Universität Kaiserslautern
Fachbereich Informatik
AG Datenbanken und Informationssysteme
Prof. Dr. Dr. h.c. Theo Härder

Konzeption und Realisierung eines intuitiven CWM-basierten SQL-Editors

Diplomarbeit

von

Martin Husemann

Betreuer:

univ. dipl. inz. Jernej Kovse

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig und unter ausschließlicher Verwendung der angegebenen Literatur angefertigt habe.

Kaiserslautern, den 24.08.2003

Martin Husemann

Inhaltsverzeichnis

Kapitel 1 Einleitung

Die systematische Software-Entwicklung ist eine nach wie vor junge Ingenieurswissenschaft. Mittlerweile existieren erprobte Verfahrensweisen für die Organisation langfristiger Projekte und die Entwicklung großer Softwaresysteme, doch im Vergleich zu etablierten Wissenschaften wie dem Maschinenbau oder dem Bauingenieurwesen mangelt es immer noch an der breiten Akzeptanz von allgemein gültigen Vorgehensweisen. In der Praxis werden so Anwendungen von Einzelfall zu Einzelfall von Grund auf neu und individuell entwickelt, und bei nachfolgenden Projekten findet bestenfalls eine Wiederverwendung von Implementierungskomponenten statt, wobei die Komponenten oftmals manuell an ihren neuen Einsatzzweck angepasst werden müssen. Eine Wiederverwendung größerer struktureller Einheiten ist kaum möglich, da aufgrund der Individualität der Erstentwicklung der Einsatzbereich stark eingeschränkt ist und der Aufwand für die Anpassung an einen neuen Einsatzbereich unverhältnismäßig groß wäre.

Hinsichtlich der dominierenden Fertigung individueller Einzelprodukte befindet sich die Software-Industrie heute auf einem Entwicklungsstand, den die klassische produzierende Industrie Mitte des neunzehnten Jahrhunderts innehatte. Mit der industriellen Revolution erfolgte dort der Einstieg in die Massenfertigung gleichartiger Produkte aus standardisierten Bauteilen. Der wesentliche Unterschied zur Einzelfertigung lag dabei zunächst weniger in der Maschinisierung oder Automatisierung der Produktion, auch wenn diese im zwanzigsten Jahrhundert maßgeblich zur Erhöhung des Ausstoßes und der Senkung der Kosten beitrug. Entscheidende Voraussetzung für die Massenfertigung war vielmehr der Übergang zur Herstellung von Produkten aus standardisierten und für mehrere Zwecke verwendbaren Bauteilen. Auf diese Weise lassen sich technisch ähnliche Produkte durch Auswahl leicht unterschiedlicher Bauteilgruppen und entsprechend variierte Kombination der Teile zum fertigen Produkt in der Summe wesentlich effizienter herstellen, als dies mit hochspezialisierten oder sogar von Exemplar zu Exemplar individuell angefertigten Bauteilen möglich wäre.

Aus dieser Erkenntnis entstanden für die Software-Entwicklung die Konzepte der Software-*Produktlinie* und der *Systemfamilie*. Bei einer Produktlinie handelt es sich um eine Gruppe inhaltlich verwandter Produkte (Softwaresysteme), die für ähnliche Einsatzzwecke ausgelegt sind, in ihrem internen Aufbau aber durchaus deutliche Unterschiede aufweisen können. Eine Systemfamilie umfasst dagegen eine Gruppe von Softwaresystemen, die in ihrer Realisierung technisch verwandt sind, indem sie beispielsweise alle einer bestimmten internen Architektur genügen. Im Idealfall bilden die Mitglieder einer Produktlinie auch eine Systemfamilie, so dass die inhaltlichen Gemeinsamkeiten sich im technischen Aufbau der Produkte widerspiegeln.

Die Strategie der *generativen Programmierung* setzt die Konzepte der Produktlinie und Systemfamilie im Bereich der Software-Entwicklung um. Anstelle individuelle Produkte in einzelnen

Projekten ohne Berücksichtigung der Gemeinsamkeiten zu entwickeln, legt ein Hersteller seine Tätigkeit auf einem bestimmten Anwendungsgebiet als Produktlinie an. Zu Beginn der Erschließung eines neuen Anwendungsgebietes entsteht mehr Aufwand, als es bei der Entwicklung eines Einzelproduktes der Fall wäre, da eine sinnvolle Aufteilung der Funktionalität in Komponenten gefunden werden muss, aus denen später das eigentliche Software-Produkt zusammengesetzt wird. Die Komponenten entsprechen also den standardisierten Bauteilen in der produzierenden Industrie. Sobald für ein Anwendungsgebiet alle notwendigen Komponenten implementiert sind, beschränkt sich die Fertigung weiterer Software-Produkte jedoch auf die richtige Auswahl und Kombination vorhandener Komponenten, so dass der Aufwand für die Herstellung neuer Produkte drastisch sinkt. Ziel der generativen Programmierung ist es, den Prozess der Komponentenauswahl und -konfiguration vollständig zu automatisieren. Der Software-Entwickler muss dann lediglich die Eigenschaften des gewünschten Produkts beschreiben, woraufhin ein Generator anhand der Beschreibung aus den Komponenten das fertige Produkt erstellt.

Im Rahmen dieser Diplomarbeit wurde die Nutzung generativer Programmierung für die Erstellung datenbankintensiver Anwendungen untersucht. Zu diesem Zweck wurde ein Werkzeug entwickelt, das auf mehreren Ebenen sowohl die direkte Modellierung von Datenbank-Schemas als auch die abstrakte Spezifikation von datenbankintensiven Anwendungen erlaubt. Abstrakte Spezifikationen können automatisiert in konkrete Realisierungen in Form von mit benutzerdefinierten Routinen angereicherten Datenbank-Schemas umgesetzt werden. Ein weiteres Ziel bei der Entwicklung des Werkzeugs war eine möglichst intuitive Bedienung. Bei den verschiedenen Modellierungsaktionen wird der Benutzer daher durch eine Reihe von Hilfsfunktionen unterstützt. Da die Modellierung in einem interaktiven Prozess zwischen Benutzer und Werkzeug erfolgt und das Endprodukt stets ein Datenbank-Schema ist, das als Menge von SQL-Befehlen an ein Datenbanksystem übertragen werden kann, wurde *SQLInteract* als Name für das Werkzeug ausgewählt.

SQLInteract erlaubt auf der untersten Abstraktionsebene den unmittelbaren manuellen Entwurf eines Datenbank-Schemas. Für die interne Verwaltung der modellierten Konstrukte kommt dabei eine Implementierung des *Common Warehouse Metamodel* (*CWM*) zum Einsatz. CWM wurde als Technologie für den hersteller- und produktübergreifenden Austausch von Metadaten über Datenbanken und Data Warehouses in heterogenen Umgebungen entwickelt. Es ist darüber hinaus jedoch auch für die Datenverwaltung innerhalb einer Anwendung geeignet. Die Darstellung des Datenbank-Schemas erfolgt sowohl in einer nach hierarchischen Gesichtspunkten gegliederten Baumansicht als auch in textueller Form als Menge von Befehlen in der *Data Definition Language* (*DDL*). Beide Darstellungsformen erlauben die Bearbeitung des Schemas; bei der Dateneingabe wird der Benutzer speziell in der textuellen Darstellung durch verschiedene Hilfsfunktionen unterstützt. Neben der Schemamodellierung ist auf dieser Bedienebene auch die Formulierung von Ad-hoc-Anfragen an ein Datenbanksystem möglich.

Auf einer weiteren Bedienebene kann der Benutzer auf abstrakte Art eine Produktspezifikation für eine datenbankintensive Anwendung erstellen, wie dies nach den Konzepten der generativen Programmierung vorgesehen ist. Als Beispiel für eine Produktlinie wurden in dieser Arbeit so genannte *Repositories* betrachtet, die eine versionierte Datenspeicherung erlauben. Die Bedienung bei der Erstellung einer Produktspezifikation ist identisch mit der Bedienung beim Schema-Entwurf. Die Spezifikation wird sowohl in einer Baumansicht als auch in textueller Form dargestellt, beide Darstellungsformen sind editierbar. Aus der abstrakten Spezifikation kann SQLInteract die konkrete Realisierung der modellierten Eigenschaften in Form einer datenbankintensiven Anwendung generieren. Dazu werden im Rahmen einer so genannten *Reduktion* die Konstrukte

der Spezifikation auf Tabellen, Datentypen und benutzerdefinierte Routinen umgesetzt. Das Ergebnis der Reduktion ist auf der unteren Bedienebene einsehbar und kann dort gegebenenfalls nachbearbeitet werden. Abschließend lässt sich die generierte Realisierung an ein Datenbanksystem übertragen, wo sie dann ein benutzbares Repository anlegt.

Neben den näher betrachteten Repositories können mit SQLInteract auch andere Produktlinien auf dem Gebiet der datenbankintensiven Anwendungen verarbeitet werden; das entsprechende Programmmodul für die Unterstützung einer Produktlinie ist austauschbar. Für die Erstellung eines Moduls zu einer Produktlinie müssen dabei außer dem zugrunde liegenden Metamodell für die Spezifikationsmodellierung auch Regeln angegeben werden, die die Abbildung zwischen Baumansicht und textueller Darstellung sowie die Reduktion einer Spezifikation auf konkrete Datenbank-Schemas beschreiben. Aus diesen Angaben werden dann die eigentlichen Java-Klassen für die Unterstützung der Produktlinie generiert und mit den übrigen Komponenten von SQLInteract integriert. SQLInteract ist also selbst als Produktlinie realisiert; da jedes Mitglied dieser Produktlinie ein Spezifikationswerkzeug und einen Generator für eine Produktlinie auf dem Gebiet der datenbankintensiven Anwendungen darstellt, kann SQLInteract auch als *Meta-Produktlinie* bezeichnet werden.

Die Kapiteleinteilung folgt dem dargelegten Aufbau von SQLInteract. Kapitel 2 stellt zunächst das Common Warehouse Metamodel vor, das als Verwaltungsstruktur für Datenbank-Schemas eine zentrale Stellung in SQLInteract einnimmt. Dabei werden die Intention bei der Entwicklung von CWM und die Verhältnisse zu anderen Standards beleuchtet. Das Kapitel schließt mit einer Darstellung der Architektur von CWM mit Schwerpunkt auf den für SQLInteract relevanten Teilen.

In Kapitel 3 wird beschrieben, wie der Entwurf eines Datenbank-Schemas abläuft. Die verschiedenen Komponenten der Benutzeroberfläche werden vorgestellt, wobei die Baumdarstellung und die damit verbundenen Konzepte im Mittelpunkt stehen. Weiterhin werden die Methoden für die Unterstützung der textuellen Eingabe anhand von Beispielen erläutert.

Kapitel 4 befasst sich mit Aspekten der Modellierung von Semantik. Nach einer Vorstellung des Ansatzes der *modellgetriebenen Architekturen* werden die Anforderungen erläutert, die SQLInteract an Spezifikationssprachen für die Semantikmodellierung stellt. Anschließend werden die Möglichkeiten für die Modellierung von Semantik in verschiedenen allgemeinen Spezifikationssprachen untersucht und bewertet.

Nachdem als Resultat der Bewertung die geringe Eignung allgemeiner Sprachen für die Semantikmodellierung in SQLInteract festgestellt werden muss, führt Kapitel 5 *domänenspezifische Sprachen* ein, wobei neben allgemeinen Grundlagen auch die Zusammenhänge zwischen domänenspezifischen Sprachen und generativer Programmierung betrachtet werden. Im Anschluss wird die Fallstudie der versionierten Datenspeicherung in Repositories ausführlich vorgestellt und die Modellierung von Produktspezifikationen mit SQLInteract erläutert.

Kapitel 6 beschreibt die Einbindung beliebiger Produktlinien in SQLInteract mithilfe so genannter *DSM-Module*, die das Metamodell der Spezifikationssprache sowie Regeln für die textuelle Darstellung und die Reduktion beinhalten. Darüber hinaus werden die Generierungszusammenhänge zwischen den drei Stufen behandelt, die bei der Arbeit mit SQLInteract berührt werden: Durch die Einbindung verschiedener generierter DSM-Module (1) entstehen SQLInteract-Varianten, die jeweils eine bestimmte Produktlinie unterstützen (2). Jede solche Variante generiert ihrerseits konkrete datenbankintensive Anwendungen als Mitglieder einer Systemfamilie (3).

Kapitel 7 fasst abschließend die Ergebnisse dieser Arbeit unter Bezugnahme auf verwandte Arbeiten zusammen und gibt einen Ausblick auf weitere Forschungsfelder im Zusammenhang mit generativer Programmierung und dem Ausbau von SQLInteract.

Kapitel 2 Common Warehouse Metamodel (CWM)

Die Spezifikation des *Common Warehouse Metamodel* (*CWM*) wurde im Februar 2001 von der *Object Management Group* (*OMG*) verabschiedet [OMG01b]. Die OMG ist ein internationaler Verband von Industrieunternehmen, Entwicklern und Endbenutzern und hat sich die Verbreitung objektorientierter Technologien auf dem Gebiet der Software-Entwicklung zur Aufgabe gemacht. Hauptziele sind dabei die Förderung von Wiederverwendbarkeit und Interoperabilität objektorientierter Software in verteilten Umgebungen.

2.1 Intention

CWM wurde entwickelt, um den Austausch von Metadaten über Data Warehouses und Geschäftsinformationen zwischen Warehouse-Anwendungen in heterogenen Umgebungen zu vereinfachen. Anlass für die Entwicklung war die Vielzahl der auf den Markt gebrachten Werkzeuge für Verwaltungs- und Analyseaufgaben in Data Warehouses, die oftmals proprietäre Ansätze bei der Metadatenhandhabung verfolgten und damit zu Insellösungen führten.

Gelegentlich ist es nicht möglich, sämtliche in einem Unternehmen anfallenden Metadaten einheitlich in einem globalen Repository zu verwalten, auf das alle Werkzeuge zugreifen können. CWM sieht daher den Austausch von Metadaten zwischen den Applikationen vor. Dafür wird ein Metamodell einer generischen Data-Warehouse-Architektur definiert, mit dessen Hilfe Metadaten in abstrakter Form einheitlich beschrieben werden können.

Im Detail werden mehrere Metamodelle für verschiedene Teilaspekte des Einsatzgebietes definiert, über die Metadaten anwendungsneutral modelliert werden können. Bei der Entwicklung wurde großer Wert darauf gelegt, CWM möglichst generisch zu halten und keinen zu engen Bezug auf konkrete Implementierungen zu nehmen. Um andererseits übermäßige Abstraktion zu vermeiden, wurden die in CWM aufzunehmenden Informationen aus Untersuchungen real existierender Warehouse-Systeme abgeleitet.

2.2 Verwandte OMG-Standards

CWM beruht auf drei Standards, die ebenfalls von der OMG verabschiedet wurden:

- *Unified Modeling Language* (*UML*) [OMG03a] wird als Notation verwendet.

- *Meta Object Facility* (*MOF*) [OMG02b] ist das Framework für die Organisation des CWM.
- *XML Metadata Interchange* (*XMI*) [OMG02a] dient als Format für den Datenaustausch.

UML und MOF wurden 1997 spezifiziert. Während UML sich rasch als Modellierungssprache durchsetzte, ermöglichte MOF die abstrakte Beschreibung von Metamodellen. Mit XMI wurde 1999 zusätzlich ein einheitliches Format für den Austausch von Metadaten vorgestellt. Zusammen bilden die drei Standards den Kern der *OMG Metadata Architecture*. CWM ist eine spezifische Erweiterung dieser Architektur für das Feld des Data Warehousing. Durch die Bereitstellung einheitlicher Methoden für die Handhabung von Metadaten erweitert es die Anwendbarkeit der mittlerweile etablierten Standards auf diesem Gebiet.

2.2.1 UML

Die Unified Modeling Language ist eine graphische Sprache zur Modellierung verteilter Objektsysteme. Ihr Haupteinsatzgebiet ist die objektorientierte Software-Entwicklung. UML entstand als Synthese der drei Modellierungssprachen, die zuvor dieses Gebiet dominierten: *Booch* [Boo93], *Object Modeling Technique* (*OMT*) [Rum91] und *Object Oriented Software Engineering* (*OOSE*) [Jac94]. Die eigentliche Entwicklung von UML fand in den Jahren 1994 und 1995 statt; im November 1997 wurde die Sprache schließlich von der OMG ratifiziert.

UML bietet Konstrukte zur Modellierung statischer und dynamischer Aspekte von Systemen. Wichtigstes Element der statischen Modellierung ist die *Klasse*, die durch Attribute, Operationen und Interfaces die Eigenschaften einer Gruppe gleichgearteter Entitäten beschreibt. Zwischen Klassen können Beziehungen wie *Vererbung*, *Assoziation*, *Abhängigkeit* oder *Einschluss* bestehen. Die Interaktion zwischen Klassen wird mit Hilfe von Kollaborations- oder Sequenzdiagrammen beschrieben. *Kollaborationsdiagramme* betonen dabei die Anordnung der Klassen zueinander, während *Sequenzdiagramme* vor allem den zeitliche Ordnung des Nachrichtenflusses zwischen den Klassen zeigen. Durch *Packages* können zusammenhängende Klassen gruppiert werden. Die Modellierung eines Systems lässt sich so zugunsten besser Übersichtlichkeit strukturieren.

Die formale Grundlage von UML ist das UML-Metamodell, das seinerseits unter Verwendung von UML rekursiv definiert ist. Dies ermöglicht die Beschränkung auf eine kleine Menge elementarer Terme.

Zusammenhang mit CWM

Kern von CWM ist das *Object-Model*-Package. Es basiert auf einer Variante des UML-Metamodells, die um die für das Gebiet des Data Warehousing irrelevanten Teile bereinigt wurde. Das gesamte CWM-Metamodell ist letztlich eine Erweiterung des Object-Model-Package, da alle definierten Klassen zumindest mittelbar von Klassen dieses Package erben. Aufgrund der gemeinsamen Basis ist die strukturelle Ähnlichkeit zwischen UML und CWM so groß, dass UML als Notation für die Diagramme von CWM eingesetzt wird und textuelle Constraints in der für UML genutzten *Object Constraint Language* (*OCL*) [OMG03a] angegeben werden. Durch die Verwendung von UML-Konzepten bei der Definition von CWM konnte der Aufwand für die Entwicklung neuer Konzepte vermieden werden. Darüber hinaus erleichtert der Verbreitungsgrad von UML den Einstieg in die Spezifikation.

2.2.2 MOF

Meta Object Facility ist eine Technologie für die Definition von Metadaten und ihre Repräsentation als CORBA-Objekte [OMG02c]. *Metadaten* sind Daten, die Informationen beschreiben. Generell können diese Informationen explizit, etwa in Dateien oder Datenbanken, oder implizit, beispielsweise in der Architektur eines Systems, vorliegen. Um in MOF darstellbar zu sein, müssen die Informationen in jedem Fall durch Objektmodellierungstechniken beschrieben werden können.

Im allgemeinen wird unter einem *Modell* die Beschreibung eines Realitätsausschnitts verstanden. Im Zusammenhang mit MOF ist der Begriff weiter gefasst und steht für eine Menge von Metadaten, die einer gemeinsamen abstrakten Syntax genügen und innerhalb eines semantischen Kontextes relevant sind. Da Metadaten selbst Informationen sind, können sie ihrerseits durch andere Metadaten beschrieben werden, die in der MOF-Terminologie als *Meta-Metadaten* bezeichnet werden. Konsequenterweise heißt ein Modell, das aus solchen Meta-Metadaten besteht, *Metamodell* und beschreibt die abstrakte Syntax der zugrundeliegenden Metadaten. Sind die durch ein Metamodell beschriebenen Metadaten die MOF-Repräsentation des entsprechenden Modells, wird das Metamodell als *MOF-Metamodell* bezeichnet. Um verschiedene Arten von MOF-Metamodellen abzudecken, definiert MOF eine weitere Abstraktionsschicht, das so genannte *MOF-Modell*. Das MOF-Modell ist ein Modell für Metamodelle, folglich kann es als Meta-Metamodell betrachtet werden und besteht damit aus Meta-Meta-Metadaten.

Da die Nomenklatur der „Meta“-Präfixe schnell unübersichtlich wird, werden die Schichten üblicherweise explizit benannt. Abbildung 2.1 zeigt die typische vierschichtige Architektur des MOF-Frameworks. Die Anzahl der Schichten in einer konkreten Implementierung ist jedoch nicht vorgeschrieben und kann vom Standard abweichen; folglich geben die „Meta“-Präfixe keine Auskunft über die absolute Position einer Schicht im Stapel, sondern lediglich über die relative Lage der Schichten zueinander.

Abbildung 2.1 OMG Metadata Architecture

Meta-Ebene	MOF-Bezeichnung	Beispiel
M3	Meta-Metamodell	MOF-Modell
M2	Metamodell, Meta-Metadaten	CWM-Metamodell
M1	Modell, Metadaten	CWM-Metadaten
M0	Daten	Warehouse-Daten

Die MOF-Spezifikation besteht im wesentlichen aus drei Teilen: dem *MOF-Modell*, dem *MOF-IDL-Mapping* und den *MOF-Interfaces*.

MOF-Modell

Das MOF-Modell ist das Meta-Metamodell von MOF, eine Art abstrakte Sprache zur Definition von Metamodellen. Es ist in seiner Aufgabe etwa vergleichbar mit dem UML-Metamodell, das zur Definition von UML-Modellen dient, im Stapel der Meta-Ebenen aber eine Ebene darüber angesiedelt.

Hinsichtlich der Modellierungskonzepte ähneln sich MOF und UML, so dass MOF-Metamodelle in UML-Notation dargestellt werden können. Die zentralen Modellierungselemente von MOF sind *Klasse*, *Assoziation* und *Package*. Analog zu UML verfügen Klassen über Attribute und Operationen und können durch Einfach- oder Mehrfachvererbung Eigenschaften anderer Klassen übernehmen. Assoziationen verknüpfen jeweils zwei Instanzen von Klassen, wobei strukturelle Constraints hinsichtlich der Kardinalität oder der Eindeutigkeit der Instanzen angegeben werden können. Packages bündeln inhaltlich zusammenhängende Klassen und Assoziationen und dienen so der Strukturierung eines modellierten Systems. Neben diesen Hauptkonstrukten definiert MOF einige simple Datentypen für die Verwendung als Parameter oder Attribute. Weiterhin können über Constraints semantische Einschränkungen von Modellierungselementen festgelegt werden.

MOF-IDL-Mapping

Mit dem MOF-IDL-Mapping können CORBA-IDL-Interfaces aus einem MOF-Metamodell generiert werden. Dies geschieht mit Hilfe eines Satzes von Standard-Templates, die aus dem MOF-Metamodell IDL-Interfaces für CORBA-Objekte erzeugen, die die vom MOF-Metamodell beschriebenen Metadaten repräsentieren können. Es handelt sich hier also nicht um eine Transformation des verwendeten MOF-Metamodells selbst (Meta-Ebene M2), sondern um die Bereitstellung von IDL-Interfaces für die durch das MOF-Metamodell beschriebenen, eine Schicht darunter angesiedelten Metadaten (Meta-Ebene M1).

Die zu erzeugenden IDL-Interfaces sind genau definiert, so dass verschiedene Implementierungen von MOF-Repositories zueinander kompatible Interfaces erzeugen können. Neben den im Rahmen der Generierung erzeugten spezifischen Interfaces sind außerdem einige grundlegende allgemeine Interfaces definiert, über die generisch auf die Metadaten zugegriffen werden kann, ohne Kenntnis der speziellen Interfaces zu haben.

MOF-Interfaces

Bei den MOF-Interfaces handelt es sich um IDL-Interfaces für CORBA-Objekte, die ein MOF-Metamodell (Meta-Ebene M2) repräsentieren. Über sie kann per Reflection auf die vom MOF-Metamodell beschriebenen Metadaten zugegriffen werden. Die MOF-Interfaces wurden erzeugt, indem das MOF-IDL-Mapping auf das MOF-Modell (Meta-Ebene M3) angewandt wurde.

Das MOF-Modell wurde ähnlich wie UML als Modellierungssprache für sich selbst benutzt. Daher kann es als Modell (auf Meta-Ebene M3) angesehen werden, dass einem Metamodell (auf Meta-Ebene M4) genügt, wobei dieses Metamodell das MOF-Modell selbst und damit insbesondere isomorph zum Modell auf Meta-Ebene M3 ist.

Zusammenhang mit CWM

MOF ist der OMG-Standard für die Darstellung von Metamodellen. Es lag daher nahe, CWM konform zu MOF zu entwickeln. Insbesondere können auf diese Weise diejenigen Standards mitgenutzt werden, die auf MOF aufbauen. Das Hauptaugenmerk lag dabei auf XMI als Format für den Datenaustausch und IDL als Beschreibungssprache für den programmatischen Zugriff auf Warehouse-Metadaten.

2.2.3 XMI

XML Metadata Interchange ist ein Standard für die Serialisierung und den Austausch von Modellen. Da MOF der OMG-Standard für Metadaten ist, wurde XMI gezielt auf die Unterstützung des Austauschs von MOF-Metadaten ausgelegt. Es ist unabhängig von Middleware-Technologien und setzt auch keine CORBA-Funktionalitäten voraus. Applikationen beliebiger Natur können per XMI Metadaten austauschen, sofern sie in der Lage sind, entsprechende Datenströme zu lesen und zu schreiben.

Wie der Name nahelegt, basiert XMI auf XML, der eXtensible Markup Language des W3C-Konsortiums [W3C00]. Im wesentlichen werden zwei Sätze von Produktionsregeln spezifiziert.

- Die *XML Document Production Rules* definieren die Kodierung von Metadaten in XML-Dokumente und umgekehrt die Rekonstruktion der Metadaten aus diesen Dokumenten.
- Die *XML DTD Production Rules* beschreiben die Erzeugung von XML Document Type Definitions, die als Syntax-Spezifikationen für die eigentlichen Datendokumente dienen. Mit Hilfe der DTDs können generische XML-Werkzeuge zur Verarbeitung der Dokumente genutzt werden.

Mit Blick auf MOF bedeutet dies, dass MOF-Metamodelle auf DTDs abgebildet werden, während aus MOF-Metadaten XML-Dokumente entstehen.

Zusammenhang mit CWM

Im Rahmen von CWM wird XMI als Format für den Datenaustausch genutzt, wobei aufgrund des gemeinsamen Hintergrunds MOF keine Erweiterungen der Spezifikation notwendig sind. Aus dem CWM-Metamodell wird eine Standard-DTD erstellt, mit deren Hilfe Warehouse-Metadaten in XMI-Dokumente kodiert werden können. Auch das CWM-Metamodell selbst kann in ein XMI-Dokument kodiert werden, indem die MOF-DTD als Syntaxspezifikation herangezogen wird.

2.3 Modellierung

Bei der Entwicklung von CWM wurden drei zentrale Ziele bezüglich des Designs verfolgt:

- Konzepte aus dem UML-Metamodell sollten weitestmöglich wiederverwendet werden, um sich die für dessen Entwicklung getätigten Anstrengungen zu Nutze zu machen und das Verständnis von CWM zu erleichtern. Dies wurde erreicht, indem das zentrale CWM-Package *Object Model* auf dem UML-Metamodell basierend angelegt wurde. Die weiterführenden CWM-Packages erweitern Object Model und profitieren so ebenfalls von den zugrundeliegenden UML-Konzepten.
- CWM sollte modular aufgebaut sein, um die Einarbeitung zu erleichtern und Implementierungen zu ermöglichen, die sich auf Teile des Modells beschränken. Um dies zu erreichen, wurde CWM in Packages aufgeteilt, die hierarchisch in mehreren Schichten aufeinander aufbauen.
- Das fertige Modell sollte implementierungsunabhängig sein, ohne dabei in zu große Distanz zur Praxis zu geraten. Hierzu wurden Untersuchungen an vorhandenen Implementierungen von Data Warehouses und Werkzeugen durchgeführt, um diejenigen Elemente herauszuarbeiten, die von allgemeiner Relevanz sind und in das Modell integriert werden sollten.

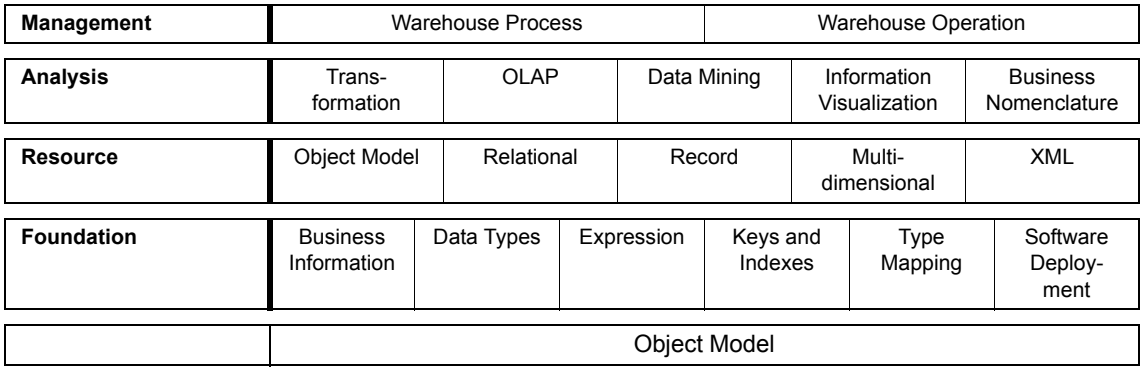
Beim Entwurf gingen die Entwickler von sechs Benutzergruppen aus. Die Bedürfnisse dieser Gruppen gingen ebenfalls in den Entwicklungsprozess ein und beeinflussten Funktionalität und Aufbau des Modells.

- Hersteller von Warehouse-Plattformen und Werkzeugen: Sie sind an der Kompatibilität ihrer Produkte mit denen anderer Hersteller interessiert und bemühen sich daher um Industriestandards und ihre Einhaltung.
- Dienstleister: Sie konfigurieren und betreiben Data Warehouses für ihre Kunden und streben dabei nach Wiederverwendung von Komponenten, einfachen Konfigurationsmöglichkeiten und leichter Integration verschiedener Werkzeuge und Datenquellen.
- Warehouse-Entwickler: Sie entwickeln Module für Data Warehouses und benötigen dafür unterstützende Werkzeuge sowie Informationen über Daten und Schnittstellen.
- Warehouse-Administratoren: Sie verwalten und pflegen in Betrieb befindliche Data Warehouses und müssen dabei Datenquellen und Werkzeuge integrieren, Ressourcen konfigurieren und Datenbestände verknüpfen.
- Endbenutzer: Sie verwenden Data Warehouses zur Informationsgewinnung und sollen möglichst umfassend durch Angaben über Strukturen und Vorgänge unterstützt werden.
- IT-Manager: Sie müssen für ihre Entscheidungen über den Entwicklungs- und Benutzungszustand der Data Warehouses informiert werden.

2.3.1 Architektur

Im Stapel der Meta-Ebenen des MOF-Frameworks ist CWM auf Meta-Ebene M2 einzuordnen. Es definiert ein Metamodell für Data-Warehouse-Modelle und eine UML-artige Notation für die Spezifikation von Data-Warehouse-Metadaten. Abbildung 2.2 zeigt den Aufbau des CWM-Metamodells.

Abbildung 2.2 Aufbau des CWM-Metamodells



Die Grobstruktur von CWM wird von vier schichtartig aufeinander aufbauenden Packages gebildet. Jedes dieser Packages enthält mehrere Subpackages; als Basis des gesamten Modells dient das Package *Object Model*. Die möglichst häufige Wiederverwendung der Konstrukte dieses Package und anderer Modellierungskonstrukte stellt ein wichtiges Konzept des Designs dar. So

wird beispielsweise Object Model auch für die Modellierung objektorientierter Datenquellen im *Resource*-Package genutzt.

Die einzelnen Packages enthalten Klassen und Assoziationen, die den entsprechenden Teilaspekt von CWM beschreiben. Dabei werden in den Klassen keine Operationen, sondern nur Attribute und Referenzen definiert. In eventuellen Erweiterungen von CWM sind jedoch auch Operationen in Klassen erlaubt.

Assoziationen können durch Vererbung oder Ableitung wiederverwendet werden. Vererbung stellt eine Wiederverwendung ohne Modifizierung der Eigenschaften der Assoziation dar. Abgeleitete Assoziationen hingegen werden gegenüber ihrer Vorlage durch OCL-Statements eingeschränkt. Neben Assoziationen kommen in CWM auch Referenzen zum Einsatz, die in der referenzierenden Klasse als Attribut vom Typ der referenzierten Klasse dargestellt, praktisch aber über Assoziationen implementiert werden. Gegebenenfalls kann die Navigierbarkeit von Assoziationen eingeschränkt werden, was typischerweise nur bei der Überschreitung von Package-Grenzen der Fall ist.

Constraints können strukturell oder textuell angegeben werden. Strukturelle Constraints befinden sich in Form von Multiplizitätsangaben, Vererbungsbeziehungen und ähnlichen Details im Modell selbst, während textuelle Constraints explizit in OCL formuliert werden.

2.3.2 Object Model

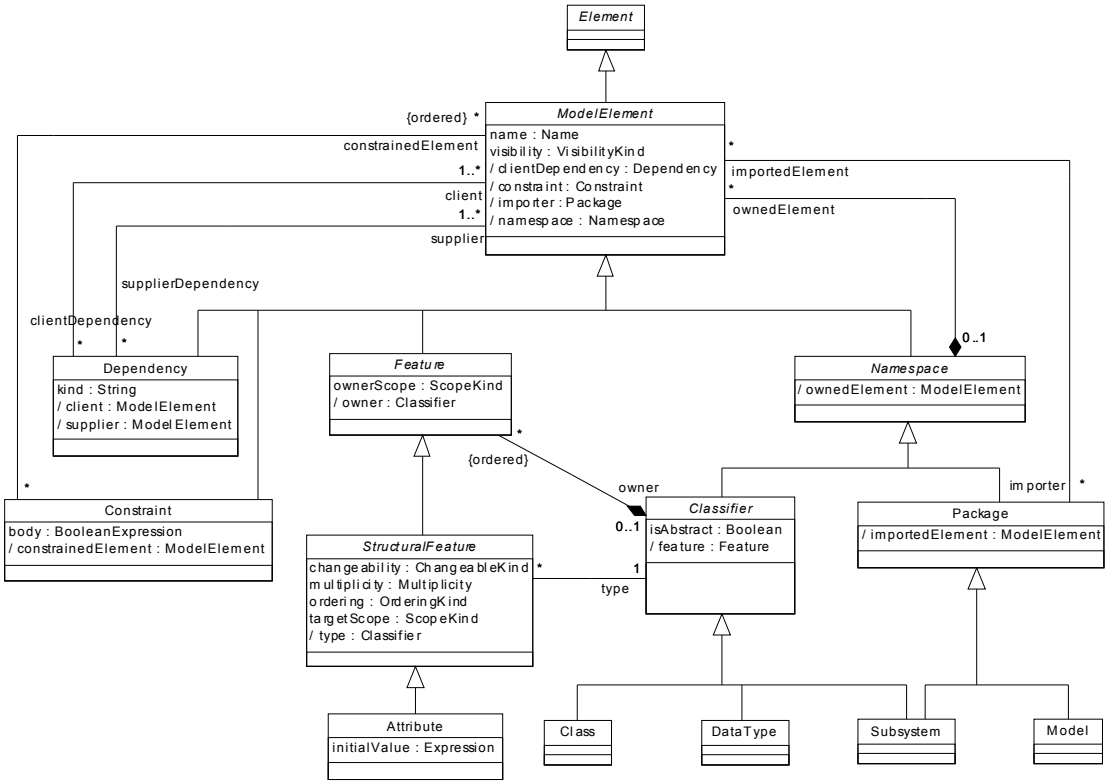
Im Package *Object Model* werden die grundlegenden Konstrukte für die Handhabung von Metamodell-Klassen in allen anderen Packages definiert. Es besteht aus einer auf das nötige reduzierten Teilmenge von UML und enthält die vier Subpackages *Core*, *Behavioral*, *Relationships* und *Instance*. Core hängt als einziges Package von keinem anderen Package ab und stellt so den Kern der CWM-Spezifikation dar. Behavioral, Relationships und Instance sind untereinander unabhängig und hängen jeweils nur von Core ab. Die Struktur des Core-Package ist aus Abbildung 2.3 ersichtlich.

Das Package *Core* spezifiziert neben einigen abstrakten Klassen an der Spitze der Vererbungshierarchie auch primitive Datentypen wie Boolean, String oder Integer. Mit der Klasse *ModelElement* findet sich hier die oberste Abstraktionsebene der Modellierungshierarchie, von der alle Modellierungs-Metaklassen in CWM erben. Weiterhin werden auf abstrakter Ebene Konstrukte zur Beschreibung objektorientierter Systeme definiert, etwa *Classifier* und *Feature*, deren Referenz *owner/feature* zwischen den konkreten Subklassen *Class* und *Attribute* ihre bekannte Bedeutung annimmt. *Namespace* und *Package* dienen über *ownedElement* und *importedElement* der Strukturierung von *ModelElement*-Instanzen. Schließlich werden mit *Dependency* und *Constraint* semantische Beziehungen zwischen Elementen beschrieben.

Im Package *Behavioral* werden Klassen und Assoziationen für die Beschreibung des Verhaltens von Elementen und den Aufruf vordefinierter Abläufe definiert. Wichtige Klassen dieses Package sind *Parameter*, *Operation* und *Method*.

Das Package *Relationships* enthält die Definitionen der beiden Arten von Beziehungen, die zwischen Elementen bestehen können: *Assoziation* und *Generalisierung*. Assoziationen können einfache Verknüpfungen oder Aggregationsbeziehungen zwischen Elementen sein. Sie verbinden zwei oder mehr Instanzen von Klassen. Generalisierungsbeziehungen zwischen Klassen führen zur Bildung von Typhierarchien, bei denen Klassen die Struktur- und Verhaltenseigenschaften

Abbildung 2.3 Struktur des *Core*-Package



ihrer übergeordneten Superklassen übernehmen und spezialisieren. CWM erlaubt Mehrfachvererbung, so dass Klassen die Vereinigungsmenge der Eigenschaften mehrerer direkter Superklassen übernehmen können.

Das Package *Instance* schafft die Möglichkeit, neben den üblichen Metadaten auch spezifische Dateninstanzen über CWM zu modellieren und auszutauschen. Hier werden Klassen wie *Instance*, *Object* und *DataValue* definiert.

2.3.3 Foundation

Im Package *Foundation* werden Konstrukte spezifiziert, die verschiedentlich in anderen Packages verwendet werden. Im Gegensatz zu Object Model sind die in Foundation definierten Konstrukte spezifisch für CWM. Dennoch sind sie derart allgemein gehalten, dass sie in im Schichtenmodell höher angesiedelten Packages Verwendung finden können, indem sie nötigenfalls spezialisiert oder erweitert werden.

Das Package ist in sechs Subpackages aufgeteilt, die untereinander unabhängig sind:

- *Business Information* dient zur Modellierung von Dokumentbeschreibungen sowie Verantwortungsträgern und ihren Kontaktdaten. Die wichtigsten Klassen dieses Subpackage sind *ResponsibleParty*, *Document*, *Description* und *Contact*.
- *Data Types* enthält Konstrukte zur Kreierung spezifischer Datentypen wie *Union* oder *Enumeration*.
- *Expressions* erlaubt die Konstruktion von Ausdruckbäumen für die einheitliche Darstellung von Termen. Für die Bäume wurde eine funktionale Darstellung von Termen gewählt.

- *KeysIndexes* definiert grundlegende Konzepte für die Identifizierung von Instanzen und die Anlage von Indexstrukturen. Die wichtigsten Klassen sind hier *UniqueKey* und *KeyRelationship*, die unter anderem die Grundlage für Primär- und Fremdschlüssel im *Relational*-Package darstellen.

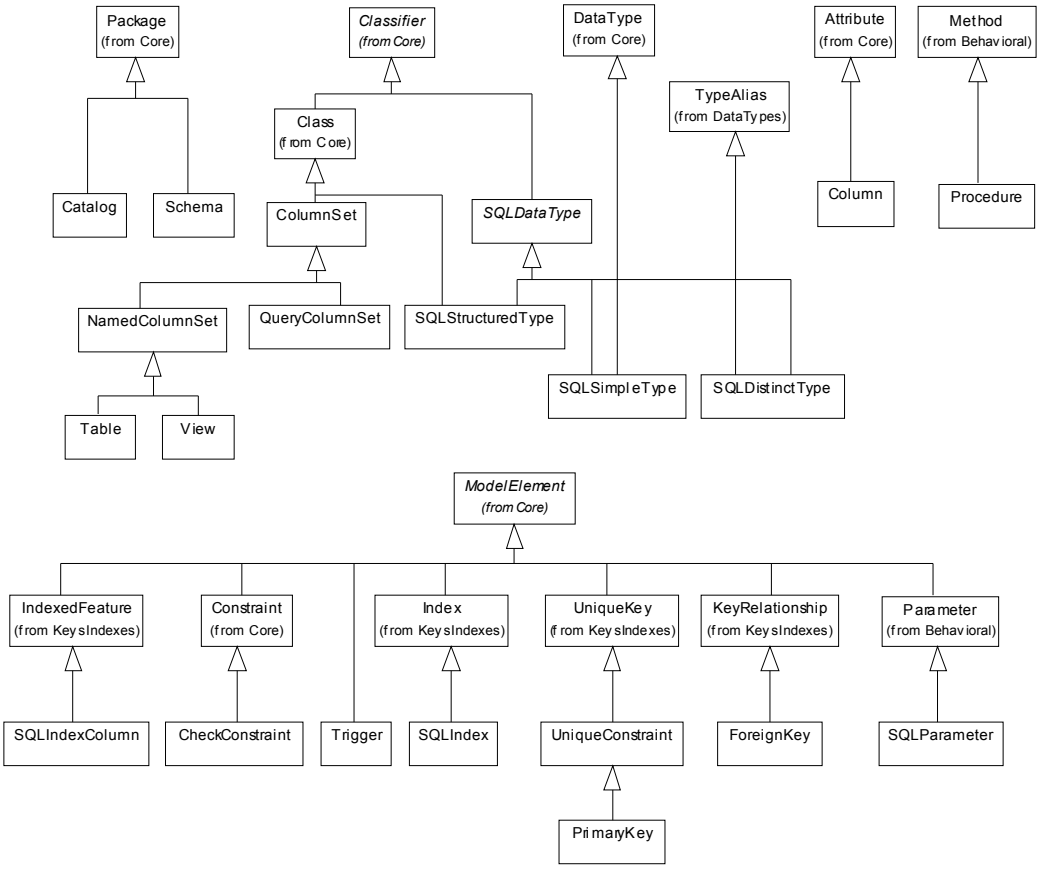
- *SoftwareDeployment* bietet Konstrukte zur Modellierung der Software-Organisation in einem Data Warehouse. Das Subpackage definiert Klassen wie *Machine*, *SoftwareSystem* und *Component*.

- *TypeMapping* erlaubt die Definition von Abbildungen zwischen Datentypen verschiedener Domänen. Anwendungsgebiet ist vor allem die Verbindung ähnlicher Datentypen auf verschiedenen Systemplattformen.

2.3.4 Relational

Das Package *Relational* dient zur Modellierung von relationalen Datenquellen, die in der Praxis typischerweise in Form von relationalen Datenbanksystemen auftreten. Dementsprechend stützen sich weite Teile des Package auf den SQL-Standard [ANS99]. Innerhalb von CWM baut das Package neben *Object Model* auf *DataTypes* und *KeysIndexes* aus Foundation auf. Abbildung 2.4 zeigt die genauen Abhängigkeiten sowie einen Überblick über die Package-interne Vererbungshierarchie.

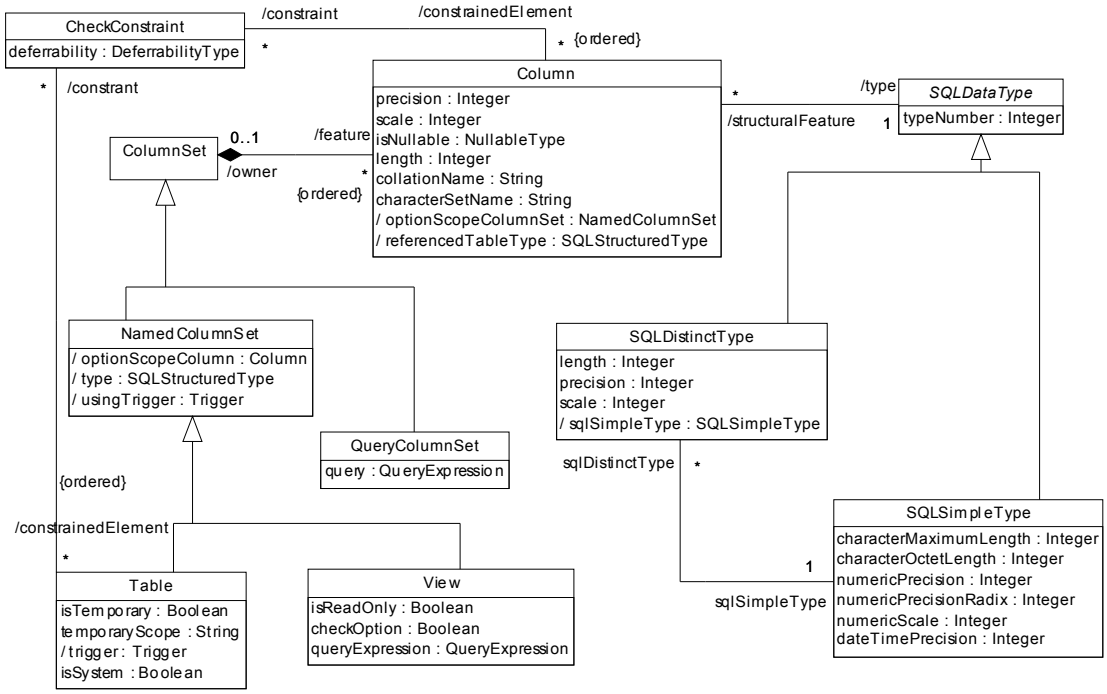
Abbildung 2.4 Vererbungshierarchie des Package *Relational*



Aufbau

Als Top-Level-Container wird *Catalog* definiert. Ein Catalog repräsentiert eine gesamte Datenquelle, etwa ein Datenbanksystem. Er enthält potentiell mehrere *Schemas*, die ihrerseits *Tables*, *Procedures* und *Trigger* enthalten. Tables sind Spezialfälle benannter Spaltenmengen (*NamedColumnSets*), sie enthalten Kompositionen von *Columns*, wobei jeder Spalte ein Datentyp zugeordnet ist. *Catalog* und *Schema* sind Subklassen der Klasse *Package* aus dem Core-Package. *ColumnSets* und damit insbesondere auch *Tables* erben von *Class*, *Columns* von *Attribute*, die zugeordneten Datentypen von *Classifier*. Die im Core-Package abstrakt definierten Beziehungen nehmen so im Relational-Package konkrete Gestalt an.

