Konzept

Diplomarbeit zum Arbeitstitel Open DB Designer

dataMagus
Monika Schwitter & Jürg Zgraggen, 8Ibb-a

Revision

Version	Datum	Kommentar	Autor
1.0	16.10.2007	Erste Version erstellt	MOS
1.1	19.10.2007	Kapitel Vorgehen	JAZ/MOS
1.2	20.10.2007	Kapitel Grundlagen, Anforderungen	JAZ/MOS
1.3	21.10.2007	Anforderungen, Analysen	JAZ/MOS
1.4	24.10.2007	Definition der Elemente und Umsetzungen	MOS
1.5	25.10.2007	Generationsprozesse	JAZ
1.6	27.10.2007	Datentypen, Umsetzung	JAZ/MOS
1.7	31.10.2007	Anpassung aufgrund Sitzung	MOS
2.0	05.11.2007	Beschreibung der Architektur	MOS
2.1	09.11.2007	Beschreibung zur persistenten Seicherung	JAZ/MOS
2.2	15.11.2007	Definition der Generator-PlugIns	MOS
2.3	16.11.2007	Definition der Notation-PlugIns	JAZ
2.4	19.11.2007	Umschreiben der Benutzeroberfläche	JAZ
2.5	20.11.2007	Fehlerkorrektur	JAZ/MOS
2.6	20.11.2007	Testergebnisse dokumentieren	JAZ/MOS
2.7	21.11.2007	Schreiben des Fazits und Ausblicks	JAZ/MOS
2.8	21.11.2007	Fehlerkorrektur	JAZ/MOS
3.0	22.11.2007	Letzte Änderungen und Aktualisierungen	JAZ/MOS

Inhaltsverzeichnis

1 Einleitung und Ziel

1.1 Projektübersicht

Die Modellierung von Datenbanken ist ein zentrales Element bei der Entwicklung von Anwendungen. Daher spielt sie auch im Unterricht eine grosse Rolle. Es gibt verschiedene Modellierungswerkzeuge, die sich sehr oft auf das relationale Datenmodell konzentrieren. Häufig wird aus den Tools direkt Code für die Erstellung der Datenbanken erzeugt. In dieser Arbeit wird ein Open-Source-Datenbank-Modellierwerkzeug entwickelt, das gut im Unterricht eingesetzt werden kann und leicht erweiterbar ist. Abbildung 1 zeigt eine schematische Übersicht des Projekts.

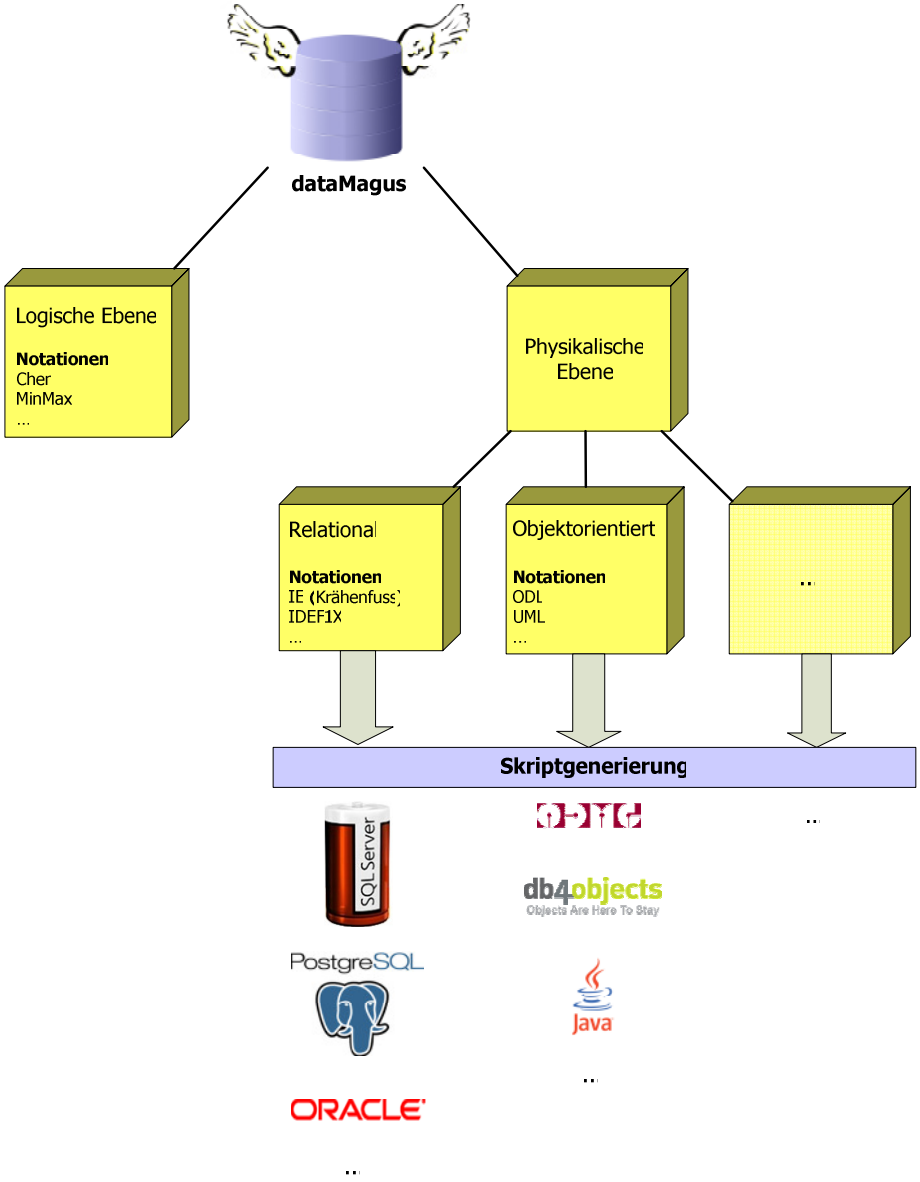


Abbildung 1 – Projektübersicht [14] [15] [16] [17] [18] [19]

Der Open DB Designer (später dataMagus genannt) ist eine Software, die wie eingangs erwähnt zur Unterrichtsoptimierung dienen soll und von Studierenden benutzt wird. Mit dataMagus sollen Datenbankschemas unabhängig von der Notation und dem Zieldialekt modelliert werden können. Das Schema wird auf einer logischen Sicht erstellt, welches dann in entsprechende physikalische Schemas umgesetzt wird. Der Code zur Erstellung des

physikalischen Schemas wird dann durch ein generisches Generatorinterface erzeugt. Bei relationalen Datenbanken ist dies SQL in verschiedenen Dialekten, bei objektorientierten Datenbanken handelt es sich um ODL oder auch Klassengenerierung für die .NET-Plattform oder für Java. Wir versuchen, die einzelnen Projektkomponenten flexibel und möglichst unabhängig voneinander zu gestalten.

1.1.1 Aufgabenstellung

Die Aufgabenstellung umfasst folgende Zielsetzungen:

- Definition der zentralen Elemente für den Open DB Designer
- Definition eines generischen Konzeptes, welches ermöglicht:
  - die vorhandenen Modellierungselemente zu erweitern
  - andere Modelle einzubeziehen (z.B. XML, objektrelational)
  - verschiedene Datenbank-Dialekte einzubinden.
- Definition der Schnittstellen, um neue Elemente und neue Dialekte einzubinden. Hierfür sind insbesondere auch verschiedene Varianten für die Schnittstelle zu prüfen und der Entscheid ist zu dokumentieren.
- Implementation des Konzeptes einschliesslich:
  - Erzeugung von SQL Code für mindestens zwei verschiedene Dialekte (MS SQL und PostgreSQL)
  - Abbildung der modellierten Elemente in das objektorientierte Modell (nach ODMG oder db4O[1])
- Erstellung eines Berichtes, der das Konzept im Detail beschreibt und wesentliche Elemente zur Implementation enthält.
- Die Applikation wird mit C# auf der .NET Plattform realisiert.

1.1.2 Projektziele

Die folgende Liste definiert die konkreten Ziele des Projektes:

- Es wird eine zielgruppenorientierte Software für Studierende konzipiert. Die Applikation wird deshalb auf schulischen Lernzielen aufbauen. Es ist nicht als Konkurrenztool im Markt (z.B. zum ERwin[2]) zu sehen.
- Die Software wird nur Kernfunktionalitäten unterstützen. Neue Funktionalität soll zu einem späteren Zeitpunkt einfach eingebaut werden können. Dies kann entweder durch eine Erweiterung des Programms mit einer direkten Source-Code-Anpassung passieren oder durch Schreiben von zusätzlichen PlugIns.
- Dokumentation und Konzeption der Software und Schnittstellen (inkl. Statusberichte, Sitzungsprotokolle, Management Summary, Webabstract, Source-Code und Source-Code-Dokumentation).

[1] Db4O (database for objects) ist eine objektorientierte Datenbank für Java und .NET.
[2] Der ERwin Data Modeler ist eine kommerzielle Modellierungssoftware für relationale Datenbanken der Firma CA.

1.2 Informationen zum Dokument

Auf Wunsch der Professorin Frau Dr. C. Class konzentrieren wir uns in diesem Projekt nicht auf die Dokumente des IEEE-Standards mit SPMP, SRS, STD und SDD. Die Dozentin möchte die Konzeption des gesamten Projekts in einem Bericht vorfinden, welcher von Anfang bis Ende einen konsistenten roten Faden hat.

Folgende Dokumente gehören auch zum vorliegenden Konzept, werden aber separat geführt:

- Aufgabenstellung [Aufgabenstellung.pdf]
- Projektplan mit Arbeitspaketen und Zeitplan sowie Kontrolle [Projektplan.pdf]
- Risikoanalyse [Risikoanalyse.pdf]
- Risikodiagramm [Risikodiagramm.pdf]
- Programmierrichtlinien [Programmierrichtlinien.pdf]
- Technische Schnittstellenbeschreibung für die Generatoren (englisch) [TechnicalInterfaceDescriptionGenerator.pdf]
- Technische Schnittstellenbeschreibung für die Notationen (englisch) [TechnicalInterfaceDescriptionNotation.pdf]

1.3 Definitionen und Abkürzungen

Begriff	Bedeutung
asap.	As soon as possible (So schnell wie möglich)
CDR	Critical Design Review
Column	Feld, Spalte in einer relationalen Tabelle
DB	Datenbank
ER	Entity-Relationship-Diagramm (Datenbankschema)
ERM	Entity-Relationship-Model (Datenbankschema)
Erwin	AllFusion Erwin Data Modeler: Datenbank Designer
FDR	Final Design Review
FK	Foreign Key (Fremdschlüssel)
GUI	Graphical User Interface (graphische Benutzeroberfläche)
HSLU	Hochschule Luzern - Technik und Architektur
ODL	Object Definition Language
ODMG	Object Management Group
PK	Primary Key (Primärschlüssel)
Q-Punkte	Punkte zur Qualitätssicherung
SDD	Software Design Description
SPMP	Software Project Management Plan
SRS	Software Requirement Specification
STD	Software Test Document
SQL	Structured Query Language
T-Punkte	Punkte die mit einem Termin (Abgabe-Termin oder Sitzungstermin) in Zusammenhang stehen.
USDP	Unified Software Development Process
WWW	World Wide Web
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

Tabelle 1 – Definitionen und Abkürzungen

2 Vorgehen

2.1 Projektorganisation

2.1.1 Prozessmodell

Als Prozessmodell wird das erweiterte Wasserfallmodell mit Rücksprungmöglichkeiten gewählt. Die Projektgrösse ist überschaubar und so würden sich Modelle wie USDP nicht lohnen, da diese meist eher für grössere Projekte gedacht sind. Wir konzentrieren uns zunächst auf eine genaue und exakte Analyse. Alle Teile müssen zuerst definiert und mittels Variantenbildung evaluiert werden, bevor wir die Software implementieren. Bei der Implementation werden wir die kritischen Teile zuerst in einem vertikalen Prototyp umsetzen, den wir danach ausbauen. Wir haben bewusst kein Prozessmodell mit Iterationsschritten gewählt, da wir finden, dass diese für ein solches Projekt in der vorgegebenen Zeit zu schwerfällig und aufwändig wirken. Das erweiterte Wasserfallmodell ist zwar nicht so flexibel wie iterative Modelle, doch dies wird durch unsere aufwändige und genaue Analyse ausgeglichen. Auf eine Erklärung und Darstellung des Modells wird aufgrund seiner Bekanntheit verzichtet.

2.1.2 Organisationsstruktur

Für das Projekt ist eine reine Projektorganisation vorgesehen. Dies ergibt sich schon durch die Situation, zusätzlich neben der Arbeit und Schule ein Projekt durchzuführen. Das Projekt wird von Monika Schwitter und Jürg Zgraggen für die HSLU durchgeführt. Wir werden von Prof. Dr. Christina Class als Betreuerin begleitet. Prof. Dr. Thomas Olnhoff steht uns mit Rat und Tat zur Seite und gibt uns neben Frau Prof. Dr. Class wertvolle Inputs zum Projekt. Herr Louis-Pierre Gagnaux ist uns als Experte aus der Wirtschaft zugeteilt worden. Obwohl er nebenamtlich an der HSLU doziert, sehen wir ihn in unserem Projekt komplett als externen, unabhängigen Experten. Die Organisationsstruktur ist wie folgt festgelegt:

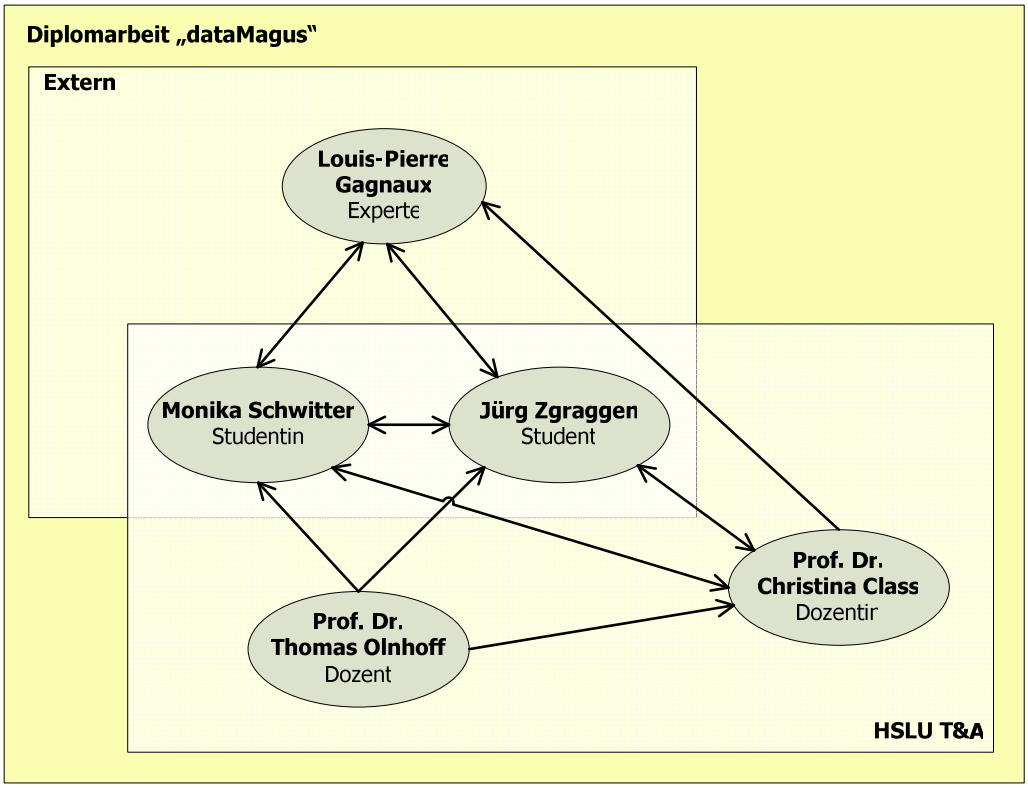


Abbildung 2 – Projektorganisation

Prof. Dr. Christina Class:	Dozentin an der HSLU und Projektbetreuerin
Prof. Dr. Thomas Olnhoff:	Dozent an der HSLU
Louis-Pierre Gagnaux:	Experte
Monika Schwitter:	Diplomandin und Entwicklerin
Jürg Zgraggen:	Diplomand und Entwickler

2.1.3 Projektspezifische Organisation und Schnittstellen

Arbeitsvorgänge und Probleme werden immer zusammen mit der HSLU ausdiskutiert. Es werden wöchentliche Sitzungen (mittwochs) abgehalten, bei denen ein Protokoll geführt wird. Die Sitzungsprotokolle werden an alle beteiligten Personen gesendet. In den Sitzungen wird über den aktuellen Projektstand informiert und über Probleme und Lösungen diskutiert. Die Kommunikation ausserhalb der Sitzungen läuft über Email.

Die diversen Ansprechpartner und ihre Koordinaten sind in der folgenden Tabelle ersichtlich:

Ansprechpartner	Email	Telefon
Prof. Dr. Christina Class	christina.class@hslu.ch	041 349 34 64
Prof. Dr. Thomas Olnhoff	thomas.olnhoff@hslu.ch	041 349 35 08
Louis-Pierre Gagnaux	louis-pierre@gagnaux.ch	041 799 29 53
Monika Schwitter	moni.schwitter@gmx.net	079 362 04 11
Jürg Zgraggen	j.zgraggen@bluewin.ch	078 659 53 33

Tabelle 2 – Ansprechpartner

2.2 Management-Prozess

2.2.1 Managementziele und Prioritäten

Es wurden folgende Ziele und Prioritäten definiert:

- Termingerechte Ablieferung einer lauffähigen Version der Software, wobei die Funktionalität und Qualität gegenüber einer ästhetischen graphischen Oberfläche den Vorzug haben. Die einzelnen Anforderungen dafür werden später noch genauer priorisiert und definiert.

- Einhaltung der gestellten Qualitätsanforderungen

- Vollständigkeit der Dokumentation

2.2.2 Voraussetzungen, Abhängigkeiten und Einschränkungen

Von der HSLU liegen keine Spezifikationen zu einem Programm wie dataMagus auf Papier vor. Abläufe und Prozesse sind nicht dokumentiert und wurden bislang nur mündlich besprochen. Deshalb werden wir uns auf die Kernprozesse konzentrieren und grossen Wert auf die Analyse setzen.
Das Projekt wird mit dem Microsoft .NET Framework mit der Programmiersprache C# programmiert.
Das Programm läuft auf jeder Plattform, die das Microsoft .NET Framework vollumfänglich unterstützt. Mit dem Mono-Projekt von Novell könnte es zu einem späteren Zeitpunkt auch möglich sein, das Programm unter Linux laufen zu lassen. Bis zum heutigen Tag werden von Mono die graphischen WinForms-Komponenten noch nicht unterstützt.

2.2.3 Risikomanagement

Die Risikoanalyse wird zu Beginn des Projekts initialisiert. Sie ist in einem separaten Excel-Dokument ausgearbeitet worden, weil dort auch die Diagrammfunktionalitäten etc. besser ausgenutzt werden können.
Die Risiken werden permanent im Auge behalten und bei Bedarf angepasst.

Die genauen Risiken sind im Dokument Risikoanalyse [Risikoanalyse.pdf] aufgeführt.

Aus der Risikoplanung gehen die folgenden zwei Hauptrisiken hervor:

- **Implementationsprobleme (R3)**
  Eintrittswahrscheinlichkeit = 20%
  Auswirkung = gross
  Die beiden kritischen Punkte hier betreffen die GUI-Programmierung sowie die Umsetzung der Datenbank-Modellierungsregeln. Falls grössere Probleme auftauchen, müssen wir uns hier Ad-hoc-Knowhow aneignen.

- **Notfallsituation (R1)**
  Eintrittswahrscheinlichkeit = 20%
  Auswirkung = gross

  Ein Ausfall oder eine grosse Behinderung bei den aktuell produktiven Systemen, die die Studierenden für ihre Firma betreuen, tritt ein. Solche Notfallsituationen haben die höchste Priorität und müssen umgehend behoben werden. Dadurch kann sich die

Projektplanung verschieben, jedoch ist das Projektende am 23.11.2007 fix und kann unter keinen Umständen geändert werden.

Um das Hauptrisiko R3 zu minimieren, werden wir in der Analyse- und Design-Phase mit Mini-Prototyping arbeiten.

2.2.4 Monitoring und Controlling

In allen Dokumenten ist eine Versionskontrolle zu führen, welche jeweils im ersten Kapitel des betreffenden Dokuments eingetragen wird. In der wöchentlichen Besprechung wird über den Projektstatus und -fortschritt informiert. Alle Sitzungen, Zwischenberichte und Statusberichte sind im separaten Projektplan [Projektplan.pdf] unter den T-Punkten aufgeführt.
Zur Qualitätssicherung sind diverse Q-Punkte im Projektplan in Form von Reviews geplant.

2.3 Technischer Prozess

2.3.1 Lizenzrechte

Die Software steht unter der GNU GPL (General Public Licence) [9].

2.3.2 Methoden, Tools und Techniken

Die Programme werden mit der Entwicklungsumgebung SharpDevelop 2.2 [10] in der Programmiersprache C# entwickelt. UML-Diagramme werden mit der Entwicklungsumgebung oder mit dem Microsoft Visio gezeichnet.
Die Dokumentation und Präsentation wird mit der gängigen Microsoft Office-Palette erstellt.
Die Versionskontrolle wird durch Subversion 1.4.5 [11] gewährleistet. Die notwendigen Systeme werden täglich in der Nacht gesichert.
Wir halten uns an die von uns erstellten Programmierrichtlinien [Programmierrichtlinien.pdf].

2.3.3 Softwaredokumentation

Microsoft .NET unterstützt eine direkte Dokumentation des Codes, die XML-Dokumentation. Daraus kann eine Hilfe im CHM-Format[3] mit dem Tool NDoc2 [12] für das Projekt generiert werden.

2.3.4 Vorschriften

- An einer Quellcodedatei dürfen nicht zwei EntwicklerInnen gleichzeitig arbeiten.
- Die Reviews müssen von der Dozentin abgenommen werden.

[3] Compiled Html Help (CHM)

2.4 Arbeitspakete, Zeitplan und Budget

Der Projektstrukturplan mit den Arbeitspaketen und der Zeitplanung sind im Projektplan ersichtlich [Projektplan.pdf]. Zum Leseverständnis hier einige Hinweise:

- Projektstart und –ende ist vorgegeben: 15.10.2007 – 23.11.2007

- Für diese Diplomarbeit stehen rund 420 h über die Dauer von 6 Wochen zur Verfügung. Die Pensen sind wie folgt festgelegt:
  1. – 3. Woche: Pro Woche 40 h = 120 h in 3 Wochen
  4. – 6. Woche: Pro Woche 100h = 300 h in 3 Wochen
  Die Stunden werden von uns (Monika Schwitter und Jürg Zgraggen) gleichmässig geleistet. Eine Budgetplanung entfällt, da es sich um eine Diplomarbeit für die HSLU handelt.

- Im Projektplan sind die Stunden in der Spalte *Aufwand geplant* relevant und nicht die Tage in der Spalte *Dauer*. Diese Stunden wurden jeweils für ein Arbeitspaket geschätzt. In der Spalte *Aufwand effektiv* wird die effektive Arbeitszeit festgehalten. Wir führen für unsere Kontrolle einen internen Arbeitsrapport. Damit ist für uns auch eine Projektkontrolle bzw. ein IST-SOLL-Vergleich möglich.

- Geschätzt wurde nach einer einfachen Schätzmethode und vor allem nach unseren eigenen Erfahrungen. Wir sind beide seit ca. 9 Jahren in der Softwareentwicklung tätig.

- Die Dokumentation erstreckt sich über die ganze Periode und wird parallel zu den einzelnen Phasen nachgeführt.

- Für den Projektmanagement-Aufwand wurde ebenfalls Zeit eingeplant, die sich über die ganze Periode erstreckt.

- Pro Phase wurden 5% Zeitpuffer eingeplant.

T-Punkte

Aus der Planung sind folgende T-Punkte (Sitzungen, Zwischenberichte und Statusbereichte) entstanden:

- 17.10.2007 T1.Besprechung mit Dozentin Prof. Dr. C. Class
- 19.10.2007 T2.Besprechung mit Prof. Dr. T. Olnhoff
- 19.10.2007 T3.Abgabe Statusbericht 1
- 24.10.2007 T4.Besprechung mit Dozentin Prof. Dr. C. Class
- 26.10.2007 T5.Abgabe Statusbericht 2
- 31.10.2007 T6.Besprechung mit Dozentin Prof. Dr. C. Class und Abgabe Zwischenbericht
- 07.11.2007 T7.Besprechung mit Dozentin Prof. Dr. C. Class und Experte Louis-Pierre Gagnaux
- 09.11.2007 T8.Abgabe Statusbericht 3
- 14.11.2007 T9.Besprechung mit Dozentin Prof. Dr. C. Class
- 19.11.2007 T10.Abgabe Statusbericht 4
- 19.11.2007 T11.Besprechung mit Dozentin Prof. Dr. C. Class

Q-Punkte

Aus der Planung sind folgende Q-Punkte (Qualitätssicherung) entstanden:

- 01.11.2007 Q1: Review (CDR)
- 05.11.2007 Q2: Review (FDR)
- 20.11.2007 Q3: Tests verabschieden

Meilensteine

Aus der Planung sind folgende Meilensteine entstanden:

- 01.11.2007 Freigabe Analyse
- 05.11.2007 Freigabe Design
- 16.11.2007 Releasekandidat
- 20.11.2007 Tests verabschieden
- 23.11.2007 Abgabe

3 Grundlagen der Datenmodellierung

In diesem Kapitel werden kurz die Grundlagen zur Datenmodellierung aufgezeigt. Die Ausführungen sind einerseits für das weitere Verständnis von Vorteil, andererseits wird in den hinteren Kapiteln auf diese Grundlagen Bezug genommen.

3.1 Begriffserklärung

Nachfolgend werden die wichtigsten Begriffe im Datenbankmodellierungsumfeld erklärt:

Begriff	Erklärung
Attribut	Attribute dienen dazu, Gegenstände (Entitätsmengen) bzw. Beziehungen zu charakterisieren. Siehe auch [2] auf S. 36.
Beziehung	Eine Beziehung beschreibt den Zusammenhang zwischen zwei Entitätsmengen. Siehe auch [2] auf S. 35.
DB-Notation	Graphische Repräsentation eines Datenbankmodells, meist mit symbolischen Zeichen.
Entitätsmenge	Entitätsmengen können auch als Gegenstände gesehen werden. Diese sind wohlunterscheidbare physisch oder gedanklich existierende Konzepte der zu modellierenden Welt. Siehe auch [2] auf S.35.
Kardinalität	Mögliche Anzahl der an einer Beziehung beteiligten Entitäten.
Schlüssel	Eine minimale Menge von Attributen, deren Werte die zugeordnete Entität eindeutig innerhalb aller Entitäten seines Typs identifiziert, nennt man Schlüssel. Siehe auch [2] auf S. 37.
Identifying Relationship (identifizierende Beziehung)	Eine als *identifying* markierte relationale Beziehung zwischen zwei Tabellen sagt aus, dass eine Tabelle als sog. Elterntabelle fungiert. Nur mit der Beziehung auf diese Elterntabelle ergeben die Einträge in der zweiten Tabelle erst Sinn. Technisch gehört das Fremdschlüsselattribut der Elterntabelle zum Primärschlüssel der untergeordneten Tabelle.
Non-Identifying Relationship (nicht-identifizierende Beziehung)	Bei einer relationalen Beziehung, die als *non-identifying* markiert ist, gehört das entsprechende Fremdschlüsselattribut nicht zum Primärschlüssel.

Tabelle 3 – Begriffe der Datenmodellierung

3.2 Datenmodellierung

Bei der Datenmodellierung wird ein Ausschnitt der realen Welt abgebildet. Dazu werden alle relevanten Objekte, ihre Eigenschaften und die zwischen den Objekten bestehenden Beziehungen analysiert. Diese Elemente werden schlussendlich in einem Modell grafisch dargestellt.
Diese Abbildung der realen Welt in ein Schema nennt man Abbildung in ein **logisches Schema**. Die Abstraktionsebene hier ist dabei meist recht hoch. Um die Daten physikalisch persistent halten zu können, benötigt man als Weiteres eine Datenbank. Auf dem Markt sind unterschiedliche Datenbank- und Objektmodelle vorhanden. Eine Aufstellung folgt im nächsten Kapitel. Im Normalfall lässt sich jedoch das logische Modell nicht direkt auf eine Datenbank abbilden. Es muss vielmehr in ein **physikalisches Modell** umgesetzt werden, das der Form und den Möglichkeiten der Datenbank entspricht. Jedes Modell (z.B. relational, objektorientiert, objektrelational etc.) hat seine eigenen Umsetzungsregeln.

In dataMagus wird das logische Schema in der logischen Ebene modelliert. Für jedes Datenbankmodell kann es eine eigene physikalische Ebene geben. Das logische Schema wird nach definierten Umsetzungsregeln in die verschiedenen Ebenen umgesetzt.

3.3 Datenbankmodelle

Die Grundlage für die Strukturierung von Daten und ihren Beziehungen zueinander stellen Datenbank- und Objektmodelle dar. Je nach Modell wird das jeweilige Schema an bestimmte Strukturierungsmöglichkeiten angepasst. Aufgrund verschiedenartiger Datenbanken gibt es auch unterschiedliche Abbildungsarten zur Modellierung der Strukturen. Die Haupttypen werden nachfolgend kurz umrissen. Daneben existiert auch noch eine Vielzahl von Modellen in abgewandelten oder Misch- und Nebenformen wie z.B. objektrelationale Modelle, die sich aus objektorientierten und relationalen Modellen zusammensetzen.

3.3.1 Hierarchisches Datenbankmodell

Bei hierarchischen Modellen stehen die Datenobjekte in einer verdrahteten Eltern-Kind-Beziehung zueinander. Es ist die älteste Modellierungsart und erhielt durch die Speicherung der Daten in XML wieder Aufschwung.

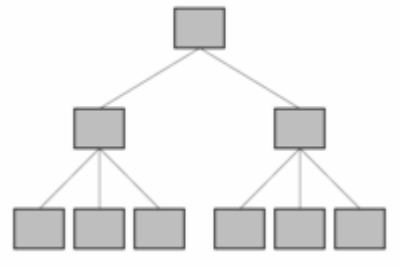


Abbildung 3 – Hierarchisches Datenmodell [22]

3.3.2 Netzwerkdatenbankmodell

Bei netzwerkartigen Modellen werden die Datenobjekte miteinander in Netzen verbunden.

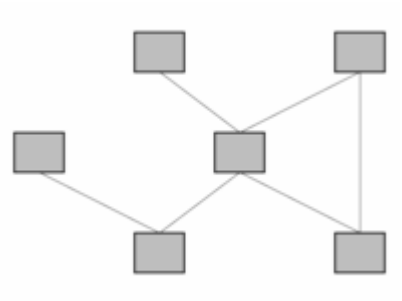


Abbildung 4 – Netzwerkdatenbankmodell [23]

3.3.3 Relationales Datenbankmodell

Die relationalen Modelle stellen die Datenobjekte in flachen Tabellen und Beziehungen dar. Die Beziehungen ergeben sich jeweils aus redundant geführten Werten. Hierbei handelt es sich um das am Weitesten verbreitete Modell.

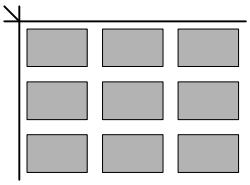


Abbildung 5 – Relationales Datenbankmodell

3.3.4 Objektorientiertes Datenbankmodell

Die Datenobjekte bei objektorientierten Modellen werden miteinander verbunden und sind immer durch eine Objekt-Id vom System eindeutig identifizierbar. Der Vorteil eines objektorientierten Modells liegt in der Möglichkeit, Objekte ineinander zu schachteln, um komplexe Strukturen abbilden zu können.

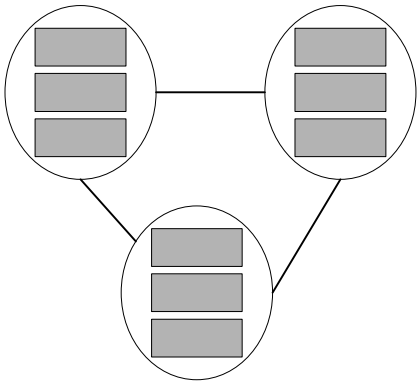


Abbildung 6 – Objektorientiertes Datenbankmodell

3.4 Notationen für Datenbankmodelle

Im Nachfolgenden werden die von uns als wichtig erachteten Notationen kurz erläutert. Unsere Aufstellung soll vor allem dem Grundverständnis dienen. Obwohl eine Notation von ihrem Erschaffer eigentlich formal und abschliessend definiert worden ist, ist es in der Praxis oft so, dass die gleiche Notation unterschiedlich eingesetzt wird, nur Teile daraus verwendet werden oder sie vom jeweiligen Autor leicht abgeändert wird.
Unsere Ausführung hier ist ein Zusammenzug mehreren Quellen [1], [2], [3] und [13].
Die Beispielbilder der unterschiedlichen Notationen stützen sich immer auf die gleiche Sachlage: Eine Person kann mehrere Haustiere haben. Ein Tier gehört höchstens einer Person.

3.4.1 Chen

Bei der Chen-Notation handelt es sich um eine nach dem Informatiker Peter Chen benannten Notation bzw. Darstellung eines ER-Modells. Sie eignet sich hervorragend zum Zeichnen von logischen Modellen.

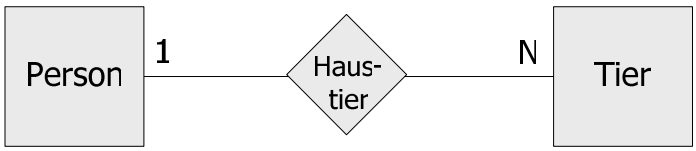
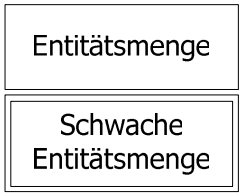
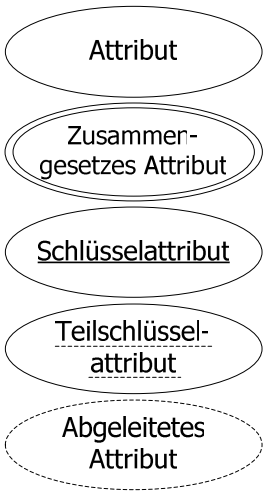


Abbildung 7 – Beispiel Chen-Notation

3.4.1.1 Elemente

Die Chen-Notation besteht aus folgenden Elementen:

Element	Beschreibung
	**Entitätsmengen**: Es existieren normale und schwache Entitätsmengen. Die schwachen Entitätsmengen sind in ihrer Existenz von einer anderen, übergeordneten Entitätsmenge abhängig und oft nur in Kombination mit dem Schlüssel der übergeordneten Entität eindeutig identifizierbar.
	**Attribute**: Normale Attribute werden durch eine Ellipse notiert. Ein doppelt umrandetes Attribut weist auf ein zusammengesetztes, mehrwertiges Attribut hin. Hat das Attribut Schlüsseleigenschaft, wird es unterstrichen. Ist es ein Teilschlüssel, wird es gestrichelt unterstrichen. Abgeleitete Attribute werden mit gestrichelten Ellipsen gezeichnet. Ein Beispiel dafür wäre das Alter, welches vom Geburtsdatum abgeleitet werden kann.

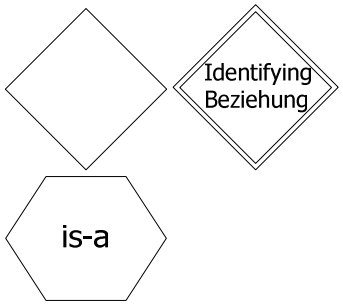
Element	Beschreibung
	**Beziehungen**: Eine Beziehung ist eine logische Verknüpfung zwischen zwei oder mehreren (n-stelligen) Entitätsmengen. Sie kann auch Attribute besitzen. Identifying-Beziehungen werden doppelt umrandet gezeichnet. Der Rhombus wird durch Linien mit den Entitätsmengen verbunden. Es sind 1:1-, 1:n- und n:m-Beziehungen modellierbar. Eine Spezialisierung einer Entitätsmenge wird mit der is-a-Beziehung modelliert.

Tabelle 4 – Elemente der Chen-Notation

3.4.1.2 Kardinalitäten

Für nachfolgende Beschreibung der Kardinalitäten werden die Begriffe wie folgt eingesetzt:
❶ = erste Entitätsmenge
❷ = zweite Entitätsmenge

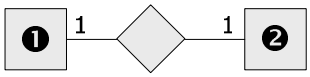


Abbildung 8 – Beziehung mit Definition der Begriffe

Kardinalität		Beschreibung
1:1	(0 oder 1) zu (1 oder 0)	Jede Entität aus der ersten Entitätsmenge kann mit **höchstens einer** Entität aus der zweiten Entitätsmenge in Beziehung stehen.
1:n	(0 oder 1) zu (beliebig vielen)	Jede Entität aus der ersten Entitätsmenge kann mit **beliebig vielen** Entitäten aus der zweiten Entitätsmenge in Beziehung stehen. Jede Entität aus der zweiten Entitätsmenge kann mit **höchsten einer** Entität aus der ersten Entitätsmenge in Beziehung stehen.
n:m	(beliebig viele) zu (beliebig vielen)	Jede Entität aus der ersten Entitätsmenge kann mit **beliebig vielen** Entitäten aus der zweiten Entitätsmenge in Beziehung stehen, und umgekehrt.

Tabelle 5 – Kardinalitäten der Chen-Notation

3.4.2 Modifizierter Chen

Ein Nachteil der Chen-Notation ist, dass mit *1:1*, *1:n* und *m:n* nur beschränkte Aussagen zu einer Beziehung gemacht werden kann. Aus diesem Grund wurden die Kardinalitäten der Chen-Notation etwas erweitert, was heute unter der Notation *Modifizierter Chen* bekannt ist.

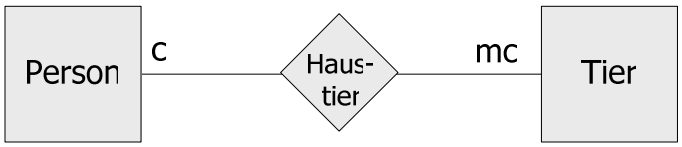


Abbildung 9 – Beispiel MC-Notation

3.4.2.1 Elemente

Alle Elemente wurden von der Chen-Notation übernommen.

3.4.2.2 Kardinalitäten

Auch hier gilt die obige Begriffsdefinition:
❶ = erste Entitätsmenge
❷ = zweite Entitätsmenge

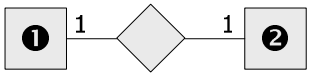


Abbildung 10 – Beziehung mit Definition der Begriffe

Kardinalität		Beschreibung
1:1	1 zu 1	Jede Entität der ersten Entitätsmenge steht mit **genau einer** Entität der zweiten Entitätsmenge in Beziehung, und umgekehrt.
1:c	1 zu (0 oder 1)	Jede Entität der ersten Entitätsmenge steht mit **mindestens einer** Entität der zweiten Entitätsmenge in Beziehung. Jede Entität der zweiten Entitätsmenge steht mit **genau einer** Entität der ersten Entitätsmenge in Beziehung.
1:m	1 zu (mindestens 1)	Jede Entität der ersten Entitätsmenge kann mit **höchstens einer** Entität der zweiten Entitätsmenge in Beziehung stehen. Jede Entität der zweiten Entitätsmenge steht mit **genau einer** Entität der ersten Entitätsmenge in Beziehung.
1:mc	1 zu (beliebig viele)	Jede Entität der ersten Entitätsmenge kann mit **beliebig vielen** Entitäten der zweiten Entitätsmenge in Beziehung stehen. Jede Entität der zweiten Entitätsmenge steht mit **genau einer** Entität der ersten Entitätsmenge in Beziehung.
c:c	(1 oder 0) zu (0 oder 1)	Jede Entität der ersten Entitätsmenge kann mit **höchstens einer** Entität der zweiten Entitätsmenge in Beziehung stehen, und umgekehrt.

Kardinalität		Beschreibung
c:m	(0 oder 1) zu (mindestens 1)	Jede Entität der ersten Entitätsmenge steht mit **mindestens einer** Entität der zweiten Entitätsmenge in Beziehung. Jede Entität der zweiten Entitätsmenge kann mit **höchstens einer** Entität der ersten Entitätsmenge in Beziehung stehen.
c:mc	(0 oder 1) zu (beliebig viele)	Jede Entität der ersten Entitätsmenge kann mit **beliebig vielen** Entitäten der zweiten Entitätsmenge in Beziehung stehen. Jede Entität der zweiten Entitätsmenge kann mit **höchstens einer** Entität der ersten Entitätsmenge in Beziehung stehen.
m:m	(mindestens 1) zu (mindestens 1)	Jede Entität der ersten Entitätsmenge steht mit **mindestens einer** Entität der zweiten Entitätsmenge in Beziehung, und umgekehrt.
m:mc	(mindestens 1) zu (beliebig viele)	Jede Entität der ersten Entitätsmenge kann mit **beliebig vielen** Entitäten der zweiten Entitätsmenge in Beziehung stehen. Jede Entität der zweiten Entitätsmenge steht mit **mindestens einer** Entität der ersten Entitätsmenge in Beziehung.
mc:mc	(beliebig viele) zu (beliebig viele)	Jede Entität der ersten Entitätsmenge kann mit **beliebig vielen** Entitäten der zweiten Entitätsmenge in Beziehung stehen, und umgekehrt.

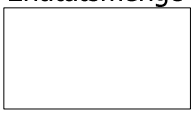
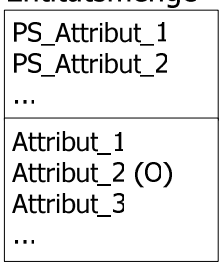
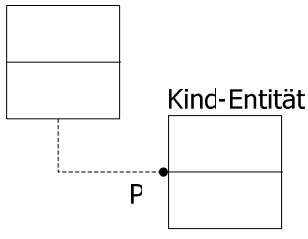
Tabelle 6 – Kardinalitäten der MC-Notation

3.4.3 IDEF1x

IDEF1x ist eine Modellierungssprache im ICAM[4]-Standard und gehört zur Gruppe der IDEF-Sprachen[5]. IDEF1x fokussiert sich auf physikalische ER-Modelle.

Abbildung 11 – Beispiel IDEF1x-Notation

3.4.3.1 Elemente

Element	Beschreibung
Entitätsmenge  Schwache Entitätsmenge 	**Entitäten:** Normale Entitätsmengen werden mit einem Rechteck dargestellt. Der Name (oder eine Nummer) wird darüber gestellt. Existenzabhängige Entitätsmengen werden als abgerundetes Rechteck modelliert.
Entitätsmenge  PS_Attribut_1 PS_Attribut_2 ... Attribut_1 Attribut_2 (O) Attribut_3 ...	**Attribute**: Die Namen der Attribute werden in das Entitäts-Rechteck geschrieben. Attribute, die zum Primärschlüssel gehören stehen über, alle anderen unter einem Trennungsstrich. Zusätzlich zu den Namen werden weitere Kennzeichnungen aufgenommen, wie z.B. "(O)" für optional.
Eltern-Entität  Kind-Entität P	**Beziehungen:** Eine Beziehung läuft von der Eltern-Entität zur Kind-Entität. Das Kind-Ende der Beziehung ist mit einem kleinen schwarzen Kreis gekennzeichnet. Zusätzlich wird durch Linienart, -ende und Kommentierung die Kardinalität der Beziehung dargestellt. Die Kardinalität der Eltern-Seite der Beziehung kann nun als *Kann* oder *Muss* definiert werden. Eine optionale Beziehung wird am Linienende der Eltern-Seite mit einer Raute versehen, eine zwingende Beziehung hat ein

[4] **I**ntegrated **C**omputer **A**ided **M**anufacturing. ICAM ist eine Initiative zur Standardisierung bei der computerintegrierten Produkten, die von der US Air Force veranlasst wurde.
[5] **I**ntegrated **Def**inition Methods. Gruppe von Modellierungssprachen.

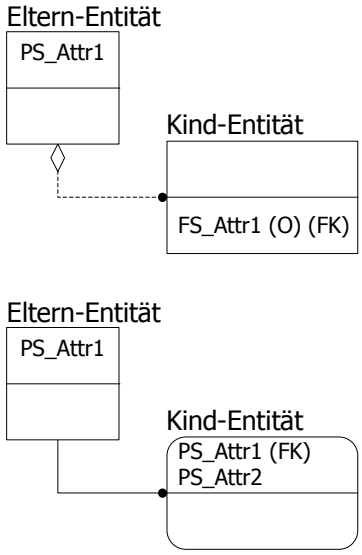
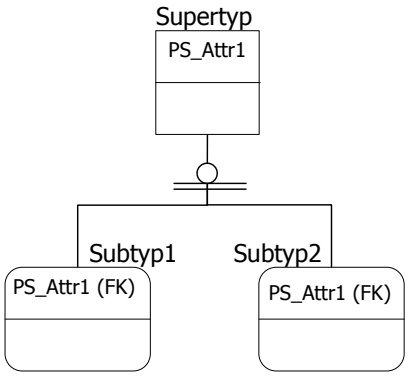
	normales Linienende. Dies bedeutet, dass in IDEF1x keine n:m-Beziehungen erlaubt sind. Eine n:m-Beziehung muss bei der Modellierung in zwei 1:n-Beziehungen aufgebrochen werden. Eine weitere Besonderheit gegenüber anderen Notationen ist die explizite Darstellung von Fremdschlüssel-Attributen, obwohl diese aus der Modellierungssicht redundant sind. Bei einer identifizierenden Beziehung ist die Beziehungslinie durchzogen (eine nicht identifizierende Beziehung ist gestrichelt) und die Kind-Entität wird mit einem abgerundeten Rechteck dargestellt.
	**Vererbung:** Das Konzept der Vererbung, bei der Datenmodellierung als *Sub- und Supertyp* bezeichnet, wird über eine *Kategorisierung* dargestellt. Diese Kategorisierung ist eine Menge spezieller Beziehungen vom Supertyp zu seinem Subtypen, die mit einem Diskriminator versehen werden. Eine durchgezogene Linie geht vom Supertyp zum Diskriminator, einem Kreis mit einer waagerechten einfachen oder Doppellinie darunter. Von der Diskriminatorlinie aus geht jeweils eine durchgezogene Linie zu jedem Subtyp. Die Doppellinie steht für die Vollständigkeit der aufgeführten Subtypen.

Tabelle 7 – Elemente der IDEF1x-Notation

3.4.3.2 Kardinalitäten

Folgende Kardinalitäten können definiert werden, aber nur bei der Kind-Entitätsmenge:

Kardinalität	Beschreibung
nichts	Null, eins oder beliebig viele
P	Eins bis beliebig viele
Z	Null oder eins
n	Genau n
n-m	Zwischen n und m
(I)	Verweis auf Notiz

Tabelle 8 – Kardinalitäten der IDEF1x-Notation

3.4.4 Krähenfuss / Martin / Information Engineering (IE)

Die Information Engineering (IE)-, Krähenfuss- oder auch Martin-Notation wurde von James Martin erfunden. Sie ist weit verbreitet und eignet sich besonders gut zur Darstellung von physikalischen Modellen. Wir bevorzugen den Namen *Krähenfuss*.
Zu dieser Notation existieren leider kaum Dokumente, die eine eindeutige Definition enthalten. Wir werden daher versuchen, die Notation aus diversen Quellen und Implementationsregeln (aus Erwin oder DBDesigner, siehe *Kapitel 3.5 Modellierungssoftware*) zusammenzustellen.

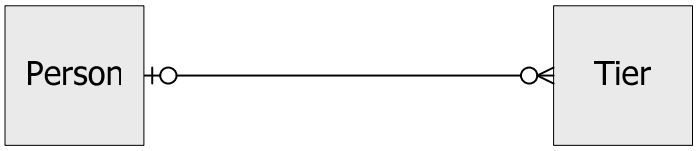


Abbildung 12 – Beispiel Martin-/ IE- / Krähenfuss-Notation

3.4.4.1 Elemente

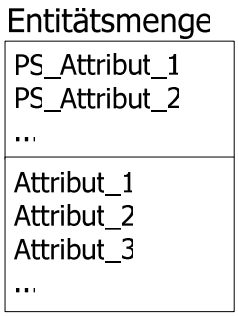
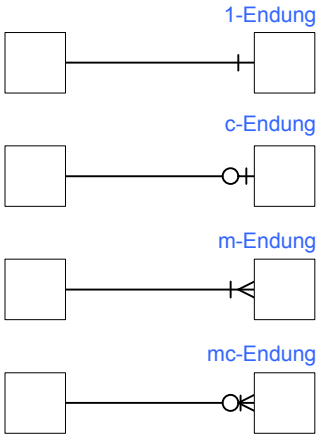
Elemente	Beschreibung
	**Entitätsmengen**: Eine Entitätsmenge wird mit einem Rechteck gezeichnet. Sie enthält den Namen und allfällige Attribute.
	**Attribute**: Die Attribute zu einer Entitätsmenge werden durch einen waagrechten Strich vom Namen der Entitätsmengen getrennt. PK- und evtl. FK-Attribute sollten noch zusätzlich gekennzeichnet werden, um sie von den anderen zu unterscheiden. Oft wird das über ein Bild eines Schlüssels getan, das dem Attribut vorgestellt wird. Man kann sie aber auch nochmals mit einer Linie trennen.
	**Beziehungen**: Die Krähenfuss-Notation hat ihren Namen von der Darstellungsart der Beziehungen. Um die Krähenfüsse genau zu verstehen, vergleichen wir diese mit der MC-Notation. Eine 1-Endung wird mit einem Strich gekennzeichnet. Eine c-Endung wird mit einem Kreis und einem senkrechten Strich gezeichnet (0 bis 1). Die m-Endung wird mit einem Strich und dem Krähenfuss angezeigt. Die mc-Endung hat einen Kreis, einen Strich und den Krähenfuss, welcher 0, 1 oder mehrere Beziehungen darstellen soll.

Tabelle 9 – Kardinalitäten der Martin-/ IE- / Krähenfuss-Notation

3.4.5 Min-Max-Notation

Die Min-Max-Notation (auch (min,max)-Notation) ist eine Art, die Kardinalität einer Beziehung zwischen Entitätsmengen in einem Entity-Relationship-Modell zu beschreiben. Sie wurde eingeführt, weil die Chen-Notation, mit *1:1*, *1:n* und *m:n*, nur beschränkte Aussagen zu einer Beziehung erlaubt. Mit der Min-Max-Notation können sowohl Unter- als auch Obergrenzen präzise ausgedrückt werden.

Bei der Min-Max-Notation wird für jede an einer Beziehung beteiligte Entitätsmenge ein 2-stelliger Tupel mit einem Minimal- und einem Maximalwert angegeben. Diese Werte geben an, wie viele Entitäten zweier verschiedener Mengen minimal und maximal zueinander in Beziehung gebracht werden können.

3.4.5.1 Formalisierung

Eine Beziehung hat die Kardinalität (a, b), wenn zu jeder Entität aus der Entitätsmenge X mindestens a, aber höchstens b verschiedene Entitäten aus der Entitätsmenge Y in Beziehung gebracht werden können.

- 0 steht für kein Vorkommen
- 1 steht für einmaliges Vorkommen
- * steht für beliebig oftes Vorkommen (0.. unendlich)

3.4.5.2 Gegenüberstellung von Min-Max-Notation, Chen-Notation und MC-Notation

Zu beachten ist, dass die Min-Max-Angaben gegenüber der Chen- oder MC-Notation invers zu betrachten sind.

Entität 1	Chen-Notation	MC-Notation	Entität 2
(0,1)	1:1	c:c	(0,1)
(0,*)	1:n	c:mc	(0,1)
(0,*)	--	1:mc	(1,1)
(0,*)	m:n	mc:mc	(0,*)
(0,1)	--	1:c	(1,1)
(1,*)	--	c:m	(0,1)
(1,1)	--	1:1	(1,1)
(1,*)	--	1:m	(1,1)
(1,*)	--	mc:m	(1,*)
(1,*)	--	m:m	(1,*)

Tabelle 10 – Gegenüberstellung verschiedener Notationen

Die Min-Max-Notation hat Vor- aber auch Nachteile gegenüber anderen Notationsregeln, speziell hinsichtlich der Konsistenzbedingungen. So können in der reinen Min-Max-Notation keine Beziehungen modelliert werden, an denen drei oder mehr Tabellen beteiligt sind. Daher wird sie meist zusammen mit der Chen-Notation verwendet, d.h. man modelliert in der Chen-Notation und verwendet die Kardinalitäten der Min-Max-Notation.

3.4.6 ODLG (Object Definition Language Graphical)

Die ODMG (Object Database Management Group) beschreibt in ihrem Buch [3] eine Notation zur Darstellung von objektorientierten Modelle. Für diese Notation lässt sich leider nirgends ein Name ausfindig machen. Wir nennen diese Notation somit ODLG (Object Definition Language Graphical). Diese Namenskreation stammt von ODL (Object Definition Language) ab und soll zum Ausdruck bringen, dass wir diese grafisch darstellen.

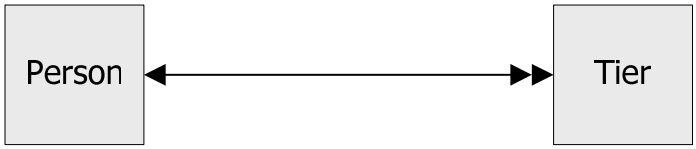


Abbildung 13 – Beispiel ODLG-Notation

3.4.6.1 Elemente

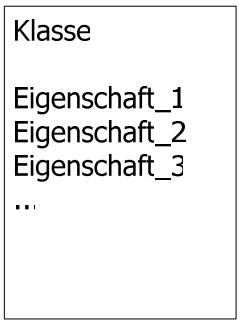
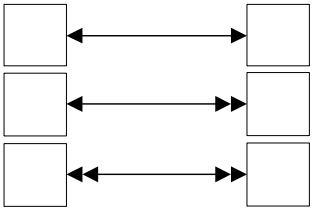
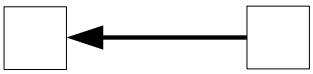
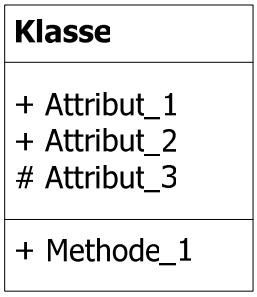
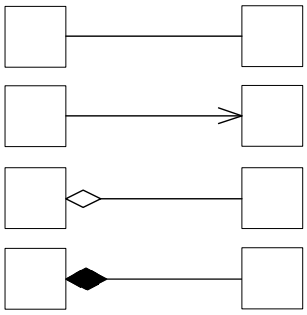
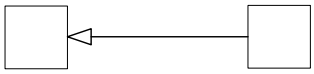
Elemente	Beschreibung
	**Entitätsmengen**: Eine Entitätsmenge wird mit einem Rechteck gezeichnet. Sie enthält den Namen und allfällige Attribute. Eine Entitätsmenge wird als Klasse bezeichnet. **Attribute (oder Eigenschaften)**: Wie Attribute zu einer Klasse dargestellt werden, ist durch die ODMG nicht definiert. Wir führen die Attribute unterhalb des Klassennamens.
	**Beziehungen**: Die Beziehungen werden mit verschiedenen Arten von Pfeilen dargestellt. Dabei existieren Einfach- und Doppelpfeile, um die jeweiligen Kardinalitäten 1 oder n zu definieren.
	**Vererbungen**: Ein Pfeil mit grösserer Pfeilspitze und breiterer Linie wird für Vererbungen gebraucht.
	**Schnittstellen**: Für die Implementation von Schnittstellen verwendet ODMG einen schwach gedruckten Pfeil mit noch breiterer und grösserer Pfeilspitze als der Vererbungspfeil.

Tabelle 11 – Elemente der ODLG-Notation

3.4.7 UML (Unified Modeling Language)

UML ist eine von der Object Management Group (OMG) entwickelte und standardisierte Sprache für die Modellierung von Software und anderen Systemen. Das Klassendiagramm mit seinen Beziehungen eignet sich hervorragend für eine objektorientierte Modellierung.

3.4.7.1 Elemente

Elemente	Beschreibung
	**Entitätsmengen**: Eine Entitätsmenge wird mit einem Rechteck gezeichnet. Sie enthält den Namen und allfällige Attribute. Eine Entitätsmenge wird in UML als Klasse bezeichnet. **Attribute**: Die Attribute zu einer Entitätsmenge werden durch einen waagrechten Strich vom Entitätsmengen-Namen getrennt. Öffentliche, geschützte und private Attribute werden zusätzlich gekennzeichnet, um sie von anderen zu unterscheiden. Vielmals wird dies über ein Symbol („+" für public, „#" für protected, „~" für package und „-" für private) getan. Attribute sind Eigenschaften einer Klasse. **Methoden**: Die einzelnen Methoden einer Klasse werden nochmals mit einem waagrechten Strich von den Attributen getrennt. Die Sichtbarkeit wird wiederum mit einem Symbol (+, #, ~, -) dargestellt.
	**Beziehungen**: Die Beziehungen werden mit verschiedenen Arten von Pfeilen und Rauten dargestellt. Eine Linie zwischen zwei Klassen stellt eine Beziehung dar. Ein offener Pfeil symbolisiert eine gerichtete Beziehung. Rauten werden gebraucht, um Teil-Beziehungen zu zeigen. Bei einer Aggregation (nicht ausgefüllte Raute) können die „Teile des Ganzen" auch einzeln existieren, bei der Komposition (ausgefüllte Raute) nur, wenn auch das „Ganze" existiert. An beiden Enden der Verbindungslinie wird oft die Kardinalität vermerkt, die angibt, mit wie vielen Objekten diese Beziehung zustande kommen kann.
	**Vererbungen**: Ein geschlossener, nicht ausgefüllter Pfeil zeigt eine Vererbung.

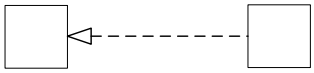
Elemente	Beschreibung
	**Schnittstellen**: Für die Implementation von Schnittstellen werden geschlossene, nicht ausgefüllte Pfeile mit gestrichelter Linie verwendet.

Tabelle 12 – Elemente der UML-Notation

3.5 Modellierungssoftware

Auf dem Markt existiert eine Reihe verschiedener Modellierungssoftware, um Datenmodelle zu erschaffen. Der Grossteil hat sich auf die Modellierung von relationalen Datenbankmodellen spezialisiert.
Nachfolgend werden einige ausgewählte Modellierungstools kurz vorgestellt und deren Stärken und Schwächen analysiert.

3.5.1 Erwin Data Modeler (AllFusion)

Auf dem Markt ist bereits die Version 7.2 erhältlich. Wir arbeiten jedoch noch mit einer Schullizenz der Version 4.1.2208.

