Bachelorarbeit

Heiko Bohnsack

Generierung von Prototypen auf der Basis
vorläufiger UML-Klassendiagramme

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Heiko Bohnsack

Generierung von Prototypen auf der Basis vorläufiger UML-Klassendiagramme

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Jörg Raasch
Zweitgutachter : Prof. Dr. Olaf Zukunft

Abgegeben am 14. September 2007

Heiko Bohnsack

Thema der Bachelorarbeit

Generierung von Prototypen auf der Basis vorläufiger UML-Klassendiagramme

Stichworte

MDA, MDSD, openArchitectureWare, XPand, Template, Workflow, Prototyp, UML, Modellgetriebene Softwareentwicklung, Swing, Hibernate, Cartridge, GUI, Transformation, Klassendiagramm, Code-Generierung, Domäne, DSL, Metamodell, abstrakte und konkrete Syntax

Kurzzusammenfassung

Diese Arbeit beschäftigt sich mit der Generierung von Prototypen aus vorläufigen UML-Klassendiagrammen. Für viele Softwareprojekte dient als Ausgangspunkt ein UML-Modell. Anhand von diesem Modell, werden Protoypen entwickelt. Leider ist dieses sehr zeitaufwendig und fehleranfällig. Durch Fehlinterpretation seitens der Entwickler oder nicht konsequenter Pflege der Modelle bei Änderung am Prototypen, kommt es zu Inkonsistenzen zwischen den Modellen und Prototypen. Deswegen wäre es wünschenswert, wenn bereits aus einem vorläufigen UML-Klassendiagramm ein Prototyp generiert werden kann. So ist es möglich, dem Kunden zeitnah einen bereits lauffähigen Prototypen zu präsentieren. Um dies zu ermöglichen, braucht man leistungsfähige MDA-Werkzeuge. In dieser Arbeit werden einige von Ihnen vorgestellt und verwendet, um einen lauffähigen Prototypen zu generieren. Dieser automatisch generierte Prototyp verfügt über eine GUI und eine Datenbankanbindung.

Heiko Bohnsack

Title of the paper

Generation of prototypes from provisional UML class-diagrams

Keywords

MDA, MDSD, openArchitectureWare, XPand, Template, Workflow, Prototype, UML, Model Driven Softwaredevelopment, Swing, Hibernate, Cartridge, GUI, Transformation, class-diagram, Code-Generation, Domane, DSL, Metamodel, abstract and concrete Syntax

Abstract

This work is concerned with the generation of prototypes from provisional UML class-diagrams. For many software projects an UML model serves as a starting point. On the basis of this model, prototypes are developed. Unfortunately this is very time-consuming and error-prone. By misinterpretation on the part of the developers or inconsistent care of the models during the alteration of the prototype, it comes to inconsistencies between the models and prototypes. Therefore it would be desirable to be able to already generate a prototype from a provisional UML class-diagram. Thus it is possible to present an already executable prototype to the customer in a contemporary way. To make this possible, one needs efficient MDA-tools. In this work some of them will be presented and applied to generate an executable prototype. This automatically generated prototype is equipped with a GUI and a database binding.

Jeder dumme Mensch kann einen Käfer zertreten, aber alle Professoren der Welt können keinen herstellen.
Arthur Schopenhauer

Danksagung

Ich möchte vor allem meinem Betreuer Prof. Dr. rer. nat. Jörg Raasch danken, der mir immer mit Rat und Tat zur Seite stand. Für die Hilfe bei der Übersetzung der Kurzzusammenfassung bin ich Frau Julia Mohr überaus dankbar. Außerdem dankbar bin ich meiner Familie, die mich bis zuletzt unterstützt hat.

Inhaltsverzeichnis

1 Einführung **9**
1.1 Motivation . . . 9
1.2 Ziel der Arbeit . . . 9
1.3 Aufbau der Arbeit . . . 10

2 Vision **12**
2.1 Kurzfristiges Ziel . . . 12
2.2 Zukunft . . . 13

3 Marktanalyse **14**
3.1 Anforderungsprofil . . . 14
3.2 MDA . . . 14
3.3 MDSD . . . 15
3.3.1 MDA/MDSD-Werkzeuge . . . 15
3.4 Fazit . . . 21

4 Grundlagen **22**
4.1 UML . . . 22
4.1.1 Klassendiagramm . . . 22
4.1.2 Notationselemente . . . 23
4.2 MDSD . . . 29
4.2.1 Domäne . . . 29
4.2.2 Metamodell . . . 30
4.3 openArchitectureWare . . . 32
4.3.1 Template . . . 32
4.4 Hibernate . . . 33
4.4.1 Hibernate API . . . 34
4.4.2 Hibernate Cartridge . . . 36
4.5 Swing . . . 36

5 Architektur **37**
5.1 Fachliche Architektur . . . 37
5.1.1 Fachliche Architektur des Werkzeugs . . . 37

5.1.2 Fachliche Architektur der Anwendung 38
5.2 Technische Architektur 39
5.2.1 Technische Architektur des Werkzeugs 39
5.2.2 Technische Architektur der Anwendung 40
5.3 Spezifikationen 40

6 Realisierung **41**
6.1 Installation von oAW 41
6.2 Transformationsregeln 41
6.2.1 Datenbankschicht 41
6.2.2 Anpassungschicht 43
6.2.3 User-Interface 44
6.3 Erzeugen der Datenbankschicht 45
6.3.1 Hibernate Cartridge 45
6.3.2 Installation 46
6.3.3 Konfiguration 46
6.3.4 Modellierung 46
6.3.5 Generierung 49
6.4 Erzeugen der Anpassungschicht 49
6.5 Erzeugen des User-Interfaces 49
6.5.1 Tabellenwartung 50
6.5.2 Detailwartung 50
6.6 Machbarkeitsbeweis 51

7 Schluss **52**
7.1 Zusammenfassung 52
7.2 Grenzen des Werkzeuges 52
7.3 Weiterentwicklungsmöglichkeiten 53
7.4 Fazit 53

Literaturverzeichnis **54**

1 Einführung

Diese Arbeit beschäftigt sich mit MDSD (Modell Driven Software Development) und MDA (Modell Driven Architecture). Es soll aus einem vorläufigen UML-Klassendiagramm ein lauffähiger Prototyp mit Datenbankanbindung und Grafischer Benutzeroberfläche (GUI) generiert werden. Es gibt bereits eine Vielzahl von MDA und MDSD - Werkzeugen. Einige von ihnen werden in dieser Arbeit vorgestellt und auch verwendet. In diesem Kapitel wird kurz auf die Motivation, das Ziel und den Aufbau dieser Arbeit eingegangen.

1.1 Motivation

In vielen Softwareprojekten werden UML-Klassendiagramme zur Darstellung der Architektur der Zielanwendung verwendet. Nach der Analyse der Geschäftsprozesse wird ein erstes vorläufiges UML-Klassendiagramm modelliert. Auf dieser Basis wird ein erster Prototyp entwickelt. Da es in der frühen Projektphase immer wieder zu Änderungen am Modell kommen kann, muss ebenfalls der Prototyp jedes Mal modifiziert oder gar neu programmiert werden. Dadurch kann es zu Inkonzistenzen zwischen dem implementierten Quellcode und dem Modell kommen. Besonders bei iterativen Softwareprojekten kommt es oft zu Änderungen an der Architektur und somit auch am Prototypen. Es würde Zeit und damit auch Kosten sparen, wenn dieser Prototyp nach jedem Zyklus automatisch generiert werden könnte, ohne das individueller Code verloren geht. Dadurch könnte dem Kunden immer zeitnah ein neuer Prototyp präsentiert werden. Dies ist abhängig von der Verfügbarkeit effizienter Werkzeuge zur Erzeugung der Prototypen.

1.2 Ziel der Arbeit

Ziel dieser Arbeit ist es, Prototypen möglichst direkt aus einem UML-Klassendiagramm zu erzeugen. Auch bei iterativen, evolutionären Projekten entsteht im Hintergrund ein UML-Klassendiagramm. Dieses unterliegt entsprechend den Gesprächsfortschritten über Geschäftsprozesse, Anwendungfällen und Funktionsanforderungen, einer heftigen Weiterentwicklung. Mit dem hier vorgeschlagenen Werkzeug soll es möglich werden, auch auf Basis

sehr vorläufiger UML-Klassendiagramme bereits ausführbare und in Zyklen zu evaluierende Prototypen ohne besonderen Aufwand zu generieren.

Ein Teil der Arbeit wird sein, einige Werkzeuge der modellgetriebenen Softwareentwicklung zu vergleichen. Das Hauptziel dieser Arbeit ist es, einen Prototypen mit Hilfe von MDA-Werkzeugen zu generieren. Dabei können schon vorhandene Cartridges[1] verwendet werden. Zum Erreichen des Zieles sollen ausschließlich OpenSource Komponenten verwendet werden. Hauptaugenmerk wird dabei auf die Generierung der GUI und der Anpassungsschicht fallen.

1.3 Aufbau der Arbeit

Diese Arbeit ist in sieben Kapitel aufgeteilt. Eine kurze Zusammenfassung der jeweiligen Kapitel soll einen Überblick über diese Arbeit geben.

Kapitel 1

Dieses Kapitel gibt einen kurzen Überblick über die Motivation, das Ziel und den Aufbau dieser Arbeit.

Kapitel 2

In diesem Kapitel wird nochmal genauer auf das kurzfristige Ziel dieser Arbeit eingegangen und ein Blick in die Zukunft gewagt.

Kapitel 3

Bevor mit der Realisierung begonnen werden kann, gilt es ein Anforderungsprofil zu erstellen und den Markt nach vorhanden Konzepten und Werkzeugen zu durchsuchen. Im Kapitel 3 werden die durch die Marktanalyse erworbenen Kenntnisse vorgestellt und miteinander verglichen. Zum Schluss erfolgt eine Entscheidung, welche Konzepte und Werkzeuge in dieser Arbeit verwendet werden.

[1] Cartridges sind vorgefertigte Generierungsschablonen

Kapitel 4

Im vierten Kapitel dieser Arbeit wird auf einige Grundlagen eingegangen, die zum Verständnis beitragen sollen. Vorgestellt wird unter anderem die UML mit Ihrem Klassendiagramm und dessen Notationselementen. Des weiteren wird genauer auf das Konzept der MDSD (Model Driven Software Development) eingangen, welches das Kernkonzept dieser Arbeit darstellt. OpenArchtictureWare ist ein MDSD-Werkzeug, welches verwendet und ebenfalls in diesem Kapitel vorgestellt wird. Da als Datenbankschicht das Hibernate-Framework verwendet wird, gib es einen kurzen Einblick in dieses Framework. Außerdem wird noch kurz auf das Swing Komponenten-Set als GUI eingegangen.

Kapitel 5

Dieses Kapitel stellt die fachliche und technische Architektur genauer vor. Des weiteren wird im fünften Kapitel die Spezifikationen des zu generierenden Prototypen festgelegt.