Abbildung 2.5 Beziehungen zwischen Table, Column und DataType

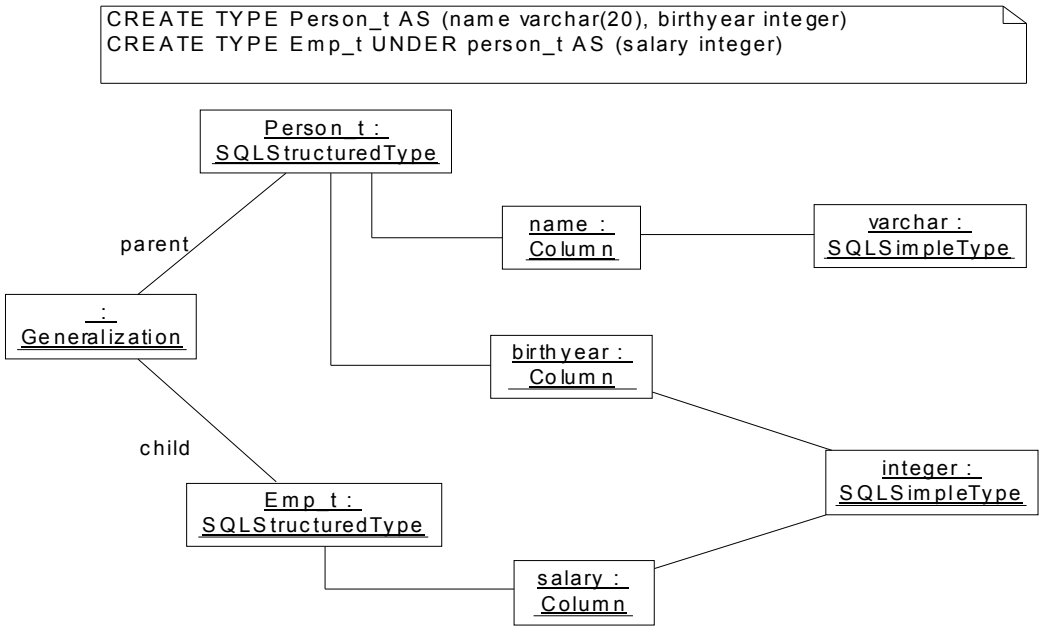


Strukturierte Datentypen

Abbildung 2.5 zeigt die Beziehungen zwischen Tabellen, Spalten und Datentypen. Neben den beiden direkt spezifizierten Datentypen *SQLSimpleType* und *SQLDistinctType* können komplexe Datentypen definiert werden. Ein solcher *SQLStructuredType* entsteht durch die Zusammenfassung mehrerer Spalten. Strukturierte Datentypen erweitern relationale Datenquellen um Konzepte aus dem Gebiet der objektorientierten Modellierung. So wird auch eine Typhierarchie zwischen strukturierten Datentypen ermöglicht, die durch Generalisierungsbeziehungen zwischen den einzelnen Typdefinitionen beschrieben wird. Beispiel 2.1 demonstriert die Definition von strukturierten Datentypen unter Verwendung von Generalisierung.

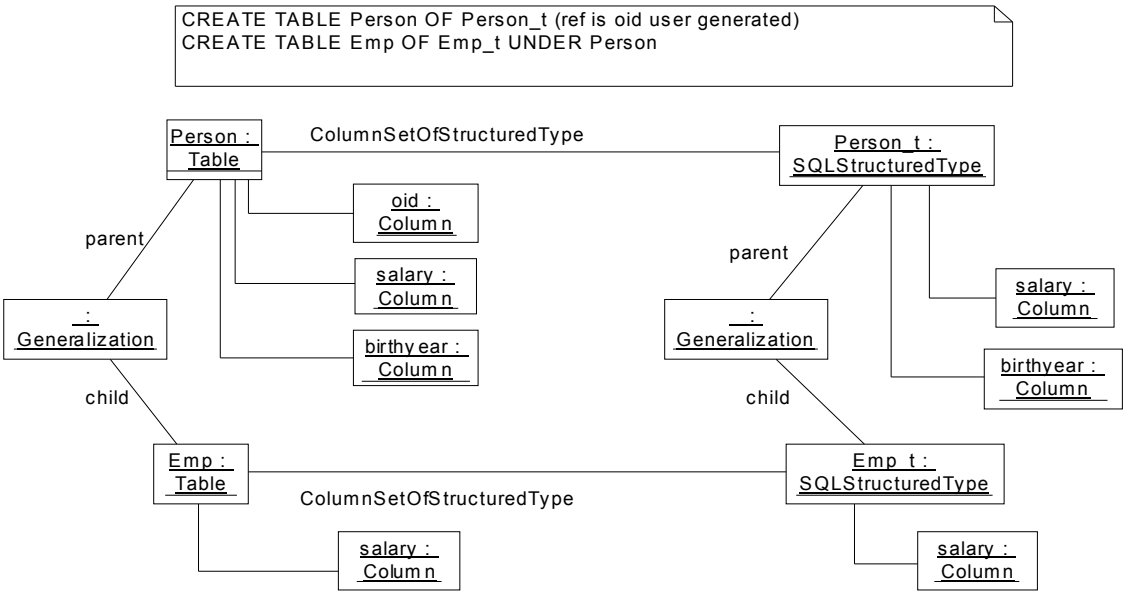
Strukturierte Datentypen können nicht nur als Datentypen für einzelne Spalten, sondern auch für ganze Tabellen verwendet werden. Dabei wird die Tabelle mit konventionellen Spalten erzeugt, die den strukturierten Datentyp nachbilden. Auf diese Weise können auch Applikationen auf die Tabelle zugreifen, die keine objektorientierten Erweiterungen unterstützen. Über eine Assozia-

Beispiel 2.1 Strukturierte Datentypen und Generalisierungsbeziehung



tion zwischen Tabelle und zugrundeliegendem strukturiertem Datentyp bleibt die Modellierung gewahrt. In Beispiel 2.2 werden aufbauend auf den in Beispiel 2.1 definierten strukturierten Datentypen entsprechende Tabellen definiert.

Beispiel 2.2 Tabellendefinitionen mit strukturierten Datentypen



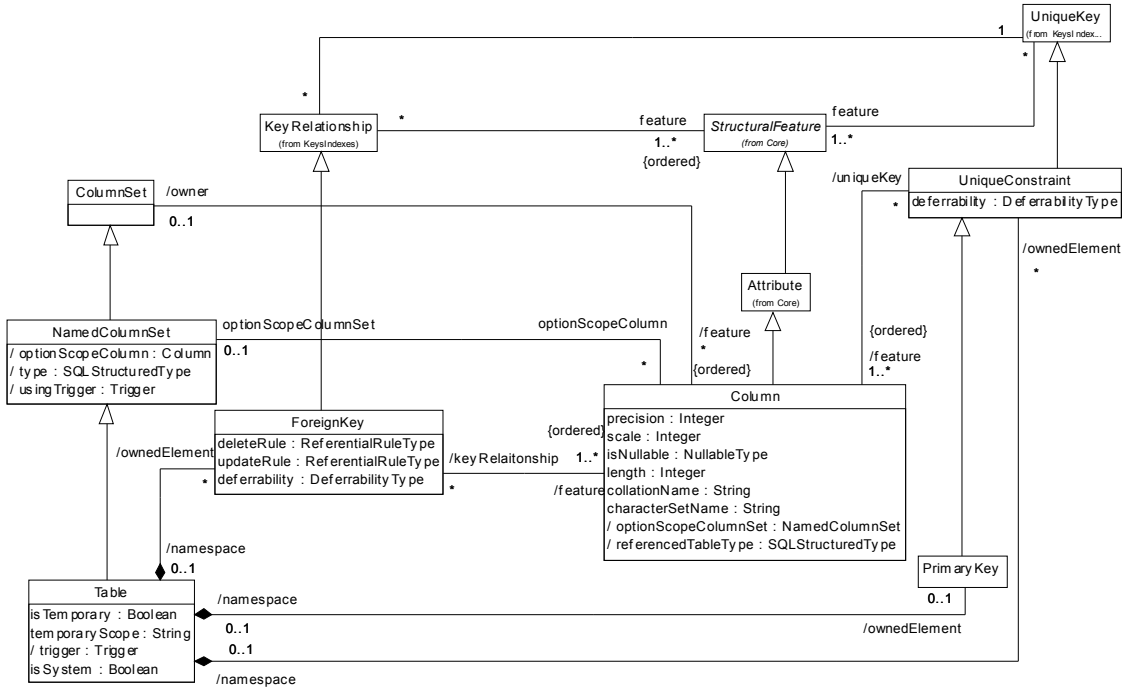
Primär- und Fremdschlüssel

Für die Beschreibung von Primär- und Fremdschlüsseln werden Klassen aus dem Subpackage *KeysIndexes* von Foundation verwendet, die entsprechend den Anforderungen der relationalen

Modellierung spezialisiert werden. Die Assoziationen zwischen den beteiligten Klassen werden jedoch durch Vererbung unverändert übernommen.

Ein *PrimaryKey* ist über die *namespace*-Referenz einer Tabelle zugeordnet. Über die *feature*-Referenz wird die Menge der Spalten definiert, die zusammen den Primärschlüssel bilden. Analog verfügt ein *ForeignKey* über Referenzen, die Tabellenzugehörigkeit und einbezogene Spalten festlegen. Über eine zusätzliche *keyRelationship*-Referenz wird ein Schlüsselkandidat einer anderen Tabelle referenziert. Diese Referenz stellt somit die eigentliche Verknüpfung zwischen den beiden beteiligten Tabellen her. Abbildung 2.6 zeigt die gesamten Beziehungen im Kontext von Primär- und Fremdschlüsseln.

Abbildung 2.6 Primär- und Fremdschlüssel



2.3.5 Weitere Packages

Als umfassendes Metamodell definiert CWM weitere Packages, um der Komplexität und Funktionalität von Data Warehouses gerecht zu werden. Im Rahmen dieser Arbeit sind diese weiterführenden Packages nicht von Belang, sie sollen hier daher nur kurz angerissen werden.

Resource

Im *Resource*-Package werden neben relationalen Datenquellen noch vier andere Arten von Datenquellen beschrieben. Für die Modellierung objektorientierter Datenquellen wird das *Object-Model*-Package wiederverwendet, das bereits näher betrachtet wurde.

Das *Record*-Subpackage beschreibt Record-Datenquellen. Dabei werden sowohl die klassische Datenhaltung in Dateien und Datenbanken als auch strukturierte Datentypen in Programmiersprachen berücksichtigt. Darüber hinaus kann dieses Subpackage erweitert werden, um beliebige

Informationsstrukturen zu beschreiben, die einem hierarchischem oder verschachtelten Aufbau genügen.

Das *Multidimensional*-Subpackage enthält eine generische Beschreibung multidimensionaler Datenquellen, wie sie beim Online Analytical Processing (OLAP) Verwendung finden. Vom Versuch eine genauen oder umfassenden Beschreibung derartiger Datenquellen wurde Abstand genommen, da auf diesem Gebiet nur wenige Standards existieren und entsprechende Datenbanken oftmals proprietäre Strukturen aufweisen.

Das *XML*-Subpackage enthält Klassen und Assoziationen zur Beschreibung von XML-Datenquellen. Es orientiert sich dabei an den Spezifikationen der W3C-Gruppe.

Analysis

Das *Analysis*-Package befasst sich mit der Analyse und Bewertung der in Data Warehouses gespeicherten Daten. Es enthält fünf Subpackages.

Im *Transformation*-Subpackage werden Metadaten für Transformationen definiert. Transformationen können zwischen beliebigen Datenquellen und Zielen ablaufen. Zu den Hauptaufgaben des Subpackage zählen die Verbindung zwischen Quelle, Transformation und Ziel, die Implementierung verschiedener Arten von Transformationen und die Bündelung von Transformationen in logischen Einheiten.

Das *OLAP*-Subpackage leistet Unterstützung für Funktionen des Online Analytical Processing. Dazu definiert es ein Metamodell der essentiellen und weit verbreiteten OLAP-Konzepte und eine Umgebung für Instanzen dieses Metamodells. Darüber hinaus bietet es Zugang zu den Funktionalitäten anderer CWM-Packages.

Das *Data-Mining*-Subpackage beinhaltet Klassen und Assoziationen für die Unterstützung von Data-Mining-Prozessen. Es definiert ein generisches Modell für Data-Mining-Algorithmen sowie die notwendigen Einstellungen und Attribute.

Im *Information-Visualization*-Subpackage werden Metadaten für die Visualisierung von Informationen beschrieben. Da die zu visualisierenden Informationen sehr verschiedener Natur sein können, beschränkt sich das Subpackage auf die Definition generischer Konstrukte.

Das *Business-Nomenclature*-Subpackage enthält Klassen und Assoziationen zur Repräsentierung von Geschäfts-Metadaten. Die beiden Hauptkonstrukte sind *Taxonomy*, die eine Sammlung von Konzepten für die Beschreibung eines Kontextes darstellt, sowie *Glossary*, eine Sammlung von semantisch verwandten Begriffen

Management

Im *Management*-Package werden auf hoher Ebene die Abläufe in Data Warehouses beschrieben. Das *Warehouse-Process*-Subpackage dokumentiert die komplexen Prozesse, die im Rahmen der Ausführung von Transformationen durchgeführt werden. Es erlaubt dabei verschiedene Detailstufen. Im Gegensatz dazu enthält das *Warehouse-Operation*-Subpackage Klassen für die Aufzeichnung alltäglicher Aktionen. Dazu gehören die Messung von Start und Ende von Transformationen oder die Protokollierung von Änderungen an Modellierungselementen.

Kapitel 3 CWM-basierter Schema-Entwurf

Im Rahmen dieser Arbeit wurde ein Werkzeug für die Modellierung von Datenbank-Schemas und die interaktive Nutzung von Datenbanksystemen entwickelt, wobei zwei Aspekten besondere Bedeutung zugemessen wurde: Das CWM-Metamodell sollte für die interne Speicherung und Verwaltung der modellierten Schemas verwendet werden, und die Bedienung des Werkzeugs sollte möglichst intuitiv erfolgen können. Bei der Entwicklung des intuitiven CWM-basierten SQL-Editors *SQLInteract* galt es daher unter anderem das CWM-Metamodell auf geeignete Weise in eine Java-Applikation zu integrieren. Die Benutzerschnittstelle wurde unter Einbeziehung von Konzepten aus dem Forschungsfeld der Mensch-Maschine-Kommunikation entworfen.

In diesem Kapitel werden der Aufbau und die besonderen Eigenschaften von SQLInteract vorgestellt. Die programminterne Ausrichtung auf das CWM-Metamodell ist von zentraler Bedeutung und beeinflusst auch die Darstellung modellierter Datenbank-Schemas gegenüber dem Benutzer sowie die Bedienung bei der Schema-Modellierung; dieser Effekt wurde bewusst in die Gestaltung des Editors miteinbezogen. Die in die verschiedenen Details der Benutzeroberfläche eingeflossenen HCI-Konzepte werden anhand ihrer Umsetzung in SQLInteract erläutert.

3.1 Übersicht der Benutzeroberfläche

Der Begriff der Mensch-Maschine-Kommunikation (*Human-Computer Interaction, HCI*) umfasst den Entwurf, die Implementierung und die Beurteilung von interaktiven Systemen und ihren Benutzerschnittstellen sowie die akademische Beschäftigung mit den Phänomenen im Spannungsfeld zwischen Mensch und Maschine [Shn92].

Unter einer Benutzerschnittstelle (*User Interface, UI*) versteht man alle Elemente eines Systems, über die der Benutzer Eingaben an das System tätigen kann oder über die das System Ausgaben an den Benutzer richten kann. Idealerweise wird eine Benutzerschnittstelle kaum als eigener Teil eines Systems wahrgenommen, sondern ist derart auf den Verwendungszweck und den Benutzerkreis zugeschnitten, dass eine Konzentration auf die bearbeiteten Inhalte möglich ist. Die Benutzerschnittstelle wird dadurch praktisch transparent, und der Benutzer interagiert auf eine ihm natürlich erscheinende Weise mit dem System [IBM03].

Aus der Perspektive der wissenschaftlichen Beschäftigung mit HCI kann man die Interaktion von Mensch und Maschine sehr abstrakt betrachten und grundsätzliche Fragestellungen erörtern. Hier soll auf diese Grundlagen jedoch nicht eingegangen werden. SQLInteract ist als Java-Programm

mit graphischer Benutzoberfläche (*Graphical User Interface, GUI*) konzipiert. Der Benutzer tätigt Eingaben über Maus und Tastatur, Ausgaben erfolgen auf dem Bildschirm. Der Dialog zwischen Benutzer und Applikation läuft in einem Hauptfenster und verschiedenen unterstützenden Dialogfeldern ab, die gemäß den Vorgaben des verwendeten Betriebssystems dargestellt werden und handzuhaben sind. Konzeptionelles Ziel von SQLInteract ist also nicht die Entwicklung völlig neuer Theorien für Interaktionsmethoden, sondern die innovative Kombination bekannter GUI-Elemente, um den Benutzer bei seiner Arbeit zu unterstützen.

Abbildung 3.1 Hauptfenster von SQLInteract

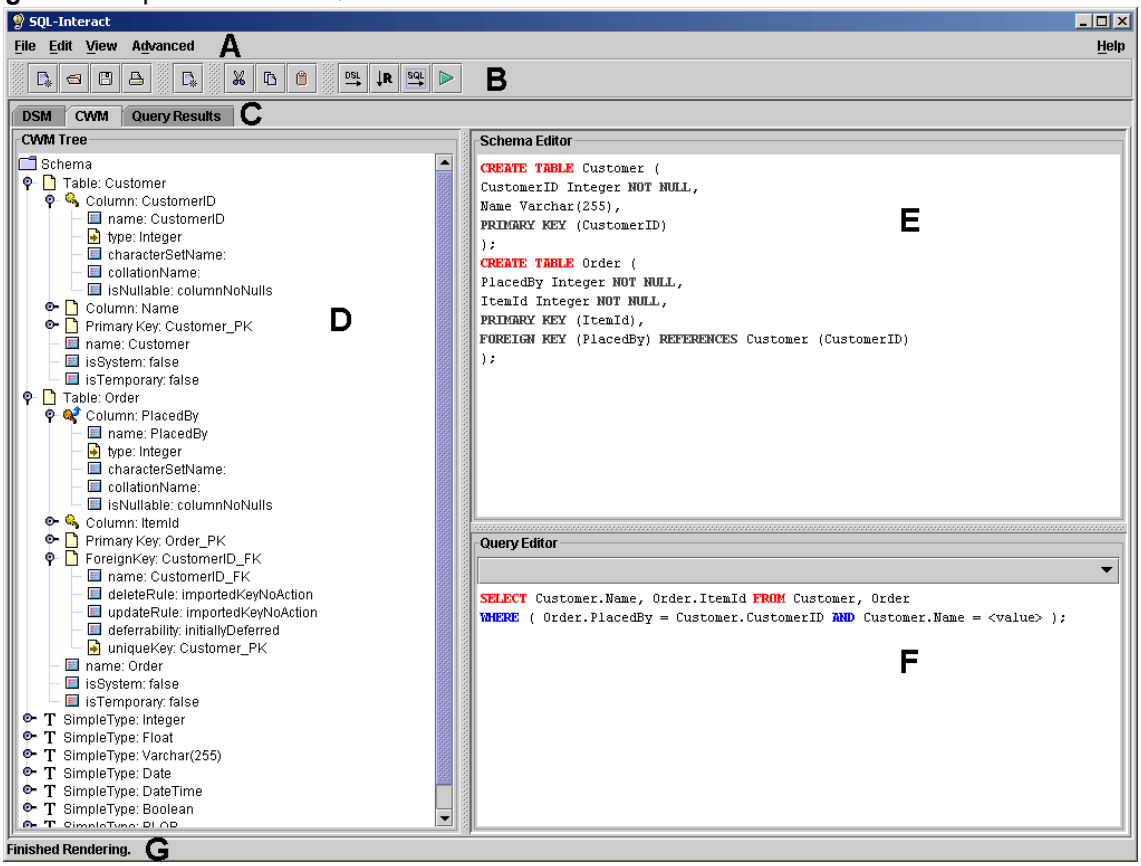


Abbildung 3.1 zeigt das Hauptfenster von SQLInteract mit seinen Komponenten. Eine Benutzeroberfläche sollte grundsätzlich eine größtmögliche Übersichtlichkeit aufweisen und so einfach und schlicht gestaltet sein, wie es die zu bedienende Funktionalität zulässt. Die Aufmerksamkeit des Benutzers soll zu den relevanten Elementen der Oberfläche geführt werden, ohne den Benutzer dabei zu stark in seinen Aktionen einzuschränken oder zu bevormunden. Um diese Ziele zu erreichen, ist die Oberfläche idealerweise an die unterliegende Funktionalität angepasst, ohne dabei einschlägige Konventionen zu verletzen, wie sie beispielsweise von Herstellern vorgegeben werden [Sun01a] oder sich im Laufe der Zeit etabliert haben. Bei der Aufteilung der Benutzeroberfläche von SQLInteract wurden derartige Konventionen berücksichtigt. Die Oberfläche lässt sich in drei Teile gliedern: Am oberen Rand befinden sich Menü- und Symbolleiste, die größte Fläche nehmen die Felder für CWM-Baum, Schema-Editor und Query-Editor ein, und den unteren Rand bildet eine Statusleiste. Im Folgenden werden die einzelnen Komponenten vorgestellt, bevor auf die zentralen Elemente ausführlich eingegangen wird.

3.1.1 Menüleiste und Symbolleiste

Die Menüleiste (A) bietet Zugriff auf globale Funktionen, die in verschiedenen Bedienzuständen sinnvoll ausgeführt werden können und typischerweise nicht der Manipulation einzelner Objekte dienen. Neben Standardfunktionen wie „Programm beenden“ oder „Informationen über das Programm anzeigen“ sind dies im Überblick:

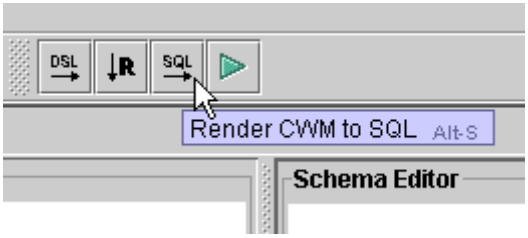
- *Import und Export von Datenbank-Schemas im XMI-Format*
  XMI ist das bevorzugte Format, um CWM-Modelle zu speichern und zu übertragen. Diese Menüpunkte erlauben Speichern modellierter Datenbank-Schemas als XMI-Datei und das Einlesen von Schemas aus XMI-Dateien.
- *Import und Export der SQL-Befehle aus dem Schema-Editor als Textdatei*
  Dieser Menüpunkt schreibt den Inhalt des Schema-Editor-Feldes in eine einfache Textdatei. Eine solche Textdatei kann beispielsweise von anderen Anwendungen als Befehlsskript an ein Datenbanksystem geschickt werden.
- *Aufruf des Dialogfensters zur Bearbeitung der Programm-Voreinstellungen*
  In den Voreinstellungen können die Parameter für eine Datenbank-Verbindung festgelegt werden. Über diese Verbindung können dann modellierte Schemas und Ad-hoc-Anfragen an ein Datenbanksystem geschickt werden.
- *Wahl von Darstellungsoptionen des CWM-Baums*
  Über diese Menüpunkte kann die Anzeige von Details der Elemente im CWM-Baum in mehreren Stufen eingestellt werden.
- *Ausschneiden, Kopieren und Einfügen von Textfragmenten in Schema- und Anfrage-Editor*
  Dies sind klassische Textoperationen, wie sie die meisten Anwendungen bieten.
- *Anzeigen der Struktur der Befehle in Schema- und Anfrage-Editor*
  Der textuellen Darstellung der SQL-Befehle liegt intern eine hierarchische Struktur zugrunde, die über diesen Menüpunkt als Baum angezeigt werden kann.
- *Öffnen einer Konsole für die direkte Kommunikation mit einem Datenbanksystem*
  Über diesen Menüpunkt lässt sich eine einfache Konsole öffnen, die die Eingabe und Ausführung beliebiger SQL-Befehle ohne Unterstützung der Editierung ermöglicht.

Die Symbolleiste (B) ist in mehrere Gruppen von Symbolen unterteilt, die sich individuell anordnen lassen und auch aus der Symbolleiste herausgezogen und als kleine Werkzeug-Fenster dargestellt werden können. Die Gruppierung inhaltlich verwandter Funktionen zusammen mit einer sinnvollen Anordnung der Bedienelemente trägt entscheidend zur Übersichtlichkeit einer Benutzerschnittstelle bei; auch die über Menüs anwählbaren Funktionen sind daher nach inhaltlichen Kriterien gruppiert. Die standardmäßige Positionierung der Menü- und Symbolleiste sowie die Anordnung der einzelnen Menüpunkte und Symbole entspricht etablierten Konventionen bei der Gestaltung von Benutzeroberflächen: Von links nach rechts sind zunächst globale Dateioperationen, dann Editierungsfunktionen und schließlich weitere anwendungsspezifische Funktionen zu finden. Die Symbolleiste ermöglicht einen alternativen Zugriff auf die am häufigsten benötigten Funktionen aus der Menüleiste; darüber hinaus sind die verschiedenen Funktionen auch über Tastaturbefehle, so genannte *Shortcuts*, zu erreichen.

Die Bereitstellung mehrerer Möglichkeiten zur Auslösung einer Funktion erhöht den Bedienkomfort erheblich, da die Wahl der bevorzugten Bedienmethode dem Benutzer überlassen bleibt.

Der Benutzer sollte bei der Bedienung einer Anwendung stets über Alternativen verfügen. Neben der Berücksichtigung verschiedener persönlicher Präferenzen werden auf diese Weise vor allem Benutzer verschiedener Erfahrungsstufen in die Lage versetzt, die Anwendung auf die für sie optimale Weise zu bedienen. Grundsätzlich fällt es Menschen leichter Dinge wiederzuerkennen, als sich frei zu erinnern. Ungeübte Benutzer können daher die Funktionen aus den Menüs aufrufen, wo sie über aussagekräftige Menüpunkte wiederzuerkennen sind. Alternativ steht die Symbolleiste zur Verfügung, in der die Funktionen anhand ihrer Symbole zu identifizieren sind. Die Symbole können außerdem einen Hinweis darauf geben, was für Aktionen eine Funktion auslöst. Verharrt der Benutzer mit dem Mauszeiger über einem Funktionssymbol, wird eine kurze Erklärung der Funktion eingeblendet, der so genannte *Tool Tip*. Wie Abbildung 3.2 zeigt, wird dabei auch das Tastaturkommando der zugehörigen Funktion angegeben.

Abbildung 3.2 Symbolleiste mit Tool Tip



Bietet eine Anwendung mehrere alternative Bedienmethoden, besteht ein wichtiger Aspekt in der Gleichwertigkeit der verschiedenen Methoden; der Benutzer muss also über alle Methoden die gleichen Aktionen auslösen können. Bei SQLInteract ist diese Gleichwertigkeit gegeben: Unabhängig von der Ansteuerung über Menü, Symbolleiste oder Tastaturbefehl wird intern dieselbe Funktion aufgerufen. Durch gleichwertige Bedienmethoden wird es dem Benutzer ermöglicht, längere Zeit mit der Anwendung zu arbeiten, ohne die Bedienmethode zu wechseln. Besonders zeitraubend wäre hier der Wechsel zwischen Tastatur und Maus.

3.1.2 CWM-Baum, Schema-Editor und Query-Editor

Diese drei Felder nehmen den größten Teil der Arbeitsfläche ein und bilden die Einstiegspunkte zur zentralen Funktionalität von SQLInteract. Ihre Größenverhältnisse lassen sich durch den Benutzer an seine Bedürfnisse anpassen. Im CWM-Baum (D) kann der Benutzer ein Datenbank-Schema modellieren. Das Modell des Schemas wird intern in Konstrukten des CWM-Metamodells verwaltet und gegenüber dem Benutzer in Baumform dargestellt. Aus dem Modell kann SQLInteract SQL-Code (*Data Definition Language*) generieren, der das modellierte Schema in einem Datenbanksystem anlegt. Abschnitt 3.2 führt die Bedienung und die besonderen Eigenschaften des CWM-Baums näher aus.

Im Schema-Editor (E) kann der generierte Code nachbearbeitet und es können neue Befehle eingegeben werden. Die textuelle Darstellung im Schema-Editor und das Modell des Datenbank-Schemas im CWM-Baum werden dabei synchronisiert, so dass Änderungen an einer Seite auf die jeweils andere Seite propagiert werden. Abschnitt 3.3 erläutert die Möglichkeiten bei der Schema-Editierung ausführlich.

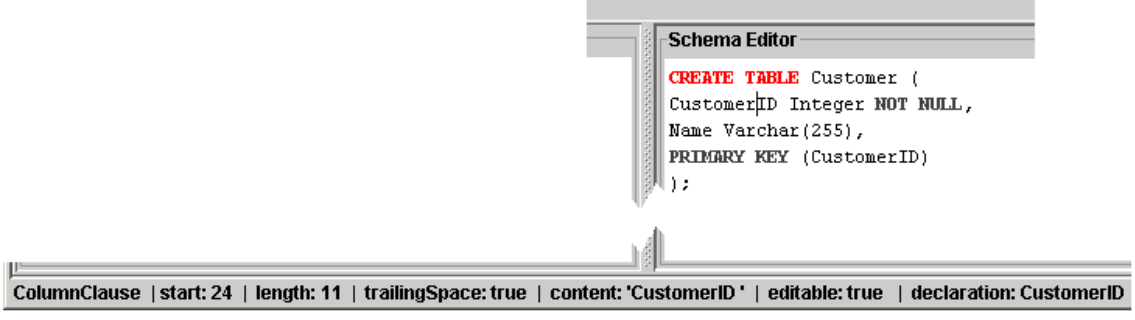
Der Query-Editor (F) erlaubt die Eingabe von Ad-hoc-Anfragen an ein Datenbanksystem. Die Eingabe der Anfragen wird durch eine Reihe von unterstützenden Funktionen erleichtert, wobei für die Bereitstellung dieser Funktionen intern auf das modellierte Schema zurückgegriffen wird. Die Anfragen können über eine vom Benutzer zu spezifizierende Verbindung an ein Datenbanksystem gesendet werden, woraufhin die Resultate in SQLInteract angezeigt werden. Abschnitt 3.4 beschreibt den Query-Editor im Detail.

3.1.3 Registertabs und Statusleiste

Registertabs (C) bieten einen Weg, die Bedienelemente einer Anwendung zu gliedern. Bei SQLInteract kann der Benutzer zwischen verschiedenen Arbeitsflächen umschalten. Die Modellierung von Datenbank-Schemas und die Formulierung von Ad-hoc-Anfragen an ein Datenbanksystem werden in der *CWM-Tab* durchgeführt; die Ergebnisse von Ad-hoc-Anfragen werden in der *Query-Results-Tab* dargestellt. Die Trennung von Eingabe einer Anfrage und Anzeige ihrer Resultate ist für sich betrachtet nicht dazu geeignet, die Übersichtlichkeit der Bedienung zu erhöhen. Durch ein weiteres Feld in der CWM-Tab wäre die Oberfläche dort jedoch überfrachtet, und insbesondere große Resultat-Mengen könnten nur ausschnittsweise dargestellt werden. Es wurde daher der Ansatz gewählt, nach dem Ansenden einer Anfrage durch den Benutzer automatisch auf die Query-Results-Tab umzuschalten, um die Resultate dort auf ausreichender Fläche präsentieren zu können.

Die Statusleiste (G) informiert den Benutzer über den Zustand der Anwendung und den Ablauf interner Vorgänge. Bei der Bearbeitung von SQL-Befehlen im Schema- oder Query-Editor werden beispielsweise Information zum unter der Eingabemarke befindlichen Befehlswort dargestellt (Abbildung 3.3); während der Generierung von SQL-Befehlen aus dem Modell wird der Verlauf des Prozesses angezeigt. Mithilfe der Statusleiste wird so der Anspruch erfüllt, den Benutzer jederzeit über seinen Arbeitskontext und den Zustand der Anwendung auf dem Laufenden zu halten. Diese Information ist für den Benutzer sehr wichtig, damit er nicht gezwungen ist, sich den aktuellen Arbeitskontext selbst zu merken. SQLInteract führt keine spontanen Zustands- oder Kontextwechsel durch, die extrem verwirrend wirken würden, und der aktuelle Arbeitskontext ergibt sich theoretisch bereits aus dem Zustand der sonstigen Elemente der Benutzeroberfläche; die Statusleiste bietet jedoch durch ihre explizite Information gesteigerten Bedienkomfort.

Abbildung 3.3 Statusleiste bei der Editierung eines SQL-Befehls



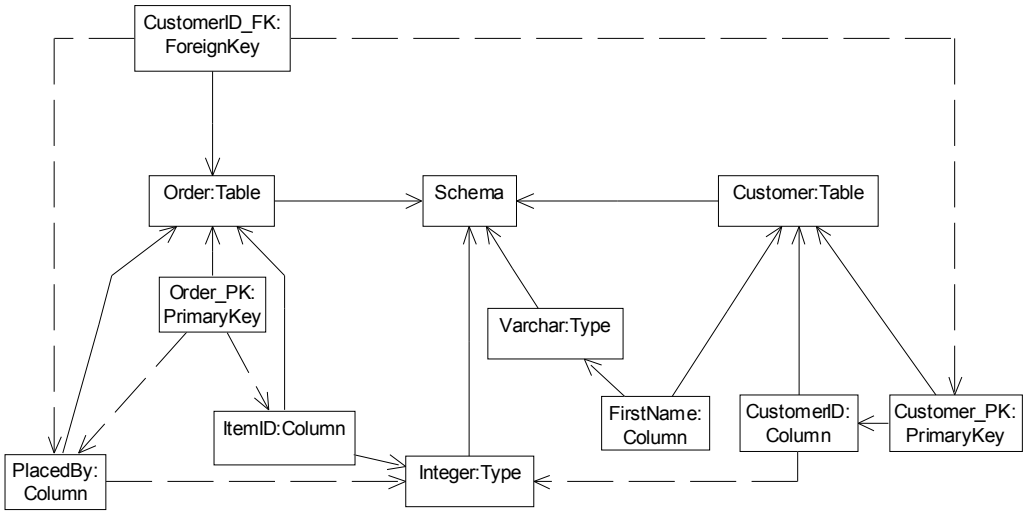
3.2 Schema-Modellierung

Die Modellierung von Datenbank-Schemas ist eine der wesentlichen Tätigkeiten, für die SQLInteract entwickelt wurde. Sie findet bevorzugt in der Baumdarstellung an der linken Seite des Hauptfensters statt, die als *CWM-Baum* bezeichnet wird. SQLInteract verwaltet die Modelle intern in einer für diesen Zweck angepassten Implementierung des CWM-Metamodells. Im Rahmen dieser Implementierung wurden die relevanten Klassen aus dem CWM-Metamodell mit ihren Attributen und Assoziationen als Java-Klassen realisiert.

3.2.1 Baumdarstellung von CWM-Modellen

CWM-Modelle sind Objektgraphen; eine Darstellung der modellierten Schemas als Graph ist für den Benutzer jedoch unübersichtlich und schwer zu handhaben. Für SQLInteract wurde daher die Darstellung der Schemas als Baumstruktur festgelegt. Aus dieser Entwurfsentscheidung ergab sich das Problem der Umsetzung eines Graphen in einen Baum. Wie Abbildung 3.4 an einem vereinfachten Beispiel zeigt, beinhalten CWM-Modelle als Objektgraphen eine beachtliche Anzahl an Zyklen, die für eine Baumdarstellung sinnvoll aufgelöst werden müssen. Entscheidende Bedeutung kommt dabei der Tatsache zu, dass Knoten in einem Baum höchstens einen Vater besitzen können. Von den verschiedenen Assoziationen eines Objektes kann also im Baum nur eine über die Assoziation zum Vater ausgedrückt werden.

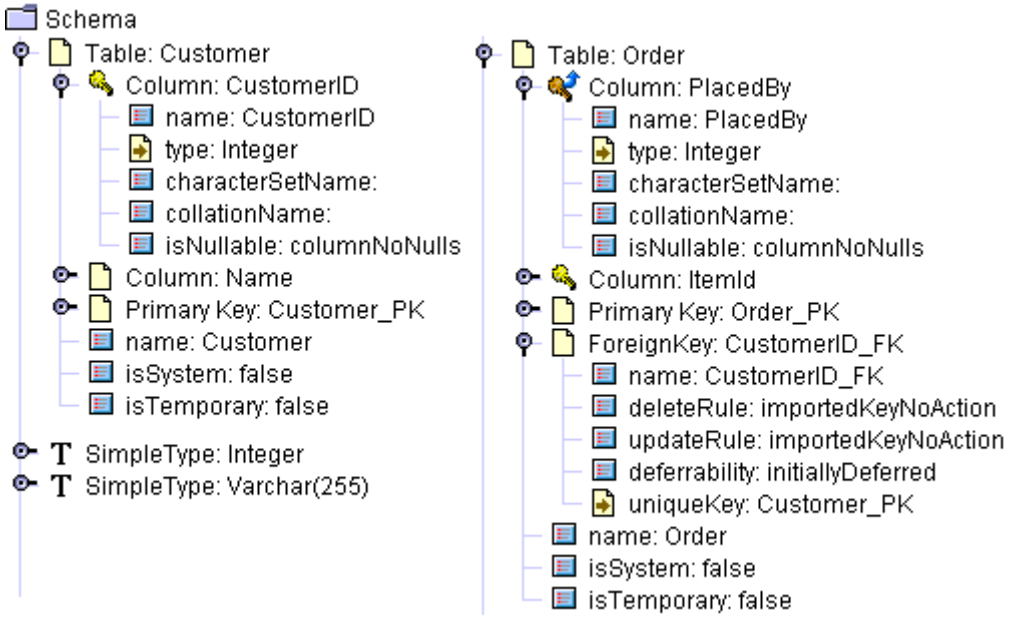
Abbildung 3.4 Vereinfachtes Beispiel für einen CWM-Objektgraphen



Aus Sicht von CWM sind alle Assoziationen gleichwertig, so dass die Wahl der über die Baumhierarchie zu modellierenden Assoziation eine reine Entwurfsentscheidung war. Für eine übersichtliche Darstellung galt es die Assoziation so zu wählen, dass einerseits der in der Baumdarstellung hinzukommende hierarchische Aspekt dem Verständnis des Benutzers entgegenkommt und andererseits die Anzahl der auf andere Weise darzustellenden Assoziationen minimiert wird. Unter diesen Gesichtspunkten bot sich die `ownedElement`-Assoziation an, die die Zugehörigkeit von Konstrukten zu übergeordneten Konstrukten modelliert. In Abbildung 3.4 sind `ownedElement`-Assoziationen mit durchgezogenen Linien symbolisiert, sonstige Assozi-

ationen mit gestrichelten Linien. Abbildung 3.5 zeigt die Darstellung dieses Datenbank-Schemas im CWM-Baum. Durch die Wahl von `ownedElement` als Assoziation für die Baumstruktur ergibt sich auch *Schema* als oberstes Element in der Hierarchie.

Abbildung 3.5 Baumdarstellung des Objektgraphen aus Abbildung 3.4

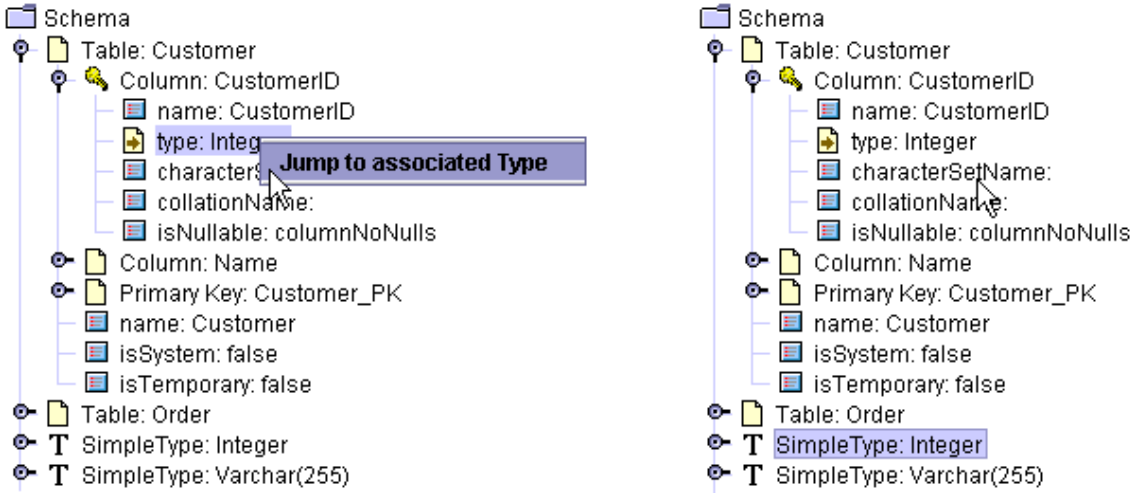


Aus Abbildung 3.5 sind außerdem die Symbole ersichtlich, mit denen die verschiedenen Modellkonstrukte veranschaulicht werden. Instanzen von CWM-Metaklassen erhalten grundsätzlich ein Instanzsymbol. Ist eine Spalte Teil des Primärschlüssels einer Tabelle, so wird das Instanzsymbol durch einen goldenen Schlüssel ersetzt; ist eine Spalte Teil eines Fremdschlüssels, wird sie mit einem dunklen Schüssel mit Pfeil gekennzeichnet. Tritt der Fall ein, dass eine Spalte sowohl Teil des Primär- als auch eines Fremdschlüssels ist, erhält sie folglich einen goldenen Schlüssel mit Pfeil. Attribute von CWM-Instanzen werden im Baum als Söhne der entsprechenden Instanz dargestellt und erhalten ein eigenes Symbol. Auch Datentypen sind anhand eines eigenen Symbols zu erkennen.

LinkNodes

Diejenigen Assoziationen, die nicht über die Baumhierarchie modelliert werden können, werden mithilfe so genannter *LinkNodes* visualisiert. LinkNodes erhalten ein Instanzsymbol mit einem zusätzlichen Pfeil und werden im Baum wie Attribute als Sohn der assoziierenden Instanz dargestellt. Durch einen Doppelklick auf einen LinkNode oder die Anwahl einer entsprechenden Funktion im Kontextmenü des LinkNodes wird die assoziierte Instanz im Baum selektiert. Abbildung 3.6 zeigt dies anhand der Assoziation zwischen einer Spalte und dem zugehörigen Datentyp. Sollte sich die assoziierte Instanz nicht im sichtbaren Bereich des Baums befinden, wird der sichtbare Ausschnitt des Baumes entsprechend verschoben. LinkNodes ermöglichen so eine verhältnismäßig übersichtliche Navigation im Baum entlang der Assoziationen zwischen Modellklassen.

Abbildung 3.6 LinkNodes im Baum



Um die Baumdarstellung eines Objektgraphen zu ermöglichen, mussten auch die Generalisierungsbeziehungen zwischen den Klassen des Metamodells angepasst werden. Eine Visualisierung dieser Beziehungen im Baum ist praktisch nicht möglich, ohne jegliche Übersichtlichkeit zu opfern. Instanzen im Baum stehen daher für sich alleine; eventuelle von anderen Klassen geerbten Attribute werden repliziert. Im zugrunde liegenden Metamodell sind die Vererbungsbeziehungen jedoch vollständig implementiert.

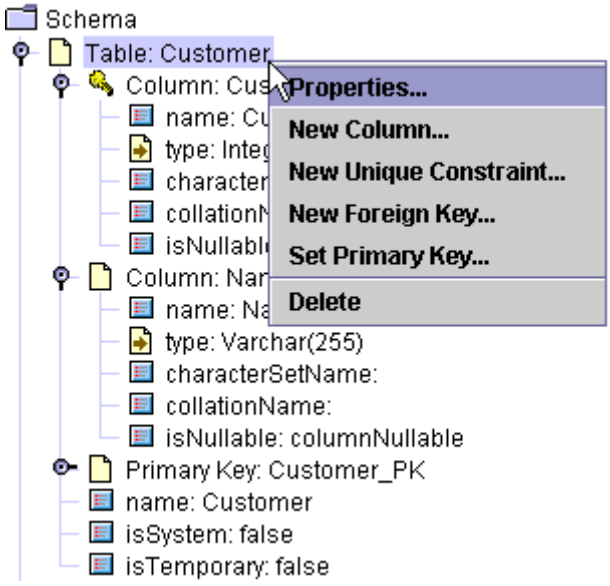
Neben den SQLInteract-spezifischen Besonderheiten wie LinkNodes und verschiedenen Symbolen für die enthaltenen Elemente bietet der CWM-Baum die übliche Funktionalität: Der Benutzer kann Zweige aufklappen oder zusammenfalten, und wenn der Baum größer ist als die in seinem Feld zur Verfügung stehende Fläche, kann über Scrollbalken am Rand der gewünschte Ausschnitt angefahren werden. Instanzknoten im Baum werden mit ihrer Klasse und ihrem individuellen Namen beschriftet, Attribute mit ihrem Namen und Wert. Einige in CWM definierte, in der Praxis der Schemamodellierung aber nur selten benötigten Attribute werden in der Voreinstellung nicht angezeigt; über einen Menübefehl können auch diese Attribute sichtbar gemacht werden. Ein weiterer Menüpunkt blendet zusätzlich die Datentypen aller Attribute ein.

3.2.2 Bedienung des CWM-Baums

Das eigentliche Anlegen und Bearbeiten eines Datenbank-Schemas erfolgt über Kontextmenüs der Elemente im Baum. Durch einen Klick mit der rechten Maustaste auf ein Element öffnet sich ein kontextsensitives Popup-Menü, das die Aktionen anbietet, die auf dem ausgewählten Element durchgeführt werden können. Das Element wird im Baum selektiert, um eindeutig kenntlich zu machen, auf welches Element sich die angebotenen Aktionen beziehen. Nur Instanzen von CWM-Metaklassen besitzen Kontextmenüs; versucht der Benutzer ein Kontextmenü eines Attributs aufzurufen, wird automatisch die das Attribut besitzende Instanz ausgewählt.

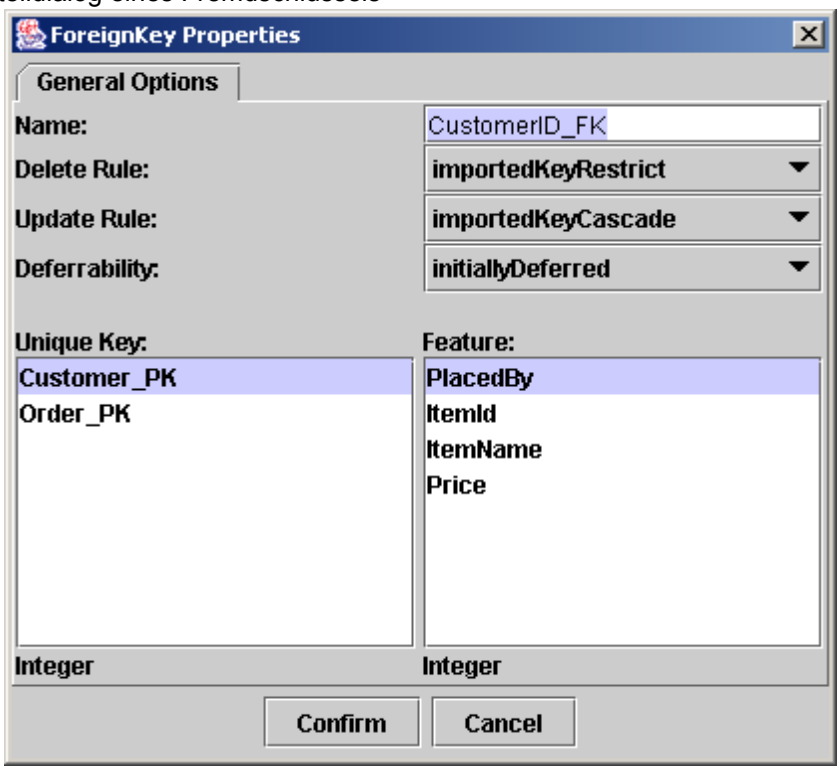
Die in einem Kontextmenü angebotenen Aktionen beinhalten immer das Löschen des Elements und die Anzeige seiner Eigenschaften. Kann das Element Sub-Elemente besitzen, werden zusätzlich Aktionen zum Anlegen neuer Sub-Elemente angeboten. Abbildung 3.7 zeigt dies am Beispiel einer Tabelle. Durch die Kontextsensitivität der Menüs erhöht sich die Übersichtlichkeit für den Benutzer, da nur tatsächliche anwendbare und sinnvolle Aktion zur Auswahl stehen. Beim

Abbildung 3.7 Popup-Menü einer Tabelle



Anlegen eines Elements oder durch späteren expliziten Aufruf werden die Eigenschaften des Elements in einem Einstelldialog präsentiert. Der Benutzer kann hier Änderungen vornehmen und bestätigen oder aber den Dialog schließen, ohne dass geänderte Werte in das Modell übernommen werden. In Abbildung 3.8 ist beispielhaft der Einstelldialog eines Fremdschlüssels dargestellt.