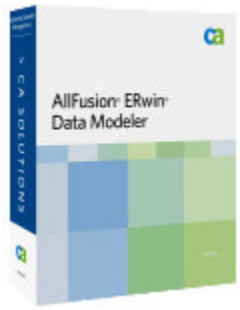


Abbildung 14 – Produkt AllFusion ERwin Data Modeler [20]

Mit Erwin Data Modeler können relationale Datenmodelle in einer logischen und physikalischen Sicht modelliert werden. Es kann Quellcode für die implementierten Zielsysteme wie MS SQL Server, Oracle, Sybase etc. generiert werden. Erwin unterstützt verschiedene Notationen und alle Elemente der relationalen Datenbank.

Vorteile:
+ Unterstützt alle Elemente der relationalen Datenbankmodellierung.
+ Unterstützt Templates für Trigger mit einer eigens dafür kreierten Sprache.
+ Unterstützt mehrere Notationen.
+ Unterstützt mehrere Zielsysteme.
+ Erlaubt eine logische und physikalische Sicht.
+ Unterstützt Reverse Engineering.

Nachteile:
- Etwas instabil. Gelegentliche Abstürze.
- Logische und physikalische Ebene sind sehr schlecht voneinander getrennt. Die logische Ebene lässt halbphysikalische Konstrukte zu.
- Sehr wenige, logische Regeln. Es können alle Varianten bzw. Kardinalitäten einer Beziehung modelliert werden, die u.U. aber gar keinen Sinn machen.
- Da es sehr mächtig ist, benötigt ein Benutzer eine relative lange Einführungszeit, bis er es produktiv nutzen kann. Das GUI ist nicht immer intuitiv.
- Bei der graphischen Umsetzung von is-a-Beziehungen und von Selbstbeziehungen bekundet Erwin Mühe.
- Erwin ist sehr kostenintensiv.
- Nicht erweiterbar.
- Unterstützt nur relationale Datenbanken.

3.5.2 DBDesigner 4 (fabFORCE.Net)

DBDesigner 4 ist ein Open Source – Projekt, welches vom fabFORCE-Team vorangetrieben wird. Die Software ist momentan in der Version 4.0.5.6 erhältlich.

Abbildung 15 – Produkt fabFORCE.net DBDesigner 4 [21]

Der DBDesigner 4 erlaubt die Modellierung von relationalen Datenbankmodellen. Es wurde ursprünglich entwickelt, um MySQL-Datenbanken abzubilden. Mittlerweile werden die gängigsten Zielsysteme abgedeckt.

Vorteile:
+ Gratis, da es sich um ein Open Source Projekt handelt.
+ Unterstützt den Import der Datenbankmodelle von Oracle Designer, IBM Rationale Rose, AllFusion Erwin etc.
+ Reverse Engineering einiger Datenbanken.
+ Generierung von SQL-Skripts.
+ Erweiterbar durch PlugIn-Technologie.
+ Hilfe zur Zusammenstellung eines Abfrage-Statements on the fly.

Nachteile:
- Sehr viele Abstürze.
- Keine wirkliche Trennung der logischen und physikalischen Ebene.
- Keine vollständige Umsetzung der Beziehungen.
- Erlaubt völlige Missbildungen von Beziehungen. Zirkulärbezüge werden zugelassen.
- Die Generierung von PK- und FK-Attribute ist fehlerhaft und kann in vielen Fällen nicht nachvollzogen werden.
- Hat zum Teil Mühe mit dem GUI-Handling (Felder können nicht mehr verlassen werden).
- Unterstützt nur relationale Datenbanken.

3.6 SQL Server 2005 (Microsoft)

Der SQL Server 2005 ist das Datenbanksystem von Microsoft.

Abbildung 16 – Produkt MS SQL Server 2005 [14]

Mit dem SQL Server 2005 kann auch eine Modellierung des Datenschemas gemacht werden. Allerdings findet die Modellierung bereits auf der physikalischen Ebene statt und ist nur für den SQL Server 2005 ausgelegt.

Vorteile:
+ Unterstützt MS SQL Server 2005 Tabellen und Beziehungen.
+ Datenmodell wird syntaktisch richtig gezeichnet.

Nachteile:
- Kostenpflichtig.
- Modellierung nur auf der physikalischen Ebene möglich und von daher werden nur 1:1- oder 1:n-Beziehungen unterstützt.
- Keine wirkliche Modellierungssoftware.

4 Konzept mit Variantenentscheid

4.1 Abgrenzung zu marktreifer Software

DataMagus soll vor allem den Datenbank-Unterricht unterstützen und den Studierenden zu wichtigen Erkenntnissen im Zusammenhang mit der Datenmodellierung auf der logischen Ebene und deren Umsetzung auf die physikalischen Ebenen verhelfen. Dieses Produktziel hat klar die höchste Priorität.
Auf dem Markt gibt es etliche Datenbank-Designer, welche aber alle auf den produktiven Einsatz spezialisiert sind. Als Konsequenz daraus muss man bei der Benutzung solcher Produkte mindestens schon halb-physikalisch denken und eine wirklich logische Abstraktion ist nicht möglich. Jedoch sind solche Tools meistens sehr mächtig. Des Weitern sind solche Datenbank-Designer meist auch „nur" für das relationale Datenbankmodell ausgelegt.
DataMagus soll ganz klar keine Konkurrenz zu solchen marktreifen Tools darstellen. Unser Programm soll dem Unterricht dienen und ausserdem einige Lücken füllen, die es bei den anderen Datenbank-Designer gibt.

4.2 Zentrale Produktziele aus didaktischer Sicht

Wir erachten folgende Produktziele als zentral, um den Lernzielen im Unterricht gerecht zu werden. Dadurch wird auch unsere Software massgeblich beeinflusst.

- **Logisches Modell vs. Physikalische Modelle**
  Den Studierenden sollen die Unterschiede der logischen und physikalischen Ebenen klar werden. Sie sollen lernen, wie ein logisches Schema auf unterschiedlichen physikalischen Modellen (relational, objektorientiert etc.) umgesetzt wird. Es soll auf einen Blick ersichtlich sein, wie z.B. die verschiedenen Beziehungstypen in einem relationalen oder einem objektorientierten System umgesetzt werden müssen. Eine n:m-Beziehung wird in einem relationalen System mit einer Zwischentabelle (eine Beziehungstabelle) realisiert, wobei hingegen in einem objektorientierten System die beiden beteiligten Klassen jeweils eine Liste mit Referenzen auf Instanzen der anderen Klasse führen. Eine genaue Beschreibung der Umsetzungsregeln erfolgt im *Kapitel 4.8.5 Umsetzung der logischen Ebene in die physikalischen* Ebenen.

  → **Konsequenzen**:
  - Eine Modellierung des Schemas ist nur auf der logischen Ebene möglich. Eine zusätzliche Modellierung auf den physikalischen Ebenen würde den Studierenden eher verwirren. Selbstverständlich können aber auf den physikalischen Ebenen die Eigenschaften der einzelnen Modellierungselemente geändert sowie spezifische Einstellungen für die zur Verfügung stehenden Zielsysteme vorgenommen werden. Ausserdem erzwingt eine Modellierung auf der logischen Ebene ein möglichst sauberes Datenbank-Design, da das Problem zuerst von der logisch abstrakten Sicht her angepackt wird und es dem Benutzer nicht erlaubt ist, irgendwelche halb-physikalische oder wirklich physikalische Lösungen zu modellieren.

  - Da keine Modellierung auf den physikalischen Ebenen möglich ist, sondern immer vom logischen Modell ausgegangen wird, setzen wir nur die logischen Elemente in die physikalischen Ebenen um. Es gibt physikalische Modelle, die weitere Modellierungselemente zur Verfügung stellen würden, welche wir aber bewusst weglassen. Das Hauptziel ist, den Studierenden die logische Abstraktion und die physikalische Umsetzung näher zu bringen. Sobald der

oder die Studierende diesen Lernprozess hinter sich hat, kann er/sie sich an die physikalische Modellierung wagen. Dazu benötigt er/sie aber immer einen eigens dafür ausgelegten, marktreifen DB-Designer, da unsere Software nicht für den professionellen Einsatz gedacht ist. Solche marktreifen Designer beherrschen dann selbstverständlich auch viele zusätzliche Elemente der jeweiligen physikalischen Ebene, die sie unterstützen.

- Um den Studierenden die Unterschiede der einzelnen physikalischen Modelle aufzuzeigen, werden wir in der ersten Version eine relationale und objektorientierte Ebene implementieren. Es könnten aber in Zukunft noch weitere Ebenen dazu programmiert werden. Wir implementieren das relationale Modell, da es in der Praxis sehr häufig eingesetzt wird. Ausserdem werden das relationale und das objektorientierte Modell im Datenbank-Unterricht an der HSLU besprochen. Die Studierenden können somit das Gelernte mit unserer Software nachvollziehen.

- **Generierung unterschiedlicher Dialekte**
  Die Studierenden sollen lernen, wie z.B. ein physikalisch relationales Datenmodell für einen MS SQL Server oder eine PostgreSQL-Datenbank in CREATE-Statements umgesetzt wird. Durch die Möglichkeit, für unterschiedliche Zielsysteme Code zu generieren, werden den Studierenden die verschiedenen Dialekte veranschaulicht. Ausserdem lernen sie die programmtechnischen Unterschiede zwischen den einzelnen Datenbankmodellen.

4.3 Ablauf des Modellierungs- und Generierungsprozesses

Die Software hat eine logische und mehrere physikalische Ebenen (in der ersten Version des Programms sind diese eine relationale und eine objektorientierte), auf welchen das Datenmodell jeweils verschieden dargestellt wird. Der Benutzer der Applikation kann nur auf der logischen Ebene modellieren ❶.

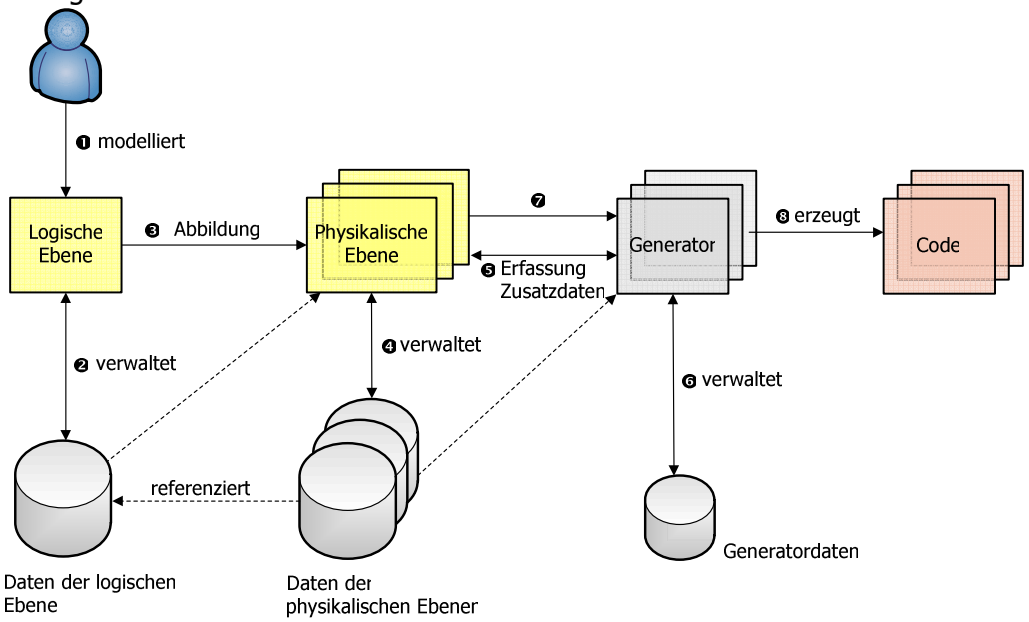


Abbildung 17 – Modellierungs- und Generierungsprozess

Die logische Ebene speichert die modellierten Entitäten und Beziehungen in einem logischen Datenbehälter ❷. Diese Daten können auch nur auf der logischen Ebene verändert werden.

Sobald der Benutzer von der logischen Ebene auf irgendeine physikalische Ebene wechselt, findet eine Abbildung ❸ im Sinne der Umsetzung des logischen Schemas auf das jeweilige physikalische statt. Für die Umsetzung kann je nach modellierten Konstrukten eine Benutzerabfrage notwendig werden.
Das erzeugte physikalische Modell wird in einem eigenen Datenbehälter abgelegt ❹. Es wird bei jeder Abbildung neu generiert. Dabei hat jedes physikalische Datenelement eine Referenz auf sein logisches Ursprungselement.
Die Daten auf der physikalischen Ebene können durch schemabedingte, generierte Elemente (z.B. einen Index) erweitert und von den Studierenden durch zielsystemspezifische Einstellungen angereichert werden. Diese physikalischen Änderungen werden persistent zwischengespeichert und nach der Abbildung wieder restauriert. Ebenfalls werden die Antworten der Benutzerabfragen als zusätzliche physikalische Änderungen abgelegt. Somit brauchen die Studierenden bei einer weiteren Umsetzung desselben physikalischen Schemas nicht immer wieder die Art der Umsetzung anzugeben. Diese physikalischen Daten können durch eine Funktion wieder zurückgesetzt werden.

Der Quellcode für ein jeweiliges Zielsystem wird durch einen eigens dafür entwickelten Generator erstellt. Generatorspezifische Daten können auf der physikalischen Ebene erfasst werden ❺. Dazu implementiert jeder Generator die benötigten Schnittstellen. Diese Daten werden aber von jedem Generator selber verwaltet ❻. Will nun ein Benutzer Quellcode für ein entsprechendes Zielsystem generieren, muss er/sie sich auf der dafür vorgesehenen physikalischen Ebene befinden und den entsprechenden Generator starten ❼. Der Generator erzeugt schliesslich den Code ❽.

4.4 Produktfunktionen

4.4.1 Funktionsliste und Anforderungskatalog

Die folgende Liste definiert die konkreten Ziele und Funktionen der Software.

Nr.	Anforderung	Prio.	Aufw.	Risiko	Kat.
1	**Logische Ebene**				
1.1	Modellierung mit logischen Elementen: Entitätsmengen mit Attributen, Relationen mit Attributen	⇧	⇧	⇩	F
1.2	Beziehungen: 1:1, 1:n, n:m, is-a	⇧	⇧	⇩	F
1.3	n-stellige Beziehungen (n>2)	⇩	⇧	⇧	AF
1.4	Schwache Entitätsmengen	⇩	⇩	⇧	AF
1.5	has-a-Beziehung	⇩	⇨	⇧	AF
2	**Physikalische relationale Ebene**				
2.1	Umsetzung der logischen Elemente (Entitäten, Relationen, Attribute) auf physikalisch relationale	⇧	⇧	⇩	F
2.2	Einstellen physikalischer Eigenschaften der Elemente	⇧	⇨	⇩	F
2.3	Generatorspezifische Eigenschaften	⇧	⇨	⇩	F
3	**Physikalische objektorientierte Ebene**				
3.1	Umsetzung der logischen Elemente (Entitäten, Relationen, Attribute) auf physikalisch objektorientierte	⇧	⇧	⇩	F
3.2	Einstellen physikalischer Eigenschaften der Elemente	⇧	⇨	⇩	F
3.3	Generatorspezifische Eigenschaften	⇧	⇨	⇩	F
4	**Code für Zielsysteme, Dialekte**				
4.1	Generisches Konzept zur Generierung von Quellcode	⇧	⇨	⇩	F
4.2	Zuordnung logischer Datentypen zu zielsystem-spezifischen				
4.3	Generieren von Code für den MS SQL Server	⇧	⇨	⇩	F
4.4	Generieren von Code für PostgreSQL	⇨	⇩	⇩	F
4.5	Generieren von Code in der ODL Syntax	⇧	⇨	⇩	F
5	**Erweiterbarkeit**				
5.1	Erweiterungsmöglichkeiten für andere Notationen	⇨	⇧	⇨	NF
5.2	Erweiterungsmöglichkeiten für andere Datenbankdialekte oder Zielsysteme	⇧	⇧	⇩	NF
5.3	Erweiterungsmöglichkeiten für andere physikalische Ebenen	⇨	⇧	⇩	NF
5.4	Erweiterungsmöglichkeiten für neue Modellierungselemente	⇨	⇨	⇧	NF
6	**Reverse Engineering**				
6.1	Analyse von Quellcodedateien	⇩	⇧	⇧	AF
6.2	Analyse von Datenbanken per Verbindung	⇩	⇧	⇧	AF

Nr.	Anforderung	Prio.	Aufw.	Risiko	Kat.
7	**Mehrsprachigkeit**				
7.1	Zentrale Speicherung von sprachabhängigen Daten	⇨	⇩	⇩	F
7.2	Erfassen von anderen Sprachen	⇩	⇩	⇧	AF
8	**Spezialelemente / -optionen**				
8.1	Relational: Indexe	⇨	⇧	⇧	AF
8.2	Relational: Trigger	⇩	⇧	⇧	AF
8.3	Relational: Primary Key- / Foreign Key Constraint	⇧	⇨	⇩	F
8.5	Relational: Unique Constraints	⇨	⇨	⇨	F
8.4	Relational: Übrige Constraints	⇨	⇨	⇧	AF
8.5	Relational: Option für referentielle Integrität	⇩	⇨	⇩	F
8.6	Objektorientiert: Methoden	⇩	⇧	⇧	AF
8.7	Objektorientiert: Schnittstellen	⇩	⇧	⇧	AF
9	**Modellierungseditor (GUI)**				
9.1	Zoomfunktionalität	⇩	⇧	⇧	AF
9.2	Ausgeklügeltes Layout	⇩	⇧	⇧	AF
9.3	Benutzerfreundlichkeit	⇨	⇨	⇨	NF
9.4	Einfachheit	⇧	⇨	⇨	NF
10	**Sonstiges**				
10.1	Import / Export Funktionalität	⇩	⇧	⇧	AF
10.2	Druckfunktionalität	⇨	⇧	⇧	AF
10.3	Stabilität	⇧	⇧	⇨	NF
10.4	Speicherung des Datenmodells	⇧	⇨	⇩	F
10.5	Speicherung von Programmeinstellungen	⇧	⇩	⇩	F
10.6	Setup zur einfachen Installation	⇧	⇩	⇨	F
10.7	Performance	⇩	⇧	⇧	AF

Tabelle 13 – Funktionsliste

Legenden:

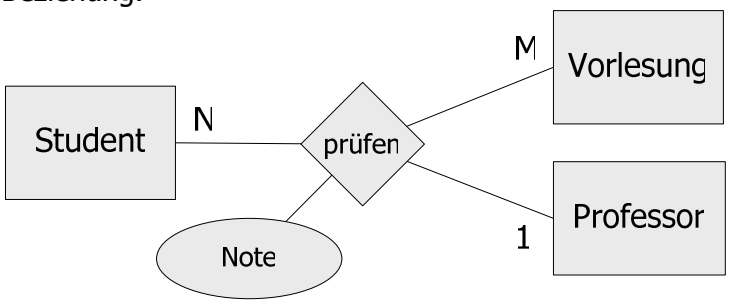
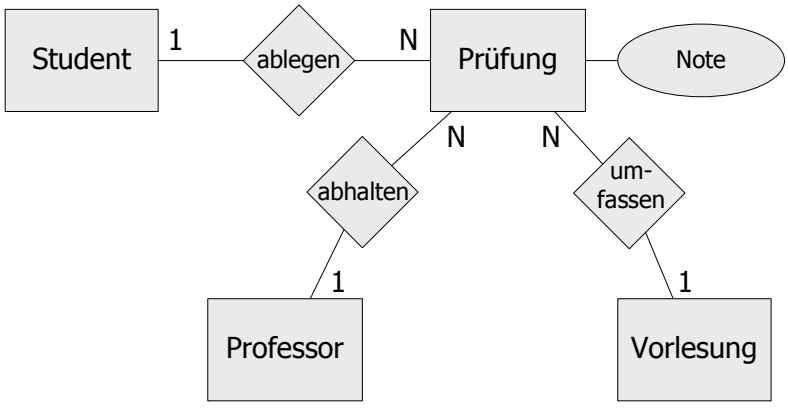
Priorität: Wichtigkeit der Funktion.
Aufwand: Zeitlicher Aufwand für die Implementierung der Funktion.
Risiko: Wahrscheinlichkeit, dass die Funktion bei Abgabe des Projekts nicht implementiert ist.

F: Funktionale Anforderung
NF: Nicht funktionale Anforderung
AF: Auszuschliessende Anforderung

⇧: Hoch
⇨: Mittel
⇩: Niedrig

4.4.2 Explizit ausgeschlossene Funktionen

Die folgenden Funktionen werden in der ersten Version, d.h. im Rahmen dieser Diplomarbeit, nicht umgesetzt.

Nr.	Anforderung	Begründung
1.3	n-stellige Beziehungen	N-stellige Beziehungen werden in der Praxis äusserst selten eingesetzt, da solche Beziehungen meist einen sehr hohen Abstraktionsgrad haben. Unser Standpunkt ist, dass der oder die Studierende mit unserer Software die Umsetzung der 2-stelligen Beziehungen lernen und begreifen soll, bevor er/sie sich an n-stellige Beziehungen wagt. Als Umsetzung einer n-stelligen Beziehung schlagen wir eine manuelle Aufsplittung auf mehrere 2-stellige Beziehungen vor. **Beispiel**: Die Beziehung *prüfen* sagt aus, dass zu einem Paar aus Studenten und Vorlesungen maximal ein Professor als Prüfer zugeordnet wird. Es handelt sich um eine ternäre Beziehung.  **Abbildung 18 – Beziehung *prüfen* als ternäre Beziehung** Diese Sachlage könnte man auch mit 2-stelligen Beziehungen ähnlich modellieren.  **Abbildung 19 – Aufsplittung der ternären Beziehung in 2-stellige Beziehungen** Dabei muss beachtet werden, dass mit einer Aufsplittung Informationen zur partiellen Funktion verloren gehen

Nr.	Anforderung	Begründung
		könnten. In unserem Beispiel sagt die ternäre Beziehung aus, dass zu einem Paar aus Student und Vorlesung höchstens eine Professorin bzw. ein Professor als Prüfer zuständig ist [2]. Das ist aber im Beispiel mit den 2-stelligen Beziehungen so nicht direkt ersichtlich. Diese Abhängigkeit müsste zusätzlich noch verbal formuliert werden. Generell kann gesagt werden, dass es sich bei n-stelligen Beziehungen um sehr komplexe Konstrukte handelt.
1.4	Schwache Entitätsmengen	Die schwachen Entitätsmengen (auch existenzabhängige Entitätsmengen genannt) wären zwar in ihrer Umsetzung nicht komplex, werden jedoch trotzdem nicht umgesetzt. Hat der oder die Studierende einmal die normalen Entitätsmengen mit den Beziehungen und ihre Umsetzung begriffen, sind die schwachen Entitätsmengen nur noch ein kleiner Schritt. Dafür braucht er/sie unsere Software nicht.
1.5	has-a-Beziehung	Aggregation (has-a) gehört ins gleiche Abstraktionskonzept wie die Generalisierung (is-a). Im Gegensatz zur is-a-Beziehung setzen wir die has-a-Beziehung in der ersten Version nicht um. Diese Beziehung kann immer mit einer ganz normalen 1:n-Beziehung ausgedrückt werden.
6	Reverse Engineering	Eine Reverse Engineering-Funktionalität ist für diese Lernsoftware absolut unnötig. Es würde versucht, das Schema einer bestehenden und evtl. schlecht konstruierten Datenbank einzulesen. Das wäre sowieso nur auf der physikalischen Ebene möglich. Wie das ursprüngliche logische Design des Modells ausgesehen hat, kann nicht mehr ermittelt werden. Es könnten sogar logische Fehlinterpretationen daraus resultieren. Dies ist nicht der Weg, den dataMagus gehen möchte und ausserdem didaktisch nicht optimal.
7.2	Unterstützung von anderen natürlichen Sprachen im GUI	Die Software wird so aufgebaut, dass alle sprachlichen Elemente in Ressource-Dateien[6] abgelegt werden. Eine allfällige Übersetzung überlassen wir der Open Source Community.
8.1 8.2	Unterstützung von relationalen Spezialelementen wie Indexe und Trigger	Indexe und Trigger sind sehr datenbankspezifisch und vielmals extrem auf Performancegewinn oder Automatismen ausgelegt. Da der Fokus von dataMagus auf das Verständnis der logischen und physikalischen Ebene gelegt wird, setzen wir solche doch sehr datenbankspezifischen Funktionen in einer ersten Version generell nicht um. Indexe, die jedoch aus einer Primärschlüssel-Angabe entstehen, werden umgesetzt und in die Skriptgenerierung aufgenommen.

[6] Bei Ressource-Dateien handelt es sich um nicht ausführbare Daten, die logisch mit einer Anwendung bereitgestellt werden. Ressourcen können verschiedene Formen von Daten enthalten u.a. Zeichenfolgen und Bilder. Ressource-Dateien können pro Sprache unterschiedlich vorhanden sein und können geändert werden, ohne dass die gesamte Applikation neu kompiliert werden muss.

Nr.	Anforderung	Begründung
8.4	Unterstützung von Constraints (ausgenommen PK, FK und Unique)	Auch Constraints sind je nach Datenbank sehr verschieden. Bei objektorientierten Datenbanken sind sie nicht einmal vorhanden und müssen durch Businessfunktionalität ausprogrammiert werden. Ausgenommen sind Primary Key- und Foreign Key-Constraints sowie Unique-Constraints, die im Zusammenhang mit der Umsetzung benötigt werden, da diese zentrale Modellierungselemente für relationale Systeme darstellen. Sie sind essentiell für die Generierung eines eindeutigen Datensatzes. Diese Eindeutigkeit ist bei objektorientierten Systemen jedoch nicht notwendig, da jedes Objekt eine eindeutige Object-Id vom System bekommt.
8.6 8.7	Unterstützung von objektorientierten Spezialelementen wie Methoden und Schnittstellen	Was die Trigger und Indexe für die relationalen Datenbanken sind, bieten die Methoden und Schnittstellen für objektorientierte. Sie werden bei der Architektur zwar so weit wie möglich angedacht, jedoch nicht als Bestandteil der Software gesehen. Spätere Erweiterungen der Software sollen den Entwicklern aber möglichst einfach gelingen.
9.1	Zoomfunktionalität	Da die Software nicht für grosse Datenbankmodelle mit einigen tausend Entitäten gedacht ist, sondern zu Lernzwecken an der HSLU eingesetzt wird, entfällt eine Zoomfunktionalität.
9.2	Ausgeklügeltes Layout	Die Software soll eine Hilfestellung für Studierende im Datenbankunterricht sein.
10.1	Import / Export Funktionalität	Wir möchten unsere Objekte nicht in bereits fixe Speicherformate für andere Applikationen pressen. Wir werden versuchen, mit einem möglichst gut durchdachten Speicherformat in XML unsere Daten logisch abzulegen. Danach könnten andere Applikationen unser Format z.B. per XSLT[7] einfach in ihr eigenes Format umwandeln.
10.2	Druckfunktionalität	Das Umleiten vom Zeichen in einen Druckerkontext anstelle des Grafikkontexts ist nicht weiter schwierig. Doch das korrekte Errechnen und Verteilen der entsprechenden Grafik auf mehrere Seiten benötigt erweiterte Logik. Dies ist nicht Kernthema dieser Diplomarbeit. Wir werden aber in der Software eine Funktion zum Speichern in ein Bild in Betracht ziehen. Des Weiteren können die Studierenden Screenshots ihres Bildschirms erstellen und diese dann ausdrucken.
10.3	Performance	Die Software wird als Lernsoftware eingesetzt und nicht als marktreife Lösung. Wir stellen daher die funktionale Korrektheit des Programms über die Anforderung an eine möglichst performante Applikation.

Tabelle 14 – Ausgeschlossene Funktionen

[7] Mit einer XSL-Transformation kann ein XML-Dokument in ein anderes Dokument umgewandelt werden.

4.5 Software-Layout

Unsere Software besteht aus den vier zentralen Elementen Menüleiste, Navigationsfenster, Eigenschaftsfenster und Modellierungsfläche. Diese sind farblich hervorgehoben, um ein grobes Layout zu erkennen.

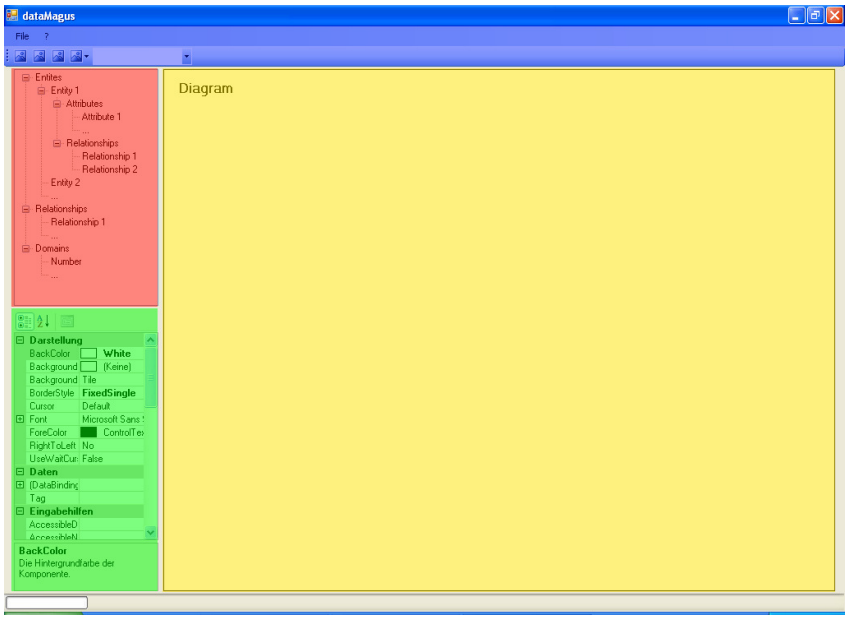


Abbildung 20 – Grobes Layout der Software

Blauer Bereich: Menüleiste

Im blauen Bereich befindet sich die Menüleiste. Sie enthält Funktionen und Programmoptionen.

Roter Bereich: Navigationsfenster

Die Navigation besteht aus einer Baumstruktur. Sie enthält je nach selektierter Ebene (logisch, physikalisch) alle entsprechenden Modellierungselemente. Über die Navigation können an verschiedenen Punkten mit einem Kontextmenü weitere Elemente hinzugefügt resp. gelöscht werden. Der Inhalt der Navigation wird im Modellierungsfenster (gelber Bereich) visualisiert. Neue Elemente werden oben links (Position 0.0) eingefügt. Wie die Elemente schlussendlich dargestellt werden, ist der jeweiligen Notation überlassen. Aus diesem Grunde findet das Verwalten aller Elemente auch im Navigationsbaum und nicht direkt im Modellierungsfenster statt. Ansonsten müsste die Notation viel zu viel Businesslogik implementieren. Weil wir diese Schnittstelle so einfach wie möglich halten möchten, verzichten wir in dieser Version auf ein „richtiges" Modellieren mit Drag-Drop-Funktionalität aus einer Toolbox. Wird nun ein Element im Baum selektiert, werden seine Eigenschaften im Eigenschaftsfenster geladen.

Grüner Bereich: Eigenschaftsfenster

Dieses Fenster dient zum Verwalten von Eigenschaften jeglicher Art auf den Businessobjekten (Entitätsmengen, Beziehungen, Attribute, etc.).

Gelber Bereich: Modellierungsfenster

Hier werden die Schemaelemente je nach eingestellter Notation und selektierter Ebene dargestellt. Die Elemente können mit der Maus positioniert werden.

4.6 Geschäftsfälle

Die folgenden Geschäftsfälle ergeben sich aus dem Anforderungskatalog und umreissen unser System dataMagus in den Grundzügen. Sie sollen es einer externen Person erleichtern, einen Überblick über die funktionalen Anforderungen und deren Abhängigkeiten zu verschaffen. Ausserdem bietet die Modellierung von Geschäftsfällen eine gute Grundlage für das spätere Testen.
Wir halten uns bei der Beschreibung der Geschäftsfälle eher kurz, da diese von der Dozentin nicht explizit verlangt worden sind. Wir versuchen, eine optimale, kurze und prägnante Beschreibung zu liefern.

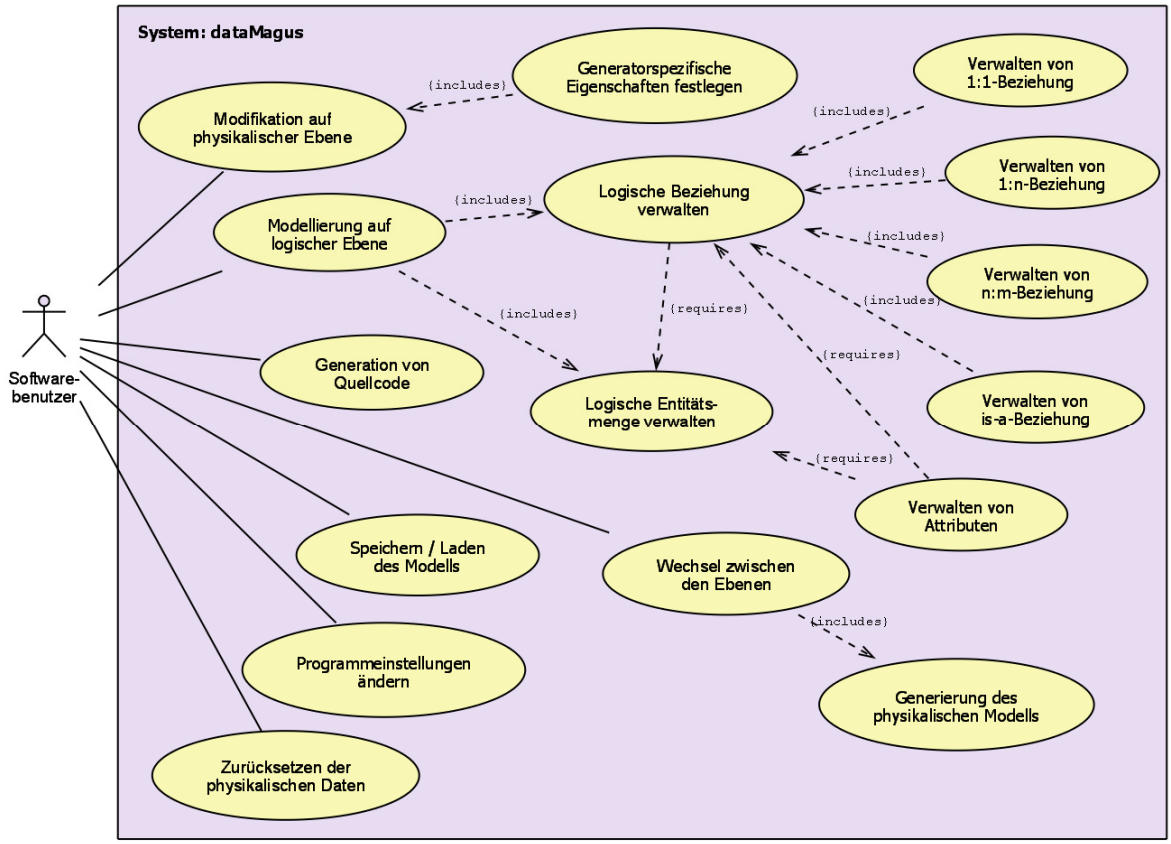


Abbildung 21 – Geschäftsfälle-Diagramm

In unserem System gibt es nur einen Akteur, den "Softwarebenutzer". Dieser Akteur verwendet dataMagus als Lernprogramm und hat anfangs ein nur sehr begrenztes Wissen der Datenmodellierung.

4.6.1 Modellierung

Die Modellierung von Schemas kann mit folgenden Geschäftsfällen umschrieben werden:

Eigenschaft	Beschreibung
Name	**Modellierung auf logischer Ebene**
Akteur	Softwarebenutzer
Kurzbeschreibung	Auf der logischen Ebene können die Entitätsmengen, Beziehungen und Attribute modelliert werden.
Standardablauf	Der Akteur kann über Funktionen im Kontextmenü des Navigationsbaumes Modellierungselemente anlegen.
Bedingungen	Der Akteur befindet sich auf der logischen Ebene.
Regeln	Keine

Tabelle 15 – Beschreibung des Geschäftsfalles *Modellierung auf logischer Ebene*

Eigenschaft	Beschreibung
Name	**Logische Entitätsmenge verwalten**
Akteur	Softwarebenutzer
Kurzbeschreibung	Eine logische Entitätsmenge wird erstellt, modifiziert oder gelöscht.
Standardablauf	Der Akteur kann über die Funktion „Add Entity" im Kontextmenü des Navigationsbaumes auf der Gruppe „Entities" eine neue Entitätsmenge anlegen. Um die Entitätsmenge zu modifizieren, muss der Akteur diese nur selektieren und im Eigenschaftsfenster verändern. Hat der Akteur eine Entität selektiert, findet er im Kontextmenü die Funktionen „Delete Entity"
Bedingungen	Der Akteur befindet sich auf der logischen Ebene.
Regeln	Wird ein Entitätsmengenelement gelöscht, werden seine Attribute sowie auch die Beziehungen (mit Attributen) zu anderen Entitätsmengen entfernt. Zusätzlich werden allfällig gespeicherte Benutzerinteraktionen zur Entität gelöscht sowie jegliche gespeicherte physikalische Änderungen von physikalischen Elementen, die aufgrund der gelöschten Entität existieren.

Tabelle 16 – Beschreibung des Geschäftsfalles *Logische Entitätsmenge verwalten*

Eigenschaft	Beschreibung
Name	**Logische Beziehung verwalten**
Akteur	Softwarebenutzer
Kurzbeschreibung	Eine logische Beziehung zwischen zwei Entitätsmengen wird erstellt, modifiziert oder gelöscht.
Standardablauf	Der Akteur kann über die Funktion „Add Relationship" im Kontextmenü des Navigationsbaumes eine neue Beziehung anlegen. Dafür muss er eine Entität selektiert haben. Um die Beziehung zu modifizieren, muss der Akteur diese nur selektieren und im Eigenschaftsfenster verändern. Hat der Akteur eine Beziehung selektiert, findet er im Kontextmenü die Funktion „Delete Relationship"
Bedingungen	Der Akteur befindet sich auf der logischen Ebene. Es muss mindestens eine Entitätsmenge im System vorhanden sein.
Regeln	Es müssen immer beide Beziehungsteilnehmer angegeben werden. Wird ein Beziehungselement gelöscht, werden seine Attribute auch entfernt. Zusätzlich werden jegliche gespeicherte physikalische Änderungen von physikalischen Elementen, die aufgrund der gelöschten Beziehung existieren, eliminiert.

Tabelle 17 – Beschreibung des Geschäftsfalles *Logische Beziehung verwalten*

Eigenschaft	Beschreibung
Name	**Verwalten von Attributen**
Akteur	Softwarebenutzer
Kurzbeschreibung	Ein logisches Attribut einer Entitätsmenge oder Beziehung wird erstellt, modifiziert oder gelöscht.
Standardablauf	Der Akteur kann in der logischen Ebene über die Funktion „Add Attribute" im Kontextmenü des Navigationsbaumes ein neues Attribut anlegen. Dafür muss er eine Entitätsmenge oder Beziehung selektiert haben. Um das Attribut zu modifizieren, muss der Akteur dieses nur selektieren und im Eigenschaftsfenster verändern. Hat der Akteur ein Attribut selektiert, findet er im Kontextmenü die Funktion „Delete Attribute"
Bedingungen	Es muss mindestens eine Entitätsmenge im System vorhanden sein.
Regeln	Es werden die gespeicherten physikalischen Änderungen, die auf den physikalischen Elementen aufgrund dieses logischen Attributs bestehen, gelöscht.

Tabelle 18 – Beschreibung des Geschäftsfalles *Verwalten von Attributen*

Eigenschaft	Beschreibung
Name	**Verwalten von 1:1-Beziehung**
Akteur	Softwarebenutzer
Kurzbeschreibung	Eine logische 1:1-Beziehung zwischen zwei Entitätsmengen wird erstellt, modifiziert oder gelöscht.
Standardablauf	Ist zu verwalten wie eine logische Beziehung. Die Kardinalität wird automatisch festgelegt.
Bedingungen	Keine
Regeln	Die Definition von Schlüsselattributen ist nicht erlaubt.

Tabelle 19 – Beschreibung des Geschäftsfalles *Verwalten von 1:1-Beziehung*

Eigenschaft	Beschreibung
Name	**Verwalten von 1:n-Beziehung**
Akteur	Softwarebenutzer
Kurzbeschreibung	Eine logische 1:n-Beziehung zwischen zwei Entitätsmengen wird erstellt, modifiziert oder gelöscht.
Standardablauf	Ist zu verwalten wie eine logische Beziehung. Die Kardinalität wird automatisch festgelegt.
Bedingungen	Keine
Regeln	Es ist keine Definition von Schlüsselattributen erlaubt.

Tabelle 20 – Beschreibung des Geschäftsfalles *Verwalten von 1:n-Beziehung*

Eigenschaft	Beschreibung
Name	**Verwalten von n:m-Beziehung**
Akteur	Softwarebenutzer
Kurzbeschreibung	Eine logische n:m-Beziehung zwischen zwei Entitätsmengen wird erstellt, modifiziert oder gelöscht.
Standardablauf	Ist zu verwalten wie eine logische Beziehung. Die Kardinalität wird automatisch festgelegt.
Bedingungen	Keine
Regeln	Keine

Tabelle 21 – Beschreibung des Geschäftsfalles *Verwalten von n:m-Beziehung*

Eigenschaft	Beschreibung
Name	**Verwalten von is-a-Beziehung**
Akteur	Softwarebenutzer
Kurzbeschreibung	Eine logische is-a-Beziehung zwischen zwei Entitätsmengen wird erstellt, modifiziert oder gelöscht.
Standardablauf	Ist zu verwalten wie eine logische Beziehung. Die Kardinalität wird automatisch festgelegt.
Bedingungen	Keine
Regeln	Es dürfen dabei keine zirkulären Beziehungen entstehen (somit ist eine Selbstbeziehung verboten). Es ist keine Mehrfachvererbung erlaubt. D.h. die Entitätsmenge darf nur eine solche Beziehung zu einem Supertyp besitzen. Es sind keine Attribute auf einer is-a-Beziehung erlaubt.

Abbildung 22 – Beschreibung des Geschäftsfalles *Verwalten von is-a-Beziehung*

4.6.2 Physikalische Modelle

Für die Umsetzung und Bearbeitung der physikalischen Modelle sind die folgenden Geschäftsfälle zuständig:

Eigenschaft	Beschreibung
Name	**Wechsel zwischen den Ebenen**
Akteur	Softwarebenutzer
Kurzbeschreibung	Zwischen der logischen und den verschiedenen physikalischen Ebenen kann gewechselt werden.
Standardablauf	Der Akteur wählt die jeweilige Ebene in einem Dropdown der Menüleiste aus.
Bedingungen	Keine
Regeln	Wird von der logischen auf eine physikalische Ebene gewechselt, wird der Generationsprozess erneut durchgeführt (siehe Geschäftsfall „Generierung des physikalischen Modells").

Tabelle 22 – Beschreibung des Geschäftsfalles *Wechsel zwischen den Ebenen*

Eigenschaft	Beschreibung
Name	**Generierung des physikalischen Modells**
Akteur	Softwarebenutzer
Kurzbeschreibung	Ein physikalisches Modell wird aufgrund des logischen Modells erstellt.
Standardablauf	Das logische Modell mit den Elementen wird in das entsprechend ausgewählte physikalische Modell überführt. Hierbei kann es auch noch zu Benutzerinteraktionen kommen. Die getroffenen Entscheidungen müssen dann intern gespeichert werden. Falls schon gespeicherte Benutzerinteraktionen vorhanden sind, werden diese auf ihre Gültigkeit überprüft. Sind sie für das aktuelle Schema nicht mehr gültig, wird der Softwarebenutzer wiederum abgefragt, ansonsten werden die gespeicherten Entscheidungen verwendet. Nach dem erfolgreichen Umsetzen des physikalischen Schemas bekommen alle passenden Generatoren die Gelegenheit, ihre intern gespeicherten Daten mit dem neuen Schema gegenzuprüfen und falls notwendig, zu löschen.
Bedingungen	Keine
Regeln	Siehe Umsetzungsprozess.

Tabelle 23 – Beschreibung des Geschäftsfalles *Generierung des physikalischen Modells*

Eigenschaft	Beschreibung
Name	**Modifikation auf physikalischer Ebene**
Akteur	Softwarebenutzer
Kurzbeschreibung	Auf der physikalischen Ebene können nachträglich Eigenschaften verändert werden. Eine Modellierung ist auf dieser Ebene nicht mehr möglich.
Standardablauf	Um die physikalischen Elemente zu modifizieren, muss der Akteur diese nur selektieren und im Eigenschaftsfenster verändern.
Bedingungen	Der Akteur befindet sich auf einer physikalischen Ebene.
Regeln	Das Programm merkt sich im Hintergrund die gemachten physikalischen Änderungen.

Tabelle 24 – Beschreibung des Geschäftsfalles *Modifikation auf physikalischer Ebene*

Eigenschaft	Beschreibung
Name	**Zurücksetzen der physikalischen Daten**
Akteur	Softwarebenutzer
Kurzbeschreibung	Die gespeicherten physikalischen Daten (Benutzerinteraktionen oder sonstige physikalische Änderungen) können zurückgesetzt werden. Dies hat die Konsequenz, dass der Softwarebenutzer beim nächsten Umsetzen wieder nach Aktionen abgefragt wird und sich die physikalischen Modelle wieder im Anfangszustand befinden.
Standardablauf	Der Akteur kann über einen Menüeintrag entweder alle gespeicherten Daten, nur die Benutzerinteraktionen oder nur die physikalischen Änderungen auf dem Schema löschen.
Bedingungen	Der Akteur befindet sich auf der logischen Ebene.
Regeln	Keine

Tabelle 25 – Beschreibung des Geschäftsfalles *Zurücksetzen der physikalischen Daten*

4.6.3 Generierung von Quellcode

Die folgenden Geschäftsfälle dienen zur Aufnahme von generatorspezifischen Eigenschaften und zur Generierung des Quellcodes:

Eigenschaft	Beschreibung
Name	**Generatorspezifische Eigenschaften festlegen**
Akteur	Softwarebenutzer
Kurzbeschreibung	Auf der physikalischen Ebene können je nach Generator auf den jeweiligen Elementen spezifische Eigenschaften zur Quellcodeerstellung gespeichert werden.
Standardablauf	Dem jeweiligen Generator wird der Typ des selektierten Elementes und dessen spezifische Daten mitgeteilt (falls vorhanden). Er übermittelt danach unserer Software eine Anzeigemöglichkeit (z.B. Fenster).
Bedingungen	Der Akteur befindet sich auf einer physikalischen Ebene. Der Generator muss seine Daten selber verwalten.
Regeln	Keine

Tabelle 26 – Beschreibung des Geschäftsfalles *Generatorspezifische Eigenschaften festlegen*

Eigenschaft	Beschreibung
Name	**Generation von Quellcode**
Akteur	Softwarebenutzer
Kurzbeschreibung	Aufgrund der eingestellten physikalischen Ebene und des ausgewählten Generators kann Quellcode zur Erstellung der jeweiligen Datenbank erzeugt werden.
Standardablauf	Der Akteur kann über einen Menüeintrag einen Generatordialog öffnen. Danach wählt er den entsprechenden Generator und kann noch Zusatzeinstellungen zur Erzeugung des Quellcodes vornehmen. Danach wird der generierte Quellcode in einem Viewer angezeigt.
Bedingungen	Der Akteur befindet sich auf der physikalischen Ebene. Es sind Generatoren vorhanden.
Regeln	Keine

Tabelle 27 – Beschreibung des Geschäftsfalles *Generation von Quellcode*

4.6.4 Speichern/Laden

Folgender Geschäftsfall definiert das Speichern und Laden des Modells:

Eigenschaft	Beschreibung
Name	**Speichern/Laden des Modells**
Akteur	Softwarebenutzer
Kurzbeschreibung	Speichern und Laden des logischen Modells und der geänderten physikalischen Daten.
Standardablauf	Speichern: Der Akteur wählt im Datei-Menü den Punkt Speichern oder Speichern unter... aus. Laden: Der Akteur wählt im Datei-Menü den Punkt Öffnen aus. Der Akteur befindet sich auf der logischen Ebene.
Bedingungen	Keine
Regeln	Keine

Tabelle 28 – Beschreibung des Geschäftsfalles *Speichern/Laden des Modells*

4.6.5 Programmeinstellungen

Folgender Geschäftsfall definiert die Verwaltung von Programmeinstellungen:

Eigenschaft	Beschreibung
Name	**Programmeinstellungen ändern**
Akteur	Softwarebenutzer
Kurzbeschreibung	Der Akteur kann Programmoptionen einstellen.
Standardablauf	Über einen Menüeintrag kann sich der Benutzer die Programmoptionen anzeigen lassen und ggf. Änderungen vornehmen.
Bedingungen	Keine
Regeln	Keine

Tabelle 29 – Beschreibung des Geschäftsfalles *Programmeinstellungen ändern*

4.7 Analysen

Die nachfolgenden Analysen dienen der Entscheidungsfindung und beinhalten bereits konkrete Ideen zur Implementation. Die Architektur der Software ist massgeblich von unseren Entscheidungen abhängig.

4.7.1 Analyse zur Erweiterbarkeit

Aus den Produktfunktionen gehen Anforderungen der Erweiterbarkeit des Programms für Notationen, physikalische Ebenen, Modellierungselemente und Datenbankdialekte oder Zielsysteme hervor. Um festzulegen, wie jede dieser vier verschiedenen Anforderungen erweiterbar gemacht werden kann, analysieren wir zuerst die verschiedenen technischen Varianten, die zur Verfügung stehen. Danach ziehen wir ein Fazit und legen uns pro Anforderung auf eine Variante fest.

4.7.1.1 Variante 1: Erweiterung durch Source-Code-Anpassung

Die Erweiterung der Variante 1 besteht eigentlich nur in der generisch aufgebauten und flexiblen Architektur der Software. Um eine Erweiterung zu implementieren, muss der Source Code der Applikation erweitert werden.

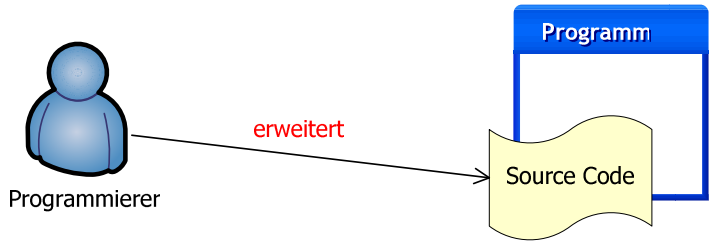


Abbildung 23 – Erweiterbarkeit durch Source-Code-Anpassung

4.7.1.2 Variante 2: Erweiterung durch PlugIns

Erweiterungen werden über PlugIns (oder Add-Ons) gesteuert. Dies sind vordefinierte generische Schnittstellen, auf die von aussen her zugegriffen werden kann. Sie interagieren mit dem System in einer einfachen Art und Weise. Der Vorteil liegt darin, dass die interne Implementation der Software weder komplett verstanden, noch darauf zugegriffen werden muss. Der Nachteil besteht in der Implementation einer solchen Technologie für diese Software. Die Architektur muss auf PlugIns Rücksicht nehmen und evtl. Businessfunktionalität abstrahieren und umsetzen können.

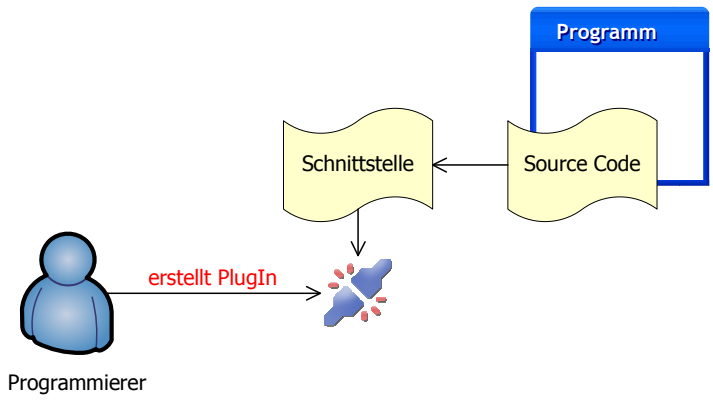


Abbildung 24 – Erweiterbarkeit durch PlugIns

4.7.1.3 Variante 3: Erweiterung durch Konfiguration

Es gibt eine Konfigurationsschnittstelle für Erweiterungen. Diese Variante schränkt die Möglichkeiten für die Erweiterbarkeit sehr ein. Es können nur Sachen verändert werden, die über die Konfigurationsschnittstelle einstellbar sind. Dies bedeutet, dass sich die jeweiligen Komponenten in der Software sehr einfach extrem flexibel konfigurieren lassen – eine Erweiterung im eigentlichen Sinne ist nur mit sehr viel Aufwand und einer evtl. eigenen zu interpretierenden Befehlssprache möglich. Der Vorteil liegt darin, dass eine Erweiterung sehr einfach und meist auch ohne Programmierfähigkeiten durchführbar ist.

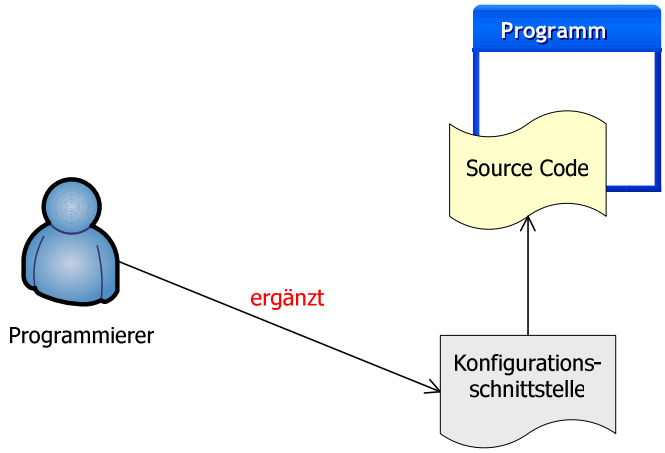


Abbildung 25 – Erweiterbarkeit durch Konfiguration

4.7.1.4 Fazit: Entscheidungen

Aus obigen Varianten entscheiden wir uns wie folgt:

Anforderung	Entscheid und Begründung
Erweiterbarkeit der Notationen	Bei der Erweiterbarkeit von Notationen entscheiden wir uns für die Variante 2. Sie bietet eine sehr flexible und somit ideale Plattfor. Variante 1 ist zu unflexibel und Variante 3 zu aufwändig. Variante 2 ist der goldene Mittelweg.
Erweiterungen der Datenbankdialekte oder Zielsysteme	Hier eignet sich hervorragend die Variante 2 mit ihrer PlugIn-Technologie, weil es für eine externe Umsetzung sehr flexibel ist. Ein Entwickler eines PlugIns kann seinen Datenbankdialekt grundsätzlich so implementieren wie er will. Er muss sich nur an unsere Schnittstellendefinition halten.
Erweiterungen der physikalischen Ebenen	Die Erweiterung der physikalischen Ebenen ist nicht eine Kernfunktionalität der Software. Wir sehen darin momentan auch wenig Sinn, weil die relationalen und objektorientierten physikalischen Ebenen ja bereits abgedeckt sind. Mit der Variante 1 sollte es einem Entwickler zu einem späteren Zeitpunkt problemlos möglich sein, weitere Ebenen einzubauen. Selbstverständlich stellen wir intern eine Reihe von Interfaces zur Verfügung, die implementiert werden können, was die Erweiterung für den Programmierer vereinfacht.
Erweiterung der Modellierung-selemente	Für die Erweiterung der Modellierungselemente legen wir uns auf die Variante 1 fest. Die gängigsten Elemente werden von uns implementiert. Zu einem späteren Zeitpunkt können immer noch weitere Elemente implementiert werden. Wir werden hier wiederum mit Interfaces arbeiten.

Tabelle 30 – Entscheidungen zur Erweiterbarkeit

4.7.2 Analyse der Notationen

Im *Kapitel 3.4 Notationen für Datenbankmodelle* haben wir einige Notationen aufgelistet. Nun werden wir die passenden Notationen für die verschiedenen Ebenen unserer Applikation evaluieren.

4.7.2.1 Variante 1: Eine Notation für alle Ebenen

Wir implementieren genau eine Notation für die logische und alle physikalischen Ebenen. Dies bedeutet, dass wir eine eigene Notation erfinden müssten, weil keine der uns bekannten Notationen die jeweiligen Bedürfnisse aller Ebenen abdecken kann. Mit der Chen- oder der MinMax-Notation ist die Modellierung auf logischer Ebene sehr einfach und sinnvoll. Es sind auch die beiden einzigen Notationen, die Attribute auf einer Beziehung zulassen. Die Krähenfuss- oder die IDEF1x-Notation hingegen unterstützt keine oder nur sehr begrenzte Möglichkeiten auf einer logisch abstrakten Ebene. Dafür sind sie für eine physikalisch relationale Ebene sehr zweckmässig und gut verständlich. Eine ODL-Notation wie auch UML eignen sich wiederum nur für die physikalisch objektorientierte Darstellung. Somit müssten wir eine eigene evtl. zusammengesetzte Notation erfinden.

4.7.2.2 Variante 2: Passende Notation für jede Ebene

In Variante 1 wurden die Vor- und Nachteile der entsprechenden Notationen kurz erläutert. Diese Variante vertritt die Idee, auf jeder entsprechenden Ebene mit der dafür vorgesehenen Notation aufzusetzen. Diese Lösung hätte als Nachteil, dass nicht nur eine, sondern für unser Projekt insgesamt drei verschiedene Notationen umgesetzt werden müssten. Aus Nähe zum Lehrmittel und Verbreitungsgrad würde auf der logischen Ebene mit Chen bzw. mit der modifizierten Chen-Notation modelliert. Die physikalisch relationale Ebene würde mit der Krähenfuss-Notation implementiert, weil diese Notation sehr einfach zu lesen ist. Die physikalisch objektorientierte Ebene stellt ihre Elemente in der ODL-Notation dar.

4.7.2.3 Fazit

Wir entscheiden uns aus folgenden Gründen für die Variante 2: Obwohl für die beschriebenen Notationen kein eindeutiger Standard existiert, kann die Entwicklung einer eigenen Notation für Studierende sehr verwirrend sein und sie hätten auch keinen Vergleich zum Lehrmittel.

4.8 Definition und Umsetzung

In diesem Kapitel werden die logischen Modellierungselemente und deren Umsetzung auf die physikalischen Ebenen beschrieben. Zur Vereinfachung werden wir die Elemente in der Chen-Notation visualisieren. Es versteht sich, dass unsere Elemente nicht von einer Notation abhängig sind. Ebenfalls gilt für die nachfolgende Ausführung:
❶ = erste Entitätsmenge (oder links)
❷ = zweite Entitätsmenge (oder rechts)

4.8.1 Definition der logischen Modellierungselemente

Aus den Produktzielen und dem Anforderungskatalog gehen die folgenden Modellierungselemente hervor, die wir in einer ersten Version auf der logischen Ebene umsetzen werden. Selbstverständlich sieht unser Konzept vor, dass die Elemente jederzeit durch eine Erweiterung des Programms ergänzt werden können.

