.

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR, Found = FALSE

   Ergebnis: Der Objekttyp *Leser* wurde aus der Analysis– und der Split–Tabelle gelöscht. Die Funktion arbeitet korrekt.

Funktion: DBopt_Optimize
Voraussetzung: Es sind beliebige Testdaten für alle bisher beschriebenen Funktionen eingegeben worden.
Testfälle:

1. Die Funktion ruft alle bisher beschriebenen Funktionen auf.

   Vorbereitung: keine

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR, ToOptimize = <*entsprechend der Ausgaben der aufgerufenen Funktionen*>

   Ergebnis: Die Funktion arbeitet korrekt.

Funktion: DBopt_WhichOptimize
Voraussetzung: Es sind beliebige Testdaten für alle bisher beschriebenen Funktionen eingegeben worden.
Testfälle:

1. Es wurde noch keine Optimierung durchgeführt.

   Vorbereitung: keine

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR, ToOptimize = (FALSE, ..., FALSE)

   Ergebnis: Die Funktion arbeitet korrekt.

2. Es wurde schon eine Optimierung durchgeführt.

   Vorbereitung: keine

   Eingabe: DM_Key = 111

   Ausgabe: NO_ERROR, ToOptimize = <*entsprechend der Einträge in die Analysis–Tabelle*>

   Ergebnis: Die Funktion arbeitet korrekt.

Literaturverzeichnis

[AJ91] V. Ambriola, M. L. Jaccheri. *Definition and Enactment of Oikos Software Process Entities*. First European Workshop on Software Process Modelling, Milano, 1991.

[Ame92] *American National Standard for Information Systems – Database Language – SQL*. American National Standards Institute, New York, 1992. (ANSI Standard X3.135–1992).

[BFG91] S. Bandinelli, A. Fuggetta, C. Ghezzi. *Software Processes as Real Time Systems: a Case Study using High–Level Petri Nets*. First European Workshop on Software Process Modelling, Milano, 1991.

[BKK+92] R. Budde, K. Kautz, K. Kuhlenkamp, H. Züllighoven. *Prototyping: An Approach to Evolutionary System Development*. Springer–Verlag, 1992.

[BLG82] P. Bertaina, A. Di Leva, P. Giolito. *EASYMAP an automatic Tool for logical Database Design in the DATAID Project*. Interner Bericht der Universität Turin, 1982.

[BLG83] P. Bertaina, A. Di Leva, P. Giolito. *Logical Design in CODASYL and Relational Environment*. In S. Ceri, Hrsg., Methodology and Tools for Database Design, S. 85–117. North–Holland, 1983.

[Cad76] J. M. Cadiou. *On semantic Issues in the Relational Model of Data*. In Proceedings of the 5th Symposium on Mathematical Foundations of Computer Science, S. 23–38. Springer–Verlag, 1976.

[Che76] P. P. Chen. *The Entity–Relationship Model: Toward a Unified View of Data*. In ACM Transactions on Database Systems, S. 9–36, 1976.

[Che77] P. P. Chen. *The Entity–Relationship Approach to Logical Database Design.* Q.E.D. Information Science, 1977.

[Cod70] E. F. Codd. *A Relational Model for Large Shared Data Banks.* In Communications of the ACM, 13: S. 377–387, 1970.

[Cod72] E. F. Codd. *Further Normalization of the Data Base Relational Model.* In R. Rustin, Hrsg., Data Base Systems (Courand Computer Science Symposia Series). Prentice Hall, 1972.

[Cod74] E. F. Codd. *Recent Investigations into Relational Data Base Systems.* In Proceedings of the IFIP Congress, 1974.

[Cod79] E. F. Codd. *Extending the Database Relational Model to Capture More Meaning*, ACM Transactions on Database Systems, 4: S. 397–434, 1979.

[Dat86] C. J. Date. *An Introduction to Database Systems*, (Systems Programming Series), 4: S. 361–409. Addison–Wesley Publishing Company, 1986.

[Dat90] C. J. Date. *An Introduction to Database Systems, Volume 1*, (Systems Programming Series), 5. Addison–Wesley Publishing Company, 1990.

[Des90] B. C. Desai. *An Introduction to Database Systems.* West Publishing Company, 1990.

[DGZ94] G. Dinkhoff, V. Gruhn, M. Zielonka. *Praxisorientierte Aspekte der LEU–Datenmodellierung.* In EMISA Forum, 1: S. 15–27, 1994.

[Dow91] M. Dowson, Hrsg. *Proceedings of the 1st International Software Process Conference – Manufactoring Complex Systems*, Redondo Beach, California, USA, 1991.

[EG91] W. Emmerich, V. Gruhn. *FUNSOFT Nets: A Petri–Net based Software Process Modeling Language.* In Proceedings of the 6th International Workshop on Software Specification and Design, Como, Italien, 1991.

[EGH+90] G. Engels, M. Gogolla, U. Hohenstein, K. Hülsmann, P. Löhr–Richter, G. Saake, H.–D. Ehrich. *Conceptual Modelling of Database Applications Using an Extended ER–Model.* Technischer Bericht der TU Braunschweig, Nr. 90–05, 1990.

[EHH+90] G. Engels, U. Hohenstein, K. Hülsmann, P. Löhr–Richter, H.–D. Ehrich. *CADDY: Computer–Aided Design of Non–Standard Databases.* In N. Madhavji, H. Weber, W. Schäfer, Hrsg., Internal Conference on System Development Environments & Factories. Pitman Publishing, 1990.

[EL91] G. Engels, P. Löhr–Richter. *Incremental Design of Conceptual Donceptual Schemata with CADDY.* Universität Leiden, 1991.