Abbildung 3.8 Einstelldialog eines Fremdschlüssels



Die Einstelldialoge sind gleichartig aufgebaut, soweit dies die unterschiedlichen Inhalte zulassen. Beim Aufruf ist stets das Textfeld für den Elementnamen aktiviert, so dass der Benutzer hier direkt eine Änderung vornehmen kann. Bei neu angelegten Elementen wird, wenn möglich, ein Default-Name vergeben, der sich aus dem vorhandenen Modell ableitet, etwa Tabellenname_PK bei der Definition eines Primärschlüssels. Wenn das Textfeld aktiv ist, schließt ein Druck auf die Eingabetaste den Dialog und übernimmt die Änderungen. Auch in den restlichen Teilen des Dialogs kann der Benutzer mit der Tastatur navigieren, was bei entsprechender Übung die Eingabe oder Änderung von Daten deutlich beschleunigt. Abgesehen vom Textfeld für den Namen wurde weitgehend auf freie Dateneingabe verzichtet. Stattdessen lassen sich Werte aus Kombo-Boxen und Listen auswählen, wodurch fehlerhafte Eingaben vermieden werden. Sofern es sich nicht um statische Optionen handelt, werden die angebotenen Alternativen beim Aufruf des Einstelldialogs aus dem Modell ermittelt. Im Beispiel des Fremdschlüssels sind etwa `Delete Rule` und `Update Rule` statische Optionen; es handelt sich hier um die referenziellen Constraints bei Fremdschlüsseln. Die beiden Listen `Unique Key` und `Feature` werden jedoch dynamisch bei jedem Aufruf gefüllt. Unter `Feature` werden die referenzierenden Spalten der aktuellen Tabelle, also die Teile des eigentlichen Fremdschlüssels, ausgewählt, unter `Unique Key` wird das Ziel der Referenz angegeben. CWM-spezifisch können hier nicht direkt Spalten angegeben werden, sondern das Ziel der Referenz ist ein so genannter *Unique Constraint*, ein Schlüsselkandidat. Derartige Unique Constraints kann der Benutzer selbst definieren; darüber hinaus sind die Primärschlüssel der Tabellen implizit auch Schlüsselkandidaten. Im konkreten Beispiel referenziert die Spalte `PlacedBy` der Tabelle `Order` den Primärschlüssel der Tabelle `Customer`.

Um den Benutzer vor Fehlern und inkonsistenten Modellen zu schützen, werden Eingaben auf Korrektheit und Plausibilität geprüft und gegebenenfalls durch Sicherheitsabfragen untermauert oder völlig abgelehnt. In Abbildung 3.9 wurde gegenüber Abbildung 3.8 eine neue referenzierende Spalte gewählt, deren Datentyp nicht mit dem referenzierten Primärschlüssel übereinstimmt. Eine derartige Inkonsistenz kann zugelassen werden, da der Benutzer vielleicht im nächsten Schritt den Datentyp des Primärschlüssels ändern will. In diesem Fall ist die Entscheidungshoheit des Benutzers über die Reihenfolge seiner Arbeitsschritte anzuerkennen, so dass das System lediglich eine Sicherheitsabfrage präsentiert. Bis zur Behebung der Inkonsistenz werden die betroffenen Elemente im Baum farbig markiert, um auf die ausstehende Korrektur hinzuweisen.

In Abbildung 3.10 versucht der Benutzer jedoch, ein Element zu löschen, das noch von anderen Elementen referenziert wird. Derartige Löschaktionen können schnell das ganze Modell unbrauchbar machen, da dem System Informationen fehlen, um die auftretenden Inkonsistenzen aufzulösen. Der Benutzer muss also die Modellstruktur selbst ändern und erhält dafür zumindest die Angaben, an welcher Stelle genau die Probleme auftraten, die das Löschen verhinderten.

Abgesehen von solchen Sonderfällen steht dem Benutzer die Organisation seiner Arbeit frei. Natürlich lassen sich beispielsweise Fremdschlüssel erst definieren, wenn das Ziel der Referenz im Modell existiert. Solche Zwänge ergeben sich aus der Modellstruktur; SQLInteract als Anwendung zwingt dem Benutzer aber keine darüber hinausgehenden Einschränkungen auf. Die Modellierung eines Datenbankschemas ist eine Folge kleiner Einzelschritte, die von der Durchführung her nicht aneinander gekoppelt sind, sondern höchstens Abhängigkeiten von den bearbeiteten Daten aufweisen. Durch die unmittelbare Sichtbarkeit der Auswirkungen jeder Aktion im CWM-Baum erhält der Benutzer direkte Rückmeldung über seine Aktionen und lernt schnell, die Anwendung sicher zu bedienen. Das konsistente Bedienkonzept für die Manipulation der ver-

Abbildung 3.9 Sicherheitsabfrage bei zweifelhaften Eingaben

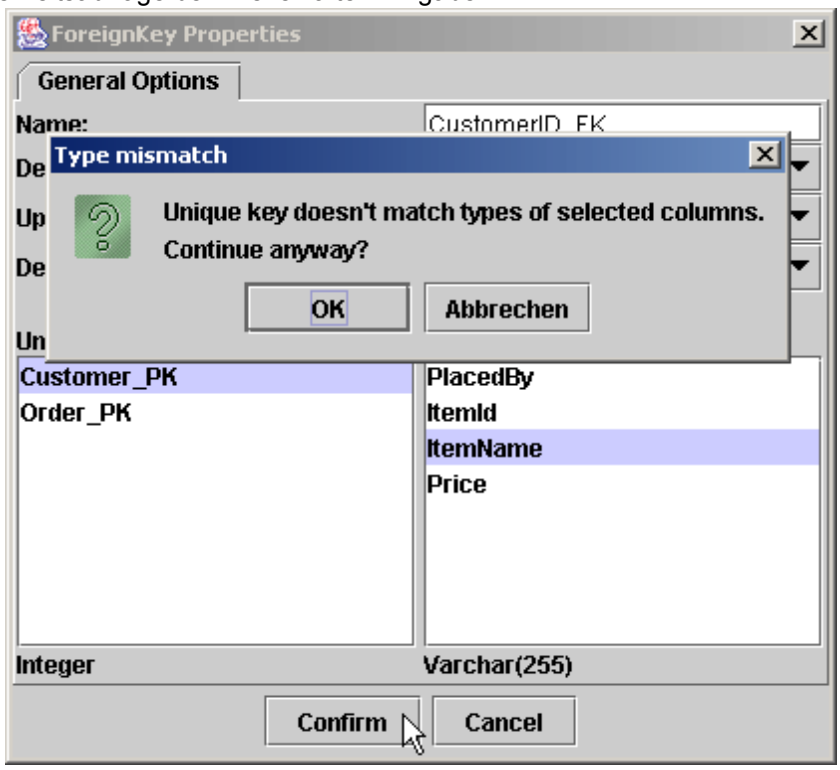
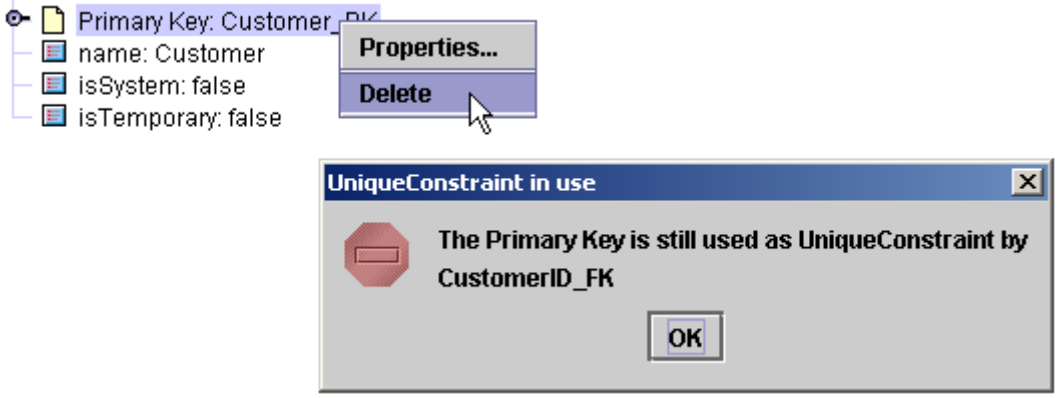


Abbildung 3.10 Löschsperre bei einem referenzierten Element



schiedenen Elemente erlaubt mit etwas Erfahrung außerdem, die Auswirkungen von Aktionen schon im Vorfeld abzuschätzen. In der Summe kann der Benutzer nach kurzer Einarbeitung effizient mit dem CWM-Baum arbeiten.

3.2.3 Generierung von SQL-Code

Aus einem im CWM-Baum modellierten Datenbank-Schema kann SQLInteract die nötigen DDL-Befehle generieren, um das Schema in einem Datenbanksystem anzulegen. Die Code-Generierung wird vom Benutzer über Menüpunkt, Symbolleiste oder Tastaturbefehl ausgelöst. Abbildung 3.1 zeigt ein Beispiel-Schema mit dem zugehörigen generierten Code. Der generierte Code wird im Schema-Editor angezeigt und kann mit den dort zur Verfügung stehenden Metho-

den weiterbearbeitet werden. CWM-Baum und Code sind synchronisiert; Änderungen auf einer Seite werden durch das System auf der jeweils anderen Seite nachvollzogen. Hier kommt wieder das Konzept der gleichwertigen Bedienmethoden zum Einsatz: Das Modell im CWM-Baum und die DDL-Befehle im Schema-Editor sind zwei verschiedene, gleich mächtige Darstellungsweisen derselben Inhalte. Dem Benutzer wird die Möglichkeit geboten, seine bevorzugte Methode für die Schema-Bearbeitung zu wählen. Oftmals bevorzugen erfahrene Benutzer tastaturlastige Bedienmethoden gegenüber mauslastigen Methoden, und bei komplexen Modellen kann die textuelle Darstellung eine bessere Übersicht bieten.

Intern beruht auch die textuelle Darstellung der SQL-Befehle auf einer eigens entwickelten Baumstruktur, die in Abschnitt 3.3 näher erläutert wird. Die Code-Generierung kann daher als Baumtransformation angesehen werden. Die zugrunde liegenden Regeln lassen sich mit einer geeigneten Spezifikationssprache abstrakt beschreiben; in SQLInteract werden Regeln allerdings nicht zur Laufzeit interpretiert, sondern als Methoden implementiert, um eine bessere Performanz zu erzielen.

3.3 Schema-Editierung

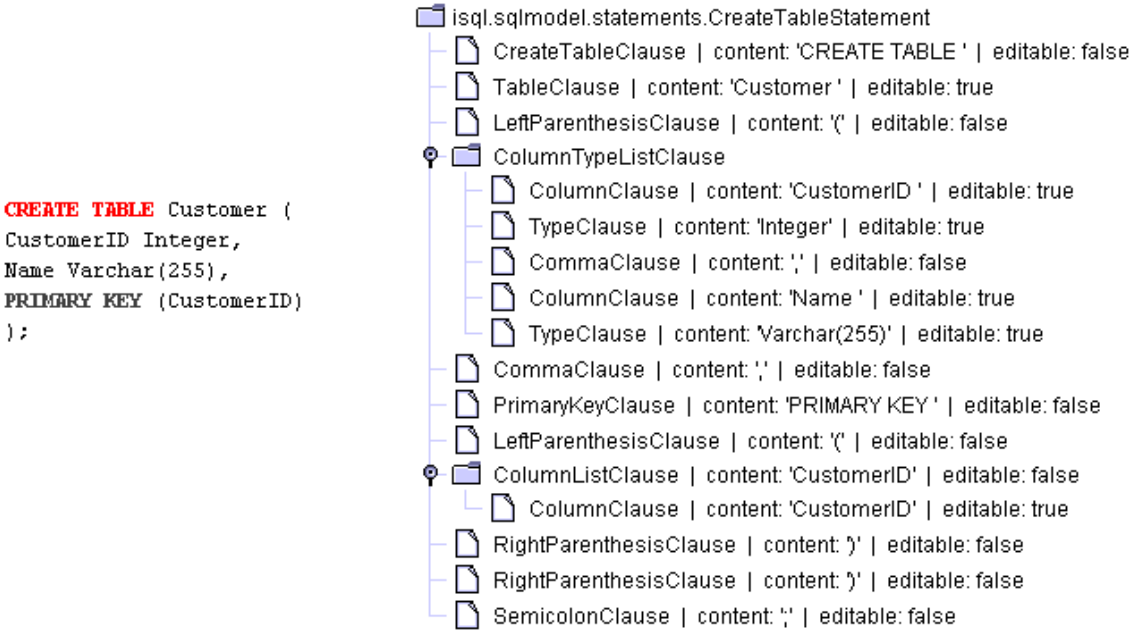
Aufgabe des Schema-Editors ist es, die Eingabe und Bearbeitung von SQL-Befehlen möglichst benutzerfreundlich und intuitiv zu gestalten. Gegenüber einem einfachen Texteditor verfügt der Schema-Editor daher über eine Reihe von Funktionen, die die Bedienung deutlich vereinfachen. Die Grundlage für die Bereitstellung dieser Funktionen wird von der internen Verwaltungsstruktur für die SQL-Befehle gebildet.

3.3.1 Interne Struktur

SQLInteract verwaltet die im Schema-Editor dargestellten Befehle in einer Baumstruktur; Abbildung 3.11 zeigt dies beispielhaft anhand eines `CREATE-TABLE`-Befehls. Die einzelnen Bestandteile eines Befehls werden als *Clauses* bezeichnet. Im Schema-Editor sichtbare Clauses sind grundsätzlich Blätter im Baum; innere Knoten treten als so genannte *ListClauses* an Stellen auf, wo im SQL-Befehl mehrere gleichartige Elemente aufeinander folgen. Im Beispiel ist dies bei der Definition der Tabellenspalten und des Primärschlüssels der Fall, auch wenn der Primärschlüssel aktuell nur eine Spalte umfasst. ListClauses verwalten das Hinzufügen und Entfernen der untergeordneten Elemente und abstrahieren gegenüber höheren Baumebenen von deren genauer Anzahl. Zur Darstellung der Clauses im Schema-Editor wird die Blattebene des Baums ausgelesen. Für die Synchronisation zwischen Schema-Editor und CWM-Baum halten Clauses Referenzen auf die zugehörigen Elemente im CWM-Baum.

Der Benutzer kann im Schema-Editor die SQL-Befehle nicht beliebig bearbeiten. Wie aus Abbildung 3.11 ersichtlich ist, sind einige der Clauses als nicht editierbar markiert. Befindet sich die Schreibmarke in einer solchen Clause, werden Texteingaben ignoriert. Innerhalb eines Befehls sind lediglich die Clauses zur Bearbeitung freigegeben, die Elemente des Schemas bezeichnen, etwa die Namen von Tabellen, Spalten und Datentypen. Die syntaktische Struktur der SQL-Befehle ist dem Editor also bekannt, und aufgrund der Bearbeitungssperren kann der Benutzer sie nicht verletzen. Der Vorteil dieses Ansatzes besteht darin, dass auf einen Befehls-Parser verzichtet werden kann, da zu jedem Zeitpunkt die Struktur des Befehls konsistent und die Bedeutung der einzelnen Clauses bekannt ist. Im Schema-Editor werden die Clauses zwar als

Abbildung 3.11 `CREATE-TABLE`-Befehl und zugehörige interne Struktur



zusammenhängender SQL-Befehl dargestellt, so dass der Benutzer die interne Baumstruktur normalerweise nicht zu sehen bekommt und sich auch nicht mit ihr befassen muss; der Text im Editor-Feld ist jedoch technisch gesehen nur die Visualisierung der internen Datenstruktur. Die Bearbeitungsmöglichkeiten des Benutzers ergeben sich ausschließlich aus dieser Struktur, und Eingaben werden innerhalb der Struktur unter Berücksichtigung der Syntaxregeln verarbeitet. Da die Darstellung im Editor-Feld jederzeit konsistent zur internen Struktur gehalten wird, wirkt der Schema-Editor trotzdem wie ein Text-Editor mit zusätzlicher Funktionalität. Die genauen Möglichkeiten bei der Eingabe von Befehlen erläutert der nächste Abschnitt.

3.3.2 Eingabe von DDL-Befehlen

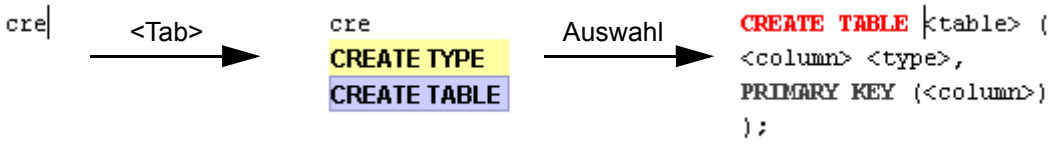
Neben der Bearbeitung von durch Generierung entstandenen Befehlen kann der Benutzer auch im leeren Editor-Feld beginnend DDL-Befehle manuell eingeben. Die Methoden für die Unterstützung des Benutzers bei seinen Eingaben gelten sinngemäß in beiden Fällen.

Vervollständigung von Eingaben

Beginnt der Benutzer eine Eingabe im leeren Editor-Feld, kann diese zunächst keinem bekannten SQL-Befehl zugeordnet werden. Der Benutzer kann jedoch nach Eingabe einiger Zeichen durch Druck auf die Tab-Taste eine Popup-Liste aller Befehle aufrufen, die mit dieser Zeichenfolge beginnen. Wählt er aus der Liste einen Eintrag aus, wird seine begonnene Eingabe zum Grundgerüst eines entsprechenden Befehls komplettiert, das aus uneditierbaren Schlüsselwörtern und sonstigen syntaktischen Elementen (etwa Klammern und Kommata) sowie Platzhaltern für noch einzugebende Bezeichner besteht (Abbildung 3.12). In der internen Verwaltungsstruktur wird gleichzeitig die bisher nicht zuzuordnende Zeichenfolge in eine Instanz des ausgewählten SQL-Befehls konvertiert.

Sowie das eingegebene Textfragment eindeutig als Beginn eines SQL-Befehls erkannt wird, komplettiert der Editor auch ohne Betätigung der Tab-Taste die Eingabe zu einem vollständigen

Abbildung 3.12 Vervollständigung einer begonnenen Eingabe



Befehlsgerüst. Hinter diesem Verhalten stehen die HCI-Konzepte der *Vorschläge* (*Suggestions)* und der *automatischen Vervollständigung* (*Auto Completion*). Ist eine Eingabe des Benutzers noch nicht abgeschlossen, aber bereits zweifelsfrei identifiziert, ergänzt die automatische Vervollständigung den fehlenden Teil, um dem Benutzer die unnötige Arbeit zu ersparen. Vorschläge an den Benutzer kann eine Anwendung aus der aktuellen Situation oder der Analyse der vorherigen Eingaben ableiten, wenn die aktuelle Eingabe keinen zweifelsfreien Schluss auf die Absichten des Benutzers erlaubt. In diesem Fall wird eine Liste der möglichen Alternativen zur Auswahl angeboten.

Strukturelle Erweiterung von Befehlen

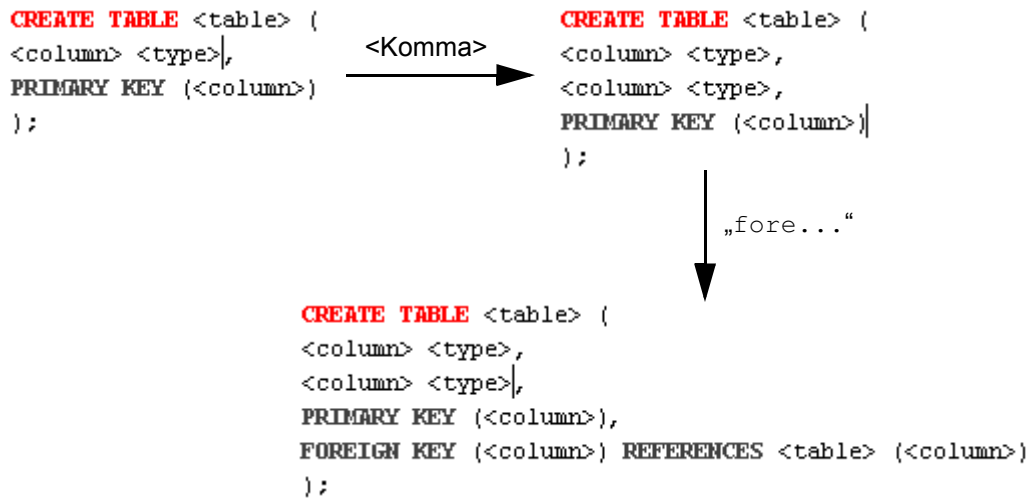
Nachdem der Editor das Grundgerüst eines SQL-Befehls aufgebaut hat, kann der Benutzer die Platzhalter mit Inhalten füllen. Sofern dies möglich ist, wird auch bei diesen Arbeitsschritten automatische Vervollständigung angeboten, etwa bei der Auswahl von Datentypen für Spalten, wo bereits definierte Datentypen gewählt werden können, oder bei der Definition des Primärschlüssels auf den Spalten der Tabelle.

Zusätzlich kann der Benutzer den Befehl *strukturell erweitern*. Das vom Editor vorgegebene Grundgerüst dient nur als Einstieg und beinhaltet die wichtigsten Elemente des Befehls. An Stellen, die die Aufzählung mehrerer Elemente erlauben, können durch einfaches Drücken der Komma-Taste weitere Platzhalter eingeblendet werden. Größere Erweiterungen der Befehlsstruktur werden ähnlich wie die Ersteingabe des Befehls durch Eingabe einiger Zeichen und automatische Vervollständigung vorgenommen. Bei einem `CREATE-TABLE`-Befehl können so beispielsweise weitere Spaltenplatzhalter über die Komma-Taste eingefügt werden, während sich eine FOREIGN-KEY-Klausel über die Eingabe ihrer ersten Zeichen an der entsprechenden Stelle im Befehl hinzufügen lässt (Abbildung 3.13). Der Editor vermerkt strukturelle Erweiterungen in der internen Verwaltungsstruktur, so dass der Benutzer sie wieder zurücknehmen kann, etwa wenn sie sich als unnötig erwiesen haben. Für diese Funktionalität kommt das HCI-Konzept des *Verlaufs* (*History*) zur Anwendung. Der Benutzer kann in der Abfolge seiner Arbeitsschritte zurückgehen, was in den Auswirkungen mit einer *Rückgängig*-(*Undo*)-Funktion zu vergleichen ist. Allgemein können Verlaufsaufzeichnungen außer für Vorwärts- und Rückwärtsnavigation zwischen Arbeitsschritten auch als Vorlage für die Wiederholung von mehrschrittigen Aktionen dienen.

Hervorhebung der Syntax

Aus Abbildung 3.13 ist zu ersehen, dass die SQL-Befehle bei ihrer textuellen Darstellung mit unterschiedlichen Schriftstilen ausgezeichnet werden. Die Auszeichnung der Schlüsselwörter und Platzhalter ergibt sich aus den Syntaxregeln für SQL-Befehle; konkret werden beispielsweise Befehlswörter in roter Fettschrift dargestellt, weitere Schlüsselwörter in dunkelgrauer Fettschrift und Platzhalter sowie an entsprechenden Stellen eingefügte Benutzereingaben in normaler

Abbildung 3.13 Strukturelle Erweiterung von SQL-Befehlen



schwarzer Schrift. Durch die verschiedenen Schriftstile ist die syntaktische Struktur der Befehle für den Benutzer leichter erkennbar, man spricht daher von *Hervorhebung der Syntax* oder *Syntax Highlighting*.

Syntax Highlighting ist ein bekanntes Konzept zur Erhöhung der Lesbarkeit von formalen Texten wie Quellcode oder Spezifikationen. Es ermöglicht insbesondere, syntaktische Fehler schneller zu erkennen. Typischerweise wird Syntax Highlighting allerdings rein lexikalisch anhand von Definitionsdateien mit Schlüsselwörtern realisiert; eine Berücksichtigung der tatsächlichen Code-Struktur beschränkt sich in aller Regel auf Klammern, Anführungszeichen und Kommentare, also leicht zu identifizierende Strukturen. Damit genügt ein solches Syntax-Highlighting streng genommen nicht seiner eigenen Bezeichnung, da die eigentliche Syntax nicht berücksichtigt wird. Wie Beispiel 3.1 verdeutlicht, werden viele syntaktische Fehler auf diese Weise nicht angezeigt, so dass der Nutzen des Syntax-Highlighting beschränkt bleibt.

Beispiel 3.1 Lexikalisches Syntax-Highlighting

```
public void foo() {}
```

Eine korrekte Methodendefinition in Java. Die Schlüsselwörter „public“ und „void“ werden in der Definitionsdatei gefunden und in diesem Fall mit Fettdruck ausgezeichnet.

```
public public foo() {}
```

Ein syntaktisch nicht korrektes Code-Fragment, das vom Compiler zurückgewiesen wird. Beide Vorkommnisse des Schlüsselworts „public“ werden im Rahmen des Syntax-Highlighting jedoch getrennt betrachtet und folglich mit Fettdruck ausgezeichnet. Für den Benutzer ist beim raschen Überlesen des Codes kein Fehler zu erkennen.

Für eine optimale Unterstützung des Benutzers wäre ein Syntax-Highlighting wünschenswert, das tatsächlich die Syntax der bearbeiteten Sprache berücksichtigt und damit alle syntaktischen Fehler anzeigen kann. Ein solches vollwertiges Syntax-Highlighting ließe sich jedoch nur über eine vollständige Implementierung der Grammatik der bearbeiteten Sprache in Verbindung mit

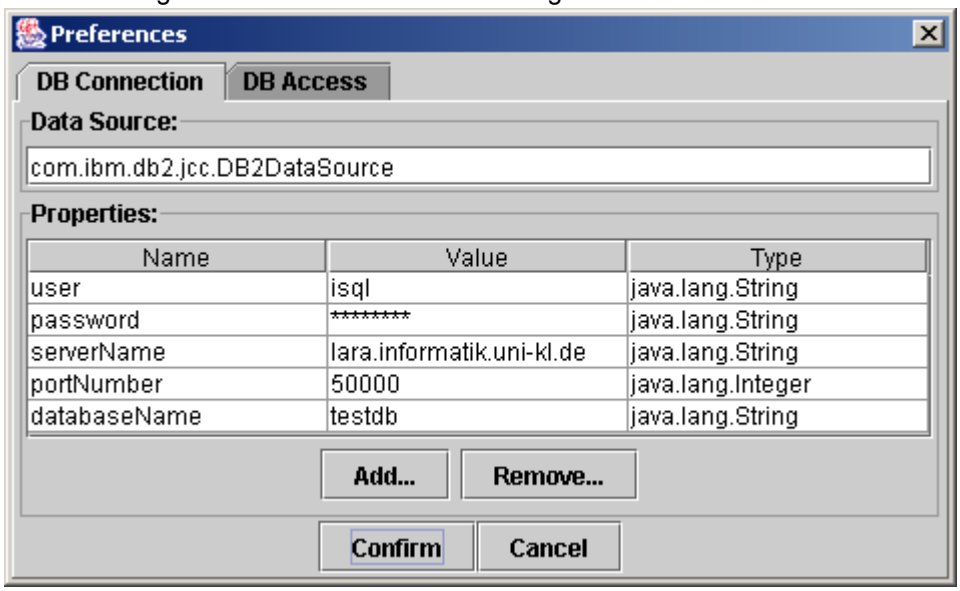
einem entsprechenden Parser realisieren. Neben dem schieren Entwicklungsaufwand dieses Ansatzes können beachtliche Kosten für die Lizenzierung der Sprachgrammatiken entstehen; hinzu kommen Performanzprobleme durch die Komplexität des Parsers.

Aufgrund der internen Baumstruktur für die Verwaltung der SQL-Befehle kommt SQL gänzlich ohne Definitionsdateien oder Parser aus. Da der Benutzer wie beschrieben die Befehle nicht frei bearbeiten kann, sondern immer den strukturellen Vorgaben des Editors folgen muss, sind die Befehle stets frei von syntaktischen Fehlern. Die Verwaltungsstruktur enthält außerdem die Regeln für die Schriftauszeichnung der einzelnen Befehlsteile. Für die Realisierung eines Syntax Highlighting genügt es unter diesen Voraussetzungen, bei der Darstellung der Befehle im Editor-Feld die Schriftstile tatsächlich entsprechend den Vorgaben zu zeichnen.

3.4 Anfrage-Editierung

Mithilfe des Query-Editors können Ad-hoc-Anfragen an ein Datenbanksystem gestellt werden. Der Benutzer kann zu diesem Zweck über die Menüleiste eine Datenbank-Verbindung spezifizieren (Abbildung 3.14). Nach Erstellung einer Anfrage im Editor-Feld kann sie an das Datenbanksystem gesendet werden. Eventuelle Resultate werden in einer eigenen Arbeitsfläche, der *Query Result Tab*, dargestellt, um auch bei großen Ergebnismengen eine übersichtliche Präsentation zu ermöglichen.

Abbildung 3.14 Voreinstellungen für die Datenbank-Verbindung



Die vom Query-Editor zur Verfügung gestellten Methoden der Eingabeunterstützung sind hinsichtlich der technischen Hintergründe mit denen des Schema-Editors identisch. Bei der Verwendung des Query-Editors wird jedoch davon ausgegangen, dass das Schema der verbundenen Datenbank dem im CWM-Baum modellierten Schema entspricht. Damit kann bei der Formulierung der Anfragen auf bestehende Schemainformationen in einem wesentlich größeren Umfang zurückgegriffen werden, als dies bei der Schema-Modellierung der Fall ist.

Beispiel 3.2 Eingabe eines `SELECT`-Befehls

Eingabe von 'sel' und Druck auf <Tab>:

```
SELECT <column> FROM <table>;
```

Erweiterung auf jeweils zwei Platzhalter für Tabellen und Spalten:

```
SELECT <column>, <column> FROM <table>, <table>;
```

Eingabe der Tabellennamen mithilfe der Vorschläge des Editors:

```
SELECT <column>, <column> FROM Customer, <table>;
```

Order
Customer

Eingabe der Spaltennamen, die nur aus den relevanten Tabellen vorgeschlagen werden:

```
SELECT Customer.CustomerID, <column> FROM Customer, Order;
```

*
Order.*
Order.PlacedBy
Order.ItemID
Customer.*
Customer.CustomerID
Customer.Name

Erweiterung um eine `WHERE`-Klausel durch Eingabe von '`whe`' am Ende der Zeile:

```
SELECT Customer.CustomerID, Order.ItemID FROM Customer, Order
WHERE <search condition>;
```

Nach Erweiterung auf zwei Suchbedingungen Auswahl des vom Editor vorgefertigt angebotenen Joins über die beteiligten Tabellen:

```
SELECT Customer.CustomerID, Order.ItemID FROM Customer, Order
WHERE ( <search condition> AND <search condition> );
```

Order.PlacedBy = Customer.CustomerID
<column> = <column>
<column> = <value>
(<search condition> AND <search condition>)
(<search condition> OR <search condition>)

Auswahl einer Spalte für den Vergleich mit einem Suchkriterium:

```
SELECT Customer.CustomerID, Order.ItemID FROM Customer, Order
WHERE ( Order.PlacedBy = Customer.CustomerID AND <column> = <value> );
```

Order.PlacedBy
Order.ItemID
Customer.CustomerID
Customer.Name

Die fertige Anfrage nach Eingabe des Suchkriteriums:

```
SELECT Customer.CustomerID, Order.ItemID FROM Customer, Order
WHERE ( Order.PlacedBy = Customer.CustomerID AND Customer.Name = 'Koch' );
```

Die Unterstützung des Benutzers durch den Query-Editor wird anhand von Beispiel 3.2 erläutert, das auf dem in Abbildung 3.1 dargestellten Datenbank-Schema beruht. Nach der Eingabe des Befehlsanfangs '`sel`' und Druck auf die Tab-Taste wird das Grundgerüst eines `SELECT`-Befehls dargestellt, das sich mit der geschilderten Methode auf jeweils zwei Platzhalter für Tabellen und Spalten erweitern lässt. Anders als bei der Schema-Bearbeitung kann der Editor bei der Eingabe der Tabellennamen Unterstützung leisten, indem er die im Schema vorhandenen Tabellen zur Auswahl anbietet. Gleiches gilt für die Spaltennamen. Sind im Befehl bereits Tabellennamen ausgewählt, beschränkt sich die Menge der vorgeschlagenen Spaltennamen auf die Spalten der entsprechenden Tabellen.

Durch Eingabe der ersten Zeichen '`whe`' wird der `SELECT`-Befehl um eine `WHERE`-Klausel erweitert. Die Suchbedingung einer `WHERE`-Klausel lässt sich über die per Tab-Taste aufzurufende Liste auf mehrere per AND oder OR verknüpfte Einzelbedingungen ausbauen. Sind mehrere Tabellen von dem `SELECT`-Befehl betroffen, bietet der Editor in dieser Liste auch ein Join-Kriterium über die Primärschlüssel an. Schließlich können konkrete Suchkriterien in der Form <Spalte> = <Wert> angegeben werden.

Hat der Benutzer bereits ein oder mehrere Zeichen in einem Platzhalter eingegeben, bevor er per Tab-Taste die zugehörige Auswahlliste aufruft, werden nur diejenigen Einträge tatsächlich angezeigt, deren Anfang mit den eingegebenen Zeichen übereinstimmt. Sollte eine Auswahlliste nur einen Eintrag enthalten, so wird beim Aufruf die Liste nicht erst dargestellt, sondern automatisch der einzige Eintrag im Befehl eingefügt.

Im Vergleich zum Schema-Editor kann der Query-Editor die HCI-Konzepte des Vorschlags und der automatischen Vervollständigung effektiver einsetzen, da verstärkt auf die Schemainformationen im CWM-Baum zurückgegriffen werden kann.

Kapitel 4 Möglichkeiten für die Modellierung von Semantik

Sowohl beim Entwurf von Datenbank-Schemas als auch bei der Erstellung von Ad-hoc-Anfragen formuliert der Benutzer seine Vorstellungen von Struktur und Verhalten einer Datenbank auf formale Weise. Anfragen liegen dabei direkt als DML-Befehle (*Data Manipulation Language*) vor; aus Schema-Modellen, die im CWM-Baum angelegt wurden, können automatisiert DDL-Befehle (*Data Definition Language*) generiert werden. In beiden Fällen wird die in den Befehlen ausgedrückte Funktionalität nach der Übermittlung an ein Datenbanksystem von diesem realisiert.

Dieses Kapitel erläutert zunächst das Konzept der modellgetriebenen Architekturen, bei denen die gesamte Funktionalität einer Anwendung zunächst unabhängig von der technischen Realisierung in einem Modell spezifiziert wird, um dann auf eine konkrete Zielplattform umgesetzt zu werden. Im Anschluss werden der Begriff der Semantik und die weitere Zielsetzung für SQLInteract näher ausgeführt. Abschließend werden verschiedene Ansätze für die abstrakte Spezifikation von Semantik vorgestellt und auf ihre Eignung für den Einsatz in SQLInteract untersucht.

4.1 Modellgetriebene Architekturen

Modellgetriebene Architekturen trennen die Spezifikation der eigentlichen Funktionalität einer Anwendung von den Details ihrer konkreten Realisierung. Die Anwendung wird dabei ohne Betrachtung technischer Realisierungsaspekte und unabhängig von einer Implementierungsplattform abstrakt in Form eines Modells entworfen. Dieses Modell wird dann, eventuell in mehreren Schritten, konkretisiert und unter Einbeziehung technischer Aspekte verfeinert, bis sich schließlich ausführbarer Code in einer konkreten Implementierungssprache daraus ableiten lässt.

Prominenter Vertreter der modellgetriebenen Architekturen ist die *Model Driven Architecture* (*MDA*) der Object Management Group (OMG). Sie wurde vor dem Hintergrund einer wachsenden Zahl von Middleware-Technologien entwickelt [Sol+00]. Nach Untersuchungen der OMG kommt oft bereits innerhalb eines Unternehmens mehr als eine Middleware-Technologie zum Einsatz, etwa aufgrund unterschiedlicher Anforderungen an die Middleware oder historisch bedingt. Spätestens im Geschäftsverkehr mit anderen Unternehmen stoßen verschiedene Technologien zusammen. Betrachtet man die Entwicklung auf dem Gebiet, zeichnet sich ab, dass die

parallele Existenz mehrerer Technologien langfristig anhalten wird, auch wenn ältere Vertreter hin und wieder durch Neuentwicklungen verdrängt werden.

Nimmt man den Zwang zur Berücksichtigung verschiedener Technologien mit ihren jeweils eigenen Plattformen und Beschreibungssprachen als gegeben hin, stellt sich die Frage nach einer möglichst effizienten Umsetzung der parallelen Implementierung von Geschäftsfunktionalität auf den einzelnen Zielplattformen. Die OMG schlägt hier einen Ansatz für die Systemspezifikation vor, bei dem die eigentliche Funktionalität getrennt von der Implementierung dieser Funktionalität auf einer bestimmten Technologieplattform spezifiziert wird. Die Spezifikationen werden dabei als Modelle ausgedrückt, und ein Funktionalität spezifizierendes Modell kann mithilfe von Abbildungsregeln auf verschiedenen Zielplattformen implementiert werden [OMG01a].

4.1.1 Grundlegende Begriffe

MDA definiert ein *Modell* als Repräsentation eines Ausschnitts der Funktionalität, der Struktur und/oder des Verhaltens eines Systems. Modelle können in graphischer oder textueller Form dargestellt werden; ihnen muss jedoch eine wohldefinierte Syntax und Semantik zugrunde liegen. Durch die formale Spezifikation von Semantik in Form von Modellen wird eine höhere Qualität bei der Entwicklung von Software erzielt. Modelle werden bevorzugt in UML erstellt, betrachtet und bearbeitet; so können sie mittels XMI übertragen und in MOF-Repositories gespeichert werden. Es ist aber auch möglich, UML-Erweiterungen wie *Profile* oder *Heavyweight Extensions* oder ein vollständig selbst definiertes MOF-konformes Metamodell zu verwenden. UML-Profile erlauben die Erweiterung der Semantik von UML innerhalb der Sprache durch *Stereotypes*. Ein Stereotype definiert beispielsweise Bedingungen (*Constraints*) oder Name-Wert-Paare (*Tagged Values*). Elemente in einem UML-Modell können mit Stereotypes ausgezeichnet werden und übernehmen so die in den Stereotypes definierten Eigenschaften. Auf diese Weise lassen sich die Elemente für besondere Einsatzzwecke spezialisieren. Im Gegensatz zu Profilen erweitern Heavyweight Extensions das UML-Metamodell und damit die Sprache selbst, indem neue Klassen in die Struktur des Metamodells eingefügt werden. Im direkten Vergleich besteht der Vorteil von Profilen in der breiten Unterstützung durch UML-Werkzeuge, da der eigentliche Sprachumfang nicht erweitert wird, während proprietäre Klassen von Heavyweight Extensions den Werkzeugen nicht bekannt sind.

Unter *Abstraktion* wird die Auslassung irrelevanter Details verstanden. Man spricht von einer *höheren Abstraktionsstufe*, wenn mehr Details ausgelassen werden, und entsprechend von einer *niedrigeren Abstraktionsstufe*, wenn weniger Details ausgelassen werden. Beschreiben zwei Modelle dieselben Inhalte auf unterschiedlichen Abstraktionsstufen, stehen sie in einer *Verfeinerungsrelation* zueinander.

Unter dem Begriff *Plattform* werden technische Details und Aspekte der Realisierung zusammengefasst, die für die eigentliche Funktionalität einer Komponente nicht von Bedeutung sind. Zu jeder Plattform existiert eine Spezifikation in Form eines Modells oder eines UML-Profils. Ein *plattformunabhängiges Modell* (*Platform-Independent Model, PIM*) beschreibt ein System unter Abstraktion von technischen Details, während ein *plattformspezifisches Modell* (*Platform-Specific Model, PSM*) auf solche Details Bezug nimmt und die Realisierung der im PIM spezifizierten Funktionalität auf einer bestimmten Zielplattform beschreibt. Ein PSM wird dabei in den Konstrukten der entsprechenden Plattform-Spezifikation ausgedrückt.

Platform-Independent Models

Der erste Schritt bei der Entwicklung eines Software-Produkts ist die Spezifikation der Funktionalität in einem PIM. Hintergrund dieser zunächst abstrakten Modellierung ist der Gedanke, dass Dienste oder Funktionalitäten in ihrer semantischen Reinform unabhängig von Fragen der technischen Umsetzung sind. Auf PIM-Ebene kann sich der Entwickler voll auf die Spezifikation der Funktionalität konzentrieren, und nach erfolgter Modellierung können Implementierungen auf verschiedenen Zielplattformen erzeugt werden, ohne den Kern der Funktionalität ändern zu müssen.

PIMs werden normalerweise in UML oder einer ähnlichen Spezifikationssprache modelliert und können dann sowohl graphisch als auch textuell dargestellt werden. UML bietet im Vergleich zu anderen Modellierungssprachen verhältnismäßig viele Möglichkeiten, die über reine Spezifikation struktureller Semantik hinausgehen. So können über die zu UML gehörende *Object Constraint Language* (*OCL*) etwa Invarianten oder Pre- und Post-Constraints formal definiert werden, um das geforderte Verhalten genauer festzulegen. Dieses Konzept wird *Design By Contract* [MM01] genannt. Durch die formale Festlegung von Constraints werden Mehrdeutigkeiten gegenüber informaler Beschreibung deutlich reduziert, und das genau definierte Vokabular der formalen Beschreibung lässt sich für automatisierte Verarbeitung nutzen.

Platform-specific Models

Nach der Fertigstellung des PIM wird die Funktionalität normalerweise zunächst in ein plattformspezifisches Modell umgesetzt, aus dem in einem späteren Schritt ausführbarer Code generiert wird. Dieses PSM ist mit den Konstrukten des UML-Profils der Zielplattform modelliert. Das Profil beschreibt die Abbildung der Elemente des PIM auf die konkreten technischen Eigenschaften der Zielplattform. Das PSM übernimmt so die funktionelle Semantik des PIM und fügt technische Semantik für die Implementierung hinzu. UML-Profile für Zielplattformen existieren beispielsweise für CORBA [OMG02b] oder, im Entwurfsstadium, für Enterprise JavaBeans [JCP01].

Für die Umsetzung von PIM nach PSM sind verschiedene Vorgehensweisen möglich [OMG01a]:

1. Die Umsetzung erfolgt vollständig manuell, wobei insbesondere die Verfeinerung vom Entwickler unter ausschließlicher Betrachtung des Einzelfalls durchgeführt wird.
2. Die Umsetzung erfolgt manuell unter Anwendung bekannter Standardverfeinerungen (*Refinement Patterns*) für typische Konstrukte, so dass möglichst wenig individuelle Verfeinerungsarbeit durchgeführt werden muss.
3. Ein Algorithmus generiert aus dem PIM ein PSM-Skelett, das vom Entwickler nachbearbeitet und komplettiert wird. Diese Methode lässt sich mit Standardverfeinerungen kombinieren, um den Anteil manueller Arbeit weiter zu senken.
4. Das gesamte PSM wird durch einen Algorithmus aus dem PIM generiert. Hierfür muss das PIM vollständig und frei von Mehrdeutigkeit spezifiziert sein; ebenso ist eine genau definierte und interpretierbare Verfeinerungsrelation zwischen den beiden Modellen vonnöten.

Die Vorgehensweisen beinhalten einen zunehmenden Automatisierungsgrad. Die vollständig automatisierte Umsetzung nach Methode 4 erfordert ein semantisch reiches Ausgangsmodell und sehr hochwertige Transformationsalgorithmen. In der Praxis wird sie daher nur in wenigen Fällen einsetzbar sein.

Abbildungen

Unter einer *Abbildung* wird in MDA eine Menge von Regeln und Techniken für die Transformation eines Modells in ein anderes Modell verstanden. Abbildungen sind ein zentrales Konzept in MDA. Neben den erwähnten Abbildungen auf eine niedrigere Abstraktionsebene im Zuge der Umsetzung von PIM nach PSM sind auch Abbildungen innerhalb einer Abstraktionsebene und auf eine höhere Abstraktionsebene möglich.

- PIM nach PIM
  Diese Abbildungsart wird verwendet, wenn ein plattformunabhängiges Modell bearbeitet wird. Solche Operationen sind typischerweise Verfeinerungen der Semantik, also Präzisierungen der Systemfunktionalität unabhängig von technischen Realisierungsdetails.
- PIM nach PSM
  Dies ist die oben beschriebene Umsetzung eines genügend ausdefinierten PIM in ein PSM.
- PSM nach PSM
  Transformationen innerhalb der PSM-Ebene sind ähnlich wie Transformationen innerhalb der PIM-Ebene typischerweise Verfeinerungen der Semantik, hier allerdings unter Bezugnahme auf die technischen Gegebenheiten der Zielplattform.
- PSM nach PIM
  Die Abbildung auf eine höhere Abstraktionsebene ist die wohl anspruchsvollste der vier Abbildungsarten. Hierbei geht es darum, aus existierenden Implementierungen abstrakte, plattformunabhängige Modelle zu extrahieren. Ein solcher *Mining*-Prozess muss also die eigentliche Funktionalität von plattformabhängigen Realisierungsdetails trennen; eine Aufgabe, die nur schwer zu automatisieren ist. Es sind jedoch Werkzeuge denkbar, die den Entwickler bei dieser Arbeit unterstützen.

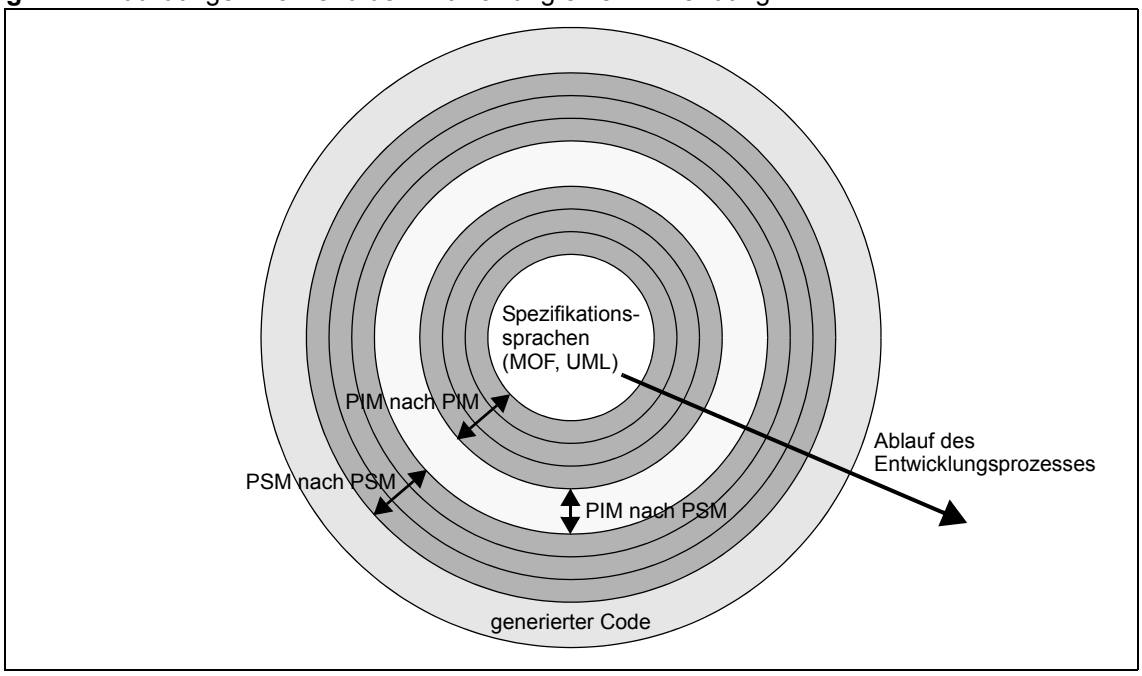
Ziel der OMG ist, mehrere Standard-Abbildungen von der PIM-Ebene auf die PSM-Ebene zu definieren, mit denen die Umsetzung eines plattformunabhängigen Modells auf ein entsprechendes plattformspezifisches Modell sich weitgehend automatisieren lässt. Solche Abbildungen müssen neben den Metamodellen von Abbildungs-Quelle und -Ziel auch Regeln für die Zuordnung der jeweiligen Konstrukte beinhalten. Sowohl die Metamodelle als auch die Abbildungsregeln sollen in UML definiert werden, um die vorhandene Werkzeugunterstützung und die Verarbeitungsmöglichkeiten dieser Sprache zu nutzen. In Abbildung 4.1 wird die Abfolge der verschiedenen Abbildungen während der Entwicklung einer Anwendung dargestellt.