Kapitel 6

Im sechsten Kapitel werden die Transformationsregeln formal beschreiben. Außerdem wird genauer auf die Realisierung der folgenden Teilbereiche eingegangen:

- Erzeugen der Datenbankschicht mit Hilfe eines Cartridges.
- Erzeugen der Anpassungsschicht (DAOs)
- Erzeugen der GUI für den Prototypen mit Swing.

Kapitel 7

Im letzten Kapitel gibt es eine kurze Zusammenfassung dieser Arbeit und einen Ausblick, welche Möglichkeiten, Grenzen und Weiterentwicklungsmöglichkeiten das hier entwickelte Tool hat.

2 Vision

Bei iterativen Projekten gibt es häufig Änderungen an der Architektur und somit auch am Modell und am Prototypen. In den ersten Iterationschritten werden oft nur einfache Datentypen verwendet, wie z.B. Strings. Zum Anfang wird zum Beispiel eine Klasse Person modelliert mit dem Attribut Adresse. Diese Adresse wird erst als String definiert. Später wird wahrscheinlich eine Klasse Adresse mit Attributen wie PLZ, Ort und Strasse modelliert. Auf diesem Weg wird das Modell immer umfangreicher. Es wäre eine große Aufwanderleichterung, wenn nach jedem Iterationschritt ein Prototyp automatisch erzeugt werden könnte.

Dieses Kapitel beschäftigt sich mit den Möglichkeiten von modellgetriebener Softwareentwicklung und der Generierung von Prototypen auf der Basis von Modellen. Der große Vorteil von generierten Protoytpen ist, dass nach jeder Modelländerung zeitnah ein Prototyp automatisch und ohne besonderen Aufwand erstellt werden könnte. Dies spart Zeit und damit auch Kosten. Ein weiterer Vorteil ist, dass die Anwendungen alle gleich aussehen und damit der Wiedererkennungswert erhöht wird. Gleiche Funktionalitäten müssen immer nur ein mal programmiert werden. Auch die Fehlerquote in der Programmierung kann so erheblich gesenkt werden.

Es gibt also einige Vorteile für die Verwendung von MDSD. Die folgenden Unterkapitel stellen die kurzfristigen Ziele dieser Arbeit dar und geben einen kurzen Ausblick in die Zukunft.

2.1 Kurzfristiges Ziel

Mit dieser Arbeit soll gezeigt werden, dass es möglich ist, aus einem UML-Klassendiagramm ohne weiteren Aufwand einen lauffähigen Prototypen zu erzeugen (siehe Abbildung 2.1). Dieser Prototyp soll über eine grafische Benutzeroberfläche und einer Datenbankanbindung verfügen. Es soll mögliche sein, Datensätzen aus einer Datenbanktabelle anzuzeigen, zu editieren, zu löschen und anzulegen.

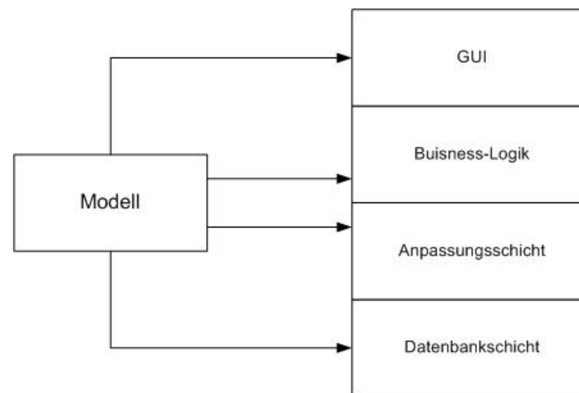

Abbildung 2.1: Die zu erzeugenden Schichten aus einem Modell

2.2 Zukunft

Mit den in dieser Arbeit vorgestellten Techniken ist es durchaus möglich, auch komplexere Anwendungen zu generieren. Zwar muss die Business Logik auch in Zukunft individuell implementiert werden, aber bereits sehr ausgereifte Rumpfanwendungen könnten generiert werden. Plausibilisierungen können abhängig vom Datentyp erstellt werden. Auch die Darstellung in der Benutzeroberfläche wäre dann abhängig von den Datentypen. Z.B. die Anzeige eines Datums würde immer mit einer PopUp-Fenster mit einem Kalender versehen. Dadurch dass für jeden Datentyp eine einheitliche Darstellung und Plausibilisierung generiert wird, erreicht man eine einheitlich Anwendung.

3 Marktanalyse

Wenn es um die Generierung von Programmcode aus Modellen geht, kommt man an MDA oder MDSD nicht vorbei. Deswegen werden beide Konzepte im Folgenden näher betrachtet und von einander abgegrenzt. Außerdem werden noch einige Werkzeuge miteinander verglichen, nachdem ein Anforderungsprofil erstellt wurde. Im Fazit wird sich auf Grundlage des Anforderungsprofils für ein MDA-Werkzeug entschieden.

3.1 Anforderungsprofil

Das MDA-Werkzeug zum Generieren des Programmcodes sollte bestimmte Anforderungen erfüllen, damit es für diese Arbeit in Frage kommt. Da es oft schwierig ist, für Bachelorarbeiten eine Lizenz für eine Software zu erhalten, sollten die Werkzeuge OpenSource Projekte sein. Außerdem wäre es wünschenswert, wenn für das Werkzeug eine Entwicklungumbegung bereitgestellt wird oder dieses als Eclipse Plugin zur Verfügung steht. Ein Eclipse Plugin hat zudem den Vorteil, dass keine zusätzliche Einarbeitungszeit in die Entwicklungsumgebung nötig ist. Es sollten auch bereits vorgefertigte Cartridges zur Verfügung stehen, damit der Prototyp nicht komplett selbst implementiert werden muss. Auch die Zielformate Java und XML sollten unterstützt werden. Als Eingabeformat für den Generierungsprozess sollte ein UML-Klassendiagramm dienen. Zudem ist es wünschenswert, wenn über ausreichend Dokumentation verfügt werden kann.
Da das verwendete Werkzeug in einem iterativen Projekt eingesetzt werden soll, muss bereits bereits individueller Code erhalten bleiben und darf nicht bei jedem Generierungsprozess überschrieben werden. Zudem sollte auch die Wiederholung eines Generierungsprozesses mit nicht all zu großem Aufwand verbunden sein und er sollte beliebig oft wiederholbar sein.

3.2 MDA

Model Driven Architecture (MDA) ist ein formaler Standard der OMG (Object Management Group) zur Entwicklung von Software auf der Basis von Modellen [OMG (2007)]. Ziel von

MDA ist es in der Softwareentwicklung einen hohen Grad der Wiederverwendbarkeit und Wartbarkeit zu erreichen, sowie den Entwicklungprozess vom Modell bis zum fertigen Programm zu formalisieren. Das Hauptaugenmerk wird dabei auf die Modellierung gesetzt und nicht auf die Transformation [Stahl u. a. (2007)].

Mit MDA soll erreicht werden, dass der gesamte Prozess der Softwareentwicklung in Modellen abgebildet wird. Dies beinhaltet die fachliche Architektur, Anwendungsfälle, Geschäftprozesse und Programmablaufpläne. Zudem möchte man erreichen, dass die technischen Aspekte komplett von den inhaltlichen getrennt werden. [Form4 (2007)] Durch Tranformationen in Quellcode wird ein hoher Grad an Konzistenz erreicht, da der Quellcode immer zum Modell passt. In früheren Softwareprojekten war es oft so, dass zum Anfang ein Modell modelliert und später von Entwicklern implementiert wurde. Dabei kann es zu Fehlinterpretationen seitens der Entwickler kommen. Auch wurden oft Änderungen im Quellcode nicht konsequent im Modell nachgepflegt. Aus diesen Gründen kam es immer wieder zu Inkonsistenzen zwischen dem Quellcode und dem Modell. Mit MDA sollen Änderungen der Architektur nur noch am Modell vorgenommen werden.

3.3 MDSD

Model Driven Software Development (MDSD) folgt einem ganz ähnlichen Ziel wie MDA. Allerdings steht bei MDSD die Generierung von Code im Vordergrund. Dieses ist auch ein wesentlicher Unterschied zu MDA. Außerdem ist MDSD unabhängig vom Modell, wogegen MDA mit UML arbeitet. Auf MDSD wird im Kapitel Grundlagen noch näher eingegangen.

Dabei sind die Gründe für die Verwendung von MDSD, die Verbesserung der Softwarequalität, die Weiderverwendbarkeit und die Steigerung der Entwicklungseffizienz. Immer wieder anfallende Arbeiten des Softwareentwicklers sollen mit MDSD automatisiert werden. Als wichtigster Grund wird die höhere Abstraktionsebene, auf der programmiert wird, genannt.
Formale Modelle müssen nicht unbedingt UML-Modelle sein. In einigen Fällen macht es sogar Sinn, eher textuelle Modelle zu verwenden [Stahl u. a. (2007)].

3.3.1 MDA/MDSD-Werkzeuge

Es gibt bereits eine Vielzahl von leistungsfähigen MDA-Werkzeugen, deswegen werden im Folgenden nur die wichtigsten kurz vorstellt.

Folgende Anforderungen werden an ein MDA-Werkzeug zur Codegenerierung gestellt:

- OpenSource

- UML-Modell als Eingabeformat
- Entwicklungsumgebung sollte vorhanden sein
- ausreichende Dokumentation
- es sollten bereits Cartridges vorhanden sein
- Zielformate XML und Java müssen unterstützt werden
- individueller Code muss erhalten bleiben

XDoclet

XDoclet ist eine Open Source Code-Generierungs-Engine, welche attributorientierte Programmierung in Java ermöglicht [XDoclet (2007)]. XDoclet funktioniert, indem man den Java-Code um spezielle JavaDoc Tags erweitert. Anhand dieser im JavaDoc eingegebenen Informationen werden bestimmte Ausgabedateien erstellt. Mögliche Zielformate sind XML, Java oder J2EE Patterns. Dabei sind bereits eine Vielzahl von Modulen im XDoclet integriert. Ein Modul dient als eine Art Generierungs-Schablone [Köhler (2003)]. Dies sind nur einige dieser Module:

- EJB
- Web
- Hibernate
- Spring
- JDO
- JMX

Mit XDoclet wird die Arbeit von Java-Entwicklern erheblich vereinfacht, da lästige Programmierung ähnlicher Artefakte zu einer JavaBean wegfällt. Es muss nur noch die JavaBean Klasse implementiert werden und alle anderen Dateien zu dieser Klasse, wie zum Beispiel eine Hibernate Mapping-Datei können generiert werden.

Leider ist XDoclet für diese Arbeit nicht geeignet, da als Eingabeformat eine Java-Klasse erwartet wird und keine UML-Modelle. XDoclet dient nur zum Generieren ähnlicher Artefakte und nicht zum Generieren von lauffähiger Software.