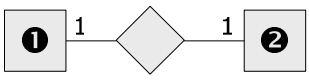
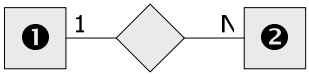
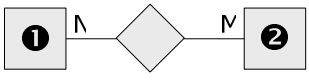
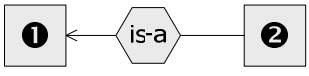
Element	Visuelle Darstellung	Beschreibung
Entitätsmenge		Eine Entität ist ein materielles oder abstraktes Objekt der Wirklichkeit. Eine Entitätsmenge ist eine Typisierung gleichartiger Entitäten.
Attribut		Ein Attribut ist eine Eigenschaft einer Entitätsmenge oder einer Beziehung.
1:1-Beziehung		Eine 1:1-Beziehung ist eine Beziehung zwischen zwei Entitätsmengen, bei der jede Entität aus der ersten Entitätsmenge mit höchstens einer Entität aus der zweiten Entitätsmenge in Beziehung stehen kann.
1:n-Beziehung		Eine 1:n-Beziehung ist eine Beziehung zwischen zwei Entitätsmengen, bei der jede Entität aus der ersten Entitätsmenge mit beliebig vielen Entitäten aus der zweiten Entitätsmenge in Beziehung stehen kann. Jede Entität aus der zweiten Entitätsmenge kann mit höchstens einer Entität aus der ersten Entitätsmenge in Beziehung stehen.
n:m-Beziehung		Eine n:m-Beziehung ist eine Beziehung zwischen zwei Entitätsmengen, bei der jede Entität aus der ersten Entitätsmenge mit beliebig vielen Entitäten aus der zweiten Entitätsmenge in Beziehung stehen kann, und umgekehrt.
is-a-Beziehung		Eine is-a-Beziehung ist eine Vererbung bzw. eine Kategorisierung, wobei die erste Entitätsmenge einen Supertyp bezeichnet und die zweite Entitätsmenge einen Subtyp.

Tabelle 31 – Modellierungselemente

Folgende Unterkapitel beschreiben die Modellierungselemente auf der **logischen** Ebene. Die jeweils aufgeführte Liste der Eigenschaften sind abstrakt zu verstehen und nicht abschliessend. Es werden nur die wichtigsten Eigenschaften aufgeführt.

4.8.2 Definition der Entitätsmenge

In der ersten Version machen wir keine Unterscheidung zwischen normalen und schwachen Entitätsmengen. Eine Entitätsmenge charakterisiert sich nur durch einen Namen. Eine Entitätsmenge kann Attribute besitzen. Zusätzlich führt sie auch eine Liste der Schlüsselattribute.

Eigenschaften einer Entitätsmenge:
- Name
- Liste von Attributen
- Liste von Schlüsselattributen

4.8.3 Definition der Attribute

Wir unterstützen normale Attribute und Schlüsselattribute und keine zusammengesetzten oder berechneten Attribute. Ein Attribut ist charakterisiert durch einen Namen. Attribute können einer Entitätsmenge oder auch auf einer 1:1-, 1:n- oder n:m-Beziehung modelliert werden.

Eigenschaften eines Attributs:
- Name

4.8.4 Allgemeine Definition der Beziehungen

Da wir n-stellige Beziehungen explizit ausgeschlossen haben, sind an einer Beziehung nur jeweils zwei Entitätsmengen beteiligt. Ein Spezialfall der 2-stelligen Beziehung ist die rekursive Beziehung. Diese ist aber bei einer is-a-Beziehung ausgeschlossen, weil es sich dabei um einen zirkulären Bezug handeln würde.

4.8.4.1 Definition der 1:1-Beziehung

Wie oben bereits erwähnt, definieren wir unsere 1:1-Beziehung als eine Beziehung zwischen zwei Entitätsmengen, bei der jede Entität aus der ersten Entitätsmenge mit **höchstens** einer Entität aus der zweiten Entitätsmenge in Beziehung stehen kann.

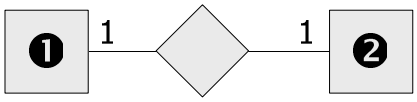


Abbildung 26 – 1:1-Beziehung in der Chen-Notation

Die Aussage *höchstens* definiert also die folgenden zulässigen Kardinalitäten:

Kardinalität	Beschreibung
(0 oder 1) zu (1 oder 0)	Jede Entität aus der ersten Entitätsmenge kann mit **höchstens einer** Entität aus der zweiten Entitätsmenge in Beziehung stehen.
1 zu (0 oder 1)	Jede Entität der ersten Entitätsmenge kann mit **höchstens einer** Entität der zweiten Entitätsmenge in Beziehung stehen. Jede Entität der zweiten Entitätsmenge steht mit **genau einer** Entität der ersten Entitätsmenge in Beziehung. Die umgekehrte Variante gilt natürlich auch.
1 zu 1	Jede Entität der ersten Entitätsmenge steht mit **genau einer** Entität der zweiten Entitätsmenge in Beziehung, und umgekehrt.

Tabelle 32 – Zulässige Kardinalitäten für eine 1:1-Beziehung

Die oben stehenden Kardinalitäten erlauben eine präzisere Aussage zur Beziehung auf der logischen Ebene.

Um eine Beziehung genauer zu beschreiben, kann man an den jeweiligen „Ausgängen" der Beziehung einen Rollennamen hinterlegen. Eine Rolle dokumentiert die Art der Beziehungen der Entitätsmengen. Auf einer 1:1-Beziehung können Attribute definiert werden, jedoch keine Schlüsselattribute, da es sich ansonsten um eine n:m-Beziehung handeln würde.

Eigenschaften einer 1:1-Beziehung:

- Name
- Entität 1
- Entität 2
- Kardinalität Entität 1
  - Null bis eins
  - Genau eins
- Kardinalität Entität 2
  - Null bis eins
  - Genau eins
- Name der Rolle 1
- Name der Rolle 2
- Liste von Attributen

4.8.4.2 Definition der 1:n-Beziehung

Wir definieren eine 1:n-Beziehung als eine Beziehung zwischen zwei Entitätsmengen, bei der jede Entität aus der ersten Entitätsmenge mit **beliebig vielen** Entitäten aus der zweiten Entitätsmenge in Beziehung stehen kann. Jede Entität aus der zweiten Entitätsmenge kann mit **höchstens** einer Entität aus der ersten Entitätsmenge in Beziehung stehen.

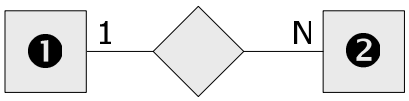


Abbildung 27 – 1:n-Beziehung in der Chen-Notation

Die Aussagen *beliebig viele* und *höchstens* erlauben folgende Kardinalitäten:

Kardinalität	Beschreibung
(0 oder 1) zu (beliebig vielen)	Jede Entität aus der ersten Entitätsmenge kann mit **beliebig vielen** Entitäten aus der zweiten Entitätsmenge in Beziehung stehen. Jede Entität aus der zweiten Entitätsmenge kann mit **höchstens einer** Entität aus der ersten Entitätsmenge in Beziehung stehen.
1 zu (mindestens 1)	Jede Entität der ersten Entitätsmenge kann mit **höchstens einer** Entität der zweiten Entitätsmenge in Beziehung stehen. Jede Entität der zweiten Entitätsmenge steht mit **genau einer** Entität der ersten Entitätsmenge in Beziehung.
1 zu (beliebig vielen)	Jede Entität der ersten Entitätsmenge kann mit **beliebig vielen** Entitäten der zweiten Entitätsmenge in Beziehung stehen. Jede Entität der zweiten Entitätsmenge steht mit **genau einer** Entität der ersten Entitätsmenge in Beziehung.
(0 oder 1) zu (mindestens 1)	Jede Entität der ersten Entitätsmenge steht mit **mindestens einer** Entität der zweiten Entitätsmenge in Beziehung. Jede Entität der zweiten Entitätsmenge kann mit **höchstens einer** Entität der ersten Entitätsmenge in Beziehung stehen.
(0 oder 1) zu (exakt n)	Jede Entität der ersten Entitätsmenge steht mit einer **exakt definierten Anzahl** Entitäten (n > 1) der zweiten Entitätsmenge in Beziehung. Jede Entität der zweiten Entitätsmenge kann **mit höchstens einer** Entität der ersten Entitätsmenge in Beziehung stehen.

Tabelle 33 – Zulässige Kardinalitäten für eine 1:n-Beziehung

Auf einer 1:n-Beziehungen können Attribute definiert werden, jedoch keine Schlüsselattribute, da es sich ansonsten um eine n:m-Beziehung handeln würde. Es können Rollennamen definiert werden.

Eigenschaften einer 1:n-Beziehung:

- Name
- Entität 1
- Entität 2
- Kardinalität Entität 1
  - Null oder eins
  - Genau eins
- Kardinalität Entität 2
  - Null, eins und mehr
  - Eins und mehr
  - Exakte Angabe
- Name der Rolle 1
- Name der Rolle 2
- Liste von Attributen

4.8.4.3 Definition der n:m-Beziehung

Wir definieren eine n:m-Beziehung als eine Beziehung zwischen zwei Entitätsmengen, bei der jede Entität aus der ersten Entitätsmenge mit **beliebig vielen** Entitäten aus der zweiten Entitätsmenge in Beziehung stehen kann, und umgekehrt.

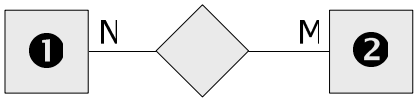


Abbildung 28 – n:m-Beziehung in der Chen-Notation

Die Aussage *beliebig viel* erlaubt folgende Kardinalitäten:

Kardinalität	Beschreibung
(mindestens 1) zu (mindestens 1)	Jede Entität der ersten Entitätsmenge steht mit **mindestens einer** Entität der zweiten Entitätsmenge in Beziehung, und umgekehrt.
(mindestens 1) zu (beliebig vielen)	Jede Entität der ersten Entitätsmenge kann mit **beliebig vielen** Entitäten der zweiten Entitätsmenge in Beziehung stehen. Jede Entität der zweiten Entitätsmenge steht mit **mindestens einer** Entität der ersten Entitätsmenge in Beziehung.
(beliebig viele) zu (beliebig vielen)	Jede Entität der ersten Entitätsmenge kann mit **beliebig vielen** Entitäten der zweiten Entitätsmenge in Beziehung stehen, und umgekehrt.
(exakt n) zu (exakt m)	Jede Entität der ersten Entitätsmenge steht mit einer **exakt definierten Anzahl** Entitäten (n > 1 und m > 1) der zweiten Entitätsmenge in Beziehung, und umgekehrt.

Tabelle 34 – Zulässige Kardinalitäten für eine n:m-Beziehung

Auf einer n:m-Beziehung können normale Attribute und Schlüsselattribute definiert werden Ebenfalls können Rollennamen zugewiesen werden.

Eigenschaften einer n:m-Beziehung:

- Name
- Entität 1
- Entität 2
- Kardinalität Entität 1
  - Null, eins oder mehr
  - Eins oder mehr
  - Exakte Angabe
- Kardinalität Entität 2
  - Null, eines oder mehr
  - Eins oder mehr
  - Exakte Angabe
- Name der Rolle 1
- Name der Rolle 2
- Liste von Attributen
- Liste von Schlüsselattributen

4.8.4.4 Definition der is-a-Beziehung

Eine is-a-Beziehung drückt aus, dass es sich bei der untergeordneten Entitätsmenge (die zweite Entitätsmenge) um eine Spezialisierung der übergeordneten Entitätsmenge (die erste Enitätsmenge) handelt.

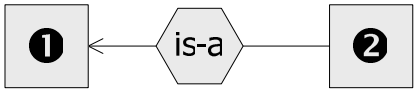


Abbildung 29 – Is-a-Beziehung in der Chen-Notation

Diese Aufteilung ermöglicht es, spezifische Eigenschaften zu modellieren, die nicht unbedingt für jede Entität der übergeordneten Entitätsmenge relevant sind. Hier sind eigentlich keine Rollennamen nötig. Wir lassen jedoch eine Definition zu. Auf einer is-a-Beziehung können keine Attribute definiert werden. Sobald eine Entitätsmenge als zweite Entitätsmenge an einer is-a-Beziehung beteiligt ist, können auf ihr keine Schlüsselattribute mehr definiert werden. Sie erhält die Schlüssel von der ersten Entitätsmenge.

Eigenschaften einer is-a-Beziehung:

- Name
- Entität Supertyp
- Entität Subtyp
- Name der Rolle Supertyp
- Name der Rolle Subtyp

Eine Mehrfachvererbung wird schon während des Modellierens unterbunden, d.h. folgendes Konstrukt ist nicht möglich:

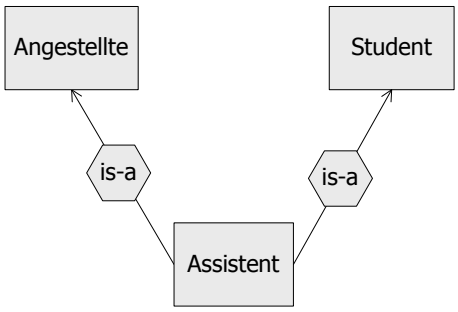


Abbildung 30 – Mehrfachvererbung mit is-a-Beziehungen

Des Weiteren werden auch zirkuläre Bezüge verboten. Auch folgendes Konstrukt ist nicht möglich:

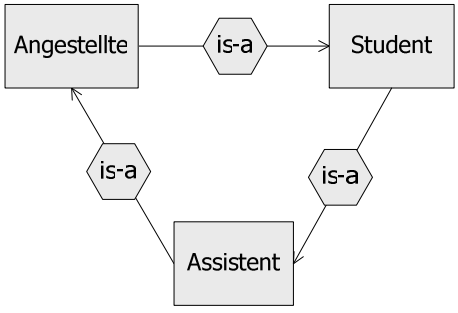


Abbildung 31 – Zirkuläre Bezüge mit is-a-Beziehungen

4.8.5 Umsetzung der logischen Ebene in die physikalischen Ebenen

Die nachfolgende Ausführung beschreibt zuerst die Umsetzung der logischen Modellierelemente in die physikalischen Ebenen. Es werden Regeln definiert, nach welchen die Umsetzung erfolgen muss. Des Weiteren werden physikalische Eigenschaften aufgeführt, die später auch zum Finden der implementationsbedingten Klassen und Attribute in der Design-Phase helfen können. Die Eigenschaften und die Umsetzung sind sehr stark von der eingesetzten physikalischen Ebene abhängig. Am Schluss dieses Kapitels werden die benötigten Elemente der jeweiligen physikalischen Ebene kurz zusammengefasst.

Die formale Beschreibung der Umsetzung auf die physikalisch relationale Ebene wird an die Schreibweise der Schema-Definition angelehnt, wie sie in [2] ab S. 70 verwendet wird. Im Speziellen ergänzen wir sie noch um folgende Definitionen: Primärschlüssel werden unterstrichen gezeichnet, Fremdschlüssel-Attribute mit einem (fk)-Postfix und Unique-Constraints mit einem (u)-Postfix markiert.

Die formale Beschreibung der Umsetzung auf die physikalisch objektorientierte Ebene wird an den ODMG-Standard angelehnt.

Die Namensgebung bei den Umsetzungsregeln ist von uns logisch gewählt, damit die Beispiele gut verständlich sind. Die Namen sind nach keinem speziellen Mechanismus vergeben worden.

4.8.5.1 Umsetzung der Entitätsmengen

Umsetzung auf die relationale Ebene

Die logische Entitätsmenge wird in Form einer Tabelle umgesetzt. Sofern nichts anderes definiert ist, wird der logische Name der Entitätsmenge als Vorschlag für den Tabellennamen übernommen. Eine relationale Tabelle sollte einen Primärschlüssel haben. Ist auf der logischen Entitätsmenge kein Schlüsselattribut modelliert, so wird immer eine technische Schlüssel-Id „Id" generiert.

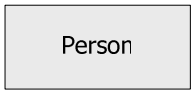
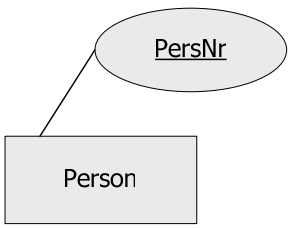
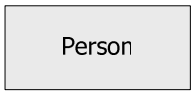
Logische Ebene	Physikalische Ebene
	`Person : { Id : integer }`
	`Person : { PersNr }`

Tabelle 35 – Umsetzung der Entitätsmenge auf die physikalisch relationale Ebene

Umsetzung auf die objektorientierte Ebene
Die logische Entitätsmenge wird in Form einer Klasse umgesetzt. Der Klassennamen wird wiederum von der logischen Entitätsmenge übernommen. In der objektorientierten Welt sind normalerweise keine Schlüsseldefinitionen notwendig, da jedes Objekt eine eindeutige Object-Id vom jeweils eingesetzten System bekommt. Der ODL-Standard jedoch kennt eine *key*-Angabe bei dem Schlüsselwort *extent*.

Beispiel:
```
class Person (extent AllePersonen key PersNr)
{
  attribute short PersNr;
}
```

Logische Ebene	Physikalische Ebene

	`class Person`
	`{`
	`}`

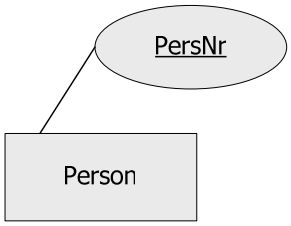
	`class Person`
	`{`
	`  attribute short PersNr;`
	`}`

Tabelle 36 – Umsetzung der Entitätsmenge auf die physikalisch objektorientierte Ebene

4.8.5.2 Umsetzung der Attribute

Umsetzung auf die relationale Ebene

Ein Attribut wird in Form eines Tabellenfeldes (auch Column genannt) umgesetzt. Sofern nichts anderes definiert ist, wird der logische Name des Attributs als Vorschlag übernommen. Zusätzlich kann pro Attribut ein Datentyp definiert werden und angegeben werden, ob das Feld Nullwerte akzeptiert oder nicht. Ist das Attribut Teil des Schlüssels, so wird Feld als Primärschlüssel markiert und sichergestellt, dass es keine Nullwerte akzeptiert. Bei einem Feld, das Teil des Primärschlüssels ist, kann die Einstellung der Nullwerte nicht mehr geändert werden.
Im Zusammenhang mit den Datentypen muss sichergestellt werden, dass bei einer Spalte, die Teil eines Fremdschlüssel-Contraints ist, der Datentyp nicht gesetzt werden kann. Der Datentyp für solche Felder wird immer durch den referenzierten Primärschlüssel-Constraint bestimmt. Demnach muss beim Setzen eines Datentyps auf einem Feld, welches Teil eines Primärschlüssel-Constraints ist, auch alle Datentypen vorhandener Fremdschlüssel-beziehungen nachgeführt werden.

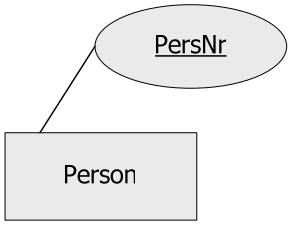
Logische Ebene	Physikalische Ebene
	`Person : { PersNr }`

Tabelle 37 – Umsetzung eines Attributs auf die physikalisch relationale Ebene

Umsetzung auf die objektorientierte Ebene

Ein Attribut wird in eine Eigenschaft (auch Property genannt) der Klasse umgewandelt, in der es definiert ist. Die Schlüsselangaben werden übernommen. Der Name wird vom logischen Attribut her als Vorschlag übernommen. Es kann pro Attribut ein Datentyp definiert werden.

Logische Ebene	Physikalische Ebene

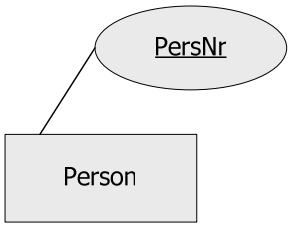
	`class Person`
	`{`
	`  attribute short PersNr;`
	`}`

Tabelle 38 – Umsetzung eines Attributs auf die physikalisch objektorientierte Ebene

4.8.5.3 Allgemeines zu den Beziehungen

Beziehungen können nicht mehr eins-zu-eins in physikalische Beziehungen übersetzt werden. Je nach Typ oder einer expliziten Benutzerinteraktion gibt es mehrere unterschiedliche Varianten der Umsetzung. Ausserdem ist die Umsetzung sehr stark vom Typ der physikalischen Ebene (d.h. relational, objektorientiert, etc.) abhängig. Es gibt z.B. bei relationalen Datenbanken die referentielle Integrität. Wie wir solche Integritätschecks in unserem Programm behandeln, ist unter *Kapitel 4.8.6 Referentielle Integrität* nachzulesen. Objektorientierte Modelle kennen keine oder nur begrenzte Integritätschecks (*inverse*-Angaben bei ODL). Meist werden sie über Businessregeln gelöst.

Im Nachfolgenden werden die einzelnen Beziehungstypen genauer betrachtet und die Umsetzungsregeln definiert.

4.8.5.4 Umsetzung der 1:1-Beziehung

Folgende Schemas sind logisch modellierbar:

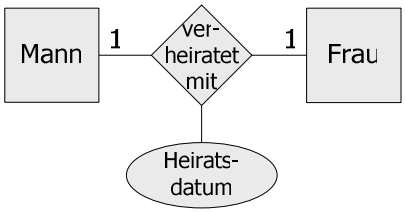
Nr.	Logische Ebene	Beschreibung
1		Nach schweizerischem Recht kann ein Mann nur mit höchstens einer Frau zur selben Zeit verheiratet sein.
2		Zur Beziehung *verheiratetMit* wird noch das Attribut Heiratsdatum definiert.

Tabelle 39 – Varianten logischer 1:1-Beziehungen

Zu jeder Variante existiert der Spezialfall der rekursiven Beziehung. Dieser kann aber genau gleich aufgelöst werden wie die jeweilige „normale" Variante.

Umsetzung auf die relationale Ebene

Tabelle 40 zeigt die Umsetzungsvarianten der obengenannten Modelle. Für die Umsetzung der logischen Konstrukte wird vorausgesetzt, dass für jede logisch modellierte Entitätsmenge ein Schlüsselattribut vorhanden ist. Dieses wird entweder vom Benutzer selbst definiert oder in einem vorgängig durchgeführten Prozess nach unseren Regeln generiert („Id"). Gewisse Konstrukte können auf unterschiedliche Varianten physikalisch abgebildet werden. Falls die Umsetzung nicht klar ist, wird die gewünschte Variante beim Benutzer abgefragt. Müssen sog. Zwischentabellen generiert werden, können diese optional mit einem generierten technischen Schlüssel oder mit dem zusammengesetzten natürlichen Schlüssel erstellt werden.
Die unten beschriebenen Regeln ändern sich für selbst definierte Schlüssel oder auch zusammengesetzte Schlüssel nicht. Bei zusammengesetzten Schlüsseln muss nur beachtet werden, dass alle Schlüsselattribute in eine Fremdschlüssel-Beziehung aufgenommen werden.

Nr.	Variante	Abbildung
1	1	Mann : { Id, verheiratetMit(fk)(u) } Frau : { Id }
1	2	Mann : { Id } Frau : { Id, verheiratetMit(fk)(u) }
1	3	Mann : { Id } Frau : { Id } verheiratetMit : { FrauId(fk), MannId(fk)(u) }
1	4	Mann : { Id } Frau : { Id } verheiratetMit : { MannId(fk), FrauId(fk)(u) }
1	5	Mann : { Id } Frau : { Id } verheiratetMit : { Id, FrauId(fk)(u), MannId(fk)(u)}
2	1	Mann : { Id, verheiratetMit(fk), Heiratsdatum } Frau : { Id }
2	2	Mann : { Id } Frau : { Id, verheiratetMit(fk), Heiratsdatum }
2	3	Mann : { Id } Frau : { Id } verheiratetMit : { FrauId(fk), MannId(fk)(u), Heiratsdatum }
2	4	Mann : { Id } Frau : { Id } verheiratetMit : { MannId(fk), FrauId(fk)(u), Heiratsdatum }
2	5	Mann : { Id } Frau : { Id } verheiratetMit : { Id, FrauId(fk)(u), MannId(fk)(u), Heiratsdatum }

Tabelle 40 – Umsetzung der Varianten der 1:1-Beziehung auf die physikalisch relationale Ebene

Bei den Varianten 3, 4 und 5 muss beachtet werden, dass {FrauId} und {MannId} niemals zusammen einen Schlüssel bilden können, da per Definition ein Datensatz über die {FrauId} bzw. den {MannId} schon eindeutig identifizierbar ist. Vielmehr müssen beide Schlüsselinformationen in der Datenbank getrennt realisiert werden. Für unsere Variante 3 bedeutet das konkret, dass {FrauId} der Primärschlüssel ist und dass {MannId} mit dem Unique-Constraint (u) versehen werden muss.

Die Realisierung der 1:1-Beziehung mit einer Zwischentabelle hat folgende Vorteile [5]:

- Keine Nullwerte bei Personen, die nicht verheiratet sind
- Schnelleres Durchsuchen der Beziehung (mit Indexunterstützung)
- Die Suche nach Ehepartnern wird in beide Richtungen gleich gut unterstützt.

Schlüsselattribute auf Beziehungen sind nicht erlaubt, weil dann der Schlüssel nicht mehr minimal wäre und man des Weiteren eine n:m-Beziehung modelliert hätte.

Aus obigen Varianten entsteht folgende Benutzeraktion, die abgefragt werden muss:

Benutzerinteraktion**:**

- 1. Schritt: Frage, ob…
  A) der Fremdschlüssel auf der linken Tabelle (erste Entitätsmenge) erstellt werden soll.
  B) der Fremdschlüssel auf der rechten Tabelle (zweite Entitätsmenge) erstellt werden soll.
  C) eine Zwischentabelle generiert werden soll.

- 2. Schritt: Falls C) gewählt wurde, Frage ob...
  A) ein technischer Primärschlüssel erstellt werden soll.
  B) für den natürlichen Schlüssel der Primärschlüssel der linken Tabelle (erste Entitätsmenge) verwendet werden soll.
  C) für den natürlichen Schlüssel der Primärschlüssel der rechten Tabelle (zweite Entitätsmenge) verwendet werden soll.

Umsetzung auf die objektorientierte Ebene
Obengenannte Modelle werden wie folgt auf die objektorientierte Ebene umgesetzt.

Nr.	Variante	Abbildung

1	1	`class Mann`
		`{`
		`  relationship Frau Gattin inverse Frau::Gatte;`
		`}`
		
		`class Frau`
		`{`
		`  relationship Mann Gatte inverse Mann::Gattin;`
		`}`

2	1	`class Mann`
		`{`
		`  relationship verheiratetMit verheiratetMit inverse`
		`  verheiratet::Gatte;`
		`}`
		
		`class Frau`
		`{`
		`  relationship verheiratetMit verheiratetMit inverse`
		`  verheiratet::Gattin;`
		`}`
		
		`class verheiratetMit`
		`{`
		`  attribute Date Heiratsdatum;`
		`  relationship Mann Gatte inverse`
		`  Mann::verheiratetMit;`
		`  relationship Frau Gattin inverse`
		`  Frau::verheiratetMit;`
		`}`

Tabelle 41 – Umsetzung der Varianten der 1:1-Beziehung auf die physikalisch objektorientierte Ebene

4.8.5.5 Umsetzung der 1:n-Beziehung

Folgende Schemas sind logisch modellierbar:

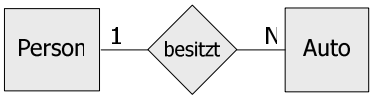
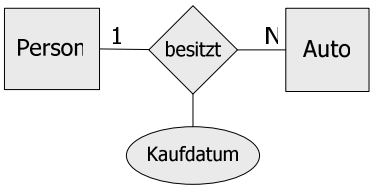
Nr.	Logische Ebene	Beschreibung
1		Eine Person kann beliebig viele Autos besitzen. Ein Auto gehört zu höchstens einer Person.
2		Zur Beziehung *besitzt* wird noch das Attribut *Kaufdatum* definiert.

Tabelle 42 – Varianten logischer 1:n-Beziehungen

Zu jeder Variante existiert der Spezialfall der rekursiven Beziehung. Dieser kann aber genau gleich aufgelöst werden wie die jeweilige „normale" Variante.

Umsetzung auf die relationale Ebene

Obengenannte Modelle können wie folgt auf die relationale Ebene abgebildet werden:

Nr.	Variante	Abbildung
1	1	`Person : { Id }` `Auto : { Id, PersonId(fk) }`

1	2	`Person : { Id }`
		`Auto : { Id }`
		`Besitzt: { Id, PersonId(fk), AutoId(fk)(u) }`

1	3	`Person : { Id }`
		`Auto : { Id }`
		`Besitzt: { AutoId(fk), PersonId(fk) }`

2	1	`Person : { Id }` `Auto : { Id, PersonId(fk), Kaufdatum }`

2	2	`Person : { Id }`
		`Auto : { Id }`
		`Besitzt: { Id, PersonId(fk), AutoId(fk)(u),`
		`Kaufdatum }`

2	3	`Person : { Id }`
		`Auto : { Id }`
		`Besitzt: { AutoId(fk), PersonId(fk), Kaufdatum }`

Tabelle 43 – Umsetzung der Varianten der 1:n-Beziehung auf die physikalisch relationale Ebene

Schlüsselattribute auf Beziehungen sind nicht erlaubt, weil dann der Schlüssel nicht mehr minimal wäre und man des Weiteren auch eine n:m-Beziehung modelliert hätte.

Zwischentabellen können gewünscht sein, um viele Nullwerte in der Beziehungstabelle zu vermeiden [5]. Die Studierenden sollen mit folgender Benutzeraktion gezwungen werden, sich diesen Sachverhalt zu überlegen.

Benutzerinteraktion:

- 1. Schritt: Frage, ob…
  A) eine Zwischentabelle generiert werden soll.

- 2. Schritt: Falls A) gewählt wurde, Frage ob...
  A) ein technischer Primärschlüssel erstellt werden soll.
  B) der natürliche Schlüssel verwendet werden soll.

Umsetzung auf die objektorientierte Ebene
Obengenannte Modelle können wie folgt auf die objektorientierte Ebene abgebildet werden:

Nr.	Variante	Abbildung

1	1	`class Person`
		`{`
		`  relationship set<Auto> besitzt inverse`
		`  Auto::Besitzer;`
		`}`
		
		`class Auto`
		`{`
		`  relationship Person Besitzer inverse`
		`  Auto::besitzt;`
		`}`

2	1	`class Person`
		`{`
		`  relationship set<Auto> besitzt inverse`
		`  Auto::Besitzer;`
		`}`
		
		`class Auto`
		`{`
		`  attribute Date Kaufdatum;`
		`  relationship Person Besitzer inverse`
		`  Person::besitzt;`
		`}`

Nr.	Variante	Abbildung

2	2	`class Person`
		`{`
		`  relationship set<besitzt> besitzt inverse`
		`  besitzt::Besitzer;`
		`}`
		
		`class Auto`
		`{`
		`  relationship besitzt besitzt inverse`
		`  besitzt::gehört;`
		`}`
		
		`class besitzt`
		`{`
		`  attribute Date Kaufdatum;`
		`  relationship Person Besitzer inverse`
		`  Person::besitzt;`
		`  relationship Auto gehört inverse Auto::besitzt;`
		`}`

Tabelle 44 – Umsetzung der Varianten der 1:n-Beziehung auf die physikalisch objektorientierte Ebene

Aus obigen Varianten entstehen die folgenden Benutzeraktionen, die je nach modellierter Konstellation abgefragt werden müssen.

Benutzerinteraktionen:

- Bei zwei Entitätsmengen ohne Attribute auf der Beziehung
  - Keine Benutzerinteraktion

- Bei zwei Entitätsmengen mit Attributen auf der Beziehung
  - Frage, ob…
    A) die Attribute auf die rechte Klasse (zweite Entitätsmenge) transferiert werden sollen.
    B) eine Zwischenklasse generiert werden soll.

4.8.5.6 Umsetzung der n:m-Beziehung

Folgende Schemas sind logisch modellierbar.

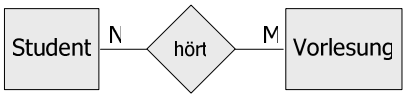
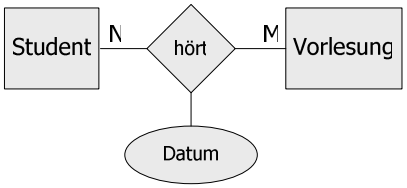
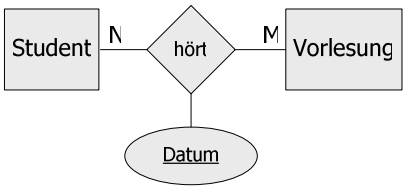
Nr.	Logische Ebene	Beschreibung
1		Eine Vorlesung wird von mehreren Studenten besucht. Ein Student hört sich mehrere Vorlesungen an.
2		Zur Beziehung *hört* wird noch das Attribut *Datum* definiert.
3		Ein Student kann die gleiche Vorlesung nur einmal pro Tag besuchen. Das Attribut *Datum* wird zusätzlich als Schlüsselattribut markiert.

Tabelle 45 – Varianten logischer n:m-Beziehungen

Zu jeder Variante existiert der Spezialfall der rekursiven Beziehung. Dieser kann aber genau gleich aufgelöst werden wie die jeweilige „normale" Variante.

Umsetzung auf die relationale Ebene
Obengenannte Modelle können wie folgt auf die relationale Ebene abgebildet werden:

Nr.	Variante	Abbildung
1	1	Student : { Id } Vorlesung : { Id } hört : { Id, [StudentId(fk), VorlesungId(fk)](u) }
1	2	Student : { Id } Vorlesung : { Id } hört : { StudentId(fk), VorlesungId(fk) }
2	1	Student : { Id } Vorlesung : { Id } hört : { Id, StudentId(fk), VorlesungId(fk), Datum }
2	2	Student : { Id } Vorlesung : { Id } hört : { StudentId(fk), VorlesungId(fk), Datum }
3	1	Student : { Id } Vorlesung : { Id } hört : { Id, [StudentId(fk), VorlesungId(fk), Datum](u) }
3	2	Student : { Id } Vorlesung : { Id } hört : { StudentId(fk), VorlesungId(fk), Datum }

Tabelle 46 – Umsetzung der Varianten der n:m-Beziehung auf die physikalisch relationale Ebene

Aus obigen Varianten entsteht folgende Benutzeraktion:

Benutzerinteraktion:

- Frage ob...
  A) ein technischer Primärschlüssel erstellt werden soll.
  B) der natürliche Schlüssel verwendet werden soll.

Umsetzung auf die objektorientierte Ebene
Obengenannte Modelle können wie folgt auf die objektorientierte Ebene abgebildet werden:

Nr.	Variante	Abbildung

1	1	`class Student`
		`{`
		`  relationship set<Vorlesung> hört inverse`
		`  Vorlesung::Hörer;`
		`}`
		
		`class Vorlesung`
		`{`
		`  relationship set<Student> Hörer inverse`
		`  Student::hört;`
		`}`

2	1	`class Student`
		`{`
		`  relationship set<hört> hört inverse hört::Hörer;`
		`}`
		
		`class Vorlesung`
		`{`
		`  relationship set<hört> hört inverse hört:hört;`
		`}`
		
		`class hört`
		`{`
		`  attribute Date Datum;`
		`  relationship Student Hörer inverse Student:hört;`
		`  relationship Vorlesung hört inverse`
		`  Vorlesung:hört;`
		`}`

3	1	`class Student`
		`{`
		`  relationship set<hört> hört inverse hört::Hörer;`
		`}`
		
		`class Vorlesung`
		`{`
		`  relationship set<hört> hört inverse hört::hört;`
		`}`
		
		`class hört`
		`{`
		`  attribute Date Datum;`
		`  relationship Student Hörer inverse Student::hört;`
		`  relationship Vorlesung hört inverse`
		`  Vorlesung::hört;`
		`}`

Tabelle 47 – Umsetzung der Varianten der n:m-Beziehung auf die physikalisch objektorientierte Ebene

4.8.5.7 Umsetzung der is-a-Beziehung

Folgende Schemas sind logisch modellierbar:

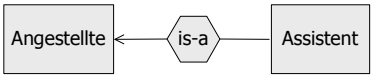
Nr.	Logische Ebene	Beschreibung
1		Ein Assistent ist auch ein Angestellter.

Tabelle 48 – Logische is-a-Beziehung

Umsetzung auf die relationale Ebene

Obengenanntes Modell kann wie folgt auf die relationale Ebene abgebildet werden:

Nr.	Variante	Abbildung
1	1	`Angestellte : { Id }` `Assistent : { AngestellteId(fk) }`

Tabelle 49 – Umsetzung der is-a-Beziehung auf die physikalisch relationale Ebene

Umsetzung auf die objektorientierte Ebene

Obengenanntes Modell kann wie folgt auf die objektorientierte Ebene abgebildet werden:

Nr.	Variante	Abbildung

1	1	`class Angestellte`
		`{`
		`}`
		
		`class Assistant extends Angestellte`
		`{`
		`}`

Tabelle 50 – Umsetzung der is-a-Beziehung auf die physikalisch objektorientierte Ebene

4.8.5.8 Zusammenfassung der physikalischen Elemente

Zusammenfassend kann gesagt werden, dass die Umsetzung der logischen Elemente in die physikalischen Elemente nicht eins-zu-eins erfolgen kann. Die Umsetzung erfordert manchmal zusätzliche Eigenschaften oder auch ganz neue Elemente, die wir so auf der logischen Ebene nicht kennen.

Relationale Ebene

Bei der physikalisch relationalen Ebene gibt es neben der Umsetzung der Entitätsmengen, Beziehungen und Attribute in Tabellen und Feldern auch noch die Generation von Zwischentabellen, Fremdschlüssel-Feldern, Unique- und PK-Constraints. Da wir die Constraints (ausser PK-, FK- und Unique-Constraints) bei den Anforderungen ausgeschlossen haben, wird es kein spezielles Konstrukt zu Constraints geben (auch keine programmiertechnischen Klassen). Der Primarykey-Constraint ist durch eine Auflistung der Primärschlüssel-Felder in der Tabelle definiert und die Fremdschlüssel werden durch Beziehungen[8] mit einem Column-Mapping (PK-Column ←→ FK-Column) abgehandelt. Es bleiben einzig die Unique-Constraints übrig. Da Datenbanken wie der SQL Server 2005 oder die PostgreSQL-Datenbank sowieso pro Unique-Constraint implizit einen Index anlegen, werden wir die Unique-Constraints ebenfalls in Indexe überführen. Dies begründen wir zusätzlich auch mit der Tatsache, dass die Erweiterung des Programms um saubere Constraints eine eher aufwändige Sache ist, die gut durchdacht sein will. Indexe hingegen sind sehr einfach und hier können wir es uns gut vorstellen, dass die Open Source-Community schnell eine Erweiterung vornehmen wird, damit noch weitere Indexe zum relationalen Schema erfasst werden können. Daher bieten wir hier keinen Ansatz für die Constraints, sondern für die Indexe.

Doppelbeziehungen zwischen zwei Entitätsmengen sind grundsätzlich erlaubt, sofern sie nicht durch eine andere Regel ausgeschlossen werden (vgl. Mehrfachvererbung bei einer is-a-Beziehung). Durch Doppelbeziehungen und der Benutzerinteraktion wird es theoretisch möglich, dass zwei Entitätsmengen voneinander abhängig werden. Dies lassen wir jedoch zu.

Logische is-a-Beziehungen werden im relationalen Modell immer mit einer 1:1-Beziehung gelöst; logische n:m-Beziehungen müssen zwingend in zwei 1:n-Beziehungen aufgeteilt werden.

Die Umsetzungsprozesse haben demnach zu folgenden physikalisch relationalen Elementen und deren Eigenschaften geführt:

[8] Diese Beziehungen können nicht mit den Beziehungen der logischen Sicht gleichgesetzt werden. Im relationalen Datenbankmodell gibt es nur noch Relationen, die Beziehungen werden mit Spalten und Foreignkey-Constraints in den entsprechenden Relationen umgesetzt.

- **Tabelle:**
  - Name
  - Liste von Primärschlüssel-Feldern
  - Liste von Feldern (einfache und Fremdschlüssel-Felder)

- **Feld:**
  - Name
  - Datentyp
  - Sind Nullwerte erlaubt?

- **Index:**
  - Name
  - Liste von Feldern
  - Typ (Unique oder normal)

- **1:1-Beziehung:**
  - Name
  - Tabelle 1
  - Tabelle 2
  - Kardinalität Tabelle 1
    - Null bis eins
    - Genau eins
  - Kardinalität Tabelle 2
    - Null bis eins
    - Genau eins
  - Name der Rolle 1
  - Name der Rolle 2
  - Column-Mapping

- **1:n-Beziehung:**
  - Name
  - Tabelle 1
  - Tabelle 2
  - Kardinalität Tabelle 1
    - Null oder eins
    - Genau eins
  - Kardinalität Tabelle 2
    - Null, eins und mehr
    - Eins und mehr
  - Name der Rolle 1
  - Name der Rolle 2
  - Column-Mapping

Bei den Kardinalitäten muss beachtet werden, dass es auf den relationalen Datenbanken keine Angaben zu einer exakten Anzahl Datensätze gibt, die miteinander in Verbindung stehen könnten. Solche Regeln müssten z.B. mit einem Trigger oder Businesslogik in der Applikation gelöst werden. Daher wird eine exakte Angabe in die Kardinalität *Eins und mehr* überführt.
Die Kardinalitäten können im physikalischen Modell nicht mehr geändert werden, da sonst das physikalische Schema nicht mehr mit dem logischen übereinstimmen würde.
Auf den Beziehungen sind keine Felder erlaubt.

Objektorientierte Ebene
Bei der physikalisch objektorientierten Ebene können Zwischenklassen und neue Attribute aus Beziehungen entstehen. Im Gegensatz zum relationalen Modell müssen beim objektorientierten Modell bei Assoziationen keine Mapping-Informationen geführt werden. Die jeweiligen Rollennamen bestimmen die Beziehungs-Referenzobjekte auf den Klassen.

Die Umsetzungsprozesse haben demnach zu folgenden physikalisch objektorientierten Elementen und deren Eigenschaften geführt:

- **Klasse:**
  - Name
  - Liste von Key-Properties
  - Liste von Properties

- **Property:**
  - Name
  - Datentyp

- **is-a-Assoziation:**
  - Name
  - Klasse 1
  - Klasse 2
  - Name der Rolle 1
  - Name der Rolle 2

- **1:1-Assoziation:**
  - Name
  - Klasse 1
  - Klasse 2
  - Kardinalität Klasse 1
    - Null bis eins
    - Genau eins
  - Kardinalität Klasse 2
    - Null bis eins
    - Genau eins
  - Name der Rolle 1
  - Name der Rolle 2

- **1:n-Assoziation:**
  - Name
  - Klasse 1
  - Klasse 2
  - Kardinalität Klasse 1
    - Null oder eins
    - Genau eins
  - Kardinalität Klasse 2
    - Null, eins und mehr
    - Eins und mehr
  - Name der Rolle 1
  - Name der Rolle 2

- **n:m-Assoziation:**
  - Name
  - Klasse 1
  - Klasse 2
  - Kardinalität Klasse 1
    - Null, eins oder mehr
    - Eins oder mehr
  - Kardinalität Klasse 2
    - Null, eins und mehr
    - Eins und mehr
  - Name der Rolle 1
  - Name der Rolle 2

Auch beim objektorientierten Schema gibt es keine exakten Angaben auf den Assoziationen. Diese werden in die Kardinalität *Eins und mehr* überführt. Des Weiteren gibt es auch hier keine Properties auf Assoziationen. Eine logische n:m-Beziehung mit Attributen muss zwingend in zwei 1:n-Assoziationen aufgelöst werden.
Die Kardinalitäten können im physikalischen Modell nicht mehr geändert werden, da sonst das physikalische Schema nicht mehr mit dem logischen übereinstimmen würde.

4.8.6 Referentielle Integrität

Die relationalen Datenbanken lassen bei Beziehungen keine inkonsistente Datenhaltung zu, was unter dem Begriff referentielle Integrität bekannt ist. Ein DELETE- oder UPDATE-Statement wird deshalb zurückgewiesen statt durchgeführt. Um das Statement trotzdem durchzuführen und dabei die Konsistenz zu wahren, können Integritätsregeln definiert werden. Es kann definiert werden, wie abhängige Daten beim Löschen oder Ändern eines Datensatzes behandelt werden sollen. Durch Kaskadierung kann z.B. ein Datensatz und all seine abhängigen Daten gelöscht werden [7] [8]

Im SQL-92 Standard gibt es z.B. für das DELETE- und UPDATE-Statement folgende Regeln:

- **NO ACTION (default)**
  → Durchführen der Operation und Verifizieren der Fremdschlüsselbeziehung am Ende durch das Datenbanksystem.
  Konkret heisst das, dass der Primärschlüssel nur dann gelöscht werden kann, wenn die Semantik der Anweisung und alle involvierten Trigger dafür sorgen, dass die Fremdschlüsselbeziehung nicht verletzt wird.

- **CASCADE**
  → Propagiert die Änderungen.
  Wird z.B. ein Primärschlüssel gelöscht, werden ebenfalls alle abhängigen Datensätze der Fremdschlüsselbeziehung gelöscht.

- **SET NULL**
  → Verweise auf NULL setzen.
  Wird z.B. ein Primärschlüssel gelöscht, werden alle abhängigen Verweise auf NULL gesetzt.

- **SET DEFAULT**
  → Verweise auf Defaultwerte setzen.
  Wird z.B. ein Primärschlüssel gelöscht, werden alle abhängigen Verweise auf den Defaultwert der Spalte gesetzt.

Leider ist es so, dass nicht alle relationalen Datenbanksysteme genau einen Standard einhalten. Die Hersteller definieren vielmehr selber, was sie unterstützen möchten.
Der SQL Server 2000 kennt z.B. nur ON UPDATE CASCADE und ON UPDATE NO ACTION bzw. ON DELETE CASCADE und ON DELETE NO ACTION.
Oracle kennt nicht einmal das ON UPDATE CASCADE – aber eigentlich aus gutem Grunde. CASCADE-Constraints beziehen sich auf Primärschlüssel und diese sollten als solche nach dem Anlegen unveränderbar sein, ansonsten sind es keine. Ein UPDATE auf einem Primärschlüssel ist demzufolge bei einem vernünftigen Design nicht notwendig und deshalb braucht es auch kein ON UPDATE CASCADE.

Für dataMagus bedeutet dies, dass wir per se auf den physikalisch relationalen Beziehungen keine Integritätsregeln anbieten. Diese sind zielsystemabhängig und es wird somit den PlugIns überlassen, diese zu programmieren. Die PlugIns können ja generatorspezifische Eigenschaften in den physikalischen Ebenen auf den einzelnen Elementen ablegen. Für uns bedeutet dies sogleich auch ein Proof of Concept für solche Eigenschaften.

4.8.7 Definition der automatischen Namensgebung und Regeln zur Auflösung der Beziehungen

Der Benutzer kann auf dem logischen Modell seine Entitätsmengen, Beziehungen und Attribute benennen. Diese werden von uns bei der Umsetzung berücksichtigt.
Zusätzlich kann die Umsetzung des logischen Modells in die physikalischen Modelle u.U. die automatische Generierung von Zwischentabellen bzw. –klassen, technische Primärschlüssel oder Fremdschlüssel-Feldern erfordern, für die der Benutzer keinen expliziten Namen angegeben hat. Für solche Objekte werden nun nach untenstehenden Regeln Namen generiert. Der Benutzer kann danach immer noch auf den physikalischen Ebenen die Namen selber ändern.

Die folgenden Beispiele in den Abbildungen sind in keiner korrekten Notation beschrieben, sondern nur einer Chen-, Krähenfuss- oder ODLG-Notation angelehnt. Es sind schematische Zeichnungen.

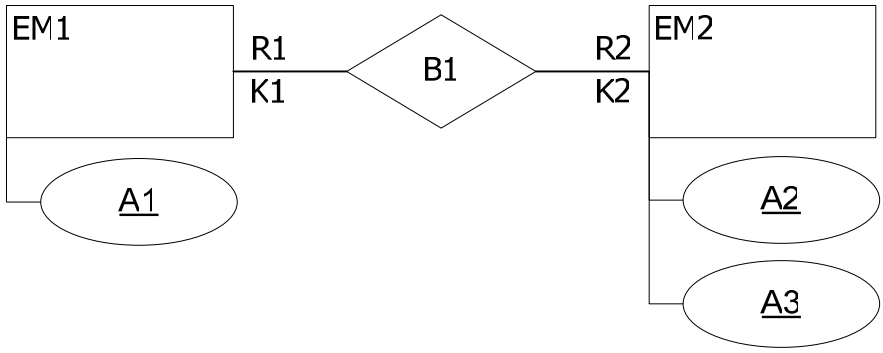


Abbildung 32 - Ausgangslage in der logischen Ebene

Für die Abbildung gelten folgende Regeln/Definitionen:

- EM = Entitätsmenge
- B = Beziehungsname
- A = Attribut
- R = Rollenname

Vorschlagswerte während der logischen Modellierung, d.h. falls nichts anderes definiert ist:

- EM = Entity<#>
  → # wird durch die Anzahl der vorhanden Entitätsmengen + 1 ersetzt
- B = <EM1><EM2>
  → der Beziehungsname wird aus den beiden beteiligten Entitätsmengen gebildet
- A = Attribute<#>
  → # wird durch die Anzahl der Attribute der Entitätsmenge + 1 ersetzt
- R = <EM><B>
  → der Rollenname wird durch die Entitätsmenge und die Beziehung definiert.

Die Namensgebung wird bewusst von uns mit obigen Regeln durchgeführt. Theoretisch müssten wir keine Regeln definieren – wir könnten einfache Namen mit einem angehängten Zähler generieren. Da wir aber davon ausgehen, dass der oder die Studierende für ein sauberes Design auf der logischen Ebene die Entitätsmengen, die Beziehungen, die Attribute und die Rollennamen mit sprechenden Namen versieht, können für die generierten Zwischentabellen oder –klassen bzw. Felder oder Eigenschaften einigermassen verständliche Namen vom System vergeben werden. Dadurch muss der Student auf den physikalischen Ebenen weniger Elemente umbenennen, weil bereits ein sinnvoller Name besteht.

Beispiel:
In folgendem sich durchziehenden Beispiel gehen wir davon aus, dass der oder die Studierende auf der logischen Ebene jedes Element entsprechend benennt. Die Spalte *Systemname* zeigt, wie die Namen aussehen, falls keine benutzerspezifischen Namen angegeben sind.

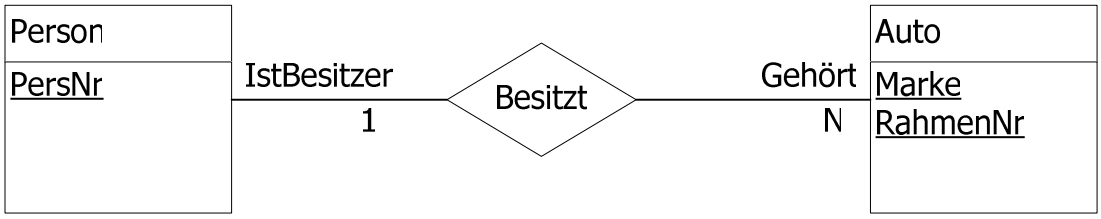


Abbildung 33 – Visualisiertes Beispiel zur Namensgebung auf der logischen Ebene

Element	Benutzername	Systemname, wenn nichts angegeben ist
EM1	Person	`Entity1`
EM2	Auto	`Entity2`
B1	besitzt	`Entity1Entity2`
A1	PersNr	`Attribute1`
A2	Marke	`Attribute2`
A3	RahmenNr	`Attribute3`
R1	istBesitzer	`Entity1Entity1Entity2`
R2	gehört	`Entity2Entity1Entity2`
K1	1	`1`
K2	2	`N`

Tabelle 51 – Beispiel zur Namensgebung auf der logischen Ebene

4.8.7.1 Relationale Ebene

Beziehungen ohne Zwischentabelle

Die Regeln für die Umsetzung auf die physikalisch relationale Ebene sind in Abbildung 34 ersichtlich. Die meisten Namen werden von der logischen Definition beibehalten (schwarze Angaben) und sind in jeden Fall vorhanden. Muss aufgrund der Beziehung bei einer Tabelle ein neues Attribut (z.B. für den Fremdschlüssel) generiert werden, wird die Namensgebung aufgrund des Rollen- und Attributnamens der Komplementärtabelle erzeugt (blaue Angaben). Ist kein Rollenname definiert, so wird der Name der referenzierten Tabelle (hier EM1) verwendet.

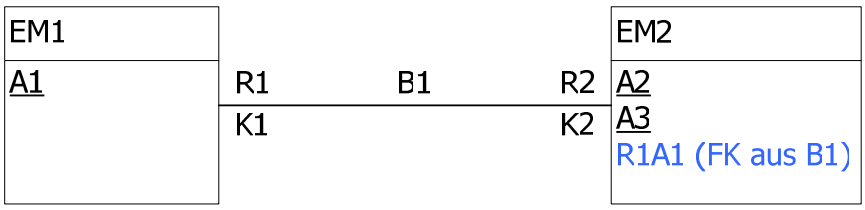


Abbildung 34 – Namensgebung bei Beziehungen ohne Zwischentabellen auf der physikalisch relationalen Ebene

Beispiel:
Hier wird nun das obige Beispiel auf die physikalische Ebene umgesetzt. Wir gehen von einer aussagekräftigen Namensgebung auf der logischen Ebene aus. Allerdings gibt es Felder, die nirgends in der logischen Ebene auftauchen und automatisch durch den Umsetzungsprozess generiert werden. Solche Felder sind blau markiert. Wie man sieht, entstehen aus unseren definierten Regeln lesbare und verständliche Namen.

Abbildung 35 – Visualisiertes Beispiel einer Beziehung ohne Zwischentabelle zur Namensgebung auf der physikalisch relationalen Ebene

Element	Benutzername	Systemname, falls nichts angegeben ist
EM1	Person	`Entity1`
EM2	Auto	`Entity2`
B1	Besitzt	`Entity1Entity2`
A1	PersNr	`Attribute1`
A2	Marke	`Attribute2`
A3	RahmenNr	`Attribute3`
R1	IstBesitzer	`Entity1Entity1Entity2`
R2	Gehört	`Entity2Entity1Entity2`
R1A1	IstBesitzerPersNr	`Entity1Entity1Entity2Attribute1`
K1	1	`1`
K2	2	`N`

Tabelle 52 – Beispiel einer Beziehung ohne Zwischentabelle zur Namensgebung auf der physikalisch relationalen Ebene

Beziehungen mit Zwischentabellen

Muss eine Beziehung mit einer Zwischentabelle aufgelöst werden, muss das System automatisch eine neue Tabelle (die Zwischentabelle B1) und zwei Beziehungen (Beziehungen EM1 ⇔ B1 und B1 ⇔ EM2) generieren. Auch hier gilt: die schwarzen Angaben werden direkt vom logischen Schema übernommen, die blauen sind von uns generiert.
Die Zwischentabelle erhält den Namen der Beziehung. Die Kardinalitäten der neu generierten Beziehungen können auch übernommen werden, wobei die Enden bei der Zwischentabelle vom logischen Modell stammt (natürlich „verkehrt herum") und die anderen Enden mit einer eins bezeichnet werden müssen. Muss für die Zwischentabelle ein automatischer Primärschlüssel generiert werden, so heisst dieser immer „Id".
Die Attributnamen für z.B. Fremdschlüsselbeziehungen werden wiederum vom Rollen- und Attributnamen der Komplementärtabelle abgeleitet.

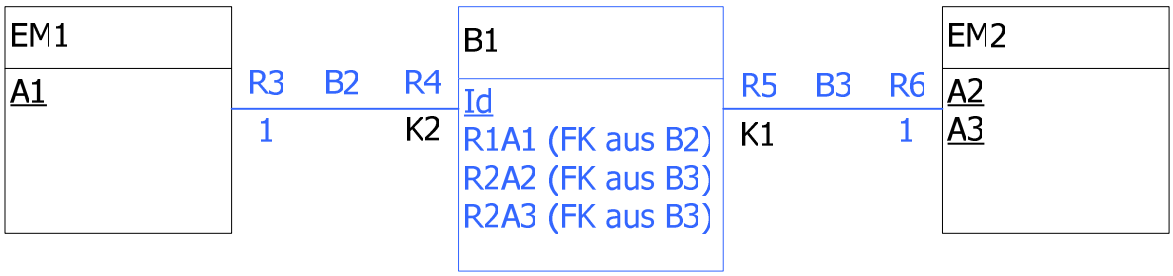


Abbildung 36 – Namensgebung bei Beziehungen mit Zwischentabellen auf der physikalisch relationalen Ebene

Die weiteren Elemente werden analog der logischen Regeln wie folgt benannt:

- B2 = EM1B1
- B3 = B1EM2
- R3 = EM1B2
- R4 = B1B2
- R5 = B1B3
- R6 = EM2B3

Beispiel:
Die blauen Angaben werden wie gewohnt vom System generiert. Die rot markierten Namen werden zwar generiert, spielen aber für die weitere Verwendung (z.B. für die Skriptgenerierung) eine eher untergeordnete Rolle. Aus Platzgründen sind diese nur in der Tabelle beschrieben.

Abbildung 37 – Visualisiertes Beispiel einer Beziehung mit Zwischentabelle zur Namensgebung auf der physikalisch relationalen Ebene

Element	Benutzername	Systemname, falls nichts angegeben ist
EM1	Person	Entity1
EM2	Auto	Entity2
B1	Besitzt	Entity1Entity2
B2	PersonBesitzt	Entity1Entity1Entity2
B3	BesitztAuto	Entity1Entity2Entity2
A1	PersNr	Attribute1
A2	Marke	Attribute2
A3	RahmenNr	Attribute3
R1	IstBesitzer	Entity1Entity1Entity2
R2	Gehört	Entity2Entity1Entity2
R3	PersonPersonBesitzt	Entity1Entity2
R4	BesitztPersonBesitzt	Entity1Entity2Entity1Entity1Entity2
R5	BesitztBesitztAuto	Entity1Entity2Entity1Entity2Entity2
R6	AutoBesitztAuto	Entity2Entity1Entity2Entity2
R1A1	IstBesitzerPersNr	Entity1Entity1Entity2Attribute1
R2A1	GehörtMarke	Entity2Entity1Entity2Attribute1
R2A2	GehörtRahmenNr	Entity2Entity1Entity2Attribute2
K1	1	1
K2	2	N

Tabelle 53 – Beispiel zur Beziehung mit Zwischentabelle zur Namensgebung auf der physikalisch relationalen Ebene

4.8.7.2 Objektorientierte Ebene

Beziehungen ohne Zwischenklasse

Die Regeln für die Umsetzung auf die physikalisch objektorientierte Ebene sind in Abbildung 38 ersichtlich. Die meisten Namen werden von der logischen Definition beibehalten (schwarze Angaben) und sind in jeden Fall vorhanden.
Muss aufgrund der Beziehung bei einer Tabelle eine neue Eigenschaft angelegt werden, wird die Namensgebung aufgrund des Rollennamens des angrenzenden Beziehungendes erzeugt (blaue Farbe).