Code-Generierung

Aus dem plattformspezifischen Modell einer Applikation wird nach ausreichender Verfeinerung innerhalb der PSM-Ebene schließlich Code für die Zielplattform generiert. Neben dem eigentlichen Programmcode gehören hierzu auch Schnittstellendefinitionen, Konfigurationsdateien und mehr. Inwieweit die Code-Generierung automatisiert werden kann, hängt entscheidend von der Genauigkeit und Vollständigkeit der Spezifikation ab.

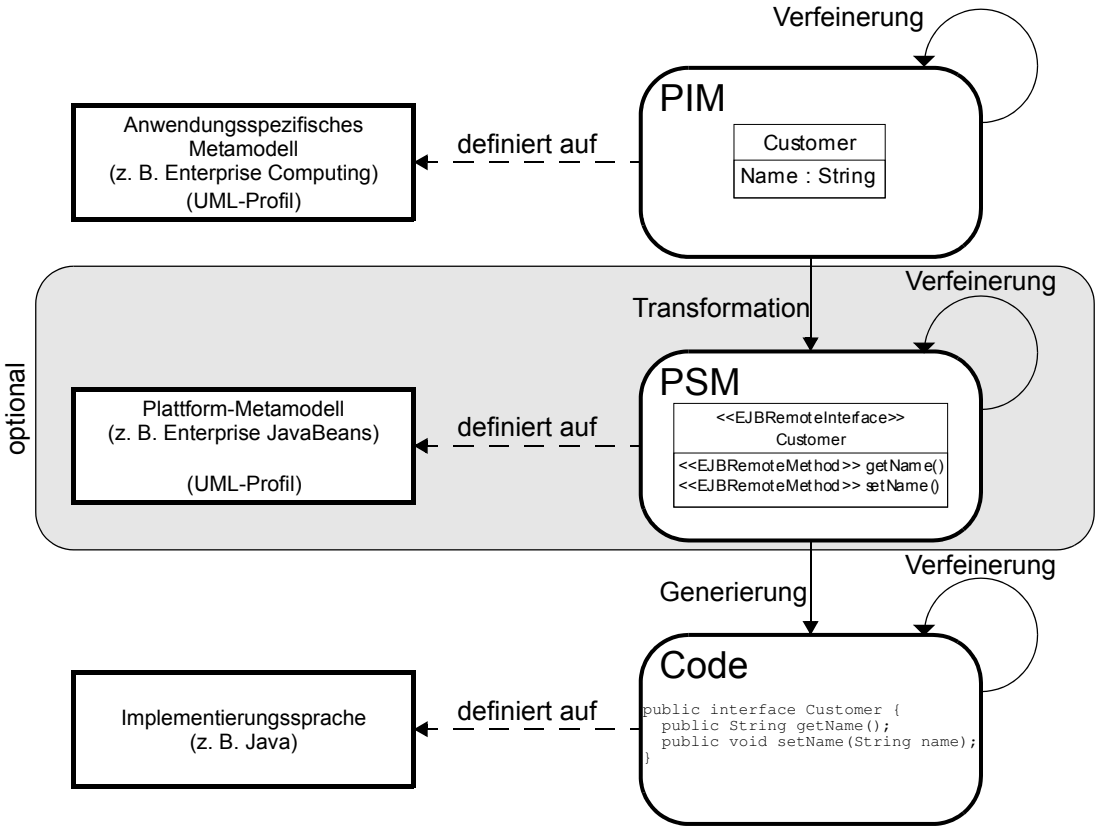
Es ist auch möglich, den Zwischenschritt über das plattformspezifische Modell auszulassen und direkt aus dem plattformunabhängigen Modell Code zu generieren, wenn die Spezifikation der Anwendung dort ausreichend detailliert vorliegt und der Code-Generator mächtig genug für diese Abbildungsleistung ist. Die Erstellung eines plattformspezifischen Modells ist also ein optionaler Schritt im Entwicklungsprozess. Unabhängig von der genauen Art der Code-Generie-

Abbildung 4.1 Abbildungen während der Entwicklung einer Anwendung



rung lässt sich die einmal plattformunabhängig spezifizierte Funktionalität verhältnismäßig leicht auf neue Zielplattformen abbilden. Abbildung 4.2 zeigt zusammenfassend die Schritte bei der Software-Entwicklung mit MDA.

Abbildung 4.2 Model Driven Architecture im Überblick



4.1.2 Bewertung

Die Model Driven Architecture ist eine vielversprechende Technik, um Software-Entwickler von mühseliger und wenig kreativer Handarbeit bei der Realisierung ihrer Entwürfe zu entlasten. Die Architektur von SQLInteract greift Konzepte von MDA auf, beispielsweise beim modellbasierten Schemaentwurf im CWM-Baum mit anschließender Generierung von SQL-Befehlen für die Ausführung in einem Datenbanksystem. Auch die erwähnten Ansätze für die Verfeinerung von Semantik und die Abbildung zwischen verschiedenen Modellen finden Berücksichtigung.

MDA enthält jedoch keine konkreten Verfahren für die Verfeinerung von Semantik und die Abbildung auf niedrigere Abstraktionsstufen. Stattdessen wird eingeräumt, dass die Möglichkeiten von UML für die Abbildung von Verfeinerungsbeziehungen bislang nur rudimentär sind und erweitert werden müssen. Ebenso wird anerkannt, dass Code für die Implementierung struktureller Eigenschaften wesentlich einfacher zu generieren ist als Code für die Implementierung von Verhalten. Insbesondere CORBA als favorisierte Plattform bietet in der Beschreibungssprache IDL kaum Möglichkeiten, die über die Definition syntaktischer Eigenschaften hinausgehen: Semantische Zusatzinformationen werden traditionellerweise in informalem Englisch angegeben. Insgesamt sieht die OMG zunächst weder bei der Abbildung von PIM auf PSM noch bei der Generierung von Code Möglichkeiten für eine weitgehende Automatisierung [OMG01a]. MDA liefert daher grundlegende Konzepte für SQLInteract; die genaue Realisierung dieser Konzepte bleibt jedoch selbst zu entwickeln.

4.2 Semantikmodellierung in SQLInteract

Über die intuitive Modellierung von Datenbank-Schemas und die Formulierung von Ad-hoc-Anfragen hinaus soll SQLInteract auch für die Entwicklung so genannter *Thick Database Applications* eingesetzt werden können. Es handelt sich dabei um Client-Server-Anwendungen, bei denen der Großteil der Funktionalität im Datenbanksystem implementiert ist, so dass sich die Aufgabe des Clients auf die Darstellung der Benutzeroberfläche und die Entgegennahme der Benutzereingaben beschränkt (*Thin Client*). Um die Entwicklung solcher Anwendungen zu unterstützen, muss SQLInteract Möglichkeiten für die Modellierung von Anwendungslogik bieten.

4.2.1 Einordnung der Begriffe

Semantik ist die Lehre von der Bedeutung sprachlicher Zeichen. Im Kontext von Software-Entwicklung steht Semantik für die in graphischer oder textueller Darstellung ausgedrückten Inhalte, im Gegensatz zur *Syntax*, die strukturelle Regeln für die Zusammensetzung gültiger Konstrukte beschreibt. Syntaktisch verschiedene Darstellungsformen können dieselbe Semantik beschreiben, sofern ihre Ausdrucksmächtigkeit identisch ist.

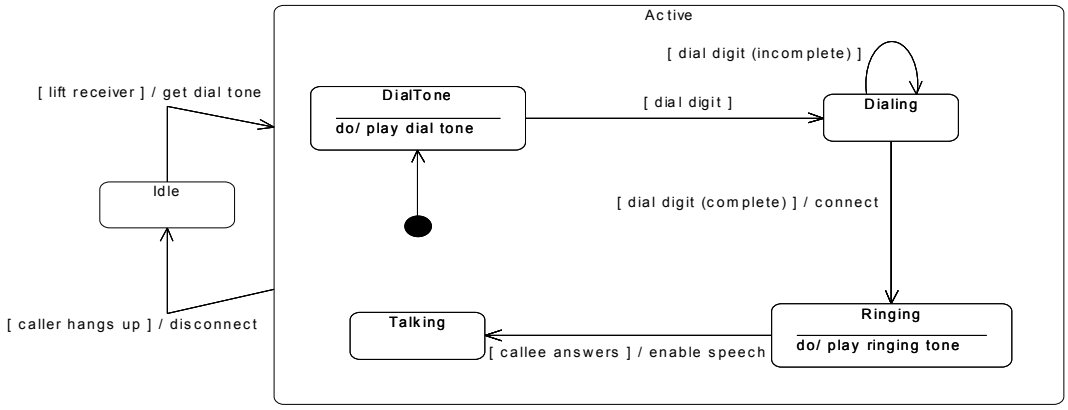
Grundsätzlich kann zwischen *struktureller Semantik* und *Verhaltenssemantik* unterschieden werden. Strukturelle Semantik beschreibt statische Aspekte von Systemen, bei der Software-Entwicklung etwa Klassen und ihre Attribute oder die Beziehungen zwischen Klassen, bei Datenbanksystemen entsprechend Tabellen mit ihren Spalten und referenzielle Bedingungen. Verhaltenssemantik beschreibt dynamische Aspekte von Systemen, also Zustandsänderungen, Vorgänge oder Aktionen. Konkrete Beispiele sind Methoden oder Prozeduren in der Software-

Entwicklung und Anfragen, benutzerdefinierte Routinen (*UDRs*) oder Trigger bei Datenbanksystemen.

Es fällt auf, dass trotz vereinzelter Gegenbeispiele die Beschreibung von Verhaltenssemantik typischerweise deutlich komplexer ist als die Beschreibung struktureller Semantik. Bei der Modellierung von Systemen wird strukturelle Semantik im Regelfall als graphisches Modell dargestellt, da im Vergleich zu einer rein textuellen Darstellung eine bessere Übersichtlichkeit gegeben ist, die sich vor allem aus der intuitiven Erfassbarkeit graphischer Symbole ergibt. Aus diesem Grund wird während der Modellierung auch die Verhaltenssemantik graphisch dargestellt. In UML beispielsweise werden für die Darstellung statischer Strukturen vor allem Klassendiagramme genutzt, für die Darstellung dynamischer Aspekte unter anderem Statechart- und Aktivitäts-Diagramme.

Bei näherer Betrachtung zeigt sich, dass die Ausdrucksmächtigkeit graphisch dargestellter Modelle für die Beschreibung von Verhaltenssemantik schnell an ihre Grenzen stößt. Wie Abbildung 4.3 anhand eines Statechart-Diagramms zeigt, werden graphische Elemente nur zur Strukturierung und für die Darstellung des Kontrollflusses genutzt. Die entscheidenden Aspekte der Verhaltensmodellierung wie Aktionen und Bedingungen werden bereits auf dieser noch sehr abstrakten Verfeinerungsstufe textuell formuliert. Mit fortschreitender Verfeinerung der Seman-

Abbildung 4.3 UML-Statechart-Diagramm



tik muss ein immer größerer Anteil der Verhaltenssemantik textuell beschrieben werden. Letztlich ist in einem lauffähigen System zwar auch die strukturelle Semantik nicht mehr graphisch, sondern formal-textuell definiert; im Gegensatz zur Definition der Verhaltenssemantik handelt es sich hier aber lediglich um einen Wechsel der Repräsentationsform zum Zwecke der maschinellen Verarbeitung des semantischen Modells. Die Ausdrucksmächtigkeit der graphischen Repräsentation von struktureller Semantik ist der Ausdrucksmächtigkeit der textuellen Repräsentation ebenbürtig. Für die Beschreibung von Verhaltenssemantik kommen jedoch bei fast allen Spezifikationssprachen ab einer gewissen Verfeinerungsstufe textuelle Konstrukte zum Einsatz.

4.2.2 Anforderungen von SQLInteract

SQLInteract soll die effiziente, intuitive Modellierung beider Arten von Semantik erlauben. Die Anwendungsdomäne ist dabei auf die Entwicklung so genannter Thick Database Applications festgelegt. Die Modellierung struktureller Semantik besteht in diesem Fall aus dem Entwurf eines Datenbank-Schemas mit Tabellen und referenziellen Constraints. Für die Bereitstellung derartiger Funktionalität eignet sich das CWM-Metamodell in Verbindung mit einer geeigneten Benutzeroberfläche, wie sie in Kapitel 3 beschrieben wird.

Verhalten wird in Datenbanksystemen über Trigger, ECA-Regeln oder UDRs spezifiziert. Die Modellierung dieser Konstrukte könnte direkt in SQL erfolgen; dies würde jedoch den Anforderungen an eine intuitive Bedienung nicht genügen, da der Benutzer direkt auf der Implementierungsebene programmieren müsste. Es gilt daher, eine Möglichkeit für die abstraktere Spezifikation von Verhalten zu finden, ähnlich wie CWM-Modelle eine von konkreten DDL-Befehlen abstrahierende Spezifikation von Datenbank-Schemas erlauben.

CWM selbst bietet im zentralen Package *Object Model* zwei Klassen *StructuralFeature* und *BehavioralFeature* und scheint so auf den ersten Blick Möglichkeiten für die Modellierung beider Arten von Semantik zu bieten. Für die Speicherung von Verhaltenssemantik existieren die Klassen *Operation*, *Method*, *Procedure* und *Parameter*, die untereinander auch entsprechend verknüpft sind, um etwa den Aufruf von Prozeduren und die Übergabe von Parametern modellieren zu können. Allerdings bauen alle Verhalten modellierenden Klassen auf der Klasse *Expression* auf, um Ausdrücke oder Rümpfe von Prozeduren zu verwalten. *Expression* speichert diese Daten aber lediglich als String. Eine detailliertere Berücksichtigung des Inhalts ist nicht vorgesehen, auf CWM-Ebene bleibt die enthaltene Semantik folglich eine *Black Box*.

Die Speicherung von Verhaltenssemantik in CWM-Modellen ist also möglich, die eigentliche Modellierung und Interpretation der Semantik muss aber außerhalb von CWM erfolgen. Damit gibt es in CWM selbst keine nennenswerte Möglichkeit für die Modellierung von Verhaltenssemantik. Die CWM-Implementierung in SQLInteract verwendet dementsprechend zwar die Klasse *Procedure* zum Speichern von benutzerdefinierte Routinen (UDRs); für die Spezifikation der Inhalte muss aber eine andere Lösung gefunden werden, wenn der Benutzer nicht doch zur manuellen SQL-Programmierung gezwungen werden soll. Im Folgenden werden daher die Möglichkeiten zur Modellierung von Verhaltenssemantik in anderen Spezifikationssprachen betrachtet.

4.3 Verhaltenssemantik in UML

In der Spezifikation von UML [OMG03a] findet sich im Abschnitt über die Sprachsemantik auch ein ausführliches Kapitel zum Package *Behavioral Elements* mit der Beschreibung mehrerer Subpackages für die Modellierung von dynamischem Verhalten. Die bereits in früheren Versionen der UML-Spezifikation enthaltenen Bestandteile des Package sollen hier kurz vorgestellt werden. Anschließend erfolgt eine Betrachtung des in UML 1.5 neu eingeführten *Actions*-Subpackage sowie der zu UML gehörenden formalen Spezifikationssprache *Object Constraint Language* (*OCL*).

4.3.1 Behavioral Elements

Das Package *Behavioral Elements* ist innerhalb von UML für die Spezifikation von dynamischen Modellaspekten und Verhalten zuständig. Es gliedert sich in die Subpackages *Common Behavior*, *Collaborations*, *Use Cases*, *State Machines* und *Activity Graphs*. Mit Version 1.5 der UML-Spezifikation kam das Subpackage *Actions* hinzu.

Common Behavior

Das Subpackage *Common Behavior* bildet die Basis des Behavioral-Elements-Package. Hier werden grundlegende Konstrukte für die Modellierung dynamischer Elemente definiert, auf die die anderen Subpackages zurückgreifen. Common Behavior fällt damit eine ähnliche Rolle zu wie *Core* im Foundation Package. Im Detail definiert das Subpackage Klassen für das Senden und Empfangen von Signalen, für die Verwaltung von Instanzen und Verbindungen zwischen ihnen und für die Definition von Prozeduren.

Collaborations

Collaborations enthält Klassen für die Spezifikation von Kooperation zwischen Instanzen. Hierfür sind zwei zentrale Aspekte zu betrachten.

- Die strukturelle Beschreibung der kooperierenden Instanzen, das heißt vor allem ihre Verhältnisse untereinander, genannt *Collaboration*.
- Die Kommunikation zwischen den kooperierenden Instanzen, die für die Erfüllung der gemeinsamen Funktionalität notwendig ist, genannt *Interaction*.

Das Subpackage definiert daher Rollen, Interaktionsnachrichten und spezielle Assoziationen.

State Machines

Das *State-Machines*-Subpackage spezifiziert Konstrukte, mit denen Verhalten in Form von endlichen Automaten (*Finite State Machines*) modelliert werden kann. Dabei ist sowohl die Modellierung des Verhaltens einzelner Entitäten als auch die Modellierung von Interaktion möglich. Endliche Automaten beziehen ihren Namen aus der endlichen Menge von Zuständen, die sie einnehmen können. Zwischen Zuständen können Transitionen definiert werden. Eine Transition überführt bei Auslösung einen Zustand in einen anderen Zustand, wenn die entsprechenden Vorbedingungen erfüllt sind. Innerhalb von Zuständen und Transitionen können Aktionen ausgeführt werden. Das Subpackage enthält die nötigen Klassen, um diese Funktionalität darzustellen, etwa *State*, *Transition*, *Guard* und *Event*.

Activity Graphs

Aktivitäts-Graphen (Activity Graphs) sind eine Erweiterung des Konzepts der endlichen Automaten; dementsprechend ist *Activity Graphs* ein Subpackage von *State Machines*. Bei Aktivitäts-Graphen liegt der Schwerpunkt auf der Reihenfolge und auf den Vorbedingungen der ausgeführten Aktionen, so dass die meisten Zustände atomare Aktionen darstellen. Die Übergänge zu solchen Zuständen können durch verschiedene Events ausgelöst werden. *Activity Graphs* erweitert *State Machines* um die für die Modellierung von Aktivitäts-Graphen notwendigen Klassen.

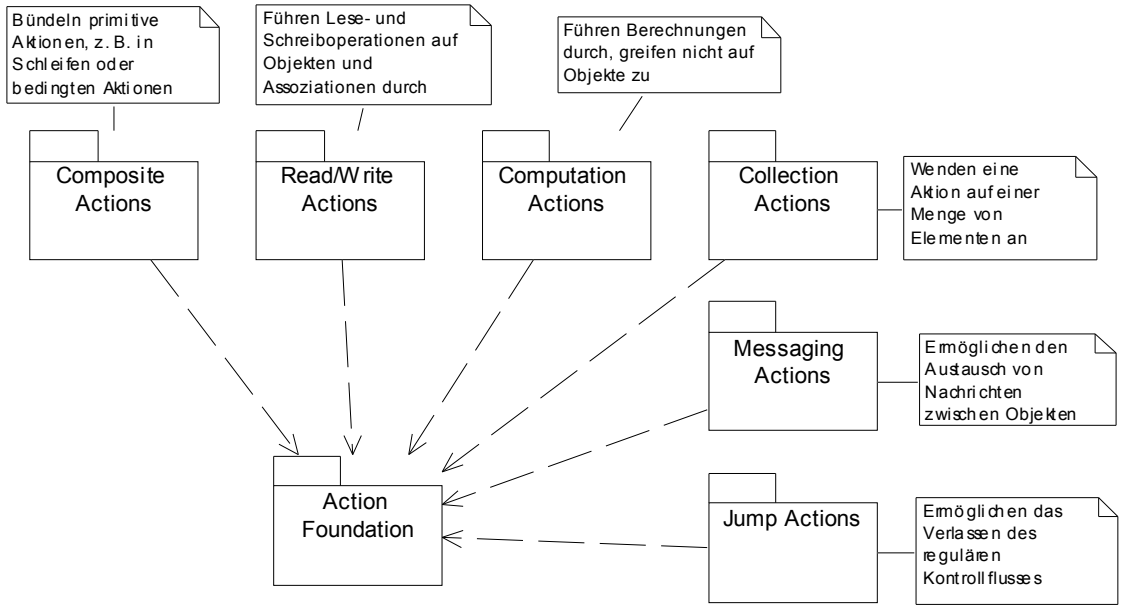
4.3.2 Actions

Das *Actions*-Subpackage ist eine wichtige Neuerung von UML 1.5, die Unzulänglichkeiten früherer Versionen beheben soll. Die bisher betrachteten Packages für die Modellierung von Verhaltenssemantik bieten bei näherer Untersuchung tatsächlich kaum Möglichkeiten, komplexes Verhalten zu spezifizieren. *Core* definiert als zentrales Element der Verhaltensmodellierung die Klasse *Procedure*. Das entscheidende Attribut einer Prozedur ist die in ihr enthaltene Verhaltenssemantik. *Procedure* sieht hier aber nur einen simplen String zur Aufnahme des Prozedurrumpfes vor. In Verbindung mit dem ebenfalls gespeicherten Namen der Implementierungssprache des Prozedurrumpfes wird die Interpretation der Semantik auf die Entwicklungswerkzeuge im konkreten Einzelfall abgewälzt und aus dem UML-Modell völlig herausgehalten. Das Geflecht von Klassen und Assoziationen, dass *Core* ansonsten definiert, bleibt ohne die in *Procedure*-Instanzen gespeicherte Verhaltenssemantik ein statisches Gerüst, das lediglich strukturelle Semantik beinhaltet.

Collaborations bietet keine eigenen Konstrukte, die mit *Procedure* vergleichbar wären, erschließt folglich auch keine neuen Möglichkeiten für die Modellierung von Verhaltenssemantik. *State Machines* scheint auf den ersten Blick vielversprechend; die innerhalb von Zuständen oder Transitionen auszuführenden Aktionen werden jedoch über *Procedure*-Instanzen spezifiziert, so dass die hiermit zusammenhängenden Schwächen des Modells von *Common Behavior* übernommen werden.

Das *Actions*-Subpackage soll die Lücke im Modell nun schließen, indem es *Action Semantics* definiert, und in der Tat scheint es geeignet, eine echte Modellierung von Verhaltenssemantik zu ermöglichen. Wie Abbildung 4.4 zeigt, wurden ausführbare Aktionen in sechs Kategorien unterteilt. Bewährten Modellierungsansätzen folgend, setzen Subpackages für die sechs Kategorien auf einer gemeinsamen Basis, dem *Action-Foundation*-Package, auf und erweitern sie jeweils um spezielle Konstrukte.

Abbildung 4.4 Action Package (vereinfacht)



Die Anbindung an die anderen Subpackages von *Behavioral Elements* erfolgt über die Klasse *Procedure*. Auf diese Weise wird die Modellierungsmächtigkeit von *Procedure* erheblich gesteigert, und die Spezifikation der anderen Subpackages kann unverändert bleiben. *Actions* und die restlichen Subpackages ergänzen sich also gegenseitig, wobei *Procedure* die Schnittstelle bildet: Konstrukte aus *Common Behavior* oder *State Machines* spezifizieren, wann eine Prozedur ausgeführt wird, und Action Semantics definieren den genauen Inhalt der Prozedur sowie die zu bearbeitenden Daten.

Das *Action*-Subpackage definiert als kleinste Einheit so genannte *primitive Aktionen*, möglichst feingranulare Konstrukte, um Abbildungen auf viele verschiedene Zielplattformen zu ermöglichen. Primitive Aktionen führen entweder Berechnungen aus oder greifen auf Datenstrukturen zu, so dass sich die Spezifikation des Datenmodells von der der Operationen trennen lässt. Aus primitiven Aktionen lassen sich komplexe Aktionen (*Composite Actions*) zusammensetzen, indem sie über Kontrollstrukturen wie Schleifen und Verzweigungen verknüpft werden.

Ohne die explizite Angabe eines Kontroll- oder Datenflusses besteht keine Garantie über die Ausführungsreihenfolge von Aktionen; ebenso treffen Aktionen keinerlei Annahmen über ihren Ausführungskontext. Als Grund für diese Entwurfsentscheidung wird die häufige Überspezifikation der Ausführungsreihenfolge in traditionellen Programmiersprachen genannt, die eine flexible Ausführung der Semantik, etwa durch Umordnung von Aktionen, verhindert. Da der Entwickler bei der Modellierung von Verhalten mit *Action Semantics* gezwungen ist, Abhängigkeiten explizit anzugeben, wird im Modell die größtmögliche Entkopplung der einzelnen Aktionen realisiert. Voraussetzung hierfür ist allerdings, dass der Entwickler keine für die Funktionalität irrelevanten Abhängigkeiten modelliert.

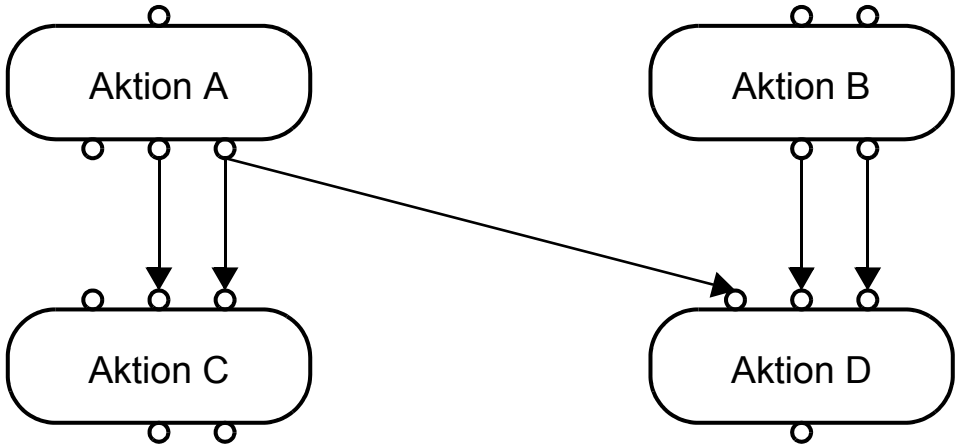
Abhängigkeiten zwischen Aktionen können durch Daten- oder Kontrollflüsse definiert werden. Über einen Datenfluss wird die Ausgabe einer Aktion zur Eingabe einer anderen Aktion. Solange die erste Aktion nicht vollständig ausgeführt ist, steht die Ausgabe nicht zur Verfügung, und die Ausführung der zweiten Aktion ist blockiert. Kontrollflüsse modellieren sequenzielle Abhängigkeiten zwischen Aktionen, bei denen keine Daten ausgetauscht werden. Eine solche Konstellation ist etwa bei Aktionen denkbar, die abwechselnd auf dieselbe Datenstruktur zugreifen. Besteht zwischen zwei Aktionen bereits ein Datenfluss, definiert dieser implizit auch einen Kontrollfluss, um die Ausführung der zweiten Aktion zu blockieren, bis die vollständige Ausführung der ersten Aktion die benötigten Daten bereitstellt. Durch die Verbindung von Aktionen über Kontroll- und Datenflüsse ergibt sich als Gesamtstruktur aller Aktionen ein azyklischer, gerichteter Graph, da eine Aktion nicht gleichzeitig Vorgänger und Nachfolger einer anderen Aktion sein kann.

Jegliche Ein- und Ausgabe von Aktionen wird über so genannte *Pins* spezifiziert. Ein Pin gibt Typ und Multiplizität der jeweiligen Ein- oder Ausgabe an. Konkrete Werte müssen diesen Bedingungen genügen, wobei im Falle komplexer Typen auch Spezialisierungen des vom Pin geforderten Typs zulässig sind. Datenflüsse verbinden Ausgabe-Pins einer Aktion mit Eingabe-Pins einer Folgeaktion. Auch hierbei muss die Typ- und Multiplizitätskonformität erhalten bleiben: Der Typ des Ausgabe-Pins muss dem des Eingabe-Pins oder einer Spezialisierung desselben entsprechen, und die erlaubten Kardinalitäten der Pins müssen gleich sein.

Ein Ausgabe-Pin kann mit beliebig vielen Eingabe-Pins verbunden sein oder auch unverbunden bleiben; ein Eingabe-Pin kann jedoch mit höchstens einem Ausgabe-Pin verbunden sein. Folglich können Datenflüsse verzweigen (*Fan Out*), nicht jedoch mehrere Datenflüsse zu einem ver-

einigt werden (*Fan In*). Abbildung 4.5 erläutert dies anhand eines informalen Beispiels, das nicht der offiziellen Notation entspricht.

Abbildung 4.5 Pins und Datenflüsse



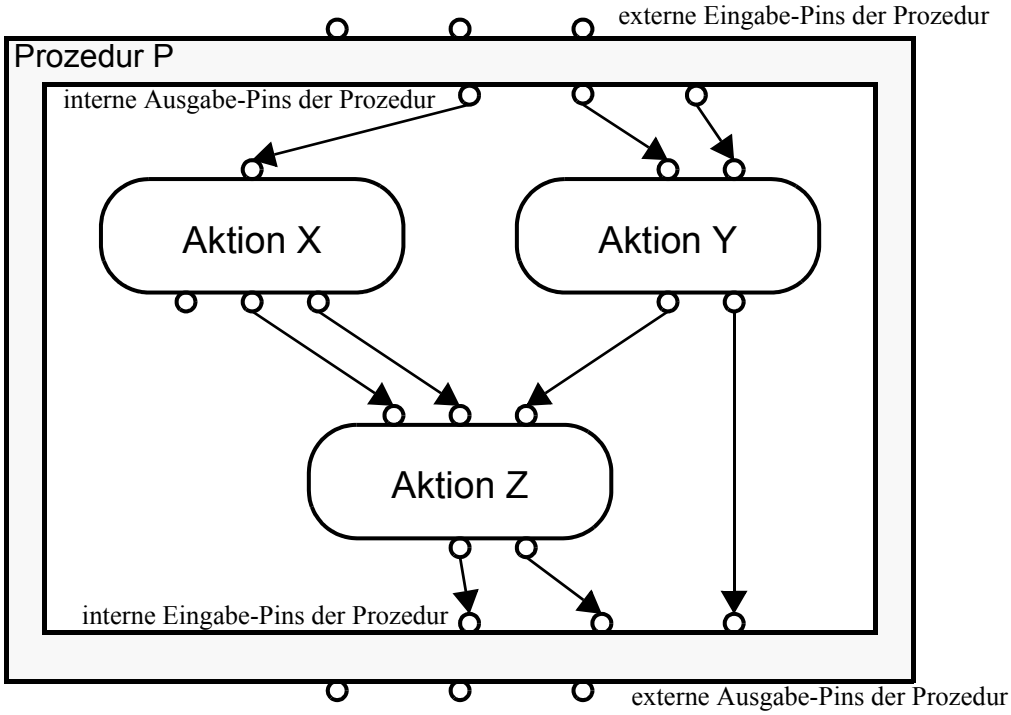
Aktionen können auf verschiedene Weisen und auf mehreren Ebenen verschachtelt werden. Die *Procedure* ist die höchste dieser Ebenen. Eine Prozedur fasst eine Menge von Aktionen zusammen, die so im Gesamtmodell mit anderen Konstrukten assoziiert werden können. Ihr können Argumente übergeben werden, und sie kann Rückgabewerte liefern. Intern ist die Prozedur mit den enthalten Aktionen über Pins derart verbunden, dass die üblichen Regeln für Datenflüsse gelten. Die der Prozedur übergebenen Argumente werden über Ausgabe-Pins der Prozedur an Eingabe-Pins der enthaltenen Aktionen weitergeleitet, und die Rückgaben der Aktionen werden über deren Ausgabe-Pins an Eingabe-Pins der Prozedur geleitet. Die einzige Verschärfung gegenüber Datenflüssen im Allgemeinen besteht darin, dass kein Eingabe-Pin der Prozedur unverbunden bleiben darf. Abbildung 4.6 zeigt informal ein Beispiel für die Struktur einer Prozedur.

Eine tiefer angesiedelte Verschachtelung von Aktionen sind *komplexe Aktionen* (*Composite Actions*). Eine komplexe Aktion ist im einfachsten Fall eine Menge von primitiven Aktionen, die je nach Modellierung durch den Entwickler nebenläufig oder unter Berücksichtigung von Daten- und Kontrollflüssen strukturiert ausgeführt werden. In diesem Fall wirkt die komplexe Aktion nur als gliederndes Element. Daneben gehören auch *bedingte Aktionen* (*Conditional Actions*) und *Schleifenaktionen* (*Loop Actions*) zu den komplexen Aktionen.

Bedingte Aktionen enthalten eine Menge von Klauseln, die jeweils aus einem Test und einem Rumpf bestehen. Liefert der Test einer Klausel die Rückgabe ’wahr’, wird der zugehörige Rumpf ausgeführt. Ohne explizite Strukturierung durch den Entwickler gibt es keine Garantien über die Ausführungsreihenfolge der einzelnen Tests, so dass von nebenläufiger Ausführung ausgegangen werden muss. Es wird jedoch immer nur ein Rumpf ausgeführt; liefern mehrere gleichzeitig ausgeführte Tests die Rückgabe ’wahr’, so wird einer der zugehörigen Rümpfe indeterministisch für die Ausführung bestimmt. Die Verantwortung für eine sinnvolle und korrekte Spezifikation der Klauseln und die Strukturierung ihrer Ausführung liegt also allein beim Entwickler.

Schleifenaktionen enthalten genau eine Klausel, die wie bei bedingten Aktionen aus einem Test und einem Rumpf besteht. Der Rumpf wird wiederholt ausgeführt, solange die Testrückgabe ’wahr’ lautet. Die Ausgaben einer Rumpf-Iteration sind dabei die Eingaben der nächsten Itera-

Abbildung 4.6 Prozedur und Aktionen



tion und werden daher als Schleifenvariablen bezeichnet. Schleifenaktionen sind für die allgemeine Modellierung von Iteration bestimmt; für die Ausführung von Aktionen auf einer Menge gleichartiger Elemente existieren spezielle *Sammlungsaktionen* (*Collection Actions*).

Sammlungsaktionen wenden eine Aktion auf eine Menge von Datenelementen an, wobei die Ausführung der Aktion auch parallel auf mehreren oder allen der Elemente erfolgen kann. Durch die Bereitstellung spezieller Aktionen für die Bearbeitung von Sammlungen erübrigt sich für den Entwickler die manuelle explizite Modellierung des iterativen Zugriffs auf jedes einzelne Element etwa mithilfe von Schleifen. Eine Sammlungsaktion enthält eine interne Aktion, die für jedes Element der übergebenen Sammlung einmal ausgeführt wird. Es gibt vier verschiedene Arten von Sammlungsaktionen:

- Eine *Map Action* wendet die interne Aktion parallel auf alle Elemente an. Sie kommt daher zum Einsatz, wenn die Ausführung der internen Aktionen keine Seiteneffekte hat. Die Ausgabe entspricht in Art und Größe der Eingabe.
- Eine *Filter Action* selektiert eine Teilmenge der Elemente anhand des Bool'schen Rückgabewertes der internen Aktion, die auf jedes Element angewendet wird. Die selektierten Elemente werden in einer neu angelegten Sammlung zusammengefasst und bilden die Ausgabe.
- Eine *Iterate Action* wendet die interne Aktion sequenziell auf die Elemente an; sie ist damit am ehesten mit einer konventionellen Schleife vergleichbar und enthält wie eine *Loop Action* Schleifenvariablen. Als Ausgabe dienen die Inhalte der Schleifenvariablen nach Abarbeitung aller Elemente.
- Eine *Reduce Action* wendet die interne Aktion auf jeweils zwei benachbarte Elemente der Sammlung an, wobei das Eingabe-Paar durch das Resultat der internen Aktion ersetzt wird. Bei jeder Iteration verringert sich die Anzahl der Elemente in der Sammlung so um eins, bis

nach vollständiger Abarbeitung genau ein Element übrig bleibt. Die interne Aktion muss assoziativ sein, und die Reihenfolge der Bearbeitung der Elemente ist unbestimmt; diese Konventionen erlauben unter anderem die performanzsteigernde parallele Anwendung der internen Aktion auf mehrere Paare von Elementen. Ein Beispiel für eine Operation, die sich als Reduce Action durchführen lässt, ist die Aufsummierung aller Elemente der Sammlung.

Die restlichen Typen von Aktionen sind Lese- und Schreibaktionen (*Read/Write Actions*), Berechnungsaktionen (*Computation Actions*), Nachrichtenaktionen (*Messaging Actions*) und Sprungaktionen (*Jump Actions*).

Lese- und Schreibaktionen greifen beispielsweise auf Objekte, Attribute und Assoziationen zu. Wie die Bezeichnung nahelegt, modifizieren Lesenaktionen dabei nicht die Werte, auf die sie zugreifen. Schreibaktionen sind im Regelfall so angelegt, dass sie minimale Auswirkungen zeigen. Die Instanziierung eines Objekts etwa führt keine Konstruktoren aus; dies ist Aufgabe einer zweiten Schreibaktion.

Berechnungsaktionen arbeiten direkt auf Werten; sie greifen nicht auf Datenstrukturen zu und interagieren nicht mit anderen Aktionen, ihre Funktionalität ist also kontextfrei gekapselt. Sie werden für die Modellierung primitiver mathematischer Funktionen genutzt, aus denen sich dann komplexere Berechnungen zusammensetzen lassen.

Nachrichtenaktionen erlauben es Objekten, Nachrichten auszutauschen. Eine initiale Nachricht kann synchron oder asynchron versandt werden. Im Falle der asynchronen Versendung spricht man von einem *Send*; das sendende Objekt führt seine Funktionalität weiter aus und ignoriert das durch den Send ausgelöste Verhalten des Empfängers sowie eventuelle Antwortnachrichten. Eine synchron versandte Nachricht heißt *Call*; das sendende Objekt unterbricht seine Ausführung und wartet auf eine Antwort des Empfängers, die auch Rückgabewerte enthalten kann. Wie der Empfänger die eingehende Nachricht verarbeitet und gegebenenfalls die Antwort erstellt, ist im Rahmen der Nachrichtenaktionen nicht näher spezifiziert.

Sprungaktionen ermöglichen das Verlassen des regulären Kontrollflusses. Sie sind vor allem für die Behandlung von Ausnahmesituationen bestimmt und entsprechen damit Statements wie *break* oder *continue* in bekannten Programmiersprachen. Auch die Ausnahmebehandlung in Form von *Exceptions* wird durch Sprungaktionen abgedeckt.

4.3.3 Object Constraint Language

Die *Object Constraint Language* ist Bestandteil der UML-Spezifikation und dient, wie der Name schon suggeriert, zur Definition von Bedingungen auf Modellkonstrukten. Die Notwendigkeit einer solchen Sprache erwächst aus der Erkenntnis, dass UML-Diagramme allein nicht geeignet sind, alle Aspekte einer Spezifikation in ausreichender Verfeinerungstiefe zu modellieren [OMG03a]. Erfolgt die Beschreibung der nicht im Modell enthaltenen Informationen in natürlicher Sprache, sind in der Praxis Mehrdeutigkeiten unvermeidbar. Formale Sprachen, die Mehrdeutigkeiten ausschließen, sind jedoch oft an mathematischen Ausdrucksweisen orientiert und ohne entsprechendes Hintergrundwissen schwer verständlich.

OCL wurde als formale Sprache mit dem Ziel entwickelt, auch ohne ausgeprägtes mathematisches Wissen seitens des Entwicklers leicht les- und schreibbar zu sein. Es handelt sich um eine reine Ausdruckssprache: Innerhalb von OCL ist nur die Auswertung von Ausdrücken möglich, deshalb kann es zu keiner Beeinflussung von Objektzuständen durch die Auswertung von Aus-

drücken kommen. Auch wenn die Spezifikation von Ausnahmebehandlungen (*Exception Checking*) möglich ist, kann keine Programmlogik und kein Kontrollfluss in OCL implementiert werden.

OCL ist für ein klar umrissenes Einsatzgebiet entwickelt worden, und die Spezifikation von Verhaltenssemantik ist nicht Teil dieses Einsatzgebietes. Über Pre- und Postconstraints können zwar etwa Zustandsänderungen spezifiziert werden; weitergehende Ansätze, OCL für Modellierung von Verhalten einzusetzen sind jedoch stets ein Kampf gegen die Natur der Sprache und stoßen recht bald an absolute Grenzen. Für die in dieser Arbeit untersuchten Aspekte spielt OCL daher keine relevante Rolle.

4.3.4 Action Specification Language

Die *Action Specification Language* (*ASL*) ist eine frei verfügbare Sprache für die implementierungsunabhängige Spezifikation von Verhaltenssemantik [WKC+01]. Die Entwicklung von ASL begann bereits 1993; die aktuelle Version 2.5c berücksichtigt die Action Semantics von UML 1.5 und kann für die textuelle Darstellung von Action-Semantics-Modellen genutzt werden. Es existieren verschiedene Verfahren für die Abbildung der mit ASL spezifizierten Semantik auf Implementierungssprachen, von manueller Umsetzung durch Entwickler bis zur automatisierten Code-Generierung, etwa durch den konfigurierbaren Code-Generator I-CCG von Kennedy Carter [KC02].

ASL modelliert Verhalten als eine Menge interagierender Automaten, die nebenläufig arbeiten und über synchrone Nachrichten (Calls) kommunizieren. Ein solcher Automat kann auf eine Nachricht mit einem Zustandswechsel reagieren, innerhalb dessen Funktionalität ausgeführt wird. Die Funktionalität wird mithilfe einer textuellen Sprache definiert, die einer Programmiersprache recht ähnlich sieht. Statements werden grundsätzlich sequenziell abgearbeitet, wobei Kontrollstrukturen für bedingte Aktionen, Verzweigungen und Schleifen zur Verfügung stehen. Die spezifizierte Semantik wird in *Segmente* gegliedert, die den Prozeduren von Programmiersprachen entsprechen. ASL bietet Konstrukte für Signalisierung zwischen verschiedenen Automaten und Zugriff auf die in Signalen enthaltenen Daten. Weiterhin kann in Segmenten auf die im zugehörigen Klassendiagramm spezifizierten Daten zugegriffen werden, ebenso existieren Möglichkeiten für die Manipulation von Objekten und ihren Attributen, die Definition und Verwaltung komplexer Datentypen sowie die Ausführung logischer und arithmetischer Operationen.

4.3.5 Object Action Language

Die *Object Action Language* (*OAL*) entstand als Spezifikationssprache für Action Semantics nach UML 1.5 im Entwicklerwerkzeug BridgePoint [Pro03], ist aber nicht an dieses gekoppelt [Pro02]. Hauptziele bei der Entwicklung der Sprache waren gute Lesbarkeit, Ausführbarkeit der Spezifikationen und automatisierbare Abbildung auf Implementierungsarchitekturen. BridgePoint erlaubt folglich auch die Generierung von lauffähigem Code aus Spezifikationen in OAL.

Auch OAL modelliert Verhalten über nebenläufige, interagierende Automaten, die Signale empfangen und daraufhin Zustandswechsel vornehmen können, innerhalb derer Funktionalität ausgeführt wird. Diese Funktionalität wird über textuelle OAL-Statements spezifiziert, die grundsätzlich sequenziell abgearbeitet werden. Neben Kontrollstrukturen für bedingte Aktionen und Schleifen wird auch eine spezielle *For Each* Anweisung geboten, die eine Aktion auf jedes Element einer Menge anwendet. Darüber hinaus existieren Anweisungen für die Manipulation

von Objekten und die Verwaltung von Assoziationen sowie für das Senden und Empfangen von Nachrichten. Im Vergleich zu ASL sind die Möglichkeiten für die Definition und Verarbeitung von Ausdrücken detaillierter. Ähnlich wie in ASL können auch Statements in OAL auf in Signalen übermittelte Daten und im Klassendiagramm modellierte Objektattribute zugreifen. Beispiel 4.1 zeigt, wie in OAL eine Nachfolgeversion eines versionierten Objekts vom Typ *Task* angelegt wird. In Kapitel 5 wird versionierte Datenspeicherung in so genannten *Repositories* ausführlich vorgestellt.

Beispiel 4.1 Anlegen einer Nachfolgeversion in OAL

```
if (currentTask.successors < currentTask.maxSuccessors)
  create object instance newTask of Task;
  newTask.taskName = currentTask.taskName;
  newTask.objId = currentTask.objId;
  newTask.verId = currentTask.verId + 1;
  newTask.frozen = false;
  newTask.successors = 0;
  newTask.maxSuccessors = currentTask.maxSuccessors;
  currentTask.successors = currentTask.successors + 1;
  relate newTask to currentTask across r1.predecessor;
end if;
```

4.4 Abstract State Machine Language

Die *Abstract State Machine Language* (*AsmL*) wurde als ausführbare Sprache für die Spezifikation von Verhalten bei Microsoft Research entwickelt [BS01]. Sie basiert auf dem Konzept abstrakter Automaten, das um Objektverwaltung, Ausnahmebehandlung und ein Typsystem erweitert wurde. Motivation für die Entwicklung waren die mangelnden Möglichkeiten für die Spezifikation von Verhalten bei der Komponentenentwicklung. *Design by Contract* [MM01], die Angabe von Invarianten, Vor- und Nachbedingungen, erlaubt zwar mehrdeutigkeitsfreie Angaben, wird jedoch in der Praxis nur ungern benutzt. Stattdessen kommen oft informale Verhaltensbeschreibungen zum Einsatz, denen es dann an Präzision und Eindeutigkeit mangelt. AsmL bietet die Möglichkeit, Verhalten präzise und formal zu spezifizieren.

Komponenten werden in AsmL über *Interfaces* modelliert, wobei speziell Verhalten, Invarianten über der Verfeinerungshierarchie und die Komposition von Komponenten im Vordergrund stehen. Realisierende Klassen müssen stets Verhaltens-Spezialisierungen (*Behavioral Subtypes*) der zugehörigen Interfaces sein. Ein Interface legt dabei das geforderte Verhalten fest; eine Spezialisierung, das heißt ein anderes Interface oder eine realisierende Klasse, darf das Verhalten beliebig verfeinern oder ausbauen. Innerhalb der Verfeinerungshierarchie müssen Elemente jedoch stets gegen weiter verfeinerte Elemente austauschbar sein, so dass kein Element gegen das von abstrakteren Elementen spezifizierte Verhalten verstoßen darf. Insbesondere ist die Syntax der Aufrufschnittstelle festgeschrieben. Weitere Details für verfeinernde Elemente können über Invarianten und Constraints angegeben werden.

In den Interface-Spezifikationen wird Verhalten über *Modellvariablen* und *Modellprogramme* modelliert. AsmL bietet dabei transaktionsbasierte Semantik und Ausnahmebehandlung. Anstelle durch ein konventionelles Modellprogramm kann die Funktionalität auch über Vor- und

Nachbedingungen spezifiziert werden. Interfaces können parametrisiert werden, so dass sie generisch nutzbar sind und im Falle der Benutzung mit einem konkreten Typ initialisiert werden müssen. Weiterhin können Interfaces Referenzen auf andere Interfaces erzeugen. Auf diese Weise lassen sich Komponenten durch Komposition zusammensetzen. Die Verbindung zwischen zwei Interfaces kann über gemeinsame Daten oder gegenseitige Aufrufe erfolgen.

4.5 Hintergründe der Semantik-Modellierung

Eine ideale abstrakte Spezifikationssprache für die Beschreibung von Semantik erfüllt zwei Bedingungen: Ihr Vokabular ist *minimal*, und ihre Abstraktionen sind *orthogonal*. Minimalität des Vokabulars bedeutet, dass nur ein Ausdruck für jeden beschreibbaren Aspekt existiert, so dass Ausdruck und Inhalt einander eineindeutig zugeordnet sind. Sind die Abstraktionen der Sprache orthogonal, so sind die von ihnen beschriebenen Inhalte überlappungsfrei. Zwei Ausdrücke können sich in ihrer Bedeutung so nicht beeinflussen, mehrdeutige Zustände werden vermieden.