AndroMDA

AndroMDA ist ein Open-Source Framework zur Codegenerierung auf Basis von UML-Diagrammen. AndroMDA ist komplett in Java implementiert und wird von Sourceforge.net verwaltet. Leider gibt es dafür bis heute noch keine Entwicklungsumgebung und auch ein eclipse Plugin ist noch nicht realisiert. Die Kernkomponente ist ein templatebasierter Generator, welcher aus UML-Diagrammen Programmcode für beliebige Plattformen und Progamiersprachen generieren kann [Schulz (2005)]. Unterstützt wird der Softwareentwickler durch vorgefertigte Cartridges. Diese Cartridges sind jar-Files mit bereits fertigen Templates. Tauscht man ein Cartridge aus, kann aus einem UML-Modell unterschiedlicher Quellcode für unterschiedliche Plattformen generiert werden. Konfiguriert wird das Framework über eine XML-Datei, in welcher auch das zu benutzende Cartridge eingestellt werden kann.

AndroMDA erstellt zwei unterschiedliche Gruppen von Ausgabedateien, zum einen Dateien, die nur generiert werden und bei jeden Generierungsprozess überschrieben werden, und zum anderen Dateien, die für individuelle Programmierung generiert werden. Die zuletzt genannten Dateien sind in der Regel nur ein Rahmenwerk, welche nur einmal erstellt und nicht wieder überschrieben werden.

Die Steuerung der Codegenerierung übernimmt im Regelfall das Build-Tool Maven [Schulz (2005)]. Maven ist ein Tool zum Automatisieren immer wieder anfallender Prozesse in der Softwareentwicklung [Apache (2007)]. Es wurde von der Apache Software Foundation entwickelt und ist ein Open-Source-Projekt.

Anhand von Stereotypen werden im UML-Modell die einzelnen Modellelemente klassifiziert. Diese Klassifikationen legen beim Generierungsprozess fest, welche Art von Code generiert werden soll. Es ist möglich, einem Modellelemente mehrere Stereotypen zuzuweisen, und es können unterschiedliche Cartridges zu gleichen Stereotypen unterschiedlichen Quellcode erzeugen.

Um auch von der verwendeten Datenbank unabhängig zu sein, können die Datenbankabfragen in OCL (Object Constraint Language) formuliert werden [Schulz (2005)]. OCL ist seit der UML-Version 1.1 Bestandteil der UML und dient zum Formulieren von Ausdrücken [Warmer (2005)].

AndroMDA ist ein leistungsstarkes Tool zum Generieren von Anwendungen. Allerdings ist negativ zu erwähnen, dass es noch kein Ecplise Plugin dafür gibt, ist aber bereits angekündigt.

OpenMDX

OpenMDX gilt als führende MDA-Plattform auf Basis von der OMG standardisierten Model Driven Architecture und unterstützt J2SE, J2EE und .NET [OpenMDX (2007)]. OpenMDX steht als Plugin für Eclipse zur Verfügung. Entwickelt wird es von der Schweizer Firma Omex und gewann mit ihrem Produkt OpenMDX den Swiss IT Award 2004. Das modellgetriebene Framework arbeitet mit vielen einzelnen Bausteinen, welche übereinheitliche Schnittstellen miteinander kommunizieren. Omex verspricht mit diesem Framework eine Reduzierung der Entwicklungszeit für Anwendungen von bis zu 75%. Das Besondere an OpenMDX ist, dass kein generativer Übergang vom PIM zum PSM erfolgt.

OpenMDX implementiert die OMG MDA Standards MOF, UML, XMI, JMI und enthält zusätzlich noch weitere Standards, wie zum Beispiel die Java Data Objects (JDO). Das Application Framework OpenMDX bildet eine Abstraktionsschicht zwischen der Anwendungslogik und den darunter liegenden Plattformen wie J2EE, CORBA oder .NET. OpenMDX erlaubt es, Modelle direkt zu interpretieren und auszuführen, ohne dass die Generierung von Code nötig ist. Dadurch dass OpenMDX einen Framework-Ansatz verfolgt, ist es nicht nötig, bei jeder Modelländerung die Anwendung neu zu generieren.

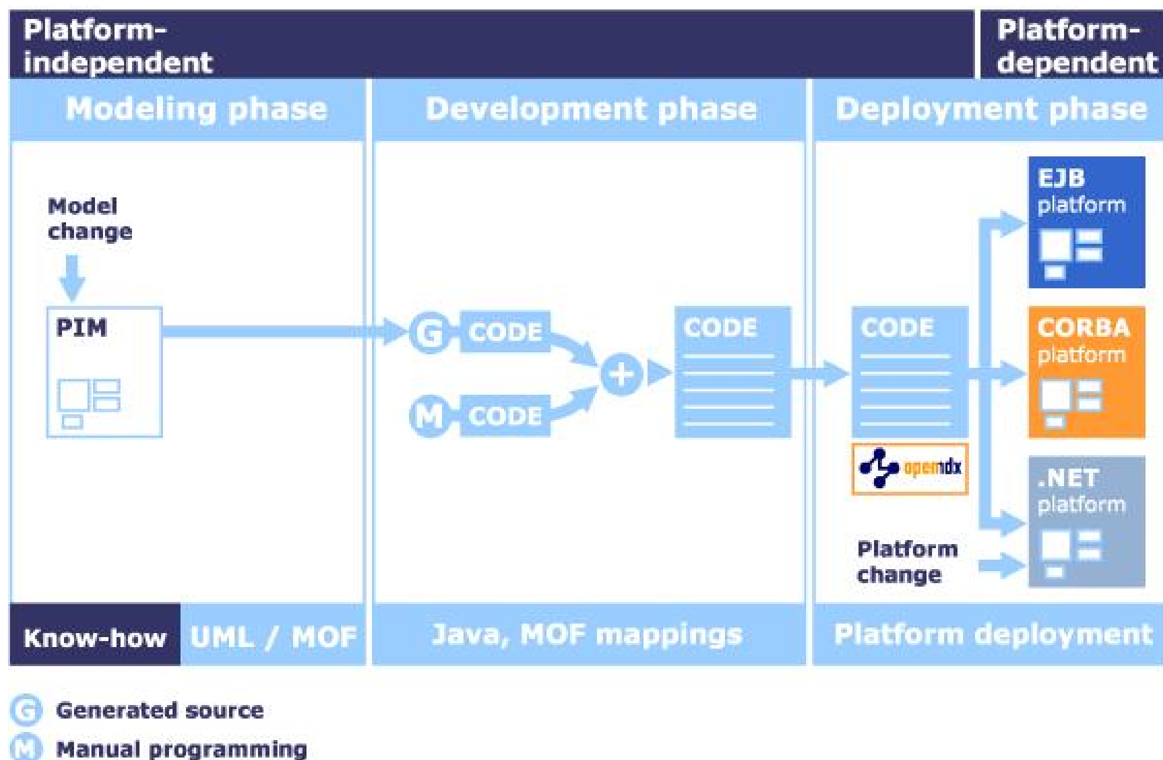

Abbildung 3.1: Anwendungsentwicklung mit openMDX [OpenMDX (2007)]

Der Vorteil von OpenMDX ist, dass der Entwickler die Anwendung nur einmal implementieren muss, auch wenn diese für mehrere Zielplattformen generiert werden soll (siehe Abbildung 3.1). Bei den meisten anderen MDA-Tools muss für jede einzelne Plattform die Anwendung implementiert werden. OpenMDX enthält bereits Plugins für die Persistence und GUI. Beides kann erstellt werden, ohne dass eine Zeile Code generiert wird.

ArcStyler

Auch die Entwickler von ArcStyler bezeichnen ihr Werkzeug als das führende MDA-Entwicklungstool. Entwickelt wird ArcStyler von der Freiburger Firma Interactive Objects. Die Firma ist seit 1993 Mitglied in der OMG und wurde 1991 gegründet. ArcStyler ist eine kommerzielle MDA-Umgebung mit dem integrierten UML-Tool Magic Draw und ist in zwei Versionen erhältlich:

- Enterprise Edition
- Architecture Edition (Ist eine erweiterte Enterprise Edition)

Die Abbildung 3.2 soll die Architektur des ArcStyler näher verdeutlichen. Die UML-Engine ist eine leistungsfähige an MDA orientierte Entwicklungsumgebung. MDA-Enigine ist verantwortlich für die Code-Generierung und die MDA-Cartridges sind die Transformationsregeln für J2EE und .NET. Die Cartridges sind erweiterbar und anpassbar. Das JMI Repository sorgt für den Zugriff auf die UML-Modelle durch das Java Metadata Interface (JMI) und verfügt zusätzlich über eine XMI-Schnittstelle.

ArcStyler ist hat eine eigene Entwicklungsumgebung und kann durch den so genannten Tool Adapter Standard (TAS) beliebig erweitert werden. TAS ist vergleichbar mit den Ecplise Plugins. So ist bereits der UML-Modellierer Magic Draw integriert. Eigen implementierter Code wird durch geschützte Bereiche, die bei einer Neugenerierung nicht überschrieben werden, erhalten. Alle einzelnen Prozesse können anhand von Ant-Tasks gestartet werden.

ArcStyler umfasst mittlerweile 18 Diagrammtypen und verfügt über einen integrierten Modellierer (Magic Draw). Ausserdem ist mit diesem Tool ein Mehrbenutzerbetrieb möglich und es ist eine Versionsverwaltung vorhanden. Zusätzlich werden viele Standards, wie zum Beispiel ANT, JUnit und XMI unterstützt.
Leider ist aber kein Code-Editor integriert und auch keine GUI spezifierbar. Für ArcStyler gibt es leider nur kommerzielle Lizenzen und ist kein OpenSource.

openArchitectureWare

OpenArchitectureWare (oAW) ist ein OpenSource-Projekt, welches den Ansatz von MDSD verfolgt. OAW hat den 3.Preis beim JAX Award 2007 erhalten. OAW ist als Eclipse Plugin

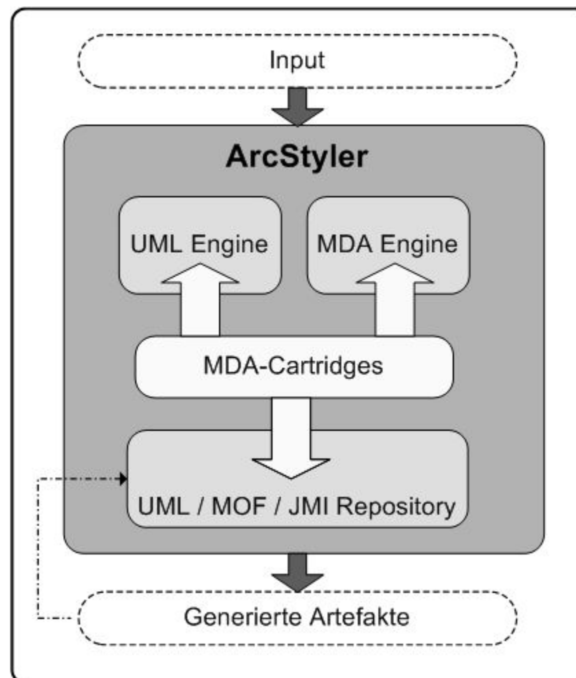

Abbildung 3.2: Architektur vom ArcStyler [Softwarekompetenz (2007)]

erhältlich und muss deshalb nicht über eine eigene Entwicklungsumgebung verfügen. Besonders hilfreich ist dabei die Autovervollständigung. OAW ist templatebasiert. Da oAW über keinen eigenen Modellierer verfügt, wird Magic Draw als UML-Tool empfohlen. MagicDraw verfügt über eine Exportfunktion, die in in das Format exportiert, welches oAW als Eingabeformat dient. Als Eingabeformat wird EMF vom Eclipse Project verwendet. Da es sich um ein Eclipse Plugin handelt, ist die Installation relativ einfach.