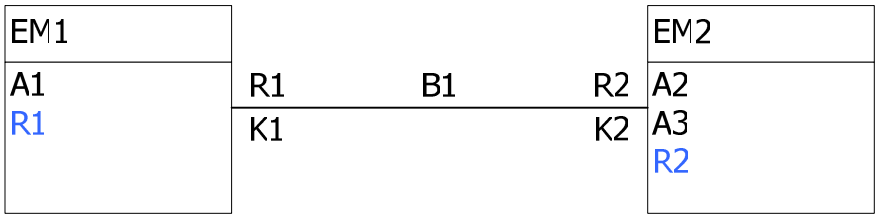


Abbildung 38 – Namensgebung bei Beziehungen ohne Zwischenklassen auf der physikalisch objektorientierten Ebene

Beispiel:
Wir gehen wieder von unserem logischen Beispiel aus. Die blauen Eigenschaften werden vom System generiert. Es entstehen für diese auch auf der objektorientierten Ebene lesbare und verständliche Namen.

Abbildung 39 – Visualisiertes Beispiel einer Beziehung ohne Zwischentabelle zur Namensgebung auf der physikalisch objektorientierten Ebene

Element	Benutzername	Systemname, falls nichts angegeben ist
EM1	Person	`Entity1`
EM2	Auto	`Entity2`
B1	Besitzt	`Entity1Entity2`
A1	PersNr	`Attribute1`
A2	Marke	`Attribute2`
A3	RahmenNr	`Attribute3`
R1	IstBesitzer	`Entity1Entity1Entity2`
R2	Gehört	`Entity2Entity1Entity2`
K1	1	`1`
K2	2	`N`

Tabelle 54 – Beispiel einer Beziehung ohne Zwischentabelle zur Namensgebung auf der physikalisch objektorientierten Ebene

Beziehungen mit Zwischenklassen

Muss eine Beziehung mit einer Zwischenklasse aufgelöst werden, muss das System automatisch eine neue Klasse (die Zwischenklasse B1) und zwei Beziehungen (Beziehungen EM1 ⇔ B1 und B1 ⇔ EM2) generieren. Auch hier gilt: die schwarzen Angaben werden direkt vom logischen Schema übernommen, die blauen sind von uns generiert.
Die Zwischenklasse erhält den Namen der Beziehung. Die Kardinalitäten der neu generierten Beziehungen können auch übernommen, wobei das Ende bei der Zwischenklasse vom logischen Modell stammt (natürlich „verkehrt herum") und das andere Ende mit einer eins bezeichnet werden muss.
Die Eigenschaftennamen für Referenzen werden wiederum vom Rollennamen der Komplementärtabelle abgeleitet.

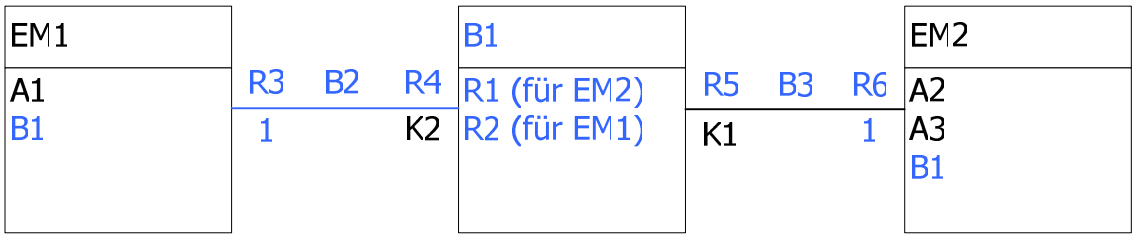


Abbildung 40 – Namensgebung bei Beziehungen mit Zwischenklassen auf der objektorientierten Ebene

Die weiteren Elemente werden analog der logischen Regeln wie folgt benannt:

- B2 = EM1B1
- B3 = B1EM2
- R3 = EM1B2
- R4 = B1B2
- R5 = B1B3
- R6 = EM2B3

Beispiel:
Die blauen Angaben werden wie üblich vom System generiert. Die rot markierten Namen werden zwar generiert, spielen aber für die weitere Verwendung (z.B. für die Skriptgenerierung) eine eher untergeordnete Rolle. Aus Platzgründen sind diese nur in der Tabelle beschrieben.

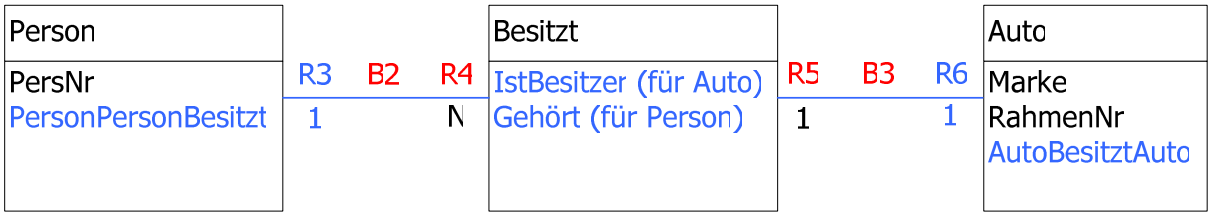


Abbildung 41 – Visualisiertes Beispiel einer Beziehung mit Zwischentabelle zur Namensgebung auf der physikalisch objektorientierten Ebene

Element	Benutzername	Systemname
EM1	Person	Entity1
EM2	Auto	Entity2
B1	Besitzt	Entity1Entity2
B2	PersonBesitzt	Entity1Entity1Entity2
B3	BesitztAuto	Entity1Entity2Entity2
A1	PersNr	Attribute1
A2	Marke	Attribute2
A3	RahmenNr	Attribute3
R1	IstBesitzer	Entity1Entity1Entity2
R2	Gehört	Entity2Entity1Entity2
R3	PersonPersonBesitzt	Entity1Entity2
R4	BesitztPersonBesitzt	Entity1Entity2Entity1Entity1Entity2
R5	BesitztBesitztAuto	Entity1Entity2Entity1Entity2Entity2
R6	AutoBesitztAuto	Entity2Entity1Entity2Entity2
K1	1	1
K2	2	N

Tabelle 55 – Beispiel einer Beziehung ohne Zwischentabelle zur Namensgebung auf der physikalisch objektorientierten Ebene

4.8.7.3 Existierende Namen

Sollte der Namenbildungsprozess einen Namen generieren, welcher auf der entsprechenden Ebene im Modell bereits existiert, würde dies spätestens beim Verwenden des allfälligen Quellcodes zu Problemen führen. Der Name muss deshalb programmatisch mit einer eindeutigen Nummer versehen werden.

4.8.8 Eindeutigkeit der physikalischen Elemente

Der Modellierungs- und Generierungsprozess im *Kapitel 4.3* hat aufgezeigt, dass die physikalischen Modelle bei einer Umsetzung vom logischen jeweils komplett neu erstellt werden. Anschliessend werden die zuvor gemachten Änderungen auf den physikalischen Ebenen wieder restauriert. Wie die Definition der Umsetzung ergeben hat, gibt es zu gewissen Konstrukten Benutzerinteraktionen. Auch diese werden zwischengespeichert.
Da jeweils die physikalischen Modelle verworfen werden, muss ein Mechanismus existieren, der besagt, wie Zusatzdaten (Benutzeraktionen oder sonstige Änderungen auf den physikalischen Modellen) abgespeichert werden und wie sie im Nachhinein den neu generierten physikalischen Elementen wieder zugeordnet werden können.
Aus diesem Grund werden wir dem logischen Schema und jedem logischen Element (Entitätsmenge, Attribut und Beziehung) eine eindeutige Id zuordnen. Wir bezeichnen diese im Folgenden als „**logische Id**". Für die Generierung einer globalen eindeutigen Id gibt es zahlreiche Varianten. Die wohl einfachste ist die Verwendung einer GUID[9]. Da das logische Schema beständig ist, wird diese Id bis zum Lebensende eines logischen Elementes nicht mehr verändert.
Jedes physikalische Schema und noch viel wichtiger, jedes physikalische Element, wird auch mit einer eindeutigen Id markiert. Wir nennen diese die „**physikalische Id**". Diese Id wird nun – im Gegensatz zur logischen Id – nicht immer neu kreiert. Sie wird viel mehr aus den Ids der logischen Elemente abgeleitet, die für das jeweilige physikalische Element verantwortlich sind. Somit ist gewährleistet, dass die physikalischen Elemente immer dieselbe physikalische Id erhalten, auch wenn sie neu erstellt werden. Müssen mehrere logische Ids zu einer physikalischen Id zusammengesetzt werden, so trennen wir diese mit einem Platzhalter, welcher in der Id selber nicht vorkommt. Dies gewährtleistet das Auffinden einer einzelnen logischen Id in der physikalischen Id.

Die Benutzerinteraktionen werden jeweils für ein logisches Konstrukt abgefragt und dementsprechend wird auch die Antwort in einem separaten Datenbehälter zur *logischen Id* abgelegt. Über diese kann die Antwort immer wieder eindeutig gesucht und gefunden werden. Wird ein logisches Element gelöscht, so müssen ebenfalls alle Benutzerinteraktionen gelöscht werden, die unter derselben logischen Id gespeichert sind.

Änderungen zu einem physikalischen Element werden immer in einem separaten Datenbehälter unter der betreffenden *physikalischen Id* abgelegt. Sobald ein logisches Element gelöscht wird, müssen zusätzlich alle Änderungen der physikalischen Elemente, die aufgrund des zu löschenden logischen Elementes bestehen, eliminiert werden. Da die physikalische Id immer aus einer oder mehreren logischen Ids zusammengesetzt wurde und die logischen Ids eindeutig sind, können nun alle Einträge im Datenbehälter gelöscht werden, die die logische Id enthalten.

Nachfolgend wird die Zusammensetzung der physikalischen Ids pro Schema beschrieben.
Für die Ausführungen gilt wieder:
❶ = erste Entitätsmenge
❷ = zweite Entitätsmenge

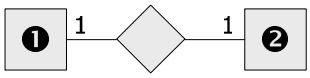


Abbildung 42 – Beziehung mit Definition der Begriffe

[9] Globally Unique Identifier (GUID): Globaler eindeutiger Bezeichner

4.8.8.1 Relationale Ebene

Die physikalischen Ids werden nach folgenden Regeln für die relationalen Elemente generiert. Wichtig ist, dass sie eindeutig über das ganze physikalische Schema sind.

Logisches Konstrukt	Physikalisches Element	Zusammensetzung der physikalischen Id
Eine Entitätsmenge		
Entitätsmenge	Tabelle	Logische Id der Entitätsmenge
Attribut	Feld	Logische Id des Attributs
Schlüsselattribut	Feld (markiert als Primärschlüssel)	Logische Id des Attributs
	Technisches Primärschlüssel-Feld	Logische Id der Entitätsmenge + „_Id"
Beziehungen ohne Zwischentabelle (alle Varianten)		
Beziehung	Beziehung	Logische Id der Beziehung
Attribut	Feld	Logische Id des Attributs
	Fremdschlüssel-Feld	Id des Primärschlüssel-Feldes + „_" + Id der Tabelle, die den Primärschlüssel definiert + „_" + Id der Tabelle, die das Fremdschlüssel-Feld hält + „_" + Logische Id der Beziehung
	Unique-Index	Logische Id der Beziehung + „_" + Id der Tabelle, die das Fremdschlüssel-Feld hält
Beziehungen ohne Zwischentabelle (alle Varianten)		
Beziehung	Zwischentabelle	Logische Id der Beziehung
Attribut	Feld	Id des logischen Attributs
	Technisches Primärschlüssel-Feld	Id der Zwischentabelle + „_Id"
Schlüsselattribut	Feld (markiert als Primärschlüssel)	Logische Id des Attributs
	Beziehung der ersten Entitätsmenge zur Zwischentabelle	Id der logischen ersten Entitätsmenge + „_" + Id der Zwischentabelle + „_" + Id der logischen Beziehung

Logisches Konstrukt	Physikalisches Element	Zusammensetzung der physikalischen Id
	Beziehung der zweiten Entitätsmenge zur Zwischentabelle	Id der Zwischentabelle + „_“ + Id der logischen zweiten Entitätsmenge + „_“ + Id der logischen Beziehung
	Unique-Index für die Fremdschlüssel-Felder, die aufgrund der ersten Entitätsmenge entstehen	Id der Zwischentabelle + „_“ + „Idx“ + „_“ + „1“
	Unique-Index für die Fremdschlüssel-Felder, die aufgrund der zweiten Entitätsmenge entstehen	Id der Zwischentabelle + „_“ + „Idx“ + „_“ + „2“
	Unique-Index für alle Fremdschlüssel-Felder	Id der Zwischentabelle + „_“ + „Idx“
	Fremdschlüssel-Feld aufgrund der ersten Entitätsmenge	Id des Primärschlüssel-Feldes + „_“ + Id der Tabelle, die das Fremdschlüssel-Feld hält + „_“ + Id der Tabelle, die den Primärschlüssel definiert + „_“ + Logische Id der Beziehung
	Fremdschlüssel-Feld aufgrund der zweiten Entitätsmenge	Id des Primärschlüssel-Feldes + „_“ + Id der Tabelle, die den Primärschlüssel definiert + „_“ + Id der Tabelle, die das Fremdschlüssel-Feld hält + „_“ + Logische Id der Beziehung

Tabelle 56 – Physikalische Ids auf der relationalen Ebene

4.8.8.2 Objektorientierte Ebene

Die physikalischen Ids werden nach folgenden Regeln für die objektorientierten Elemente generiert. Auch hier ist ganz wichtig, dass sie über das ganze physikalische Schema eindeutig sind.

Logisches Konstrukt	Physikalisches Element	Zusammensetzung der physikalischen Id
Eine Entitätsmenge		
Entitätsmenge	Klasse	Logische Id der Entitätsmenge
Attribut	Property	Logische Id des Attributs
Schlüsselattribut	Property (markiert als Key)	Logische Id des Attributs
Beziehungen ohne Zwischenklasse		
Beziehung	Association	Logische Id der Beziehung
Attribut	Property	Logische Id des Attributs
Beziehungen ohne Zwischentabelle (alle Varianten)		
Beziehung	Zwischenklasse	Logische Id der Beziehung
Attribut	Property	Logische Id des Attributs
Schlüsselattribut	Property (markiert als Primärschlüssel)	Logische Id des Attributs
	Beziehung der ersten Entitätsmenge zur Zwischenklasse	Id der logischen ersten Entitätsmenge + „_“ + Id der Zwischenklasse + „_“ + Id der logischen Beziehung
	Beziehung der zweiten Entitätsmenge zur Zwischenklasse	Id der Zwischenklasse + „_“ + Id der logischen zweiten Entitätsmenge + „_“ + Id der logischen Beziehung

Tabelle 57 – Physikalische Ids auf der objektorientierten Ebene

4.8.9 Ablauf der Umsetzung

Von der logischen Ebene muss ein Generationsprozess für die jeweilige physikalische Ebene durchgeführt werden.

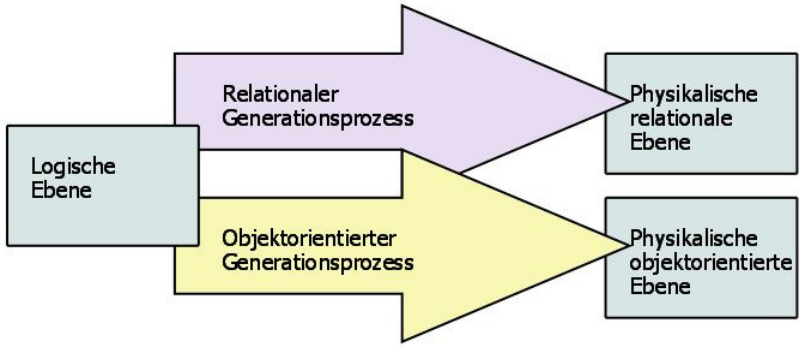


Abbildung 43 – Ablauf der Umsetzung von logisch zu physikalisch

4.8.9.1 Relationaler Generationsprozess

Der Generationsprozess für die physikalisch relationale Ebene wird weiter in folgende Unterprozesse aufgeteilt:

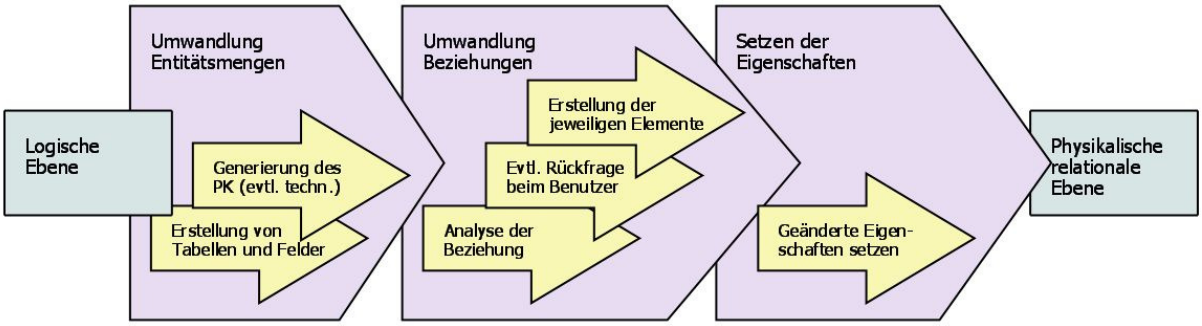


Abbildung 44 – Relationaler Generationsprozess

Zuerst werden alle logischen Entitätsmengen in relationale Tabellen umgesetzt. Aus den Schlüsselattributen werden Primärschlüssel generiert. Tabellen ohne Primärschlüssel werden automatisch mit einem technischen Schlüssel ergänzt (Feld namens „Id"). Bei logischen is-a-Beziehungen wird jeweils der Schlüssel des Supertypen gesucht und in den Subtypen übernommen. Die Attribute einer Entitätsmenge werden in Felder umgewandelt.
Danach folgt sukzessive die Umsetzung der einzelnen Beziehungen. Jede Beziehung wird durch das Programm nach den Ausführungen im *Kapitel 4.8.5 Umsetzung der logischen Ebene in die physikalischen Ebenen* analysiert und stellt dem oder der Studierenden die notwendigen Fragen. Sobald die Umsetzung klar ist, werden die benötigten physikalisch relationalen Elemente erstellt (Zwischentabelle, Beziehungen, Fremdschlüsselfelder, Primärschlüsselfelder und Indexe). Jedes Element erhält eine berechnete physikalische Id, wie es im *Kapitel 4.8.8 Eindeutigkeit der physikalischen Elemente* beschrieben ist.
Schlussendlich werden die gespeicherten physikalischen Änderungen auf den neuen Elementen restauriert.

4.8.9.2 Objektorientierter Generationsprozess

Der Generationsprozess für die physikalisch objektorientierte Ebene wird weiter in folgende Unterprozesse aufgeteilt:

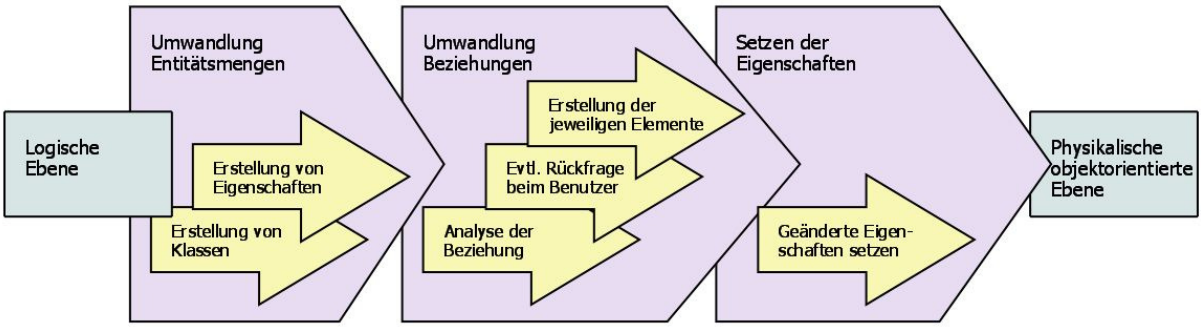


Abbildung 45 – Objektorientierter Generationsprozess

Auch der objektorientierte Generationsprozess wandelt zuerst die Entitätsmengen um. Pro Entitätsmenge wird eine Klasse erstellt. Die Attribute werden in Properties überführt. Danach folgt sukzessive die Umsetzung der einzelnen Beziehungen. Jede Beziehung wird durch das Programm nach den Ausführungen im *Kapitel 4.8.5 Umsetzung der logischen Ebene in die physikalischen Ebenen* analysiert und stellt dem oder der Studierenden die notwendigen Fragen. Sobald die Umsetzung klar ist, werden die benötigten physikalisch objektorientierten Elemente erstellt (Zwischenklasse, Properties und Assoziationen). Wiederum erhält jedes physikalische Element die berechnete Id. Mit Hilfe dieser Id werden die gespeicherten physikalischen Änderungen restauriert.

4.8.10 Definition der Datentypen

Die Software muss schlussendlich konkreten Code zur Erstellung einer Datenbank erzeugen können. Attribute auf einer Datenbank haben immer einen definierten Datentypen. Dies ist notwendig, um die gespeicherten Daten zu identifizieren und zu interpretieren. Die meisten Datenbanken haben standardisierte und eigene datenbankspezifische Datentypen. Die standardisierten Typen sind oft mit anderen Datenbanken kompatibel, obwohl es auch erschwerend dazu kommen kann, dass sie eine andere interne Repräsentation haben (z.B. Grösse des Wertebereiches). Ein Integer eines Oracle-Systems muss nicht die gleichen Wertebereiche haben, wie ein Integer des Microsoft SQL Servers.

Datentypen sind in unserer Software somit auch unablässig. Auf der logischen Ebene ist ein Datentyp jedoch noch nicht notwendig, weil die eigentliche Datenrepräsentation des zu führenden Attributes auf dieser abstrakten Ebene noch unwichtig ist.
Auf den physikalischen Ebenen hingegen müssen spätestens Datentypen zugewiesen werden. Weil jedoch auch zu diesem Zeitpunkt noch nicht bekannt ist, für welches Zielsystem bzw. welchen Dialekt der Code schlussendlich generiert werden muss, ist dieser Typ ein allgemeiner physikalischer Datentyp.

Wir sehen die folgenden physikalischen Datentypen vor:

Datentyp	Beschreibung
String	Eine Zeichenkette beliebiger Länge.
Number	Ein ganzzahliger Wert.
Float	Eine Gleitkommazahl.
Boolean	Ein Wert, der den Zustand *wahr* oder *falsch* annehmen kann.
Datetime	Ein Datum mit einer Uhrzeit.
Binary	Ein beliebiger binärer Strom.
Unknown	Ein unbekannter Datentyp. Dieser Typ wird als Standardwert während der Umsetzung verwendet.

Tabelle 58 – Physikalische Datentypen

Jeder Generator führt ein Standardmapping für die abstrakten Basis-Datentypen. Erst bei der Generierung von Quellcode für das jeweilige Zielsystem wird der entsprechend abstrakt zugewiesene Datentyp durch den effektiven ersetzt. Ein Generator kann ein Eingabeformular für ein Datentypmapping zur Verfügung stellen.

Aufgrund dieser abstrakten Datentypen können benutzerspezifische Typen erfasst werden. Neu erfasste, geänderte oder gelöschte Datentypen werden den Generatoren mitgeteilt. Solche benutzerspezifische Datentypen gehören immer zu dem Modell, das gerade in Bearbeitung ist. Möchte man sich eine Reihe von selbstdefinierten Datentypen für jedes Modell anlegen, so empfehlen wir die Speicherung einer Art Vorlagedatei, auf der jedes neue Schema aufgebaut wird.

4.8.11 Umsetzung der Generatoren

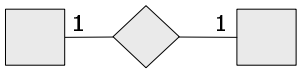
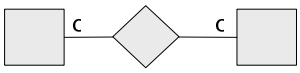
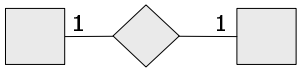
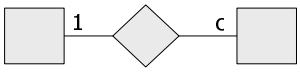
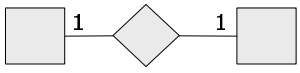
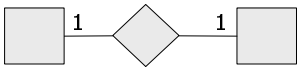
Auf den physikalischen Ebenen können Generatoren benutzt werden, um aufgrund des physikalischen Modells Quellcode für ein Zielsystem oder einen Datenbankdialekt generieren zu lassen. Bei den Generatoren handelt es sich um installierte PlugIns, die aufgrund der Schnittstellendefinition [TechnicalInterfaceDescriptionGenerator.pdf] implementiert sind. Ein Generator wird genau für eine physikalische Ebene und für ein Zielsystem oder für einen Datenbankdialekt geschrieben. Er definiert seine Datentypen und auch ein Standardmapping für die physikalischen Datentypen, die im *Kapitel 4.8.10 Definition der Datentypen* beschrieben sind. Zusätzlich kann ein Generator für die einzelnen physikalischen Elemente weitere Eigenschaften definieren, die auf der physikalischen Ebene eingestellt werden können. Sie werden auch dort abgespeichert.
Schlussendlich kann der Generator noch eigene Eigenschaften für seinen Generationsprozess angeben, z.B. eine Option, ob die Fremdschlüssel in SQL mit einem ALTER-Statement oder direkt im CREATE-Statement erstellt werden sollen.
Ein Generator erhält das komplette physikalische Schema. Zusammen mit seinen eigenen gespeicherten Daten kann er daraus seinen Quellcode generieren. Es ist dem PlugIn-Entwickler selber überlassen, mit welchen Techniken oder Methoden er diesen Quellcode erzeugt.

4.8.12 Umsetzung der Notationen

Unsere Analyse im *Kapitel 4.7.2 Analyse der Notationen* hat ergeben, dass wir für jede Ebene die jeweils passende Notation umsetzen:

- **Logische Ebene**
  → Chen-Notation und/oder die modifizierte Chen-Notation
- **Physikalisch relationale Ebene**
  → Krähenfuss-Notation
- **Physikalisch objektorientierte Ebene**
  → ODLG-Notation

Daraus folgt mit Einbezug der Ausführungen im Kapitel *3.4 Notationen für Datenbankmodelle* folgende Umsetzung der Elemente für die logische Ebene:

Element	Chen (logisch)	Modifizierter Chen (logisch)
Entitäts-menge		
Attribut		
Schlüssel-Attribut		
1:1-Beziehung		
(0 oder 1) zu (1 oder 0)		
1 zu (0 oder 1)		
1 zu 1		

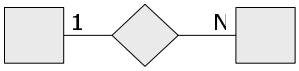
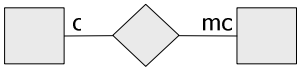
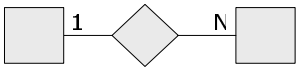
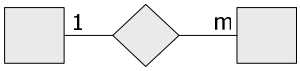
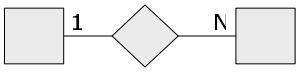
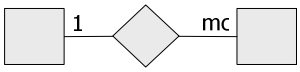
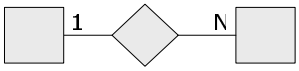
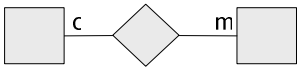
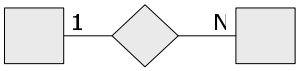
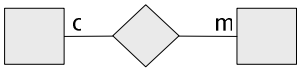
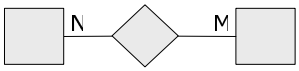
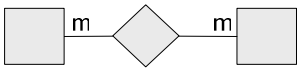
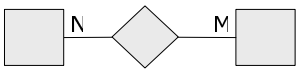
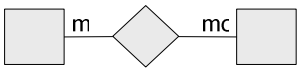
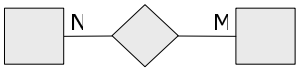
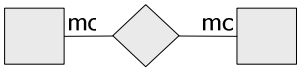
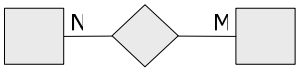
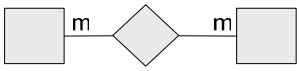
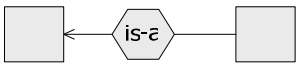
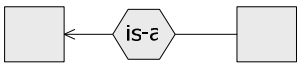
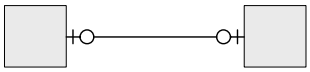
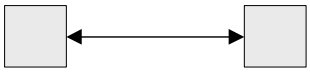
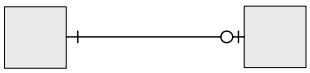
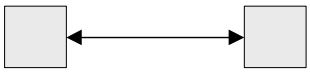
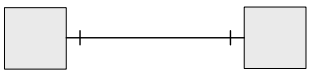
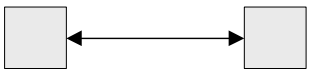
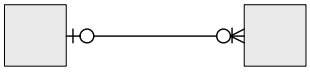
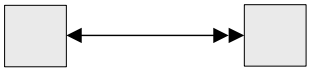
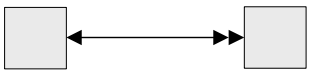
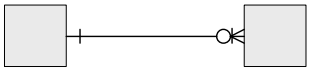
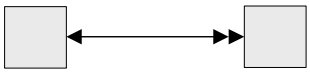
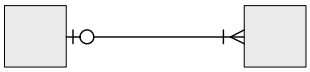
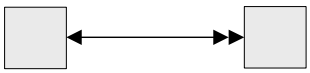
Element	Chen (logisch)	Modifizierter Chen (logisch)
1:n-Beziehung		
(0 oder 1) zu (beliebig vielen)		
1 zu (mindestens 1)		
1 zu (beliebig vielen)		
(0 oder 1) zu (mindestens 1)		
(0 oder 1) zu (exakt n)		
n:m-Beziehung		
(mindestens 1) zu (mindestens 1)		
(mindestens 1) zu (beliebig vielen)		
(beliebig viele) zu (beliebig vielen)		
(exakt n) zu (exakt n)		
is-a-Beziehung		
is-a-Beziehung		

Tabelle 59 – Visuelle Darstellung der Modellierungselemente auf logischer Ebene

Die logischen Modellierungselemente werden wie folgt auf den physikalischen Ebenen dargestellt. Es werden hier sämtliche theoretischen Varianten aufgeführt, obwohl nicht alle möglich sind.

Element	Krähenfuss (physikalisch relational)	ODLG (physikalisch objektorientiert)
Entitätsmenge mit Attributen (bzw. Tabellen mit Feldern und Klassen mit Eigenschaften)	Tabelle PK_Feld_1 PK_Feld_2 … Feld_1 Feld_2 Feld_3 … Primärschlüssel-Felder werden vor den anderen Feldern genannt. Eine Trennung zwischen Tabellennamen und Feldern wird nicht dargestellt.	Klasse Eigenschaft_1 Eigenschaft_2 Eigenschaft_3 …
1:1-Beziehung		
(0 oder 1) zu (1 oder 0)		
1 zu (0 oder 1)		
1 zu 1		
1:n-Beziehung		
(0 oder 1) zu (beliebig vielen)		
1 zu (mindestens 1)		
1 zu (beliebig vielen)		
(0 oder 1) zu (mindestens 1)		

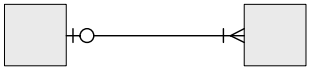
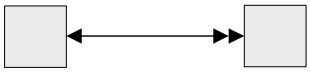
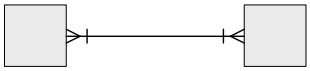
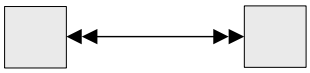
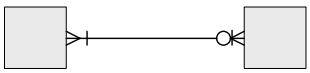
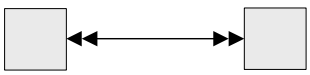
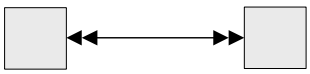
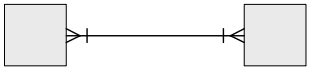
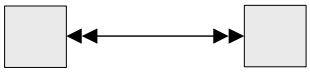
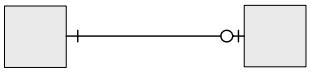
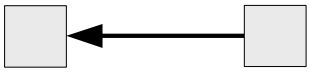
Element	Krähenfuss (physikalisch relational)	ODLG (physikalisch objektorientiert)
(0 oder 1) zu (exakt n)		
n:m-Beziehung		
(mindestens 1) zu (mindestens 1)		
(mindestens 1) zu (beliebig vielen)		
(beliebig viele) zu (beliebig vielen)		
(exakt n) zu (exakt n)		
is-a-Bezieung		
is-a-Beziehung		

Tabelle 60 – Visuelle Darstellung der Modellierungselemente auf den physikalischen Ebenen

5 Implementation des Kerns

In diesem Kapitel werden die Grundkonzepte der Systemimplementation beschrieben. Die Umsetzungen der konkrekten PlugIns sind in den Folgekapiteln aufgeführt.

5.1 Systemarchitektur

5.1.1 Architekturziele

Das Programm soll auf einer stabilen Architektur aufgebaut werden, die die Aufgaben klar trennt, leicht erweiterbar und wieder verwendbar ist. Werden die Aufgaben nämlich nicht sauber getrennt, so ist die GUI-Ebene üblicherweise das Auffangbecken für Logik, die eigentlich in andere Anwendungsebenen gehört. Zusätzlich besteht die Gefahr, eine Unmenge von dupliziertem Code zu haben.
Zu einer Architektur nach dem State of the Art gehört auch dazu, dass die Daten von ihrer Repräsentation getrennt werden. So kann zu einem späteren Zeitpunkt die View ausgetauscht werden, ohne die Businesslogik anpassen zu müssen.

Die obigen Architekturziele sind bekannt und in den meisten Fällen auch allgemein gültig. Wir haben zusätzlich noch folgende, etwas spezifischere Architekturziele oder Designvorschriften definiert:

- Externe Komponente oder PlugIns sollen möglichst keinen Zugriff auf die modellierten Daten haben. Der Zugriff auf den internen Datencontainer, welcher die logischen und physikalischen Modelle speichert, wird architektonisch verhindert. Dafür verwenden wir bei den entsprechenden Klassen den *internal*-Modifier, damit sie nur innerhalb des Assemblies[10] angesprochen werden können. An Stellen, bei denen Instanzen einzelner Elemente des logischen Modells oder der physikalischen Modelle an ein PlugIn weitergereicht werden, übergeben wir nach Möglichkeit nur eine Kopie (ein Klon). Falls ein PlugIn versehentlich oder auch mutwillig Daten auf einem Schema ändert, hat dies den Vorteil, dass die Änderung nur das geklonte Objekt beeinflussen und nicht das echte.

- Wir setzen ein Exception-Handling ein und werten keine Fehlercodes aus.

- Wir setzten das Exception-Handling eher sparsam ein. Das bedeutet, dass wir uns nach dem Grundsatz "so wenig wie möglich; so viel wie nötig" richten. Um den Main-Thread herum wird ein Exception-Handling gelegt, um allfällige, nicht abgefangene Fehler sauber handhaben zu können. Alle Event-Methoden, die durch eine direkte Benutzerinteraktion ausgelöst wurden, sind mit einem try-catch-Konstrukt ausgestattet. Die Ausnahme bilden hier die Paint-Methoden. Diese sind allgemein sehr sensibel und müssen fehlerfrei funktionieren. Sollte darin eine Exception auftreten, würde der Bildschirm nach der abgefangenen Exception wieder neu gezeichnet und eine erneute Exception würde ausgelöst werden, was in einem Endlosloop enden könnte. Diesen Zustand könnte man stabilisieren, indem man die Exceptions in den Paint-Methoden jeweils einfach ignoriert. Dies wird von uns jedoch als schlecht erachtet, denn daraus könnten Fehldarstellungen oder gar fehlerhafte Daten resultieren. Wir haben uns entschlossen, Exceptions in diesen Bereichen mit einem kontrollierten Programmabbruch zu handhaben. Somit weiss der Benutzer

[10] .NET spezifischer Ausdruck für eine kompilierte Komponente. Dies kann eine ausführbare Datei oder eine Bibliothek sein.

oder der PlugIn-Entwickler bzw. die PlugIn-Entwicklerin, dass es ein Problem gegeben hat und kann es beheben.
Trotzdem gibt es in unserem Programm Exceptions, die abgefangen, jedoch nicht behandelt werden. Ein solches Konstrukt wird vor allem beim Laden und Speichern des Datenmodells gebraucht. Wir verhindern damit, dass während der Speicherung der jeweiligen PlugIn-Daten über eine Exception das Speicherformat zerstört oder beschädigt wird. Mit unserem Konstrukt würden in so einem Fehlerfall ganz einfach die Teile des betreffenden PlugIns nicht gespeichert. Dies bedeutet jedoch nicht, dass unser Speicherformat so flexibel gestaltet ist, dass es mit einer fehlerhaften Struktur zurechtkommen kann. Wir versuchen lediglich, auch bei Fehlern in den PlugIns, ein möglichst korrektes XML zu erzeugen.

- Alle Exceptions, die abgefangen und nicht behandelt resp. ignoriert werden, müssen über einen Trace[11] an alle interessierten Listeners gemeldet werden, damit diese zu einem späteren Zeitpunkt auf einfache Art und Weise visualisiert werden können. Dies könnte z.B. über ein Debug-Fenster geschehen.

- Wir implementieren eine saubere Businesslogik, und zwar dort, wo sie anfällt. Die Trennung der Daten von ihrer Repräsentation verlangt, dass das Businessobjekt über eine Änderung informiert. Dies passiert mit dem Feuern eines Changed-Events direkt beim Setzen einer Information. Ausserdem kann das Ändern eines Datenmembers weitere Logik und Abhängigkeiten nach sich ziehen. Solche Sachen müssen auch beim Setzen der Information ausgeführt werden. In .NET werden oft generische Listen als Properties[12] (state of the art) öffentlich zugänglich gemacht. Über ein solches Property kann relativ einfach ein neues Element der Liste hinzugefügt oder gelöscht werden. Jedoch hat ein solches Konstrukt den gewaltigen Nachteil, dass bei einem *Add()* oder *Remove()* nicht mehr eingegriffen werden kann. Die dazugehörende Businesslogik muss separat ausgeführt werden, was zu einer Fehlerquelle werden könnte. Aus diesem Grund haben wir uns entschlossen, solche Listen nicht als Properties einzusetzen. Wir führen die Liste zwar intern, gegen „Aussen" stellen wir die *Add()* und *Remove()*-Methoden selber zur Verfügung. Ausserdem schreiben wir für jede interne Liste eine *Get()*-Methode, die einen Enumerator[13] zurückliefert, mit dem man durch die Liste iterieren kann.

- Literale werden weitgehend durch Konstanten ersetzt. Konstanten haben den Vorteil, dass sie kompilierbar sind und man sich somit nur an einem Ort verschreiben kann. Eine Ausnahme stellt der Programmcode zur XML-Serialisierung dar. Dort haben wir uns zugunsten der Lesbarkeit für String-Literale entschieden.

- Datenmember werden versteckt und über Properties (analog zu *Get()* und *Set()*-Methode in Java) zugänglich gemacht.

- Wir überdecken keine Methoden, d.h. wir durchbrechen den Polymorphismus nicht. .NET würde es zulassen, eine Methode mit dem Schlüsselwort *new* zu deklarieren. So wird die Methode überdeckt und neu definiert und der Polymorphismus wird

[11] Eine der Möglichkeiten zur Fehlerprotokollierung in .NET.
[12] .NET-Property (Eigenschaft): Eigenschaften können z.B. analog zu Java eine Get- und/oder Set-Methode für einen privaten Datenmember definieren. Sie werden häufig auch für intelligente sowie berechnete Felder eingesetzt.
[13] Der Enumerator ist von .NET-übliches Konstrukt und ist nach dem Design-Pattern Iterator programmiert.

unterbrochen. Wir erachten dieses Konstrukt als nicht optimal, da sich schwer überschaubare Nebeneffekte bilden können. Ausserdem ist das Konstrukt wegen der Komplexität schlecht wartbar.

5.1.2 Schichtenmodell

Die Grafik zeigt die Aufteilung der Architektur in seine Schichten. Die MS .NET-Plattform lässt eine saubere Schichtentrennung zu.

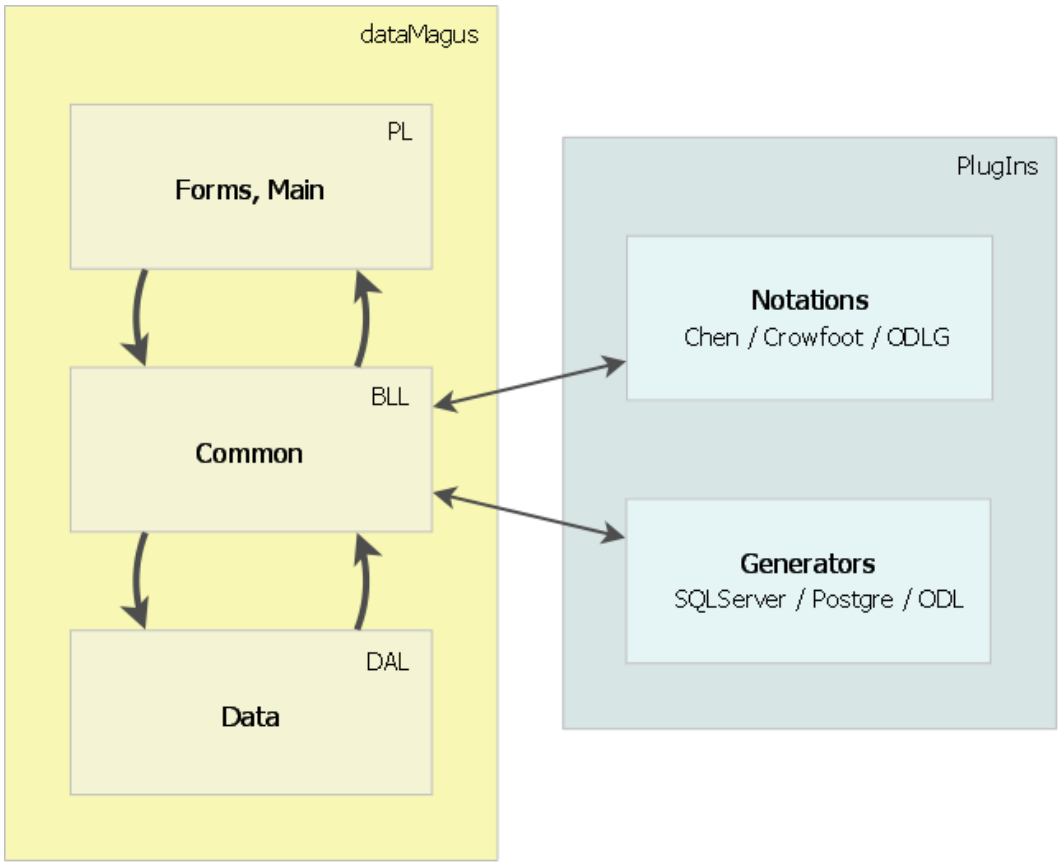


Abbildung 46 – Schichtenmodell

PL: Presentation Layer (Präsentations-Schicht)
BLL: Business Logic Layer (Geschäftslogik-Schicht)
DAL: Data Access Layer (Datenzugriffs-Schicht)

Datenzugriffs-Schicht
Diese Schicht regelt den Zugriff auf persistent gespeicherte Daten.

Geschäftslogik-Schicht
Diese Schicht implementiert die Geschäftslogik des Programms. Hier befinden sich die Businessobjekte mit ihren Daten (Model) und ihrer Logik. Bei jeder Zustandsänderung wirft das Model einen Changed-Event.

Präsentations-Schicht
Diese Schicht beinhaltet die Elemente zur visuellen Darstellung der Daten (View) und die dafür benötigte Steuerungslogik (Controller). Die einzelnen Elemente können sich auf die Changed-Events der Businessobjekte registrieren und bei einer Änderung ihre Darstellung aktualisieren. Dieses Konstrukt erlaubt die saubere Trennung der Daten von ihrer Repräsentation. Ausserdem kann so ein Businessobjekt auf unterschiedliche Weise dargestellt werden.

PlugIns
Diese Schicht steht allen zur Verfügung, die PlugIns für Notationen und Generatoren erstellen. Ein PlugIn baut auf den Schnittstellen und Businessobjekten der Geschäftslogik-Schicht auf. Die kompilierte DLL-Datei muss sich schlussendlich im selben Verzeichnis befinden wie die ausführbare Programmdatei der Applikation dataMagus.

5.1.3 Subsystemmodell

Die nachfolgende Grafik zeigt die Architektur des Programms aus Sicht der einzelnen Pakete und deren Abhängigkeiten. Es wurde darauf geachtet, dass keine zirkulären Referenzen vorhanden sind. Ein Paket definiert einen eindeutigen Namensraum, wobei hier aus Gründen der Übersichtlichkeit nur die Hauptpakete aufgezeigt werden. Jedes Paket baut natürlich auf dem .NET-Framework 2.0 auf.

Die einzelnen Pakete sind wie folgt eingefärbt:

weiss	Hauptapplikation
rot	Präsentations-Schicht
grün	Geschäftslogik-Schicht
gelb	Datenzugriffs-Schicht
blau	PlugIns

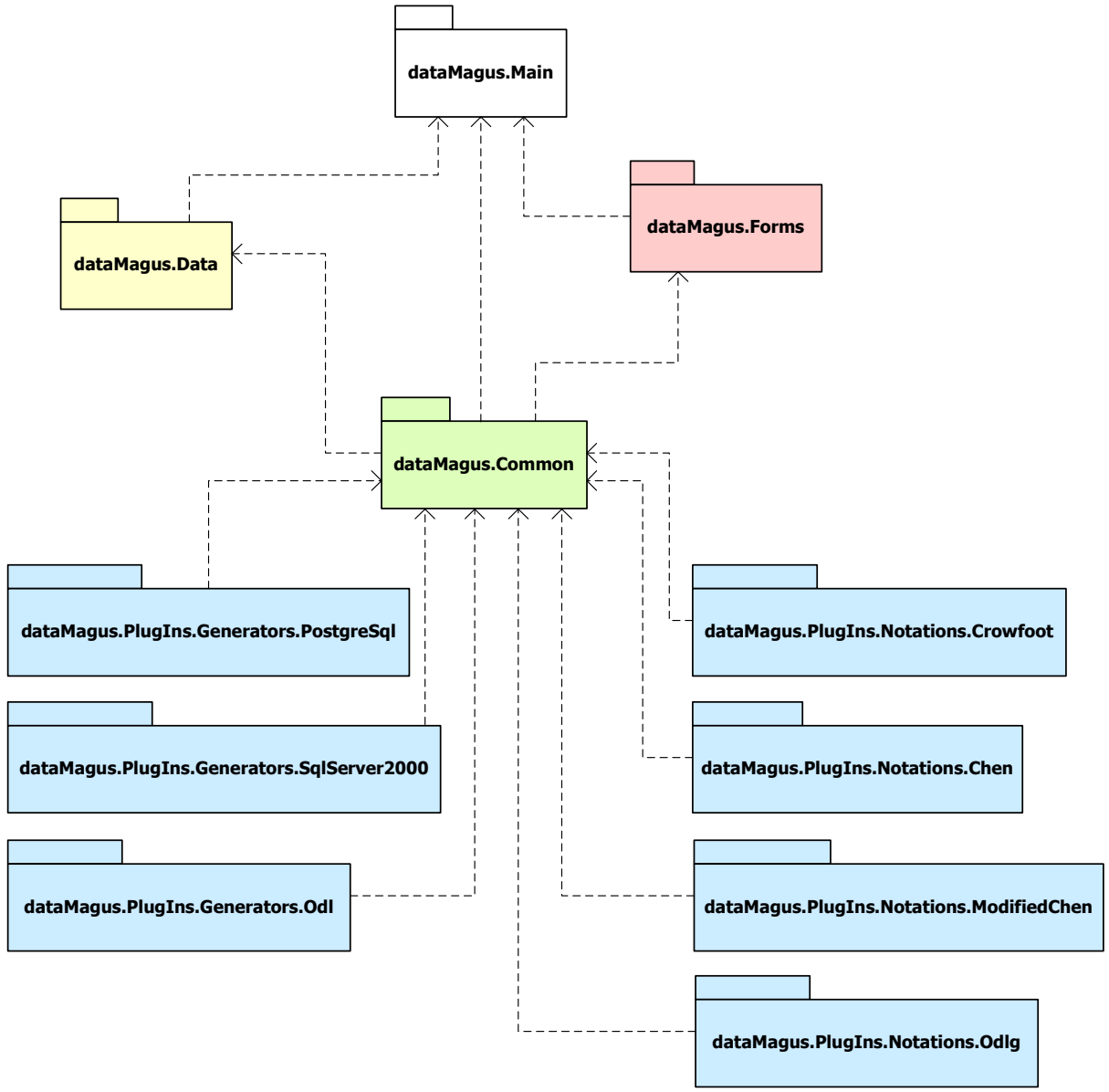


Abbildung 47 – Paketdiagramm

Die Pakete mit ihren Unterpaketen werden wie folgt eingesetzt:

Paket oder Unterpaket	Beschreibung
dataMagus.Main	
dataMagus.Main	Dieses Paket beinhaltet die Hauptapplikation mit dem Main-Thread.
dataMagus.Forms	
dataMagus.Forms	In diesem Paket sind die Elemente der Präsentations-Schicht enthalten.
dataMagus.Forms.Controls	Hier befinden sich die Hauptelemente des Programms. Es handelt sich um Controls für den Navigations-, Eigenschaften- und Modellierungsbereich.
dataMagus.Forms.Elements	In diesem Unterpaket sind alle zusätzlichen Eingabemasken vorhanden, die benötigt werden, um Änderungen auf den einzelnen

Paket oder Unterpaket	Beschreibung
	logischen oder physikalischen Elementen vorzunehmen.
dataMagus.Forms.Handlers	Die Handlers dienen zur Steuerungslogik.
dataMagus.Common	
dataMagus.Common	Das Common-Paket definiert die Geschäftslogik.
dataMagus.Common.Logical	Hier befinden sich alle Klassen für die Elemente, die auf dem logischen Schema benötigt werden.
dataMagus.Common.Physical	Dieses Unterpaket enthält Klassen, die für alle physikalischen Ebenen gemeinsam sind.
dataMagus.Common.Physical.Objectoriented	Hier befinden sich die Elemente für das physikalisch objektorientierte Schema.
dataMagus.Common.Physical.Relational	Hier befinden sich die Elemente für das physikalisch relationale Schema.
dataMagus.Common.PlugIn.Generator	Dieses Unterpaket definiert die benötigten Schnittstellen und Basisklassen für die Implementation eines spezifischen Generators.
dataMagus.Common.PlugIn.Notation	In diesem Unterpaket sind die benötigten Schnittstellen und Basisklassen für die Implementation einer spezifischen Notation vorhanden.
dataMagus.Data	
dataMagus.Data	Das Data-Paket regelt den Zugriff auf persistente Daten.
dataMagus.PlugIns.Notations.Chen	
dataMagus.PlugIns.Notations.Chen	Dieses Paket implementiert die Chen-Notation.
dataMagus.PlugIns.Notations.ModifiedChen	
dataMagus.PlugIns.Notations.ModifiedChen	Dieses Paket ermöglicht den Einsatz der modifizierten Chen-Notation.
dataMagus.PlugIns.Notations.Crowfoot	
dataMagus.PlugIns.Notations.Crowfoot	Dieses Paket stellt die Krähenfuss-Notation zur Verfügung.
dataMagus.PlugIns.Notations.Odlg	
dataMagus.PlugIns.Notations.Odlg	Dieses Paket implementiert die ODLG-Notation.
dataMagus.PlugIns.Generators.PostgreSql	
dataMagus.PlugIns.Generators.PostgreSql	Dieses Paket implementiert einen Generator für die PostgreSQL-Datenbank.
dataMagus.PlugIns.Generators.SqlServer2000	
dataMagus.PlugIns.Generators.SqlServer2000	Dieses Paket stellt den Generator für die SQL Server 200-Datenbank.
dataMagus.PlugIns.Generators.Odl	
dataMagus.PlugIns.Generators.Odl	Dieses Paket generiert Code in der ODL-Notation.

Tabelle 61 – Beschreibung der Pakete und ihren Unterpaketen

5.2 Programmarchitektur

Nachfolgend werden wichtige Aspekte der Programmarchitektur erläutert. Die Klassendiagramme wurden mit dem Class Designer der Entwicklungsumgebung erstellt. Sie entsprechen nicht dem offiziellen UML-Standard, sind jedoch auf die speziellen .NET-Objekte wie Properties oder Events abgestimmt und können diese fehlerfrei darstellen. Der Class Designer kann leider keine Aggregationen oder Kompositionen zeichnen. Dort, wo es wichtig für das Verständnis ist, haben wir manuell Linien eingefügt. Ansonsten zeigen wir die Properties mit Typenangaben. So sollte auch klar werden, wie die Klassen miteinander interagieren. Der Namespace (Namensraum) der Klasse haben wir in einer gelben Kommentarbox unmittelbar neben, unter- oder oberhalb der Klasse angegeben.
Die Klassendiagramme werden nur von den im Kontext relevanten Klassen erstellt. Sie sollen eine Hilfe sein, um die Funktionsweise und den Aufbau der Applikation besser verstehen zu können. Es werden auch nur ausgewählte Klassenmember, Properties, Methoden und Events dargestellt. Eine vollständige Dokumentation wird aus dem kommentierten Source-Code generiert und steht dann als Windowshilfe im CHM-Format zur Verfügung.
Während der folgenden Klassendiskussion werden wir auch auf die Erweiterbarkeit für Notationen, physikalische Ebenen, Modellierungselemente und Datenbankdialekte oder Zielsysteme eingehen. Eine genaue Anleitung zur Erweiterung der einzelnen Elemente findet sich in den verschiedenen technischen Beschreibungen, die wir anbieten.
Wir sind uns bewusst, dass einige Klassendiagramme eine sehr kleine Schrift aufweisen. Da wir aber gerade den Zusammenhang einzelner Klassen aufzeigen wollten, war es manchmal fast nicht möglich, eine grössere Schrift zu nehmen.
Im Weiteren setzen wir in unserer Software den Begriff „Schema" dem Begriff „Ebene" gleich.

5.2.1 Ebenen

Für die logische Ebene und die physikalischen Ebenen wird ein allgemeines Interface *ISchema* definiert. Jede Ebene zeichnet sich durch eine eindeutige Id, die vom System vergeben wird und sich an einer GUID anlehnt, und einem Namen aus. Jede Ebene kann in verschiedenen dafür vorgesehenen Notationen dargestellt werden. Die aktuell ausgewählte Notation wird auf dem Schema zwischengespeichert. Die Trennung der Daten von ihrer Repräsentation erfordert den Event *SchemaChanged*, der gefeuert wird, sobald sich die Daten eines Schemas ändern. Anzeigecontrols (Views) können auf diesen Event hören und ihre Darstellung entsprechend aktualisieren. Damit z.B. in einem Initialisierungsvorgang oder beim Umsetzen des logischen Schemas in physikalische viele Änderungen gemeinsam gemacht werden können, ohne dass bei jeder Änderung ein Event gefeuert wird, gibt es die Methode *BeginInit()*. Mit ihr werden keine Events mehr gefeuert, bis die Methode *EndInit()* aufgerufen wird. *EndInit()* feuert nur bei Bedarf einmalig den ChangedEvent. Beide Methoden funktionieren kaskardierend, d.h. sie leiten sich selber durch alle Elemente des Schemas weiter.

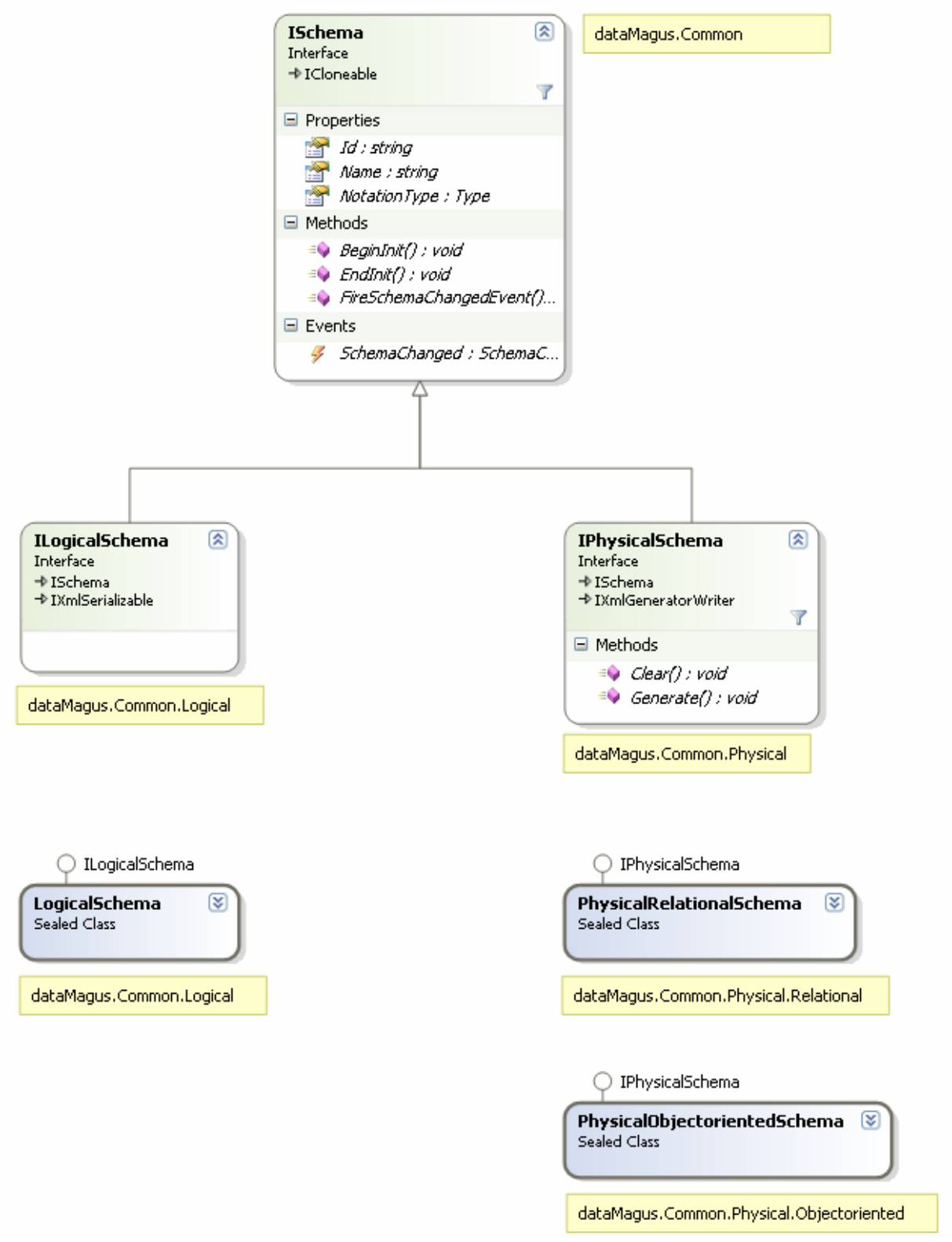


Abbildung 48 – Klassendiagramm *Ebenen*

Vom Schema kann ein Klon erstellt werden. Dies wird durch die Implementierung des Interfaces *ICloneable*[14] gewährleistet. Ein Klon wird durch die Binary-Serialisierung[15] von

[14] Interface im Namensraum System aus dem .NET-Framework
[15] Unter Serialisierung versteht man die persistente Speicherung von Objekten.