Verfeinerung von Semantik

Bei der Entwicklung von Softwaresystemen ist Verfeinerung ein zentraler Begriff. Nach Analyse der Anforderungen wird zunächst ein Grobentwurf des Systems angefertigt, der im Verlauf der weiteren Entwicklung schrittweise verfeinert wird, bis am Ende die Funktionalität so feingranular dargestellt ist, dass sie von Maschinen durch Kompilierung oder Interpretation ausgeführt werden kann. Die genaue Vorgehensweise hängt dabei vom angewandten Entwicklungsprozess ab, doch die Verfeinerung von Funktionalität bis hin zur Ausführbarkeit ist allen Prozessen gemein. Funktionalität ist in diesem Zusammenhang ein zusammenfassender Ausdruck für strukturelle Semantik und Verhaltenssemantik. Der Anteil der Verhaltenssemantik ist typischerweise wesentlich höher als der Anteil der strukturellen Semantik: Ein System wird erst durch ausführbare Aktionen nutzbar. Komplexe strukturelle Semantik ohne zugehörige dynamische Komponenten, also Verhalten, macht keinen Sinn. Selbst im Fall außergewöhnlich komplexer struktureller Semantik steht dieser also eine entsprechende Menge an Verhaltenssemantik gegenüber, um die Strukturen nutzbar zu machen. Mit Blick auf das Mengenverhältnis und die größere Problematik bei der Formulierung von Verhaltenssemantik stellt die Verhaltenssemantik bei der Software-Entwicklung den schwierigeren Teil dar.

Im Zuge der Verfeinerungsschritte liegt daher das Hauptaugenmerk auf der Verfeinerung der Verhaltenssemantik. Von entscheidender Bedeutung ist dabei der Begriff der *Verhaltens-Subtypen* (*Behavioral Subtypes*) [LW94], der speziell bei der Entwicklung von AsmL besonders berücksichtigt wurde, sich aber auch bei anderen Spezifikations- und Implementierungssprachen wiederfindet. Ein durch semantische Verfeinerung entstandenes Konstrukt der Verhaltensbeschreibung, also eine Methode oder Funktion, muss ein Verhaltens-Subtyp des entsprechenden Konstrukts der vorherigen Verfeinerungsstufe sein. Das Konzept der Typhierarchie ist hier so zu verstehen, dass eine Instanz des Subtyps in allen Belangen dasselbe Verhalten zeigen muss wie eine Instanz des Ausgangstyps, wenn sie wie eine solche behandelt wird. Auf der Ausgangsstufe ist das Verhalten nicht vollständig definiert; die verfeinerte Variante kann also die auf der Ausgangsstufe definierten Verhaltensmerkmale auf verschiedene Weisen erfüllen. Der Verfeinerungsschritt besteht folglich darin, eine geeignete Weise der Verhaltenserfüllung auszu-

wählen. Dieser Auswahlprozess kann von beträchtlicher Komplexität sein, da im Regelfall beliebig viele Möglichkeiten für die genaue Ausgestaltung der Verfeinerung zur Verfügung stehen. Die weitaus meisten dieser Möglichkeiten sind zwar unsinnig; dennoch wird der große Möglichkeitsraum spätestens dann zur Schwierigkeit, wenn der Verfeinerungsprozess automatisiert werden soll. In Beispiel 4.2 ist die überflüssige Aufblähung des Verhaltens leicht ersichtlich. In der Praxis sind jedoch vielfach komplexere Verhaltensbeschreibungen der Regelfall, so dass die Auswahl der optimalen Verfeinerung entsprechend schwerer fällt.

Beispiel 4.2 Verfeinerung von Verhaltenssemantik

```
increment(i)
```

Diese Funktion soll den Wert von i inkrementieren. Der Einfachheit halber sei angenommen, dass „inkrementieren“ als „um eins erhöhen“ definiert ist.

```
increment(i) {
  i = i + 1;
}
```

Die nahe liegende Verfeinerung.

```
increment(i) {
  i = i * 2;
  i = i + 2;
  i = i / 2;
}
```

Eine offensichtlich unsinnige, aber korrekte Verfeinerung.

Sieht man von Ausführungsoptimierungen in modernen Mikroprozessoren ab, ist das letzte Glied in der Kette von Verfeinerungen der Compiler. Er nimmt die letzte Verfeinerungsstufe des Software-Entwicklers, den Quellcode, entgegen und erzeugt ein Maschinenprogramm, dass die spezifizierten Verhaltensmerkmale erfüllt. So genannte optimierende Compiler sind dabei ein praktisches Beispiel für das Konzept der Verhaltens-Subtypen. Verschiedene Compiler werden aufgrund ihrer Optimierungsstrategien aus demselben Quellcode typischerweise nicht denselben Maschinencode generieren, und doch halten alle Maschinenprogramme das im Quellcode spezifizierte Verhalten ein. Optimierende Compiler stellen allerdings einen Grenzfall der reinen Verfeinerung dar, da sie bei der Generierung des Maschinenprogramms gewisse Modifikationen im Vergleich zum Quellcode vornehmen, etwa das Umsortieren von Operationen oder den Ersatz kurzer Schleifen durch wiederholte Codeblöcke (Loop Unrolling). Auf diesem Grund mag das Verhalten des Maschinenprogramms nicht in allen Details dem im Quellcode spezifizierten Verhalten entsprechen, doch sind solche etwaigen Unterschiede nicht relevant, da der Quellcode selbst nur eine womöglich suboptimale Instanz eines abstrakter spezifizierten Verhaltens ist, das das Maschinenprogramm vollständig einhält.

Auf dem Weg von der ersten Grobkonzeption eines Systems bis zum Quellcode werden Verfeinerungsschritte historisch meist von Menschen durchgeführt. In jüngerer Vergangenheit nimmt die Werkzeugunterstützung zu. Entwicklungsumgebungen generieren etwa aus UML-Klassendia-

grammen Quellcode, doch auch hier beschränkt sich die Automatisierung auf die strukturelle Semantik. Aus den Elementen eines Klassendiagramms werden Klassen mit ihren Attributen generiert, wobei auch Typhierarchien und Assoziationen umgesetzt werden; bei Methoden kommt die Generierung jedoch nicht über das Anlegen der Methodenrahmen hinaus. Die eigentlichen Inhalte müssen nach wie vor manuell erstellt werden. Eine Ausnahme bilden Methoden für die Manipulation von Instanzvariablen, so genannte Akzessoren und Mutatoren, in der Praxis oft in Form von `getXY()` / `setXY()`. Ihre Verhaltenssemantik ist im Standardfall jedoch immer gleich, so dass die Generierung lediglich aus dem Einfügen eines trivialen Codefragments besteht. Soll die Methode mehr leisten, muss auch hier wieder manuell eingegriffen werden.

Die Unterstützung durch Entwicklungsumgebungen nimmt dem Software-Entwickler also letztlich nur lästige, gleichförmige Tipparbeit ab. Die Transformationsleistung vom Klassendiagramm zum Quellcode stellt keine Verfeinerung dar, sondern lediglich eine Abbildung, einen Wechsel der Darstellungsform unter Beibehaltung von Abstraktionsgrad und Ausdrucksmächtigkeit. Eine derartige Code-Generierung ist zweifellos sinnvoll, da sie den Entwickler entlastet und Fehler vermeidet, die bei manueller Code-Erstellung unterlaufen können. Sie bringt jedoch keine tatsächliche automatisierte Verfeinerung von Verhaltenssemantik.

An diesem Punkt setzen die in Abschnitt 4.1 vorgestellten modellgetriebenen Architekturen an. Ihr konzeptionelles Ziel ist es, die Entwickler möglichst vollständig von der Verfeinerungs- und Implementierungsarbeit zu entlasten, indem die entsprechenden Abbildungen zwischen den funktionalen Modellen der zu entwickelnden Anwendung sowie die abschließende Umsetzung in ausführbaren Code automatisiert werden. Hintergrund dieses Ansatzes ist die Erkenntnis, dass sich die Schritte bei der Verfeinerung von Semantik abstrahiert spezifizieren lassen, so dass eine algorithmische Implementierung der Verfeinerung möglich ist. Modellgetriebene Architekturen stellen damit einen großen Fortschritt für die Software-Entwicklung dar; die Formalisierung der Verfeinerung ist allerdings eine schwierige Aufgabe, und bislang ist es nur in inhaltlich eng begrenzten Fällen möglich, die Verfeinerung vom Modell bis zum Code tatsächlich vollständig automatisiert durchführen zu lassen.

4.6 Bewertung der betrachteten Spezifikationssprachen

Für die Bewertung der Nutzbarkeit in SQLInteract lassen sich die betrachteten Sprachen und Technologien teilweise zusammenfassen. CWM bietet gegenüber UML keine Erweiterung der Spezifikationsmöglichkeiten für Verhaltenssemantik; seit der Veröffentlichung von UML 1.5 hängt CWM in dieser Hinsicht sogar hinterher. ASL und OAL sind sich im Ansatz und in der Struktur sehr ähnlich, die Unterschiede beschränken sich weitgehend auf syntaktische Details. AsmL verfolgt zwar einen anderen Ansatz, modelliert Verhalten letztlich aber auch über sequenziell abzuarbeitende Modellprogramme.

UML

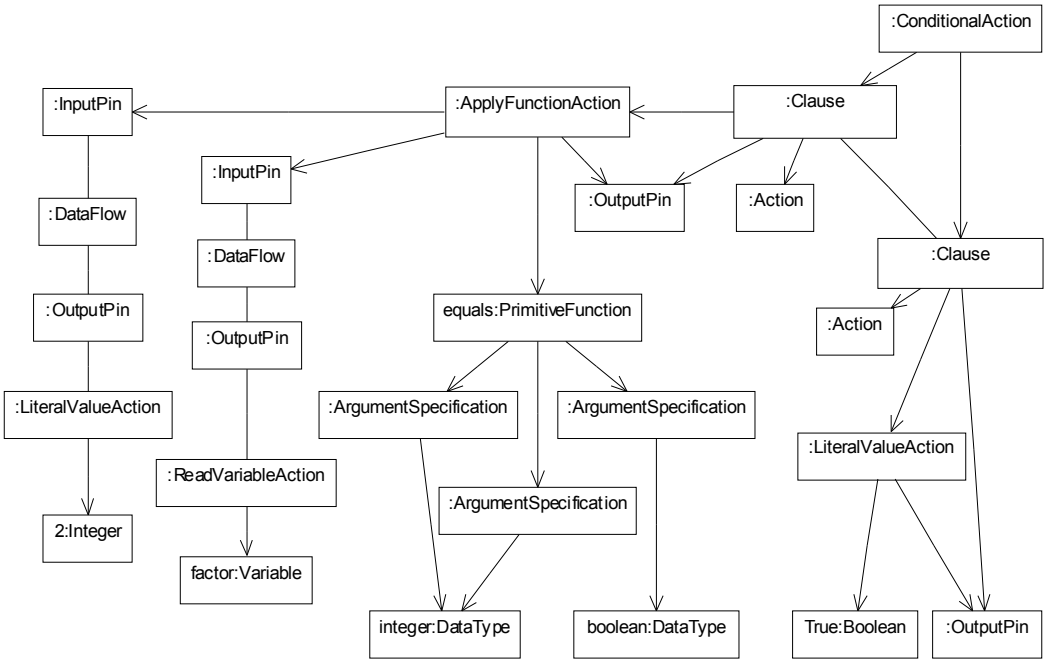
Mit den in UML 1.5 eingeführten *Action Semantics* wird erstmalig eine ernsthafte Modellierung von Verhaltenssemantik in UML möglich. Die schon in früheren Versionen vorhandenen Subpackages von *Behavioral Elements* werden so ergänzt, dass bestehende Modelle um die Spezifikation von Verhaltenssemantik erweitert werden können, ohne dass vorhandene Strukturen nennenswert geändert werden müssen. Die wachsende Modellierungsmächtigkeit von UML

kommt auch der *Model Driven Architecture* zugute: In plattformunabhängigen und plattformspezifischen Modellen kann das Verhalten eines Systems genauer spezifiziert werden.

Der Ansatz, möglichst primitive Aktionen als Grundelemente zu verwenden und komplexere Konstrukte in mehreren Stufen zu definieren, erlaubt eine detaillierte Spezifikation von Verhaltenssemantik in Form von Modellen; im Gegensatz zu anderen graphischen Modellierungstechnologien ist es nicht nötig, große Teile der Information in textueller Form zu notieren. Der Nachteil eines solchen flexiblen und umfassenden Ansatzes liegt in der beachtlichen Komplexität der Modelle. Beispiel 4.3 zeigt den Objektgraphen für eine bedingte Aktion. Soll komplexe Semantik modelliert werden, entstehen schnell enorm große und nur schwer durchschaubare Objektgraphen. Unter dem Aspekt der Übersichtlichkeit und der Benutzerfreundlichkeit sind die Vorteile eines solchen Ansatzes gegenüber konventionellen textuellen Spezifikationssprachen damit fraglich. In der Praxis werden die UML Action Semantics in Entwicklerwerkzeugen daher auch oft mit einer weiteren Spezifikationssprache wie OAL oder ASL gekoppelt, um Verhalten zu modellieren. Für die automatisierte Verarbeitung, insbesondere Verfeinerung und Code-Generierung, ist die Modelldarstellung jedoch definitiv von Vorteil. Da die Spezifikation bereits als Modell vorliegt, entfällt im Vergleich zu textuellen Sprachen das Parsen und die interne Modellkonstruktion.

Beispiel 4.3 UML Action Semantics versus textuelle Darstellung

Für die Darstellung der einfachen Verhaltensbeschreibung
`if (factor==2) doSomething`
wird in UML dieser Objektgraph benötigt:



Das Haupteinsatzgebiet von UML Action Semantics ist die objektorientierte Software-Entwicklung; die Konstrukte sind für die Modellierung von Verhaltenssemantik in möglichst vielen Anwendungsdomänen bestimmt. Um eine breite Nutzung für die Entwicklung ganz verschiedener Softwareprodukte zu ermöglichen, müssen die Elemente der Sprache sehr feingranular sein,

so dass jedes einzelne nur ein Mindestmaß an Semantik beinhaltet und durch die Kombination vieler Elemente nahezu beliebige komplexe Semantik spezifiziert werden kann. Mit der feinen Granularität und der universellen Verwendbarkeit geht also zwingend ein Verlust an Ausdrucksmächtigkeit des einzelnen Elements einher. Eine allgemein einsetzbare Spezifikationssprache besitzt daher nur einen sehr niedrigen Abstraktionsgrad. Der Informationsgehalt eines einzelnen Elements ist gering, um die Verwendung nicht einzuschränken, und erst durch das Zusammenkommen vieler Elemente und unter Betrachtung ihrer Beziehungen kann komplexe Semantik ausgedrückt werden.

Der Preis der Flexibilität der Action Semantics liegt also in der Komplexität der entstehenden Modelle für die Spezifikation von Verhalten. Im betrachteten Anwendungsszenario bietet die Flexibilität jedoch keinen Mehrwert, so dass das Kosten-Nutzen-Verhältnis sehr ungünstig ausfällt. Der Abstraktionsgrad der Action Semantics ist nicht höher als der der angestrebten Implementierungssprache SQL/SPL, die Ausdrucksmächtigkeit dementsprechend auf ähnlichem Niveau. Die Umsetzung von Action Semantics nach SQL/SPL bringt folglich keine echte Verfeinerung der Verhaltenssemantik im Sinne modellgetriebener Architekturen, sondern entspricht eher einer einfachen Abbildung zwischen zwei verschiedenen Darstellungen einer Abstraktionsebene.

In SQLInteract bietet sich aus diesen Gründen die Nutzung von UML Action Semantics nicht an. Für die Darstellung der spezifischen Eingenarten der Anwendungsdomäne wäre ein großer Unterbau an semantischen Konstrukten notwendig, und die typischen Modellstrukturen ließen sich selbst damit nicht vor dem Benutzer kapseln. Die Modellierung des eigentlichen Verhaltens durch den Benutzer würde folglich erheblich erschwert, eine übersichtliche Baumdarstellung des Modells wäre aufgrund der Vielzahl an Elementen praktisch nicht möglich. Eine rein interne Verwendung von Action Semantics im Rahmen der Code-Generierung wäre denkbar, doch auch hier brächte die feingranulare Struktur einen erheblichen Verwaltungsaufwand mit sich, dem keinerlei Vorteile gegenüberständen, da die Flexibilität der Action Semantics bezüglich der Anwendungsdomäne in SQLInteract nicht benötigt wird.

ASL, OAL und AsmL

Die drei textuellen Spezifikationssprachen unterscheiden sich zwar in ihrer genauen Syntax, und AsmL verfolgt auch einen anderen Ansatz als die beiden an UML angelehnten Sprachen; entscheidende grundlegende Eigenschaften sind ihnen jedoch gemein.

Im Gegensatz zu den UML Action Semantics sind diese drei Sprachen textbasiert. Gegenüber den in der Praxis hoch komplexen Objektgraphen ist die textuelle Darstellung von Verhaltenssemantik für Menschen leichter lesbar; gute Lesbarkeit war auch bei allen Sprachen Ziel der Entwicklung. Trotz ihrer textuellen Natur sind Spezifikationen in ASL, OAL und AsmL ausführbar, wobei hier anders als bei UML Action Semantics die Spezifikationen erst von einem Parser eingelesen werden müssen.

Alle drei Sprachen sind für allgemeinen Einsatz konzipiert. Um auch ohne Festlegung auf eine Anwendungsdomäne die Möglichkeit zur Erstellung reichhaltiger, fein spezifizierter Modelle zu bieten, die frei von Mehrdeutigkeiten sind, so dass sie ausgeführt und automatisiert auf Implementierungssprachen abgebildet werden können, müssen die Elemente der Sprachen auch hier sehr feingranular angelegt sein. Daher ähneln diese Spezifikationssprachen letztlich Programmiersprachen; ihre Elemente besitzen nur einen niedrigen Abstraktionsgrad und verfügen über

eine entsprechend geringe Ausdrucksmächtigkeit. Mit dieser Charakteristik eignen sich die Spezifikationssprachen eher für den objektorientierten Software-Entwurf als für die Definition von Verhalten im hier betrachteten Anwendungsszenario.

Fazit

Insgesamt zeigt sich, dass der Einsatz von Spezifikationssprachen für den allgemeinen Gebrauch in SQLInteract problematisch ist. In jedem Fall müssen Kontextinformationen, die sich aus der Anwendungsdomäne „Strukturelle Semantik und Verhaltenssemantik in Datenbanksystemen“ ergeben, explizit formuliert werden, um in solchen Sprachen verarbeitet werden zu können. Die explizite Formulierung dieser Informationen bläht die Spezifikation erheblich auf und reduziert so die Übersichtlichkeit für den Benutzer. Die Problematik der allgemeinen Spezifikationssprachen liegt in ihrer feingranularen Struktur, die zur Erreichung der benötigten Ausdrucksmächtigkeit die Kombination vieler Sprachkonstrukte erfordert. Für die Nutzung in SQLInteract besser geeignet sind Sprachen mit einer gröberen Granularität, die zwar weniger Flexibilität bieten, aber in ihrem Detaillierungsgrad auf den speziellen Einsatzzweck abgestimmt sind.

Einen solchen Ansatz realisieren Sprachen, die auf eine bestimmte Anwendungsdomäne zugeschnitten sind. Derartige Sprachen nennt man *domänenspezifische Sprachen* (*Domain Specific Languages, DSL*s). Die Abstraktionen einer domänenspezifischen Sprache sind mächtiger als die einer allgemeinen Sprache, da die Festlegung auf eine Anwendungsdomäne die Rahmenbedingungen definiert, innerhalb derer die Sprachabstraktionen zu interpretieren sind. Eine Funktion `createTable()` beispielsweise ist im allgemeinen Fall hinsichtlich ihrer Funktionalität nicht eingeschränkt, so dass sich ohne weitere Angaben keine Aussage über ihr Verhalten machen lässt. Ist jedoch als Anwendungsdomäne „Verwaltung von Datenbank-Schemas“ festgelegt, ergibt sich daraus, dass die Methode eine Tabelle in einer Datenbank anlegen soll, und nicht etwa die 3D-Darstellung eines Tisches in einem CAD-Programm. Die höhere Ausdrucksmächtigkeit zeigt sich also darin, dass bei Betrachtung zweier vergleichbar komplexer Sprachkonstrukte das Konstrukt der domänenspezifischen Sprache gegenüber dem Konstrukt der allgemeinen Sprache einen größere Aussagewert hat. Das folgende Kapital stellt das Konzept der domänenspezifischen Sprachen und ihren Nutzen für SQLInteract ausführlich vor.

Kapitel 5 Domänenspezifische Sprachen für DB-Anwendungen

Über den bereits beschriebenen direkten Entwurf eines Datenbank-Schemas mithilfe von CWM-Modell und Bearbeitung von SQL-Befehlen hinaus bietet SQLInteract auch Möglichkeiten für die abstrakte Modellierung von Datenbank-Anwendungen mit beliebigen domänenspezifischen Modellierungssprachen. Bei dieser Vorgehensweise spezifiziert der Benutzer die zu speichernden Daten und die auf ihnen durchführbaren Operationen auf einer abstrakten Ebene. SQLInteract generiert aus dieser Spezifikation ein Datenbank-Schema und entsprechende Stored Procedures, die die gewünschte Funktionalität realisieren. Die generierten Konstrukte lassen sich anschließend an ein Datenbanksystem übertragen, wo sie als Grundlage für eine datenbankintensive Anwendung genutzt werden können. Die Integration der Routinen für die Datenverwaltung in das Datenbanksystem folgt dem Konzept der serverlastigen Funktionalitätsverteilung (*Fat Server, Thin Client*). Bei diesem Konzept wird ein möglichst großer Anteil der Geschäfts- und Präsentationslogik auf dem Server untergebracht. Die Aufgaben von auf einem solchen Server aufsetzenden Clients beschränken sich dann auf die Darstellung der Benutzeroberfläche und die Entgegennahme von Benutzereingaben. Ist die gesamte serverseitige Funktionalität im Datenbank-Server implementiert, sprich man vom *Thick-Database*-Ansatz.

Dieses Kapitel erläutert die Konzepte, die für die generative Entwicklung von DB-Anwendungen relevant sind, und ihren Einsatz in SQLInteract. Zunächst werden Grundlagen zu domänenspezifischen Sprachen, Produktlinien und Generatoren ausgeführt. Im Anschluss wird die in dieser Arbeit als Beispiel gewählte versionierte Speicherung von Objekten in Repositories vorgestellt. Abschließend erfolgt eine Beschreibung der Unterstützung von domänenspezifischen Sprachen und ihrer Anwendung für die Modellierung von Repositories in SQLInteract.

5.1 Grundlagen

Wie das vorige Kapitel zeigt, sind übliche Spezifikationssprachen für den allgemeinen Gebrauch bei der Software-Entwicklung bestimmt. Ihre feingranulare Struktur ermöglicht einen universellen Einsatz unabhängig vom Anwendungsfeld des zu entwickelnden Software-Systems. Als Nachteil dieses Ansatzes ergibt sich die geringe Aussagekraft eines einzelnen Sprachkonstrukts, so dass erst in der Verbindung vieler Konstrukte komplexe Semantik ausgedrückt werden kann.

Solche Sprachen sind damit sehr implementierungsnah; sie ähneln in ihrer Struktur den Implementierungssprachen, mit denen die spezifizierte Semantik realisiert wird. Der mangelnde Abstand zu den Implementierungssprachen ist bei der konventionellen Anwendungsentwicklung möglicherweise zu verschmerzen; sollen die Spezifikationssprachen jedoch, wie es hier der Fall ist, für Modellierungszwecke in einem genau definierten und abgegrenzten Kontext eingesetzt werden, erschwert die strukturelle und semantische Nähe zu den Implementierungssprachen unnötig den Modellierungsprozess. Für derartige Aufgaben sind domänenspezifische Sprachen besser geeignet.

5.1.1 Domänenspezifische Sprachen

Der Begriff *Domäne* bedeutet soviel wie Spezialgebiet. Die formale Spezifikation einer Domäne fällt oft schwer, da die enthaltenen Elemente und die Beziehungen zwischen ihnen beliebig detailliert angeben werden können und sich somit nur schlecht formal ausdrücken lassen. Für die Modellierung von Systemen versteht man unter einer Domäne ein Wissensgebiet, das [CE00]

- in seiner Ausdehnung und Gestalt den Anforderungen seiner Nutzer angepasst ist,
- eine Menge von Konzepten und Fachausdrücken aufweist, die von den Nutzern auf gleiche Weise verstanden werden,
- das Wissen für die Konstruktion von Software-Systemen auf diesem Gebiet beinhaltet.

Eine *domänenspezifische Sprache* (*Domain Specific Language, DSL*) nimmt in ihren Ausdrücken auf das in einer Domäne enthaltene Wissen Bezug, indem Fachausdrücke und Konzepte eingebunden werden, die außerhalb der Domäne mit anderer Bedeutung oder überhaupt nicht definiert sind. Eine Domäne bildet damit einen Kontext, in dem durch die Einigung auf die Bedeutung von Konzepten und Fachausdrücken die damit verbundenen Inhalte in knappen Ausdrücken eindeutig beschrieben werden können, während ohne eine solche Bezugsbasis wesentlich längere und komplexere Ausdrücke nötig wären, um dieselben Inhalte zu beschreiben.

Domänenspezifische Sprachen erreichen durch die Beschränkung auf einen definierten Kontext also einen höheren Abstraktionsgrad als allgemeine Sprachen, und ihre Konstrukte besitzen eine größere Ausdrucksmächtigkeit als die der allgemeinen Sprachen. Je enger dabei die Domäne definiert ist, desto mächtiger sind die Sprachabstraktionen. Ein höherer Abstraktionsgrad ist für den Benutzer von Vorteil, der mit einer Sprache Semantik in einem bestimmten Zusammenhang spezifizieren will, ohne dabei für ihn unwichtige Details mit spezifizieren zu müssen oder sich in komplexen syntaktischen Strukturen zu verstricken. Dieser Vorteil wird zum einen durch den beschränkten Einsatzbereich erkauft; eine domänenspezifische Sprache ist außerhalb ihrer Domäne nicht zu gebrauchen. Zum anderen sind Spezifikationen in Konstrukten einer domänenspezifischen Sprache nicht ohne weiteres ausführbar, sondern sie müssen zunächst in eine allgemein verständliche Form gebracht werden, in der die Interpretation der modellierten Semantik auch ohne Kenntnis des Domänenkontextes möglich ist. Natürlich sind theoretisch auch andere Spezifikations- und sogar Implementierungssprachen nicht direkt ausführbar, sofern es sich nicht um nativen Maschinencode handelt. Bei allgemeinen Sprachen beschränkt sich die Umsetzung jedoch mehr oder weniger auf strukturell-syntaktische Operationen, während bei domänenspezifischen Sprachen im Rahmen der Umsetzung Kontextinformation aus der Domäne explizit in die modellierte Semantik übertragen werden muss. Das Domänenwissen muss daher im Umsetzer (Interpreter, Compiler) integriert sein.

Zur Definition einer Sprache müssen ihre *Syntax* und ihre *Semantik* angegeben werden. Bei der Syntax kann zwischen der *abstrakten Syntax* und der *konkreten Syntax* unterschieden werden. Die abstrakte Syntax beschreibt die abstrakten Syntaxbäume, die bei der Verarbeitung von Spezifikationen für die interne Repräsentation von Sprachkonstrukten zum Einsatz kommt, während die konkrete Syntax die Struktur der Repräsentation von Sprachkonstrukten gegenüber dem Benutzer beschreibt. Zu einer abstrakten Syntax können also mehrere konkrete Syntaxen existieren, zwischen denen verlustfrei abgebildet werden kann; dieses Konzept findet beispielsweise im Intentional Programming Verwendung [Sim95].

Die Semantik einer Sprache lässt sich auf verschiedene Weisen angeben [CE00]:

- Attributgrammatiken
  Angabe von Regeln für die Berechnung der Eigenschaften von Sprachkonstrukten
- Übersetzungssemantik
  Angabe eines Übersetzungsschemas für die Übertragung in eine einfachere Sprache
- Operationssemantik
  Spezifikation der Semantik durch Bereitstellung eines abstrakten Interpreters
- Bedeutungssemantik
  Angabe der Bedeutung von Sprachkonstrukten über mathematische Funktionen
- Axiomatische Semantik
  Definition einer mathematischen Theorie für den Beweis von Programmeigenschaften

Sofern eine geeignete einfachere Sprache zur Verfügung steht, erscheint die Definition von Semantik mithilfe eines Übersetzungsschemas am praktikabelsten.

5.1.2 Generative Programmierung

Generative Programmierung ist eine Fortführung der Konzepte der komponentenbasierten Software-Entwicklung (*Component-Based Software Engineering*, *CBSE*), die sich mit der Erstellung von Software-Systemen aus vorgefertigten und wiederverwendbaren Komponenten befasst. Beim CBSE werden diese Komponenten jedoch von Hand an die Anforderungen eines konkreten Projektes angepasst und zu einem System zusammengefügt, während bei der generativen Programmierung die automatisierte Erstellung von Software-Produkten das Ziel ist.

Grundgedanke der generativen Programmierung ist das Überwinden der derzeitigen Vorgehensweise bei der Software-Entwicklung, nach der von Projekt zu Projekt einzelne Produkte entstehen, die nicht auf einer gemeinsamen systematischen Basis beruhen. Um dieses Ziel zu erreichen, gilt es anstelle von Einzelprodukten ganze Systemfamilien zu entwickeln, deren Mitglieder aus einer zugrunde liegenden gemeinsamen Menge von Komponenten erstellt werden können. Der Zusammenbau der Komponenten zu einem konkreten Produkt soll dann automatisiert durch Generatoren vorgenommen werden.

Eine *Systemfamilie* fasst eine Menge von verwandten Produkten zusammen. Die Ähnlichkeit zwischen den Produkten ist dabei technischer Natur; die Produkte können aus einer zugrunde liegenden Menge von Bausteinen erstellt werden und basieren auf der gleichen Architektur. Im Gegensatz dazu fasst eine *Produktlinie* Produkte nach ihrem Einsatzzweck zusammen. Die Mitglieder einer Produktlinie besitzen gemeinsame Merkmale, die auf die Anforderungen eines bestimmten Marktes zugeschnitten sind. In ihrer technischen Umsetzung können sie sich aller-

dings unterscheiden. Im Idealfall bilden die Mitglieder einer Produktlinie jedoch auch eine Systemfamilie, weisen also Gemeinsamkeiten bei Einsatzzweck und interner Struktur auf.

Einer Systemfamilie liegt ein gemeinsames so genanntes *generatives Domänenmodell* zugrunde, das aus drei Elementen besteht:

- Eine Terminologie zur Spezifikation von Mitgliedern der Systemfamilie, genannt *Problemraum*. Der Problemraum enthält domänenspezifische Konzepte und Merkmale.
- Eine Menge von Implementierungskomponenten, genannt *Lösungsraum*. Diese elementaren Komponenten sind auf maximale Kompatibilität und Flexibilität sowie minimale Redundanz der enthaltenen Semantik optimiert und genügen einer gemeinsamen Architektur.
- Eine Abbildung zwischen Spezifikation im Problemraum und konkreter Realisierung im Lösungsraum, genannt *Konfigurationswissen*. Hierunter fallen neben den eigentlichen Regeln zur Konstruktion eines Produkts aus Implementierungskomponenten anhand einer Spezifikation auch Regeln für Optimierungen, Vorgabewerte und die Erkennung unzulässiger Kombinationen in der Spezifikation. Ein möglichst großer Teil des Konfigurationswissens soll in Programmform vorliegen.

Unter generativer Programmierung versteht man dann ein Software-Engineering-Paradigma, bei dem Systemfamilien derart modelliert werden, dass sich mit einer gegebenen Anforderungsbeschreibung ein optimiertes, genau auf den konkreten Bedarfsfall zugeschnittenes Produkt automatisiert aus elementaren, wiederverwendbaren Implementierungskomponenten konstruieren lässt, indem Konfigurationswissen zur Anwendung kommt [CE00].

Anstelle also wie beim CBSE ein Produkt manuell zusammenzubauen und dabei die Komponenten auch noch an den speziellen Einsatzzweck anpassen zu müssen, erstellt der Entwickler bei der generativen Programmierung eine *Bestellung* oder *Konfiguration*, eine Spezifikation des von ihm gewünschten Produkts innerhalb der Grenzen, die die Systemfamilie vorgibt. Dabei kann er sich auf die Angabe von für ihn relevanten Eigenschaften beschränken und für die restlichen Eigenschaften die Vorgabewerte der Systemfamilie übernehmen. Anhand der Konfiguration konstruiert ein *Generator* dann das fertige Produkt aus den elementaren Komponenten. Der Generator ist also die programmatische Umsetzung des Konfigurationswissens; unter Berücksichtigung der verschiedenen Regeln konfiguriert er die Implementierungskomponenten und baut sie zu einem Software-System zusammen.

Die Angabe einer Bestellung erfolgt mithilfe einer *Konfigurationssprache*, einer domänenspezifischen Sprache speziell für die Spezifikation von Bestellungen. Der Benutzer kann mit ihren Konstrukten ein konkretes Produkt definieren, dass aus den Implementierungskomponenten erstellt werden soll. Die Konfigurationssprache soll es dem Benutzer ermöglichen, seine Bestellung mit einer von ihm gewünschten Detailtiefe anzugeben, so dass er für ihn irrelevante Details nicht spezifizieren muss. Auf diese Weise wird eine unnötige Komplexität der Bestellung vermieden. Vom Benutzer ausgelassene Angaben werden durch den Generator mit Vorgabewerten belegt.

Die Konfigurationssprache für eine bestimmte Domäne kann aus dem Domänenmodell abgeleitet werden. Dabei werden nur Elemente in die Konfigurationssprache übernommen, die variable Eigenschaften der Mitglieder der Systemfamilie beschreiben; invariable Eigenschaften müssen nicht explizit in einer Konfiguration aufgeführt werden. Zu den aus dem Modell übernommenen

Elementen kommen gegebenenfalls noch weitere Eigenschaften, etwa Möglichkeiten für die Angabe von Optimierungskriterien für den Generatorlauf.

Nach der Zusammenstellung der Elemente der Konfigurationssprache muss deren Struktur unter Umständen gegenüber der Struktur des Modells verändert werden, um eine bessere Verständlichkeit für den Benutzer zu erzielen oder Einschränkungen der Implementierungsplattform zu berücksichtigen. Abschließend gilt es Vorgabewerte für die Fälle festzulegen, in denen der Benutzer Angaben auslässt.

Bei der Modellierung einer Konfigurationssprache müssen technische Aspekte und die Benutzerfreundlichkeit der Sprache gegeneinander abgewogen werden. Auf technischer Seite sollte die Sprache möglichst leicht zu erweitern und zu analysieren sein, was beispielsweise durch eine sinnvolle Modularisierung unterstützt wird. Mit Blick auf die Benutzerfreundlichkeit ist eine leicht verständliche Sprache erstrebenswert, die der Benutzer schnell erlernen und effektiv einsetzen kann. Darüber hinaus sollte die Sprache auch effizient sein, das heißt mit wenigen Sprachelementen soll die Formulierung möglichst verschiedener Bestellungen erfolgen können.

5.2 Anwendungsbeispiel: Versionsverwaltung

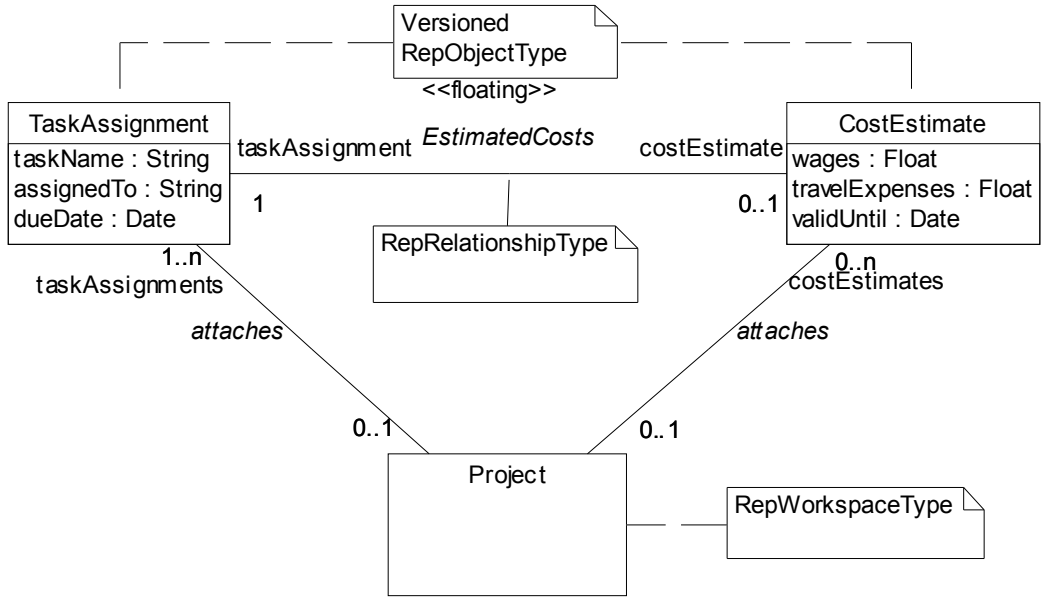
Ein Beispiel für eine Produktlinie sind *Repositories*, hier betrachtet in der Variante der *objektorientierten Repositories*, die Daten in Form von Objekten speichern. In Repositories können Daten *versioniert* gespeichert werden, so dass mehrere Versionen eines bestimmten Objekts parallel existieren können. Für einen gezielten Zugriff auf ein Objekt ist dann neben dem Objektbezeichner auch die Angabe der Version notwendig. Bedarf für eine derartige Datenspeicherung besteht beispielsweise bei Entwicklungs- und Ingenieursanwendungen, bei denen mehrere Benutzer auf einem gemeinsamen Datenbestand arbeiten. Verschiedene Versionen eines im Repository gespeicherten Objekts spiegeln dann zum einen die zeitliche Abfolge der Bearbeitungsschritte wider, zum anderen können mehrere Benutzer parallel an ihrer jeweils eigenen Version arbeiten, ohne untereinander in Konflikte zu geraten.

5.2.1 Objektorientierte Repositories

Objektorientierte Repositories speichern Daten in Form von *Repository-Objekten*. Ein Repository-Objekt ist eine Instanz eines *Repository-Objekttyps*. Es existieren Repository-Objekttypen sowohl für *versionierte* als auch für *unversionierte* Repository-Objekte, so dass der Einsatz versionierter Speicherung auf die Repository-Objekte beschränkt werden kann, bei denen dies für die Anwendung Sinn macht. Bei Repository-Objekttypen für versionierte Objekte ist es möglich, die maximal erlaubte Zahl an direkten Nachfolgeversionen einer Instanz festzulegen. Repository-Objekttypen für unversionierte Objekte können daher technisch als Spezialfall von Repository-Objekttypen für versionierte Objekte mit einer maximalen Zahl von null direkten Nachfolgeversionen betrachtet werden. Zwischen Repository-Objekten können *Repository-Beziehungen* bestehen, die ihrerseits Instanzen von *Repository-Beziehungstypen* sind. Repository-Beziehungstypen beschreiben semantische Verbindungen zwischen Repository-Objekttypen. Im hier vorgestellten Beispiel sind Repository-Beziehungstypen grundsätzlich binär, verbinden also genau zwei Repository-Objekttypen. Sowohl Repository-Objekte als auch Repository-Beziehungen können Attribute enthalten, in denen sich Daten ablegen lassen. Abbildung 5.1 zeigt das Modell eines einfachen Beispiel-Repository für die Speicherung von Projektmanagement-Daten. Die beiden

Repository-Objekttypen für versionierte Objekte *TaskAssignment* und *CostEstimate* sind über den Repository-Beziehungstyp *EstimatedCosts* assoziiert.

Abbildung 5.1 Modell eines Repository



Repository-Objekte können in *Repository-Workspaces* zusammengefasst werden. Ein solcher Repository-Workspace enthält sinnvollerweise semantisch innerhalb eines Arbeitskontextes zusammenhängende Repository-Objekte. Die Zugehörigkeit eines Repository-Objekts zu einem Repository-Workspace wird als *Attachment* bezeichnet und über eine *attaches*-Beziehung modelliert. In einem Repository-Workspace kann zu einem gegebenen Zeitpunkt nur jeweils eine Version eines jeden enthaltenen Repository-Objekts existieren, deren Ein- und Ausgliederung über entsprechende Operationen im Repository erfolgt. Einmal eingegliedert, ist damit innerhalb eines Repository-Workspace Zugriff auf versionierte Repository-Objekte ohne erneute Angabe der Version möglich. Ein Repository-Workspace bildet damit einen Arbeitskontext für Aktionen, die die Manipulation mehrerer Repository-Objekte erfordern. Die Abstraktion von der versionierten Speicherung vereinfacht die manuelle und automatisierte Bearbeitung der enthaltenen Objekte. In einem Repository nach Abbildung 5.1 enthalten Repository-Workspaces vom Typ *Project* ein oder mehrere *TaskAssignments* mit den zugehörigen *CostEstimates*, so dass bei der Bearbeitung alle Daten eines Projekts zusammen vorliegen.

Die Struktur eines Repository, also die enthaltenen Repository-Objekttypen und Repository-Beziehungstypen, wird in einem so genannten *Informationsmodell* beschrieben. Ein solches Informationsmodell lässt sich im Falle eines objektorientierten Repository als UML-Klassendiagramm darstellen, wie es in Abbildung 5.1 der Fall ist. Das Diagramm zeigt Repository-Objekttypen als Klassen und Repository-Beziehungstypen als Assoziationen zwischen den Klassen.

5.2.2 Aspekte der Versionierungsunterstützung

In einem Repository können sowohl versionierte als auch unversionierte Objekte existieren. Die Festlegung, ob ein Objekt versioniert gespeichert wird, geschieht bei der Definition des entsprechenden Objekttyps. Instanzen von Objekttypen für versionierte Objekte können dann im Repository ausschließlich versioniert existieren. Beim Zugriff auf das Repository bedeutet der daraus resultierende Zwang, stets zu einer Objekt-ID auch die gewünschte Version anzugeben, jedoch eine Steigerung der Komplexität im Vergleich zur unversionierten Speicherung, sowohl bei manueller Datenabfrage durch einen Benutzer als auch bei der automatisierten Datenverarbeitung auf algorithmischer Basis.

Gerade bei Entwurfsanwendungen werden Daten aus einem Repository typischerweise für eine recht lange Zeit bearbeitet. Anstelle des üblichen ACID-Transaktionsmodells, dessen Sperrstrategie für eine kurze Transaktionsdauer optimiert ist, kommen daher `checkout`- und `checkin`-Operationen zum Einsatz, die die betroffenen Objekte im Repository als gesperrt markieren und gegebenenfalls eine lokale Arbeitskopie anlegen.

Um den Zugriff auf im Repository gespeicherte Daten übersichtlicher zu gestalten, existieren mehrere Ansätze, die eine versionsfreie Handhabung versionierter Objekte erlauben:

- *Regelbasierte Auswahl*
  Beim Zugriff auf ein versioniertes Objekt ohne Angabe einer Version wählt das Repository anhand von vorgegebenen Regeln eine Version aus, die zurückgeliefert wird. Die Auswahlregeln können je nach Art des Repository und Typ des angeforderten Objekts verschiedene Kriterien beinhalten und sowohl vom Benutzer als auch vom Hersteller des Repository definiert sein. Ein typisches Beispiel für eine Auswahlregel ist die Auswahl der neuesten vorhandenen Version (*latest version rule*).

- Explizite Festlegung (*Pinning*)
  Der Benutzer kann für ein Objekt eine bevorzugte Version oder Voreinstellung (*Default*) angeben, die bei Anfragen ohne Versionsangabe zurückgeliefert werden soll. Die Festlegung einer solchen bevorzugten Version wird als *Pinning* bezeichnet, die Rücknahme der Festlegung als *Unpinning*.

- Arbeitskontexte (*Workspaces*)
  Arbeitskontexte fassen inhaltlich verwandte Objekte zusammen. Der Typ und die Anzahl der in einen Arbeitskontext einfügbaren Objekte werden bei der Definition des Arbeitskontext-Typen im Informationsmodell des Repository festgelegt. Ebenso lässt sich hier angeben, ob die Objekte *exklusiv* oder *nichtexklusiv* eingefügt werden (*exclusive attachment* vs. *non-exclusive attachment*). Bei exklusiver Einfügung wird ein eingefügtes Objekt für weitere Einfügeoperationen gesperrt, kann also in keinen weiteren Arbeitskontext eingefügt werden, während dies bei nichtexklusiver Einfügung möglich ist. Zur Laufzeit kann immer nur eine Version eines Objektes in einen Arbeitskontext eingefügt werden. Auf diese Weise kann innerhalb des Arbeitskontextes auf die Angabe der Version verzichtet werden, so dass eine von der versionierten Speicherung abstrahierende Objektmanipulation ermöglicht wird. Der Benutzer kann also beispielsweise für eine längere Zeit in Anspruch nehmende Entwurfstätigkeit einen Arbeitskontext spezifizieren und dabei einmalig für die beteiligten Objekte festlegen, welche Version eingefügt werden soll. Nach dem Checkout des Arbeitskontextes aus dem Repository ist dann während der Arbeit an den enthaltenen Objekten ein übersichtlicherer versionsfreier Zugriff möglich. Im Repository selbst werden die Objekte stets versioniert gespeichert.

Gleitende Beziehungs-Enden

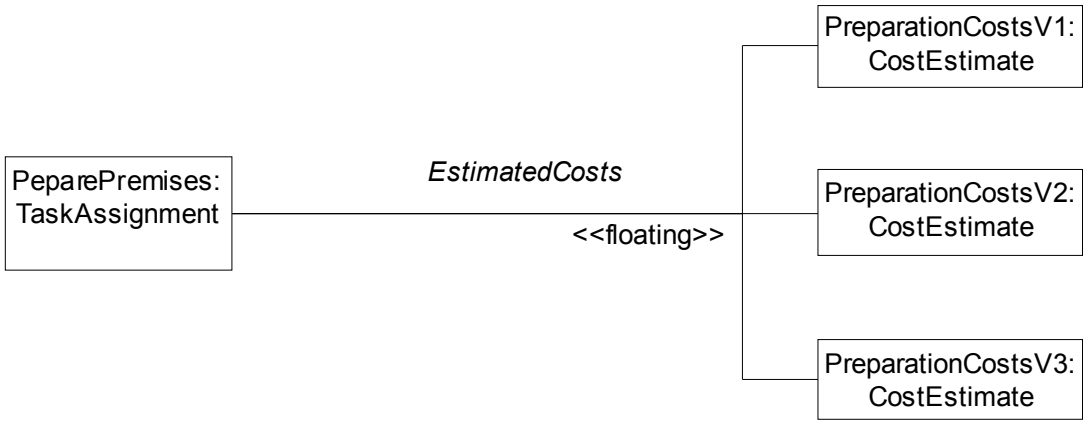
Gleitende Beziehungs-Enden (*Floating Relationship Ends*) erlauben eine detaillierte Modellierung der semantischen Beziehung zwischen zwei Objekten, die über eine Beziehung assoziiert sind. Ein gleitendes Beziehungs-Ende macht nur Sinn, wenn es mit einem versionierten Objekt verbunden ist. Ist beispielsweise ein versioniertes Objekt A mit einem (nicht unbedingt versionierten) Objekt B assoziiert, so ist diese Assoziation von B ausgehend nicht notwendigerweise auf eine bestimmte Version von A beschränkt, sondern sie kann sich auch auf mehrere oder sogar alle Versionen ausdehnen. Die Menge der potenziell assoziierten Versionen seitens A wird als *Kandidatenmenge* der Versionen (*Candidate Version Collection*) bezeichnet.