Der große Vorteil von oAW ist, dass das Tool modellunabhänging ist. Das heisst, dass nicht nur UML-Modelle als Eingabe dienen können, sondern jede Art von formalen Modellen. Diese müssen nur in das EMF-Format exportiert werden können. Außerdem kann in jedes beliebige Zielformat generiert werden.

Es sind bereits einige vorgefertigte Cartridges verfügbar, zum Beispiel für Hibernate oder das Spring Framework. Diese werden einfach als jar-Files in das oAW Projekt geladen und können mit Hilfe des Worklows aufgerufen werden. Der Workflow ist eine XML-Datei, welche den Ablauf der Generierung steuert. Hier wird festgelegt, welche Templates in welcher Reihenfolge aufgerufen werden.

Die Templates werden in der extra für oAW entwickelten Sprache XPand geschrieben. Diese Sprache dient dazu, komfortabel durch das Eingabemodell navigieren zu können. Besonders

hilfreich ist dabei die Autovervollständigung, welche man auch aus der Programmierung mit Java in Eclipse gewohnt ist (STRG + SPACE).

OAW ist ein Teil des Eclipse Projekts GMT. GMT steht für Generative Modeling Tools. GMT hat sich zum Ziel gesetzt, eine Sammlung von Tools aus dem Bereich MDSD zusammenzustellen [Eclipse (2007)].

Fazit

Zum Bearbeiten dieser Arbeit habe ich mich für das MDA-Tool OpenArchitectureWare und das UML-Tool Magic Draw entschieden. Magic Draw wurde nahe zu von allen MDA-Tools als UML-Modellierer empfohlen oder verwendet.

Der Große Vorteil von oAW ist die Leichte Handhabung. Dadurch dass oAW ein Eclipse Plugin ist, ist die Installation sehr einfach. Man hat schnell sein erstes Projekt eingerichtet und auch schnell den ersten Quellcode generiert. Außerdem wichtig für die Entscheidung war, dass es schon ein Cartridge für Hibernate gibt und auch die Klassenrümpfe bereits automatisch generiert werden können. Es muss also nur noch die GUI und die Anpassungschicht selbst implementiert werden.

3.4 Fazit

Dieses Kapitel hat gezeigt, dass es bereits eine Vielzahl von leistungsfähigen MDA-Tools auf dem Markt vorhanden sind. Allerdings kann keines dieser Tools bereits ein lauffähigen Prototypen mit GUI und Datenbank erzeugen, ohne dass programmiert werden muss. In dieser Arbeit wird mit den MDSD-Tool openArchitectureWare in Verbindung mit dem Hibernate Cartridge und dem UML-Tool Magic Draw versucht, genau so einen Protoypen zu erzeugen.

4 Grundlagen

Um diese Arbeit erfolgreich bearbeiten zu können, sind einige Grundlagen erforderlich. Dazu gehören eine Modellierungssprache wie UML und Konzepte zur modellgetriebenen Softwareentwicklung (MDSD). Außerdem wird auf das in dieser Arbeit verwendete MDSD-Werkzeug openArchitectureWare eingegangen. Des weiteren gibt es eine kurze Einführung in Swing als User-Interface und Hibernate als Datenbankschicht.

4.1 UML

Die UML (Unified Modeling Language) dient als vereinheitlichte Modellierungssprache. Bei der Softwareentwicklung hilft sie Ideen, Modelle, Vorgehensweisen und Lösungen zu veranschaulichen. Dabei unterstützt die UML den Softwareentwickler mit Diagrammen und Notationselementen. Sie ist sprachunabhängig, obwohl sie eher den objektorientierten Ansatz verfolgt.

Diese Arbeit beschäftigt sich mit UML 2.0. Obwohl es erhebliche Änderungen von Version 1.5 auf 2.0 gab, hat sich das Klassendiagramm fast gar nicht verändert. UML umfasst eine Vielzahl von Diagrammen, allerdings befasst sich diese Arbeit ausschließlich mit dem Klassendiagramm, deshalb wird im Folgenden nur auf dieses näher eingegangen.

4.1.1 Klassendiagramm

Das Klassendiagramm ist das Kernstück der UML und ist heute nicht mehr aus der objektorientierten Softwareentwicklung wegzudenken. Hauptsächlich werden statische Bestandteile (Z.B. Klassen und Attribute) des Systems abgebildet sowie in welchen Beziehungen die einzelnen Elemente stehen können. Die einzigen dynamischen Aspekte stellen die Definition der Methoden der Klassen dar. Das Klassendiagramm gehört zur Familie der Strukturdiagramme.

Das Klassendiagramm wird vorwiegend in den ersten Entwicklungszyklen eines Softwareprojektes verwendet. Allerdings kann auch in späteren Zyklen ein Nutzen aus dem Diagramm

gezogen werden. Es dient nicht nur zum Entwurf/Design und zur Vereinfachung der Implementierung, sondern kann auch zur Dokumentation eines Projektes dienen.

4.1.2 Notationselemente

UML-Klassendiagramme bestehen aus einer Vielzahl von sogenannten Notationselementen. Um ein Verständnis über deren Aufgaben zu vermitteln, werden die wesentlichen Notationselemente hier näher betrachtet.

Klasse

Eine Klasse beschreibt eine Gruppe von Objekten mit ähnlichen Eigenschaften, gemeinsamen Verhalten (Methoden) und gemeinsamen Relationen zu anderen Objekten[Hansen (1998)]. In der Abbildung 4.1 ist ein einfache Klasse zu sehen, wie sie in UML dargestellt wird. In dem Beispiel wurde eine Klasse vom Namen Person modelliert, die weder Attribute noch Operationen aufweist.

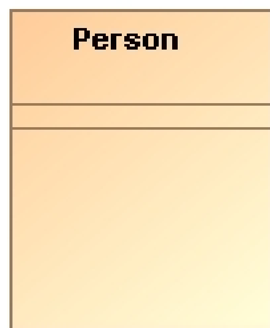

Abbildung 4.1: Darstellung einer Klasse

Attribut

Als Attribute bezeichnet man die statischen Eigenschaften einer Klasse. Die Attribute werden unterhalb vom Klassenamen angezeigt und können verschiedene Bestandteile enthalten, wobei einige optional sind (eckige Klammern).

[Sichtbarkeit] [/] Name [:Typ] [Multiplizität] [=Vorgabewert] [{Eigenschaftswert}]

Die Sichtbarkeit legt die Lese- und Schreibrechte auf die Attribute externer Klassen fest. Folgende Symbole verwendet die UML zu Darstellung der Sichtbarkeit von Attributen:

+ public Dieses Attribut ist für alle Klassen sichtbar

 # protected Ist nur innerhalb der Vererbungshierachie sichtbar

- private Ist nur in der Klasse selbst sichtbar

˜ package Nur Klassen innerhalb des gleichen Package könne auf das Attribut zugreifen

Der Name ist der einzige nicht optionale Bestandteil des Attributs. Im Prinzip gibt es in der UML keine Einschränkung bei der Namensvergabe, allerdings wird empfohlen, sich an die Namenskonventionen der Programmiersprache des Zielsystems zu halten. Hinter dem Namen kann der Typ des Attributs angegeben werden, hier dürfen alle Datentypen des Zielsystems verwendet werden. Zusätzlich können alle im Modell definierten Klassen als Datentyp verwendet werden. Mit der Multiplizität wird die Anzahl der Ausprägungen bezeichnet. Hier können entweder ein fester Wert oder ein Intervall mit Unter- und Obergrenze definiert werden. Außerdem ist es möglich, dem Attribut einen Vorgabewert zu geben. Mit diesem Wert wird das Attribut bei der Instanziierung vorbelegt. Besondere Merkmale könnnen mit dem Eigenschaftswert spezifiziert werden.

In der Regel wird zwischen Instanzattributen und Klassenattributen unterschieden. Dabei sind die Instanzattribute an die aus den Klassen erzeugten Objekten und Klassenattribute an die Klasse gebunden. Klassenattribute existieren damit immer nur einmal im System. In der UML werden die Klassenattribute unterstrichen dargestellt. Siehe Abbildung 4.2.

Abbildung 4.2: Darstellung von Attributen

Operation

Als Operationen werden die Dienstleistungen einer Klasse bezeichnet und bestehen aus folgenden Bestandteilen (optionale Bestandteile in eckigen Klammern):

[Sichtbarkeit] Name ([Parameter-Liste]) [:Rückgabetyp] [{Eigenschaft}]

Die Parameter-Liste setzt sich wie folgt zusammen, dabei werden die einzelnen Parameter jeweils durch ein Komma getrennt:

[Übergabemodus] Name :Typ [Multiplizität] [=Vorgabewert] [{Eigenschaft}]

Operationen werden von dem Generator erkannt, und es werden Methodenrümpfe mit Kommentaren erzeugt. Das Ausprogrammieren muss allerdings weiterhin manuell erfolgen, deswegen haben die modellierten Methoden im Prototypen eher einen informellen Zweck.

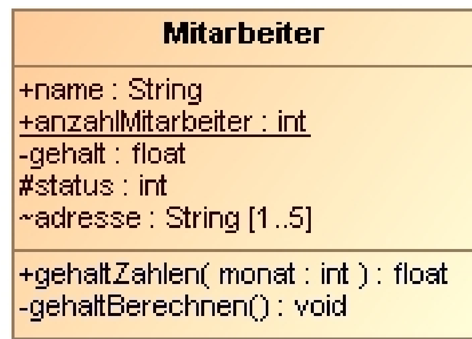

Abbildung 4.3: Darstellung von Operationen

Assoziation

Mit einer Assoziation wird eine semantische Beziehung zwischen zwei Klassen dargestellt. Eine Assoziation kann aus mehreren Bestandteilen bestehen. Als erstes ist der Assoziationsname zu nennen. Zudem Namen kann zum Steigern des Verständnis eine Leserichtung mit angegeben werden. Zusätzlich sollten die Multiplizitäten mit angegeben werden. Oft wird hier auch von Kardinalitäten gesprochen. Außerdem können noch Rollen, Eigenschaften, Navigierbarkeit und Einschränkungen dargestellt werden.