.NET erstellt. Dazu muss jedes Objekt mit dem *Serializable-Attribute*[16] von .NET ausgestattet werden. Die Durchführung einer Serialisierung zur Erstellung eines Klons hat gegenüber einer handgestrickten *Clone()*-Implementation den Vorteil, dass ganz bestimmt alle Referenzen kopiert werden. Ausserdem können schnell Probleme auftauchen, wenn man ein Element in einem Baum klonen möchte, das noch eine Referenz auf den Vaterknoten hat. Denn dann müsste zuerst der Vater geklont und dessen Referenz auf dem zu klonenden Objekt wieder gesetzt werden etc. Dies kann also sehr schnell ziemlich kompliziert werden. Die Binary-Serialisierung stellt .NET selber zur Verfügung und kann ohne Probleme so einen Baum serialisieren. Ausserdem ist sie auf Geschwindigkeit optimiert.

Im Weiteren gibt es für die logische Ebene ein Markerinterface *ILogicalSchema*, das von den Interfaces *ISchema* und *IXmlSerializable*[17] abgeleitet ist. *IXmlSerializable* wird für die persistente Speicherung der Modelle benötigt. Für eine genaue Beschreibung verweisen wir auf das *Kapitel 5.2.10 Persistente Daten.*

Für die physikalischen Ebenen wird das Interface *IPhysicalSchema* definiert, das von den Interfaces *ISchema* und *IXmlGeneratorWriter* abgeleitet ist. Das Interface *IPhysicalSchema* definiert u.A. die Methoden *Generate()* und *Clear()*, um von einem logischen Modell das physikalische zu generieren oder das gesamte physikalische Modell zu löschen. *IXmlGeneratorWriter* wird benötigt, um den Generatoren das physikalische Schema im XML-Format zur Verfügung zu stellen. Die genaue Beschreibung hierzu findet sich im *Kapitel 5.2.7 Code-Generatoren*.
Die effektiven logischen und physikalischen Ebenen müssen das entsprechende Interface *ILogicalSchema* bzw. *IPyhsicalSchema* implementieren. Die Klasse *LogicalSchema* repräsentiert das logische Modell, Klasse *PhysicalObjectorientedSchema* steht für die physikalisch objektorientierte Ebene und die Klasse *PhysicalRelationalSchema* für das physikalisch relationale Modell.
Wir haben unsere Schema-Klassen als *sealed* deklariert, d.h. von ihnen können keine weiteren Klassen mehr abgeleitet werden. Es gibt in der Praxis jeweils auch nur ein logisches Modell oder ein physikalisch relationales Modell.
Soll das Programm nun um eine physikalische Ebene (z.B. objektrelational) erweitert werden, so muss eine neue Schema-Klasse erstellt werden, die das Interface *IPhysicalSchema* implementiert.

Die Klasse *PhysicalSchemaManager* (wiederum *sealed* und als *Singleton*[18] implementiert) findet automatisch alle pyhsikalischen Ebenen, indem es nach Klassen mit dem implementierten Interface *IPhysicalSchema* sucht und sich eine Instanz der Klassen hält. Über diese Klasse können die physikalischen Schemas angesprochen werden.

[16] Attribute sind „.NET-eigene" Konstrukte. Sie werden mit eckigen Klammern vor einer Klasse, Methode, Property etc. angegeben und stellen einen Mechanismus dar, wie man Metadaten im Code ablegen kann. Diese lassen sich dann zur Laufzeit über die Reflektion auswerten.
[17] Interface im Namensraum System.Xml.Serialization aus dem .NET-Framework
[18] Es existiert höchstens eine Instanz der Klasse.

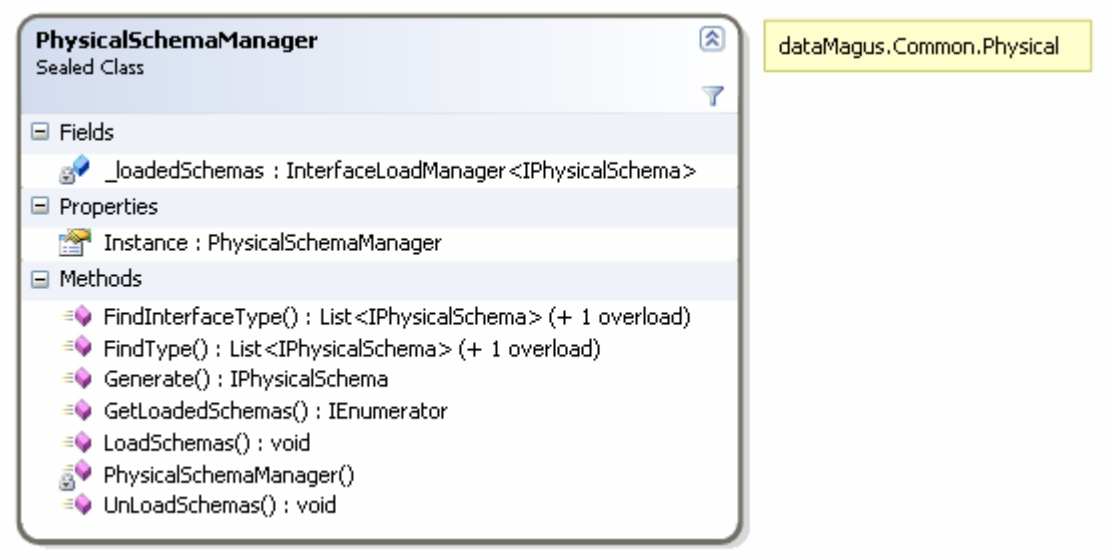


Abbildung 49 – Klasse *PhysicalSchemaManager*

Der *PhysicalSchemaManager* sucht sich die Implementationen der Interfaces mit Hilfe der generischen Klasse *InterfaceLoadManager*. Wir haben für das Laden der Interfaces eine allgemeine Lösung erstellt, weil wir dies bei den Notationen und den verschiedenen Generatoren wiederverwenden können.

5.2.2 Elemente der Ebenen

Für die logischen Modellierungselemente und die jeweiligen physikalischen Elemente ist das allgemeine Interface *IElement* definiert. Jedes Element weist sich durch eine eindeutige Id, einen Namen und eine Lokation aus. Die Id wird nach den Regeln im *Kapitel 4.8.8 Eindeutigkeit der physikalischen Elemente* erzeugt.
Werden die Daten eines Elements geändert, so wird der Event *ElementChanged* gefeuert. Des Weiteren kann jedes Element geklont werden, weshalb sich *IElement* zusätzlich von *ICloneable* ableitet.

Die Lokationsdaten beinhalten die visuelle Position des Elements auf dem Schema. Hier handelt es sich um die einzige Ausnahme, bei der wir Anzeigedaten auf dem Businessobjekt zwischenspeichern. Es sind zwar auch Daten, aber sie haben eigentlich keinen direkten Bezug zu den reinen Businessdaten. Wir sind uns bewusst, dass hier die Daten nicht ganz sauber von der Repräsentation getrennt sind. Nach Absprache mit Frau Prof. Dr. C. Class ist die strikte Trennung der Applikationsdaten vom Model in diesem Fall eher zweitrangig. Ausserdem handelt es sich um eine kleine Vereinfachung, indem die Position pro Element, und das bedeutet auch pro Ebene, zwischengespeichert ist. Konkret heisst das, dass die Elemente in allen Notationen einer Ebene am selben Ort gezeichnet werden. Dies vereinfacht die Programmierung der Notationen. Ausserdem erachten wir das durchaus als praktikabel, da die vorhandenen Elemente (Entitätsmenge, Attribute, Beziehungen) pro Ebene fix sind und eine andere Notation derselben Ebene nicht plötzlich neue Elemente definiert. Durch die Speicherung der Lokationsdaten direkt auf den Elementen ergibt sich nun den Vorteil, dass bei der Umsetzung in physikalische Elemente die Position übernommen werden kann und die physikalischen Elemente in vielen Fällen bereits schön positioniert werden können. Müssen Zwischentabellen oder –klassen generiert werden, kann die Position sogar mit den Positionen der an der Beziehung beteiligten Entitätsmengen berechnet werden. Dies wäre nicht möglich, wenn die Notationen selber für die Position zuständig wären. Denn dann wären diese Daten in einem PlugIn ausgelagert, von dem nicht ausgegangen werden darf, dass es auch installiert und eingesetzt wird.

Als Grundlage aller logischen Modellierungselemente dient die abstrakte Klasse *LogicalElement*, welche nun das Interface *IElement* implementiert. Zusätzlich wird diese Klasse auch XML-serialisierbar gemacht, da ja das gesamte logische Modell persistent gehalten wird.

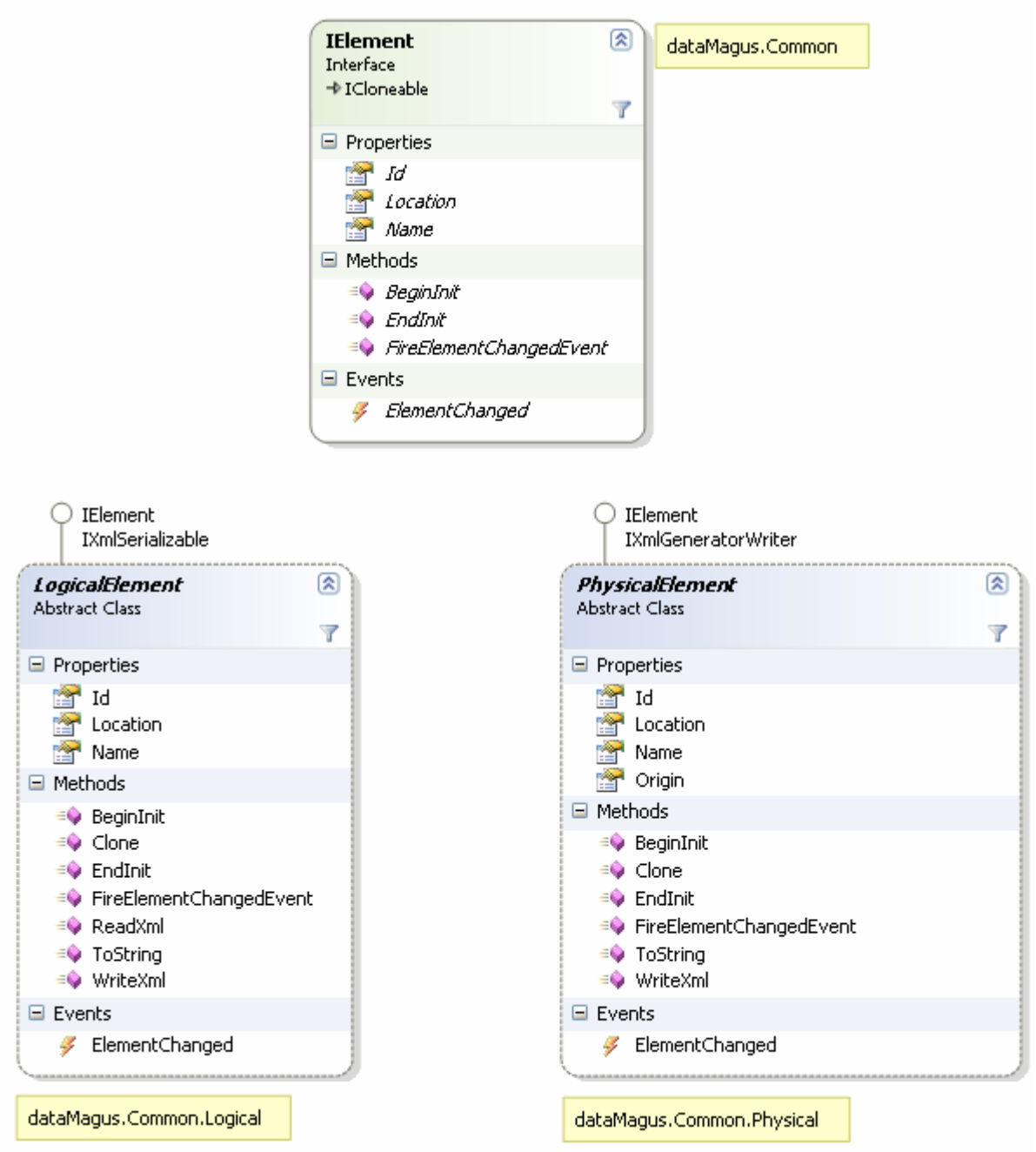


Abbildung 50 – Klassendiagramm *Elemente der Ebenen*

Für physikalische Modelle und deren Elemente steht die abstrakte Klasse *PhyiscalElement* zur Verfügung. Zu jedem physikalischen Element kann eine Referenz auf das Ursprungselement, wegen welchem es überhaupt angelegt wird, geführt werden. Zusätzlich kann auch eine Referenz auf das physikalische Schema, zu dem es gehört, abgelegt werden.
Um den Generatoren die physikalischen Elemente im XML-Format zur Verfügung zu stellen, implementiert auch *PhysicalElement* das Interface *IXmlGeneratorWriter*.

5.2.3 Logische Ebene

Alle Klassen für die logischen Modellierungselemente leiten nun von der abstrakten Klasse *LogicalElement* ab. Die Klasse *LogicalEntity* repräsentiert eine Entität, *LogicalAttribute* vertritt ein Attribut auf einer Entität. Eine Entität kann eine Liste von Attributen führen. Schlüsselattribute werden als referenzierte Instanz des Attributes in der Liste *KeyAttributes* geführt.
Für die verschiedenen Beziehungsarten wird eine abstrakte Klasse *LogicalRelationship* definiert. Jede Beziehung steht zwischen zwei Entitäten und jedes Beziehungsende hat einen Rollennamen. Die effektiven Beziehungsarten leiten von dieser abstrakten Klasse ab. Es sind dies die Klasse *IsALogicalRelationship* für eine is-a-Beziehung, die Klasse *OneToOneLogicalRelationship* für eine 1:1-Beziehung, die Klasse *OneToManyLogicalRelationship* für eine 1:n-Beziehung und die Klasse *ManyToManyLogicalRelationship* für eine n:m-Beziehung. Diese Klassen halten sich intern an die erlaubten Merkmale gemäss der Definition im *Kapitel 4.8*.

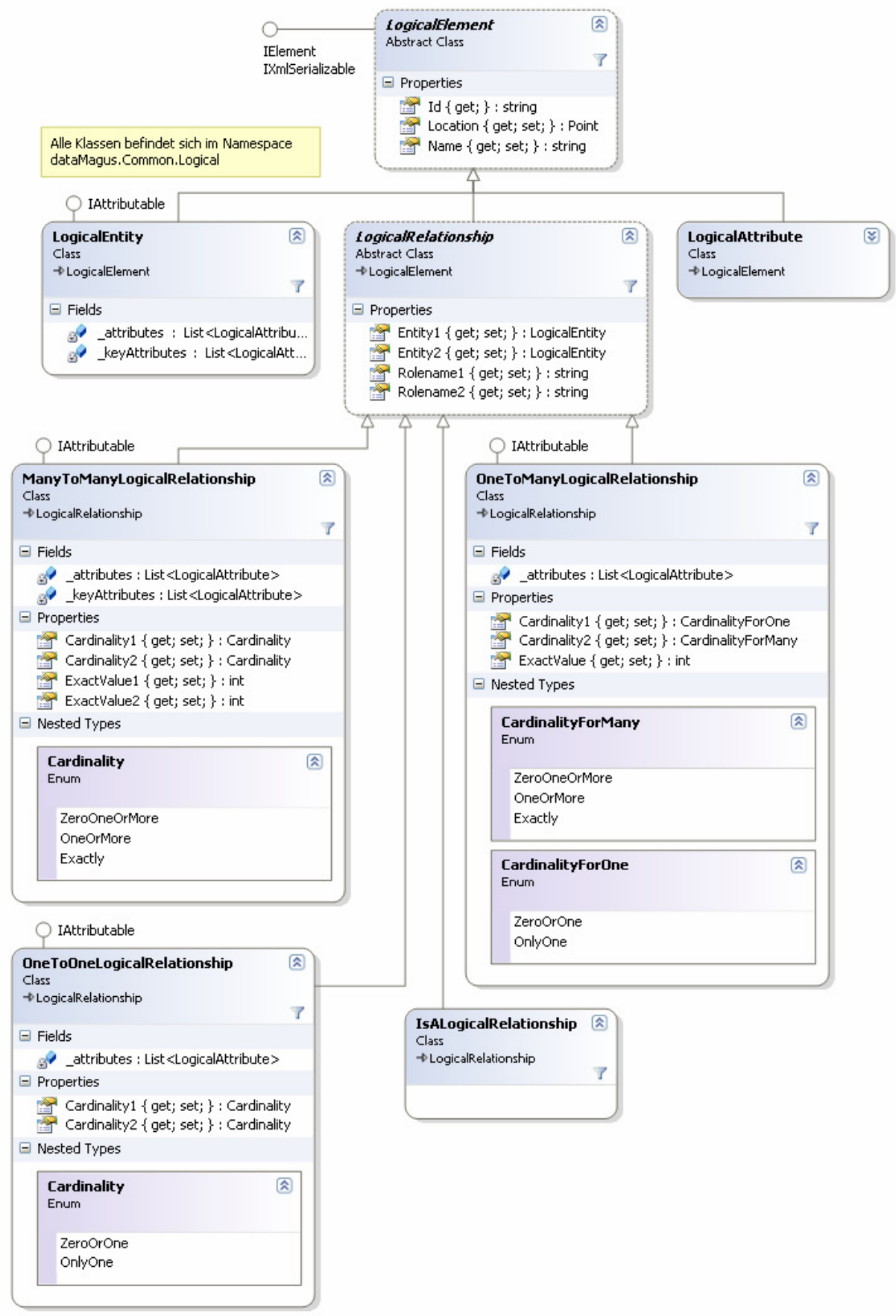


Abbildung 51 – Klassendiagramm *Logische Ebene*

Das modellierte logische Schema wird in der Klasse *LogicalSchema* gehalten. Sie führt die Entitäten in einer internen Liste mit Instanzen der Klasse *LogicalEntity* sowie die Beziehungen in einer Liste mit Instanzen der Klasse *LogicalRelationship*.

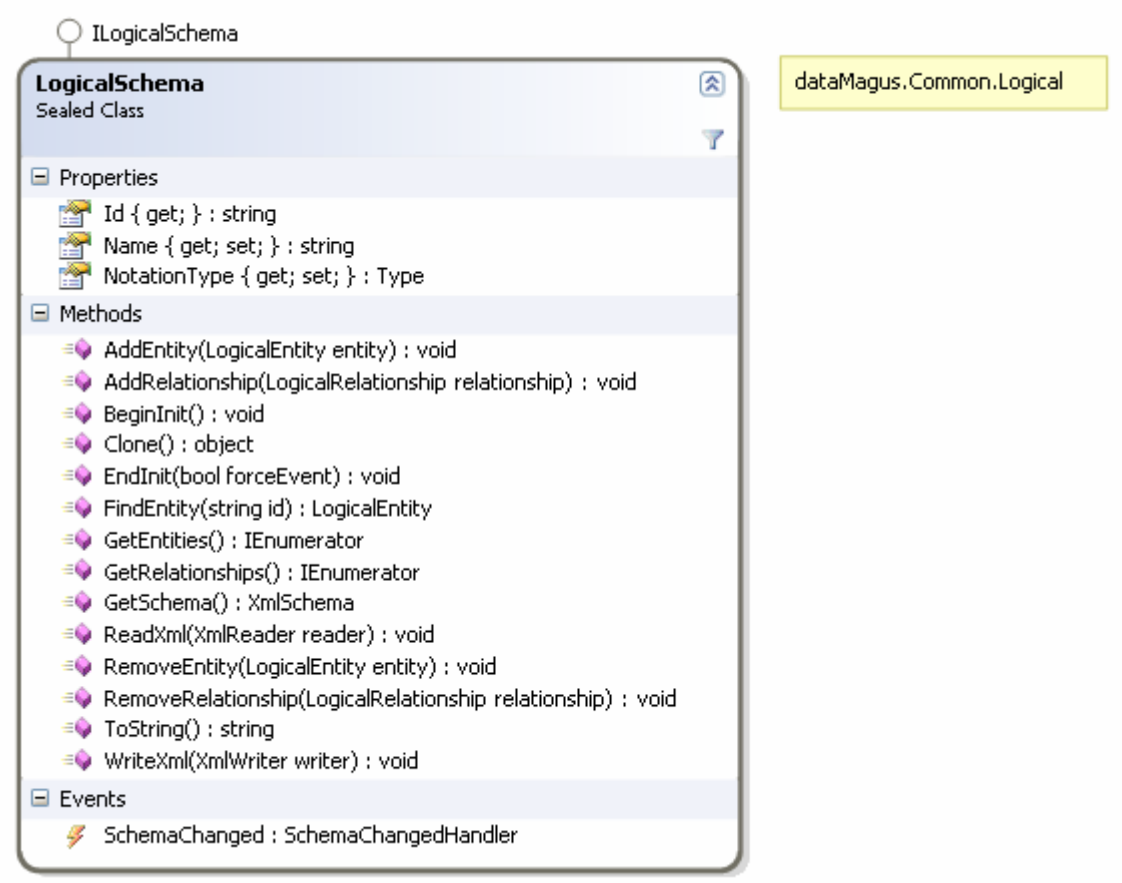


Abbildung 52 – Klasse für die Repräsentation der logischen Ebene

5.2.4 Physikalisch relationale Ebene

Alle Elemente der physikalisch relationalen Ebene sind von der abstrakten Klasse *PhysicalElement* abgeleitet. Diese Klassen sind spezifisch für das relationale Modell ausgelegt. Die unterschiedlichen physikalischen Modelle haben nichts mehr gemeinsam. Jedes physikalische Modell setzt das logische Modell nach seinen Konzepten um.
Die Klassen *PhysicalRelationalTable*, *PhysicalRelationalColumn* und *PhysicalRelationalIndex* stehen für die Relationen bzw. Tabellen, Spalten und Indexe eines relationalen Modells. Die abstrakte Klasse *PhysicalRelationalRelationship* abstrahiert die verschiedenen Beziehungsarten im relationalen Modell. Sie kann nicht eins-zu-eins mit der logischen Beziehung (*LogicalRelationship*) gleich gesetzt werden. In einem relationalen Modell gibt es „nur" Relationen, die Beziehungen werden mit Spalten und Foreign-Key-Constraints in den Tabellen abgehandelt. Technisch gesehen drücken wir dies jedoch mit einer *PhysicalRelationalRelationship* aus, welche dann auch dafür gebraucht wird, um die Beziehung (bzw. den Link) zwischen zwei Relationen visuell darzustellen.
Gemäss Umsetzung im *Kapitel 4.8.5* kennt das relationale Modell nur eine 1:1-Beziehung (Klasse *OneToOnePhysicalRelationalRelationship*) und eine 1:n-Beziehung (Klasse *OneToManyPhysicalRelationalRelationship*). Die Kardinalitäten der logischen Beziehung müssen entsprechend auf die relationalen umgelegt werden, die exakten Angaben der logischen Beziehungen entfallen.

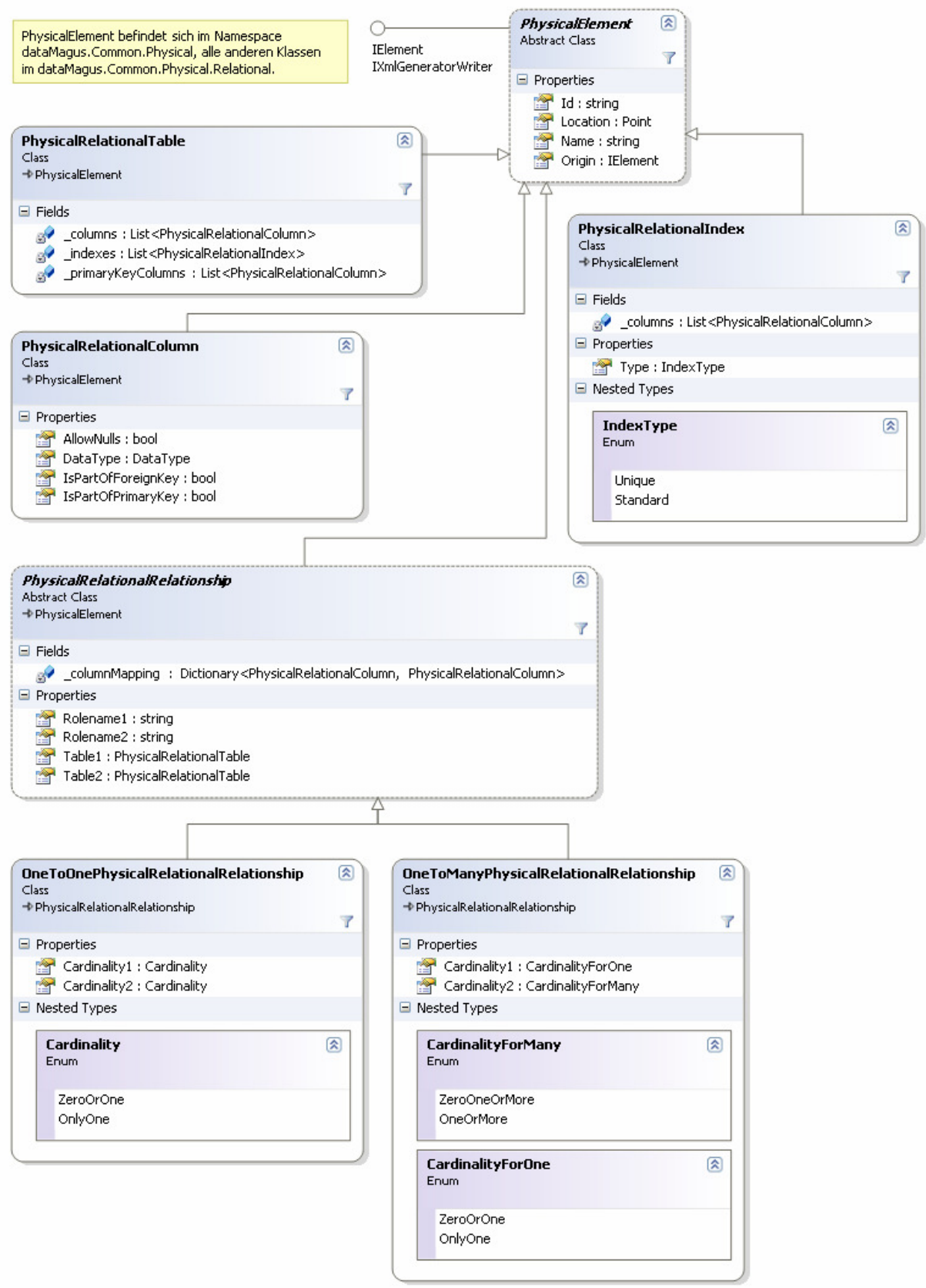


Abbildung 53 – Klassendiagramm *Physikalisch relationale Ebene*

Jeder Spalte kann ein Datentyp zugeordnet werden. Dabei handelt es sich um die physikalischen, abstakten Datentypen, die wir im *Kapitel 4.8.10 Definition der Datentypen* beschrieben haben. Per se sind sieben Basistypen vorhanden, die in der Enumeration *BasisDataType* definiert sind. Jeder Basistyp hat das Flag *isDerived* (*istAbgeleitetet*) auf *false*. Der Benutzer kann nun aufgrund dieser Basistypen weitere abstrakte Datentypen definieren. Solche benutzerspezifische Datentypen müssen immer zu genau einem Basistyp passen. Sie werden intern mit dem Flag *IsDerived=true* markiert.

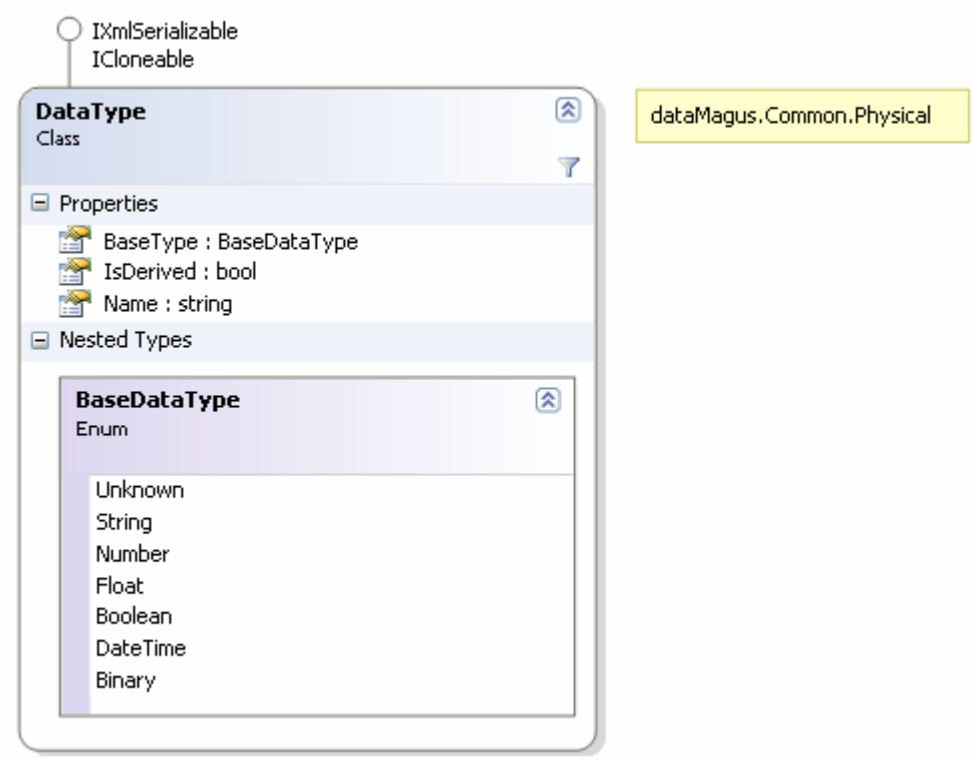


Abbildung 54 – Klasse für die physikalischen, abstrakten Datentypen

Das gesamte physikalisch relationale Schema wird in der Klasse *PhysicalRelationalSchema* gespeichert. Sie führt eine interne Liste der Tabellen (*List<PhysicalRelationalTable>*) und eine interne Liste der Beziehungen (*List<PhysicalRelationalRelationship>*).

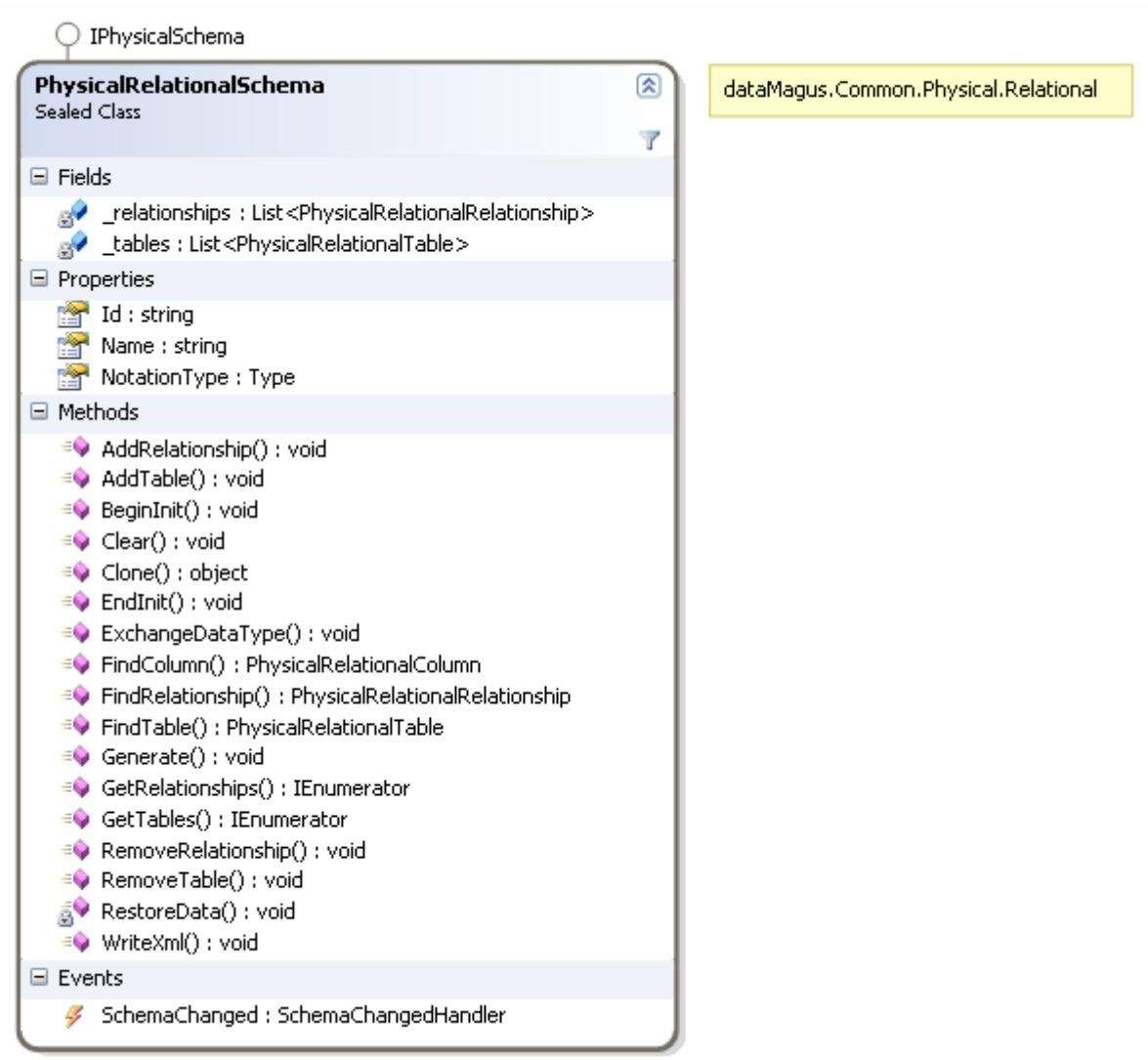


Abbildung 55 – Klasse für die Repräsentation der physikalisch relationalen Ebene

5.2.5 Physikalisch objektorientierte Ebene

Alle Elemente der physikalisch objektorientierten Ebene sind ähnlich der relationalen Ebene von der abstrakten Klasse *PhysicalElement* abgeleitet.
Das logische Modell wird in Klassen (Klasse *PhysicalObjectorientedClass*), Properties (Klasse *PhyiscalObjectorientedProperty*) und Assoziationen (Klasse *PhysicalObjectorientedAssociation*) umgesetzt. Eine Klasse kann analog der Attribute und Schlüsselattribute eine Liste von Properties und KeyProperties führen. Einem Property kann ein physikalisch abstrakter Datentyp zugeordnet werden.
Die Klasse *PhysicalObjectorientedAssociation* ist wiederum abstrakt. Eine Assoziation steht zwischen zwei Klassen, wobei es auch zwei Mal die gleiche Klasse sein kann. In einem physikalisch objektorientierten Modell sind Vererbungen (Klasse *IsAPhysicalObjectorientedRelationship*), 1:1-Assoziationen (Klasse *OneToOnePhysicalObjectorientedRelationship*), 1:n-Assoziationen (Klasse *OneToManyPhysicalObjectorientedRelationship*) und n:m-Assoziationen (Klasse *ManyToManyPhysicalObjectorientedRelationship*) zugelassen.

Das gesamte physikalisch objektorientierte Schema wird in der Klasse *PhysicalObjectorientedSchema* gespeichert. Sie führt eine Liste der Klassen (*List<PhysicalObjectorientedClass>*) und eine Liste der Assoziationen (*List<PhysicalObjectorientendAssociations>*).

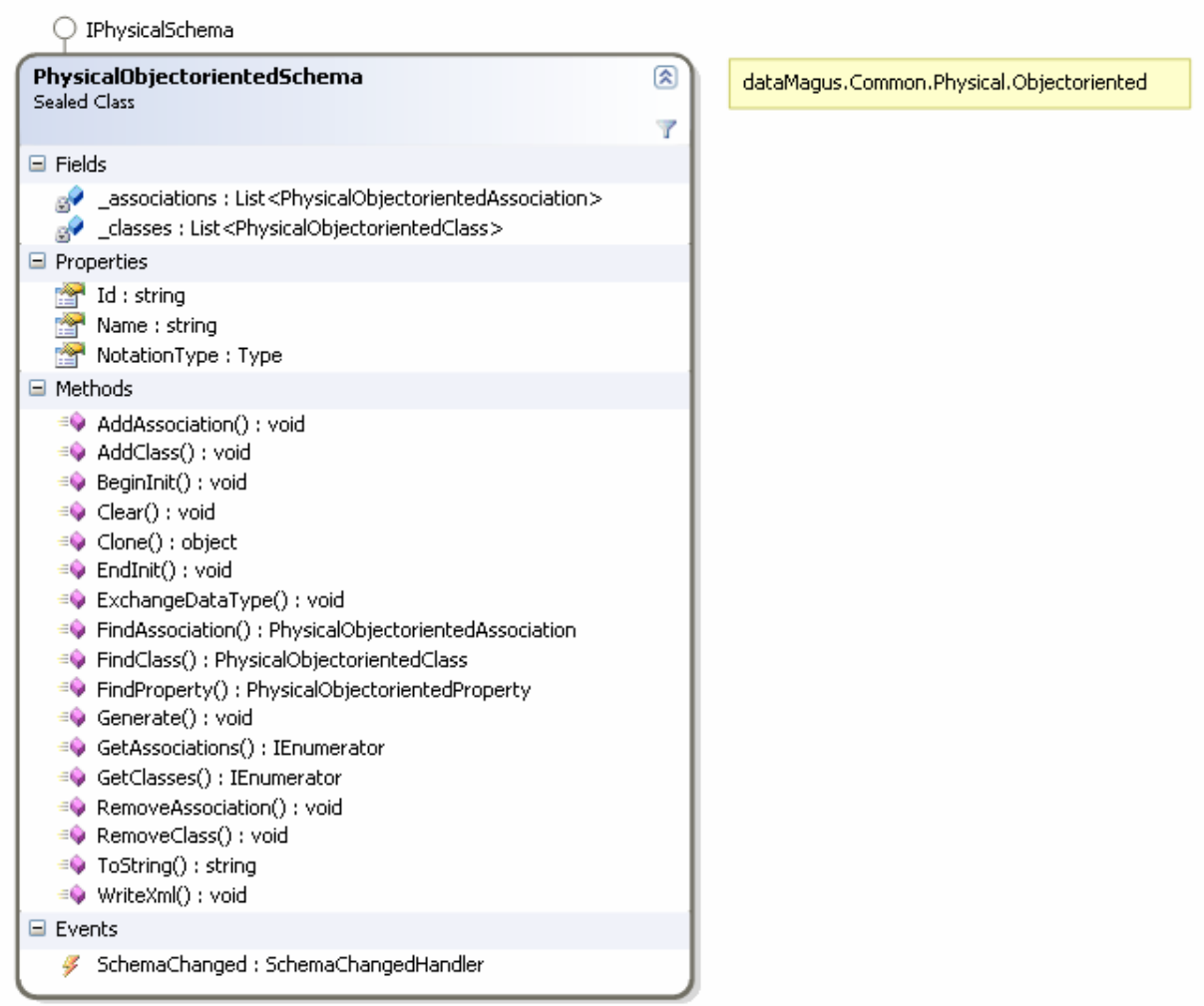


Abbildung 56 – Klasse zur Repräsentation der physikalisch objektorientierten Ebene

Abbildung 57 – Klassendiagramm *Physikalisch objektorientierte Ebene*

5.2.6 Umsetzung der logischen Ebene in die physikalischen Ebenen

Für die Umsetzung eines logischen Modells in das physikalische Modell kann über das Interface *IPhysicalSchema* die Methode *Generate()* aufgerufen werden. Diese Methode nimmt als Parameter eine Instanz einer Klasse mit implementierten *IUserInteraction* auf. Unsere Analyse zur Umsetzung hat ergeben, dass der Benutzer je nach modelliertem Konstrukt bestimmen muss, wie die Elemente umgesetzt werden sollen. Aus technischer Sicht ist dafür das Interface *IUserInteraction* definiert mit einer Methode *AskQuestion()*. Sollte während der Umsetzung eine Benutzerinteraktion notwendig sein, kann über dieses Interface die Antwort vom Benutzer abgeholt werden. Dieses Interface kann z.B. von einer Form-Klasse implementiert sein (Klasse *UserInteractionForm*), das eine Dialogbox anzeigt und die Benutzereingaben entgegen nimmt und direkt zurückleitet.
Für die Umsetzung des physikalisch relationalen Modells bedient sich hier die Klasse *PhysicalRelationalSchema* in der Methode *Generate()* der Hilfsklasse *PhysicalRelationalSchemaConverter*.
Nach der Umsetzung wird auf allen implementierten Generatoren zum jeweiligen physikalischen Schema die Methode *CleanExtendedData()* aufgerufen. Dieser Methode wird das soeben umgesetzte Schema mitgegeben. Falls diese Methode vom PlugIn-Entwickler oder von der PlugIn-Entwicklerin ausprogrammiert wurde, kann der Generator nun seine internen Daten mit dem neuen Schema gegenprüfen und ggf. seine Daten aktualisieren.

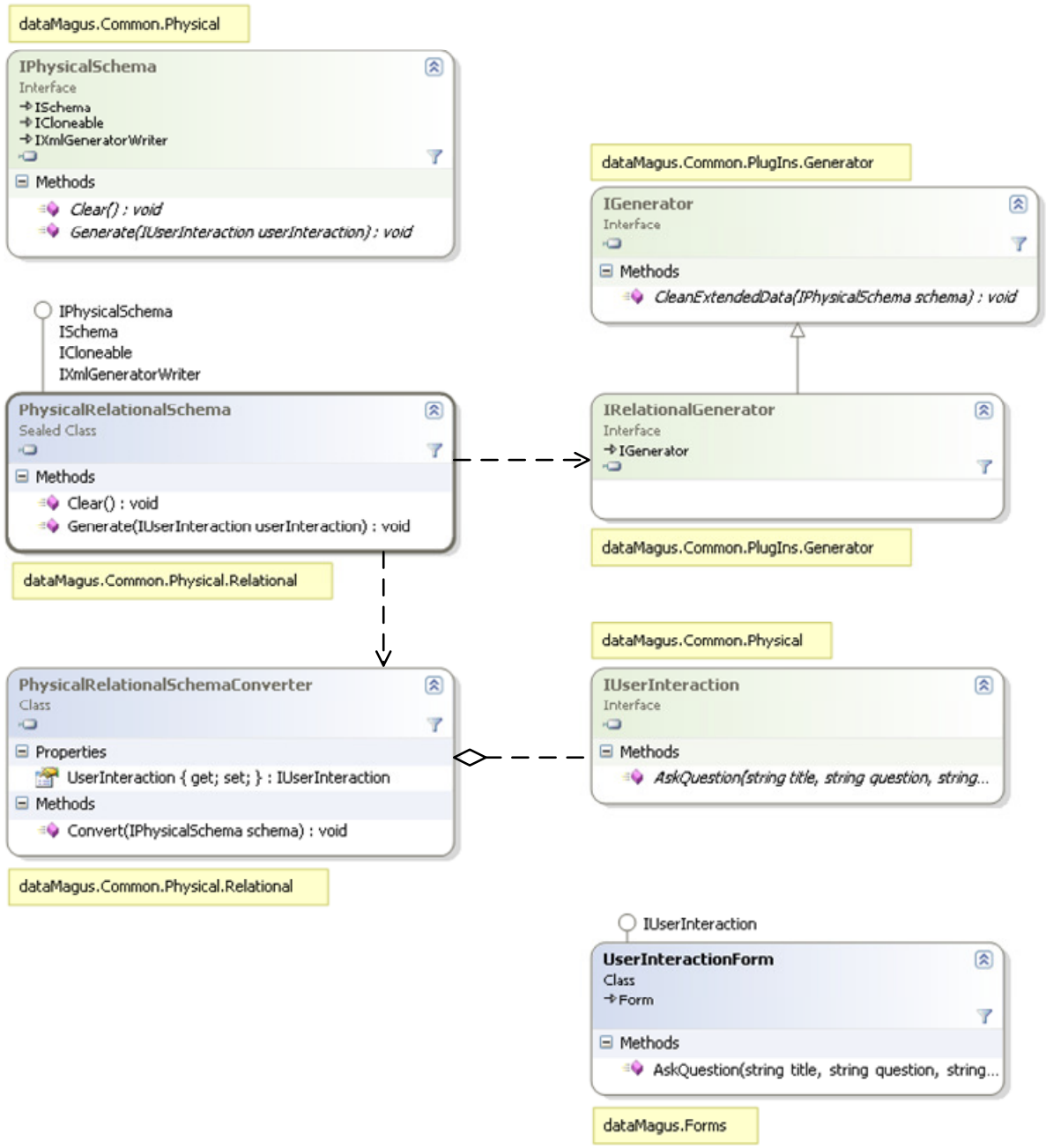


Abbildung 58 – Klassendiagramm zur Umsetzung der logischen Ebene in physikalische

5.2.7 Code-Generatoren

Um dataMagus mit weiteren Code-Generatoren anzureichern, stehen die Interfaces *IGenerator*, *IRelationalGenerator* und *IObjectorientedGenerator* zur Verfügung. *IGenerator* definiert die allgemeinen Elemente der Generatoren. Ein Generator hat einen Namen und kann lediglich für eine physikalische Ebene implementiert werden. Das Schema ist der Einfachheit halber auf *IGenerator* definiert, muss jedoch von jeder expliziten Implementation eines Generators auf den entsprechenden Typen untersucht werden.

Jeder Generator kann über interne generatorspezifische Daten verfügen. Er kann z.B. zusätzliche Attribute für eine Tabelle, Spalte, Index oder für das Schema oder über Optionen für den jeweiligen Generierungsprozess definieren. Solche Zusatzdaten muss der Generator selber verwalten. Wie und in welcher Struktur das passiert, ist jedem Generator-PlugIn-EntwicklerIn selber überlassen. Will er seine Daten auch persistent mit dem Modell speichern lassen, muss er seine Daten auf einer Klasse halten, die das Interface *IExtendedData* zwingend implementiert. Dieses Interface schreibt zusätzlich das Interface *IXmlSerializable* vor. Während des Serialisierungsprozesses werden bei allen vorhandenen Generatoren die Methode *WriteXml()* oder *ReadXml()* der Instanz der Klasse mit implementiertem *IExtendedData* aufgerufen. Ein PlugIn-Entwickler kann dort selber seine Daten in XML-Form in einen XML-Writer ein- oder auspacken.
Über die Methode *CleanExtendedData()* wird dem Generator zu bestimmten Zeitpunkten ein physikalisches Schema übermittelt, mit dem er seine internen Daten auf Richtigkeit überprüfen kann. Solche Aufrufe werden z.B. nach dem Umsetzen des logischen Schemas in die physikalischen Modelle getätigt. Es könnte sein, dass eine logische Beziehung gelöscht worden ist, daher die physikalische Beziehung auch nicht mehr umgesetzt wird, aber der Generator sich noch Daten zu dieser Beziehung gemerkt hat. In der *CleanExtendedData()*-Methode kann der Generator seine Daten gegenprüfen. Wir haben dieses Konstrukt extra etwas rudimentär gehalten, damit wir nicht für jedes Element eine Methode in den entsprechenden *IGenerator*-Interfaces definieren müssen.

Damit ein Generator dem Benutzer zu den jeweiligen physikalischen Elementen wie Schema, Tabellen, Spalten etc. Eingabemöglichkeiten anbieten kann, fragen wir auf dem entsprechenden Element jeweils bei den Generatoren an, ob sie dafür ein erweitertes *GeneratorControl* anbieten. Er kann also ein eigenes Control erstellen, aber es muss von *GeneratorControl* abgeleitet sein. Für die Entgegennahme der Eingabedaten sowie deren Verwaltung ist er, wie oben bereits erwähnt, selber zuständig. Dabei bieten sich die jeweiligen Events *Load*[19], *Closing*[20] resp. *Enter*[21] und *Leave*[22] an.
Die Methode *GetExtendedSchemaDataControl()* wird beim Dialogfenster für die Schema-Optionen aufgerufen und ermöglicht es dem Generator, weitere eigene Schema-Optionen aufzunehmen und abzulegen.
Die Methode *GetGeneratorDataTypeMappingDataControl()* ermöglicht die Aufnahme eines Datentyp-Mappings für die physikalisch abstrakten Datentypen.
An solchen Methoden wird das physikalische Element, zu dem weitere Attribute erfasst werden können, mitgegeben, damit eventuell noch nützliche Sachen abgefragt werden können. Natürlich handelt es sich dabei nur um eine Kopie des Elementes (**ein Klon**), da ein Generator **in keinem Falle das physikalische Schema verändern sollte**.

[19] Load: Beim Laden des Controls.
[20] Closing: Während des Schliessens des Controls.
[21] Enter: Das Control erhält den Fokus.
[22] Leave: Das Control verliert den Fokus.

Die Methode *GetGeneratorOptionDataControl()* dient für die Aufnahme von Optionen für den eigentlichen Generationsprozess. Dieser Prozess wird über die Methode *Generate()* angestossen, dem ein *Stream* mitliefert wird, in welchem der generierte Source-Code geschrieben werden kann. Das Resultat wird dem Benutzer danach angezeigt. Das Schema muss vorgängig gesetzt worden sein. Selbstverständlich wird auch hier nur eine Kopie des Schemas (**ein Klon**) übergeben.
Theoretisch hätten wir den Generatoren das physikalische Schema auch in XML-Form übergeben können. Dadurch wäre sichergestellt, dass es nicht verändert werden kann, weil es nicht zurückgelesen wird. Jedoch haben wir uns für eine Kopie des Schemas entschieden, weil es dem PlugIn-Entwickler bzw. der PlugIn-Entwicklerin dadurch leichter fällt, sich durch das ganze Schema zu traversieren und den Source-Code dafür zu generieren.

Für die Source-Code-Generierung stehen dem PlugIn-Entwickler grundsätzlich alle Türen offen. Da wir auf sämtlichen physikalischen Elementen das Interface *IXmlGeneratorWriter* implementiert haben, kann sich ein Generator-PlugIn-Entwickler z.B. das physikalische Schema mit der Methode *WriteXml()* als XML abholen, den Source-Code mit einer XSLT-Transformation erstellen lassen und in den Resultat-Stream schreiben. Natürlich gibt es auch weitere Möglichkeiten, wie sich z.B. durch das Schema zu bewegen und den Code in eine String-Variable zu schreiben.

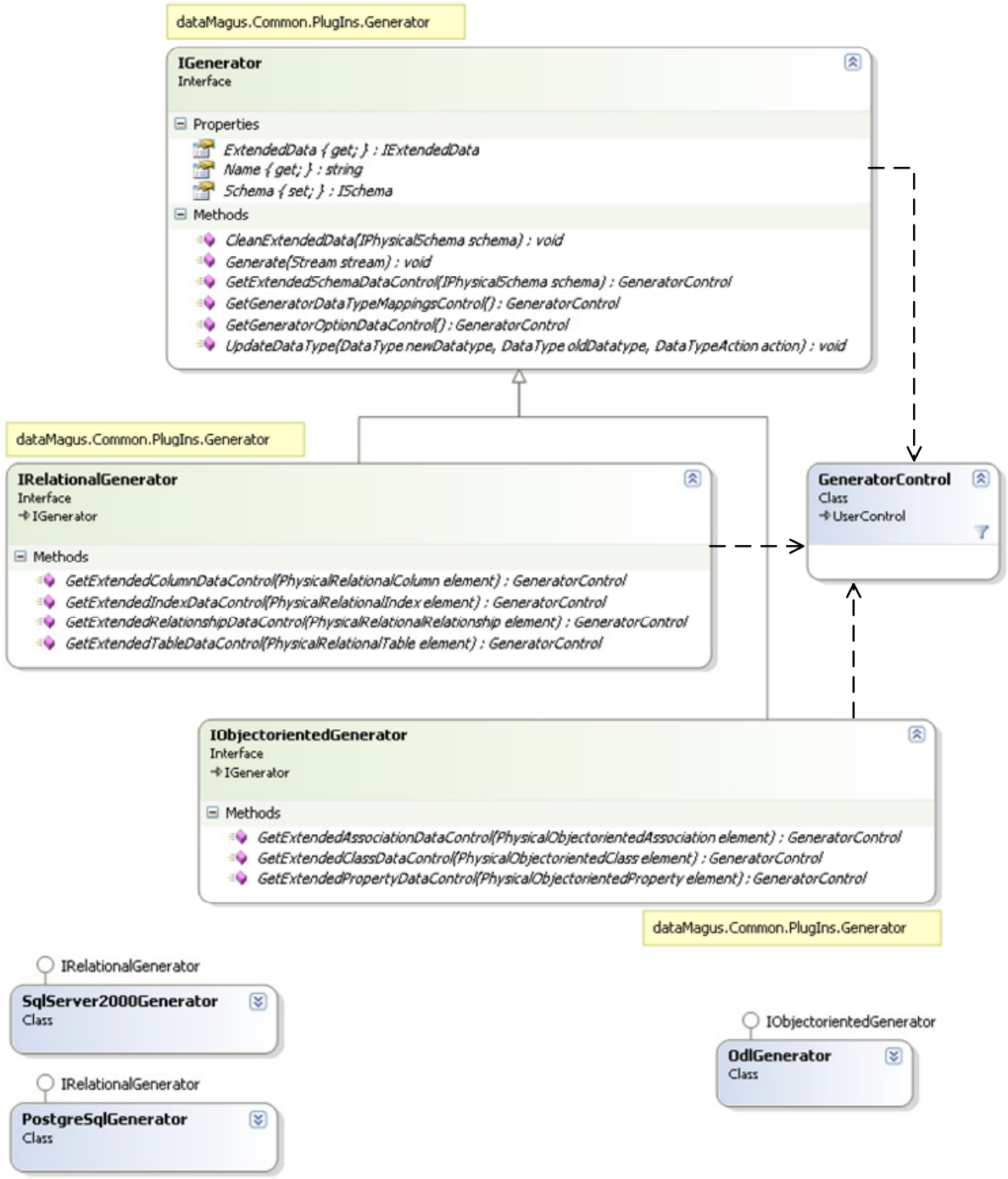


Abbildung 59 – Code-Generatoren

Generatoren, die Quellcode für das physikalische relationale Schema erstellen, müssen das Interface *IRelationalGenerator* implementieren, welches von *IGenerator* abgeleitet ist. Es definiert zusätzlich noch Methoden, um für die spezifischen relationalen Elemente Zusatzdaten angeben zu können. So kann z.B. eine Eingabemöglichkeit für Zusatzdaten für Spalten (*PhysicalRelationalColumn*) mit der Methode *GetExtendedColumnDataControl()* zur Verfügung gestellt werden. Es gibt für jedes physikalische Element (*PhysicalRelationalTable*, *PhysicalRelationalIndex*, *PhysicalRelationalRelationship* und *PhysicalRelationalColumn*) eine korrespondierende Methode.

Das Interface *IObjectorientedGenerator* ist das Pendant für das objektorienterte Schema.

Mit der ersten Version von dataMagus liefern wir zwei spezifische Generatoren aus, die für das relationale Schema Quellcode generieren. Es handeln sich um die Zielsysteme SQL Server 2000 (Klasse *SqlServer2000Generator*) und PostgreSQL (Klasse *PostgreSqlGenerator*). Eine Beschreibung der Implementationen findet sich im *Kapitel 6.1 SQL Server 2000* und *6.2 PostgreSQL*.
Für das objektorientierte Schema existiert eine Codegenerierung nach dem ODMG-Standard (Klasse *OdlGenerator*). Die genaue Beschreibung kann im *Kapitel 6.3 ODL* nachgelesen werden.

Die Klasse *GeneratorManager* (*sealed* und als *Singleton* implementiert) findet automatisch alle Generatoren, indem es nach Klassen mit dem implementierten Interface *IGenerator* sucht und sich eine Instanz der Klassen hält. Über diese Klasse können die Generatoren angesprochen werden.

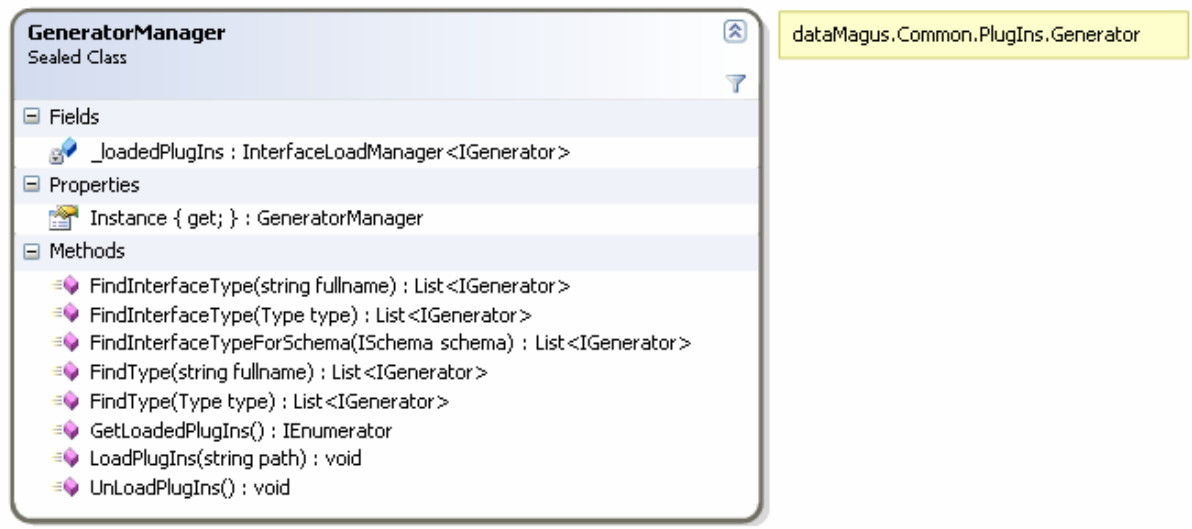


Abbildung 60 – Klasse *GeneratorManager*

Der *GeneratorManager* sucht sich die Implementationen der Interfaces wiederum mit Hilfe der generischen Klasse *InterfaceLoadManager*.