Ob ein Beziehungs-Ende gleitend ist, wird im Informationsmodell des Repository bei der Definition des Beziehungstyps festgelegt. Von entscheidender Bedeutung ist dabei, dass ein gleitendes Beziehungs-Ende nicht mit einer Änderung der Kardinalität gleichzusetzen ist. Der Benutzer kann zur Laufzeit für eine konkrete Beziehung mit gleitendem Beziehungs-Ende die Kandidatenmenge bearbeiten. Für den Zugriff auf ein versioniertes Objekt über eine Beziehung mit gleitendem Beziehungs-Ende gelten ähnliche Vorgehensweisen, wie sie bereits für den Umgang mit versionierten Objekten im Allgemeinen beschrieben wurden:

- Der Benutzer kann in der Kandidatenmenge eine *bevorzugte Version* auswählen, die beim Zugriff über die Beziehung zurückgeliefert wird.
- Ist vom Benutzer keine bevorzugte Version festgelegt worden, kann das System *regelbasiert* eine Version auswählen und diese zurückliefern. Typischerweise handelt es sich hierbei um die neueste Version des angeforderten Objekts.
- Der Benutzer kann sich jedoch auch die *gesamte Kandidatenmenge* zurückliefern lassen und selbst entscheiden, mit welcher Version des Objekts er weiter arbeiten möchte.

In Abbildung 5.1 wird davon ausgegangen, dass die Beziehung *EstimatedCosts* zwischen *TaskAssignment* und *CostEstimate* auf der Seite von *CostEstimate* ein gleitendes Ende besitzt. Die Kardinalität der Beziehung bleibt bei 1:1, es können jedoch mehrere Versionen von *CostEstimates* als Assoziationskandidaten existieren, unter denen erst zu einem späteren Zeitpunkt die tatsächlich mit *TaskAssignment* assoziierte Version ausgewählt wird. In Beispiel 5.1 ist das TaskAssignment *PreparePremises* über ein gleitendes Beziehungs-Ende mit drei Versionen des CostEstimate *PreparationCosts* verbunden. Eine solche Konstellation ist etwa bei einem großen Bauprojekt denkbar, bei dem sich erst im Laufe der Zeit herausstellt, wie stark der Boden belastet ist, so dass die Kostenplanung für die Sanierung mehrfach nach oben korrigiert werden muss. Ohne weitere Vorgaben des Benutzers liefert das Repository bei einer Anforderung von *PreparationCosts* über die Beziehung *EstimatedCosts* die neueste Version V3 zurück, so dass die 1:1-Kardinalität der Beziehung gewahrt wird. Will der Projektmanager jedoch beispielsweise gegenüber der Bauleitung die Situation schönfärben, so wird er V2 als bevorzugte Version festlegen, so dass bei einer Abfrage die älteren Zahlen zurückgeliefert werden. Auch in diesem Fall bleibt die Kardinalität der Beziehung bei 1:1. Sollte die Bauleitung den Plan des Projektmanagers allerdings durchschauen, wird sie sich die gesamte Kandidatenmenge des gleitenden Beziehungs-Endes zurückliefern lassen. Selbst in diesem Fall ändert sich die Kardinalität der Beziehung nicht, weil letztlich zu jedem Zeitpunkt nur eine Version von *PreparationCosts* aktuell ist.

Beispiel 5.1 Gleitendes Beziehungs-Ende



5.2.3 Operationen in objektorientierten Repositories

Bei der Verwendung objektorientierten Repository werden neben den typischen Operationen für das Schreiben und Lesen von Daten auch Operationen für die Versionsverwaltung, lange Transaktionen und Navigation zwischen den Objekten benötigt. Weiterhin gilt es festzulegen, inwiefern Operationen sich über Beziehungen auch auf assoziierte Objekte auswirken (*Operations-Propagierung*, *Propagation*). In diesem Abschnitt werden die verschiedenen Operationen und die Möglichkeiten, ihre Propagierung zu spezifizieren, vorgestellt.

Objekt- und Versionsverwaltung

Neben den im Informationsmodell des Repository spezifizierten Attributen, die praktisch die Nutzdaten der Objekte darstellen, enthält jedes Objekt eine *Objekt-ID*, die der Identifikation der Instanz dient, sowie eine *Versions-ID*, die für die Verwaltung der Versionsabfolge benötigt wird. Bei unversionierten Objekten lautet die Versions-ID immer 0, bei versionierten Objekten ist jedoch die Angabe beider IDs nötig, um eine Version einer Instanz eindeutig zu definieren. Aus diesem Grund werden Versions-ID und Objekt-ID zur *Global-ID* zusammengefasst, über die jedes Objekt und jede Version im Repository eindeutig angesprochen werden können.

Zusätzlich zu den IDs enthält jedes Objekt ein Feld `checkedOut`, in dem gespeichert wird, ob ein Benutzer das Objekt per Checkout für längerfristige Bearbeitung gesperrt hat. Versionierte Objekte enthalten darüber hinaus zwei weitere Felder. Im Feld `frozen` wird vermerkt, dass die betreffende Version des Objekts gegen weitere Änderungen gesperrt wurde, während das Feld `predecessor` einen Verweis auf den direkten Vorgänger in der Versionshierarchie enthält. Die folgenden Operationen benötigen die genannten Felder für die Verwaltung der Versionshierarchie und die Unterstützung des Datenzugriffs.

- `new` - Legt eine neue Instanz eines Objekttyps an. Die Attribute werden auf voreingestellte Standardwerte oder die Werte übergebener Parameter gesetzt. Die Instanz erhält eine Objekt-ID. Handelt es sich um ein versioniertes Objekt, wird die Version auf 0 gesetzt, der Verweis auf den direkten Vorgänger auf null.
- `copy` - Legt ähnlich wie `new` eine neue Instanz eines Objekttyps an, übernimmt aber die Werte der Attribute von einer bestehenden Instanz. Wie bei `new` wird eine Objekt-ID vergeben und im Falle eines versionierten Objekts die Version auf 0 sowie der Verweis auf den

direkten Vorgänger auf `null` gesetzt. Das Feld `frozen` wird in jedem Fall auf `false` gesetzt, auch wenn es in der Quellinstanz für die `copy`-Anweisung bereits auf `true` gesetzt sein sollte. Auf diese Weise können also die Attributwerte, die ein per `freeze` gesperrtes Objekt enthält, weiter bearbeitet werden.

- `freeze` - Sperrt eine Objektversion gegen weitere Bearbeitung und Änderung der Attributwerte. Die Version wird mit dieser Operation also festgeschrieben. Sollen doch noch Änderungen an den Werten vorgenommen werden, muss per `createSuccessor` eine Nachfolgeversion oder per `copy` ein neues Objekt angelegt werden.

- `createSuccessor` - Legt eine Nachfolgeversion eines versionierten Objekts an, wenn die maximale Anzahl an direkten Nachfolgern noch nicht erreicht ist. Die Objekt-ID und Attributwerte werden von der aktuellen Version übernommen, die Versions-ID wird erhöht. Das Feld `frozen` wird ähnlich wie bei `copy` auf `false` gesetzt, im Feld `predecessor` wird der Verweis auf die aktuelle Version eingetragen.

- `attach/detach` - Per `attach` wird ein Objekt in einen Arbeitskontext eingefügt, sofern es nicht bereits in einen anderen Arbeitskontext exklusiv eingefügt wurde und deswegen gegen weitere Einfügeoperationen gesperrt ist. Die Struktur des Arbeitskontextes mit den Angaben über einfügbare Objekttypen und Exklusivität der Einfügeoperation wird im Informationsmodell des Repository definiert; `attach` dient lediglich dazu, eine gemäß der Struktur zulässige Instanz auszuwählen und die *attaches*-Beziehung zwischen Arbeitskontext und Instanz herzustellen. `Detach` macht die Aktionen von `attach` rückgängig und entfernt die Instanz aus dem Arbeitskontext.

- `checkout/checkin` - Mit `checkout` wird eine längerfristige Sperre auf einem Objekt angelegt, die im Feld `checkedOut` entsprechend vermerkt wird. Die `checkout`-Operation sperrt ein Objekt über eine Transaktion nur für kurze Zeit, um den Vermerk im Feld `checkedOut` anzulegen und gegebenenfalls eine lokale Arbeitskopie für den Benutzer zu erstellen. Während der Benutzer anschließend nach Belieben Änderungen vornehmen kann, bestehen im Repository keine potenziell Blockaden auslösenden Sperren. Die Verwaltung des nebenläufigen Zugriffs auf Repository-Objekte wird durch die Umgehung der Datenbank-eigenen Sperren allerdings komplexer. Der Benutzer, der ein Objekt per `checkout` gesperrt hat, kann nach Abschluss der Bearbeitung die Änderungen mit `checkin` im Repository speichern. Die Werte des gesperrten Objekts werden neu gesetzt und die Sperre im Feld `checkedOut` aufgehoben. Ist ein Objekt bereits mit einer Bearbeitungssperre versehen, können andere Benutzer es nicht ein weiteres Mal sperren, sondern sie müssen statt dessen etwa per `createSuccessor` eine neue Version erstellen, mit der sie dann arbeiten können. Hier zeigt sich, dass in einer Mehrbenutzerumgebung verschiedene Versionen eines Objekts nicht nur im Rahmen der Entwicklung des Objekts im Laufe der Zeit, sondern auch bei der parallelen Bearbeitung durch verschiedene Benutzer entstehen können.

- *Operationen für die Navigation im Versionsbaum*
  Für die Navigation im Versionsbaum eines versionierten Objekts existieren mehrere Operationen. Mit `getRoot` wird die Wurzel des Baums, also die erste Version des Objekts angesprochen. Die Operationen `getPredecessor` und `getSuccessors` liefern den unmittelbaren Vorgänger beziehungsweise die direkten Nachfolger einer gegebenen Version. Mit `getAlternatives` schließlich erhält man die Versionen, die im Versionsbaum auf derselben Stufe stehen wie die gegebene Version, also die Geschwister.

Navigation zwischen Objekten

Jeder Objekttyp im Informationsmodell des Repository definiert Methoden für die Navigation entlang der Beziehungen zwischen Objekten. Für ein konkretes Objekt im Repository aufgerufen, liefern diese Methoden die assoziierten Objekte zurück. Diese Art der Navigation wird als *direkte Navigation* bezeichnet. Zu jeder Beziehung existiert dabei eine eigene Methode, deren Name nach dem Schema `get<Rolename>OfObject<ObjectType>` gebildet wird. Der Methodenname nimmt also Bezug auf den Typ des Objekts, von dem aus navigiert wird, und den Rollennamen, den das andere Ende der zu navigierenden Beziehung trägt. Im Beispiel aus Abbildung 5.1 würde also `getCostEstimateOfObjectTaskAssignment()`, aufgerufen mit der Global-ID eines *TaskAssignment*-Objekts als Parameter, das assoziierte *CostEstimate*-Objekt zurückliefern. Handelt es sich bei der navigierten Beziehung um eine Beziehung mit gleitendem Ende, wird bei einem solchen Methodenaufruf die vom Benutzer festgelegte bevorzugte Version (*Pinned Version*) oder die regelbasiert ausgewählte Version zurückgeliefert. Bei Beziehungen mit gleitendem Ende existieren zusätzlich weitere Methoden mit dem Namensschema `get<Rolename>OfObject<ObjectType>UseCVC` und, wenn der Objekttyp mit einem oder mehreren Arbeitskontext-Typen assoziiert ist, `get<Rolename>OfObject<ObjectType>Within<WorkspaceType>`. Die Methode mit dem Suffix `UseCVC` liefert anstelle der bevorzugten Version die gesamte Kandidatenmenge, so dass der Benutzer wählen kann, mit welcher Version er weiterarbeiten möchte. Die Methoden mit dem Suffix `Within<WorkspaceType>` richten sich bei der Auswahl der zurückzuliefernden Version nicht nach Benutzervorgaben oder Regelsätzen, sondern liefern die Version zurück, die im selben Arbeitskontext wie das Objekt eingefügt ist, von dem die Navigation ausgeht.

Bei der *indirekten Navigation* wird in mehreren Schritten vorgegangen. Im ersten Schritt wird von einer Methode der Form `get<RelationshipType>OfObject<ObjectType>` die Menge aller Beziehungsinstanzen des angegebenen Typs zurückgeliefert, an denen das im Parameter übergebene Objekt teilnimmt. Im Beispiel würde `getEstimatedCostsOfObjectTaskAssignment()`, angewandt auf eine Instanz von *TaskAssignment*, nur eine Beziehungsinstanz von *EstimatedCosts* zurückliefern, da die Kardinalität der Beziehung 1:1 beträgt. Ähnlich wie bei der direkten Navigation existiert auch für den ersten Schritt bei der indirekten Navigation im Falle einer Beziehung mit gleitendem Ende eine weitere Methode `get<RelationshipType>OfObject<ObjectType>UseCVC`, die zu jeder Beziehungsinstanz nicht nur das Resultat mit der Vorgabeversion, sondern ein Resultat für jedes Element der Kandidatenmenge zurückliefert. Die Methode `get<RelationshipType>OfObject<ObjectType>WithinWorkspace` beschränkt die zurückgelieferten Beziehungsinstanzen auf diejenigen, die sich im selben Arbeitskontext befinden wie das Objekt, von dem die Navigation ausgeht. Eine Beziehung wird dabei als im Arbeitskontext enthalten angesehen, wenn die Objekte an ihren beiden Enden im Arbeitskontext enthalten sind.

Im zweiten Schritt der indirekten Navigation kann dann, auf den Resultaten des ersten Schrittes aufbauend, auf die Objekte zugegriffen werden, die durch die Beziehungsinstanzen assoziiert werden. Die entsprechenden Methoden tragen Namen nach dem Schema `getObject<Rolename>Of<RelationshipType>`. Im Beispiel würde man mit der Methode `getObjectTaskAssignmentOfEstimatedCosts` auf das *TaskAssignment*-Objekt, mit `getObjectCostEstimateOfEstimatedCosts` entsprechend auf das *CostEstimate*-Objekt zugreifen, das mit einer Beziehung vom Typ *EstimatedCosts* verbunden ist.

Propagierung von Operationen

Bei der Durchführung von Operationen auf Objekten im Repository kann es sinnvoll sein, diese Operationen über Beziehungen propagieren zu lassen, also auf den assoziierten Objekten automatisiert dieselben Operationen auch durchzuführen. Beispielsweise ist es praktisch, wenn beim Einfügen eines *TaskAssignment*-Objekts in einen Arbeitskontext das zugehörige *CostEstimate*-Objekt ebenfalls in den Arbeitskontext aufgenommen wird. Um dies zu erreichen, muss die Operation `attach` über die Beziehung *EstimatedCosts* von *TaskAssignment* nach *CostEstimate* propagiert werden. Die Propagierung von Operationen wird bei der Definition von Beziehungstypen im Informationsmodell des Repository festgelegt und ist eine Eigenschaft der Beziehungs-Enden. Im Einzelnen lässt sich die Propagierung für die Operationen `attach`/`detach`, `checkout`/`checkin`, `new`, `copy`, `freeze` und `createSuccessor` spezifizieren.

5.2.4 Repositories als Produktlinie

Wie die Ausführungen des vorstehenden Abschnitts zeigen, können die Eigenschaften eines Repository in seinem Informationsmodell spezifiziert werden. Aus diesem Grund unterscheiden sich Repositories beispielsweise hinsichtlich der in ihnen verwalteten Daten oder der Propagierung von Operationen. In jedem Fall dienen Repositories jedoch der versionierten Speicherung von Daten, so dass sie als Produktlinie für den Bereich der Datenverwaltung in Mehrbenutzerumgebungen und Entwurfsanwendungen betrachtet werden können.

Darüber hinaus lassen sich Repositories auch als Systemfamilie realisieren. Technische Gemeinsamkeiten bestehen etwa bei der Unterstützung versionierter Speicherung, dem Konzept der Arbeitskontexte sowie den Methoden für die Datenverwaltung und den navigierenden Zugriff. Als Grundlage für die Datenspeicherung dient ein relationales Datenbanksystem. Unterschiede zeigen sich bei der genauen Gestalt der zu speichernden Nutzdaten und den Anforderungen an die Versionsverwaltung. Insgesamt sind jedoch die strukturellen Ähnlichkeiten zwischen Repositories so groß, dass es möglich ist, Repositories nach dem Ansatz der generativen Programmierung aus einer Menge von Basiskomponenten zu erstellen.

Ein Hersteller von Repositories wird also seine Produktlinie sinnvollerweise als Systemfamilie realisieren, anstelle jeden Kundenauftrag von Grund auf neu zu implementieren. Eine andere Strategie bestünde darin, ein generisches Repository zu entwickeln, das universell in verschiedenen Anwendungsfällen einsetzbar ist. Ein solches generisches Repository kann jedoch naturgemäß kaum auf den konkreten Bedarf des Kunden zugeschnitten werden. Die allgemeine Verwendbarkeit resultiert in einer unnötigen Komplexität der zugrunde liegenden Datenstrukturen und Operationen, was sich im Betrieb in höherem Platzbedarf und geringerer Performanz niederschlägt. Dem Kunden nützt die universelle Eignung des Repository im konkreten Anwendungsfall nichts, so dass er lediglich ihre Nachteile zu spüren bekommt.

Sind die Repositories des Herstellers jedoch als Systemfamilie realisiert, so kann der Kunde seine Anforderungen in Form einer *Bestellung* (*Konfiguration*) dem Hersteller mitteilen, woraufhin dieser ein speziell auf den Bedarf zugeschnittenes Repository erstellt. Eine solche Bestellung umfasst Angaben über versioniert und unversioniert zu speichernde Objekttypen, über die Beziehungen, die zwischen Objekten bestehen können, über die Typen von Arbeitskontexten, die angelegt werden können und über die Propagierung von Operationen über Beziehungen.

Zur Spezifizierung einer Bestellung bietet sich nach den Ausführungen in Abschnitt 5.1 eine domänenspezifische Konfigurationssprache an. Ihre Sprachkonstrukte sind speziell auf die

Anwendung zugeschnitten, so dass der Kunde mit möglichst einfachen Mitteln seine Anforderungen formulieren kann. Die hier im Folgenden als Beispiel vorgestellte Konfigurationssprache ist in ihrer Struktur an die *Data Definition Language* (*DDL*) von SQL angelehnt, was die Verwandschaft zwischen Repositories und Datenbanken widerspiegelt und durch den Bekanntheitsgrad von SQL die Erlernbarkeit verbessert.

- `create object type`
  Diese Anweisung spezifiziert einen Objekttyp für unversionierte Objekte und seine Attribute. Ihre Schablone lautet

```
create object type <TypeName> {
 attribute <AttributeName> <AttributeType>,
 attribute <AttributeName> <AttributeType>,
 ...
}
```

- `create versioned object type`
  Diese Anweisung spezifiziert einen Objekttyp für versionierte Objekte, seine Attribute und die maximale Zahl der direkten Nachfolgeversionen, die eine Instanz dieses Typs besitzen darf. Die Schablone lautet

```
create versioned object type <TypeName> {
 attribute <AttributeName> <AttributeType>,
 attribute <AttributeName> <AttributeType>,
 ...
 maxSuccessors <NoOfSuccessors>
}
```

  Bis auf die zusätzliche Angabe der Nachfolgerzahl und das Schlüsselwort `versioned` gleicht diese Anweisung also der Anweisung zur Spezifikation von Objekttypen für unversionierte Objekte.

- `create relationship type`
  Diese Anweisung spezifiziert einen binären Beziehungstyp, der zwei Objekttypen miteinander assoziiert. Neben der Angabe der Multiplizitäten und der Rollennamen der Beziehungs-Enden lässt sich hier auch festlegen, ob ein Beziehungs-Ende gleitend sein soll. Weiterhin wird mit dieser Anweisung bestimmt, welche Operationen in welche Richtung über die Beziehung propagiert werden. Die Schablone lautet

```
create relationship type <TypeName> {
 connects <ObjectType> (<minMultiplicity>, <maxMultiplicity>)
  rolename <Rolename> [floating],
 connects <ObjectType> (<minMultiplicity>, <maxMultiplicity>)
  rolename <Rolename> [floating],
 propagates <Operation> on <ObjectType>,
 propagates <Operation> on <ObjectType>,
```

```
 ...
}
```

- `create workspace type`
  Diese Anweisung spezifiziert einen Arbeitskontext-Typ. Dazu werden die Objekttypen aufgelistet, die in den Arbeitskontext eingefügt werden können. Zu jedem Objekttyp muss dabei die Multiplizität der *attaches*-Beziehung zwischen Arbeitskontext und eingefügten Objekten festgelegt werden, die angibt, wie viele Instanzen des betreffenden Objekttyps in den Arbeitskontext eingefügt werden können. Weiterhin werden Rollennamen für die eingefügten Objekte definiert. Über das Schlüsselwort `exclusive` wird die Einfügeoperation als exklusiv definiert, so dass eingefügte Objekte gegen Einfügung in andere Arbeitskontexte gesperrt werden. Die Schablone lautet:

```
create workspace type <TypeName> {
 [exclusive] attachment <ObjectType> (<minMultiplicity>,
  <maxMultiplicity>) rolename <Rolename>,
 [exclusive] attachment <ObjectType> (<minMultiplicity>,
  <maxMultiplicity>) rolename <Rolename>,
 ...
}
```

Mit diesen vier Anweisungen lässt sich das Informationsmodell eines Repository hinreichend genau spezifizieren, um die automatisierte Erstellung des fertigen Produkts durch einen Generator zu ermöglichen. Beispiel 5.2 zeigt die Anweisungen, die für die Spezifikation des in Abbildung 5.1 dargestellten Informationsmodells notwendig sind.

5.3 Repository-Spezifikation mit SQLInteract

SQLInteract beinhaltet über die in Kapitel 3 beschriebenen Möglichkeiten für die direkte Modellierung von Datenbank-Schemas hinaus auch Konzepte der generativen Programmierung. Die unterstützte Systemfamilie ist die Gruppe der *datenbankintensiven Anwendungen*. Bei dieser Gruppe von Anwendungen wird ein möglichst großer Teil der Funktionalität direkt im Datenbanksystem realisiert. SQL:1999 [ANS99] spezifiziert für die Implementierung von Verhaltenssemantik im Datenbanksystem unter anderem *benutzerdefinierte Routinen* (*User Defined Routines*, *UDR*, oft auch als *Stored Procedures* bezeichnet) und *ECA-Regeln* (*Event-Condition-Action Rules*). In Verbindung mit *benutzerdefinierten Datentypen* (*User Defined Types*, *UDT*) lassen sich so komplexe Operationen auf den gespeicherten Daten durchführen.

5.3.1 Überblick

In Abschnitt 5.2 wurden objektorientierte Repositories vorgestellt, die als Produktlinie und Systemfamilie betrachtet werden können. Für die Realisierung eines solchen Repository bietet sich eine datenbankintensive Anwendung an. Aus diesem Grund wurde die Spezifikation eines Repository und die anschließende Generierung des entsprechenden Produkts als Anwendungsszenario

Beispiel 5.2 Spezifikation eines Repository mithilfe einer Konfigurationssprache

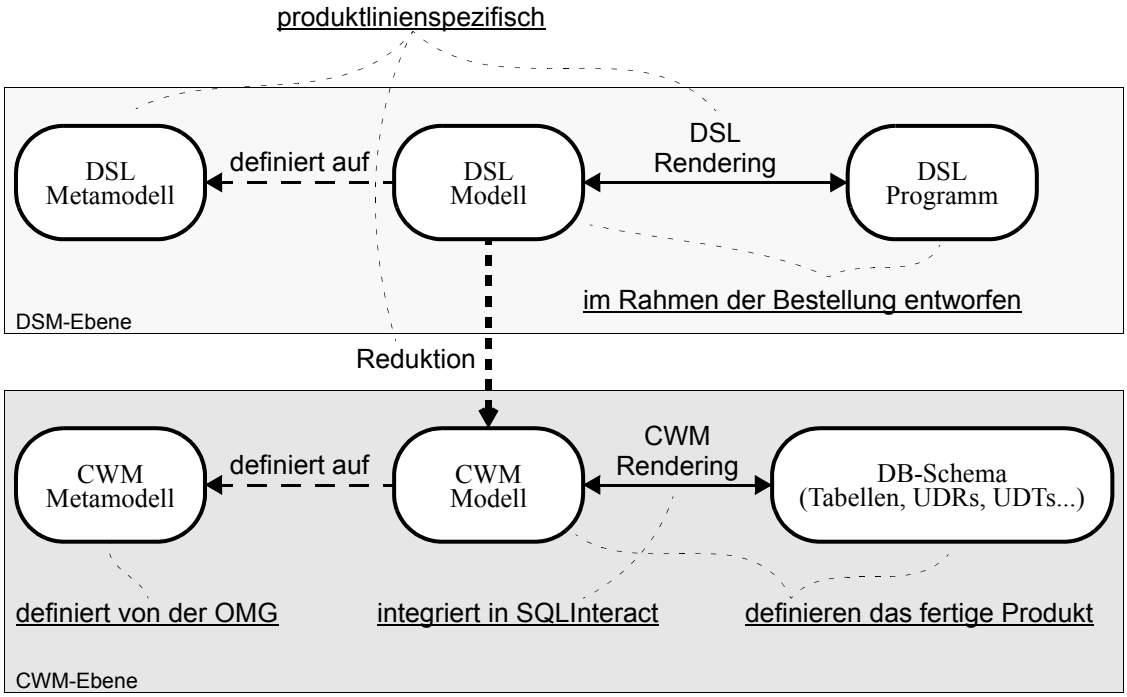
```
create versioned object type TaskAssignment {
 attribute taskName String,
 attribute assignedTo String,
 attribute dueDate Date,
 maxSuccessors 4
}

create versioned object type CostEstimate {
 attribute wages Float,
 attribute travelExpenses Float,
 attribute validUntil Date,
 maxSuccessors 4
}

create workspace type Project {
 attachment TaskAssignment (1, *) rolename taskAssignments,
 attachment CostEstimate (0, *) rolename costEstimates
}

create relationship type EstimatedCosts {
 connects TaskAssignment (1, 1) rolename taskAssignment,
 connects CostEstimate (0, 1) rolename costEstimate floating,
 propagates attachDetach on TaskAssignment,
 propagates freeze on TaskAssignment,
 propagates checkoutCheckin on TaskAssignment
}
```

für SQLInteract gewählt. Abbildung 5.2 zeigt zunächst die Zusammenhänge zwischen den verschiedenen Modellen und Darstellungsformen.

Die Architektur von SQLInteract gliedert sich anhand der zugrunde liegenden Modelle in zwei Ebenen, die *CWM-Ebene* und die *DSM-Ebene*. Die Elemente der CWM-Ebene wurden in Kapitel 3 erläutert. Hier kann der Benutzer ein Datenbank-Schema direkt modellieren, wobei die Darstellung als CWM-Modell in Form eines Baums und textuell als DDL-Code erfolgt. Die Abbildung zwischen den beiden Darstellungsformen geschieht über das so genannte *CWM-Rendering*. Der Begriff des Rendering ist dem Intentional Programming entnommen und bezeichnet die Darstellung einer modellierten Abstraktion in einer bestimmten Form. Das Rendering ist grundsätzlich nicht auf eine Zielsprache beschränkt; um eine Übertragung des Schemas an verschiedene Datenbanksysteme zu ermöglichen, ist beispielsweise die Unterstützung unterschiedlicher DDL-Dialekte der verbreiteten Datenbankprodukte sinnvoll. In Kapitel 7 werden die Konzepte des Intentional Programming näher betrachtet.

Auf der DSM-Ebene kann der Benutzer unter Abstraktion von Implementierungsdetails das Modell eines Repository spezifizieren (*Domain-Specific Model, DSM*), also im Sinne der generativen Programmierung eine Bestellung oder Konfiguration in einer domänenspezifischen Sprache erstellen. Die Bedienung der DSM-Ebene verläuft analog zur Bedienung der CWM-Ebene;

Abbildung 5.2 Modellzusammenhänge in SQLInteract



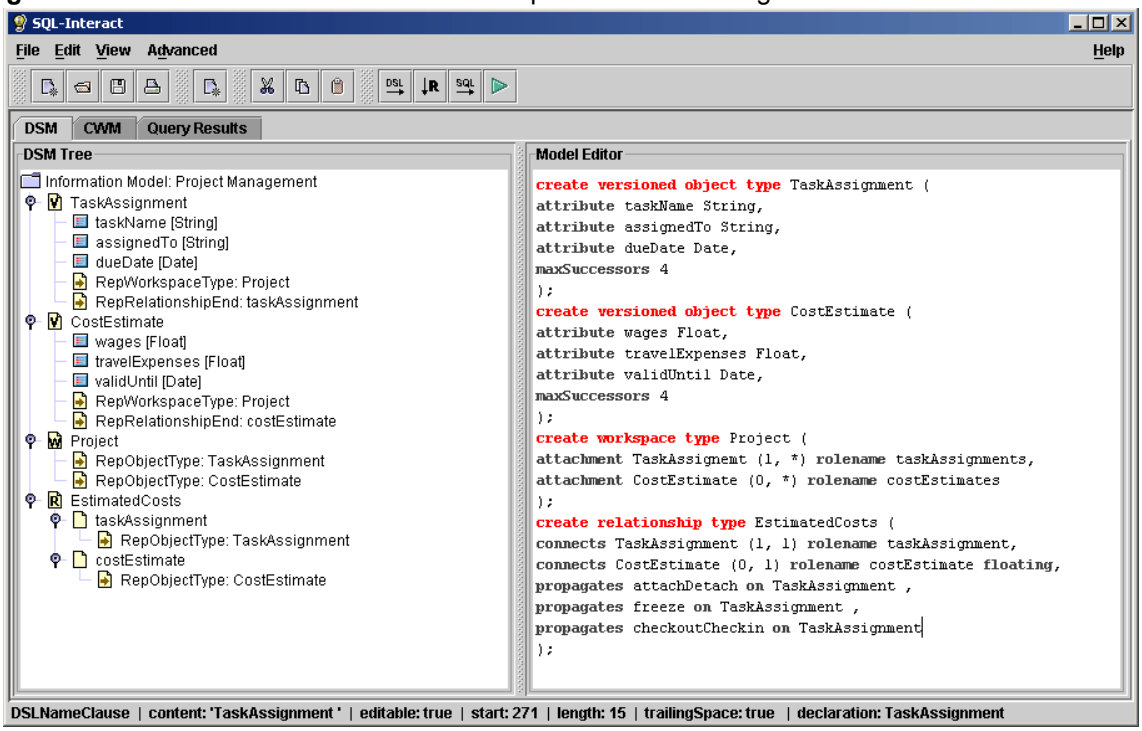
auch hier stehen eine Baumansicht auf die modellierten Konstrukte sowie eine textuelle Darstellung zur Verfügung, deren Inhalte das System im Rahmen des *DSL-Rendering* synchronisiert. Während auf der CWM-Ebene das CWM-Metamodell der OMG die Grundlage für die Modellierung in der Baumansicht darstellt, ist für die Modellierung der Bestellung in domänenspezifischen Konstrukten ein *eigenes Metamodell* vonnöten, das genau wie das DSL-Rendering spezifisch für eine bestimmte Produktlinie ist. Nach Abschluss der Spezifikation des Repository wird das domänenspezifische Modell durch eine *Reduktion* in ein CWM-Modell umgesetzt, das dann nach Wunsch auf der CWM-Ebene weiter bearbeitet und an ein Datenbanksystem gesandt werden kann. Der Begriff der Reduktion bezeichnet im Intentional Programming die Abbildung der modellierten Semantik auf eine niedrigere Abstraktionsstufe. In diesem Fall werden die domänenspezifischen Konstrukte der Konfigurationssprache auf CWM-Modellkonstrukte umgesetzt. Die Reduktion entspricht damit einem Generatorlauf im Sinne der generativen Programmierung. Auch die Regeln für die Durchführung der Reduktion sind spezifisch für eine bestimmte Produktlinie. Für die Unterstützung einer Produktlinie in SQLInteract werden folglich das Metamodell der Konfigurationssprache, eine Menge von Rendering-Regeln für die Abbildung zwischen Modell und textueller Darstellung sowie eine Menge von Reduktions-Regeln für die Umsetzung in CWM-Konstrukte benötigt. Gegebenenfalls können auch mehrere Mengen von Rendering-Regeln angegeben werden, um verschiedene textuelle Darstellungsformen zu ermöglichen.

5.3.2 Spezifikation einer Bestellung auf der DSM-Ebene

Die Benutzeroberfläche der DSM-Ebene ist im Hauptfenster von SQLInteract auf einer eigenen DSM-Tab untergebracht. Gestaltung und Bedienung sind praktisch identisch mit der Gestaltung

und Bedienung der CWM-Ebene. Abbildung 5.3 zeigt die DSM-Tab im Überblick und enthält die Spezifikation des bekannten Beispiel-Repository.

Abbildung 5.3 Oberfläche der DSM-Ebene mit beispielhafter Bestellung



DSM-Baum

Die linke Hälfte der Arbeitsfläche wird vom *DSM-Baum* eingenommen. Der Benutzer kann hier die Konfiguration des von ihm gewünschten Repository in Modellform angeben. Wie im CWM-Baum können Zweige des Baums aufgeklappt und zusammengelegt werden; verschiedene Elementtypen im Baum werden durch verschiedene den Einträgen vorangestellte Symbole visualisiert. Über Kontextmenüs können Elemente angelegt und gelöscht sowie Einstelldialoge für die Eigenschaften eines Elements aufgerufen werden (Abbildung 5.4). Auch im DSM-Baum kommen LinkNodes zum Einsatz, um eine übersichtliche Baumdarstellung eines nicht zyklenfreien Modells zu ermöglichen.

Für die baumorientierte Modellierung wird ein Metamodell benötigt, das die Konstrukte definiert, die im Modellbaum instanziiert werden können. Beim CWM-Baum übernimmt diese Aufgabe das CWM-Metamodell der OMG. Das Metamodell für die Beschreibung einer Bestellung muss vom Hersteller der Produktlinie bereitgestellt werden, da seine Konstrukte eng auf die Eigenarten der Produktlinie zugeschnitten sind. Abbildung 5.5 zeigt das in SQLInteract verwendete MOF-konforme Metamodell für die Modellierung von Repository-Bestellungen. Im DSM-Baum lassen sich auf oberster Ebene Instanzen von *RepObjectType*, *VersionedRepObjectType*, *RepWorkspaceType* und *RepRelationshipType* anlegen. An jede Instanz eines dieser Typen können dann Instanzen von *RepAttribute* angehängt werden. *RepRelationshipType* dient zur Modellierung binärer Beziehungen und wird stets zusammen mit zwei angehängten *RepRelationshipEnd*-Instanzen im Baum dargestellt. *Information Model* fasst als Wurzel der

Abbildung 5.4 Einstelldialog im DSM-Baum

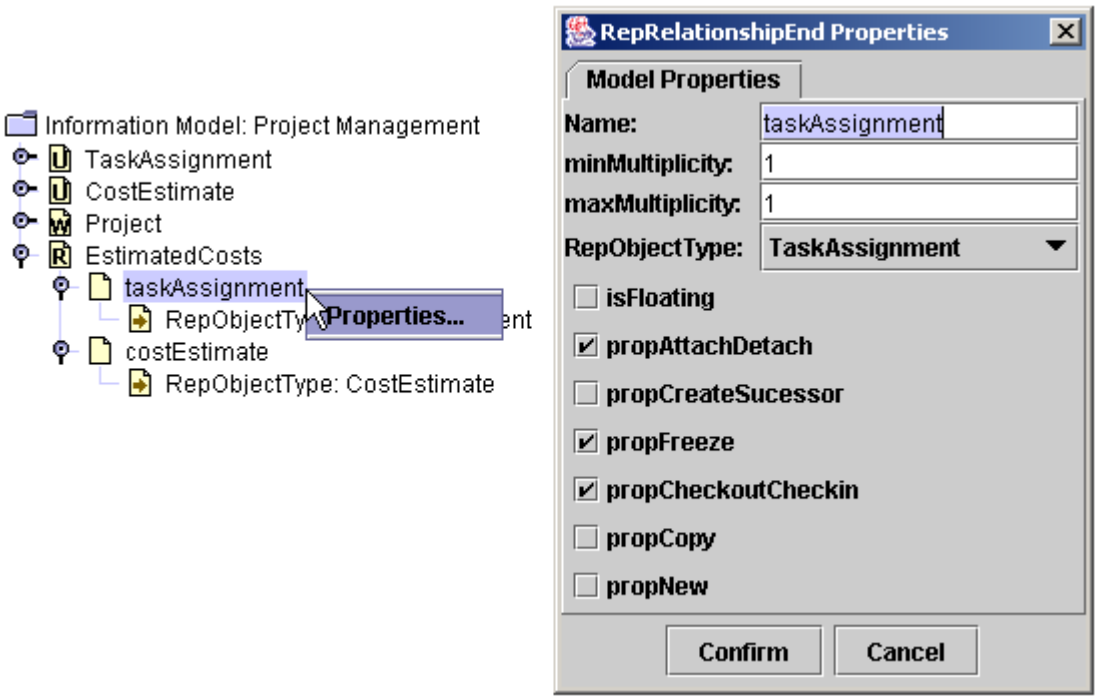
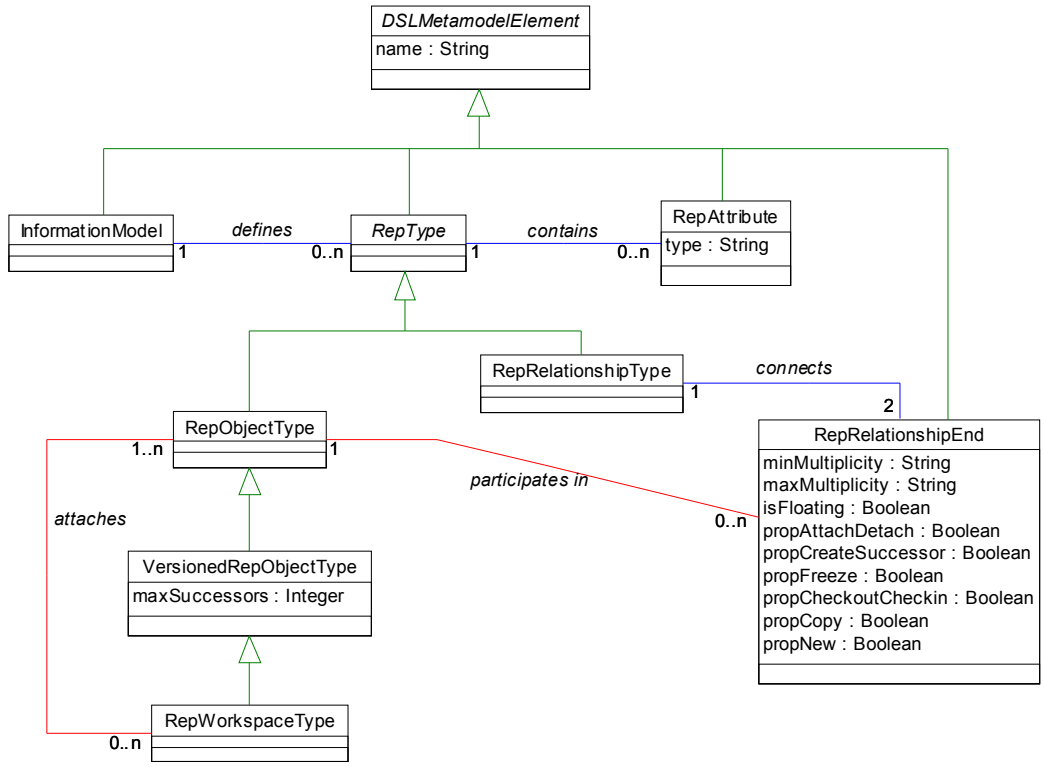


Abbildung 5.5 Metamodell für Repository-Bestellungen



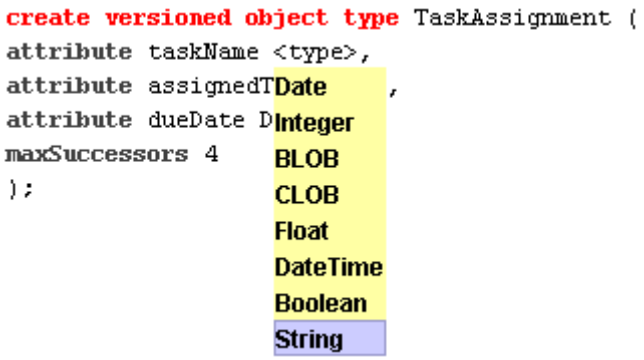
Baumhierarchie die übrigen Instanzen zusammen, ähnlich wie dies *Schema* im CWM-Baum übernimmt.

Modell-Editor

In der rechten Hälfte der DSM-Tab wird die vom Benutzer spezifizierte Bestellung in textueller Form dargestellt. Hierfür kommt die domänenspezifische Konfigurationssprache zum Einsatz, die in Abschnitt 5.2.4 vorgestellt wurde. Ähnlich wie CWM-Modell und DDL-Befehle auf der CWM-Ebene dieselbe Semantik auf zwei verschiedene Weisen visualisieren, bieten auch das *domänenspezifische Modell* (*DSM*) und das *domänenspezifische Konfigurationsprogramm* (*DSL-Programm*) zwei Sichten auf dieselben modellierten Abstraktionen.

Neben der Neueingabe von DSL-Befehlen im Modell-Editor ist auch die Weiterbearbeitung von Befehlen möglich, die durch Rendering des DSM in ein Konfigurationsprogramm entstanden sind. Der Modell-Editor bietet die gleichen Methoden der Eingabeunterstützung wie der Schema-Editor. Eingegebene Anfänge von Befehlen werden zu vollständigen Befehlsgerüsten mit Platzhaltern und nichteditierbaren Schlüsselwörtern ergänzt, sowie die Eingabe eindeutig identifiziert werden kann (automatische Vervollständigung) oder der Benutzer nach Druck auf die Tab-Taste eine Vervollständigung auswählt (Vervollständigungsvorschlag). Ebenso wird beim Ausfüllen der Platzhalter Vervollständigung angeboten (Abbildung 5.6), und die strukturelle Erweiterung der Befehlsgerüste ist möglich. Die Hervorhebung der Befehlssyntax lässt sich wie im Schema-Editor ohne Parser oder Definitionsdateien realisieren, da die interne Verwaltung der textuellen Darstellung dem gleichen Konzept einer Baumhierarchie zwischen den einzelnen Befehlsteilen folgt. Schließlich sind auch auf der DSM-Ebene textuelle Darstellung und Baumansicht untereinander synchronisiert, so dass Änderungen in einer Ansicht in der jeweils anderen Ansicht vom System nachvollzogen werden.

Abbildung 5.6 Vorschlagliste für die Eintragung eines Datentyps in einen Platzhalter



5.3.3 Reduktion des DSM auf CWM-Konstrukte

Nachdem der Benutzer die Spezifikation des Repository auf der DSM-Ebene abgeschlossen hat, kann er über die Symbolleiste die Reduktion des domänenspezifischen Modells auf ein CWM-Modell auslösen. Der Reduktionsvorgang entspricht in der Nomenklatur der generativen Programmierung der Generierung des fertigen Produkts aus einer Menge von Implementierungskomponenten anhand einer domänenspezifischen Spezifikation. In SQLInteract ist die Reduktion als Abbildung zwischen zwei Modellen realisiert (siehe dazu auch Abbildung 5.2). Das Ergebnis der Reduktion ist damit noch kein ausführbares Programm. Aus dem CWM-Modell lässt sich jedoch die DDL-Darstellung des Datenbank-Schemas mit benutzerdefinierten Routinen und

Datentypen rendern, die dann an ein Datenbanksystem übertragen werden kann und dort die spezifizierte Funktionalität realisiert.

Die Reduktion setzt die domänenspezifische Spezifikation des Benutzers in CWM-Konstrukte um. Im Zuge der Reduktion müssen also die Granularität der modellierten Semantik verfeinert und domänenspezifische Konstrukte in allgemeine CWM-Konstrukte umgewandelt werden. Anhand des DSM werden sowohl ein Datenbank-Schema als auch benutzerdefinierte Routinen für die Datenverwaltung erzeugt. Für jeden Objekttyp wird dabei zunächst eine Objekttabelle (*Repository Object Table, ROT*) erzeugt, in der Instanzen des Objekttyps mit ihren Attributen gespeichert werden können. Dazu wird aus den für den Objekttyp definierten Attributen ein benutzerdefinierter Datentyp generiert, wobei die Datentypen der Attribute auf geeignete SQL-Datentypen umgesetzt werden. Beispielsweise wird `String` zu `Varchar`, und `Boolean` wird auf `Integer` abgebildet. Mit dem benutzerdefinierten Datentyp wird eine Spalte in der Objekttabelle angelegt, die die eigentlichen Nutzdaten aufnimmt. Hinzu kommen Spalten `objId`, `versId`, `globId`, `predecessor`, `frozen` und `checkedout` für die Verwaltung.

Beziehungen zwischen Objekttypen werden soweit wie möglich normalisiert, das heißt Zwischentabellen (*Supplementary Tables, ST*) werden nur angelegt, wenn es sich um eine n:m-Beziehung handelt, die nicht über Fremdschlüssel in einer der Objekttabellen realisiert werden kann. An dieser Stelle zeigt sich der Vorteil individuell generierter Repositories gegenüber einer generischen Lösung. Das normalisierte Schema ermöglich direkte Joins zwischen den Objekt- und gegebenenfalls Zwischentabellen anstelle umständlicher indirekter Zugriffe über Metatabellen.

Neben den verschiedenen Tabellen für die Speicherung von Objekten und den Beziehungen zwischen ihnen werden auch benutzerdefinierte Routinen (*Stored Procedures*) erzeugt, die die verschiedenen Operationen im Rahmen der Datenverwaltung nach Abschnitt 5.2.3 übernehmen, so dass kein direkter Zugriff auf das Repository mit DML-Befehlen nötig ist, auch wenn die Möglichkeit hierzu auf der CWM-Ebene besteht. Als Parameter der benutzerdefinierten Routinen kommen bei der Übergabe von Nutzdaten die benutzerdefinierten Datentypen zum Einsatz, die im Zuge der Erstellung der Objekttabellen erzeugt wurden. Da die Propagierungssemantik der einzelnen Operationen über die verschiedenen Beziehungstypen in der Spezifikation des Repository festgelegt ist, kann die Realisierung der Propagierung in die benutzerdefinierten Routinen fest einkodiert werden, was wiederum Performanzvorteile gegenüber generischen Ansätzen einbringt, wo derartige Semantik in Form von ECA-Regeln mit zeitraubender Interpretation zur Laufzeit realisiert werden müsste.

Der Prozess der Reduktion lässt sich in Begriffen der MDA als Anwendung verschiedener Verfeinerungsmuster (*Refinement Patterns*) auffassen. Solche Verfeinerungsmuster beschreiben die Umsetzung bestimmter semantischer Konstellationen oder Spezifikationsteile auf eine niedrigere Abstraktionsstufe. Für die Umsetzung eines Objekttyps der DSM-Ebene in CWM-Konstrukte lautet ein solches Verfeinerungsmuster beispielsweise:

- Lege benutzerdefinierten Datentyp mit den Attributen des Objekttyps als Spalten an
- Lege eine Objekttabelle mit Spalten für Nutz- und Verwaltungsdaten an
- Lege benutzerdefinierte Routinen für die Verwaltung des Objekttyps an

Wird der Objekttyp darüber hinaus in Spezifikationen von Beziehungen oder Arbeitskontexten referenziert, werden entsprechende Erweiterungen der Objekttabelle oder der Verwaltungsrouti-

nen in deren Verfeinerungsmustern vorgenommen. Abbildung 5.7 zeigt die Reduktionsmethode der Metamodellklasse *VersionedRepObjectType*.