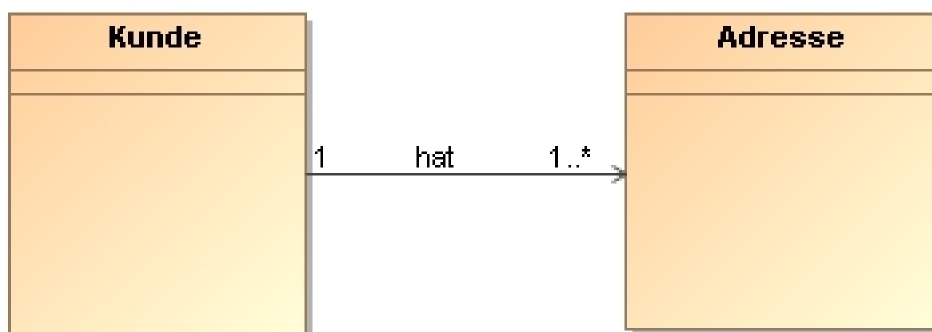

Abbildung 4.4: binäre Assoziation

In der UML können mehrere Arten von Assoziationen dargestellt werden. Zum einen die binäre Assoziation, wo zwei Klassen beteiligt sind (Abbildung 4.4). Dann die reflexive Assoziation, in der eine Klasse eine Beziehung zu sich selbst hat. Eine weitere Assoziation ist die N-äre. Hierbei stehen gleich mehrere Klassen in einer Beziehung zu einander. N-äre Assoziationen werden allerdings nicht von Java unterstützt und werden in Regel in eine eigene Klasse umgewandelt. Außerdem gibt es die qualifizierte Assoziation, bei der ein so genannter Qualifizierer eine Menge von Attributen definiert, die Objekte der zweiten Klasse eindeutig referenziert.

Aggregation

Die Aggregation stellt eine besondere Form der binären Assoziation dar. Sie beschreibt ein schwache Beziehung zwischen Objekten. Konkret bedeutet dies, dass jede Klasse zwar ein Teil eines Ganzen ist, allerdings die Teile auch ohne das Ganze existieren kann.

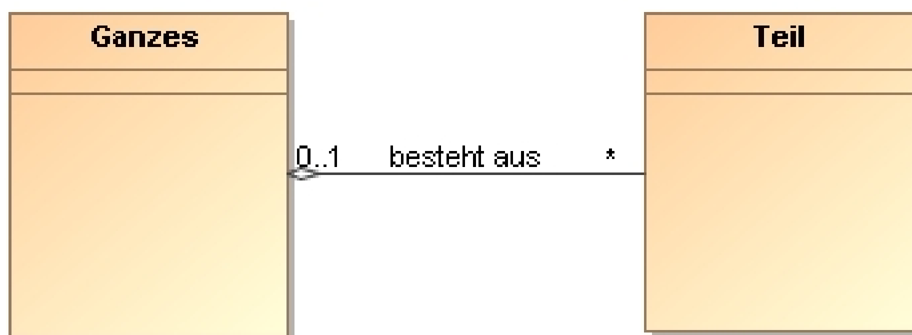

Abbildung 4.5: Aggregation

Komposition

Die Komposition ist eine Verschärfung der Assoziation, denn hier wird die Verbindung zwischen den Teilen und dem Ganzen als untrennbar definiert. Das bedeutet, dass die einzelnen Teile existenznotwendig für das Ganze sind.

Abhängigkeit

Mit dieser Art der Beziehung wird in der UML eine semantische Abhängigkeit modelliert. Man kann die Abhängigkeit auch als **Client-Supplier-Beziehung** bezeichnen, wobei der Supplier

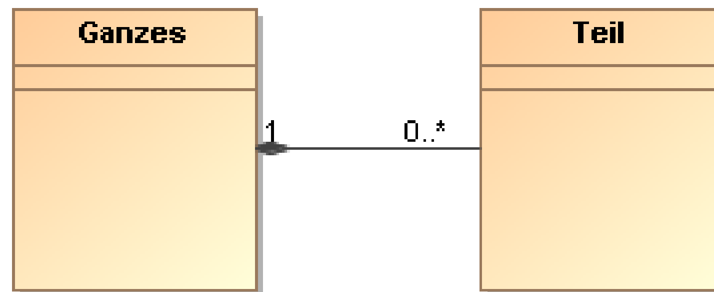

Abbildung 4.6: Komposition

semantisch nicht ohne dem Client existieren kann. Diese Beziehung dient ausschließlich der Dokumentation.

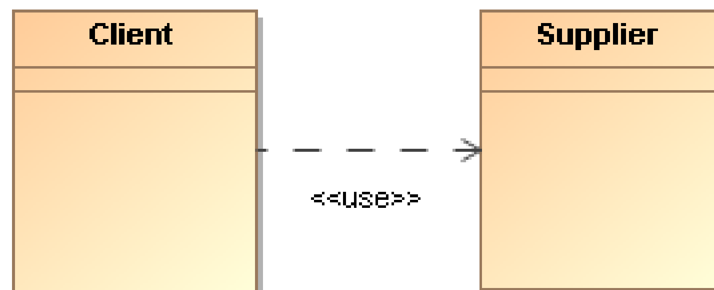

Abbildung 4.7: Abhängigkeit

Generalisierung/Spezialisierung

Mit der UML ist es auch möglich Generalisierungen und Spezialisierungen zu modellieren. Wobei in Pfeilrichtung generalisiert wird. In Richtung der Oberklasse wird generalisiert und in Richtung der Unterklasse spezialisiert. Auf das Konzept der Generalisierung wird an dieser Stelle nicht weiter eingegangen.

Stereotypen

Die Stereotypen definieren den Zweck und die Rolle eines Notationselements im Modell und sind in der MDSD ein fundamentales Konzept. Anhand von Stereotypen kann festgelegt werden, was für Programmcode aus dem Notationselement später generiert wird. Zum Beispiel,

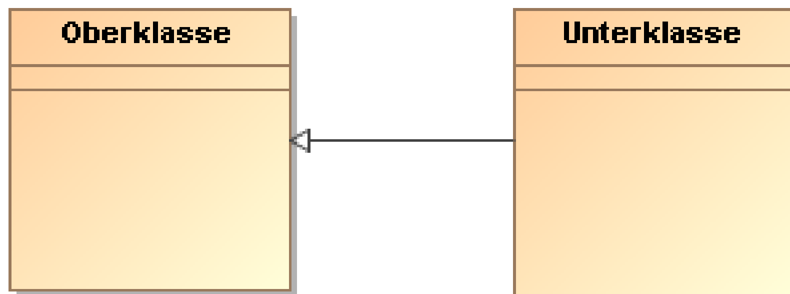

Abbildung 4.8: Generalisierung

ob es sich bei einer Klasse um eine Entität handelt und diese persistiert werden soll, dann werden für diese Klasse DAOs und Datenbanktabellen erzeugt, wobei bei Klassen ohne Stereotyp ggf. nur Rümpfe generiert werden.

Abstrakte Klasse

In der UML werden abstrakte Klassen dadurch gekennzeichnet, dass der Klassenname kursiv angezeigt wird oder durch Angabe des Schlüsselwortes *{abstract}*. Von der kursiven Schreibweise ist allerdings abzuraten, da diese im Modell leicht übersehen wird.

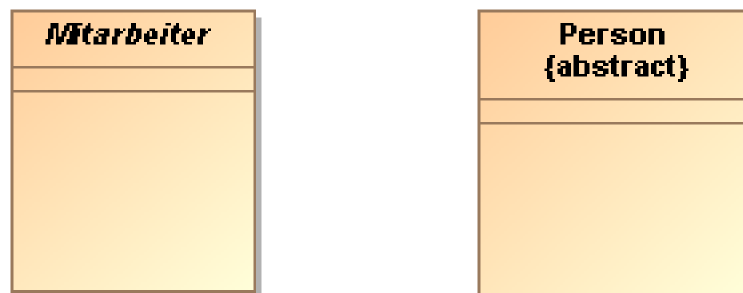

Abbildung 4.9: Abstrakte Klasse

Schnittstelle

Schnittstellen oder auch Interface werden durch den Stereotypen *interface* modelliert.

Anmerkung

Mit Anmerkungen ist es möglich, Kommentare zu den Notationselementen anzugeben. Diese werden durch ein einfaches Kästchen, wie in der Abbildung 4.10 zu sehen, modelliert.

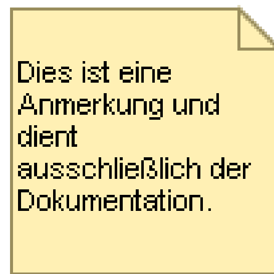

Abbildung 4.10: Anmerkung

4.2 MDSD

[Stahl u. a. (2007)] definiert MDSD wie folgt:

Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen.

Wie aus der Definition schon hervor geht, soll aus formalen Modellen Programmcode generiert werden, aber was sind formale Modelle? Modelle können in Projekten in vielen Arten und Formen auftreten, z.B. in UML. Allerdings ist nicht jedes Modell automatisch auch ein formales. Wenn aus einem Modell Code generiert werden soll, dann sind an das Modell bestimmte Forderungen zu stellen, dieses muss formal definiert sein, im Gegensatz zu informell. Formale Modelle können in unterschiedlichen Formen definiert werden, z.B. auch in einer textuellen Form.

4.2.1 Domäne

Als Domäne wird ein begrenztes Interessen- oder Wissensgebiet bezeichnet [Stahl u. a. (2007)]. Dabei ist es von großer Bedeutung, die einzelnen Domänen klar von einander zu trennen. Dies wird getan, damit die Komplexität einer einzelnen Domäne nicht zu umfangreich wird.