5.2.8 Notationen

Wie schon bei den Generatoren liegt das Augenmerk bei den Notationen auch auf der Erweiterbarkeit. Bei den Notationen gibt es keine Veränderung oder Ergänzung der jeweiligen Datenstrukturen. Trotzdem gibt es hier eine sehr enge Bindung zwischen den Businessobjekten und der Notation. Konkret bedeutet das, dass wir hier keine Klone einsetzen wollen. Unsere Entscheidung ist auf der Tatsache begründet, dass sich Klone nicht an die Changed-Events der Businessobjekte hängen können. Würden wir mit Klonen arbeiten, so müssten alle Events über das *SchemaControl* (Control für die Modellierungsfläche) weitergeleitet werden.
Dies wiederum sehen wir als eher schlechte Architektur an. Zudem betrachten wir die Notations-PlugIns als Programmbestandteile, die sich daran zu halten haben, keine Daten zu verändern.

Unser grundlegendes Konzept für die Umsetzung der Notationen basiert auf den *UserControls* vom Microsoft .NET Framework. Damit ermöglichen wir der Notation auf einfache Weise, eine Ansicht der Daten umzusetzen. Wir sehen die Elemente Entitätsmengen, Tabellen und Klassen als sog. Controls an, die Elemente Beziehungen (logisch und relational) und Assoziationen (objektorientiert) jedoch nicht. Dies hat den Grund, dass Beziehungen immer zwischen einer oder mehreren Entitätsmengen bestehen. Eine Beziehung ohne Entitätsmenge existiert nicht, sondern sie ist immer ein Link auf dieselbe oder eine andere Entitätsmenge. Deshalb schauen wir eine Beziehung zeichnerisch nicht als eine eigenständige Einheit an. Sie wird nur als Bindeglied eingesetzt und stellt lediglich die Information, wie Entitätsmengen miteinander in Beziehung stehen.
Die Notationen müssen diese Beziehung jeweils auf einem grafischen Kontext selber zeichnen. Die Controls hingegen können einfach instanziert werden und an dataMagus zurückgegeben werden. Auf sämtliche Events innerhalb dieser notationspezifischen Controls kann ein PlugIn-Entwickler oder eine PlugIn-Entwicklerin selber reagieren. Es wird keinerlei Interaktion mit dem Schema benötigt. Weil dataMagus für die Entitätsmengen (bzw. Tabellen oder Klassen) ein *UserControl* verlangt, wird es dem Entwickler oder der Entwicklerin ermöglicht, seine Controls mit einem grafischen Editor zu designen. Hätten wir das nicht so gemacht, müsste sich er oder sie das Zeichnen der Entitäten (bzw. Tabellen oder Klassen) komplett selber programmieren.

Wie bei den Generatoren gibt es auch bei den Notationen ein Basis-Control namens *ElementControl*. Es ist ein abgeleitetes *UserControl* und bietet bereits eine Kommunikationsplattform zwischen dem Schema und dem Control. Es kümmert sich darum, dass das Schema informiert wird, wenn das Control bewegt oder selektiert wird. Damit hat das Schema die Möglichkeit „controlübergreifend" zu agieren (z.B. das Verschieben noch anderer selektierter Controls). Zudem stellt es Methoden bereit, die vom abgeleiteten Control überschrieben werden sollten, um korrekt funktionieren zu können.

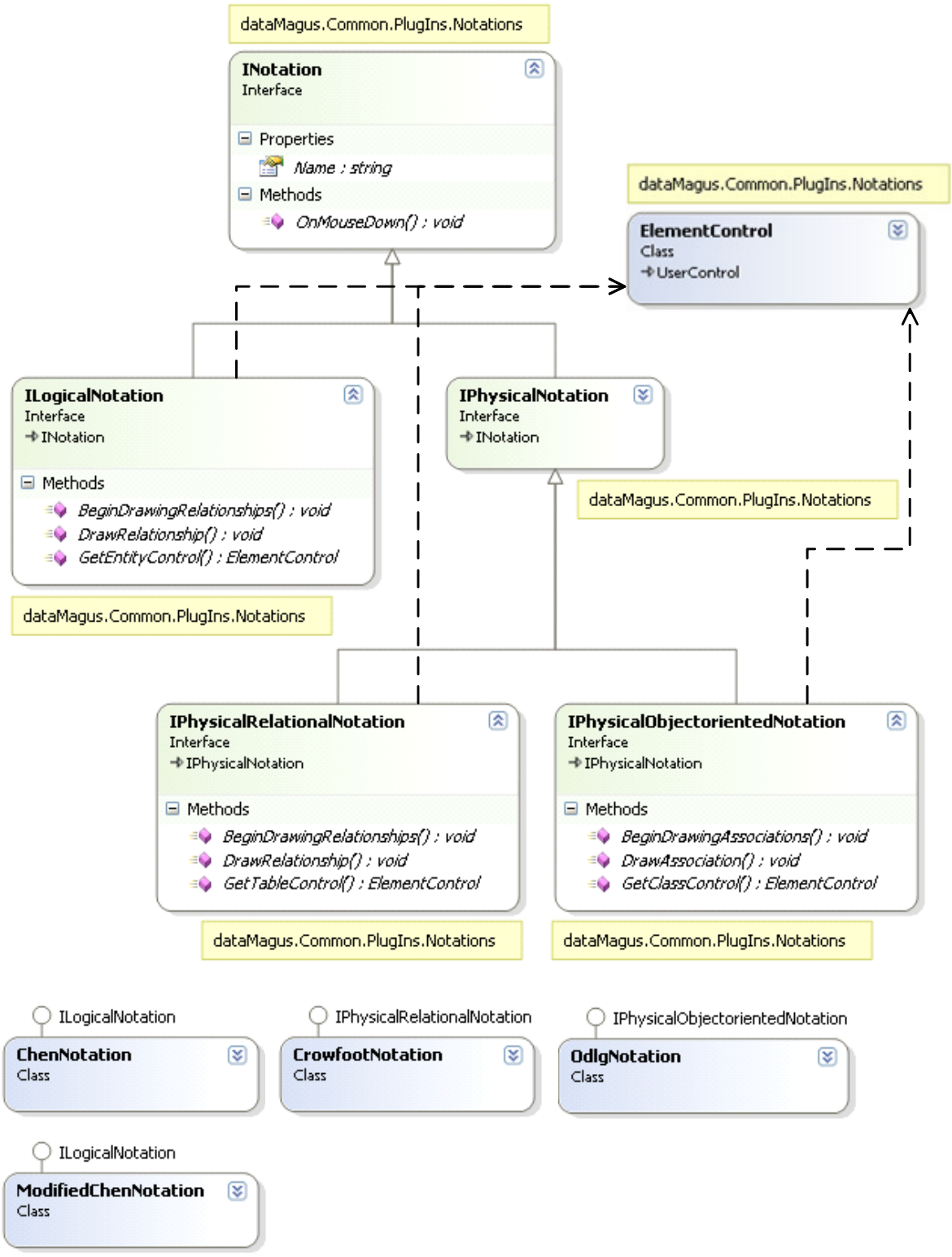


Abbildung 61 – Klassendiagramm *Notationen*

Das Interface *INotation* stellt die Basis aller Notationen dar. Da sich die jeweiligen Businessobjekte auf den verschiedenen Ebenen unterscheiden, beinhaltet dieses Interface lediglich den Namen und einen OnMouseDown-Event. Über diesen Event wird der Notation mitgeteilt, in welchen Bereich der Benutzer auf das Schema geklickt hat. Obwohl der Entwickelnde auf Events in seinem eigenen Control selber reagieren kann, hat er/sie keinen

Einfluss auf Events von aussen. Dieser Event wird weitergeleitet, damit die Notation auch auf Benutzeraktionen von gezeichneten Beziehungen reagieren kann.

Das Interface *ILogicalNotation* stellt das Basisinterface für logische Notationen dar. Es beinhaltet drei Methoden. Die Methode *BeginDrawingRelationships()* dient zu informativen Zwecken. Darin hat die Notation die Gelegenheit, Vorbereitungs- oder Aufräumarbeiten zum Zeichnen der Beziehungen durchzuführen. Die *DrawRelationship()*-Methode dient zum Zeichnen der jeweiligen Beziehung in einen grafischen Kontext. Sie wird bei jedem *OnPaint-Event* auf dem Schema wieder aufgerufen. Die Methode *GetEntityControl()* wird benutzt, um ein Control für eine Entitätsmenge zu bekommen. Dieses Control wird dann auf dem Schema platziert.

Das Interface *IPhysicalNotation* ist ein reines Markerinterface für physikalische Notationen und enthält keine Methoden. Davon leiten sich die beiden Interfaces *IPhysicalRelationalNotation* und *IPhysicalObjectorientedNotation* ab. Die Methoden innerhalb dieser Schnittstellen bilden das Pendant zu den Methoden im Interface *ILogicalNotation* und funktionieren identisch.

Die Klasse *NotationManager* (*sealed* und als *Singleton* implementiert) findet automatisch alle Notationen, indem es nach Klassen mit dem implementierten Interface *INotation* sucht und sich eine Instanz der Klassen hält. Über diese Klasse können die Notationen angesprochen werden. Sie funktioniert genau gleich wie der *GeneratorManager*.

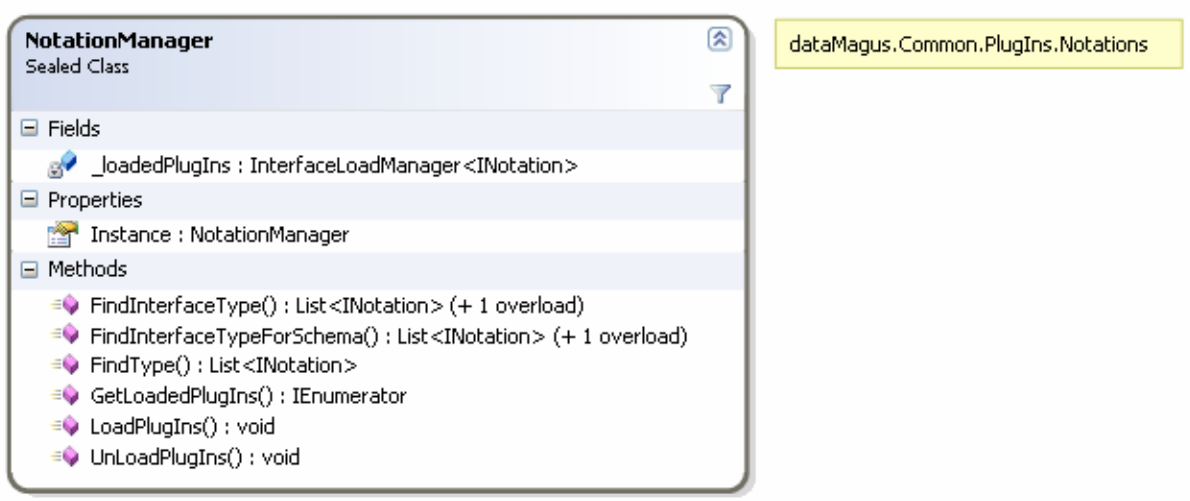


Abbildung 62 – Klasse *NotationManager*

Die Herausforderung einer Notation besteht in der korrekten Darstellung der jeweiligen logischen oder physikalischen Daten. Diese Aufgabe beinhaltet eine Menge an kleinen Stolpersteinen, die es zu lösen gilt. Damit die Notationen-PlugIn-EntwicklerIn jeweils nicht komplett auf sich selber angewiesen sind, haben wir die spezielle Klasse *NotationHelper* eingeführt. Diese Klasse beinhaltet ein paar zentrale Funktionen, die das Zeichnen erleichtern sollen. Sie enthält auch kleinere Algorithmen, um die rudimentären Probleme zu lösen. Sie haben weniger eine graphische Relevanz, sondern dienen vielmehr dem praktischen pragmatischen Ansatz.

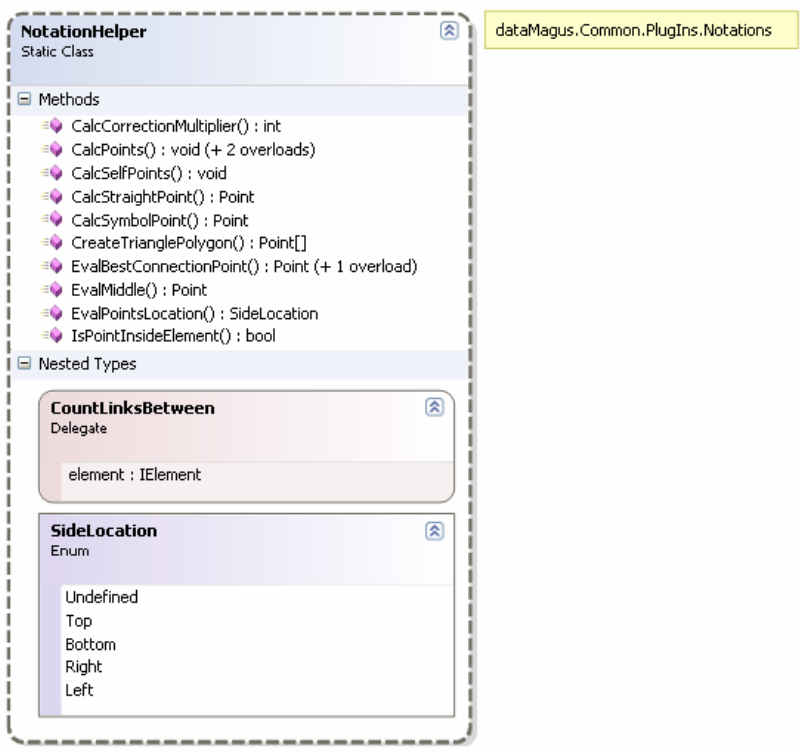


Abbildung 63 – Hilfsklasse *NotationHelper* zum Zeichnen

Nachfolgende Abschnitte erklären die Methoden der Klasse.

5.2.8.1 Koordinatensystem

Das Koordinatensystem im Grafikkontext von .NET ist folgendermassen aufgebaut: Es beginnt oben links mit dem Punkt 0/0. Punkte im Minusbereich werden nicht gezeichnet.

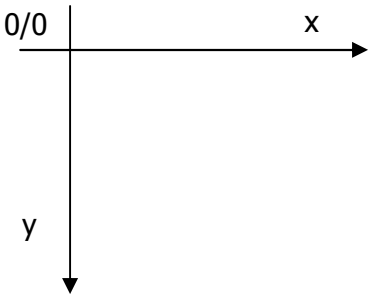


Abbildung 64 – Koordinatensystem in .NET

5.2.8.2 Berechnen von elementaren Hilfspunkten

Die Methoden *CalcPoints()*, *CalcStraightPoint()*, *CalcSymbolPoint()* und *CalcSelfPoints()* des *NotationHelpers* dienen alle zur Berechnung von Hilfspunkten, um das Zeichnen von Beziehungslinien zu erleichtern. Um die Rückgabewerte dieser Methoden zu verstehen, muss der Entwickelnde zuerst wissen, wie die Punkte richtig eingesetzt werden.

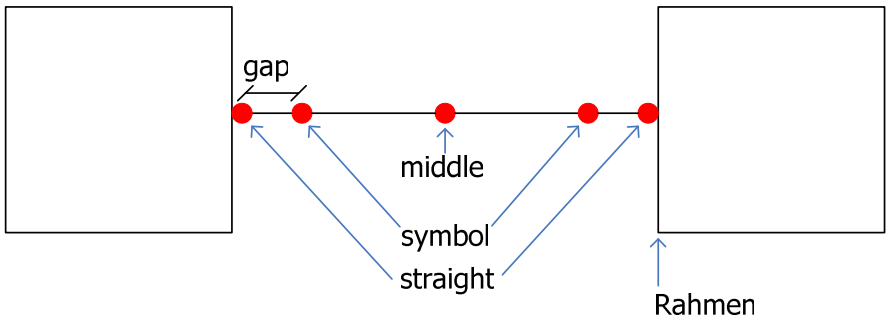


Abbildung 65 – Punkte bei einer normalen Beziehung

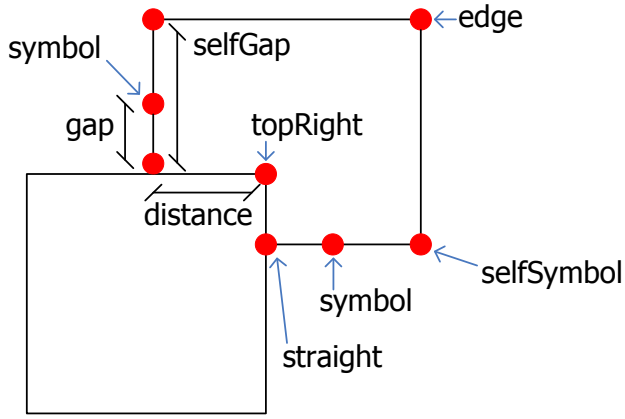


Abbildung 66 – Punkte bei einer Selbstbeziehung

Der Symbol Punkt

Mit dem Symbol Punkt wird bei einer normalen Beziehung der Ort bestimmt, wo sich der Rahmen des einen Controls mit der Gerade zwischen den Mittelpunkten der beiden Controls schneidet. Mit einem künstlich gesetzten erweiterten Rahmen (gap), kann sich dieser Punkt vom Control wiederum wegbewegen. Dieser Punkt wird normalerweise **zum Zeichnen von Kardinalitäten oder Rollennamen** benutzt.
Bei einer Selbstbeziehung liegt der Punkt auf den jeweiligen Beziehungseingängen mit Einberechnung des Gaps.
Bei einem Gap der Grösse 0 liegt der Symbol Punkt auf dem Straight Punkt.

Der Straight Punkt

Bei einem Straight Punkt ist der Punkt gemeint, welcher senkrecht (im Winkel von 90°) zu seinem Symbolpunkt steht und auf dem effektiven Rahmen des Controls zu liegen kommt. Er kann dazu verwendet werden, die **Notationssymbole oder Kardinalitätssymbole an das Control zu zeichnen**. Dieser Punkt ist von der Position seines Symbolpunktes abhängig und wird sich mit diesem zusammen jeweils verändern resp. mitbewegen.

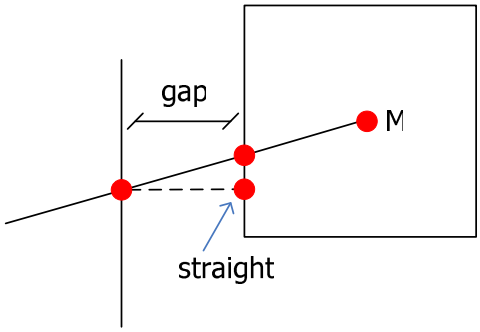


Abbildung 67 – Berechnung des Straight Punktes

Berechnung der Schnittpunkte

Die Methode *CalcSymbolPoint()* berechnet die Schnittpunkte von zwei über eine Beziehung verbundenen Controls. Diese Berechnung erlaubt dem Benutzer eine **stufenlose Linienführung entlang der Controls**. Dazu ist nur rudimentäre Trigonometrie notwendig.

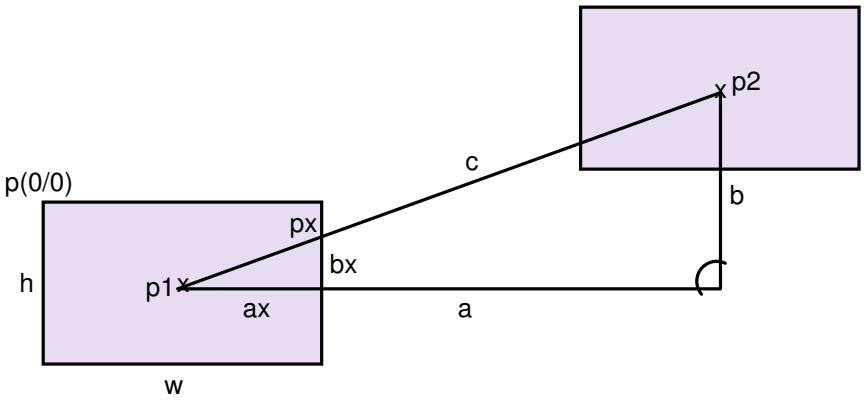


Abbildung 68 - Berechnung der Schnittpunkte

Die Punkte *p1* und *p2* sind die Referenzpunkte für eine gerade Verbindung. Die Höhe *h* und Länge *w* beider Controls sind bekannt. Somit lässt sich mit den beiden Punkten *p1* und *p2* ein rechtwinkliges Dreieck mit den Seiten *a* und *b* konstruieren. Diese Seiten können über die Koordinaten der beiden Verbindungspunkte errechnet werden. Nun lässt sich mit dem Verhältnis *a* zu *b* die Seite *bx* errechnen, da *ax* die halbe Länge *w* des Controls darstellt. Somit lässt sich im Koordinatensystem der gesuchte Punkt *px* errechnen. Die unten stehenden Formeln können dafür benutzt werden.

a, *b* und *ax* sind einfach zu berechnen:

$$a = p2_x - p1_x$$

$$b = p2_y - p1_y$$

$$ax = p1_x + \frac{w}{2}$$

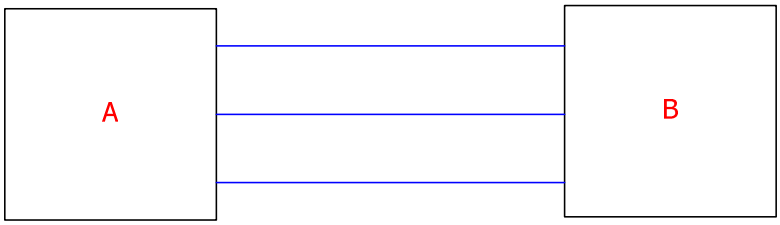
Nun ist *bx* über das Verhältnis berechenbar:

$$\frac{a}{b} = \frac{ax}{bx} \Rightarrow bx = \frac{ax \cdot b}{a}$$

5.2.8.3 Hilfspunkte zu Mehrfachbeziehungen

Die Methode *CalcCorrectionMultiplier()* ist eine Hilfsmethode, um mit dem Problem von mehreren Beziehungen zwischen den gleichen Entitätsmengen (bzw. Tabelle oder Klasse) umzugehen. Es muss berücksichtigt werden, dass Mehrfachbeziehungen nicht an gleicher Stelle gezeichnet werden, da sich diese sonst überlagern. Die Methode *CalcCorrectionMultiplier()* beinhaltet einen kleinen Algorithmus zum Berechnen eines Korrekturfaktors, welcher mit den Verbindungspunkten multipliziert werden kann, damit sie sich auseinander bewegen und beide Beziehungen sichtbar werden.

Der Algorithmus funktioniert folgendermassen:
Die Methode holt sich von der Entitätsmenge ihre Anzahl Beziehungen zu einer anderen Entitätsmenge ab.



Hash(Id von A) + Hash(Id von B) = x

Listeneintrag = (x, 3)

Abbildung 69 – Berechnung Hashcode zu den Mehrfachbeziehungen

Diese Information wird über den eindeutigen Hashcode der zusammengesetzen Ids der jeweiligen Entitätsmengen-Elemente in einer Liste abgelegt. Wir berechnen einen Hashcode über die Id der Entitätsmenge A und addieren ihn mit dem berechneten Hashcode der Id der Entitätsmenge B. So entsteht ein eindeutiger Kennzeichner für die Beziehungen zwischen den beiden Entitätsmengen. Da die Ids durch die GUID weltweit eindeutig sind, wird der Hashcode zu einer hohen Prozentzahl eindeutig sein. Theoretisch könnte die Addition zweier anderer Entitäten den gleichen Hashcode liefern. Die Wahrscheinlichkeit ist jedoch so gering, dass wir das vernachlässigt haben.
Haben z.B. zwei Entitätsmengen drei Beziehungen miteinander, wird die Zahl "drei" zum Hashcode abgelegt. Sie gilt als dekrementierender Zähler und wird bei einer geraden Anzahl von Beziehungen noch um eins erhöht. Nun wird diese Methode ja beim Zeichnen von jeder der drei Beziehungen aufgerufen. Weil die Entitätsmengen der drei Beziehungen identisch sind, kann bei jedem Aufruf wieder auf den "gemeinsamen" Zähler zugegriffen werden, welcher jedes Mal dekrementiert und durch zwei geteilt wird. Bei der daraus resultierenden Zahl werden allfällige Kommastellen abgeschnitten und bei einem ungeraden Zähler zusätzlich invertiert. Somit entsteht eine Zahlenreihe, die sich abwechslungsweise im positiven und negativen Zahlenbereich aufhält.

Beispiel:
3 Beziehungen ➔ Zähler = 3 → Reihe = -1, 1, 0
4 Beziehungen ➔ Zähler = 4+1 → Reihe = -2, 2, -1, 1, 0
5 Beziehungen ➔ Zähler = 5 → Reihe = -2, 2, -1, 1, 0
...

Diese Zahlenreihe kann nun als Faktor verwendet werden, um Punkte, die einander überdecken, zu korrigieren. Werden mit der Reihe die Mittelpunkte der Beziehungen (die sich momentan noch überlagern) korrigiert, sind danach alle Beziehungen sichtbar.

Beim Beispiel in Abbildung 70 wurde der Faktor dafür verwendet, die Y-Koordination der Mittelpunkte der Rhomben zu verändern.

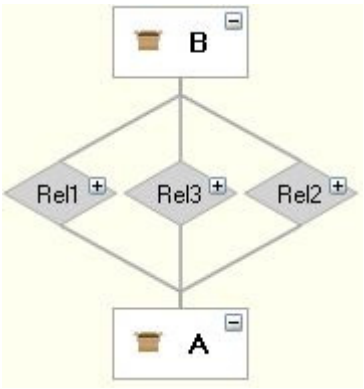


Abbildung 70 – Korrektur der Mittelpunkte auf der Y-Achse

In Abbildung 71 wurde mit dem Faktor die X-Koordinaten der Mittelpunkte der Rhomben korrigiert.

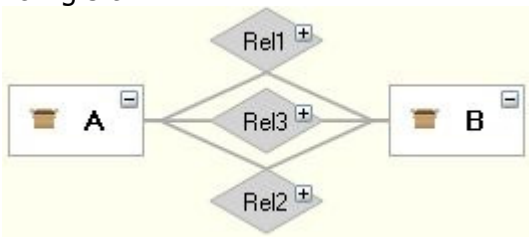


Abbildung 71 – Korrektur der Mittelpunkte auf der X-Achse

Alternativ könnte man auch die Symbol-Punkte mit den Straight-Punkten korrigieren wie in Abbildung 72.

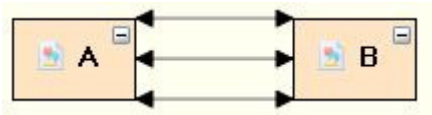


Abbildung 72 – Korrektur der Symbol- und Straight-Punkte auf der X-Achse

5.2.8.4 Weitere Hilfsmethoden

Mit der Hilfe der *CreateTrianglePolygon()*-Methode kann auf ein einfaches Dreieck-Polygon gemittelt werden, um Pfeile zu zeichnen.

Die *EvalBestConnectionPoint()*-Methode kann dazu verwendet werden, um einen möglichst optimalen Verbindungspunkt zwischen einem Punkt und vier möglichen Punkten zu suchen. Sie wird vor allem verwendet, um den Verbindungspunkt eines Rhombus zu suchen.

Wie die nachfolgende Abbildung zeigt, sind die Verbindungspunkte an unterschiedlichen Stellen der einzelnen Rhomben festgelegt.

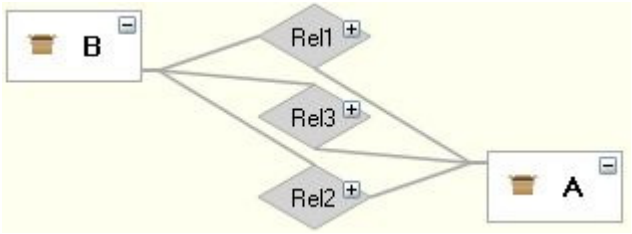


Abbildung 73 – Ermitteln des besten Verbindungspunktes

Mit der *EvalMiddle()*-Methode kann die Mitte zwischen zwei gegebenen Punkten berechnet werden. Dies ist nützlich, um in der Mitte einer Beziehung einen Namen oder ein Symbol wie z.B. einen Rhombus einzuzeichnen.

Die *EvalPointsLocation()*- Methode evaluiert aufgrund eines Symbol- und dazugehörenden Straight-Punktes, auf welcher Seite des Controls sich die beiden Punkte befinden müssen. Dabei wird davon ausgegangen, dass sich die beiden Punkte immer miteinander bewegen und auf gleicher X- resp. Y-Koordinate stehen. Zudem muss der Straight-Punkt immer näher oder gleich nahe am Control stehen wie der Symbolpunkt. Der Rückgabewert (Enumeration *SideLocation*) dieser Methode ist entscheidend, in welche Richtung z.B. Pfeiler eines Beziehungsendes gezeichnet werden müssen.

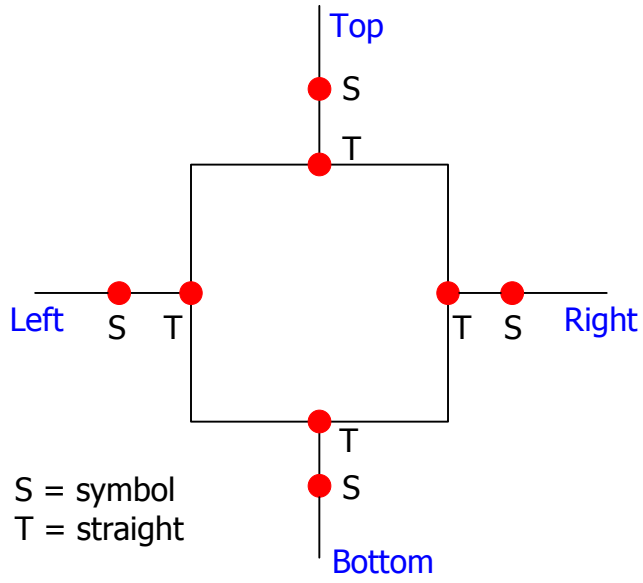


Abbildung 74 – Bestimmung der Seite

Die *IsPointInsideElement()*-Methode überprüft, ob sich ein Punkt noch innerhalb des Element-Controls befindet. Sollte sich ein Punkt über den Rand hinausbewegen, kann erkannt werden, dass evtl. ein Eckpunkt erreicht ist und die Richtung gewechselt werden muss.

5.2.8.5 Visuelle Beispiele zur Anwendung der Hilfsmethoden

Mit den oben beschriebenen Methoden können nun die Controls auf der Modellierungsfläche mit der Maus verschoben werden, wobei sich die Verbindungslinien je nach Position der Controls ändern und im Kontext bestehen bleiben. Wenn man über die kleine Unschönheit mit dem „Gap" hinweg sieht, werden die Verbindungslinien doch einigermassen „smooth" mit den Controls mitbewegt und gezeichnet.

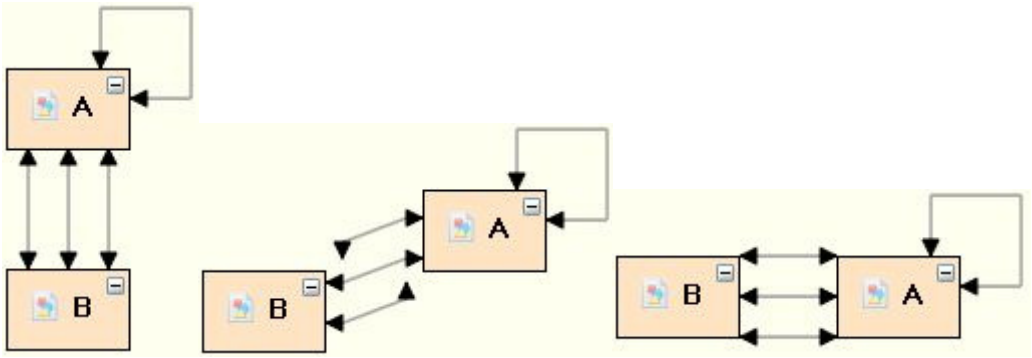


Abbildung 75 – Bilderserie zum Verschieben der Controls in der ODL-Notation

Je nachdem wie man die Punkte korrigiert (mit dem Faktor aus der Methode *CalcCorrectionMultiplier()*), können unterschiedliche Darstellungsformen abgebildet werden. Es kommt darauf an, wie und ob ein Notationsentwickler die Punkte überhaupt verwendet. Abbildung 76 zeigt ein Konstrukt von mehreren Beziehungen zwischen zwei Entitäten, bei denen die Variante der Korrektur des Mittelpunktes gewählt wurde.

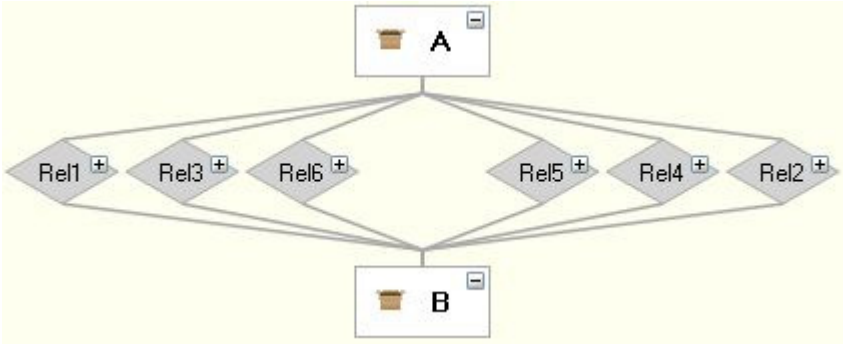


Abbildung 76 – Mehrfachbeziehungen mit Korrektur des Mittelpunktes in Chen

Wählt man nun die Variante der Verschiebung der Symbol- und Straight-Punkte, so sieht obiges Beispiel wie in Abbildung 77 aus. In diesem Fall lässt die Notation die Linien über das Control-Seitenende heraus laufen. Will man dies noch korrigieren, müsste die Programmierung des Zeichnens der Notation verfeinert werden. Es stellt sich jedoch die Frage, wie oft ein solches Konstrukt überhaupt modelliert werden muss.

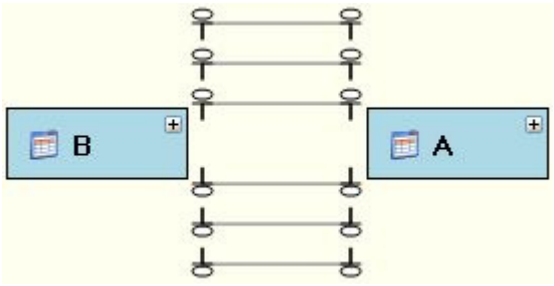


Abbildung 77 – Mehrfachbeziehungen in der Crowfoot-Notation (Spezialfall)

Durch das Verschieben können auch etwas wirre Konstrukte entstehen. Jedoch passieren diese, weil wir die Positionierung der Linien nach einem einfachen, mathematischen Algorithmus berechnen.

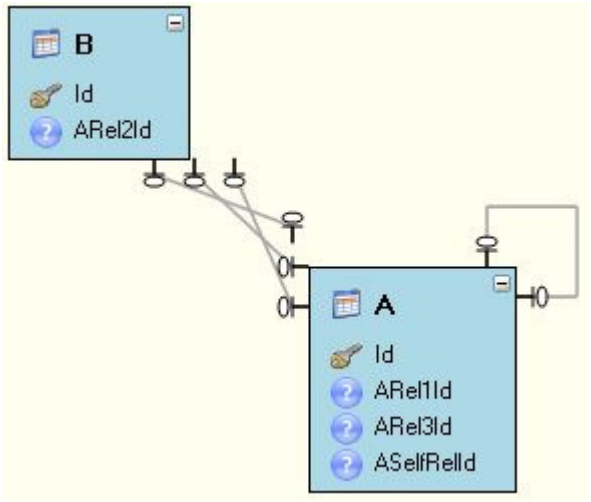


Abbildung 78 – Unschöne Darstellung

5.2.9 Datencontainer

Das modellierte logische Schema wird in einem zentralen Datencontainer abgelegt. Die Klasse *DataContainer* ist als *sealed* markiert und als *Singleton* implementiert. Ausserdem ist sie mit *internal*-Modifier nur Assembly-intern zugänglich.

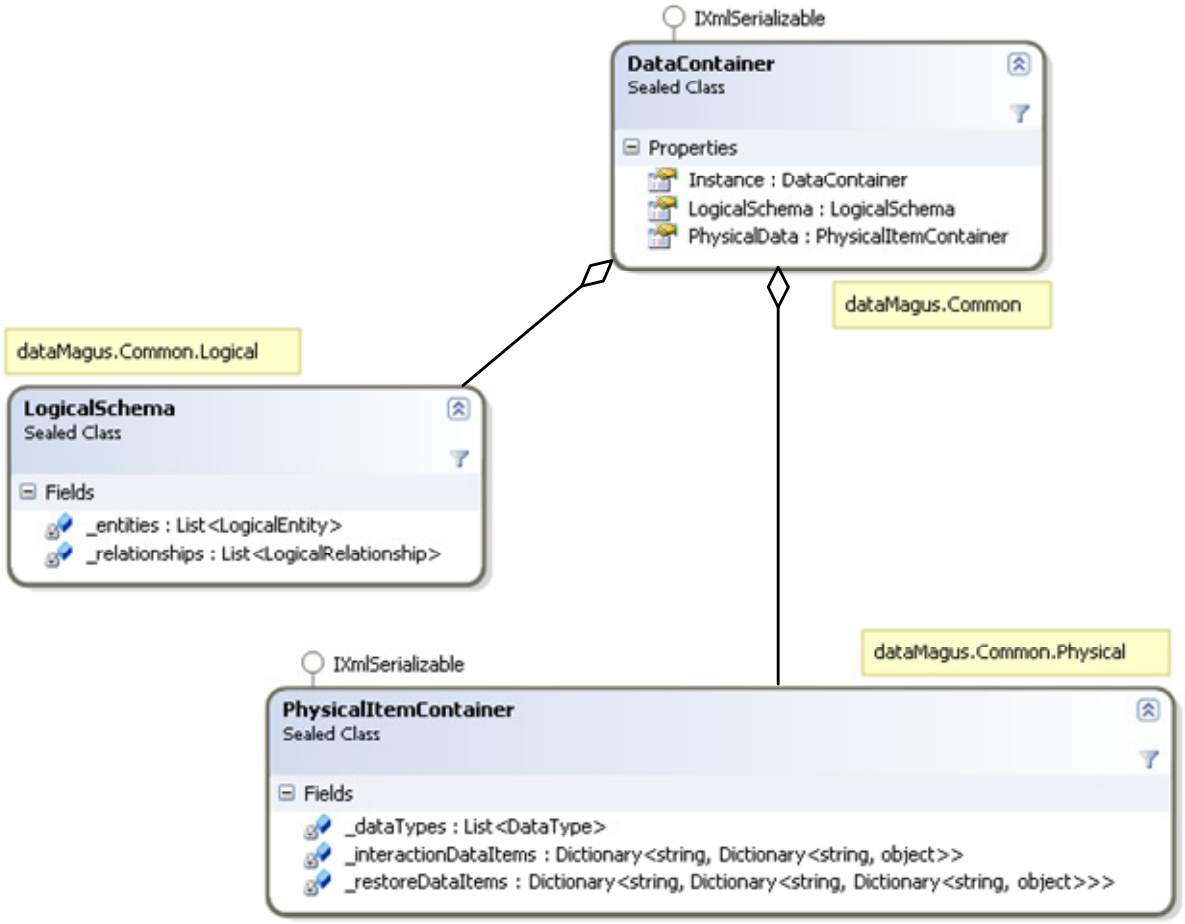


Abbildung 79 – Klassendiagramm *Datencontainer*

Im Modellierungs- und Generierungsprozess im *Kapitel 4.3* ist beschrieben, dass sobald der Benutzer von der logischen Ebene auf irgendeine physikalische Ebene wechselt, eine Abbildung im Sinne der Umsetzung des logischen Schemas auf das gewählte physikalische Schema statt findet. D.h. das physikalische Schema wird immer wieder neu aus dem logischen Schema generiert und die allfällig gemachten Änderungen der physikalischen Ebenen werden restauriert. Diese Änderungen (und nur das Delta) werden ebenfalls in unserem Datencontainer abgelegt. Dazu hält sich der Datencontainer eine Instanz der Klasse *PhysicalItemContainer*.
Der *PhysicalItemContainer* führt eine Liste der Datentypen zum Modell, eine Liste von *InteractionDataItems* und eine Liste von *RestoreDataItems*.
Unter den *InteractionDataItems* werden die **Antworten des Benutzers** zu den verschiedenen logisch modellierten Konstrukten gespeichert. Diese Angaben werden unter dem **Key des betreffenden physikalischen Schemas** und unter **der logischen Id** (die immer eindeutig ist) gespeichert. Da dies sehr schwierig zu beschreiben ist, zeigen wir hier ein Beispiel (nur Fragment), wie es im XML-Format aussieht:

```
<InteractionDataItems>
  <dataMagus.Common.Physical.Relational.PhysicalRelationalSchema>
     <Item
        id="9cd46670-db87-4fac-8e93-5cad3f5f562a"
        type="dataMagus.Common.Physical.Relational.
              PhysicalRelationalSchemaConverter+OneToOneAnswers"
        assembly="dataMagus.Common">
        WithoutLinkTableForeignKeyRight
    </Item>
  </dataMagus.Common.Physical.Relational.PhysicalRelationalSchema>
</InteractionDataItems>
```

Die *RestoreDataItems* führen die Änderungen der physikalischen Modelle ebenfalls in einem Dictionary. Hier wird unter dem **Key des betreffenden physikalischen Schemas**, unter der **physikalischen Id** (die sich immer aus ein oder mehreren logischen Ids, der dem physikalischen Element zugrunde liegenden logischen Elemente zusammensetzt) und dem **betreffenden Propertynamen der geänderte Wert** abgespeichert. Da dies auch hier etwas kompliziert klingt, stellen wir wiederum ein XML-Beispiel (nur Fragment) zur Verfügung:

```
<RestoreDataItems>
  <dataMagus.Common.Physical.Relational.PhysicalRelationalSchema>
    <Item id="226752b0-5ca0-4c7e-95cf-e50544923db4_Id">
      <Property
        name="AllowNulls"
        type="System.Boolean"
        assembly="mscorlib">
        False
      </Property>
    </Item>
  </dataMagus.Common.Physical.Relational.PhysicalRelationalSchema>
</RestoreDataItems>
```

Diese Daten werden bei einer Änderung des physikalischen Elements in diesem Container nachgeführt sowie bei der Erstellung eines Objektes wieder aus dem Container abgeholt. Um diese Daten persistent speichern zu können, müssen die abgelegten Objekte entweder elementare Datentypen[23] sein, das Interface *IConvertible*[24] oder das Interface *IXmlSerializable* implementieren.

[23] Elementare (oder primitive) Typen sind Boolean, Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64, IntPtr, Char, Double und Single [6].
[24] Interface im Namensraum System des .NET Frameworks. Das Interface definiert Methoden für die Konvertierung des Implementierungsverweis- oder Werttyps in einen Typ der Common Language Runtime mit einem entsprechenden Wert [6]

5.2.10 Persistente Daten

Um die Daten persistent zu speichern, führen wir einen Serialisierungsvorgang durch, der die Daten im XML-Format speichert. Es genügt, das logische Modell sowie die physikalischen Änderungen und die Antworten des Benutzers zu speichern. Das physikalische Modell muss nicht gespeichert werden, weil es immer wieder aus dem logischen Modell generiert werden kann.
Wir haben uns für die XML-Serialisierung entschieden, weil ein Speicherformat in XML zeitgemäss und ausserdem das Format ohne spezielle Hilfsprogramme lesbar ist (sofern keine Verschlüsselung angewendet wurde). Würden wir unser Modell rudimentär mit einer Objekt-Serialisierung speichern, könnte man erstens das Schema ohne Maschine nicht mehr lesen und zweitens ist diese Variante nicht versionsunabhängig. Sollten die Klassen ändern, kann ein gespeichertes Objekt einer alten Version der Klasse nicht mehr deserialisiert werden.

Des Weiteren eignet sich das hierarchische XML-Format für unser „baumartiges" Schema optimal. Zusätzlich ergibt sich den Vorteil, dass eine XML-Datei mit einem XSD-Schema auf Gültigkeit geprüft werden könnte.
Natürlich hat das XML-Format auch Nachteile. Die Grösse kann explodieren und das Format sehr schwerfällig werden. Jedoch denken wir, dass wir diese Nachteile vernachlässigen können, da dataMagus einerseits nicht für grosse Modelle ausgelegt ist oder man andererseits auch eine einfache Komprimierung (z.B. GZip) durchführen könnte, welche die Grösse des Formats wiederum verkleinern würde.

Serialisierungsvorgang
Für die XML-Serialisierung stellt das .NET-Framework das Interface *IXmlSerializable* zur Verfügung.

Abbildung 80 – Interface *IXmlSerializable*

Das Interface definiert drei Methoden. Zur Serialisierung gibt es die *ReadXml()*- und die *WriteXml()*-Methode. Diese Methoden nehmen einen *XmlReader*[25] bzw. einen *XmlWriter*[26] auf. Über diese können die Objekte von einem XML-Stream gelesen oder in einen XML-Stream geschrieben werden. Die Methode *GetSchema()* dient zur Validierung des XML-Streams gegen ein Schema. Sie wurden von uns jedoch nicht implementiert.

Wir haben auf den Datenbehältern sowie auf allen logischen Elementen das Interface *IXmlSerializable* implementiert. Zusätzlich müssen die abgelegten *InteractionDataItems* und *RestoreDataItems* auch das Interface implementieren, wenn es sich nicht um primitive Datentypen handelt und wenn das Interface *IConvertible* nicht implementiert ist. Ansonsten können diese Daten nicht persistent gespeichert werden.

Die Generatoren können für ihre generatorspezifischen Daten eine Instanz einer Klasse mit implementiertem *IExtendedData*-Interface zur Verfügung stellen (vgl. *Kapitel 5.2.7 Code-Generatoren*). Dieses Interface ist ebenfalls von *IXmlSerializable* abgeleitet.
Der Serialisierungsprozess wird über den *XmlSerializer* mit dem Typen *Datacontainer* angestossen. Dieser instanziert einen *XmlWriter* und ruft die Methode *WriteXml()* des Datencontainers auf. Der Datencontainer ruft seinerseits auf dem logischen Schema die Methode *WriteXml()* und gibt den *XmlWriter* weiter. Das logische Schema leitet den Aufruf kaskadierend durch alle logischen Elemente hindurch weiter. Wurde es erfolgreich traversiert, wird vom Datencontainer die *WriteXml()*-Methode des *PhysicalItemContainers* aufgerufen. Dieser speichert die Datentypen, die RestoreDataItems und die InteractionDataItems. Implementiert ein abgelegtes Objekt das Interface *IXmlSerializable*, so wird diese *WriteXml()*-Methode aufgerufen. Ansonsten handelt es sich um einen primitiven Datentypen, der mit Hilfe des *IConvertible*-Interfaces serialisiert werden kann. Zuletzt ruft der *PhysicalItemContainer* auf jedem Generator über das Interface *IGenerator* auf dem Property *ExtendendData* die *WriteXml()*-Methode auf. Ein PlugIn-Generator-Entwickler kann diese Methode ausprogrammieren und so seine Daten zum Modell speichern lassen. Natürlich erhält der Generator nicht den ursprünglichen *XmlWriter*, sondern **einen neuen XmlWriter**. Somit kann ein **PlugIn-Entwickler nichts an unserer Struktur ändern**. Sobald der Generator fertig ist, lesen wir seinen XML-Stream in unseren primären *XmlWriter* ein (aber nur, wenn seine Methode fehlerfrei durchlaufen ist).

[25] Klasse im Namensraum System.Xml des .NET Frameworks. Stellt einen Reader dar, der schnellen, nicht zwischengespeicherten Vorwärtszugriff auf XML-Daten bietet [6]
[26] Klasse im Namensraum System.Xml des .NET Frameworks. Stellt einen Writer für die schnelle, vorwärts gerichtete Generierung von Streams oder Dateien mit XML-Daten ohne Zwischenspeicherung dar [6].

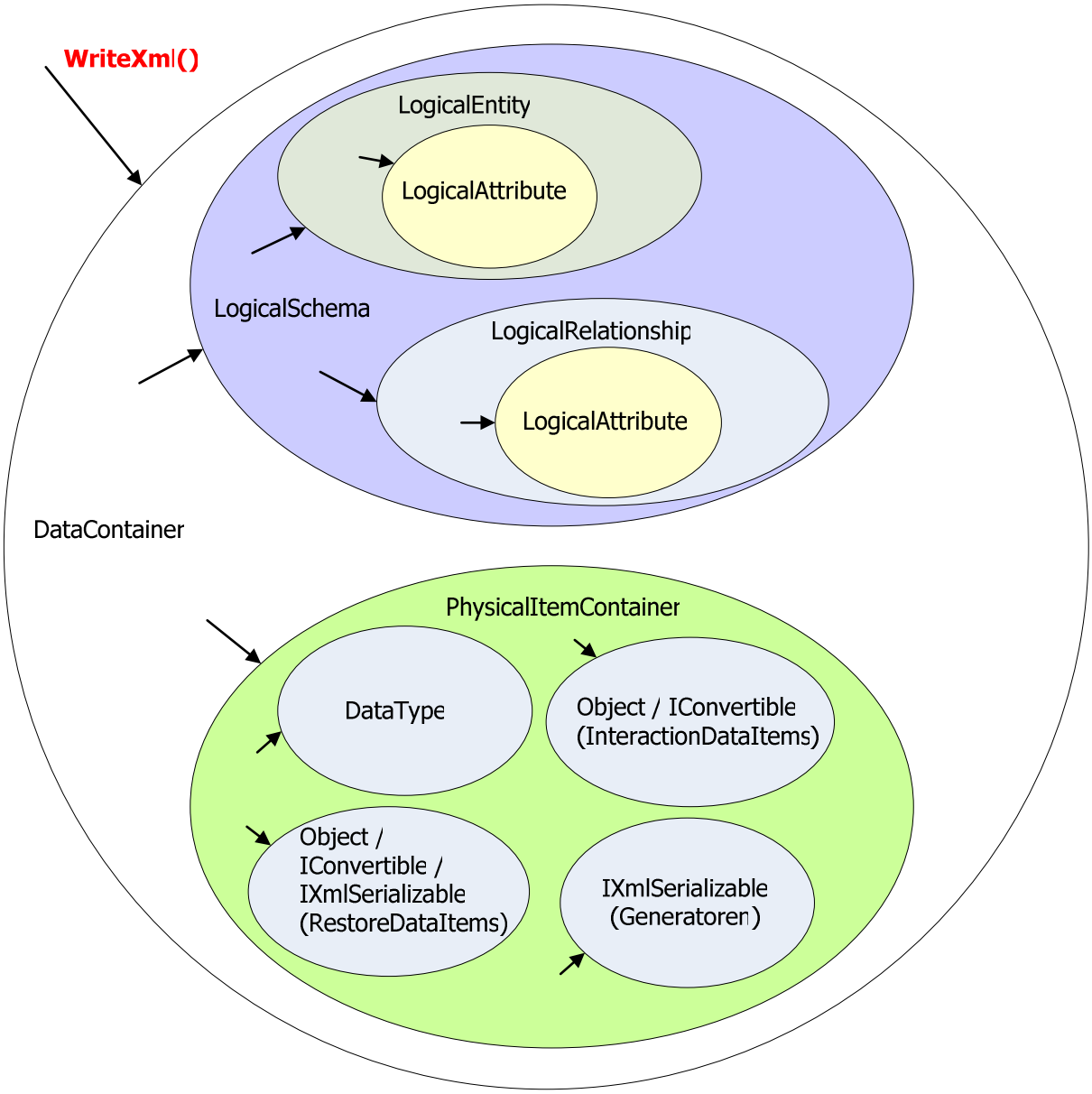


Abbildung 81 – XML-Serialisierungsprozess

Der Deserialisierungsprozess (d.h. das Einlesen einer XML-Datei) erfolgt auf die genau gleiche Weise, jedoch über die Methode *ReadXml()*. Der Generator erhält auch hier nicht unseren *XmlReader*, sondern es wird ihm über die Methode *ReadSubtree()* auf der *XmlReader*-Klasse ein neuer *XmlReader* zur Verfügung gestellt, der zum Lesen des aktuellen Knotens und aller untergeordneten Knoten verwendet werden kann.

Datenformat

Für die Beschreibung des Datenformats haben wir einen pragmatischen Ansatz gewählt und zeigen eine gespeicherte XML-Struktur. Wie oben gerade beschrieben, können viele zusätzliche Informationen der physikalischen Ebenen und der Generatoren hinzugespeichert werden. Wie diese Daten aussehen ist erst zur Laufzeit bekannt.

Wir möchten mit der folgenden Struktur einen Eindruck des Formats vermitteln. Es ist natürlich in der von uns gewählten Dokumentationsart nicht formal beschrieben. Wir erachten dies aber auch nicht als zwingend notwendig.

Im Root-Element *<DataContainerInternal>* befindet sich das logische Schema (*<LogicalSchema>*) mit den Entitäten (*<Entities>*) und Beziehungen (*<Relationships>*). Unter den Beziehungen sind die Elemente *<OneToOneRelationship>*, *<IsARelationship>*, *<OneToManyRelationship>* und *<ManyToManyRelationship>* erlaubt. Die Elemente korrespondieren mit den jeweiligen Klassen. Ausgewählte Properties werden als Attribute der Elemente serialisiert.

Die physikalischen Zusatzdaten liegen im Tag *<PhysicalData>*, auch unter dem Root-Element. Es umfasst die benutzerspezifischen Datentypen im Tag *<DataTypes>* und die gemachten Anpassungen (wie z.B. ein Name) unter den *<RestoreDataItems>*. Die gespeicherten Benutzerinteraktionen befinden sich im Element *<InteractionDataItems>*. Wie die *RestoreDataItems* und die *InteractionDataItems* abgelegt werden, ist bereits im Kapitel *5.2.9 Datencontainer* diskutiert worden.

Anschliessend folgen die persistenen Daten der Generatoren (*<GeneratorDataItems>*), sofern diese den Serialisierungsprozess unterstützen.

```
<?xml version="1.0"?>
<DataContainerInternal>
  <LogicalSchema id="63df69a1-66f0-4b8c-8476-3486b56a9ff0" name="Logical schema">
    <Entities>
      <Entity id="94f6fc16-fe51-4304-8b42-e5f5a165650b" name="Mann" locationX="450"
              locationY="330">
        <Attributes>
          <Attribute id="fce04c5c-8490-4cc7-a94a-67f3db97d68a" name="PersNr"
                     locationX="0" locationY="0" />
        </Attributes>
        <KeyAttributes>
          <Attribute id="fce04c5c-8490-4cc7-a94a-67f3db97d68a" name="PersNr"
                     locationX="0" locationY="0" />
        </KeyAttributes>
      </Entity>
      <Entity id="12cf5015-dd83-4238-a79c-1517f1ca1954" name="Frau" locationX="130"
              locationY="330">
        <Attributes />
        <KeyAttributes />
      </Entity>
    </Entities>
    <Relationships>
      <OneToOneRelationship id="a9c35b06-4f16-48bc-bb6b-b1319e9ffd9a"
                            name="VerheirtatetMit" locationX="290" locationY="330"
                            entity1="94f6fc16-fe51-4304-8b42-e5f5a165650b"
                            entity2="12cf5015-dd83-4238-a79c-1517f1ca1954"
                            rolename1="MannVerheirtatetMit"
                            rolename2="FrauVerheirtatetMit"
                            cardinality1="ZeroOrOne" cardinality2="ZeroOrOne">
        <Attributes>
          <Attribute id="d6a7fd21-9921-4ee5-bf65-1635f13b4f89" name="Heiratsdatum"
                     locationX="0" locationY="0" />
        </Attributes>
      </OneToOneRelationship>
    </Relationships>
  </LogicalSchema>
  <PhysicalData>
    <DataTypes />
    <RestoreDataItems>
      <dataMagus.Common.Physical.Relational.PhysicalRelationalSchema>
        <Item id="94f6fc16-fe51-4304-8b42-e5f5a165650b">
          <Property name="LocationX" type="System.Int32" assembly="mscorlib">640
          </Property>
          <Property name="LocationY" type="System.Int32" assembly="mscorlib">300
          </Property>
        </Item>
      </dataMagus.Common.Physical.Relational.PhysicalRelationalSchema>
      <dataMagus.Common.Physical.Objectoriented.PhysicalObjectorientedSchema>
        <Item id="94f6fc16-fe51-4304-8b42-e5f5a165650b">
          <Property name="LocationX" type="System.Int32" assembly="mscorlib">450
          </Property>
          <Property name="LocationY" type="System.Int32" assembly="mscorlib">330
          </Property>
        </Item>
      </dataMagus.Common.Physical.Objectoriented.PhysicalObjectorientedSchema>
    </RestoreDataItems>
    <InteractionDataItems>
      <dataMagus.Common.Physical.Relational.PhysicalRelationalSchema>
        <Item id="a9c35b06-4f16-48bc-bb6b-b1319e9ffd9a"
              type="dataMagus.Common.Physical.Relational.
                    PhysicalRelationalSchemaConverter+OneToOneAnswers"
              assembly="dataMagus.Common">WithLinkTableWithNaturalKeyLeft
        </Item>
      </dataMagus.Common.Physical.Relational.PhysicalRelationalSchema>
    </InteractionDataItems>
```

```
    <GeneratorDataItems>
      <dataMagus.PlugIns.Generators.Odl.OdlGenerator>
        <GeneratorData inverse="True" key="False" extent="False">
          <DataTypeMappings>
            <Mapping key="String" value="string" />
            <Mapping key="Number" value="short" />
            <Mapping key="Float" value="float" />
            <Mapping key="DateTime" value="Date" />
            <Mapping key="Boolean" value="boolean" />
            <Mapping key="Binary" value="octet" />
            <Mapping key="Unknown" value="unknown" />
          </DataTypeMappings>
        </GeneratorData>
      </dataMagus.PlugIns.Generators.Odl.OdlGenerator>
      <dataMagus.PlugIns.Generators.PostgreSql.PostgreSqlGenerator>
        <GeneratorData forceColumnConstraintForPrimaryKey="True"
                       forceForeignKeyWithCreateTable="True"
                       forceColumnConstraintForForeignKey="True"
                       forceUniqueIndexWithCreateTable="True"
                       forceColumnConstraintForUnique="True">
          <DataTypeMappings>
            <Mapping key="String" value="varchar(100)" />
            <Mapping key="Number" value="integer" />
            <Mapping key="Float" value="decimal" />
            <Mapping key="DateTime" value="timestamp " />
            <Mapping key="Boolean" value="boolean" />
            <Mapping key="Binary" value="bytea" />
            <Mapping key="Unknown" value="unknown" />
          </DataTypeMappings>
          <Relationships />
        </GeneratorData>
      </dataMagus.PlugIns.Generators.PostgreSql.PostgreSqlGenerator>
      <dataMagus.PlugIns.Generators.SqlServer2000.SqlServer2000Generator>
        <GeneratorData forceColumnConstraintForPrimaryKey="True"
                       forceForeignKeyWithCreateTable="True"
                       forceColumnConstraintForForeignKey="True"
                       forceUniqueIndexWithCreateTable="True"
                       forceColumnConstraintForUnique="True">
          <DataTypeMappings>
            <Mapping key="String" value="varchar(100)" />
            <Mapping key="Number" value="int" />
            <Mapping key="Float" value="decimal" />
            <Mapping key="DateTime" value="datetime" />
            <Mapping key="Boolean" value="bit" />
            <Mapping key="Binary" value="binary" />
            <Mapping key="Unknown" value="unknown" />
          </DataTypeMappings>
          <Relationships />
          <Columns />
        </GeneratorData>
      </dataMagus.PlugIns.Generators.SqlServer2000.SqlServer2000Generator>
    </GeneratorDataItems>
  </PhysicalData>
</DataContainerInternal>
```

5.2.11 Physikalische nicht-persistente Daten

Weiter oben wurde beschrieben, dass die Generatoren sich das physikalische Schema auch als XML exportieren lassen können. Jedes physikalische Schema sowie alle physikalischen Elemente implementieren daher das Interface *IXmlGeneratorWriter*.