[EN89] R. Elmasri, S. B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, 1989.

[ES92] G. Erdl, H. G. Schönecker. *Geschäftsprozeßmanagement – Vorgangssteuerung und integrierte Vorgangsbearbeitung.* FBO – Fachverlag für Büro und Organisationstechnik, 1992.

[Fag77] R. Fagin. *Multivalued Dependencies and a New Normal Form for Relational Databases.* In ACM Transactions on Database Systems, 1977.

[Fag79] R. Fagin. *Normal Forms and Relational Database Operators.* In ACM SIGMOD International Conference on Management of Data, 1979.

[FH89] C. C. Fleming, B. von Halle. *Handbook of Relational Database Design.* Addison–Wesley Publishing Company, 1989.

[GLD+93] V. Gruhn, K. Lichtinghagen, J. Dickmann, A. Saalmann, M. Zielonka, R. Jegelka, U. Schindler. *Die LION–Entwicklungs–Umgebung für die Bau– und Wohnungswirtschaft.* LION Gesellschaft für Systementwicklung mbH, 1994.

[Gru91] V. Gruhn. *Validation and Verification of Software Process Models.* In A. Endres, H. Weber, Hrsg., Proceedings of the European Symposium on Software Development Environments and CASE Technology, (Lecture Notes in Comupter Science 509), Springer–Verlag, 1991.

[Gru93a] V. Gruhn. *Entwicklung von Informationssystemen in der LION–Entwicklungsumgebung.* In G. Scheschonk, W. Reisig, Petri–Netze im Einsatz für Entwurf und Entwicklung von Informationssystemen, (Informatik aktuell), S. 31–45. Springer–Verlag, 1993.

[Gru93b] V. Gruhn. *Software Process Simulation in MELMAC*. Systems Analysis – Modelling – Simulation, 11, 1993.

[Gru94] V. Gruhn. *Communication Support in a Process–Centered Software Engineering Environment*. In Proceedings of the 9th International Software Process Workshop, Airlie, US, 1994.

[HE91] U. Hohenstein, G. Engels. *SQL/EER — Syntax und Semantics of an Entity–Relationship–Based Query Language*. Informatik–Berichte 91–02 der TU Braunschweig, 1991.

[HJS93] T. Hartmann, R. Jungclaus, G. Saake. *Spezifikation von Informationssystemen als Objektsysteme: Das TROLL–Projekt*. In EMISA Forum, 1, 1993.

[HOT76] P. Hall, J. Owlett, S. Todd. *Relations and Entities*. In Modelling in Data Base Management Systems. North–Holland, 1976.

[HSO90] D. Heimbiger, S. M. Sutton, L. Osterweil. *Managing Change in Process–Centered Environments*. In Proceedings of the 4th ACM/SIGSOFT Symposium on Software Development Environments, 1990.

[ISA92] *ISA Dialog Manager. Übersicht. Dokumentation. Release A.02.00* ISA GmbH, Stuttgart, 1992.

[JMS+92] M. Jarke, J. Mylopoulus, J. W. Schmidt, Y. Vassiliou. *DAIDA – An Environment for Evolving Information Systems*. In ACM Transactions in Information Systems, 10(1): S. 1–50, 1992.

[KFP88] G. E. Kaiser, P. H. Feiler, S. S. Popovich. *Intelligent Assistance for Software Development and Maintenance*. IEEE Software, S. 40–49, 1988.

[Mei92] A. Meier. *Relationale Datenbanken*. Springer–Verlag, 1992.

[ORA90a] *Oracle RDBMS Database Administrator's Guide Version 6.0*, 2. Oracle Corporation, 1990.

[ORA90b] *SQL Language Referenzhandbuch Version 6.0*, 2. Oracle Corporation, 1990.

[ORA90c] *Oracle RDBMS Performance Tuning Guide Version 6.0*, 2. Oracle Corporation, 1990.

[Per91] D. E. Perry, Hrsg. *Proceedigs of the 7th International Software Process Workshop*, Yountville, California, USA, 1991.

[PSW92] B. Peuschel, W. Schäfer, S. Wolf. *A Knowledge–Based Software Development Environment*. In International Journal of Software Engineering and Knowledge Engineering, 2: S. 79–106, 1992.

[Rei86] W. Reisig. *Petrinetze*. Springer–Verlag, 1986.

[RM90] C. Rautenstrauch, M. Moazzami. *Effiziente Systementwicklung mit Oracle*. Addison–Wesley Publishing Company, 1990.

[SNH+87] G. Saake, L. Neugebauer, U. Hohenstein, H.-D. Ehrich. *Konzepte und Werkzeuge für eine Datenbank–Entwurfsumgebung*. Institut für Programmiersprachen und Informationssysteme der TU Braunschweig, 1987.

[SS77] J. M. Smith, D. C. P. Smith. *Database Abstractions: Aggregation and Generalization*. In ACM Transactions on Database Systems, 2: S. 105–133, 1977.

[Stu93] G. Stürner. *Oracle 7 — Die verteilte Semantische Datenbank*. dbms publishing, 1993.

[Ullm88] J. D. Ullman. *Principles of Database and Knowledge–Base Systems, Volume 1: Classical Database Systems*. Computer Science Press, 1988.

[Vos87] G. Vossen. *Datenmodelle, Datenbanksprachen, und Datenbank–Management–Systeme*. Addison–Wesley Publishing Company, 1987.

[War89] B. Warboys. *The IPSE 2.5 Project: Process Modelling as the Basis for a Support Environment*. In First International Conference on System Development Environments and Factories, Berlin, 1989.