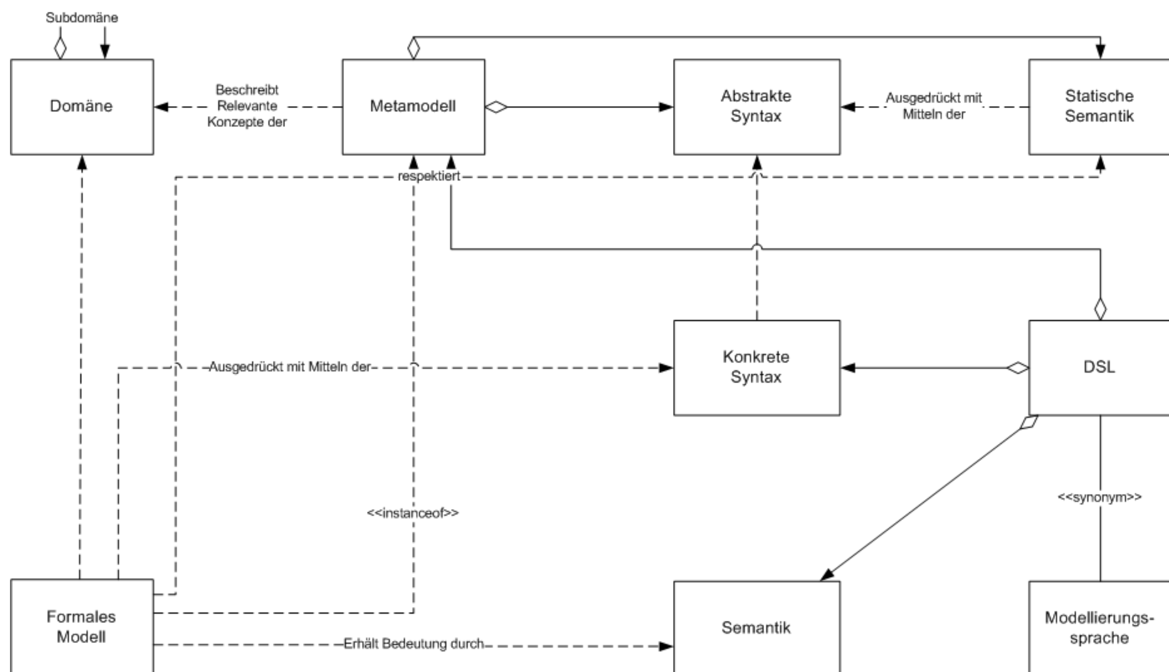

Abbildung 4.11: Begriffbildung [Stahl u. a. (2007) S. 28]

Es wird zwischen fachlichen und technischen Domänen unterschieden. Zu den technischen Domänen gehören Klassendiagramme und Datenbankmodelle. Solche Domänen werden auch *achitekturzentriert* genannt [Stahl u. a. (2007) S.28]. In einer fachlichen Domäne wird nicht die Architektur beschrieben, sondern die fachlichen Aspekte, wie z.B. werden Preise in einer Bestellposition ermittelt, und welche Rabatte fließen in den Preis mit ein.
Zur Vereinfachung der Lesbarkeit können Domänen in Subdomänen aufgeteilt werden. Somit kann die Komplexität der einzelnen Domänen begrenzt werden.

4.2.2 Metamodell

Um die Inhalte einer Domäne formal zu beschreiben, sind so genannte Metamodelle erforderlich. In einem Metamodell wird die genaue Struktur einer Domäne beschrieben. Metamodelle müssen nicht unbedingt ein UML-Modell sein. Mittlerweile ist es zunehmend attraktiv geworden, Metamodelle unabhänging von UML zu definieren [Stahl u. a. (2007)]. Um diese Struktur zu beschreiben, umfasst ein Metamodell die abstrakte Syntax sowie die statische Semantik.

Abstrakte und konkrete Syntax

Als abstrakte Syntax werden die Metamodellelemente und ihre Beziehungen untereinander bezeichnet [Stahl u. a. (2007)]. Die konkrete Syntax beschreibt dagegen den tatsächlichen Quellcode bzw. Diagramme.
Mit der abstrakten Sytax wird z.B. beschrieben, dass die Sprache Java aus Klassen, Attributen und Methoden besteht und das Klassen von einander erben können. Die konkrete Syntax legt fest, dass eine Klasse in Java mit dem Schlüssenwort *class* definiert wird. Damit könnte Java und ein UML-Klassendiagramm die gleiche abstrakte Syntax haben, aber eine komplett unterschiedliche konkrete Syntax. Daraus kann gefolgert werden, dass eine abstrakte Syntax unterschiedliche konkrete Syntaxen definieren kann. Eine konkrete Syntax kann also sowohl grafisch als auch textuell sein.

Statische Semantik

Mit der statischen Semantik werden Bedingungen festgelegt, welche ein Metamodell erfüllen muss, damit man es als *wohlgeformt* bezeichnen kann. Diese Bedingungen werden auch als sogenannte **Constraints** bezeichnet. Diese sind abhängig von der abstrakten Syntax.
Z.B. besitzen statisch getypte Programmiersprachen eine statische Semantik. Die Bedingungen dieser Semantik werden durch den Compiler geprüft und evtl. ein Fehler ausgegeben. Syntaxfehler gehören nicht dazu, diese werden in der Regel bereits vom Parser gemeldet. Ein konkretes Beispiel für eine solche Bedingung ist, dass eine Variable in Java erst definiert werden muss, bevor sie verwendet werden darf.
Aber warum ist das so wichtig für MDSD? Weil in der statischen Semantik bereits Fehler in der Modellierung erkannt werden sollen und nicht erst im Generierungsprozess. Fehler sollten stets so früh wie möglich erkannt und abgefangen werden. Dadurch kann erheblich Zeit und Aufwand gespart werden.

Domänenspezifische Sprache

Eine Domänenspezifische Sprache (Domain Specific Language, DSL) besteht aus einem Metamodell mit dazugehöriger abstrakter Syntax, Constrains sowie der konkreten Syntax und wird auch als die Programmiersprache für eine Domäne bezeichnet [Stahl u. a. (2007)].

4.3 openArchitectureWare

OpenArchitectureWare (oAW) ist ein Open-Source Framework zur modellgetriebenen Softwareentwicklung, welches in Java implementiert ist und als Eclipse Plugin erhältlich ist. oAW unterstützt den Softwareentwickler beim Parsen eines Modelles, um daraus Programmcode generieren zu können.
In diesem Abschnitt wird nur kurz auf die in dieser Arbeit verwendeten Komponenten eingegangen, um einen kleinen Überblick zu gewinnen. Im Kapitel zur Realisierung wird auf die einzelnen Komponenten nochmal näher eingegangen.

Workflow

Der Workflow ist eine XML-Datei, in welcher gesteuert wird, welche Templates mit welchem Modell, in welcher Reihenfolge aufgerufen werden soll. Wird ein Cartridge verwendet, kann dieses direkt aus dem Workflow aufgerufen werden. Das Cartridge steuert danach selbständig, welche Templates aufgerufen werden sollen. Im Workflow können Properties-File geladen werden. Zusätzlich können noch Beautifier für die generierten Artefakte aufgerufen werden. Die Beautifier sind dafür da, den generierten Code auszurichten.

4.3.1 Template

OAW ist wie bereits erwähnt templatebasiert. Das bedeutet, die Generierung der Artefakte wird über Templates realisiert. In den Templates wird der zu generierende Code implementiert. Anhand von Regeln und Bedingungen wird bestimmt, wie genau der Zielcode später aussehen soll. Zum Formulieren solcher Bedingungen und Regeln wurde die Sprache XPand entwickelt.

XPand

XPand ist eine statisch getypte Templatesprache, die eigens für oAW entwickelt wurde [Efftinge (2007)]. Gespeichert werden diese Templates mit der Endung *.xpt*. Alle Zeichen aus dem Zeichensatz ASCII können für den zu generienden Code verwendet werden, außer die französischen Anführungszeichen. Diese Zeichen werden bereits von XPand benutzt, um die Schlüsselwörter zu kennzeichnen.

Mit XPand ist es möglich, durch das Eingabemodell komfortabel zu navigieren. Die einzelnen Templates können polymorph aufgerufen werden. Als Parameter werden in der Regel

Elemente des Modell verwendet, wie zum Beispiel das Modell an sich, Klasse aus dem Modell, Eigenschaften aus einer Klasse oder alle Klassen mit einem bestimmten Stereotypen. Es kann sehr einfach über alle Klassen iteriert werden. Es ist auch sehr komfortabel, auf die einzelnen Eigenschaften der Elemente zuzugreifen.

Mit Expand bekommt man Zugriff zu den Metaklassen des Metamodells. Zum Beispiel gibt es in der UML eine Metaklasse *class*. Diese verfügt über eine Vielzahl von Methoden, die mit EXpand ausgeführt werden können. Über solche Methode ist es leicht an einen Namen der Klasse oder an alle Attribute einer Klasse zu kommen. Die Attribute wären dann eine Liste von Attribut-Klassen. Die Autovervollständigung ermöglicht es, sich leicht durch die vielen Klassen und Methoden zu recht zu finden.

4.4 Hibernate

Hibernate ist ein plattformübergreifendes Open-Source-Persistenz-Framework von Red Hat für Java. Ein großer Vorteil beim Verwenden von Hibernate ist, dass die Anwendung unabhängig von der Datenbank entwickelt werden kann. Es ist also möglich, die Datenbank zu ändern, ohne die Schichten über Hibernate anzupassen. Hibernate unterstützt mittlerweile mehr als 20 Datenbanksystemen, darunter Oracle, DB2, Sybase, SAP DB, MySQL, Microsoft SQL und HSQL. Hibernate hat außerdem die Eigenschaft, die Tabellen automatisch anzulegen, wenn noch keine vorhanden sind.

Object-Relational-Mapping

Der Inhalt von Objekten kann in einer relationalen Datenbank gespeichert werden. Andersherum können aus Sätzen einer Datenbanktabelle Objekte erzeugt werde. Dabei kann die Struktur der Objekte beliebig kompliziert sein. Dies wird auch als Object-Relational-Mapping (ORM) bezeichnet. ORM macht das Programmieren sehr komfortabel, da Objekte durch das automatische Mappen einfach gespeichert und gelesen werden können, ohne dass die einzelnen Attribute manuell abgebildet werden müssen.

Die Abbildung 4.12 zeigt, dass zwischen der Objektorientierten Anwendung und der Datenbank ein Mapping stattfinden muss, um die Inhalte der Objekte in einer relationalen Datenbank abbilden zu können. In der Regel muss so ein Mapping aufwendig programmiert werden. Hibernate nimmt dem Anwender diese Arbeit ab.

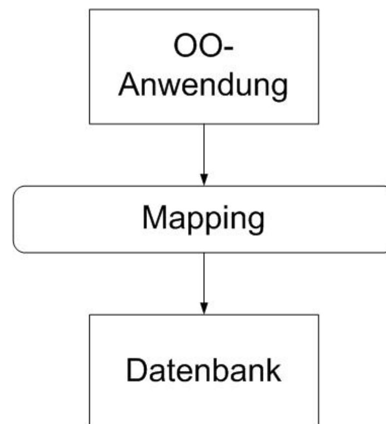

Abbildung 4.12: Object-Relational-Mapping

4.4.1 Hibernate API

Die Hibernate-API ist komplett in Java implementiert und lässt sich einfach durch ein jar-File in ein Projekt einbinden. Die API verfügt über eine Vielzahl von Schnittstellen, von denen die wichtigsten in den folgenden Unterkapiteln näher vorgestellt werden. In der Abbildung 4.13 soll verdeutlicht werden, wie diese Schnittstellen miteinander zusammenhängen. Die Configuration-Komponente lädt die Konfigurationsdateien und Mapping-Files ein. Die SessionFactory benutzt die Configuration und erstellt die Sessions, welche wiederum Transactions und Querys erzeugen. Diese Transactions und Querys werden in der Anwendung (Application Code) verwendet.