Abbildung 5.7 Reduktionsmethode der Metamodellklasse *VersionedRepObjectType*

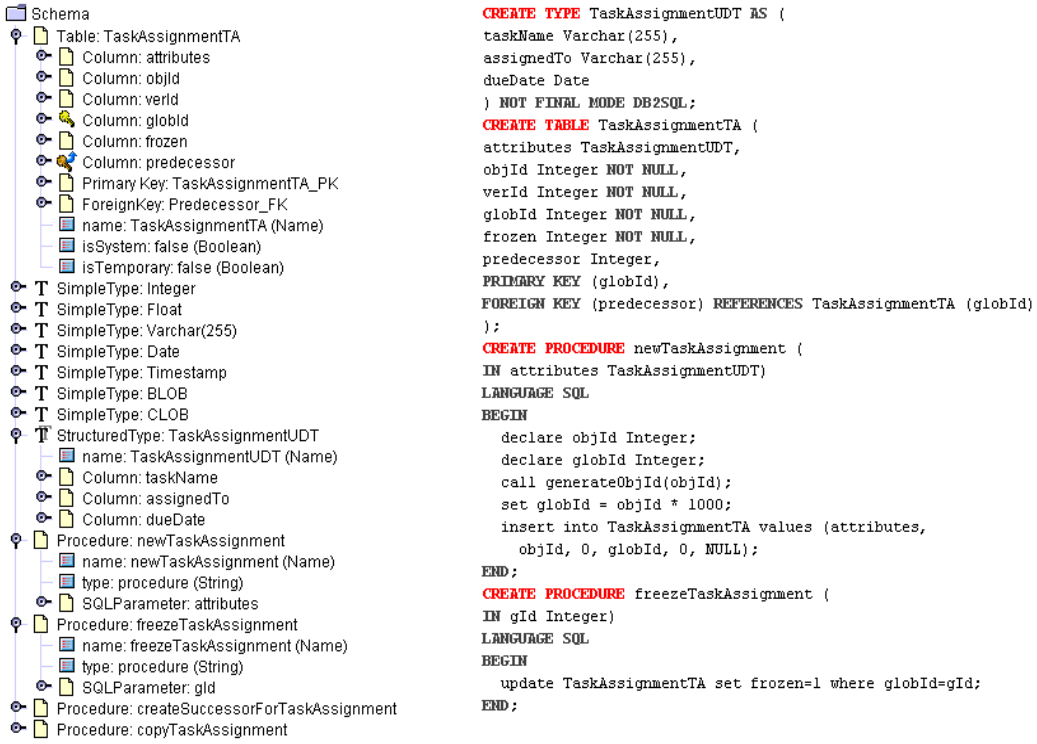
```
public void reduce(int context, SchemaDeclaration schema) {
 SQLStructuredTypeDeclaration attributes = ISQLModel.addSQLStructuredType(schema,
                                                    this.getName() + "Attributes");
 for (int i=0; i<this.getRepAttribute().size(); ++i) {
  ISQLModel.addColumn(attributes, (this.getRepAttribute().elementAt(i)).getName(), true,
                ISQLModel.DATATYPES.get(this.getRepAttribute().elementAt(i).getType()));
 }
 TableDeclaration table = ISQLModel.addTable(this.getName() + "ROT");
 ISQLModel.addColumn(table, "attributes", true, attributes);
 ISQLModel.addColumn(table, "objId", false, ISQLModel.DATATYPES.get("Integer"));
 ISQLModel.addColumn(table, "verId", false, ISQLModel.DATATYPES.get("Integer"));
 ColumnDeclaration gid = ISQLModel.addColumn(table, "globId", false,
                                                    ISQLModel.DATATYPES.get("Integer"));
 PrimaryKeyDeclaration pk = ISQLModel.setPrimaryKey(table, this.getName()+"ROT_PK", gid);
 gid.getUniqueKey().add(pk);
 // additional columns / foreign key for version management follow
 ISQLModel.addColumn(table, "frozen", false, ISQLModel.DATATYPES.get("Boolean"));
 ColumnDeclaration pred = ISQLModel.addColumn(table, "predecessor", true,
                                                    ISQLModel.DATATYPES.get("Integer"));
 ForeignKeyDeclaration fk = ISQLModel.addForeignKey(table, "Predecessor_FK", pk, pred);
 pk.getKeyRelationship().add(fk);
 pred.getKeyRelationship().add(fk);
 // end additional columns / foreign key
 // Procedure new()
 ProcedureDeclaration proc=ISQLModel.addProcedure("new"+this.getName(),"procedure", "");
 ISQLModel.addSQLParameter(proc, "attributes", "pdk_in", attributes);
 proc.getBody().setValue("  DECLARE objId INTEGER;\n  DECLARE globId INTEGER;\n  CALL
                          generateObjId(objId);\n  SET globId = objId * 1000;\n  INSERT
                          INTO " + this.getName() + "ROT VALUES (attributes, objId, 0,
                          globId, 0, NULL);");
// Procedure freeze()
...
// Procedure createSuccessor()
...
}
```

In Anlehnung an die Konzepte des Intentional Programming werden Verfeinerungsmuster in SQLInteract auch als Reduktions-Regeln (*Reduction Rules*) bezeichnet. Abbildung 5.8 zeigt beispielhaft einige der Konstrukte, die bei der Reduktion des Objekttyps *TaskAssignment*, wie er in Abbildung 5.3 definiert wurde, entstehen.

5.3.4 Rendering der textuellen Darstellungsformen

Durch das Rendering werden Modellkonstrukte im DSM-Baum und CWM-Baum auf die textuellen Darstellungsformen DSL respektive DDL abgebildet; aus Instanzen von Typen des jeweiligen Metamodells werden also Schlüsselwörter und Bezeichner. Der erste Schritt bei der Entwicklung eines algorithmischen Ansatzes für das Rendering besteht in der Analyse der Modellstruktur. Sowohl CWM als auch typische domänenspezifische Metamodelle produzieren bei Instanziierung ihrer Konstrukte Objektgraphen, die nicht zyklenfrei sind. Für das Rendering muss festgelegt werden, an welchen Stellen die Zyklen aufgetrennt werden sollen. Eine ähnliche Problemstellung ergibt sich bereits bei der Baumdarstellung der Objektgraphen, die in SQLInteract über LinkNodes gelöst wurde. Die Festlegung der Hierarchie in einem Modellbaum ist eine

Abbildung 5.8 Reduktionsergebnisse von *TaskAssignment* (Auszug)



Entwurfsentscheidung. Im CWM-Baum wurden Tabellen, benutzerdefinierte Datentypen und benutzerdefinierte Routinen als so genannte *Top-Level-Elemente* festgelegt, die in der ersten Hierarchie-Ebene unter der Baumwurzel *Schema* hängen. Auch beim DSM-Baum für Repositories ergab sich die hierarchische Gliederung aus der Betrachtung der Beziehungen zwischen den Konstrukten. Hier sind *RepObjectType*, *VersionedRepObjectType*, *RepRelationshipType* und *RepWorkspaceType* die Top-Level-Elemente.

Die für die Baumdarstellung festgelegte Hierarchie lässt sich auch beim Rendering nutzen. Da die textuelle Darstellung intern als Baum verwaltet wird, kann das Rendering als Baumtransformation realisiert werden. Zum Rendern eines kompletten Modells wird die Wurzel, also *Schema* oder *Information Model*, als Einstiegspunkt gewählt, woraufhin die Rendering-Methoden den ganzen Baum abarbeiten.

Die Rendering-Methoden sind in den Modellklassen untergebracht; ein Modellkonstrukt ist also in der Lage, sich selbst in verschiedenen Formen darzustellen. Für jede Zielsprache existiert dabei eine eigene Rendering-Methode, während die Steuerung variabler Details innerhalb des Renderings auf eine Zielsprache über Parameter der entsprechenden Rendering-Methode gesteuert wird. Typischerweise sind nämlich auch innerhalb einer Zielsprache in Abhängigkeit vom genauen Kontext verschiedene Ausgaben einer Rendering-Methode erforderlich. Die Spalte `globId` der Tabelle `TaskAssignmentTA` in Abbildung 5.8 erscheint im zugehörigen `CREATE TABLE` Statement im Schema-Editor beispielsweise zweimal: zum einen bei der Aufzählung der Tabellenspalten mit Angabe ihres Typs, zum anderen als Teil des Primärschlüssels ohne Angabe ihres Typs. Von entscheidender Bedeutung ist dabei die Tatsache, dass die Rendering-Methode des *Column*-Objekts nicht aus eigener Kraft beurteilen kann, welche Variante der Ausgabe gefordert ist. Während des Rendering-Laufs finden keinerlei Änderungen am CWM-Modell statt, so dass das *Column*-Objekt zu jeder Zeit sowohl mit einem *Table*-Objekt als auch

mit einem *PrimaryKey*-Objekt verbunden ist. Aus einer Strukturanalyse der benachbarten Knoten im Objektgraphen lässt sich daher keine Information über die richtige Ausgabe ableiten. Stattdessen muss die Information über den aktuellen *Rendering-Kontext* vom Aufrufer der Rendering-Methode übermittelt werden. In SQLInteract implementieren alle Modellklassen, die Rendering-Methoden beinhalten, aus diesem Grund das Interface `Rendering`, in dem einige Konstanten für die Übermittlung des aktuellen Rendering-Kontextes definiert sind (Abbildung 5.9).

Abbildung 5.9 Interface `Rendering` (Auszug)

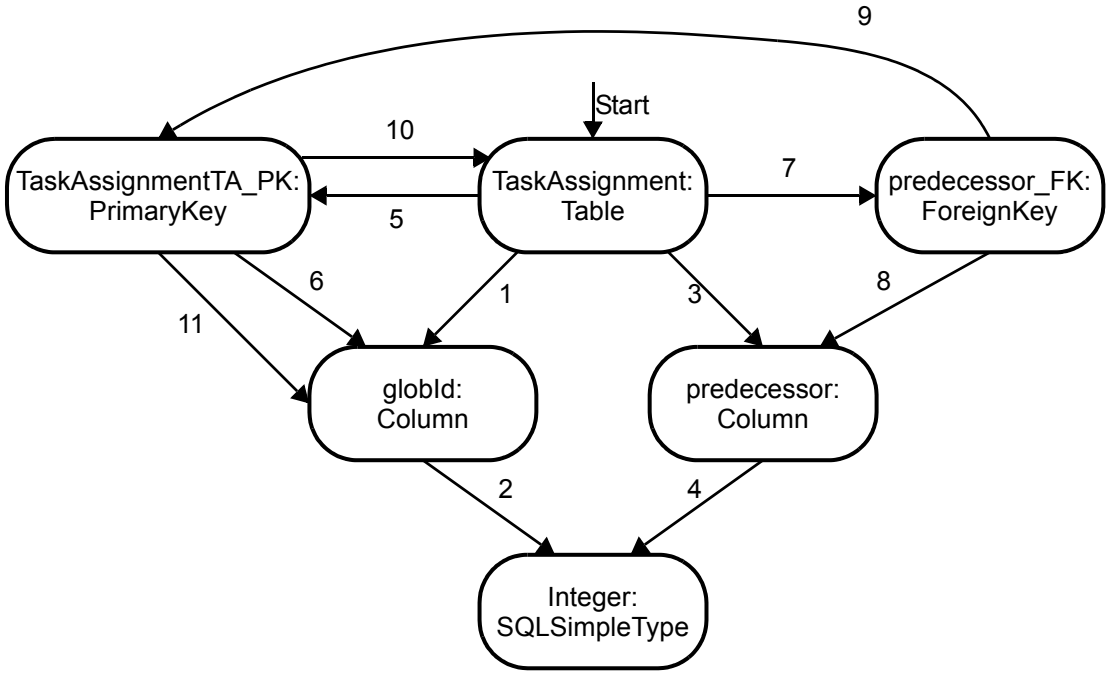
```
public interface Rendering {

  int CWM_NAME_ONLY = 0;
  int CWM_CREATE_STATEMENT = 1;
  int CWM_COLUMN_WITH_TYPE = 2;
  int CWM_COLUMN_WITH_TYPE_AND_NOTNULL = 3;
  int CWM_UNIQUECONSTRAINT_COLUMNS_ONLY = 4;
  int CWM_UNIQUECONSTRAINT_WITH_TABLE = 5;
  int CWM_FOREIGNKEY = 6;
  int CWM_FULL = 7;
  int CWM_PARAMETER_WITH_TYPE = 8;
}
```

Beim Rendering gilt das Prinzip, dass jedes Objekt für seine Darstellung nur auf eigene Daten direkt zugreift. Im Regelfall werden für die vollständige Darstellung zusätzlich Daten anderer Objekte benötigt; in diesem Fall ruft das aktuell rendernde Objekt die Rendering-Methoden der entsprechenden Objekte mit einer geeigneten Kontextangabe auf und lässt sich so die benötigten Daten liefern. Dabei können nur unmittelbar benachbarte Objekte angesprochen werden; ein navigierender Zugriff auf entfernte Objekte über eine oder mehrere Zwischenstationen ist nicht zulässig. Ein *Table*-Objekt hat also niemals Kontakt zu *DataType*-Objekten, sondern erhält die Darstellung der Datentypen zusammen mit der Darstellung der Spaltennamen von den direkt verbundenen *Column*-Objekten. Auf diese Weise bewegt sich der Kontrollfluss des Rendering einer Tiefensuche vergleichbar durch den virtuellen Objektbaum, bis am Ende das Einstiegsobjekt wieder die Kontrolle erhält und dem Aufrufer des Rendering das vollständig gerenderte *Schema* respektive *Information Model* zurückliefern kann. Abbildung 5.10 zeigt vereinfacht den Kontrollfluss beim Rendering des Table-Objekts aus Abbildung 5.8.

Die Schritte 1 bis 4 dienen der Darstellung der Spaltenaufzählung im `CREATE TABLE` Statement, Schritte 5 und 6 der Darstellung des Primärschlüssels, Schritte 7 und 8 der Darstellung des Fremdschlüssels und Schritte 9 bis 11 der Darstellung des Schlüsselkandidaten, den der Fremdschlüssel referenziert, wobei es sich dabei in diesem Beispiel um den Primärschlüssel von `TaskAssignment` selbst handelt. Aus der Abbildung ist auch zu erkennen, dass insbesondere *Column*-Objekte verhältnismäßig oft angesprochen werden und aus dem bloßen Graphenkontext nicht auf die erforderliche Darstellungsweise schließen können.

Ähnlich wie die Reduktion kann auch das Rendering über Regeln beschrieben werden. Für Table-Objekte lautet eine grobe Beschreibung des Renderings:

Abbildung 5.10 Kontrollfluss beim Rendering eines *Table*-Objekts (vereinfacht)



- Beginne `CREATE-TABLE`-Statement, ergänze Namen der zu rendernden Instanz
- Liste Spalten der Tabelle mit Datentypen auf
- Wenn Primärschlüssel definiert, füge `PRIMARY-KEY`-Klausel hinzu
- Wenn Fremdschlüssel definiert, füge für jeden Fremdschlüssel `FOREIGN-KEY`-Klausel hinzu
- Schließe `CREATE-TABLE`-Statement, liefere Daten an Aufrufer

Abbildung 5.11 zeigt beispielhaft die Rendering-Methode der Modellklasse *Column* mit der charakteristischen Fallunterscheidung anhand des vom Aufrufer übermittelten Rendering-Kontextes und Aufrufen der Rendering-Methode der Klasse *SQLDataType* im zweiten und dritten `if`-Block.

Abbildung 5.11 Rendering-Methode der Klasse *Column*

```
public Vector renderSQL(int context, Vector retVal) {
 if (retVal==null) retVal = new Vector();

 if (context==CWM_NAME_ONLY) {
  ColumnClause cc = new ColumnClause(false);
  cc.setContent(this.getElementName());
  cc.setDeclaration(this);
  cc.setSupportsCommaExtension(true);
  retVal.add(cc);

 } else if (context==CWM_COLUMN_WITH_TYPE) {
  ColumnClause cc = new ColumnClause();
  cc.setContent(this.getElementName());
  cc.setDeclaration(this);
  retVal.add(cc);
  this.getType()).getTarget()).renderSQL(CWM_NAME_ONLY, retVal);

 } else if (context==CWM_COLUMN_WITH_TYPE_AND_NOTNULL) {
  ColumnClause cc = new ColumnClause();
  cc.setContent(this.getElementName());
  cc.setDeclaration(this);
  retVal.add(cc);
  this.getType()).getTarget()).renderSQL(CWM_NAME_ONLY, retVal);
  if (this.getIsNullable().getValue() == COLUMN_NO_NULLS) {
   ((TypeClause)retVal.lastElement()).setTrailingSpace(true);
   retVal.add(new NotNullClause(false));
  }
 }

 return retVal;
}
```

Kapitel 6 SQLInteract als Metagenerator

Repositories sind lediglich ein Beispiel für eine Produktlinie im Bereich datenbankintensiver Anwendungen. Weitere Einsatzgebiete für derartige Anwendungen sind unter anderem Software für das Management von Geschäftsprozessen und Ressourcen, etwa nach den Konzepten des *Enterprise Resource Planning* (*ERP*) oder des *Customer Relationship Management* (*CRM*), und die dynamische Erstellung von HTML-Seiten bei datenintensiven Web-Anwendungen. Die Strategie der serverlastigen Funktionalitätsverteilung bietet sich überall dort an, wo große Mengen von Daten verarbeitet werden müssen und die zugreifenden Clients möglichst entlastet werden sollen.

SQLInteract soll die Flexibilität bieten, die Spezifikation von Mitgliedern verschiedener datenbankintensiver Produktlinien zu unterstützen. Die Systemarchitektur erlaubt daher den Austausch des Moduls, das für die Darstellung und Bearbeitung der modellierten Konstrukte auf der DSM-Ebene sowie für die Reduktion auf die CWM-Ebene zuständig ist. Dieses Kapitel beschreibt konzeptionell Technologien und Verfahren für die Einbindung einer Produktlinie in SQLInteract. Die vorgestellten Konzepte sind im Rahmen dieser Arbeit nicht vollständig implementiert worden.

6.1 DSM-Module

Ein *DSM-Modul* fasst in SQLInteract die Komponenten zusammen, die die Spezifizierung eines Mitglieds einer Produktlinie auf der DSM-Ebene und die anschließende Generierung des entsprechenden Produkts durch Reduktion der modellierten Spezifikation auf die CWM-Ebene ermöglichen. Für die Erstellung eines DSM-Moduls ist eine formale Beschreibung der Produktlinie notwendig, die das *Metamodell* der domänenspezifischen Konfigurationssprache sowie eine Menge von *Rendering*- und *Reduktionsregeln* umfasst.

Metamodell

Das *Metamodell* der Konfigurationssprache definiert eine domänenspezifische Sprache, in der die Spezifizierung eines Produkts innerhalb der Produktlinie erfolgen kann. Das Metamodell wird in einer *Domänenanalyse* aus den strukturellen Gemeinsamkeiten der Mitglieder der Produktlinie abgeleitet und als MOF-konformes Modell angegeben. Für diesen Zweck kommen typischerweise speziell entworfene MOF-basierte Metamodelle zum Einsatz, die eine genaue Beschreibung der domänenspezifischen Eigenheiten ermöglichen.

Das Metamodell enthält Klassen, Attribute und Assoziationen, die instanziiert werden können, um die Eigenschaften eines konkreten Mitglieds der Produktlinie zu beschreiben. Aus dem Metamodell werden *Modellklassen* generiert. Typischerweise implementiert eine Modellklasse jeweils die Attribute und Assoziationen einer Klasse des Metamodells und enthält darüber hinaus Funktionen, die die Bearbeitung dieser Merkmale über eine graphische Oberfläche erlauben. Im DSM-Baum können dann Instanzen der Metamodell-Klassen in Form von Modellobjekten als Baumknoten erzeugt und bearbeitet werden. Bei der Modellierung des Beispiel-Repository aus Kapitel 5 (Abbildung 5.1) mit dem zugehörigen Metamodell für Repositories (Abbildung 5.5) ist etwa *TaskAssignment* ein Modellobjekt und Instanz der Modellklasse *VersionedRepObjectType*.

Um die normalerweise nicht zyklenfreien Objektgraphen, die bei der Instanziierung und Verknüpfung von Modellklassen entstehen, als Baum darstellen zu können, müssen einige Assoziationen zwischen den Klassen mithilfe von LinkNodes dargestellt werden, wie es in Abschnitt 3.2.1 für den CWM-Baum erläutert wurde. Im Metamodell wird mithilfe von Stereotypes für jeden Assoziationstyp festgelegt, ob er im DSM-Baum über LinkNodes oder über die Knotenhierarchie dargestellt wird. Je nach Struktur des Metamodells kann es dabei vorkommen, dass nahezu sämtliche Vater-Sohn-Beziehungen im Baum Instanzen desselben Assoziationstyps darstellen, wie es etwa im CWM-Baum mit der *ownedElement*-Assoziation der Fall ist. Es ist aber auch möglich, dass ganz verschiedene Assoziationstypen über die Baumhierarchie dargestellt werden. Bei der Konfigurationssprache für Repositories ist dies beispielsweise bei der *defines*-, *contains*- und *connects*-Assoziation der Fall.

Renderingregeln

Renderingregeln beschreiben die Abbildung des Modells im DSM-Baum auf die textuelle Darstellung als *DSL-Programm* im Modell-Editor. Grundsätzlich können dabei mehrere Regelsätze angegeben werden, um verschiedene textuelle Darstellungsformen zu ermöglichen. Die Darstellungen können sich etwa in der Syntax der Spezifikations-Anweisungen unterscheiden, oder es können verschiedene Sichten auf das Modell angeboten werden, bei der Modellierung von Repositories beispielsweise eine objekt- und eine beziehungszentrische Ansicht. Die Renderingregeln sind von Metamodell zu Metamodell verschieden und können daher nicht allgemein spezifiziert werden. Sie werden mithilfe einer domänenspezifischen Sprache namens *RenL* (*Rendering rules Language*) angegeben, die stark vom Metamodell abhängig ist, da für die Spezifikation der Regeln Bezug auf die Konstrukte des Metamodells genommen werden muss. Bei der Produktlinie für Repositories wird beispielsweise festgelegt, dass Instanzen der Metamodell-Klassen *RepObjectType*, *RepVersionedObjectType*, *RepWorkspaceType* und *RepRelationshipType* auf entsprechende `create`-Anweisungen abgebildet werden und wie die genaue Syntax dieser Anweisungen lautet, während Instanzen von *RepAttribute* und *RepRelationshipEnd* nicht auf eigene `create`-Anweisungen abgebildet werden, sondern in die Anweisungen der anderen Typen integriert werden. Derartig detaillierte Festlegungen lassen sich nicht ohne Bezugnahme auf die konkreten Konstrukte des Metamodells vornehmen. Beispiel 6.1 zeigt die Renderingregel für *VersionedRepObjectType* aus Kapitel 5.

Aus den Renderingregeln werden *Java-Methoden* in der Art der Methode in Abbildung 5.11 generiert, die zur Laufzeit von SQLInteract die Abbildung zwischen DSM-Baum und DSL-Programm vornehmen. Theoretisch wäre es auch möglich, die Renderingregeln über eine generische Schnittstelle erst zur Laufzeit zu interpretieren; dieser Ansatz würde jedoch eine zusätzliche Indirektionsstufe zwischen den Regeln und ihrer Anwendung bedeuten, die sich letztlich in unterlegener Performanz auswirken würde. Generierte Java-Methoden sind aufgrund der in ihnen hart

Beispiel 6.1 Renderingregel für *VersionedRepObjectType*

```
render VersionedRepObjectType(vrot) {
  "create versioned object type" vrot.name "{\n"
  iterate vrot.contains(attribute) {
    "attribute" attribute.name attribute.type ",\n"
  }
  "maxSuccessors" vrot.maxSuccessors "\n}"
}
```

kodierten Regeln zwar auf eine bestimmte Produktlinie festgelegt; diese Beschränkung stellt jedoch keinen Nachteil dar, da für verschiedene Produktlinien ohne großen Aufwand jeweils eigene Renderingmethoden generiert werden können.

Reduktionsregeln

Reduktionsregeln beschreiben die Umsetzung modellierter Produktspezifikationen in konkrete Produkte durch Reduktion der domänenspezifischen Modelle auf CWM-Modelle. Die Anforderungen an Reduktionsregeln und ihre Angabe sind weitgehend mit den Anforderungen an Renderingregeln vergleichbar, auch wenn die genaue Aufgabe der Reduktionsregeln sich von der der Renderingregeln unterscheidet. Wie die Renderingregeln sind auch die Reduktionsregeln spezifisch für ein Metamodell und lassen sich nicht allgemein angeben. Sie werden in einer domänenspezifischen Sprache namens *RedL* (*Reduction rules Language*) spezifiziert, die ähnlich wie RenL engen Bezug auf die Konstrukte des Metamodells und darüber hinaus auch auf die Konstrukte des CWM nehmen muss, um die notwendige Ausdrucksmächtigkeit für die Angabe der Regeln zu erreichen. Im Unterschied zu den Renderingregeln ist das Ziel der mit den Reduktionsregeln definierten Abbildung jedoch nicht eine andere Darstellungsform auf derselben Abstraktionsebene wie das Ausgangsmodell, sondern ein CWM-Modell. Beispiel 6.2 zeigt die Reduktionsregel für *VersionedRepObjectType* aus Kapitel 5.

Beispiel 6.2 Reduktionsregel für *VersionedRepObjectType*

```
reduce VersionedRepObjectType(vrot) {
  createUDT("attributes", vrot.contains)
  createVersionedTable
  createVersionedProcedures
}
```

Aufgrund der prinzipiellen Ähnlichkeit von Rendering- und Reduktionsregeln zeigt sich auch für die Reduktionsregeln, dass ein generischer Ansatz mit Regelinterpretation zur Laufzeit nicht die ideale Lösung darstellt. Auch aus den Reduktionsregeln werden daher *Java-Methoden* wie in Abbildung 5.7 generiert, die zur Laufzeit die Modelltransformation vornehmen.

DSM-Module in SQLInteract

Die Modellklassen und die Renderingmethoden bilden zusammen das inhaltliche Rückgrat der DSM-Ebene. Die Funktionalität für die Bearbeitung eines Modells im DSM-Baum und im Modell-Editor ist fest in SQLInteract integriert, ebenso die Bedienung und die Gestaltung der

Benutzeroberfläche. Die zu bearbeitenden Inhalte ergeben sich jedoch aus den Vorgaben des Metamodells über die generierten Modellklassen, und ihre genaue Darstellung als DSL-Programm erfolgt gemäß der Renderingregeln durch die generierten Renderingmethoden. Durch die Trennung zwischen den vom jeweiligen DSM-Modul definierten Inhalten und ihrer Präsentation und Manipulation lassen sich so mit unterschiedlichen DSM-Modulen Mitglieder ganz verschiedener Produktlinien in SQLInteract spezifizieren.

Die CWM-Ebene ist mit ihren Modellklassen und Renderingmethoden fest in SQLInteract integriert. Diese Integration ist der Grund für die Einschränkung der unterstützten Produktlinien auf den Bereich der datenbankintensiven Applikationen. Der Übergang von den austauschbaren Modulen der DSM-Ebene auf die konstante CWM-Ebene ist Aufgabe der Reduktionsmethoden. Sie bilden das domänenspezifische Modell einer Anwendung auf ein allgemeines CWM-Modell ab und müssen dabei die grobgranularen, abstrakten Konstrukte der DSM-Ebene auf feingranulare CWM-Konstrukte mit deutlich geringerer Ausdrucksmächtigkeit umsetzen. Eine solche Abbildungsleistung ist nur mit genauer Kenntnis der Ausgangsdomäne und der Zielsprache zu realisieren. Die Reduktionsregeln sind Teil der Beschreibung einer Produktlinie und werden in das DSM-Modul integriert. Abbildung 6.1 zeigt, wie ein DSM-Modul durch Modellklassen und Renderingmethoden die Gerüste der Arbeitsbereiche auf der DSM-Ebene mit Inhalten füllt und über die Reduktionsmethoden die Verbindung zur CWM-Ebene herstellt.

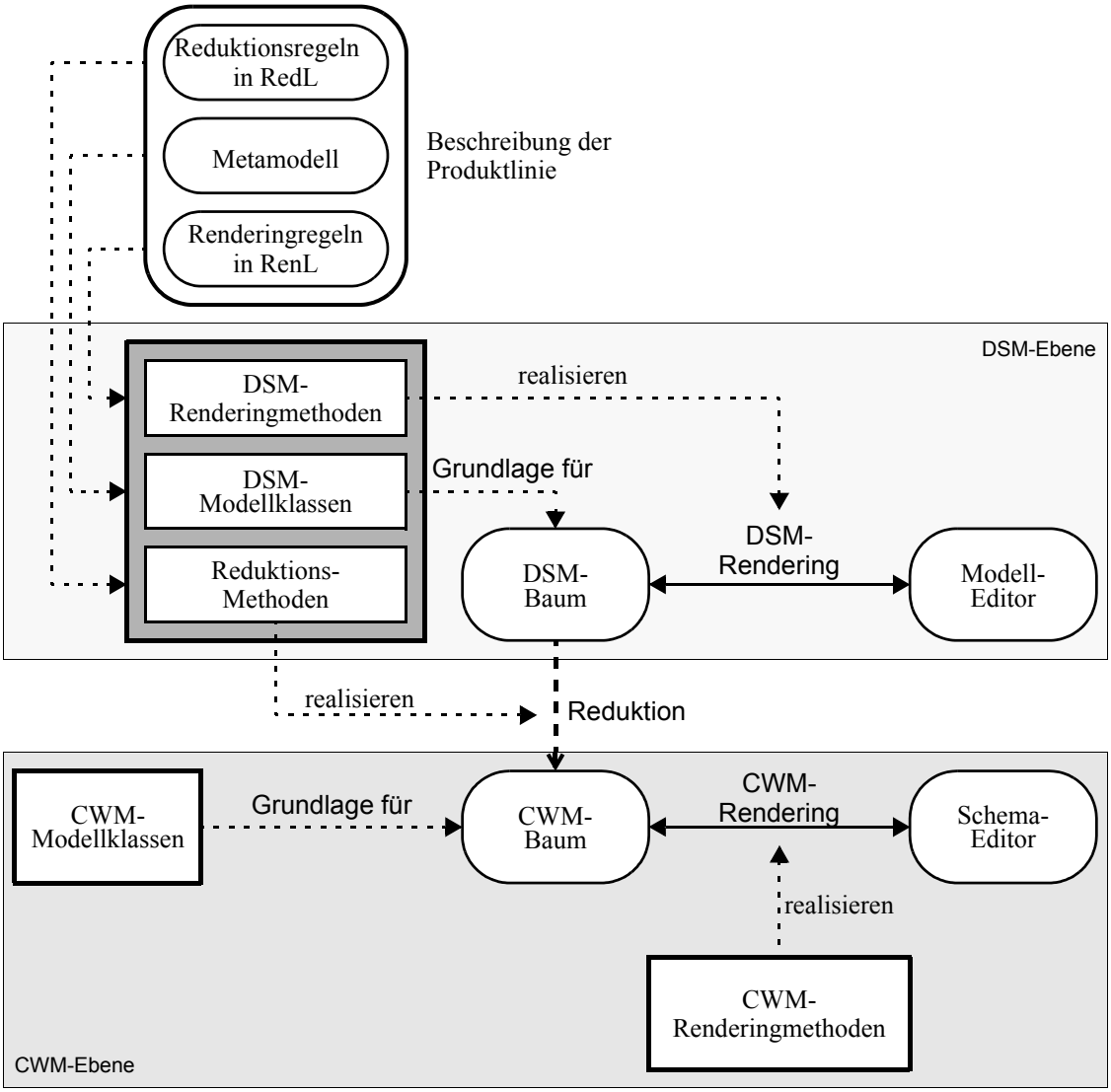
Einbindung von DSM-Modulen

Wie in den vorstehenden Abschnitten beschrieben, werden DSM-Module für die Unterstützung von Produktlinien aus den domänenspezifischen Beschreibungen von Metamodell, Renderingregeln und Reduktionsregeln generiert. Der Generator ist nicht Teil von SQLInteract, sondern als externes Programm ausgelegt. Endprodukt der Generierung sind Java-Klassen, die prinzipiell vom Generator auch bereits kompiliert werden könnten. Dennoch können DSM-Module nicht zur Laufzeit in SQLInteract eingebunden werden, da sie recht eng mit dem konstanten Teil des Systems verzahnt sind. Eine dynamische Einbindung von DSM-Modulen über *Introspection* würde eine Umgestaltung der Schnittstellen erfordern, um die Module stärker zu entkoppeln und die Erkennung der Eigenschaften eines neuen Moduls zur Laufzeit zu ermöglichen. Neben dem erheblichen Aufwand einer solchen dynamischen Einbindung brächte eine solche Architektur Indirektionen und Generizität mit sich, die sich auf die Performanz des Gesamtsystems negativ auswirken würden.

Anstelle einer dynamisch-generischen Einbindung der DSM-Module wird daher auch bei der Integration von DSM-Modulen in SQLInteract nach den Konzepten der generativen Programmierung verfahren. Die formale Beschreibung einer Produktlinie kann als Bestellung für ein entsprechendes DSM-Modul aufgefasst werden. Anhand der Beschreibung wird also ein DSM-Modul mit Modellklassen, Rendering- und Reduktionsmethoden generiert und durch den Generator mit den konstanten Modulen von SQLInteract integriert. Das Gesamtsystem entsteht so aus Komponenten, die unabhängig von der im konkreten Fall unterstützten Domäne sind und die immer vorhandene Funktionalität beinhalten, und aus Komponenten, die anhand der Beschreibung der Produktlinie für den speziellen Fall konfiguriert werden und das DSM-Modul realisieren.

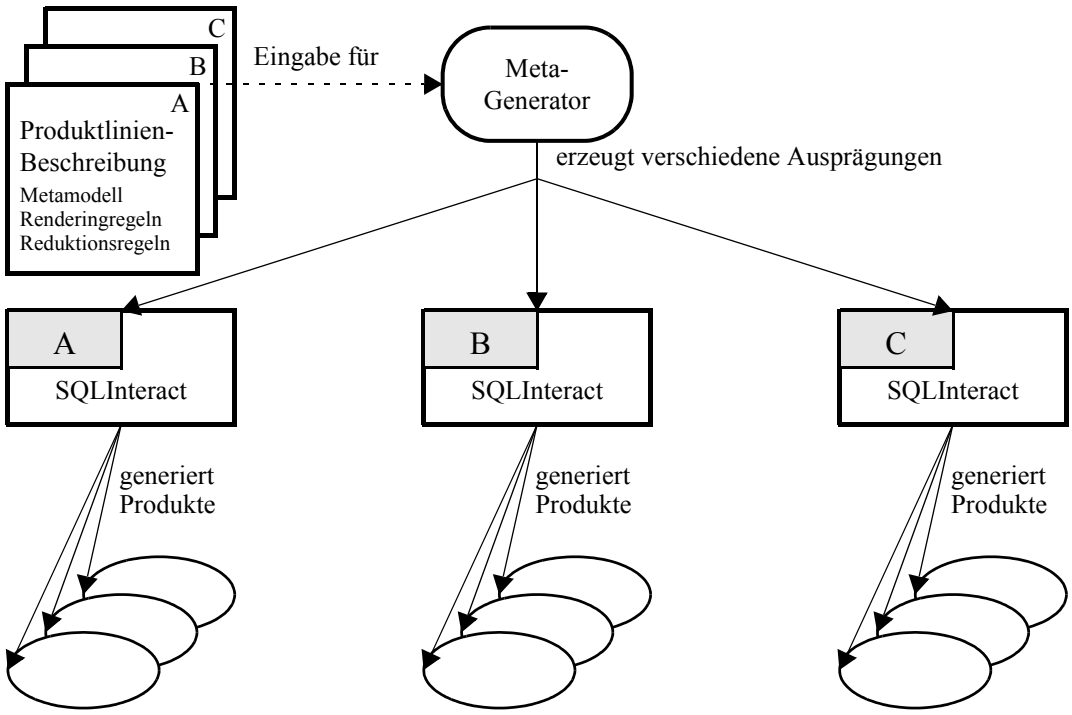
Auf diese Weise bildet SQLInteract selbst eine Produktlinie, wobei jedes Mitglied dieser Produktlinie für sich betrachtet ein Spezifikationswerkzeug und Generator für eine Produktlinie auf dem Gebiet der datenbankintensiven Anwendungen ist. Die verschiedenen Ausprägungen von

Abbildung 6.1 DSM-Modul mit Modellklassen, Renderingmethoden und Reduktionsmethoden



SQLInteract unterscheiden sich in der unterstützten Produktlinie ihres jeweiligen DSM-Moduls. Jede Ausprägung erlaubt die Modellierung von Bestellungen für Mitglieder ihrer unterstützten Produktlinie und die anschließende Generierung des entsprechenden Produkts. SQLInteract kann daher als *Meta-Produktlinie* bezeichnet werden. Der Generator, der aus den formalen Beschreibungen der verschiedenen Produktlinien die DSM-Module und damit letztlich die verschiedenen Ausprägungen von SQLInteract erzeugt, generiert unter diesem Gesichtspunkt folglich keine Produkte, sondern Umgebungen für die Unterstützung ganzer Produktlinien. Es handelt sich damit um einen Generator für Generatoren, also einen *Meta-Generator*. Abbildung 6.2 fasst die Zusammenhänge zwischen den unterschiedlichen Ebenen von Produkten, Produktlinien und Generatoren zusammen. Die folgenden Abschnitte beschreiben die technischen Aspekte der Generierung.

Abbildung 6.2 SQLInteract als Meta-Produktlinie



6.2 Velocity

Mithilfe von *Template-Engines* können textuelle Artefakte wie HTML-Seiten, Postscript-Dokumente oder auch Quellcode-Dateien von Software-Systemen automatisiert erzeugt werden. Dazu definiert der Benutzer ein oder mehrere so genannte *Templates*. Ein Template ist eine Schablone für die zu erzeugende Ausgabe oder einen Teil davon. Typischerweise enthält ein Template ein Gerüst statischer Textteile, die unverändert in die Ausgabe übernommen werden. In dieses Gerüst können dynamische Elemente wie Platzhalter, Schleifenkonstrukte oder bedingte Blöcke eingebettet sein. Während des Erzeugungsvorgangs werden die dynamischen Elemente von der *Engine* ausgewertet und die Resultate der Auswertungen anstelle der Elemente selbst in die Ausgabe geschrieben. Templates sind also parametrisierte Vorlagen für die zu erzeugenden Artefakte. Die genaue Ausprägung eines Artefakts lässt sich auf diese Weise zur Erzeugungszeit festlegen, ohne dass an den zugrunde liegenden Templates Veränderungen vorgenommen werden müssen. Damit bietet sich Template-basierte Generierung von Artefakten überall dort an, wo die verschiedenen benötigten Artefakte sich zwar in Details ihrer Gestaltung unterscheiden, grundsätzlich aber eine gemeinsame Struktur aufweisen.

Velocity [Apa03a] ist eine Template-Engine auf Basis von Java und einer speziellen Sprache für die Definition von Templates, der *Velocity Template Language* (*VTL*). Sie ist primär für die dynamische Erstellung von HTML-Seiten bestimmt und stellt auf diesem Gebiet eine Alternative zu JavaServer Pages [Sun01b] oder PHP [PHP03] dar. Der Einsatz vom Velocity ist jedoch nicht auf HTML als Endprodukt beschränkt; in den Templates lassen sich letztlich beliebige textuelle Sprachen einsetzen, da die Velocity-Engine ausschließlich VTL-Konstrukte auswertet und die Semantik der umgebenden statischen Elemente für den Erzeugungsprozess irrelevant ist.

Bei der Erzeugung der Ausgabe verbindet die VTL-Engine die Inhalte der Templates mit Daten aus einem so genannten *Kontext*. Dieser Prozess wird als *Mischen* (*Merging*) bezeichnet. Der Kontext dient dabei als Container für jene Daten, die dynamisch in das Gerüst der Templates eingebunden werden sollen und die individuelle Ausprägung der erzeugten Artefakte bestimmen. Technisch ist der Kontext einer Hashtabelle vergleichbar, in der Java-Objekte unter einem eindeutigen Namen abgelegt werden. In den Templates können diese Objekte dann über ihren Namen referenziert werden, wodurch insbesondere ihre Attribute und Methoden für die weitere Verwendung innerhalb der Templates nutzbar gemacht werden.

Grundsätzlich besteht weitgehende Freiheit hinsichtlich der Objekte, die in den Kontext eingefügt werden können. In den Templates selbst ist allerdings ausschließlich die Verarbeitung von Zeichenketten (*Strings*) möglich. Beim Zugriff auf Daten aus dem Kontext werden diese daher von der VTL-Engine automatisch durch Aufruf der Methode `toString()` in Strings umgewandelt. Von besonderem Interesse sind Sammlungs-Datentypen (*Collection Data Types*) wie *Array* oder *Collection*, über deren Elemente iteriert werden kann. Velocity ermöglicht eine solche Iterierung über spezielle Direktiven in den Templates, ohne dass das Objekt im Kontext explizit dafür vorbereitet werden muss. Die Engine ist in der Lage, anhand des Datentyps selbstständig die nötigen Vorbereitungen zu treffen und die Elemente an das aufrufende Template zu übertragen.

In den Templates werden dynamische Inhalte über drei Typen von *Referenzen* eingebunden: *Variablen*, *Attribute* (*Properties*) und *Methoden*. Der Datenfluss ist nicht auf die Richtung vom Kontext in ein Template beschränkt; Referenzen können auch schreibend auf die Objekte im Kontext zugreifen. In jedem Fall müssen Ein- und Ausgaben von Referenzen vom Datentyp *String* sein.

Über *Attribute* kann aus einem Template auf Attribute eines Objekts im Kontext zugegriffen werden. Die entsprechende Referenz hat die Form `${<Objekt>.<Attribut>}`. Die VTL-Engine setzt eine solche Referenz gegebenenfalls in einen Methodenaufruf `<Objekt>.get<Attribut>()` um. Mit *Methoden* können aus einem Template beliebige Methoden eines Objekts im Kontext aufgerufen werden. Diese Referenz hat die Form `${<Objekt>.<Methode>(<Parameterliste>)}`. Methoden können schreibend auf Objekte zugreifen, indem sie über ihre Parameter Werte an den Kontext übertragen. *Variablen* existieren ausschließlich innerhalb eines Templates und haben keinen Bezug zum Kontext, können jedoch als Parameter von Methoden genutzt werden. Ansonsten dienen sie zur Speicherung von variablen Werten innerhalb des Template, beispielsweise für die Verwendung in Schleifen oder bedingten Codeblöcken. Die Referenz hat die Form `${<Variable>}`.

Für die Steuerung des Erzeugungsprozesses kommen *Direktiven* in den Templates zum Einsatz.

- Mit der `#set`-Direktive können Werte von Attribut- und Variablen-Referenzen gesetzt werden. Die zugewiesenen Werte können aus anderen Referenzen oder aus Literalen stammen.
- Die `#if`-, `#elseif`- und `#else`-Direktiven erlauben die bedingte Übernahme von Textblöcken in die Ausgabe, wobei diese Textblöcke ihrerseits weitere dynamische Elemente beinhalten können. Als Entscheidungskriterium können numerische und logische Bedingungen angegeben werden.
- Die `#foreach`-Direktive wird für die bereits erwähnte Iterierung über Sammlungs-Datentypen wie *Vector*, *Hashtable* oder *Array* genutzt. Nach dem einleitenden `#foreach ($<Element> in $<Collection>)` kann im Schleifenrumpf das jeweils aktuelle Ele-

ment über `$<Element>` referenziert werden. Zusätzlich wird über `$velocityCount` automatisch ein numerischer Schleifenzähler zur Verfügung gestellt.

- Mit den `#include`- und `#parse`-Direktiven kann der Inhalt lokaler Dateien an der betreffenden Stelle in die Ausgabe eingefügt werden. Mit `#parse` eingefügte Dateien werden dabei im Gegensatz zu mit `#include` eingefügten Dateien als Template behandelt, und ihr Inhalt wird interpretiert. Mit `#parse` kann sich ein Template auch rekursiv selbst aufrufen.
- Die `#macro`-Direktive ermöglicht die Definition von *Velocimacros*, benannten Codeblöcken im Template, die von anderen Stellen im Template aufgerufen werden können, so dass ihre Ausgabe an diesen Stellen eingefügt wird. Ähnlich wie Methoden können auch Makros beim Aufruf Parameter übergeben werden, auf die im Rumpf der Makros zugegriffen werden kann wie auf Variablen.

6.3 Generierung von Modellklassen

Velocity überlässt den Entwicklern von Java-Anwendungen und Templates die Entscheidung, welche Objekte über den Kontext zwischen Anwendung und Template ausgetauscht werden sollen. Aus Sicht des Templates stellt der Kontext eine Schnittstelle zur Anwendung (*Application Programming Interface, API*) dar. Da die Mächtigkeit von Java als Programmiersprache wesentlich größer ist als die von VTL, wird empfohlen die Objekte im Kontext so zu gestalten, dass sich ihre Daten im Template möglichst einfach weiterverarbeiten lassen [Apa03b]. Die Last der Anpassung zwischen den Anforderungen von Java-Anwendung und Template soll also von der Java-Anwendung getragen werden.

Prepared Elements

Ein geeigneter Ansatz für die systematische Realisierung dieser Empfehlung sind spezielle Klassen für den Datenaustausch im Kontext, so genannte *Prepared Elements*. Sie stellen über Methoden die von ihnen gekapselten Daten genau in der Form bereit, die im Template benötigt wird. Dies umfasst sowohl die Darstellung als Strings als auch die Zusammenstellung und Aufbereitung der Daten aus möglicherweise verstreuten Quellen mit ganz verschiedenen Datentypen und Granularitäten. Auf diese Weise beschränkt sich die Beschaffung dynamischer Daten im Template auf einfache Methoden- oder Attribut-Referenzen, so dass sich die Menge der VTL-Direktiven möglichst gering halten lässt und das Template insgesamt übersichtlicher wird. Ein vergleichbarer Ansatz findet sich bei JavaServer Pages. Obwohl dort theoretisch beliebiger Java-Code in die Quelldatei einer Seite integriert werden kann, ist es gängige Praxis und guter Stil, Funktionalität in JavaBeans auszulagern und diese in der Seite über spezielle Tags zu referenzieren, um Übersichtlichkeit und Wartbarkeit zu verbessern.

Die Prepared Elements können als reine Container-Klassen implementiert werden, die von anderen Klassen der Java-Anwendung instanziiert und mit Daten gefüllt werden. Sinnvoller ist jedoch eine Modellierung nach dem *Adapter*-Pattern. Bei diesem Verfahren werden die Daten erst im Moment des Aufrufs der Methoden im Template aus den eigentlichen Quellen abgerufen und für die Verwendung im Template aufbereitet. Die Prepared Elements bilden also keinen eigenständigen Speicherort für Daten, sondern fungieren lediglich als Zwischenschicht mit Adapterfunktionalität beim Datenzugriff durch Referenzen im Template.

Ablauf der Generierung

Der SQLInteract-Generator erzeugt die Modellklassen eines DSM-Moduls mithilfe von Velocity. Da die Produkte der Generierung Java-Klassen sind, enthalten die Templates Fragmente von Java-Code, aus denen durch Auswertung der VTL-Direktiven und in Verbindung mit Daten aus dem Kontext vollständige Klassen entstehen. Das Metamodell einer Produktlinie wird in Form einer XMI-Datei erwartet. Für den Zugriff auf diese Datei wird das *Novosoft Metadata Framework* (*NSMDF*) verwendet [Nov02]. NSMDF ist eine Implementierung des *Java Metadata Interface* (*JMI*) [Sun02]. Es erlaubt das Einlesen von UML- oder anderen MOF-basierten Modellen aus XMI-Dateien und erstellt aus den Modelldaten ein so genanntes *In-Memory-Repository*. Das Repository erlaubt dann den Zugriff auf die Elemente des Modells über ein spezielles API.