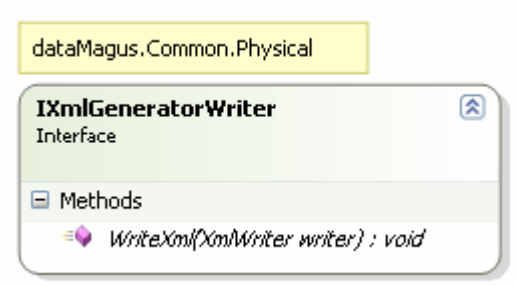


Abbildung 82 – Interface *IXmlGeneratorWriter*

Dieses Interface ist an das .NET-Interface *IXmlSerializable* angelehnt. Es definiert jedoch nur eine *WriteXml()*-Methode. Da es redundant wäre, die physikalischen Schemas persistent zu speichern, haben wir auch nicht das *IXmlSerializable*-Interface implementiert.
Jedoch möchten wir allfälligen Generator-Entwicklern eine Möglichkeit zur Verfügung stellen, das physikalische Schema in ein XML-Format exportieren zu lassen. Dadurch kann er es ggf. besser lesen oder besser verarbeiten. Deshalb unterstützen wir mit der *WriteXml()*-Methode, die auf einem physikalischen Schema aufgerufen werden kann und sich analog zum logischen Schema durch das physikalische Schema hangelt, die Erzeugung eines XML für das physikalische Schema.

6 Implementation der Generatoren-PlugIns

Nachfolgend werden die Generatoren beschrieben, die in der ersten Version ausgeliefert werden. Alle drei Generatoren bauen auf dem gleichen Prinzip auf. Daher erscheinen sie sehr ähnlich. Jedoch konnten die Klassen nicht mehr weiter zusammengefasst werden, da jeder Generator selbständig und in sich abgeschlossen sein muss.

6.1 SQL Server 2000

Dieser Generator erstellt aus dem physikalisch relationalen Schema Quellcode für den Microsoft SQL Server 2000.

Abbildung 83 – Logo MS SQL Server [14]

6.1.1 Spezielles zur Codegenerierung

Der Generator liefert im Prinzip vollfunktionstüchtigen SQL-Code. Er achtet jedoch nicht auf eine Abhängigkeit zwischen den einzelnen Tabellen. Es kann also sein, dass bei der Endanwendung auf einem SQL Server 2000 die Reihenfolge der Statements geändert werden muss, damit die Statements erfolgreich durchlaufen. Eine solche Abhängigkeit besteht immer bei einem Fremdschlüssel-Constraint. Die Tabelle, die den Primärschlüssel für die Fremdschlüsselbeziehung stellt, muss immer früher kreiert werden als die Tabelle mit dem Fremdschlüssel-Constraint, wenn dieser direkt mit dem CREATE TABLE-Statement erstellt werden soll.
Eine weitere unmögliche Konstellation ergibt sich z.B. aus einer Doppelbeziehung zwischen zwei gleichen Entitäten, bei denen der oder die Studierende einmal die Variante wählt, bei der der Fremdschlüssel in der einen Tabelle erwähnt werden soll und einmal in der anderen.

Abbildung 84 – Konstrukt mit gegenseitiger Referenzierung

Es entsteht ein Konstrukt, bei dem die eine Tabelle die andere referenziert und umgekehrt. Solche Sachen können nie mit einem einzigen CREATE-TABLE-Statement auf die Datenbank geladen werden. Minimum eine der Fremdschlüsselbeziehung muss nachträglich mit einem ALTER TABLE-Statement erstellt werden.
Darum stellt der Generator eine Möglichkeit zur Verfügung, die Fremdschlüssel-Constraints mit ALTER TABLE-Statements generieren zu lassen.

6.1.2 Funktionen

Folgende Funktionen werden vom Generator unterstützt:

- **Datentyp-Mappping**
  Der Generator definiert folgendes Standardmapping:
  - String = varchar(100)
  - Number = int
  - Float = decimal
  - DateTime = datetime
  - Boolean = bit
  - Binary = binary
  - Unknown = unknown

- **Column-spezifische Zusatzdaten**
  Der Generator lässt auf den physikalischen relationalen Spalten eine IDENTITY-Zusatzangabe zu. Das Identity-Flag kann jedoch nur auf einer einzigen Spalte pro Tabelle gesetzt werden. Ausserdem muss diese Spalte folgende Kritierien erfüllen:
  - Sie darf keine Null-Werte aufnehmen können.
  - Sie muss den Datentypen Number oder Float definiert haben.
  - Die Spalte darf nicht Teil eines Fremdschlüssels sein.
  - Es darf keine andere Spalte in derselben Tabelle als Identity markiert worden sein.

  Zur Identity-Angabe kann der *Seed* und das *Increment* angegeben werden. Der *Seed* definiert den Wert der für den ersten Datensatz verwendet wird. Mit *Increment* wird der inkrementierte Wert für den folgenden Datensatz bestimmt.

- **Relationship-spezifische Zusatzdaten**
  Zu einer Beziehung können Angaben zur referentiellen Integrität angegeben werden. Der SQL Server unterstützt:
  - ON DELETE CASCADE
  - ON DELETE NO ACTION (Default-Einstellung, wird nicht generiert)
  - ON UPDATE CASCADE
  - ON UPDATE NO ACTION (Default-Einstellung, wird nicht generiert)

  Wobei folgende Definitionen gelten:
  - **NO ACTION**:
    - Erzeugt einen Fehler, bei einer Fremdschlüssel-Constraint-Verletzung, wobei die Operation durchgeführt ist und die Fremdschlüsselbeziehung erst am Ende durch das Datenbanksystem verifiziert wird.

  - **CASCADE**:
    - Propagiert die Änderungen.

- **Generierung der Tabellen mit Primarkey-, Fremdschlüssel- und Unique-Constraints**
  Zu jeder Tabelle wird ein CREATE TABLE-Statement erzeugt. Es kann gewählt werden, ob der Primärschlüssel als Column-Constraint[27] oder als Table-Constraint[28] aufgeführt werden soll. Besteht der Primärschlüssel aus mehr als einem Feld, wird er automatisch als Table-Constraint aufgeführt. Die direkte PRIMARY KEY-Angabe ist nur für ein Feld erlaubt.

[27] Spaltenbeschränkung
[28] Tabellenbeschränkung

Bei Fremdschlüsseln kann gewählt werden, ob sie direkt als Column-Constraint, als Table-Constraints oder separat in einem ALTER TABLE-Statement angegeben werden sollen. Referenziert ein Fremdschlüssel mehr als ein Feld und ist die Option *Column Constraint* angewählt, so werden sie automatisch in einem Table-Constraint aufgeführt.

Bei Unique-Constraints kann ebenfalls gewählt werden, ob sie als Column-Constraint, als Table-Constraints oder separat in einem CREATE UNIQUE INDEX-Statement angegeben werden sollen. Ist die Option *Colum Constraint* angewählt, so wird das Schlüsselwort UNIQUE direkt zur Spaltendefinition geschrieben. Sind zu einem Unique-Constraint mehrere Spalten definiert worden, wird immer ein Table-Constraint generiert, ausser es sei die Option CREATE UNIQUE INDEX-Statement angegeben worden.

- **Generierung von Indexen**
  Falls gewünscht wird, die Indexe in einem separaten Statement generieren zu lassen, werden alle Indexe mit einem CREATE UNIQUE INDEX-Statement umgesetzt.

- **Alternative Generierung der Fremdschlüssel**
  Falls gewünscht wird, die Fremdschlüssel-Constraints in einem separaten Statement generieren zu lassen, werden diese mit einem ALTER TABLE-Statement umgesetzt.

6.1.3 Syntax

Der Code wird anhand der Definition des SQL Servers 2000s generiert. Nachfolgend sind die relevanten Statements formal beschrieben aufgelistet [8]. Alles, was fett markiert ist, kann durch den Generator umgesetzt werden.

CREATE-TABLE-Statement

```
CREATE TABLE
    [ database_name.[ owner ] . | owner. ] table_name
    ( { < column_definition >
        | column_name AS computed_column_expression
        | < table_constraint > ::= [ CONSTRAINT constraint_name ] }
            | [ { PRIMARY KEY | UNIQUE } [ ,...n ]
    )
[ ON { filegroup | DEFAULT } ]
[ TEXTIMAGE_ON { filegroup | DEFAULT } ]


< column_definition > ::= { column_name data_type }
    [ COLLATE < collation_name > ]
    [ [ DEFAULT constant_expression ]
        | [ IDENTITY [ ( seed , increment ) [ NOT FOR REPLICATION ] ] ]
    ]
    [ ROWGUIDCOL]
    [ < column_constraint > ] [ ...n ]

< column_constraint > ::= [ CONSTRAINT constraint_name ]
    { [ NULL | NOT NULL ]
        | [ { PRIMARY KEY | UNIQUE }
            [ CLUSTERED | NONCLUSTERED ]
            [ WITH FILLFACTOR = fillfactor ]
            [ON {filegroup | DEFAULT} ] ]
        ]
        | [ [ FOREIGN KEY ]
            REFERENCES ref_table [ ( ref_column ) ]
            [ ON DELETE { CASCADE | NO ACTION } ]
            [ ON UPDATE { CASCADE | NO ACTION } ]
            [ NOT FOR REPLICATION ]
        ]
        | CHECK [ NOT FOR REPLICATION ]
        ( logical_expression )
    }
< table_constraint > ::= [ CONSTRAINT constraint_name ]
    { [ { PRIMARY KEY | UNIQUE }
        [ CLUSTERED | NONCLUSTERED ]
        { ( column [ ASC | DESC ] [ ,...n ] ) }
        [ WITH FILLFACTOR = fillfactor ]
        [ ON { filegroup | DEFAULT } ]
    ]
    | FOREIGN KEY
        [ ( column [ ,...n ] ) ]
        REFERENCES ref_table [ ( ref_column [ ,...n ] ) ]
        [ ON DELETE { CASCADE | NO ACTION } ]
        [ ON UPDATE { CASCADE | NO ACTION } ]
        [ NOT FOR REPLICATION ]
    | CHECK [ NOT FOR REPLICATION ]
        ( search_conditions )
    }
```

ALTER TABLE-Statement

```
ALTER TABLE [ database_name . [ schema_name ] . | schema_name . ]
table_name
{
    ALTER COLUMN column_name
    {
        [ type_schema_name. ] type_name [ ( { precision [ , scale ]
            | max | xml_schema_collection } ) ]
        [ COLLATE collation_name ]
        [ NULL | NOT NULL ]
    | {ADD | DROP } { ROWGUIDCOL | PERSISTED }
    | DROP NOT FOR REPLICATION
    }
    | [ WITH { CHECK | NOCHECK } ] ADD
    {
        <column_definition>
      | <computed_column_definition>
      | <table_constraint>
    } [ ,...n ]
    | DROP
    {
        [ CONSTRAINT ] constraint_name
        [ WITH ( <drop_clustered_constraint_option> [ ,...n ] ) ]
        | COLUMN column_name
    } [ ,...n ]
    | [ WITH { CHECK | NOCHECK } ] { CHECK | NOCHECK } CONSTRAINT
        { ALL | constraint_name [ ,...n ] }
    | { ENABLE | DISABLE } TRIGGER
        { ALL | trigger_name [ ,...n ] }
    | SWITCH [ PARTITION source_partition_number_expression ]
        TO [ schema_name. ] target_table
        [ PARTITION target_partition_number_expression ]
}
[ ; ]

<drop_clustered_constraint_option> ::=
    {
        MAXDOP = max_degree_of_parallelism
      | ONLINE = {ON | OFF }
      | MOVE TO { partition_scheme_name ( column_name ) | filegroup
          | "default"}
    }
```

CREATE UNIQUE INDEX-Statement

```
CREATE [ UNIQUE ] [ CLUSTERED | NONCLUSTERED ] INDEX index_name
    ON <object> ( column [ ASC | DESC ] [ ,...n ] )
    [ INCLUDE ( column_name [ ,...n ] ) ]
    [ WITH ( <relational_index_option> [ ,...n ] ) ]
    [ ON { partition_scheme_name ( column_name )
         | filegroup_name
         | default
         }
    ]
[ ; ]

<object> ::=
{
    [ database_name. [ schema_name ] . | schema_name. ]
        table_or_view_name
}
```

6.1.4 Implementation

Da der SQL Server 2000 eine relationale Datenbank ist, muss unsere Generatorklasse *SqlServer2000Generator* vom Interface *IRelationalGenerator* abgeleitet sein.
Da wir diverse Optionen für den Generator definiert haben, sowie Zusatzdaten für Spalten und Beziehungen, führt der Generator intern eine Instanz der Klasse *GeneratorData*, welche das Interface *IExtendedData* implementiert. Da dieses Interface von *IXmlSerializable* abgeleitet ist, kann der Generator seine Daten mit den Methoden *WriteXml()* und *ReadXml()* auch persistent speichern bzw. wieder einlesen lassen. GeneratorData führt intern ein Dictionary für die Spalten-Zusatzdaten (*ColumnGeneratorData*) und Beziehungs-Zusatzdaten (*RelationshipGeneratorData*). Diese zwei Klassen nehmen die Einstellungen wie oben definiert auf. Damit eine Instanz der *ColumnGeneratorData* resp. *RelationshipGeneratorData* eindeutig einem Element des physikalisch relationalen Schemas zugewiesen werden kann, wird die Instanz im Dictionary zu der Id des betreffenden physikalischen Elements gespeichert.
Um Zusatzdaten erfassen zu können, müssen in diesem Fall die Methoden *GetExtendendColumnDataControl()* und *GetExtendedRelationshipDataControl()* implementiert werden. Wie im *Kapitel 5.2.7 Code-Generatoren* beschrieben, werden diese Methoden von dataMagus an bestimmten Stellen im Programm aufgerufen (z.B. wenn die Spaltendaten generell geändert werden) und es wird ihnen eine Kopie des entsprechenden physikalischen Elementes (*PhysicalRelationalColumn* oder *PhysicalRelationalRelationship*) übergeben. Die beiden Methoden liefern jeweils eine Instanz der Klasse *GeneratorControl* zurück. Um dem Benutzer nun eine optimale Eingabemöglichkeit geben zu können, ist für die Spalten-Zusatzdaten das Control *ExtendedColumnControl* und für die Beziehungs-Zusatzdaten das Control *ExtendedColumnControl* erstellt worden. Beide Controls leiten von der Framework-Basisklasse *GeneratorControl* ab. Diese Controls empfangen nun das entsprechende physikalische Element und verwalten zu seiner eindeutigen Id nun die generatorspezifischen Daten. Sie interagieren also mit dem *GeneratorData*.
Ganz im selben Stil gibt es das implementierte Control *DataTypeMappingControl* für die Datentyp-Mappings und das Control *OptionsControl* für die Angabe der Optionen zum Generierungsprozess.

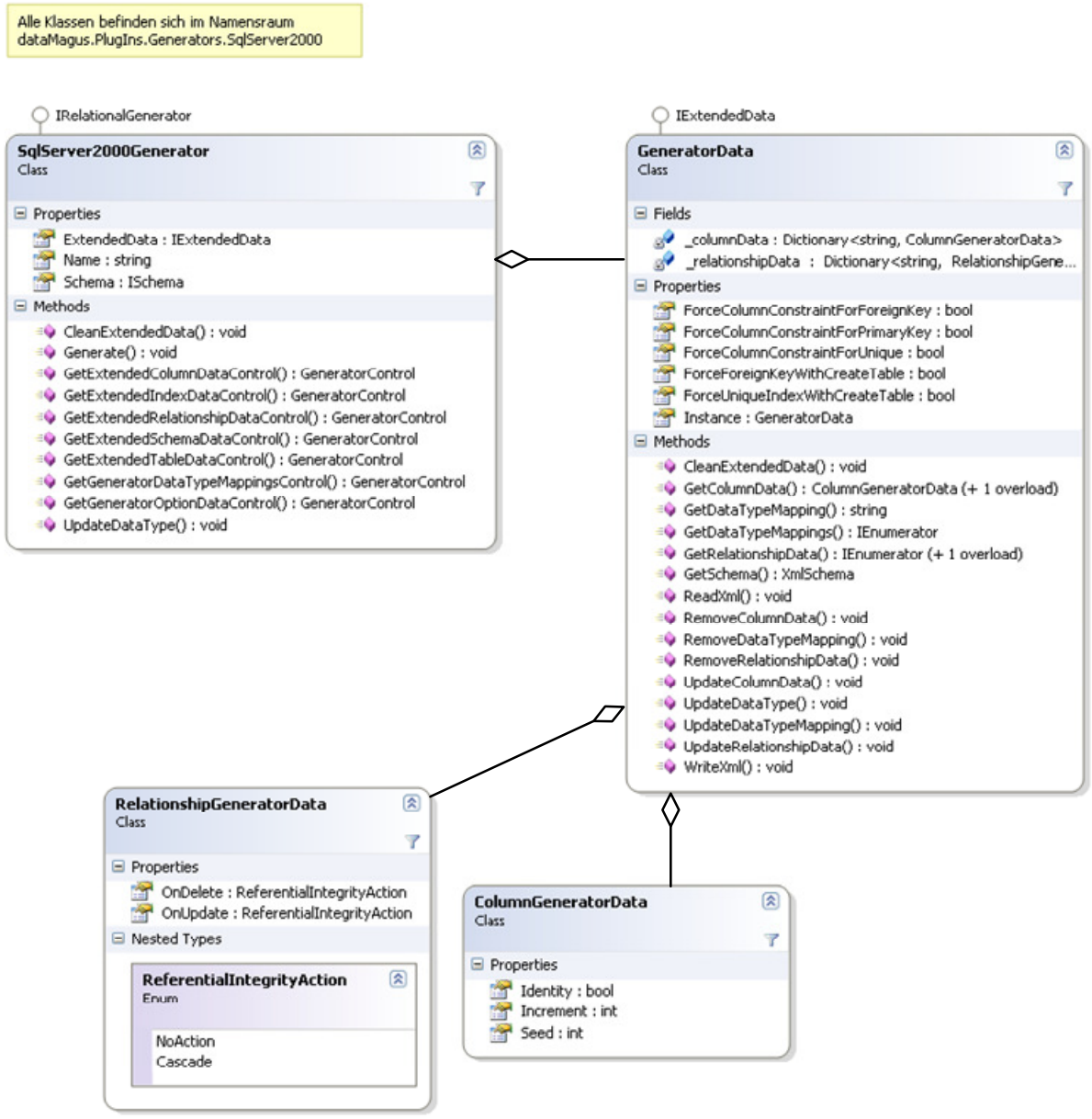


Abbildung 85 – Klassendiagramm zum SQL Server 2000 Generator

Der Generierungsprozess selber wird durch die Methode *Generate()* ausgeführt, welche wiederum von dataMagus aufgerufen wird. Der SQL Server 2000 Generator erstellt zunächst ein XML-Document aus dem physikalischen Schema, indem er auf dem Schema selber die Methode *WriteXml()* aufruft. Diese liefert ihm einen *XmlWriter* zurück mit dem Schema im XML-Format. Danach wird der Quellcode mit einer XSLT-Transformation erstellt, die mit dem soeben erstellten XML-Document zusammengeführt wird. Der Quellcode wird in den der Methode *Generate()* übergebenen Stream geschrieben. Dieser Stream wird dann von dataMagus dem Benutzer angezeigt.
Wir haben den Quellcode mit einer XSLT-Transformation erstellt, weil diese sehr flexibel konfigurierbar sind, effizient in der Ausführung und keine Kompilierung zur Designzeit benötigen. Zudem sind sie ideal im Zusammenhang mit XML-Dateien resp. mit hierarchisch aufgebauten Strukturen.

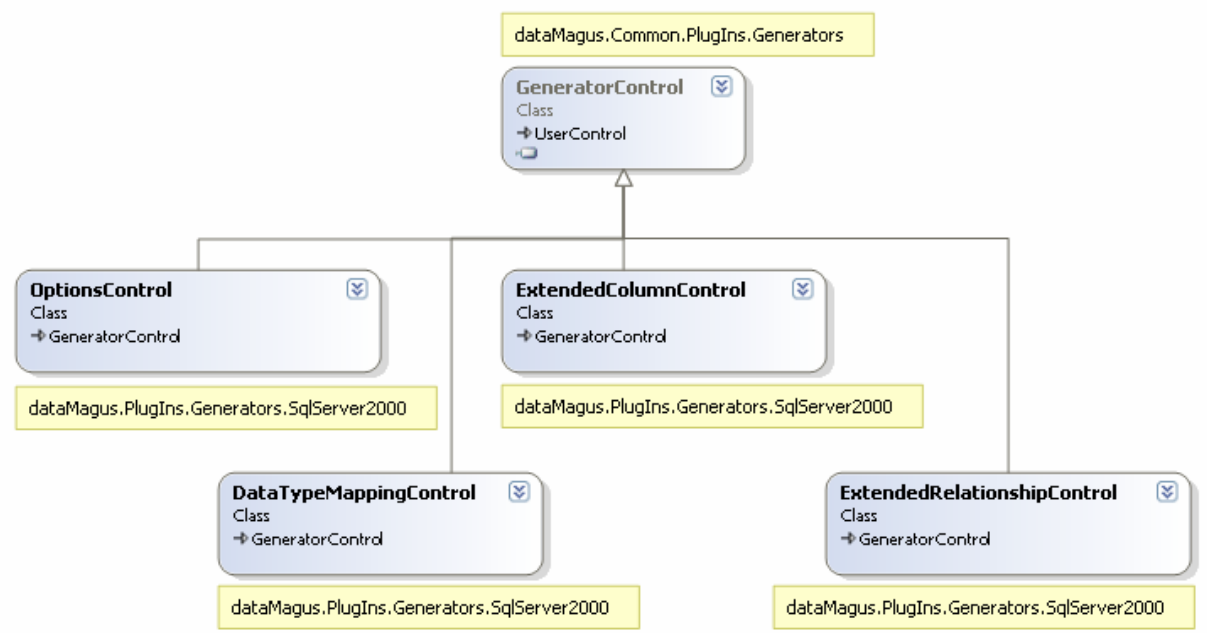


Abbildung 86 – Erweiterte Controls des SQL Server 2000 Generators

Im Nachfolgenden werden die wichtigsten, implementierten Klassen und ihre Properties und Methoden kurz in tabellarischer Form beschrieben:

Klasse (K), Property (P) oder Methode (M)		Beschreibung
K	**SqlServer2000Generator**	
P	ExtendedData	Liefert eine Instanz der Klasse *GeneratorData* zurück, welche das Interface *IExtendedData* implementiert.
P	Name	Name des Generators
P	Schema	Das Schema, von welchem der Quellcode erstellt werden soll
M	CleanExtendedData	Überprüft die gespeicherten Generator-Daten zu den Spalten und Beziehungen auf ihre Gültigkeit. Sie erhält das physikalische Schema, welches zur Prüfung benutzt werden kann.
M	Generate	Generiert den Quellcode mit einer XSLT-Transformation. Das Schema muss vorgängig gesetzt worden sein.
M	GetExtendedColumnDataControl	Liefert eine Instanz der Klasse *ExtendedColumnDataControl* zurück. Mit diesem Control können die Zusatzdaten zu den Spalten erfasst werden.
M	GetExtendedIndexDataControl	Wird nicht benötigt und liefert daher eine Null-Referenz zurück.
M	GetExtendedRelationshipData Control	Liefert eine Instanz der Klasse *ExtendedRelationshipDataControl* zurück. Mit diesem Control können die Zusatzdaten zu den Beziehungen erfasst werden.
M	GetExtendedSchemaDataControl	Wird nicht benötigt und liefert eine Null-Referenz zurück.

Klasse (K), Property (P) oder Methode (M)		Beschreibung
M	GetExtendedTableDataControl	Wird nicht benötigt und liefert eine Null-Referenz zurück.
M	GetGeneratorDataTypeMappingsControl	Liefert eine Instanz der Klasse *ExtendendDataTypeMappingsControl* zurück. Mit diesem Control können Datentyp-Mappings erfasst werden.
M	GetGeneratorOptionDataControl	Liefert eine Instanz der Klasse *OptionsControl* zurück. Das Control dient zur Aufnahme der Einstellungen für den Generierungsprozess.
M	UpdateDataType	Diese Methode wird von dataMagus aufgerufen, wenn ein benutzerspezifischer Datentyp hinzugefügt, geändert oder gelöscht worden ist. Der Generator hat seine Daten intern aktuell zu führen.
K	**GeneratorData**	
P	ForceColumnContraintFor PrimaryKey	Flag, ob der Primarykey-Constraint mit einem Column-Constraint umgesetzt werden soll. Falls nicht, wird er als Table-Constraint umgesetzt. (*)
P	ForceForeignKeyWithCreate Table	Flag, ob die Foreignkey-Constraints direkt mit dem CREATE TABLE-Statement erstellt werden sollen. Falls nicht, werden sie mit dem ALTER TABLE-Statement generiert.
P	ForceColumnContraintForForeign Key	Falls *ForceForeignKeyWithCreateTable* wahr ist, kann angegeben werden, ob die Foreignkey-Constraints als Column-Constraints umgesetzt werden sollen. Falls nicht, werden sie als Table-Constraints generiert. (*)
P	ForceUniqueIndexWithCreate Table	Flag, ob die Unique-Constraints direkt mit dem CREATE TABLE-Statement erstellt werden sollen. Falls nicht, werden sie mit dem CREATE UNIQUE INDEX-Statement generiert.
P	ForceColumnConstraintFor Unique	Falls *ForceUniqueIndexWithCreateTable* wahr ist, kann angegeben werden, ob die Unique-Constraints als Column-Constraints umgesetzt werden sollen. Falls nicht, werden sie als Table-Constraints generiert. (*)
P	Instanz	Liefert eine Instanz der Klasse *GeneratorData* zurück, welche als Singleton implementiert ist.
M	CleanExtendedData	Wird vom *SqlServer2000Generator* aufgerufen und dient zum Aufräumen der generatorspezifischen Daten.
M	GetColumnData	Liefert zu einer physikalischen Id die gespeicherten, generatorspezifischen Spaltendaten.
M	GetDataTypeMapping	Liefert zu einem Datentypen das gespeicherte Mapping.
M	GetDataTypeMappings	Liefert alle gespeicherten Datentyp-Mappings.
M	GetRelationshipData	Liefert zu einer physikalischen Id die gespeicherten, generatorspezifischen Beziehungsdaten.
M	GetSchema	Wird vom Interface *IXmlSerializable* definiert, aber nicht implementiert.

Klasse (K), Property (P) oder Methode (M)		Beschreibung
M	ReadXml	Wird vom Interface *IXmlSerializable* definiert. Liest die persistent gespeicherten generatorspezifischen Daten ein.
M	RemoveColumnData	Löscht die gespeicherten Spaltendaten zu einer physikalischen Id.
M	RemoveDataTypeMapping	Löscht das Mapping zu einem Datentypen.
M	RemoveRelationshipData	Löscht die gespeicherten Beziehungsdaten zu einer physikalischen Id.
M	UpdateColumnData	Aktualisiert die gespeicherten Spaltendaten zu einer physikalischen Id.
M	UpdateDataType	Aktualisiert die intern gespeicherten Datentypen, falls ein neuer benutzerspezifischer Datentyp erstellt, gelöscht oder aktualisiert worden ist.
M	UpdateDataTypeMapping	Aktualisiert das Mapping zu einem Datentypen.
M	WriteXml	Wird vom Interface *IXmlSerializable* definiert und dient zur persistenten Speicherung der generatorspezifischen Daten.

Tabelle 62 – Klassenbeschreibung des SQL Server 2000 Generators

* Natürlich werden nur Column-Constraints erstellt, wenn es semantisch korrekt ist.

6.1.5 Screenshots

Folgende Screenshots sollen die Implementation veranschaulichen.

Datentyp-Mapping
Mit dem *DataTypeMappingControl* können Datentyp-Mappings vorgenommen werden. Diese werden in der Spalte *TargetDataType* eingegeben.

	PhysicalDataType	TargetDataType
▶	String	varchar(100)
	Number	int
	Float	decimal
	DateTime	datetime
	Boolean	bit
	Binary	binary
	Unknown	unknown

Abbildung 87 – SQL Server 2000: Datentyp-Mapping

Column-spezifische Zusatzdaten
Mit dem *ExtendedColumnDataControl* können die Zusatzdaten zu einer Spalte gesetzt werden.

Abbildung 88 – SQL Server 2000: Column-spezifische Zusatzdaten

Relationship-spezifische Zusatzdaten

Beziehungs-spezifische Zusatzdaten werden mit dem Control *ExtendedRelationshipDataControl* vorgenommen.

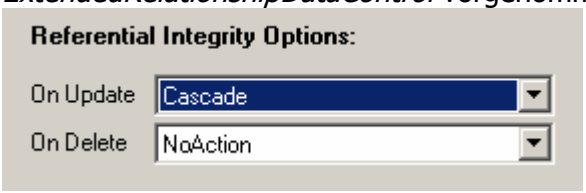


Abbildung 89 – SQL Server 2000: Relationship-spezifische Zusatzdaten

Generatoroptionen

Optionen zur Generierung können mit Hilfe des Controls *GeneratorOptions* eingestellt werden.

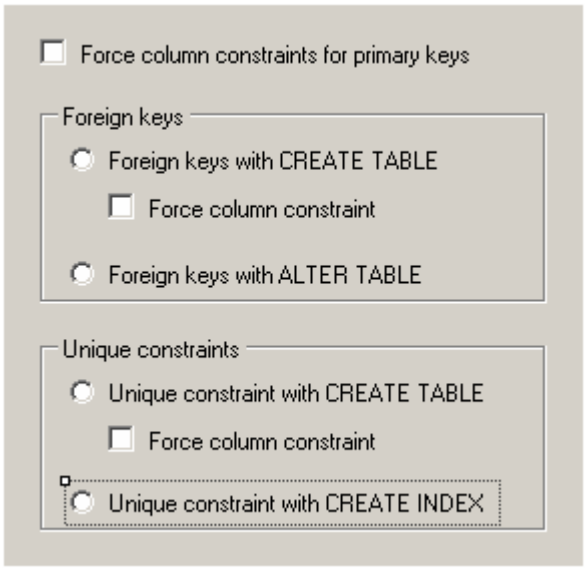


Abbildung 90 – SQL Server 2000: Generator-Optionen

6.1.6 Beispiel

Zur Veranschaulichung des Quellcodes nehmen wir wieder unser Heiratsbeispiel aus *Kapitel 4.8.5.4 Umsetzung der 1:1-Beziehung* zur Hand:

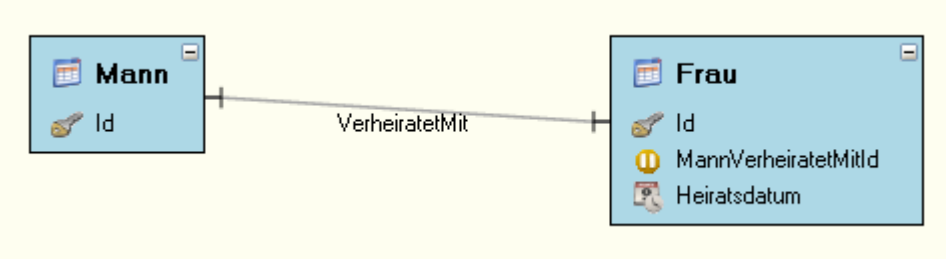


Abbildung 91 – Heiratsbeispiel (1:1-Beziehung) auf der physikalisch relationalen Ebene

Mit den Optionen
- ForceColumnContraintForForeignKey = wahr,
- ForceForeignKeyWithCreateTable = falsch und
- ForceUniqueIndexWithCreateTable = falsch

sieht der Quellcode wie folgt aus:

```
CREATE TABLE Mann
(
  Id int NOT NULL PRIMARY KEY
);

CREATE TABLE Frau
(
  Id int NOT NULL PRIMARY KEY,
  MannVerheiratetMitId int NULL,
  Heiratsdatum datetime NULL
);

CREATE UNIQUE INDEX FrauUniqueIdx ON Frau
(
  MannVerheiratetMitId
);

ALTER TABLE Frau ADD FOREIGN KEY (MannVerheiratetMitId) REFERENCES Mann;
```

6.2 PostgreSQL

Dieser Generator erstellt aus dem physikalisch relationalen Schema Quellcode für die PostgreSQL-Datenbank. Diese Datenbank gehört eigentlich in die Kategorie der objektrelationalen Modelle. Solche Datenbanken halten sich irgendwo zwischen den relationalen und objektorientierten Datenbanken auf und sind schwer zu fassen. Nach Absprache der Dozentin generieren wir für die PostgreSQL-Datenbank nur relationalen Code heraus.

Abbildung 92 – Logo PosgreSQL [15]

6.2.1 Spezielles zur Codegenerierung

Wie eingangs bereits erwähnt, werden nur relationale Code-Konstrukte generiert. Auch dieser Generator liefert vollfunktionstüchtigen SQL-Code (getestet mit der Version 8.2). Bei der Erstellung des Quellcodes gelten die gleichen Regeln und Einschränkungen wie beim SQL Server 2000.

6.2.2 Funktionen

Folgende Funktionen werden vom Generator unterstützt:

- **Datentyp-Mappping**
  Der Generator definiert folgendes Standardmapping:
  - String = varchar(100)
  - Number = integer
  - DateTime = timestamp
  - Boolean = boolean
  - Binary = bytea
  - Unknown = unknown

- **Relationship-spezifische Zusatzdaten**
  Zu einer Beziehung können Angaben zur referentiellen Integrität angegeben werden. Die PostgreSQL-Datenbank unterstützt:
  - ON DELETE NO ACTION (Default-Einstellung, wird nicht generiert)
  - ON DELETE RESTRICT
  - ON DELETE CASCADE
  - ON DELETE SET NULL
  - ON DELETE SET DEFAULT
  - ON UPDATE NO ACTION (Default-Einstellung, wird nicht generiert)
  - ON UPDATE RESTRICT
  - ON UPDATE CASCADE
  - ON UPDATE SET NULL
  - ON UPDATE SET DEFAULT

Wobei folgende Definitionen gelten:

- **NO ACTION**:
  - Erzeugt einen Fehler, bei einer Fremdschlüssel-Constraint-Verletzung, wobei die Operation durchgeführt ist und die Fremdschlüsselbeziehung erst am Ende durch das Datenbanksystem verifiziert wird.

- **RESTRICT**:
  - Erzeugt denselben Fehler wie NO ACTION mit dem Unterschied, dass die Fremdschlüsselbeziehung vor der Operation verifiziert wird.

- **CASCADE**:
  - Propagiert die Änderungen.

- **SET NULL**:
  - Verweise werden auf NULL gesetzt.

- **SET DEFAULT**:
  - Verweise werden auf die Defaultwerte gesetzt.

6.2.3 Syntax

Der Code wird anhand der Definition der PostgreSQL-Datenbank generiert. Nachfolgend sind die relevanten Statements formal beschrieben aufgelistet [7]. Alles, was fett markiert ist, kann durch den Generator umgesetzt werden.

CREATE TABLE-Statement

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } ] TABLE table_name ( [
  { column_name data_type [ DEFAULT default_expr ] [ column_constraint [
... ] ]
    | table_constraint
    | LIKE parent_table [ { INCLUDING | EXCLUDING } { DEFAULTS |
CONSTRAINTS } ] ... }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS
]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace ]

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  CHECK ( expression ) |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL |
MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE
]

and table_constraint is:

[ CONSTRAINT constraint_name ]
{ UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  CHECK ( expression ) |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [,
... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON
UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE
]

index_parameters in UNIQUE and PRIMARY KEY constraints are:

[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace ]
```

ALTER TABLE-Statement

```
ALTER TABLE [ ONLY ] name [ * ]
    action [, ... ]
ALTER TABLE [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column TO new_column
ALTER TABLE name
    RENAME TO new_name
ALTER TABLE name
    SET SCHEMA new_schema
```

where *action* is one of:

```
    ADD [ COLUMN ] column type [ column_constraint [ ... ] ]
    DROP [ COLUMN ] column [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column TYPE type [ USING expression ]
    ALTER [ COLUMN ] column SET DEFAULT expression
    ALTER [ COLUMN ] column DROP DEFAULT
    ALTER [ COLUMN ] column { SET | DROP } NOT NULL
    ALTER [ COLUMN ] column SET STATISTICS integer
    ALTER [ COLUMN ] column SET STORAGE { PLAIN | EXTERNAL | EXTENDED |
MAIN }
    ADD table_constraint
    DROP CONSTRAINT constraint_name [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET WITHOUT OIDS
    SET ( storage_parameter = value [, ... ] )
    RESET ( storage_parameter [, ... ] )
    INHERIT parent_table
    NO INHERIT parent_table
    OWNER TO new_owner
  SET TABLESPACE new_tablespace
```

CREATE UNIQUE INDEX-Statement

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] name ON table [ USING method ]
    ( { column | ( expression ) } [ opclass ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace ]
    [ WHERE predicate ]
```

6.2.4 Implementation

Da wir für die PostgreSQL-Datenbank relationalen Quellcode generieren möchten, muss die Generatorklasse *PostgreSqlGenerator* ebenfalls vom Interface *IRelationalGenerator* abgeleitet sein. Da die Implementation der des SQL Server 2000 Generators sehr ähnlich ist, verzichten wir hier auf eine ausführliche Beschreibung. Es läuft nach demselben Prinzip wie der SQL Server 2000 Generator, mit der Ausnahme, dass keine Spalten-spezifischen Zusatzdaten gespeichert werden.
Der Generierungsprozess erstellt wiederum ein XML-Document aus dem physikalischen Schema und generiert dieses wieder mit Hilfe einer XSLT-Transformation.

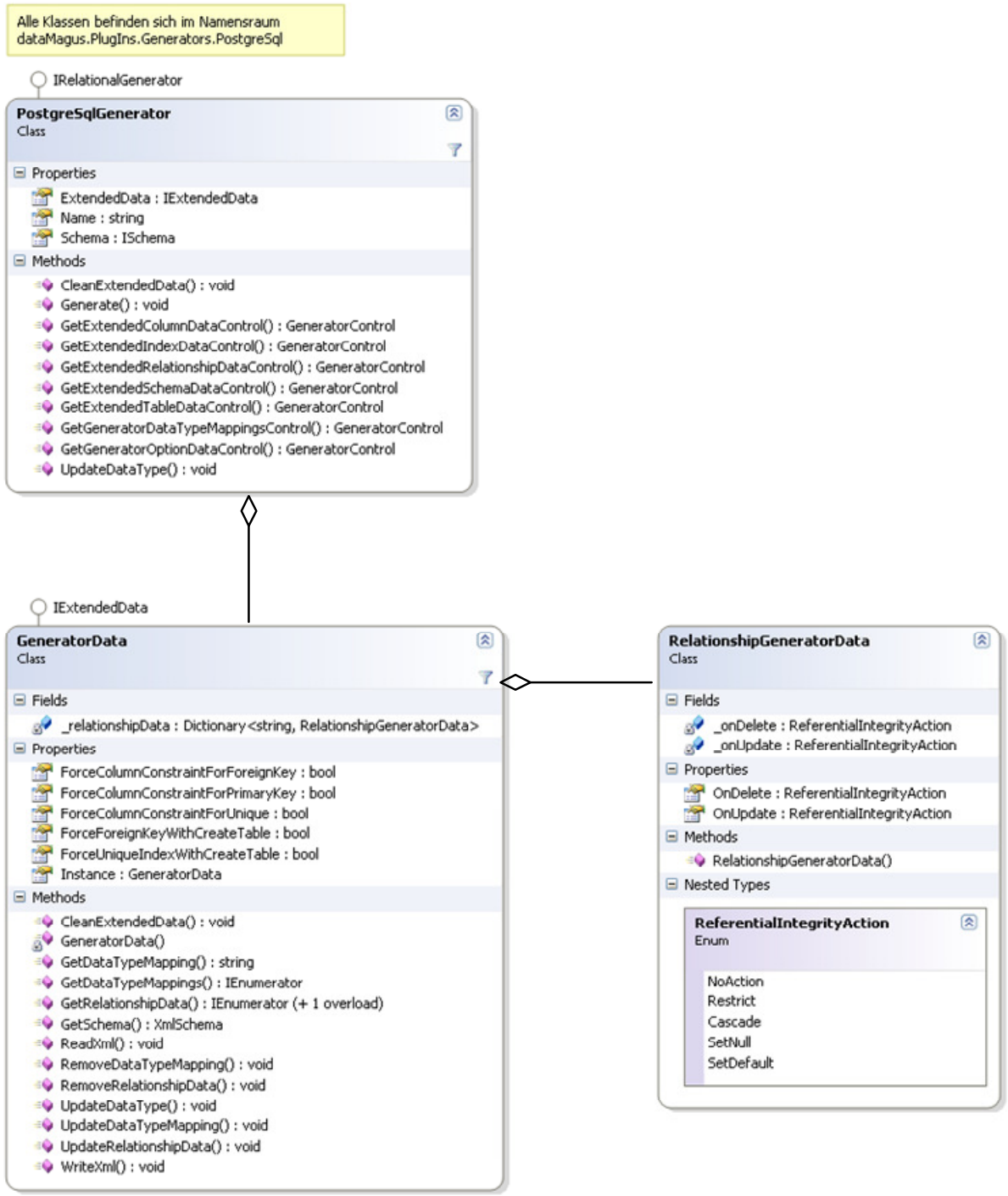


Abbildung 93 – Klassendiagramm zum PostgreSQL Generator

Die implementierten Controls sind im folgenden Klassendiagramm ersichtlich:

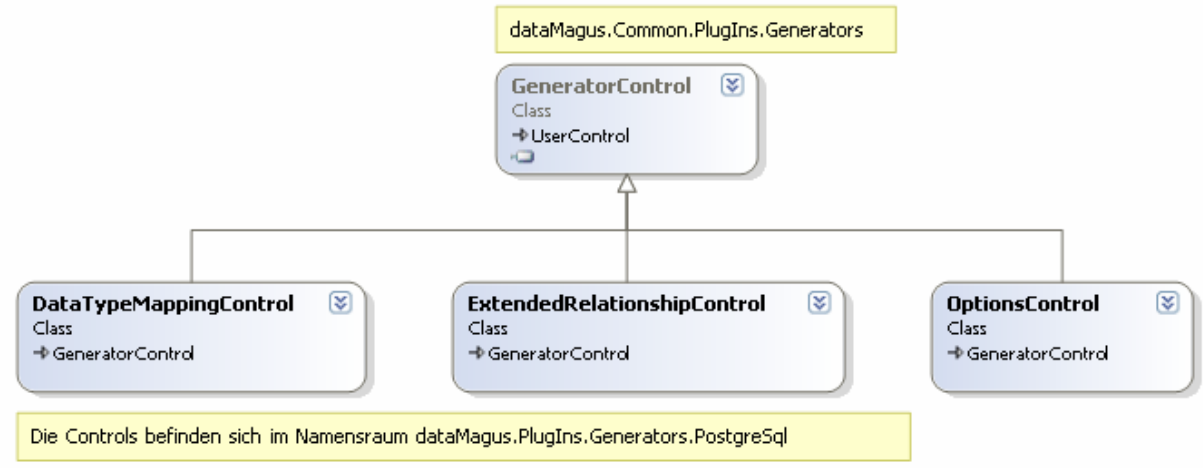


Abbildung 94 – Erweiterte Controls des PostgreSQL Generators

Im Nachfolgenden werden die wichtigsten, implementierten Klassen und ihre Properties und Methoden kurz in tabellarischer Form beschrieben:

Klasse (K), Property (P) oder Methode (M)		Beschreibung
K	**PostgreSqlGenerator**	
P	ExtendedData	Liefert eine Instanz der Klasse *GeneratorData* zurück, welche das Interface *IExtendedData* implementiert.
P	Name	Name des Generators.
P	Schema	Das Schema, von welchem der Quellcode erstellt werden soll.
M	CleanExtendedData	Überprüft die gespeicherten Generator-Daten zu den Beziehungen auf ihre Gültigkeit. Sie erhält das physikalische Schema, welches zur Prüfung benutzt werden kann.
M	Generate	Generiert den Quellcode mit einer XSLT-Transformation. Das Schema muss vorgängig gesetzt worden sein.
M	GetExtendedColumnDataControl	Wird nicht benötigt und liefert daher eine Null-Referenz zurück.
M	GetExtendedIndexDataControl	Wird nicht benötigt und liefert daher eine Null-Referenz zurück.
M	GetExtendedRelationshipData Control	Liefert eine Instanz der Klasse *ExtendedRelationshipDataControl* zurück. Mit diesem Control können die Zusatzdaten zu den Beziehungen erfasst werden.
M	GetExtendedSchemaDataControl	Wird nicht benötigt und liefert eine Null-Referenz zurück.
M	GetExtendedTableDataControl	Wird nicht benötigt und liefert eine Null-Referenz zurück.

Klasse (K), Property (P) oder Methode (M)		Beschreibung
M	GetGeneratorDataTypeMappingsControl	Liefert eine Instanz der Klasse *ExtendendDataTypeMappingsControl* zurück. Mit diesem Control können Datentyp-Mappings erfasst werden.
M	GetGeneratorOptionDataControl	Liefert eine Instanz der Klasse *OptionsControl* zurück. Das Control dient zur Aufnahme der Einstellungen für den Generierungsprozess.
M	UpdateDataType	Diese Methode wird von dataMagus aufgerufen, wenn ein benutzerspezifischer Datentyp hinzugefügt, geändert oder gelöscht worden ist. Der Generator hat seine Daten intern aktuell zu führen.
K	**GeneratorData**	
P	ForceColumnContraintFor PrimaryKey	Flag, ob der Primarykey-Constraint mit einem Column-Constraint umgesetzt werden soll. Falls nicht, wird er als Table-Constraint umgesetzt. (*)
P	ForceForeignKeyWithCreate Table	Flag, ob die Foreignkey-Constraints direkt mit dem CREATE TABLE-Statement erstellt werden sollen. Falls nicht, werden sie mit dem ALTER TABLE-Statement generiert.
P	ForceColumnContraintForForeign Key	Falls *ForceForeignKeyWithCreateTable* wahr ist, kann angegeben werden, ob die Foreignkey-Constraints als Column-Constraints umgesetzt werden sollen. Falls nicht, werden sie als Table-Constraints generiert. (*)
P	ForceUniqueIndexWithCreate Table	Flag, ob die Unique-Constraints direkt mit dem CREATE TABLE-Statement erstellt werden sollen. Falls nicht, werden sie mit dem CREATE UNIQUE INDEX-Statement generiert.
P	ForceColumnConstraintFor Unique	Falls *ForceUniqueIndexWithCreateTable* wahr ist, kann angegeben werden, ob die Unique-Constraints als Column-Constraints umgesetzt werden sollen. Falls nicht, werden sie als Table-Constraints generiert. (*)
P	Instanz	Liefert eine Instanz der Klasse *GeneratorData* zurück, welche als Singleton implementiert ist.
M	CleanExtendedData	Wird vom *PostgreSqlGenerator* aufgerufen und dient zum Aufräumen der generatorspezifischen Daten.
M	GetDataTypeMapping	Liefert zu einem Datentypen das gespeicherte Mapping.
M	GetDataTypeMappings	Liefert alle gespeicherten Datentyp-Mappings.
M	GetRelationshipData	Liefert zu einer physikalischen Id die gespeicherten, generatorspezifischen Beziehungsdaten.
M	GetSchema	Wird vom Interface *IXmlSerializable* definiert, aber nicht implementiert.
M	ReadXml	Wird vom Interface *IXmlSerializable* definiert. Liest die persistent gespeicherten generatorspezifischen Daten ein.
M	RemoveDataTypeMapping	Löscht das Mapping zu einem Datentypen.

Klasse (K), Property (P) oder Methode (M)		Beschreibung
M	RemoveRelationshipData	Löscht die gespeicherten Beziehungsdaten zu einer physikalischen Id.
M	UpdateColumnData	Aktualisiert die gespeicherten Spaltendaten zu einer physikalischen Id.
M	UpdateDataType	Aktualisiert die intern gespeicherten Datentypen, falls ein neuer benutzerspezifischer Datentyp erstellt, gelöscht oder aktualisiert worden ist.
M	UpdateDataTypeMapping	Aktualisiert das Mapping zu einem Datentypen.
M	WriteXml	Wird vom Interface *IXmlSerializable* definiert und dient zur persistenten Speicherung der generatorspezifischen Daten.

Tabelle 63 – Klassenbeschreibung des PostgreSQL Generators

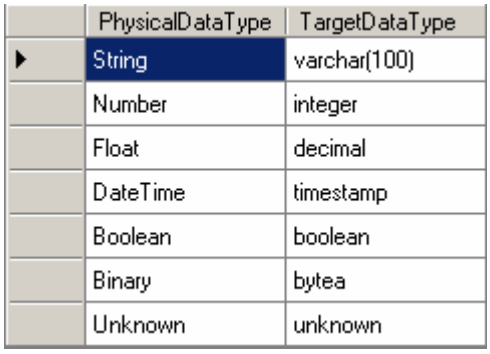
* Natürlich werden nur Column-Constraints erstellt, wenn es semantisch korrekt ist.

6.2.5 Screenshots

Folgende Screenshots visualisieren die Implementation.

Datentyp-Mapping

Mit dem *DataTypeMappingControl* können Datentyp-Mappings vorgenommen werden. Diese werden in der Spalte *TargetDataType* eingegeben.



	PhysicalDataType	TargetDataType
▶	String	varchar(100)
	Number	integer
	Float	decimal
	DateTime	timestamp
	Boolean	boolean
	Binary	bytea
	Unknown	unknown

Abbildung 95 – PostgreSQL: Datentyp-Mapping

Relationship-spezifische Zusatzdaten

Mit dem *ExtendedRelationshipDataControl* können die Zusatzdaten zu einer Beziehung gesetzt werden.

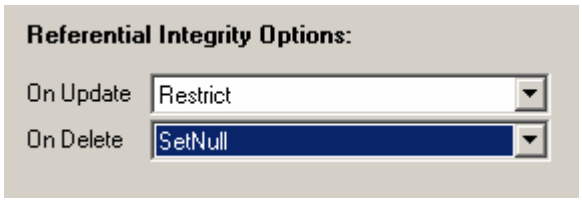


Abbildung 96 – PostgreSQL: Relationship-spezifische Zusatzdaten

Generatoroptionen

Optionen zur Generierung können mit Hilfe des Controls *GeneratorOptions* eingestellt werden.

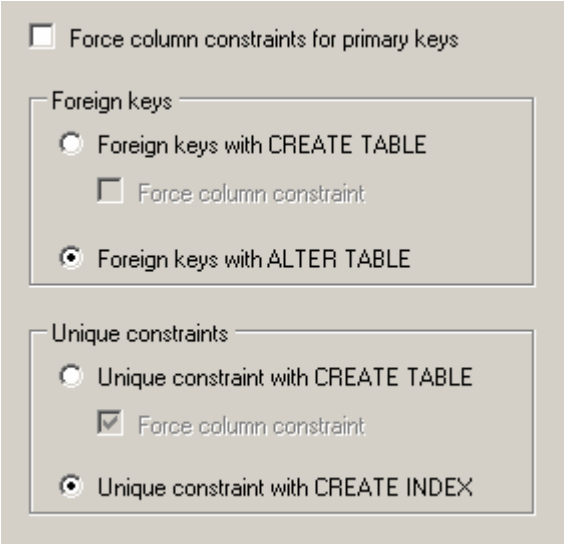


Abbildung 97 – PostgreSQL: Generatoroptionen

6.2.6 Beispiel

Auch hier zeigen wir den Quellcode für das Heiratsbeispiel auf:

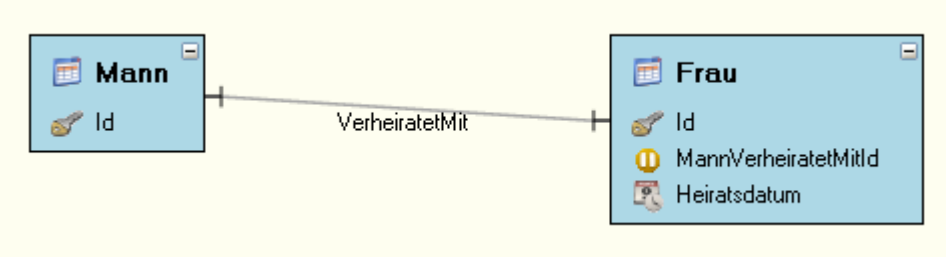


Abbildung 98 – Heiratsbeispiel (1:1-Beziehung) auf der physikalisch relationalen Ebene

Mit den Optionen
- ForceColumnContraintForForeignKey = wahr,
- ForceForeignKeyWithCreateTable = falsch und
- ForceUniqueIndexWithCreateTable = falsch

sieht der Quellcode wie folgt aus:

```
CREATE TABLE Mann
(
  Id integer NOT NULL PRIMARY KEY
);

CREATE TABLE Frau
(
  Id integer NOT NULL PRIMARY KEY,
  MannVerheiratetMitId integer NULL,
  Heiratsdatum timestamp  NULL
);

CREATE UNIQUE INDEX FrauUniqueIdx ON Frau
(
  MannVerheiratetMitId
);
ALTER TABLE Frau ADD FOREIGN KEY (MannVerheiratetMitId)
  REFERENCES Mann (Id)
```

6.3 ODL

Dieser Generator erstellt aus dem physikalisch objektorientierten Schema Quellcode nach dem ODMG-Standard in ODL.

Abbildung 99 – Logo der ODMG [17]

6.3.1 Spezielles zur Codegenerierung

Der Generator liefert funktionstüchtigen ODL-Code. Der ODMG-Standard mit seiner deklarativen Anfragesprache OQL (Object Query Language) und der implementationsunabhängigen Spezifikationssprache ODL (Object Definition Language) hat sich als de-facto Standard durchgesetzt.

6.3.2 Funktionen

Folgende Funktionen werden vom Generator unterstützt:

- **Datentyp-Mapping**
  Der Generator definiert folgendes Standardmapping:
  - String = string
  - Number = short
  - Float = float
  - DateTime = Date
  - Boolean = boolean
  - Binary = octet
  - Unknown = unknown

- **Generierung von Klassen, Properties und Assoziationen**
  Für jede visuelle Klasse wird eine Klassendefinition erzeugt. Es kann gewählt werden, ob inverse Beziehung generiert werden sollen. Dies ist zwar Standard in ODL, aber wir haben uns aus didaktischen Gründen dazu entschlossen, diese Option einzubauen. So kann den Studierenden die Vorteile einer inversen Beziehung[29] näher gebracht werden. Ausserdem kann angegeben werden, ob eine Extension[30] (mit dem Schlüsselwort *extent*) und/oder eine Schlüsselangabe[31] (mit dem Schlüsselwort *key* oder *keys*) generiert werden soll.

[29] Das inverse-Konstrukt wurde im ODMG-Objektmodell integriert, um Inkonsistenzen von Beziehungen systematisch auszuschliessen ([2] auf Seite 364).
[30] Eine Extension ist die Menge aller Instanzen eines Objekttyps. Sie kann für Anfragen der Art „Suche alle Objekte eines Typs, die eine bestimmte Bedingung erfüllen" verwendet werden ([2] auf Seite 369).
[31] Man kann zu einem Objekttyp auch Schlüssel definieren, deren Eindeutigkeit innerhalb der Extension gewährleistet wird. Diese Schlüsseldefinition werden jedoch nur als Integritätsbedingung verwendet und nicht zur Referenzierung von Objekten wie im relationalen Modell.

6.3.3 Syntax

Der Code wird anhand der ODMG-Definition generiert. Nachfolgend sind die relevanten Codekonstrukte formal beschrieben aufgelistet [3]. Alles, was fett markiert ist, kann durch den Generator umgesetzt werden.

Klassen-Deklaration

```
<class>                     ::= <class_dcl> | <class_forward_dcl>
<class_dcl>                 ::= <class_header> { <interface_body> }
<class_forward_dcl>         :: class <identifier>
<class_header>              ::= class<identifier>
                                [extends <scopedName>]
                                [<inheritance_spec>]
                                [<type_property_list>]

<inheritance_spec>          ::= <scoped_name> [,<inheritance_spec>]
<type_property_list>        ::= ([<extent_spec>][<key_spec>])
<extent_spec>               ::= extent <string>
<key_spec>                  ::= key[s] <key_list>
<key_list>                  ::= <key> | <key>, <key_list>
<key>                       ::= <property_name> |(<property_list>)

<property_list>             ::= <property_name>
                                | <property_name>, <property_list>
<property_name>             ::= <identifier>
                                | :: <identifier>
                                | <scoped_name> :: <identifier>
```

Property-Deklaration

```
<attr_dcl>                  ::= [readonly] attribute <domain_type>
                                <attribute_name> [<fixed_arry_size>]
<attribute_name>            ::= <identifier>
<domain_type>               ::= <simple_type_spec>
                                |<struct_type>
                                |<enum_type>
```

Association-Deklaration

```
<rel_dcl>                   ::= relationship
                                 <target_of_path> <identifier>
                                inverse <inverse_traversal_path>
<target_of_path>            ::= <identifier>
                                 |<coll_spec> < <identifier> >
<inverse_traversal_path>    ::= <identifier> :: <identifier>
```

6.3.4 Implementation

Da der ODL Generator auf der objektorientierten Ebene arbeitet, muss die Generatorklasse *OdlGenerator* vom Interface *IObjectorientedGenerator* abgeleitet sein.
Für die Verwaltung des Datentypmappings sowie der generatorspezifischen Optionen führt der Generator intern eine Instanz der Klasse *GeneratorData*, welche das Interface *IExtendedData* implementiert, welches von *IXmlSerializable* abgeleitet ist. Der Generator kann wiederum seine Daten mit den Methoden *WriteXml()* und *ReadXml()* persistent speichern bzw. wieder einlesen. Ansonsten verfügt der Generator über keine Zusatzdaten.