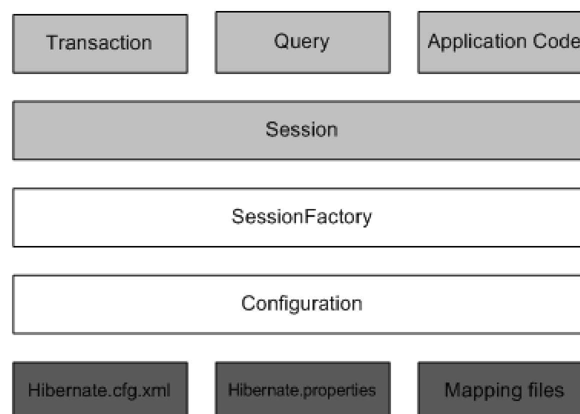

Abbildung 4.13: Hibernate Komponenten aus Peak und Heudecker (2006) Seite 52

Configuration

Zur Konfiguration von Hibernate werden in der Regel Konfigurationsdateien auf der Basis von XML verwendet. Dabei spielen zwei Konfigurationsdateien eine wesentliche Rolle.
Die *hibernate.cfg.xml* ist für die zentralen Einstellungen der Hibernate-Umgebung erforderlich. Hier werden die Datenbanktreiber, Dialekt der Datenbank und sonstige Parameter konfiguriert. Zusätzlich gibt es noch die Mapping-Dateien. In diesen werden die Datenbanktabellen konfiguriert. Hier wird die Verbindung zur JavaBean hergestellt, Tabellennamen vergeben und die einzelnen Attribute definiert, die persistiert werden sollen. Anhand dieser Mapping-Dateien können die JavaBeans von Hibernate versorgt werden.
In der aktuellen Version von Hibernate kann auf die Mapping-Files verzichtet werden, indem Annotations benutzt werden.
Die Konfiguration kann über die Schnittstelle *Configuration* in die Anwendung geladen werden.

Session

Die *Session* ist die primäre Schnittstelle in der Hibernate API. Eine Instanz von *Session* ist leichtgewichtig und kann sehr leicht erzeugt und wieder zerstört werden. Dies ist von großer Bedeutung, da ständig in der Anwendung Instanzen von *Session* erzeugt und zerstört werden müssen. Es wird empfohlen, dass immer nur ein Thread zur Zeit eine *Session* benutzen soll. Man kann die *Session* auch als den Persistence-Manager bezeichnen.
Um eine *Session* zu instanzieren, benötigt man die *SessionFactory*. Diese ist keineswegs als leichtgewichtig zu bezeichnen und ist deswegen typischerweise auch nur einmal pro Anwendung vorhanden. Arbeitet die Anwendung allerdings auf mehreren Datenbanken, ist für jede einzelne Datenbank eine eigene *SessionFactory* nötig. Die Session wird benutzt, um komfortabel Objekte aus der Datenbank zu lesen, diese wieder zu speichern und zu löschen.

Transaction

Transaction werden von der Instanz einer Session erzeugt. Sie dienen dazu, innerhalb von SQL-Befehlen eine Transaktion zu beginnen, diese zu commiten und wieder abzuschließen. Man erreicht damit, dass für Datenbankoperatioen die so genannten ACID-Eigenschaften eingehalten werden können. ACID steht für Atomarität, Konsistenz (consistency), Isoliertheit und Dauerhaft. Auf das Konzept der Transaction wird an dieser Stelle nicht weiter eingegangen.

Query

Auch Querys werden von der Instanz einer Session erzeugt. Eine Query-Instanz dient dazu, individuelle SQL-Abfragen abzusetzen. Für das Absetzen solcher SQL-Abfragen wurde eine spezielle Form des SQLs entwickelt, mit der es möglich ist, Objekte aus einer Datenbanktabelle zu laden. Man gibt nicht mehr die einzelnen Felder als Resultset an, sondern ein Objekt. Hibernate mappt dann automatisch die einzelnen Felder.

4.4.2 Hibernate Cartridge

Um die Hibernate Konfgurationsdateien, JavaBeans und Mapping-Dateien zu erzeugen, wird das Cartridge von Fornax (Fornax (2007)) verwendet. Die zu genierenden Klassen müssen im UML-Klassendiagramm mit dem Stereotyp *Entity* gekennzeichnet werden. Wie genau der Generierungsprozess funktioniert, wird im Kapitel Realisierung näher erläutert.

4.5 Swing

Für die grafische Oberfläche (GUI - graphical user interface) wird Swing verwendet. Swing ist eine Java Klassen-Bibliothek zum Gestalten von grafischen Oberflächen. Mit dieser Bibliothek wird eine umfassende Auswahl von Komponenten zur Verfügung gestellt, deswegen wird auch häufig vom Swing-Komponenten-Set gesprochen.

5 Architektur

Um mit der Realisierung beginnen zu können, muss zu Anfang die fachliche und technische Architektur modelliert werden. Die folgenden beiden Unterkapitel gehen auf diese Architekturen ein. Außerdem werden noch die Spezifikationen für den Generierungsprozess und den Prototypen festgelegt.

5.1 Fachliche Architektur

5.1.1 Fachliche Architektur des Werkzeugs

Im fachlichen Sinne geht es in dieser Arbeit darum, aus einem UML-Modell, mit Hilfe von Generatoren, einen lauffähigen Prototypen zu erzeugen. Wie die Architektur dazu aussehen soll, wird das folgende Schaubild (5.1) näher verdeutlichen.

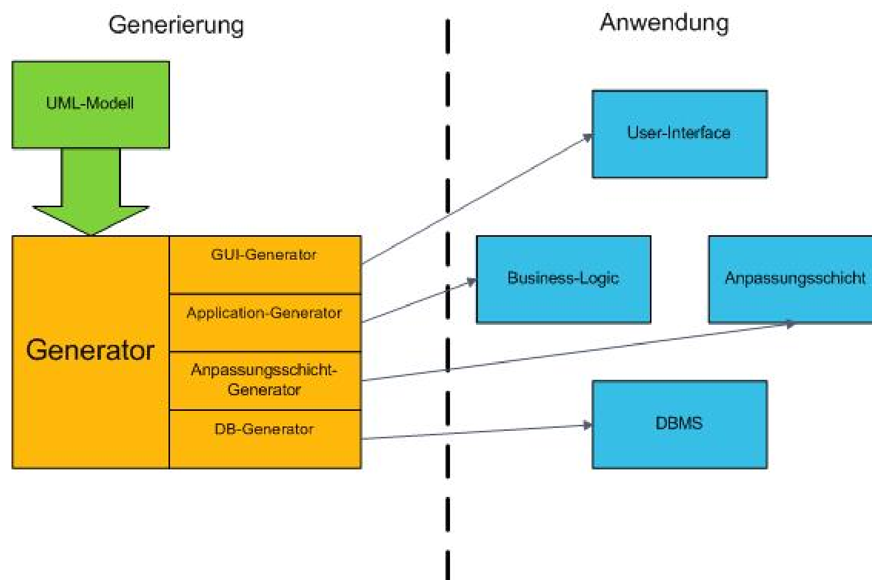

Abbildung 5.1: Fachliche Architektur

Das UML-Modell gilt als Eingabe für den Generator. Der Generator besteht im wesentlichen aus vier Teilen:

- GUI-Generator zum Generieren des User-Interfaces
- Dem Application-Generator für die Business Logic (Klassenrümpfe)
- Dem Generator für die Anpassungsschicht
- Und einen Generator für die Datenbankschicht

5.1.2 Fachliche Architektur der Anwendung

Hier wird kurz auf die fachliche Architektur des in dieser Arbeit zu erstellenden Prototypen vorgestellt. In der Abbildung 5.2 ist die Beispielanwendung zu sehen. Es soll sich um eine sehr einfache Bestellerfassung handeln. Sie besteht aus vier Klassen. Ein **Kunde** kann **Bestellungen** aufgeben. Eine **Bestellung** kann über mehrere **Bestellpositionen** verfügen. Diese **Bestellpositionen** beinhalten einen **Artikel**.

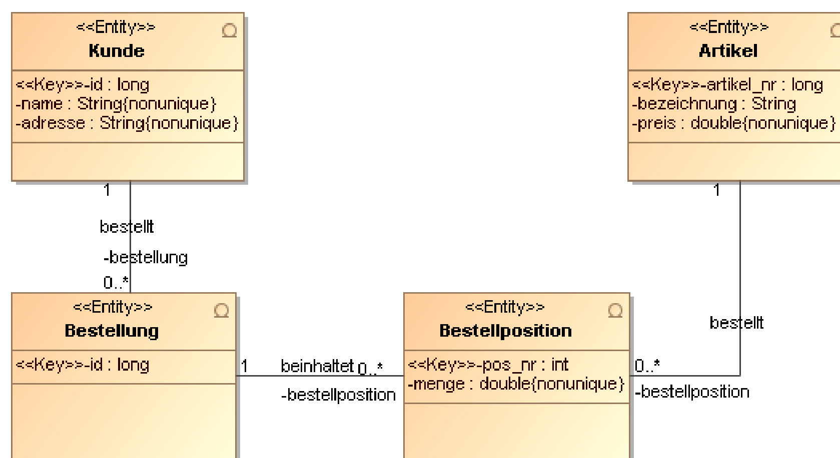

Abbildung 5.2: Fachliche Architektur der Beispielanwendung

5.2 Technische Architektur

5.2.1 Technische Architektur des Werkzeugs

Die technische Architektur ist bei weitem komplexer als die fachliche. Besonders der Generator besteht aus vielen Komponenten. Die folgende Abbildung 5.3 soll das Verständnis für die Beziehungen zwischen den einzelnen Komponenten verbessern.