Der direkte Zugriff aus Java auf Elemente des Metamodells der Produktlinie lässt sich über NSMDF recht einfach realisieren. Für die Verwendung in den Templates ist das NSMDF-API jedoch immer noch zu kompliziert, da Modelldaten unter anderem durch navigierenden Zugriff auf verschiedene Teile des Modells zusammengestellt werden müssen oder im Modell nicht im benötigten Format vorliegen. Aus diesem Grund kommen Prepared Elements zum Einsatz, die die Navigation im Modell übernehmen und die Modelldaten so weit aufbereiten, dass sie ohne weitere Bearbeitung im Template verwendet werden können. Die Prepared Elements sind gezielt für die Verwendung im SQLInteract-Generator entworfen worden und bieten über spezielle Methoden Informationen aus dem Modell an, die für die Erstellung der Modellklassen benötigt werden. So lässt sich zu einer Klasse im Metamodell beispielsweise direkt abfragen, ob eine Superklasse existiert und wie gegebenenfalls der Name dieser Superklasse lautet, wobei der für die Bereitstellung dieser Information notwendige navigierende Zugriff im Modell, die Behandlung von Sonderfällen und die Umwandlung aller möglichen Anfrageergebnisse in Strings von der entsprechenden Prepared-Element-Klasse gekapselt wird.

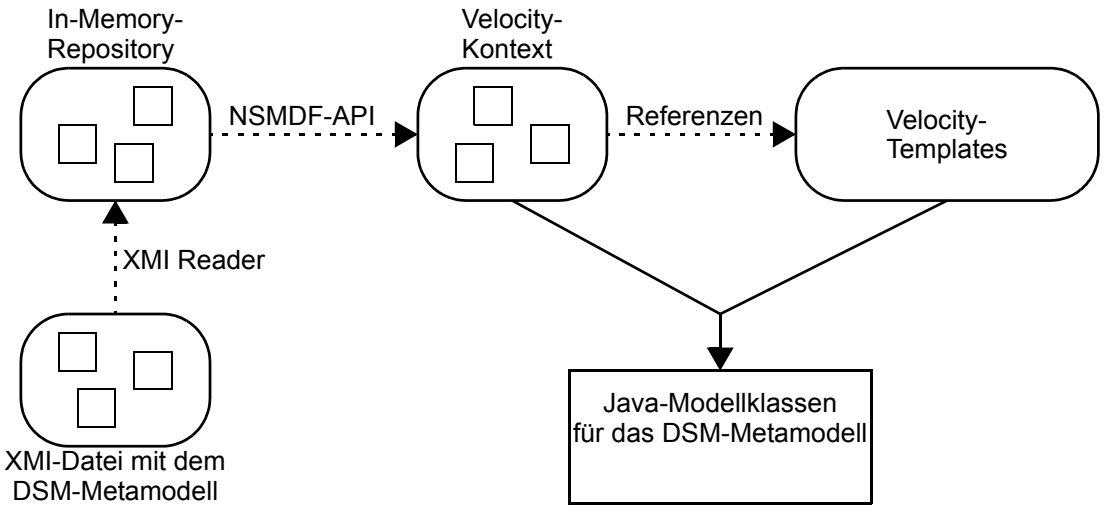
Der Generator legt die benötigten Prepared-Element-Klassen an und fügt sie in den Velocity-Kontext ein, bevor er für die eigentliche Generierung die Velocity-Engine startet. Im Generierungslauf verbindet die Engine dann die Code-Fragmente aus den Templates mit den Modelldaten, die aus der XMI-Darstellung des Metamodells stammen und über Kontext und NSMDF-API in eine nutzbare Form gebracht wurden (Abbildung 6.3).

6.4 Generierung von Rendering- und Reduktionsmethoden

Die Rendering- und Reduktionsmethoden entstehen aus den Rendering- und Reduktionsregeln, die bei der Beschreibung einer Produktlinie angegeben werden müssen. Da sie nach Abschluss des Generierungsprozesses als Java-Methoden vorliegen, können sie in die Modellklassen integriert werden. Jede Modellklasse verfügt dann über eine Methode für die Reduktion der von ihr modellierten Inhalten auf die CWM-Ebene sowie über eine oder mehrere Methoden für die Abbildung ihrer Inhalte in eine textuelle Darstellungsform.

Das Ausgangsmodell auf der DSM-Ebene ist zunächst ein zyklischer Graph. Für eine geordnete Abbildung der in ihm enthaltenen Elemente muss daher in jedem Fall eine Traversierungsstrategie für Rendering und Reduktion erstellt werden. Als Richtlinie bieten sich dabei die Assoziationen an, die im Metamodell für eine Darstellung über die Baumhierarchie im DSM-Baum ausgezeichnet sind. In der Produktlinie für Repositories aus Kapitel 5 ergeben sich so beispiels-

Abbildung 6.3 Datenfluss bei der Generierung von Modelklassen



weise *RepObjectType*, *VersionedRepObjectType*, *RepWorkspaceType* und *RepRelationshipType* als so genannte *Top-Level-Elemente*, die in der ersten Baumebenen unter der Wurzel stehen, während *RepAttribute* und *RepRelationshipEnd* untergeordnete Klassen sind. Die genaue Reihenfolge der Traversierung der verschiedenen Knotentypen im DSM-Objektgraph muss in den Rendering- und Reduktionsregeln angegeben werden, da die Ableitung der Reihenfolge aus dem Metamodell nicht detailliert genug ist und vor allem unterschiedliche Anforderungen von Rendering und Reduktion nicht erfüllen kann.

Reduktion

Bei der Reduktion eines DSM-Modells auf ein CWM-Modell werden anhand der Knoten im DSM-Objektgraphen Objekte auf der CWM-Ebene angelegt und in einen Objektgraphen eingefügt, der als CWM-Baum dargestellt wird. Die Reduktionsregeln müssen daher festlegen, wie der DSM-Objektgraph durchlaufen werden soll und welche Konstrukte auf der CWM-Ebene aus den verschiedenen Knotentypen entstehen sollen, etwa indem zu jedem Knotentyp eine geordnete Auflistung aller für die Reduktion zu besuchenden assoziierten Knotentypen sowie der anzulegenden CWM-Konstrukte angegeben wird. Bei der Generierung eines Repository beginnt die Reduktion beispielsweise bei der Wurzel des DSM-Baums, *Information Model*. Hier werden zunächst Repository-weit benötigte Konstrukte auf der CWM-Ebene angelegt, etwa Tabellen für die Speicherung von Verwaltungsdaten oder eine Stored Procedure für die geordnete Generierung von Objekt-IDs. Anschließend werden Objekttypen, dann Relationstypen und schließlich Arbeitskontexttypen auf die CWM-Ebene reduziert, um Blockaden durch Abhängigkeiten von noch nicht reduzierten Konstrukten auszuschließen.

Die Reduktionsregeln nehmen engen Bezug auf die Konstrukte im DSM-Objektgraph, um die gezielte Zuordnung von Konstrukten der CWM-Ebene zu ermöglichen. Aus einem Objekttypen entstehen so ein benutzerdefinierter Datentyp, eine Objekttabelle und eine Menge von Stored Procedures. Abgesehen von der engen Bindung eines Reduktionsschrittes an einen bestimmten Objekttyp im DSM-Modell lassen sich wiederkehrende Aktionen in den verschiedenen Reduktionsschritten erkennen. Solche Aktionen entsprechen wiederkehrenden Java-Codeblöcken in den Reduktionsmethoden.

In den Templates für die Reduktionsmethoden werden feingranularere Java-Codefragmente definiert als in den Templates für die Modellklassen. Ein Fragment entspricht einer elementaren Aktion im Rahmen der Reduktion, beispielsweise der Beschaffung des Namens eines assoziierten Elements. Bei der Generierung von Repositories wird eine solche Aktion etwa benötigt, wenn aus den Attributen eines Objekttyps die Spalten eines benutzerdefinierten Datentyps werden sollen. Anhand der Reduktionsregeln werden dann während des Generierungsprozesses aus den verschiedenen Fragmenten die Reduktionsmethoden der einzelnen Modellklassen konstruiert.

Ein Template-basierter Ansatz bietet sich auch für die fertigen Reduktionsmethoden selbst an, wenn im Rahmen der Reduktion eines DSM-Modells Stored Procedures auf der CWM-Ebene angelegt werden müssen. Anders als bei den restlichen Reduktionsvorgängen steht hier nicht die Instanziierung und Verknüpfung von Objekten, sondern die Erzeugung des textuellen Prozedurrumpfes im Vordergrund. Die Prozeduren verschiedener Objekttypen sind bis auf die Namen der Typen und vereinzelte fakultative Abschnitte oftmals identisch, so dass der Einsatz von Platzhaltern und bedingten Codeblöcken in Templates nahe liegt.

Rendering

Da auch textuelle Darstellungsformen in SQLInteract intern als Baumstruktur verwaltet werden, bestehen hinsichtlich der technischen Anforderungen und der Realisierung weitgehende Parallelen zwischen Reduktion und Rendering. Unterschiede bestehen in der Abbildungsleistung und der Granularität der Beschreibungssprachen. Bei der Reduktion ist die Abbildungsleistung höher als beim Rendering, da die Inhalte auf eine Plattform mit niedrigerem Abstraktionsgrad umgesetzt werden müssen. Die Angabe der Reduktionsregeln kann jedoch im Vergleich zum Java-Code, der schließlich die Reduktion realisiert, relativ abstrakt und grobgranular erfolgen.

Auch beim Rendering muss durch die Regeln zunächst die genaue Traversierung des DSM-Objektgraphen festgelegt werden. Die wichtigste Aktion beim Rendering ist die Ausgabe der Namen der verschiedenen Elemente an den richtigen Stellen und in der jeweils benötigten Form. Im Metamodell der Beschreibungssprache für Repositories werden Namen stets gleich dargestellt; dass auch kompliziertere Fälle denkbar sind, zeigt sich allerdings schon bei Betrachtung eines einfache `CREATE-TABLE`-Statements in SQL, wo Spalten bei der Tabellendefinition zunächst mit ihren Typen aufgelistet werden müssen, bei der Definition von Primär- und Fremdschlüsseln jedoch ohne. Diese Gestaltung des Befehls ist semantisch absolut nachvollziehbar, erfordert aber beim Rendering eine Fallunterscheidung, in welchem Kontext ein Spaltenname gerade dargestellt werden soll.

Wie die Reduktionsmethoden werden die Renderingmethoden in die Modellklassen integriert, so dass das Rendering eines DSM-Modells eine Aktion über mehrere Knoten im Objektgraphen darstellt. Grundsätzlich rendert sich jedes Objekt selbst. Neben der Ausgabe des eigenen Namens und einiger Schlüsselwörter oder sonstiger nichteditierbarer Textfragmente sind dabei oftmals Angaben aus anderen Objekten notwendig. In solchen Fällen werden die benötigten Daten bei assoziierten Objekten angefragt und die Antworten in die eigene Ausgabe integriert. Beim Rendering eines Repository-Modells werden so beispielsweise die Namen der in einen Arbeitskontext einfügbaren Objekttypen im `create-workspace-type`-Statements des Arbeitskontextes dargestellt.

Die Renderingregeln müssen also neben der Angabe der grundsätzlichen Traversierungsstrategie festlegen, über welche Assoziationen zwischen den Knotentypen Anfragen an benachbarte Kno-

ten gerichtet werden dürfen und welche Syntax die textuelle Darstellung besitzen soll. Im Gegensatz zum realisierenden Java-Code muss auf der Regel-Ebene aber nicht jede Klammer und jedes Semikolon einzeln aufgeführt werden; typische Konstellationen lassen sich in einem Ausdruck zusammenfassen. Auf diese Weise hebt sich RenL hinsichtlich Ausdrucksmächtigkeit und Abstraktionsgrad von Java ab.

In technischer Hinsicht läuft die Generierung der Renderingmethoden ähnlich wie die Generierung der Reduktionsmethoden ab. In Templates sind feingranulare Java-Codefragmente enthalten, die jeweils einer elementaren Aktion im Rahmen des Rendering entsprechen. Während der Generierung werden gemäß den Vorgaben durch die Renderingregeln aus den Codefragmenten die Renderingmethoden erzeugt und in die Modellklassen integriert.

Kapitel 7 Zusammenfassung und Ausblick

In diesem Kapitel werden die im Laufe der Beschäftigung mit den Themen dieser Arbeit gewonnenen Erkenntnisse abschließend zusammengefasst und ein Ausblick auf weiterführende Aspekte und Forschungsfelder gegeben. Zunächst werden jedoch zwei Programmiertechnologien vorgestellt, deren Konzepte Parallelen zu den in SQLInteract verwendeten Ansätzen zeigen und daher in einigen Fällen als Denkanstöße dienten.

7.1 Verwandte Arbeit

In den vorigen Kapiteln wurde mehrfach *Intentional Programming* (*IP*) angesprochen. Bei der Untersuchung von Problemstellungen im Bereich der Modellierung von Semantik, insbesondere mit Blick auf die Eingabe und Verfeinerung, stellt IP einen wichtigen Meilenstein dar. Vor einer genaueren Betrachtung von IP soll jedoch *Aspect-Oriented Programming* (*AOP*) vorgestellt werden, das Verfahren gegen die verstreute Verteilung von Funktionalität in Software-Systemen anbietet, die von IP teilweise aufgegriffen werden.

7.1.1 Aspect-Oriented Programming

Es gehört bei der Software-Entwicklung zum guten Stil, die Funktionalität des entworfenen Systems übersichtlich zu gliedern. Unter diese Zielsetzung fällt auch die räumliche Zusammenfassung inhaltlich verwandter Bestandteile des Quellcode. Eine gute Strukturierung des Quellcode erleichtert das Verständnis der Implementierung sowie die Analyse, Modifizierung und Wiederverwendung. Bei der objektorientierten Programmierung bieten sich für diesen Zweck Klassen und ihre Methoden an.

Unglücklicherweise lassen sich nicht alle Bestandteile eines Software-Systems lokal überschaubar implementieren. Während die Kernfunktionalität sich nach den Vorstellungen des Entwicklers in funktionale Einheiten gliedern lassen mag, sind an übergreifenden Mechanismen wie Zugriffssynchronisation, Persistenz oder Speicherverwaltung in der Regel mehrere Komponenten beteiligt, so dass die Funktionalität durch verstreute Codefragmente in den betroffenen Klassen realisiert wird. Diese Erkenntnis führte zur Entwicklung des *Aspect-Oriented Programming* (*AOP*) [KLM+97]. Ziel von AOP ist die Bereitstellung von Techniken und Methoden zur Aufteilung von Problemstellungen in funktionale Komponenten (*functional components*), die sich lokal begrenzt realisieren lassen, und Aspekte (*aspects*), die die Grenzen der funktionalen Komponen-

ten überschreiten (*Crosscutting*). Aus Komponenten und Aspekten sollen dann durch geeignete Komposition die Systemimplementierungen entstehen. Um dieses Ziel zu erreichen, müssen die verschiedenen Bestandteile, die in Software-Systemen auftreten können, in Komponenten und Aspekte kategorisiert werden. Weiterhin werden geeignete Mechanismen für die Komposition benötigt, und für Aspekte selbst muss eine Möglichkeit der Notation gefunden werden.

Die tatsächliche Vermischung von Aspekt-Code mit funktionalem Komponenten-Code wird als *Code Tangling* bezeichnet und tritt auf, wenn beim Systementwurf nicht auf die Trennung von Aspekten und Komponenten geachtet wurde. Beim konventionellen Software-Entwurf ist dies typischerweise verstärkt gegen Ende des Entwicklungsprozesses der Fall. Ein gewisser Anteil von Crosscutting lässt sich allerdings nie vermeiden. Ob eine bestimmte Funktionalität als Komponente oder als Aspekt einzustufen ist, hängt von der Modellierung des Systems ab. Bei einer Änderung der Modellierung zur Entfernung von Crosscutting werden so zwar Aspekte zu Komponenten, umgekehrt an anderer Stelle aber auch Komponenten zu Aspekten. In der Praxis wird sich bei Software-Systemen realistischer Größe und Funktionalität nie ein Zustand erreichen lassen, der frei von Crosscutting ist. Durch eine saubere Trennung von Aspekten und Komponenten lässt sich aber vermeiden, dass Crosscutting zu Code Tangling führt.

Die Verwendung von *Design Patterns* wird zwar gemeinhin begrüßt, bringt mit Blick auf den Systementwurf aber vor allem erhöhte Komplexität, da sich Design Patterns auf Vererbung und Objektkomposition stützen. So wird aus Konstrukten, die ursprünglich in einem Objekt realisiert worden wären, oft ein Verbund mehrerer Objekte mit entsprechend aufwändigerer Verwaltung (Object Schizophrenia). Schwerer wiegt die Tatsache, dass die Verwendung von Design Patterns auch Änderung am Entwurf erschwert (*Preplanning Problem*). Schließlich lässt sich aus dem Quellcode ohne explizite Kommentare nur schwer erkennen, ob ein Design Pattern angewandt wurde (*Traceability Problem*).

Liegt ein sauberer Entwurf nach AOP-Konzepten vor, so ist die enthaltene Funktionalität in Komponenten und Aspekte aufgeteilt. Für die Implementierung müssen die Aspekte allerdings wieder mit den Komponenten vermischt werden, um ihre Funktionalität realisieren zu können. Der Prozess der Zusammenfügung von Komponenten und Aspekten, der die in Aspekte aufgetrennten Konzepte wieder integriert, wird als *Weaving* bezeichnet. Bis zum Zeitpunkt des Weaving wird eine möglichst geringe Kopplung zwischen Komponenten und Aspekten angestrebt; diese Trennung wird bis in den Quellcode des Software-Systems aufrechterhalten. Der Entwickler hat so auch während der Erstellung des Quellcode noch die größtmögliche Übersicht. Auf den fertigen Quellcode wird das Weaving als Code-Transformation angewandt. Ergebnis der Transformation ist Code, in dem Komponenten und Aspekte vermischt sind (*Tangled Code*). Die Transformation kann dabei statisch vor der Laufzeit des Systems etwa durch einen Compiler oder Präprozessor erfolgen. Der Vorteil dieser Variante ist eine größere Effizienz des Endprodukts. Alternativ kann das Weaving auch dynamisch während der Laufzeit mithilfe von *Reflection* vorgenommen werden. In diesem Fall wird der vermischte Code nicht tatsächlich generiert, sondern der logische Kontrollfluss durch den Code rekonfiguriert, so dass auch zur Laufzeit Änderungen an der Programmstruktur möglich sind. Die Ausführung ähnelt dann eher einer Interpretation als dem Ablauf eines kompilierten Programms, was einen entsprechenden Zuwachs an Ausführungsaufwand mit sich bringt.

Die Darstellung von Aspekten während des Systementwurfs kann mithilfe von konventionellen Erweiterungsbibliotheken der Implementierungssprache, Spracherweiterungen der Implementierungssprache oder einer eigenen Beschreibungssprache für Aspekte erfolgen. Der Ansatz einer

eigenen Beschreibungssprache bietet die Möglichkeit, domänenspezifische Konstrukte zu verwenden, was eine höhere Ausdrucksmächtigkeit bei maximal kompakter Darstellung erlaubt. Je abstrakter und ausdrucksmächtiger allerdings die Beschreibungen der Aspekte erfolgen, desto größer wird die Komplexität der Weaving-Transformation, die diese Beschreibungen wieder auf die Implementierungssprache umsetzen muss.

Bei SQLInteract findet sich der Ansatz der Trennung von lokal eingrenzbarer und objektübergreifender Funktionalität vor allem auf der DSM-Ebene. Die benutzerdefinierten Routinen für die Verwaltung der Daten in der Datenbank müssen oft auf mehrere Objekte zugreifen oder andere Routinen aufrufen, etwa bei der Propagierung von Operationen über Beziehungen. Während der Modellierung auf der DSM-Ebene wird der Benutzer mit diesen komplexen Details nicht belastet. Erst bei der Generierung des Datenbank-Schemas werden die Verknüpfungen zwischen den verschiedenen Modellkonstrukten ausgewertet und spezialisierter Code für die Verwaltungsoperationen erzeugt, was einem Weaving-Prozess bei der aspektorientierten Programmierung vergleichbar ist. Darüber hinaus lassen sich durch unterschiedliche Renderingregeln verschiedene Ansichten auf das Modell realisieren, die auch als Darstellung verschiedener Aspekte des Modells ausgeführt sein können.

7.1.2 Intentional Programming

Bei *Intentional Programming* (*IP*) handelt es sich um einen vollständig neuen Ansatz für die Software-Entwicklung, der sich nicht auf eine neue Art von Sprachkonstrukten beschränkt, sondern den gesamten Entwicklungsprozess umfasst. IP bietet eine integrierte Umgebung für Programmierung und Metaprogrammierung, in dem der Entwickler seinen Systementwurf von der ersten Modellierung bis zum ausführbaren Code durchführen kann. Entscheidendes Merkmal der Umgebung ist die Erweiterbarkeit. Über Erweiterungsbibliotheken (*Extension Libraries*) kann die gesamte Umgebung mit Editor, Compiler, Debugger und mehr an die Bedürfnisse des Entwicklers angepasst werden.

Die Flexibilität und Erweiterbarkeit setzt sich bei der Bearbeitung von Quellcode fort. Der Quellcode-Editor ist kein einfacher ASCII-Editor, sondern stellt die Sprachabstraktionen in ihrer reinen Form als so genannte *Intentions* dar. Der Entwickler arbeitet so direkt auf einem abstrakten Syntaxbaum, wodurch auch die Notwendigkeit eines Parsers für den Quellcode entfällt. Dieses Eingabekonzept wird als *Active Source* bezeichnet. Der Quelltext ist kein schlichter Text, der als reines Datenobjekt vom Quellcode-Editor verwaltet wird, sondern eine eigenständige Einheit, die intern auf einer Graphenstruktur beruht und ihre Darstellung im Editor-Fenster sowie die Eingabe von Quellcode-Konstrukten über so genannte *Rendering*- und *Type-In Methods* selbstständig regelt. Diese sind nicht mit Methoden von Objekten bei der objektorientierten Programmierung zu verwechseln, sondern arbeiten direkt zur Entwurfszeit auf dem Syntaxbaum. Darüber hinaus existieren unter anderem *Debugging Methods* für die Fehlersuche und *Refactoring Methods* für die Optimierung des Syntaxbaums. Automatisiertes Refactoring fällt verhältnismäßig leicht, da der Quellcode in einer direkt zu verarbeitenden Form vorliegt und der Entwickler die Struktur mit abstrakten Informationen zur Semantik anreichern kann. Alle Methoden sind spezifisch für die verwendete Programmiersprache und können so den Entwickler bei seiner Arbeit optimal unterstützen. Die Darstellung im Editor-Fenster ist dabei nur eine Visualisierung der modellierten Sprachabstraktionen. Mehrere Visualisierungen sind möglich, etwa um verschiedene Aspekte des Quellcode herauszustellen. Der Editor entspricht so eher einem komfortablen WYSIWYG-Editor als dem gewohnten Bild eines dürren Text-Editors. Trotz der Ver-

waltung des Quellcode als Syntaxbaum sind dabei während der Bearbeitung strukturell nicht korrekte Zustände zulässig; die Struktur wird erst bei der Kompilierung geprüft.

Intentions können Sprachabstraktionen beliebiger Sprachen darstellen, wenn eine entsprechende Extension Library geladen ist. Besonders interessant sind dabei domänenspezifische Sprachen, die für den Entwickler in einer für ihn leicht verständlichen Form dargestellt, intern aber in einer ganz andere Repräsentation verwaltet werden können. Auf diese Weise lässt sich eine natürliche Notation erzielen, ohne Abstriche bei der Flexibilität oder der Performanz der späteren lauffähigen Realisierung hinnehmen zu müssen. IP entkoppelt auf diese Weise die Darstellung vollständig vom fertigen Code. Der Entwickler könnte beispielsweise im Editor mathematische Formeln in typischer mathematischer Schreibweise bearbeiten, die später bei der Kompilierung optimiert und als schwer lesbarer, aber leistungsfähiger Algorithmus realisiert werden. Durch die Flexibilität bei den Sprachen eignet sich IP auch sehr gut für die Spezifizierung von Aspekten im Rahmen der aspektorientierten Programmierung. Der Entwickler kann die Aspekte in einer speziell zugeschnittenen Sprache angeben und in verschiedenen Darstellungen bearbeiten, und das System übernimmt das Weaving bei der Kompilierung. Ebenso wird die Metaprogrammierung erleichtert, da in IP für die Implementierung von Anwendungen und Erweiterungsbibliotheken dieselben Intentions verwendet werden können, ohne jedoch den Entwickler darauf einzuschränken. IP selbst wurde ursprünglich in C implementiert; nachdem Intentions für die Sprachkonstrukte von C entwickelt worden waren, wurde das System 1995 durch ein Bootstrapping auf IP umgestellt, indem der ursprüngliche C-Quellcode importiert wurde. Seit diesem Zeitpunkt wird IP in sich selbst weiterentwickelt.

Der Kompilierungsvorgang selbst wird in IP *Reduktion* genannt und entspricht einer Baumtransformation. Die Konstrukte im abstrakten Syntaxbaum werden auf tiefere, konkretere Ebenen umgesetzt, bis letztlich eine Abstraktionsstufe erreicht ist, aus der sich direkt lauffähiger Code erzeugen lässt. Diese letzte Verfeinerungsstufe heißt *R-Code*. Es können verschiedene R-Codes für unterschiedliche Zielplattformen existieren. R-Code wird durch ein *Backend* genanntes Modul von IP in lauffähigen Code umgesetzt.

Die Reduktion wird gestartet, indem die Methode `Rcode` des Wurzelknotens des modellierten Syntaxbaums aufgerufen wird, auf die dieser letztlich mit seiner reduzierten Repräsentation antwortet. Zuvor leitet der Wurzelknoten jedoch den Aufruf an seine Söhne weiter, die ebenso verfahren, so dass der ganze Syntaxbaum traversiert wird. Die reduzierten Repräsentationen werden dann von den Blättern des Baums über die inneren Knoten bis zur Wurzel hochgereicht, wobei jeder Knoten auf dem Weg die Information ergänzt. Während des Reduktionslaufs findet eine Überprüfung und Optimierung der Baumstruktur statt. Die Optimierung kann dabei domänenspezifisch erfolgen, da die einzelnen Sprachabstraktionen ihre Optimierungsregeln selbst bereitstellen und nicht auf einen externen Optimierer angewiesen sind. Die reduzierten Repräsentation werden beim Reduktionslauf in den ursprünglichen Syntaxbaum integriert. Da Reduktionsmethoden keine Knoten oder Assoziationen löschen dürfen, wächst der Syntaxbaum bei jedem Reduktionslauf monoton. Für die Erzeugung der reduzierten Repräsentation dürfen die Knoten im Baum außerdem Anfragen nach Daten nur an direkte Nachbarn richten. Auf diese Weise kann das System den Reduktionslauf überwachen und über Abhängigkeiten zwischen den Knoten Buch führen. Die Verfolgung der Abhängigkeiten ist wichtig, da die Rückgabewerte der Knoten auf eine Anfrage während eines Reduktionslaufs konstant bleiben müssen. Sollte im Zuge der Reduktion ein Knoten bei einer wiederholten Anfrage einen anderen Rückgabewert liefern als zuvor, wird der Reduktionslauf bis zu einem Punkt zurückgerollt, wo alle Rückgabewerte konsis-

tent sind, und von dort aus in einer anderen Reihenfolge fortgefahren. Während des Reduktionslaufs wird so eine Reihenfolge der Baumtraversierung gesucht, bei der alle Rückgabewerte aller Knoten konsistent sind.

Intentional Programming als erweiterbare Programmierumgebung vermeidet die Nachteile, die bei der Programmierung mit allgemeinen Sprachen entstehen. Allgemeine Sprachen können beispielsweise in ihren Konstrukten keine komplexen Informationen über die Struktur des ursprünglichen Entwurfs speichern. Dem Code der fertigen Implementierung kann man nur schwer ansehen, welche abstrakten Konzepte zu genau dieser Realisierung geführt haben. IP verknüpft die Ausdrucksmächtigkeit der domänenspezifischen Sprachen mit der Leistungsfähigkeit und dem Verbreitungsgrad allgemeiner Programmiersprachen, indem während der Modellierung dem Entwickler domänenspezifische Konstrukte präsentiert werden, bei der Reduktion die Funktionalität aber auf allgemeine Sprachen übertragen werden kann. Schließlich ist IP auch komponentenbasierter Software-Entwicklung überlegen, da die grobe Struktur von Komponenten sich kaum für den konkreten Anwendungsfall anpassen lässt und insbesondere keine domänenspezifischen Optimierungen oder Weaving von Aspekten unterstützt.

SQLInteract bietet nicht die universelle Erweiterbarkeit von IP. In vielen Hinsichten kommen jedoch vergleichbare Konzepte zum Einsatz. Beispielsweise werden textuelle Darstellungsformen intern in einer Baumstruktur verwaltet. Die Kontrolle von Darstellung und Eingabe erfolgt allerdings durch den Editor, nicht wie bei der Active Source von IP durch den Quellcode selbst. Verschiedene Darstellungsformen der modellierten Konstrukte können dennoch durch Angabe unterschiedlicher Renderingregeln realisiert werden. Der Reduktionsvorgang ist auch bei SQLInteract grundsätzlich eine Baumtraversierung. Während bei IP jedoch der vorhandene Syntaxbaum monoton wächst, erstellt SQLInteract aus abstrakten Spezifikationen auf der DSM-Ebene neue Modelle auf der CWM-Ebene. Dabei werden, wie es auch bei IP möglich ist, domänenspezifische Konstrukte auf allgemeine Konstrukte umgesetzt.

7.2 Zusammenfassung

In dieser Diplomarbeit wurden anhand des prototypischen Entwurfswerkzeugs SQLInteract verschiedene Aspekte der Semantikmodellierung auf dem Gebiet datenbankintensiver Anwendungen untersucht. Dabei lassen sich drei Abstraktionsebenen unterscheiden. Auf der *CWM-Ebene* kann der Benutzer, unterstützt durch eine Reihe von Eingabehilfen, Datenbank-Schemas modellieren und Ad-hoc-Anfragen an ein Datenbanksystem stellen. Auf der *DSM-Ebene* können auf dieselbe Weise abstrakte Spezifikationen für Repositories erstellt werden, aus denen SQLInteract das konkrete Produkt generiert. Schließlich können mithilfe eines *Metagenerators* DSM-Module für die Unterstützung anderer Produktlinien aus entsprechenden Beschreibungen von Produktlinien generiert werden.

7.2.1 CWM-basierter Schemaentwurf und HCI-Konzepte

Auf der CWM-Ebene wird eine Implementierung des *Common Warehouse Metamodel* für die interne Speicherung der modellierten Datenbank-Schemas genutzt. CWM ist von der Object Management Group für den Austausch von Metadaten über Datenbanken und Data Warehouses in heterogenen Umgebungen entwickelt worden und basiert auf den anerkannten OMG-Standards MOF, UML, und XMI. Für den Einsatz in SQLInteract wurden die relevanten Teile des

Metamodells in Java-Klassen umgesetzt. Es handelt sich dabei größtenteils um Klassen aus den Packages *Object Model* und *Relational*.

Die Bedienung der CWM-Ebene kann auf zwei gleich mächtige Weisen erfolgen. Im *CWM-Baum* werden Instanzen der CWM-Metamodellklassen als Baumknoten hierarchisch dargestellt und können über Kontextmenüs manipuliert werden. Im *Schema-Editor* können textuelle DML-Befehle eingegeben und bearbeitet werden, wobei der Benutzer durch verschiedene Hilfsfunktionen unterstützt wird. CWM-Baum und Schema-Editor sind synchronisiert und arbeiten auf demselben Datenbestand.

7.2.2 Spezifizierung von Produktbestellungen

Auf der DSM-Ebene besteht die Möglichkeit, Produktbestellungen für Mitglieder aus der Produktlinie der objektorientierten Repositories zu erstellen. Wie auf der CWM-Ebene stehen eine Baumansicht und eine textuelle Darstellung für die Eingabe und Bearbeitung zur Verfügung, die untereinander synchronisiert sind. Aus einer Produktbestellung generiert SQLInteract das konkrete Produkt in Form eines Datenbank-Schemas mit benutzerdefinierten Datentypen und Routinen, das zur Realisierung der Funktionalität an ein Datenbanksystem übertragen werden kann.

Hintergrund der abstrakten Spezifizierung mit anschließender Generierung sind die Konzepte *der generativen Programmierung* und der *modellgetriebenen Architekturen*. Der Ansatz der modellgetriebenen Architekturen besteht darin, die Funktionalität eines Software-Systems zunächst unter Abstraktion von den technischen Details der Realisierung in Modellform zu entwerfen. Nach etwaigen Verfeinerungen auf dieser Ebene der *plattformunabhängigen Modelle* kann die Funktionalität auf ein *plattformspezifisches Modell* umgesetzt werden, das Bezug auf technische Details der Zielplattform nimmt. Aus einem plattformspezifischen Modell, bei ausreichend verfeinerter Modellierung auch direkt aus einem plattformunabhängigen Modell, kann dann ausführbarer Code generiert werden. Ziel ist dabei, die Umsetzungen auf niedrigere Abstraktionsebenen möglichst weitgehend zu automatisieren.

Auch bei der generativen Programmierung geht es um die automatisierte Erstellung der Endprodukte. Die Grundlage bilden hier jedoch eine Menge von elementaren *Implementierungskomponenten*, die als Bausteine für eine ganze Systemfamilie technisch verwandter Produkte dienen. Anhand einer Spezifikation erstellt ein *Generator* aus den Implementierungskomponenten durch geeignete Zusammenstellung und Konfiguration das gewünschte Produkt. Für die Spezifikation kommt zweckmäßigerweise eine *domänenspezifische Konfigurationssprache* zum Einsatz, deren Konstrukte exakt auf das Einsatzgebiet zugeschnitten sind, so dass der Benutzer sich nicht in irrelevanten Details verliert oder von der Komplexität der Sprache überfordert wird. Bei den Untersuchungen zeigte sich, dass allgemeine Spezifikationssprachen wie AsmL oder UML mit Action Semantics, OAL oder ASL für Zwecke der Produktspezifikation nicht geeignet sind, da ihre allgemein gehaltenen, wenig ausdrucksmächtigen Abstraktionen zu übermäßig komplexen Spezifikationen führen würden. Aus diesem Grund unterstützt SQLInteract die Umsetzung von Spezifikationen in domänenspezifischen Sprachen auf CWM-Konstrukte.

7.2.3 Generierung weitere DSM-Module

SQLInteract ist nicht auf die Produktlinie der objektorientierten Repositories festgelegt, sondern erlaubt über *DSM-Module* die Einbindung weiterer Produktlinien auf dem Gebiet der datenbankintensiven Anwendungen. Für die Erstellung eines DSM-Moduls ist die Beschreibung einer

Produktlinie mit einem Metamodell ihrer Spezifikationssprache, einer Menge von Rendering-Regeln für die Abbildung der Modellkonstrukte auf die textuelle Darstellung sowie einer Menge von Reduktions-Regeln für die Umsetzung von Spezifikationen auf konkrete Datenbank-Schemas erforderlich. Aus diesen Eingaben werden durch einen *Metagenerator* Java-Klassen generiert, die Metamodell, Rendering- und Reduktions-Regeln implementieren. Die generierten Klassen werden anschließend mit dem konstanten Teil von SQLInteract zusammengefügt, so dass eine spezielle Ausprägung von SQLInteract für eine bestimmte Produktlinie entsteht. SQLInteract bildet auf diese Weise selbst eine Produktlinie, deren Mitglieder jeweils Spezifikationswerkzeug und Generator für eine Produktlinie auf dem Gebiet der datenbankintensiven Anwendungen sind. Aus diesem Grund kann SQLInteract als *Meta-Produktlinie* bezeichnet werden.

7.2.4 Fazit

Die Beschäftigung mit den verschiedenen Aspekten der Modellierung von Semantik auf unterschiedlichen Abstraktionsstufen zeigt, dass domänenspezifische Sprachen bei der Semantikmodellierung allgemeinen Spezifikationssprachen überlegen sind. Durch die Festlegung auf eine Anwendungsdomäne erhalten die Sprachkonstrukte eine wesentlich höhere Ausdrucksmächtigkeit, so dass sich mit wenigen Ausdrücken spezifizieren lässt, wozu in einer allgemeinen Sprache ein komplexer Objektgraph oder eine seitenlange Beschreibung notwendig wäre. Die höhere Ausdrucksmächtigkeit domänenspezifischer Sprachen wird allerdings mit einem eng begrenzten Einsatzfeld erkauft. Weiterhin lassen sich Spezifikationen in domänenspezifischen Sprachen nicht ohne weiteres in ausführbaren Code wandeln. An einer Stelle auf dem Weg von der domänenspezifischen Sprache zum Maschinencode müssen die Bezüge auf den Domänenkontext aufgelöst und in Konstrukte einer allgemeinen Sprache umgesetzt werden. Wenn es gelingt, diese *Reduktion* automatisiert durchführen zu lassen, so dass der menschliche Benutzer ausschließlich auf der domänenspezifischen Ebene arbeitet, steht der Verwendung domänenspezifischer Sprachen nichts im Wege. Die Anforderungen an die Reduktion sind allerdings beträchtlich. Neben der schieren Abbilddungsleistung aus der Domäne in die Zielsprache sollte im Zuge der Reduktion beispielsweise auch eine Optimierung der modellierten Semantik durchgeführt werden, die möglichst eng auf die technischen Gegebenheiten der Zielplattform Bezug nehmen sollte. Im Beispiel der datenbankintensiven Anwendungen wäre hier etwa die Berücksichtigung unterschiedlicher SQL-Dialekte der verschiedenen Datenbank-Hersteller oder die Zusammenfassung mehrerer zusammenhängender SQL-Befehle in einer benutzerdefinierten Routine denkbar. Schließlich bleibt zu bedenken, dass sich Domänen im Laufe der Zeit verschieben oder in ihrer Gestalt ändern können, da sie sich kaum formal definieren lassen, sondern typischerweise aus einer Übereinkunft der Nutzer über die Inhalte und Ausdehnung entstehen. Eine domänenspezifische Sprache sollte daher leicht änderbar sein, damit stets die volle Ausdrucksmächtigkeit zu erhalten ist.

7.3 Ausblick

Für die Zukunft sind einige Erweiterungen an SQLInteract vorstellbar, die teilweise eher technischer Natur sind, teilweise aber auch ganz neue Forschungsfelder eröffnen.

Mit Blick auf den unmittelbaren Benutzerkomfort wäre eine explizite, mehrstufige *Undo*-Funktion erstrebenswert, um Modellierungsschritte ohne großen Aufwand zurücknehmen zu können. Ähnlich wie es in den textuellen Editorfeldern bereits der Fall ist, sollte der Benutzer auch bei der Modellierung in den Baumansichten durch Automatismen und Vorgabewerte unterstützt werden. Bei der Reduktion einer Produktspezifikation auf ein Datenbank-Schema sollte ein Systemkatalog angelegt werden, in dem Metadaten über das Repository abgelegt werden, um generischen Zugriff durch Applikationen zu ermöglichen.

Ein interessantes Forschungsfeld, angelehnt an die Konzepte von Intentional Programming, wäre das *Refactoring* erstellter Modelle, vor allem auf der DSM-Ebene. Eine Modelloptimierung könnte auch automatisch im Rahmen der Reduktion erfolgen.

Aus dem Bereich der modellgetriebenen Architekturen stammt das Konzept des *Round Trip Engineering*, bei dem die Erstellung des Produkts nicht das Ende eines linearen Produktionsvorgangs ist, sondern Endprodukte analysiert werden, um Modellinformationen aus ihnen abzuleiten. Im Falle von SQLInteract würde beim Round Trip Engineering also versucht werden, aus bestehenden Datenbank-Schemas ein abstraktes domänenspezifisches Modell zu erstellen.

Liegen mehrere Schemas vor, kann auf diese Weise unter Umständen ein gemeinsames Metamodell generiert werden. Dieser Prozess wird *Product Line Mining* genannt. Dabei geht es darum, im Rahmen einer Analyse die Gemeinsamkeiten aus ursprünglich getrennt entwickelten Produkten zu extrahieren, die auch als Systemfamilie realisiert werden könnten. Im Erfolgsfall kann so rückwirkend ein Metamodell der Systemfamilie erstellt werden.

In SQLInteract wurde nur die Erstellung von datenbankintensiven Anwendungen betrachtet. Damit der Benutzer auch im Betrieb etwa eines Repository von den technischen Details der Realisierung abstrahieren kann, ist eine Zwischenschicht notwendig, die domänenspezifische Anfragen in einer entsprechenden Anfragesprache entgegennimmt und in eine Menge von Standard-SQL-Anfragen übersetzt. Die Ergebnisse dieser Anfragen müssen dann in geeigneter Form zusammengesetzt werden, um dem Datenmodell auf der domänenspezifischen Ebene zu entsprechen, woraufhin sie an den Benutzer ausgegeben werden. Auf diese Weise arbeitet der Benutzer bei der Erstellung des Repository und im späteren Betrieb ausschließlich auf der domänenspezifischen Ebene. Die Realisierung einer solchen Zwischenschicht könnte beispielsweise in Form eines Präcompilers erfolgen, wenn programmatische Repository-Zugriffe aus einer Anwendung bedient werden sollen. Für den interaktiven Betrieb wäre ein Parser notwendig, der Ad-hoc-Anfragen umsetzen und beantworten kann.

Um Anfragen intern darstellen und verarbeiten zu können, wäre eine Erweiterung des Datenmodells auf beiden Ebenen ein geeigneter Ansatz. Analog zu den Modellbäumen könnten Anfragen in speziellen Befehlsbäumen dargestellt werden. Die Umsetzung von der DSM- auf die CWM-Ebene wäre in diesem Fall auch ein Reduktionsvorgang von Modell zu Modell. Die textuelle Darstellung von Anfragen auf der CWM-Ebene existiert bereits; auf der DSM-Ebene müsste neben dem Befehlsbaum auch noch ein Editorfeld für die textuelle Bearbeitung von Anfragen angelegt werden.

Literaturverzeichnis

[ANS99] American National Standards Institute (ANSI):
Database Languages – SQL – Part 2: Foundation (SQL/Foundation)
American National Standards Institute, New York, 1999

[Apa03a] Apache Software Foundation:
Velocity User Guide, v1.3.1
verfügbar als http://jakarta.apache.org/velocity/user-guide.html
März 2003

[Apa03b] Apache Software Foundation:
Velocity Developer's Guide, v1.3.1
verfügbar als http://jakarta.apache.org/velocity/developer-guide.html
März 2003

[Apa03c] Apache Software Foundation:
VTL Reference Guide, v1.3.1
verfügbar als http://jakarta.apache.org/velocity/vtl-reference-guide.html
März 2003

[Boo93] Booch, G.:
Object Oriented Design
Benjamin/Cummings, Redwood City, 1993

[BS01] Barnett, M. and Schulte, W.:
The ABCs of Specification: AsmL, Behavior, and Components
Informatica, 25(4): 517-526
November 2001

[CE00] Czarnecki, K. and Eisenecker, U.:
Generative Programming. Methods, Tools and Applications.
Addison-Wesley, Boston, 2000

[IBM03] IBM Corporation:
IBM Ease of Use: Design Concepts
verfügbar als http://www-3.ibm.com/ibm/easy/eou_ext.nsf/Publish/567PV
Juni 2003

[Jac94] Jacobson, I.:
Object-Oriented Software Engineering: A Use Case Driven Approach
Addison-Wesley, Reading, 1994

[JCP01] Java Community Process:
UML/EJB Mapping Specification 1.0
verfügbar als http://www.jcp.org/aboutJava/communityprocess/review/jsr026
Juni 2001

[KC02] Kennedy Carter Ltd.:
Configurable Code Generation in MDA using iCCG, v3.0
verf. als http://www.kc.com/cgi-bin/download.cgi?action=ctn/CTN_27v3_0.pdf
August 2002

[KLM+97] Kiczales, G., Lamping, J., Mendhekar, A. et. al.:
Aspect-Oriented Programming
In: Proceedings of the ECOOP 97, Finland. Springer-Verlag LNCS 1241, S. 220-242
Juni 1997

[LW94] Liskov, B. and Wing, J.:
A Behavioral Notion of Subtyping
ACM Transactions on Programming Languages and Systems, 16(6):1811-1841
November 1994

[MM01] Mitchell, R. and McKim, J.:
Design by Contract, by Example
Addison-Wesley, Reading, Oktober 2001

[Nov02] Novosoft:
Novosoft Metadata Framework, Version 0.1.4
verfügbar bei http://nsuml.sourceforge.net
Oktober 2002

[OMG01a] Architecture Board of the Object Management Group:
Model Driven Architecture - A Technical Perspective
verfügbar als http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01
Juli 2001

[OMG01b] Object Management Group:
Common Warehouse Metamodel (CWM) Specification, Version 1.0
verfügbar als http://www.omg.org/cgi-bin/doc?formal/01-10-01
Oktober 2001

[OMG02a] Object Management Group:
XML Metadata Interchange (XMI) Specification, Version 1.2
verfügbar als http://www.omg.org/cgi-bin/doc?formal/2002-01-01
Januar 2002

[OMG02b] Object Management Group:
UML Profile for CORBA Specification, Version 1.0
verfügbar als http://www.omg.org/cgi-bin/doc?formal/02-04-01
April 2002

[OMG02c] Object Management Group:
Meta Object Facility (MOF) Specification, Version 1.4
verfügbar als http://www.omg.org/cgi-bin/doc?formal/2002-04-03
April 2002

[OMG02d] Object Management Group:
Common Object Request Broker Architecture (CORBA) Specification, Version 3.0.2
verfügbar als http://www.omg.org/cgi-bin/doc?formal/02-12-02
Dezember 2002

[OMG03a] Object Management Group:
Unified Modeling Language (UML) Specification, Version 1.5
verfügbar als http://www.omg.org/cgi-bin/doc?formal/03-03-01
März 2003

[OMG03b] Object Management Group:
Common Warehouse Metamodel (CWM) Specification, Version 1.1
verfügbar als http://www.omg.org/cgi-bin/doc?formal/03-03-02
Oktober 2001

[PHP03] The PHP Group:
PHP Manual
verfügbar bei http://www.php.net/download-docs.php
August 2003

[Pro02] Project Technology Inc.:
Object Action Language Manual, Version 1.4
verfügbar als http://www.projtech.com/pdfs/bp/oal.pdf
Dezember 2002

[Pro03] Project Technology Inc.:
BridgePoint Development Suite
verfügbar als http://www.projtech.com/pdfs/xtuml/bridgepoint.pdf
Mai 2003

[Rum91] Rumbaugh, J.:
Object-Oriented Modeling And Design
Prentice Hall, Englewood Cliffs, 1991

[Shn92] Shneiderman, B.:
Designing the User Interface
Addison-Wesley, Reading, 1992

[Sie+01] Siegel, J. and the OMG Staff Strategy Group:
Developing in OMG's Model-Driven Architecture
verfügbar als ftp://ftp.omg.org/pub/docs/omg/01-12-01.pdf
November 2001

[Sim95] Simonyi, C.:
The Death Of Computer Languages, The Birth of Intentional Programming
verfügbar als http://research.microsoft.com/research/pubs/view.aspx?tr_id=4
September 1995

[Sol+00] Soley, R. and the OMG Staff Strategy Group:
Model Driven Architecture
verfügbar als ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf
November 2000

[Sun01a] Sun Microsystems, Inc.:
Java Look and Feel Design Guidelines
Addison-Wesley, Reading, 2001

[Sun01b] Sun Microsystems, Inc.:
JavaServer Pages Specification, Version 1.2
verfügbar als http://www.jcp.org/aboutJava/communityprocess/final/jsr053
September 2001

[Sun02] Sun Microsystems, Inc.:
Java Metadata Interface (JMI) Specification, Version 1.0
verfügbar bei http://java.sun.com/products/jmi
Juni 2002

[W3C00] World Wide Web Consortium:
Extensible Markup Language (XML) 1.0 (Second Edition)
verfügbar als http://www.w3.org/TR/2000/REC-xml-20001006
Oktober 2000

[WKC+01] Wilkie, I., King, A., Clarke, M. et. al.:
The UML Action Specification Language Reference Guide, v2.5c
verf. als http://www.kc.com/cgi-bin/download.cgi?action=ctn/CTN_06v2_5c.pdf
Juli 2001