Für die Aufnahme des Datentyp-Mappings sowie der Optionen für den Generationsprozess gibt es ein implementiertes Control *DataTypeMapppingControl* und ein Control *OptionsControl*. Beide Controls leiten wie gewohnt von *GeneratorControl* ab.

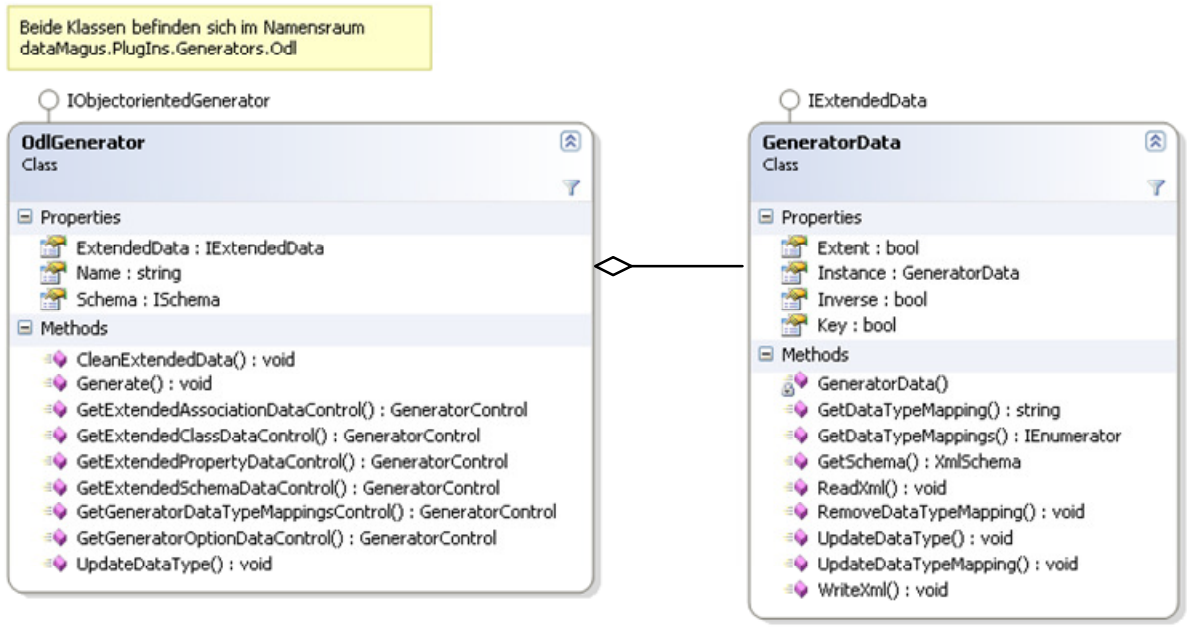


Abbildung 100 – Klassendiagramm zum ODL Generator

Der Generierungsprozess wird wie gewohnt durch die Methode *Generate()* ausgeführt. Auch hier erstellt der OdlGenerator zuerst ein XML-Document aus dem physikalischen Schema und führt anschliessend eine XSLT-Transformation durch. Der Quellcode wird in den übergebenen Stream geschrieben, welcher von dataMagus dem Benutzer angezeigt wird.

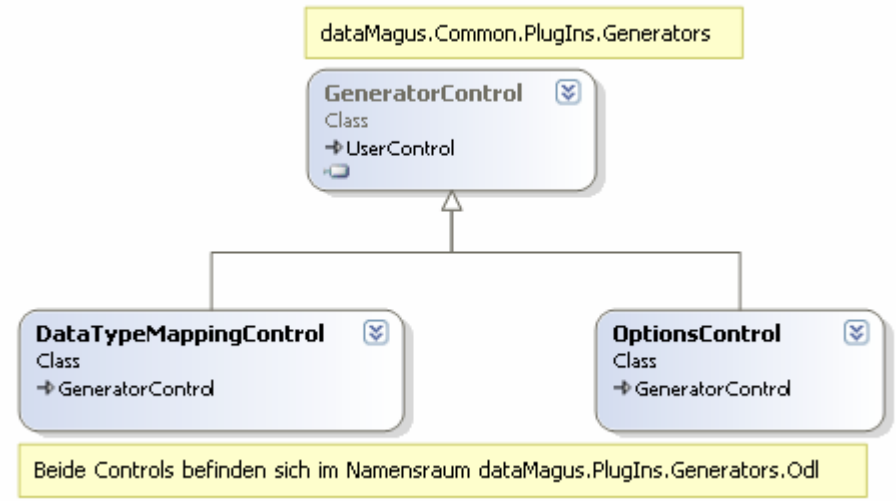


Abbildung 101 – Erweiterte Controls des ODL Generators

Im Nachfolgenden werden die wichtigsten, implementierten Klassen und ihre Properties und Methoden kurz in tabellarischer Form beschrieben:

Klasse (K), Property (P) oder Methode (M)		Beschreibung
K	**OdlGenerator**	
P	ExtendendData	Liefert eine Instanz der Klasse *GeneratorData* zurück, welche das Interface *IExtendedData* implementiert.
P	Name	Name des Generators.
P	Schema	Das Schema, von welchem der Quellcode erstellt werden soll.
M	CleanExtendedData	Wird nicht benötigt. Leere Methode.
M	Generate	Generiert den Quellcode mit einer XSLT-Transformation. Das Schema muss vorgängig gesetzt worden sein.
M	GetExtendedAssociation DataControl	Wird nicht benötigt und liefert eine Null-Referenz zurück.
M	GetExtendedClassDataControl	Wird nicht benötigt und liefert eine Null-Referenz zurück.
M	GetExtendedPropertyData Control	Wird nicht benötigt und liefert eine Null-Referenz zurück.
M	GetExtendedSchemaDataControl	Wird nicht benötigt und liefert eine Null-Referenz zurück.
M	GetExtendedTableDataControl	Wird nicht benötigt und liefert eine Null-Referenz zurück.
M	GetGeneratorDataTypeMappings Control	Liefert eine Instanz der Klasse *ExtendendDataTypeMappingsControl* zurück. Mit diesem Control können Datentyp-Mappings erfasst werden.
M	GetGeneratorOptionDataControl	Liefert eine Instanz der Klasse *OptionsControl* zurück. Das Control dient zur Aufnahme der Einstellungen für den Generierungsprozess.
M	UpdateDataType	Diese Methode wird von dataMagus aufgerufen, wenn ein benutzerspezifischer Datentyp hinzugefügt, geändert oder gelöscht worden ist. Der Generator hat seine Daten intern aktuell zu führen.
K	**GeneratorData**	
P	Extent	Flag, ob eine Extension generiert werden soll
P	Instance	Liefert eine Instanz der Klasse *GeneratorData* zurück, welche als Singleton implementiert ist.
P	Inverse	Flag, ob inverse Beziehungsangaben generiert werden sollen.
P	Key	Flag, ob Schlüsselangaben generiert werden sollen.
M	GetDataTypeMapping	Liefert zu einem Datentypen das gespeicherte Mapping.
M	GetDataTypeMappings	Liefert alle gespeicherten Datentyp-Mappings.
M	GetSchema	Wird vom Interface *IXmlSerializable* definiert, aber nicht implementiert.

Klasse (K), Property (P) oder Methode (M)		Beschreibung
M	ReadXml	Wird vom Interface *IXmlSerializable* definiert. Liest die persistent gespeicherten generatorspezifischen Daten ein.
M	RemoveDataTypeMapping	Löscht das Mapping zu einem Datentypen.
M	UpdateDataType	Aktualisiert die intern gespeicherten Datentypen, falls ein neuer benutzerspezifischer Datentyp erstellt, gelöscht oder aktualisiert worden ist.
M	UpdateDataTypeMapping	Aktualisiert das Mapping zu einem Datentypen.
M	WriteXml	Wird vom Interface *IXmlSerializable* definiert und dient zur persistenten Speicherung der generatorspezifischen Daten.

Tabelle 64 – Klassenbeschreibung des ODL Generators

6.3.5 Screenshots

Folgende Screenshots visualisieren die Implementation:

Datentyp-Mapping
Mit dem *DataTypeMappingControl* können Datentyp-Mappings vorgenommen werden. Diese werden in der Spalte *TargetDataType* eingegeben.

	PhysicalDataType	TargetDataType
▶	String	string
	Number	short
	Float	float
	DateTime	Date
	Boolean	boolean
	Binary	octet
	Unknown	unknown

Abbildung 102 – ODL: Datentyp-Mapping

Generatoroptionen
Optionen zur Generierung können mit Hilfe des Controls *GeneratorOptions* eingestellt werden.

Abbildung 103 – ODL: Generatoroptionen

6.3.6 Beispiel

Das Heiratsbeispiel auf der physikalisch objektorientierten Ebene:

Abbildung 104 – Heiratsbeispiel (1:1-Beziehung) auf der physikalisch objektorientierten Ebene

Mit den Optionen
- Inverse = wahr,
- Extent = false,
- Key = false

Sieht der Quellcode wie folgt aus:

```
class Mann
{
  relationship VerheiratetMit MannMannVerheiratetMit inverse
    VerheiratetMit::VerheiratetMitMannVerheiratetMit;
}

class Frau
{
  relationship VerheiratetMit FrauVerheiratetMitFrau inverse
    VerheiratetMit::VerheiratetMitVerheiratetMitFrau;
}

class VerheiratetMit
{
  attribute Date Heiratsdatum;
  relationship Mann VerheiratetMitMannVerheiratetMit inverse
    Mann::MannMannVerheiratetMit;
  relationship Frau VerheiratetMitVerheiratetMitFrau inverse
    Frau::FrauVerheiratetMitFrau;
}
```

7 Implementation der Notationen-PlugIns

In den folgenden Kapiteln werden die verschiedenen Notationen beschrieben, die mit der ersten Version der Software mitgeliefert werden. Sie weisen alle mehr oder weniger die gleiche Struktur auf. Die Notationen müssen als PlugIn eigenständig und unabhängig sein. Die Verbindung zu dataMagus hat das PlugIn über die Businesskomponente *Common* (vgl. das Schichtenmodell).
Die Beschreibungen sind bewusst kurz gehalten worden, da die einzelnen Notationen auf viel Businesslogik zurückgreifen können, die bereits im *Kapitel 5.2.8 Notationen* ausführlich beschrieben worden sind.

7.1 Chen-Notation

Die Chen-Notation ist eine Darstellungsform der logischen Ebene. Sie existiert auf keiner physikalischen Ebene. Sie ermöglicht, die jeweiligen Datenstrukturen möglichst nahe an der realen Welt abzubilden.
Chen sieht vor, dass Attribute auf Entitätsmengen jeweils als Ellipse um die Entitätsmenge herum angeordnet werden. Da diese Darstellung viel Platz beanspruchen kann, haben wir dies optimiert, indem unsere Implementation die Attribute jeweils direkt unter der Entitätsmenge zeichnet. Sie sind auch nicht einzeln bewegbar. Sie verhalten sich gleich, wie die Attribute auf den noch folgenden Notationen Crowfoot und ODLG. Wichtig zu erwähnen ist, dass bei dieser leicht abgeänderten Form die Darstellungsregeln von Chen nicht verletzt werden.

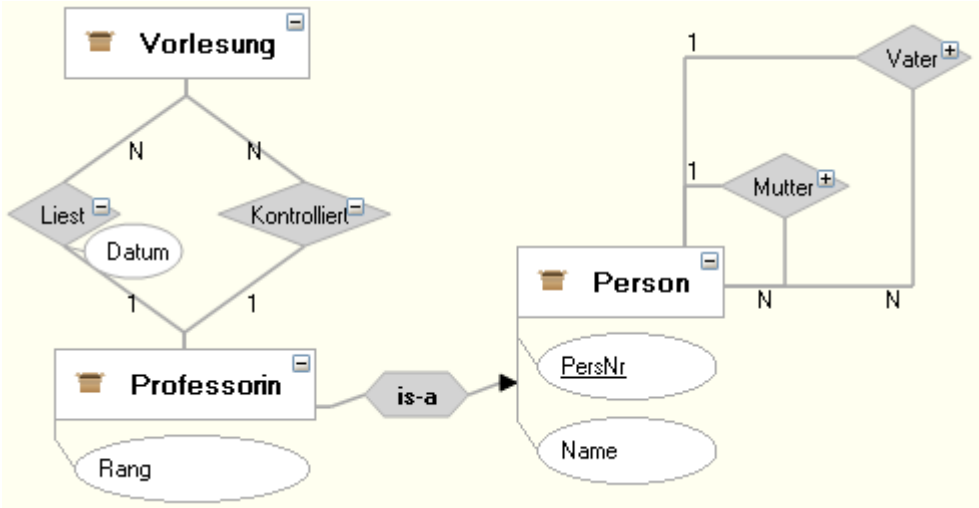


Abbildung 105 – Beispiel in der Chen-Notation

7.1.1 Elemente

Das Entitätsmengenelement
Das Element besteht aus einem Rechteck, welches den Namen der Entitätsmenge beinhaltet. Darunter werden allfällige Attribute in Ellipsenform aufgelistet und über eine Linie mit dem Entitätskopf verbunden. Aus Gründen der Übersichtlichkeit weisen sie die gleiche Breite auf. Unterschiedliche Grössen der Attributsellipsen bringen eine gewisse "Unruhe" ins Modell. Schlüsselattribute werden unterstrichen und immer als erstes aufgeführt. Mit dem „+" können die Attribute aus Überlegungen der Übersichtlichkeit versteckt werden.

Das Beziehungselement
Das Element wird (mit Ausnahme der is-a-Beziehung) als Rhombus gezeichnet. Allfällige Attribute haben auch hier eine Verbindung zum Element selber, welche wiederum mit dem „+" versteckt werden können. Bei zwei oder mehr Beziehungen zwischen den gleichen Entitätsmengen werden die Mittelpunkte der Elemente vertikal oder horizontal verschoben, um keine Überlagerungen zu bekommen. Weil der ursprüngliche Verbindungspunkt zur Entitätsmenge nicht verschoben wird, schreiben wir die Kardinalitäten jeweils zwischen dem Rombus und der Entitätsmenge an.

Der Spezialfall der Selbstbeziehung
Selbstbeziehungen werden in der rechten oberen Ecke der jeweiligen Entitätsmenge gezeichnet. Bei mehreren Selbstbeziehungen werden diese jeweils um die Breite des Vorgängers verschoben. Dies auch mit dem Hintergedanken, dass auch Selbstbeziehungen noch eigene Attribute haben können.

Der Spezialfall eines is-a-Beziehungselementes
Das Element wird als Sechseck dargestellt, enthält immer die Bezeichnung "is-a" und wird fett geschrieben. Dieses Element kann per Definition keine Attribute enthalten.

7.1.2 Implementation

Hier lohnt sich ein Klassendiagramm nicht für die Darstellung, da es sich nur um drei Klassen handelt. Sie beinhalten jeweils wenig spektakuläre Methoden. Es handelt sich mehr oder weniger um das Berechnen einzelner Punkte und das Zeichnen von Ellipsen, Rechtecke und Linien. Die einzelnen Beschreibungen wurden deshalb ausgelassen und können in der Codedokumentation nachgelesen werden.

Klasse	Beschreibung
ChenNotation	Diese Klasse stellt den Eintrittspunkt zur Notation dar und implementiert das Interface *ILogicalNotation*. Sie dient als Kommunikationsplattform zwischen dem PlugIn und der Software.
EntityControl	Diese Klasse übernimmt die Darstellung einer logischen Entitätsmenge und ist von *ElementControl* abgeleitet. Die *Draw*-Methoden werden zum Zeichnen der Elemente benutzt. Alle Methoden mit dem Prefix "Calc" sind zum Bestimmen von Längen und Breiten für die automatische Grössenberechnung des Controls zuständig.
RelationshipDrawer	Der *RelationshipDrawer* steuert das Zeichnen der Beziehungen. Ähnlich wie das *EntityControl* besteht sie auch aus "Draw" und "Calc" Methoden, um die jeweiligen Positionen der Formen, Rollennamen, Beziehungsnamen, Kardinalitäten usw. zu berechnen.

Tabelle 65 – Beschreibung der Klassen der Chen-Notation

7.2 Modified Chen-Notation

Die Modified Chen-Notation ist nur eine Sonderform der normalen Chen-Notation. Einzig die Kardinalitäten können genauer angegeben werden. Alle anderen Zeichnungselemente sind mit der normalen Chen identisch. Deshalb verzichten wir auf eine nochmalige Beschreibung.

7.3 ODLG-Notation

ODLG ist eine rein objektorientierte Darstellungsform. Sie existiert auf der logischen Ebene nicht, weil sie nicht alle Konstrukte (z.B. Attribute auf Beziehungen) abbilden kann.
Da sich in [3] keine Abbildung von Klassen mit Attributen finden liess, haben wir in unserer ODLG Notation die Attribute wie in UML direkt unter der Klasse gezeichnet. Die restlichen Elemente entsprechen den Definitionen im *Kapitel 3.4.6 ODLG (Object Definition Language Graphical)*.

Der Abbildung 106 liegt dasselbe logische Schema zu Grunde, welches wir bei der Beschreibung der Chen-Notation weiter oben verwendet haben.

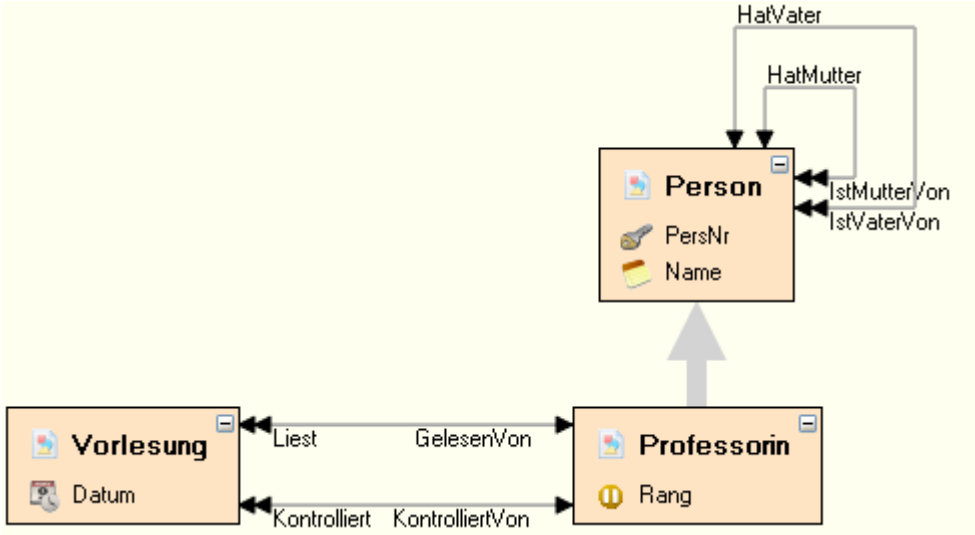


Abbildung 106 – Beispiel in der ODLG-Notation

Auch hier lohnt sich die Darstellung eines Klassendiagramms nicht, da es sich wiederum nur um drei Klassen *(OdlgNotation*, *AssociationDrawer* und *ClassControl)* handelt. Diese sind nach dem gleichen Prinzip aufgebaut, wie die Klassen in der Chen-Notation mit dem Unterschied, dass die Klasse *OdlgNotation* das Interface *IPhysicalObjectorientedNotation* implementiert.

7.4 Krähenfuss-Notation

Die Krähenfuss-Notation (Crowfoot) wurde von uns als die idealste Darstellungsform für die physikalisch relationale Ebene empfunden. Sie könnte zwar auch n:m-Beziehungen abbilden, bietet aber keine Möglichkeit, auf Beziehungen noch weitere Attribute anzuhängen. Daher ist auch sie für die logische Darstellung unbraucher. Die Elemente entsprechen der Definitionen im *Kapitel 3.4.4 Krähenfuss / Martin / Information Engineering (IE)*.

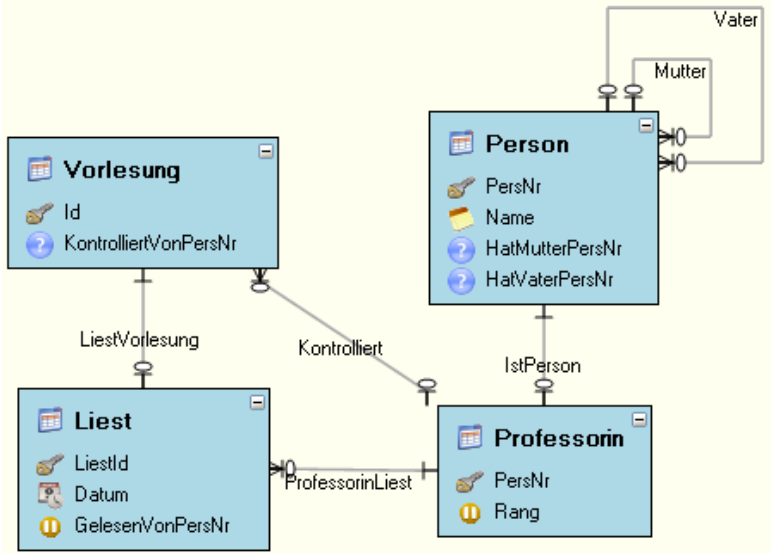


Abbildung 107 – Beispiel in der Krähenfuss-Notation

Die Krähenfuss-Notation definiert die Klassen *CrowfootNotation*, *TableControl* und *RelationshipDrawer.* Die Klasse CrowfootNotation implementiert das Interface *IPhysicalRelationalNotation*. Ansonsten ist die Implementation wiederum der Chen-Notation ähnlich, weshalb wir auch hier auf eine weitere Beschreibung verzichten.

8 Benutzeroberfläche

Dieses Kapitel beschreibt die Benutzeroberfläche von dataMagus.

8.1 Hauptfenster

Das Hauptfenster der Software wurde ansatzweise im *Kapitel 4.5 Software-Layout* bereits einmal grob beschrieben. Deshalb wird auf die Aufteilung der Bereiche nicht nochmals eingegangen.
Die nachfolgende Abbildung soll jedoch einen Eindruck der Software vermitteln.

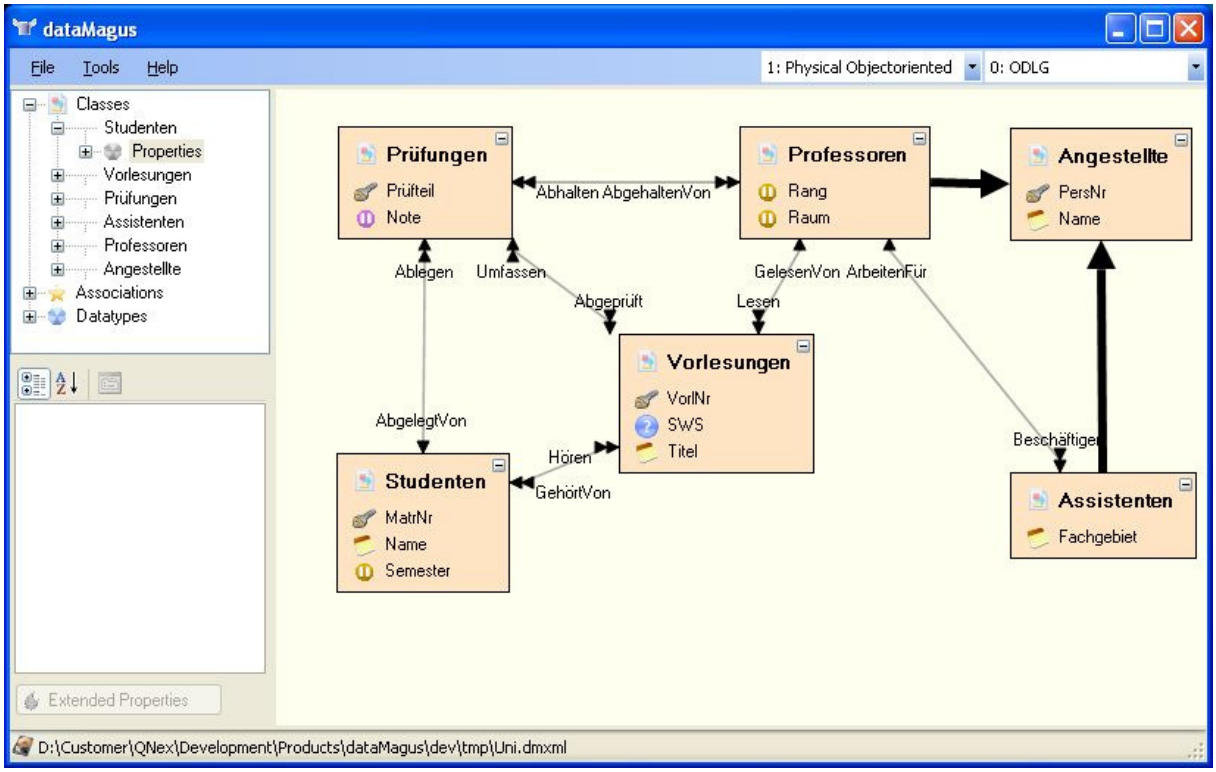


Abbildung 108 – Applikation dataMagus

8.2 Menüleiste

In der Menüleiste befinden sich einerseits die Menüs, andererseits die Dropdowns zum Wechseln zwischen den Ebenen und/oder den Notationen. Dabei beinhalten diese nur die im jeweiligen Kontext korrekten Einträge. Konkret bedeutet dies, dass z.B. auf der logischen Ebene nur logische Notationen ausgewählt werden können.

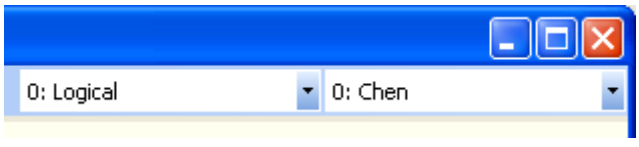


Abbildung 109 – Dropdowns zur Auswahl der Ebenen und Notationen

Im Menü *Datei (File)* sind die folgenden Menüpunkte zu finden:

New: Ein neues Schema wird erstellt.
Open: Ein bestehendes Schema kann geladen werden.
Save: Ein Schema kann gespeichert werden.
Save As: Ein Schema kann unter einem anderen Namen gespeichert werden.
Exit: Die Applikation wird beendet.

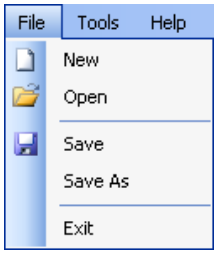


Abbildung 110 – Menü Datei

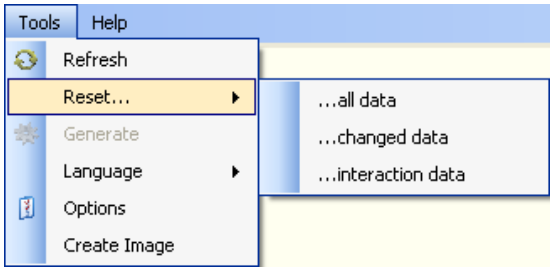


Abbildung 111 – Menü Extras

Das Menü *Extras (Tools)* beinhaltet diverse kleine Funktionalitäten.
Je nachdem, welche Ebene gerade aktiv ist, sind gewisse Menüpunkte nicht verfügbar.

Refresh: Aktualisiert die Ansichten in der Software. Dabei wird der Navigationsbaum neu aufgebaut, der Datenbereich geleert und das Schema neu gezeichnet. Bei Darstellungsproblemen ist diese Funktionalität hilfreich.
Reset...all data: Setzt alle möglichen Daten zurück.
Reset...changed data: Setzt die geänderten Daten auf der physikalischen Ebene zurück.
Reset...interaction data: Setzt die Antworten zu den Fragen bei der Umwandlung von der logischen Ebene auf die physikalischen Ebenen zurück.
Generate: Öffnet den Generator-Dialog zur Erzeugung von Quellcode.
Language: Wechselt die aktuelle Applikationssprache.
Options: Öffnet den Options-Dialog zum Ändern von globalen Applikationsoptionen.
Create Image: Erzeugt ein Bild des aktuellen Diagramms.

Das Menü *Hilfe (Help)* enthält folgende Menüeinträge:

Manual: Bedienungsanleitung der Software als PDF.
Website: Webseite zur Software.
About...: Öffnet den About-Dialog mit Informationen über die Autoren, die Lizenz und Version des Programms.

Abbildung 112 – Menü Hilfe

8.3 Navigationsbereich

Der Navigationsbereich verändert sich je nachdem, welche Ebene gerade aktiv ist. Der Navigationsbaum enthält ein Kontextmenü, welches bei einem Rechtsklick mit der Maus aktiviert wird. Das Kontextmenü ist von der Knotenart abhängig. Gewisse Knoten besitzen kein Menü. Wird ein Element im Baum selektiert, wird es direkt im Datenbereich angezeigt, damit es modifiziert werden kann.

8.3.1 Navigationsbaum in der logischen Ebene

Es ist gut ersichtlich, dass die logische Navigation nicht nur Attribute auf Entitätsmengen zulässt, sondern auch noch auf Beziehungen (mit Ausnahme der is-a-Beziehung).

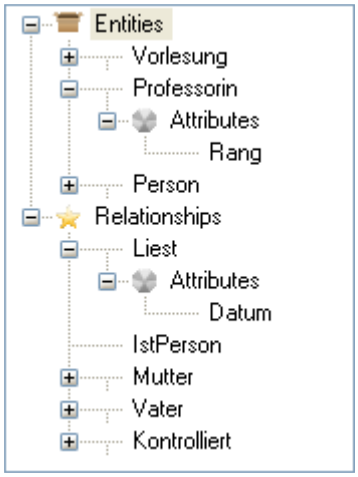


Abbildung 113 – Navigationsbaum der logischen Ebene

Kontextmenüs:
Bei Gruppenknoten werden die jeweiligen *Add new...*-Menüpunkte angeboten.

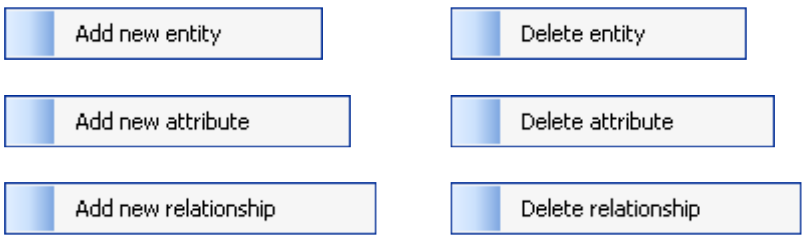


Abbildung 114 – Kontextmenüs im logischen Navigationsbaum

Ist ein konkretes Element selektiert, werden die entsprechenden *Delete...*-Funktionen angeboten.

8.3.2 Navigationsbaum auf der physikalisch relationalen Ebene

Ganz gemäss den Regeln einer relationalen Datenbank sind nun im Navigationsbaum keine Attribute mehr auf den Beziehungen möglich. Jedoch enthält der Baum den neuen Eintrag *Datatypes*. Darüber lassen sich die abstrakten Datentypen mappen und erweitern. Der restliche Baum kann nicht verändert werden.

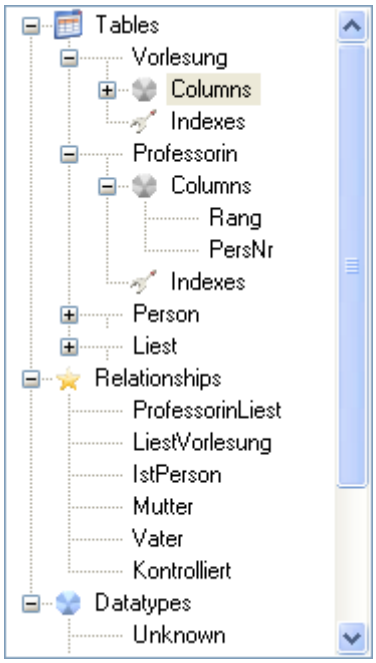


Abbildung 115 – Navigationsbaum der relationalen Ebene

Kontextmenü:
Ist der Gruppenknoten *Datatypes* selektiert, wird sein Kontext-Menü erstellt:

Abbildung 116 – Kontextmenü im relationalen Navigationsbaum

8.3.3 Navigationsbaum auf der physikalisch objektorientierten Ebene

Der Navigationsbaum auf der physikalisch objektorientierten Ebene enthält die objektorientierten Elemente und funktioniert im Weiteren genau gleich wie der Baum auf der relationalen Ebene.

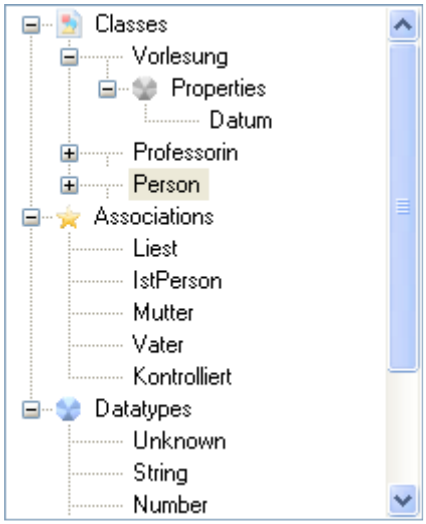


Abbildung 117 – Navigationsbaum auf der physikalisch objektorientierten Ebene

8.4 Datenbereich

Im Datenbereich können Objekte resp. Elemente zur Modifikation dargestellt werden. Das jeweils aktive Element im Navigationsbaum wird im Datenbereich angezeigt. Definiert ein Element noch zusätzliche Eigenschaften, können diese über die *Extended Properties* geladen werden. Eigenschaften, die schwer im Eigenschafts-Grid angezeigt werden können, werden separat in der Dialogbox angezeigt. Dies beinhaltet auch generatorspezifische Einstellungen zu den jeweiligen Elementen.

Abbildung 118 – Eigenschaften der Elemente

8.4.1 Erweiterte Dialoge

Die erweiterten Dialoge gehören zu den entsprechenden selektieren Elementen und werden über den *Extended Properties*-Knopf geladen. Wir werden hier nicht alle elf Dialoge auflisten und erklären, da sie in vielen Fällen sehr ähnlich oder gar identisch aufgebaut sind. Wir picken nur ein paar interessante Exemplare heraus und beschreiben sie dafür umso intensiver.

Im erweiterten Dialog einer relationalen Beziehung sind zusätzlich die Mappings der Felder für den Fremdschlüssel-Constraint ersichtlich. Zudem können über die Lasche *Target* generatorspezifische Eigenschaften festgelegt werden, sofern der Generator solche unterstützt. Diese *Target*-Lasche ist bei jedem erweiterten Dialog ersichtlich.

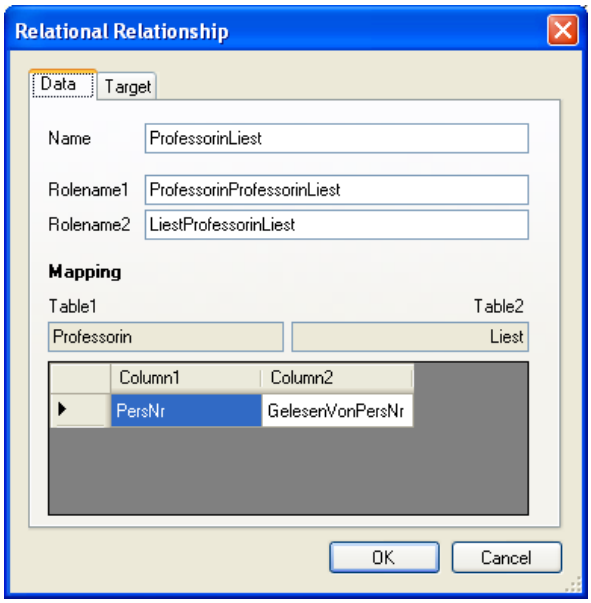


Abbildung 119 – Extended Properties auf einer relationalen Beziehung

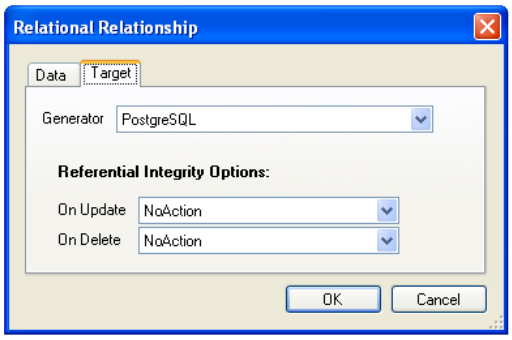


Abbildung 120 – Generatorspezifische Einstellungen auf der Lasche *Target*

Der Dialog zur Erweiterung von relationalen Feldern (oder auch objektorientierten Eigenschaften) bietet zusätzlich die Möglichkeit, dem Feld resp. der Eigenschaft einen abstrakten Datentypen zuzuweisen, damit der Generator beim Erstellen des Quellcodes die entsprechenden Typen verwenden und umsetzen kann.

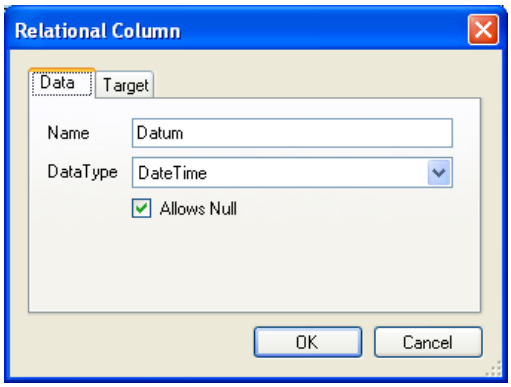


Abbildung 121 – Extended Properties auf einem relationalen Feld

8.5 Andere Programmoptionen

In der Software existieren noch andere Dialoge, die bisher nicht beschrieben wurden. Wir umreissen diese hier kurz.

8.5.1 Erstellen einer logischen Beziehung

Soll eine neue logische Beziehung erstellt werden, öffnet sich ein Erfassungsdialog dafür. Dieser ist notwendig, weil das Erfassen oder Bearbeiten einer logischen Beziehung ein recht komplexes Unterfangen ist. Es gibt diverse Businessregeln, die beim Erfassen einer solchen Beziehung eingehalten werden müssen (z.B. das Abfangen von zirkulären Referenzen oder Mehrfachvererbungen). Je nach Beziehung gibt es weitere Einstellungsmöglichkeiten bei den Kardinalitäten. Als weitere kleine Hilfe werden die Rollennamen automatisch generiert, sofern diese nicht manuell definiert werden.

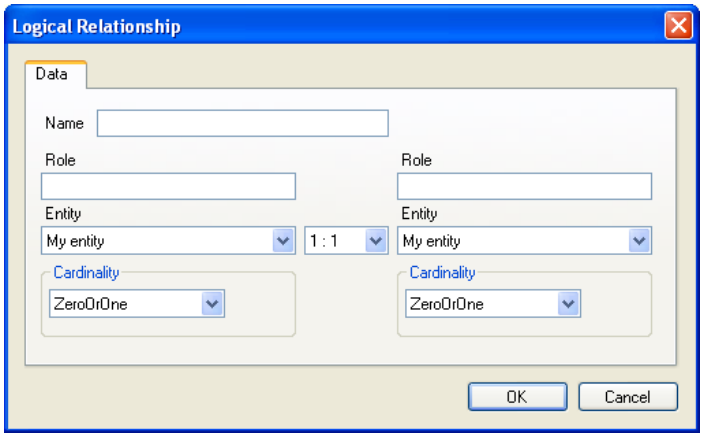


Abbildung 122 – Erfassen einer logischen Beziehung

8.5.2 Erstellen und Mappen von benutzerspezifischen Datentypen

Möchte der Benutzer der Software noch zusätzliche, abstrakte Datentypen erfassen, weil ihm oder ihr die aktuellen nicht genügen, kann er oder sie dies in den physikalischen Ebenen tun. Dazu klickt er oder sie im Navigationsbaum auf die *Datatypes* und kann mit dem Kontextmenü *Manage custom datatypes* folgenden Dialog laden.

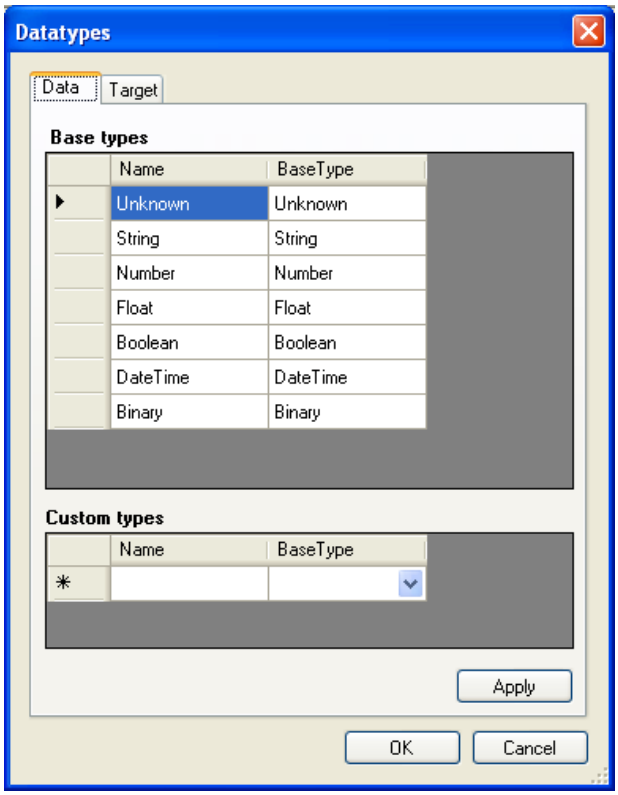


Abbildung 123 – Abstrakte physikalische Datentypen

Dieser Dialog erlaubt es, neue Typen zu erfassen, die auf einem Basistypen basieren. Neue Typen werden direkt auf dem jeweiligen Schema gespeichert. In der Lasche *Target* können alle bestehenden Typen des Schemas targetspezifisch gemappt werden.

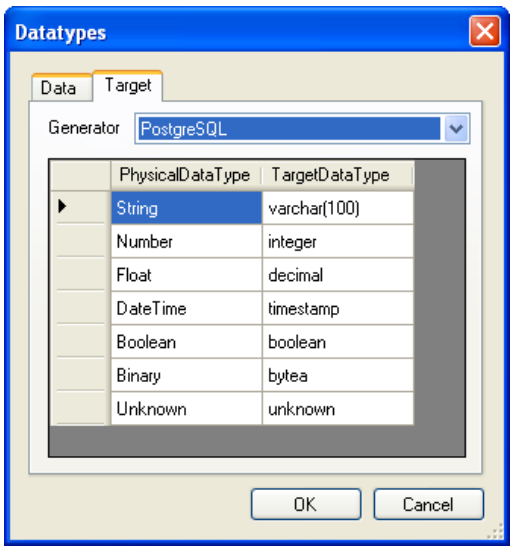


Abbildung 124 – Datentypen targetspezifisch mappen

8.5.3 Quellcode generieren

Der Dialog für die Quellcode-Generierung kann bei einem physikalischen Schema über den Menüpunkt *Generate* geöffnet werden. Er listet alle zum Schema verfügbaren Generatoren auf. Zusätzlich enthält er die Lasche *Options*, in welcher zusätzliche Optionen zum Erzeugen des Quellcodes vom jeweiligen Generator zur Verfügung gestellt werden kann. Mit einem Klick auf den *Generate*-Kopf wird die Quellcode-Erzeugung gestartet. Ist der Prozess erfolgreich durchgelaufen, werden die Code-Fragmente in der Lasche *Source Code* angezeigt.

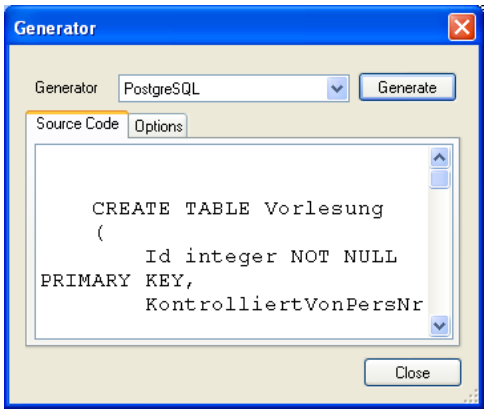


Abbildung 125 – Generierung von Quellcode

8.5.4 Schema als Bild exportieren

Vom aktuellen Schema kann ein Bild im PNG-Format erzeugt werden. Dazu erscheint eine kleine Dialogbox, welche die Grösse des Bildes abfragt. Da dieses Feature in unserer Anforderungsliste als optional gekennzeichnet wurde und der uns vorgegebene Zeitrahmen zu knapp war, haben wir darauf verzichtet, die Grösse des Bildes automatisch berechnen zu lassen.

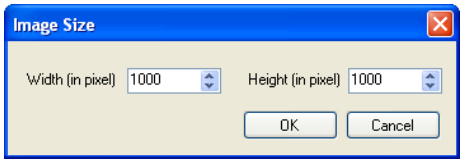


Abbildung 126 – Festlegung der Bildgrösse

8.5.5 Programmoptionen

Der Optionsdialog beinhaltet die globalen Applikationsoptionen. Diese Einstellungen werden jedoch beim Verlassen der Software nicht persistent gespeichert.

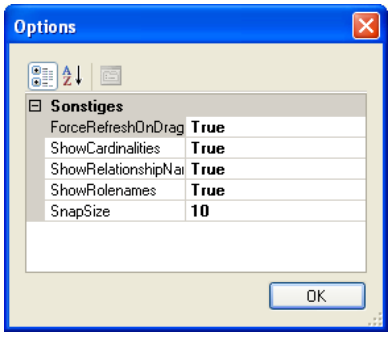


Abbildung 127 – Programmoptionen

9 Testergebnisse

Unsere anfänglich recht umfangreiche Analyse der verschiedenen möglichen Umsetzungsregeln und -prozesse haben uns aufgezeigt, dass ein Testing nach einem fixen Schema nicht funktionieren wird. Dabei gibt es einfach zu viele Möglichkeiten, die alle getestet werden müssten. Weil auch von der Dozentin keine intensiven und ausführlichen Tests gewünscht wurde, haben wir uns auf ein einfaches Testen der Geschäftsfälle mit einem speziellen Augenmerk auf die Umsetzungsregeln in *Kapitel 4.8.5 Umsetzung der logischen Ebene in die physikalischen Ebenen* entschlossen. Dabei lassen wir allgemeine und rudimentäre Vorgänge im Programm wie z.B. das Erstellen einer neuen Entität weg. Auch die Umsetzungsregeln werden hier nicht nochmals aufgeführt. Diese wurden alle mehrfach erfolgreich durchgetestet wurden. Die folgenden Kapiteln beschreiben einige auserwählte Tests.

9.1 Logische Modellierungstests

Die folgende Tabelle beschreibt die logischen Modellierungstests:

Geschäftsfall	Beschreibung der Tests	Ergebnis
Modellierung auf logischer Ebene	Bewegen von über 30 Entitäten auf dem Schemadesigner.	OK. Bewegung verläuft nicht mehr so flüssig.
Logische Entitätsmenge verwalten	Schlüsselattribute setzen.	OK.
Logische Beziehung verwalten	Erstellung mehrerer Beziehungen auf die gleiche Entitätsmenge.	OK. Alle Beziehungen werden angezeigt.
Verwalten von Attributen	Versuch, ein Key-Attribut zu erstellen bei einer Entität, die eine is-a Beziehung zu einer anderen Entität enthält.	OK. Änderung wird abgewiesen.
Verwalten von 1:1-Beziehung	Versuch, Schlüsselattribute auf der Beziehung zu erfassen.	OK. Änderung wird zurückgewiesen.
Verwalten von 1:n-Beziehung	Versuch, Schlüsselattribute auf der Beziehung zu erfassen.	OK. Änderung wird zurückgewiesen.
Verwalten von n:m-Beziehung	Versuch, Schlüsselattribute auf der Beziehung zu erfassen.	OK. Änderung wird akzeptiert.
Verwalten von is-a-Beziehung	Versuch, eine zirkuläre is-a Beziehung zu erstellen. Versuch, eine Mehrfachvererbung zu modellieren.	OK. Wird nicht zugelassen. OK. Wird nicht zugelassen.

Tabelle 66 – Logische Modellierungstests

9.2 Physikalische Tests

Die folgende Tabelle beschreibt die physikalischen Tests:

Geschäftsfall	Beschreibung der Tests	Ergebnis
Wechsel zwischen den Ebenen	Bewegen von über 30 Entitäten auf dem Schemadesigner.	OK. Bewegung verläuft nicht mehr so flüssig.
Generierung des physikalischen Modells	Zweimaliges Generieren hintereinander.	OK.
	Benutzeraktionsdialog abbrechen.	OK. Die Standard-Antwort wird verwendet.
Modifikation auf physikalischer Ebene	Ändern eines Datentyps auf einem Primärschlüssel-Feld.	OK. Datentyp aller Fremdschlüssel wird auch geändert.
	Versuch, den Datentyp eines Fremdschlüssel-Feldes zu verändern.	OK. Dies ist nicht möglich.
	Vesuch, auf einem Primärschlüssel-Feld die Option *AllowNulls* anzuhaken.	OK. Dies ist nicht möglich.
Zurücksetzen der physikalischen Daten	Testen, ob die Standardwerte wieder angezeigt werden.	OK.
	Testen, ob erneut Benutzerinteraktionen getätigt werden müssen.	OK.

Tabelle 67 – Physikalische Tests

9.3 Quellcode-Tests

Zur Generierung des Quellcodes wurden folgende Tests durchgeführt:

Geschäftsfall	Beschreibung der Tests	Ergebnis
Generatorspezifische Eigenschaften festlegen	Testen, ob Angaben zur referentiellen Integrität auf der physikalisch relationalen Ebene zu den Generatoren MS SQL Server und PostgreSQL gemacht werden können.	OK. Änderungen können gemacht werden und werden auch gespeichert.
Generation von Quellcode	Testen, ob voll funktionstüchtiger Code entsteht. Testen, ob die Angaben zur referentiellen Integrität (siehe oben) generiert werden.	OK. Die Codefragmente wurden jeweils auf der entsprechenden Datenbank laufen gelassen. OK.

Tabelle 68 – Quellcode-Tests

9.4 Andere Tests

Die folgende Tabelle definiert alle übrigen Tests:

Geschäftsfall	Beschreibung der Tests	Ergebnis
Speichern/Laden des Modells	Speicherung des Modells mit zusätzlichen generatorspezifischen Daten und Datenmappings. Gespeichertes Modell einlesen.	OK. Alles wurde korrekt serialisiert. OK. Die Daten sind wieder vorhanden.
Programmeinstellungen ändern	Einstellungen verändern (z.B. Rollennamen ausschalten).	OK. Das Schema wird entsprechend gezeichnet.

Tabelle 69 – Andere Tests

9.5 Auswertung der Testergebnisse

Die Tests liefen in den meisten Fällen problemlos durch. Dies aufgrund der Tatsache, dass wir während der Entwicklungsphase die jeweiligen Regeln immer im Hinterkopf hatten und bei Unklarheiten die Definitionen konsultierten. Andere Runtime-Probleme behoben wir eigentlich immer gerade on-the-fly. Offensichtliche, fehlerhafte Angelegenheiten waren in unseren Tests deshalb nicht mehr zu erwarten. Wir konnten uns somit auf die komplexeren Dinge konzentrieren.

Selbstverständlich liefen nicht immer alle Testfälle auf Anhieb fehlerfrei durch. Trotzdem hatten wir sehr wenig logische Unkorrektheiten zu beklagen. Schlussendlich haben wir es geschafft, alle gestellten Testfälle erfolgreich abzuschliessen.

10 Rückblick und Ausblick

Zu Beginn waren wir uns nicht im Klaren, mit welchem Aufwand diese Diplomarbeit verbunden sein wird. Vor allem am Anfang waren noch sehr viele Umsetzungsregeln oder andere zwingende Logik unklar bis unbekannt. Deshalb haben wir sehr viel Zeit in die Beschreibung von klaren Definitionen investiert. Dies schien uns als zwingend notwendig, um eine solche Software, die als Lernsoftware natürlich korrekt sein muss, überhaupt umsetzen zu können. Sie dienten uns dann später als nahrhafte Grundlage, als es an die Implementierung ging. Die Umsetzung sämtlicher Funktionalitäten mit einem vertikalen Prototypen hat uns sehr viele Vorteile und eine grosse Sicherheit gebracht. Als wir sicher waren, dass die Kommunikationswege innerhalb der Software reibungslos funktionieren, und die kritischen Teile auf Machbarkeit geprüft waren, konnten wir den Prototypen horizontal ausbauen. Dabei setzten wir auf die mittlerweile bekannte und etablierte n-Tier-Architektur, damit die Daten von ihrer Repräsentation getrennt und die Aufgaben der einzelnen Schichten klar verteilt werden.

Während der ganzen Diplomarbeit konnten wir uns immer recht gut ergänzen und haben schwierige Sitationen immer ausführlich diskutiert oder gar per XP (Extreme Programming) durchgeführt. Daraus entstand der heute recht stabile und korrekte Quellcode. Mit dem flexiblen PlugIn-Konzept für die Notationen und Generatoren haben wir eine solide Basis zur Erweiterung geschaffen. Auch weitere Modellierungselemente und physikalische Ebenen sind einfach erweiterbar.

Mit der vorliegenden Diplomarbeit ist unsere Lernsoftware dataMagus entstanden, auf welche wir sehr stolz sind. Die gesteckten Ziele wurden in jeder Hinsicht erreicht. Natürlich lässt die Materie an sich noch viel Raum für Erweiterungen.
dataMagus enthält 26'180 Zeilen reinen Code und 12'014 reine Kommentarzeilen. Die 175 Dateien haben eine Grösse von 1.78 MB und beinhalten alles in allem insgesamt 44'290 Zeilen.

Die Software hat einen sehr stabilen und korrekten Stand. Da das Programm unter der GNU GPL läuft, kann je nach Interesse mit einer Weiterentwicklung gerechnet werden. dataMagus könnte in Zukunft auch von anderen Hochschulen im Datenbankunterricht eingesetzt werden. Wir empfehlen den Einsatz des Programms im Unterricht. Dadurch ist mit dieser Diplomarbeit etwas Sinnvolles entstanden, das auch genutzt werden kann.

Literaturverzeichnis

Bücher und Dokumente

[1] Peter Pin-Shan Chen (1976). The Entity-Relationship Model – Toward a Unified View of Data
http://bit.csc.lsu.edu/~chen/pdf/erd.pdf
(Abgerufen am 19. Oktober 2007, 19:10 MEZ)

[2] Prof. Alfons Kemper, Ph. D. & Dr. André Eickler (2004, 5.Auflage). Datenbanksysteme. Eine Einführung. München: Oldenbourg Wissenschaftsverlag GmbH.
ISBN 3-486-27392-2

[3] R.G.G. Cattell & Douglas K. Barry (2000). The Object Data Standard: ODMG 3.0. San Diego: Academic Press
ISBN 1-55860-647-4

[4] Unterlagen des Datenbank-Unterrichtes bei Prof. Dr. T. Olnhoff

[5] Übung zur Vorlesung Datenbanksysteme im WS05/06
Richard Kuntschke und Martin Wimmer
http://www-db.in.tum.de/teaching/ss05/dbsysaf/exercises/blatt04/Loesung04.pdf
(Abgerufen am 25. Oktober 2007, 18:15 MEZ)

Links

[6] Microsoft .NET Online-Hilfe
http://msdn2.microsoft.com/de-de/asp.net/default.aspx
(Abgerufen am 14. November 2007, 10:25 MEZ)

[7] PostgreSQL Online-Hilfe
http://www.postgresql.org/docs/manuals/
(Abgerufen am 08. November 2007, 09:15 MEZ)

[8] Microsoft SQL Server 2000 Online-Dokumentation
http://msdn2.microsoft.com/de-de/library/ms203721.aspx
(Abgerufen am 15. November 2007, 13:05 MEZ)

[9] http://www.gnu.org/licenses/#GPL
(Abgerufen am 21. November 2007, 12:40 MEZ)

[10] http://www.icsharpcode.net/OpenSource/SD
(Abgerufen am 19. Oktober 2007, 10:05 MEZ)

[11] http://subversion.tigris.org
(Abgerufen am 19. Oktober 2007, 10:15 MEZ)

[12] http://www.kynosarges.de/NDoc.html
(Abgerufen am 21. November 2007, 12:45 MEZ)

[13] http://de.wikipedia.org
(Abgerufen am 23. Oktober 2007, 20:10 MEZ)

Bilder

[14] Logo SQL-Server: http://www.microsoft.com/sql/default.mspx
(Abgerufen am 22. Oktober 2007, 12:30 MEZ)

[15] Logo PostgreSQL: http://marakana.com/static/images/logos/course/PostgreSQL.jpg
(Abgerufen am 22. Oktober 2007, 12:45 MEZ)

[16] Logo Oracle: http://www.oracle.com/index.html
(Abgerufen am 22. Oktober 2007, 13:15 MEZ)

[17] Logo ODMG: http://www.odmg.org/odmg.gif
(Abgerufen am 22. Oktober 2007, 13:20 MEZ)

[18] Logo db4Objects: http://www.qnx.com/images/partners/qpn/db4o.gif
(Abgerufen am 22. Oktober 2007, 13:30 MEZ)

[19] Logo Java: http://joannapenabickley.typepad.com/on/images/java_logo.gif
(Abgerufen am 22. Oktober 2007, 13:35 MEZ)

[20] Produkt Erwin Data Modeler
http://shop.linkplus.com.tr/sonkullanici/images/AF_ErwinDataModelerSmall.jpg
(Abgerufen am 25. Oktober 2007, 12:50 MEZ)

[21] Produkt DB Designer 4
http://www.fabforce.net/dbdesigner4/
(Abgerufen am 25. Oktober 2007, 13:00 MEZ)

[22] http://de.wikipedia.org/wiki/Bild:Hierarchisches_Datenbankmodell.png
(Abgerufen am 25. Oktober 2007, 18:30 MEZ)

[23] http://de.wikipedia.org/wiki/Bild:Netzwerkdatenbankmodell.png
(Abgerufen am 25. Oktober 2007, 18:35 MEZ)

Abbildungsverzeichnis

Tabellenverzeichnis