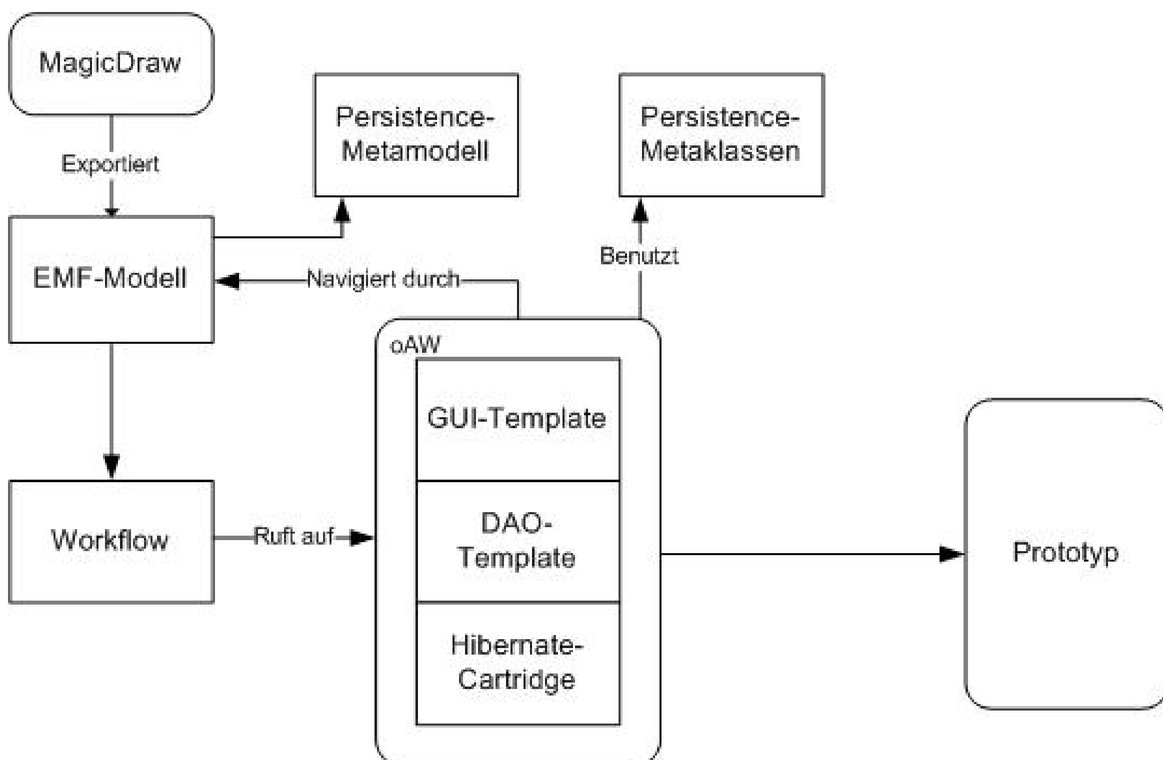

Abbildung 5.3: Technische Architektur

Als UML-Tool wird die Community Edition von Magic Draw (Version 12.5) verwendet [MagicDraw (2007)]. Der Vorteil dieses Tools liegt darin, dass es möglich ist, die Modelle in ein EMF-Modell zu exportieren, welches als Eingabeformat für das Hibernate-Cartridge und oAW dient. Um die Stereotypen der Hibernate Cartridge nutzen zu können, muss das Persistence-Metamodell Magic-Draw bekannt gemacht werden. Das exportierte Modell wird in den Workflow geladen, welcher die Templates mit dem EMF-Modell aufruft. Die Templates können mit der XPand-Sprache durch das Modell navigieren. Dazu werden die Metaklassen von dem Persistence-Metamodell verwendet, welches mit dem Hibernate Cartridge mitgeliefert wird. Die Templates generieren dann den Source-Code für den Prototypen.

Als Entwicklungsumgebung wird eclipse 3.2 mit dem openArchitectWare plugin benutzt. Die Entwicklung kann an einem einzelnen Rechner durchgeführt werden.

5.2.2 Technische Architektur der Anwendung

Der Prototyp soll komplett in Java erstellt werden. Die Datenbank wird eine HSQL-Datenbank sein, welche mit dem Framework Hibernate benutzt wird. Für die Umsetzung des User-Interface wird das Komponenentenset von Swing verwendet. In der hier erstellten Version ist der Prototyp eine Einzelplatzanwendung.

5.3 Spezifikationen

Natürlich werden bestimmte Anforderung an den Generierungsprozess und den generierten Prototypen gestellt. Diese werden in diesem Kapitel näher vorgestellt.

Grundanforderung an diese Arbeit ist es, aus einem möglichst vorläufigen UML-Klassendiagramm bereits einen lauffähigen Prototypen erzeugen zu können. Dieser Generierungprozess muss dabei beliebig oft wiederholbar sein, ohne dass bereits individuell implementierter Programmcode überschrieben wird. Ist das Modell noch unvollständig, sollen fehlende Werte durch Default-Werte ersetzt werden.

Auch der erzeugte Prototyp sollte einigen Anforderungen Genüge leisten. Mit dem Prototypen soll es möglich sein, benutzerfreundlich Datensätze zu editieren, hinzuzufügen und wieder zu löschen. Dabei soll dieser intuitiv bedienbar sein. Die Pflege dieser Daten soll über eine Grafische Benutzeroberfläche erfolgen. Assoziationen sollten abgebildet werden. Die Daten müssen dauerhaft in einer Datenbank gespeichert werden können.

6 Realisierung

Dieses Kapitel befasst sich mit der Realisierung der Generierung eines Prototypen aus einem UML-KLassendiagramm.

6.1 Installation von oAW

Da openArchitectureWare ein Eclpise Plugin ist, erweist sich die Installation als relativ einfach. OpenArchitectureWare kann bequem über die Update-Funktion von Eclipse installiert werden. Eine detaillierte Anleitung befindet sich auf der Hompage *www.openarchitectureware.com*. Außerdem ist dort ebenfalls ein Tutorial vorhanden, in welchen beschrieben wird, wie ein erstes Projekt angelegt werden kann.

6.2 Transformationsregeln

Um Code aus einem Modell erzeugen zu können, müssen Transformationsregeln festgelegt werden. Im Folgenden werden für jede einzelne Schicht die jeweiligen Tranformationsregeln festgelegt und beschrieben.

6.2.1 Datenbankschicht

Da die Datenbankschicht von einem bereits vorhandenen Cartridge erzeugt wird, müssen hier keine weiteren Transformationsregeln festgelegt werden. Trotzdem werden einige Transformationsregeln hier näher erläutert.

Entity

Aus jeder modellierten Entity werden vom Hibernate Cartridge zwei Klassen und ein Interface generiert. Ein abstrakte Klasse und eine Implementierungsklasse, welche von der abstakten Klasse erbt. Die Implementierungsklasse wird nur beim ersten Generierungszyklus erstellt und von folgenden Zyklen unberührt gelassen. Dies ist solange problemlos, bis sich Methodennamen ändern. Erst wenn dieser Fall eintritt, müssen auch die Implementierungsklassen angepasst werden. Die abstrakte Klasse implementiert das generierte Interface. Die folgende Abbildung 6.1 zeigt, welche Artefakte aus einer modellierten Entity mit dem Namen *MyEntity* und einen Attribute *Name* erzeugt werden.

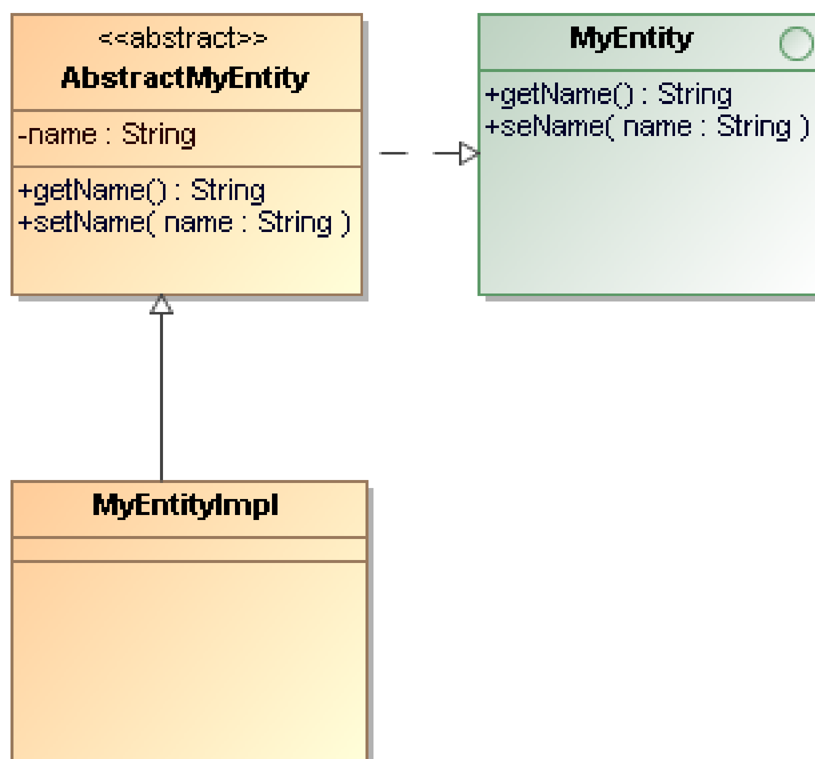

Abbildung 6.1: Erzeugte Klassen aus einer Entity

Außerdem werden noch die für das Hibernate-Framework benötigten Mapping-Files generiert. In diesem Beispiel würde eine Datei mit dem Namen *MyEntityImpl.hbm.xml* erzeugt.

Attribut

Wie schon in der Abbildung 6.1 zu erkennen, werden für jedes modellierte Attribut in der abstrakten Klasse Instanzvariablen sowie Getter- und Setter-Methoden generiert. Das Interface erhält ebenfalls für diese Attribute Getter- und Setter-Methoden. Die Attribute werden ebenfalls in den Mapping-Files als column definiert.

Assoziation

Bei den Assoziation wird zwischen drei unterschiedlichen Assoziationen unterschieden, und zwar die One-to-One, One-to-Many oder Many-to-Many. Bei einer One-to-One Assoziation wird das Key-Attribut der assoziierten Klasse jeweils als Instanzvariable definiert. Liegt eine One-to-Many Assoziation vor, wird an dem einen Assosiationsende ein Set definiert. Bei der Many-to-Many Assoziation wird ein beiden Enden jeweils ein Set implementiert.
In die Hibernate Mapping-File werden die jeweiligen Assoziationen eingetragen.

6.2.2 Anpassungschicht

Für die Anpassungsschicht werden die DAOs (Data Access Objects) generiert, welche die Verbindung von dem User-Interface zur Datenbankschicht herstellen.

Entity

Für jede einzelne Entity wird eine eigene DAO generiert. In jeder DAO sind Methoden definiert zum Löschen, Lesen, Hinzufügen und Ändern von Datensätzen.

Jede DAO soll über folgende Methoden verfügen:

- loadById(key) - Lädt genau einen Datensatz mit Primary Key
- listAll() - Lädt alle Sätze aus Tabelle
- delete(key) - Löscht Satz mit Primary Key
- save(entity) - Fügt Datensatz hinzu oder ändert diesen
- close() - Schließt die aktuelle Session

Attribut

Die Attribute werden in der Anpassungsschicht nicht speziell behandelt, da durch das Hibernate Framework die Objekte immer komplett geladen werden und nicht einzelne Attribute. Ausschließlich der Typ der Keys muss berücksichtigt werden, damit auf einen bestimmten direkt zugegriffen werden kann.

Assoziation

Die Assoziationen müssen in der Anpassungschicht nicht näher berücksichtigt werden, da Hibernate automatisch assoziierte Klassen mitlädt.

6.2.3 User-Interface

Aus dem Modell soll eine grafische Benutzeroberfläche generiert werden. Dabei sollen Swing-Komponenten verwendet werden.

Entity

Für jede modellierte Entity werden zwei GUIs generiert. Zum Einen die Tabellenwartung, zum Anderen die Detailwartung. Die Tabellenwartung dient als Einstieg in die Wartung der Entity. Hier werden die Datensätze in einer Tabelle angezeigt. Durch Auswählen einer Zeile kann diese gelöscht oder geändert werden. Von der Tabellenwartung aus ist es auch möglich, Datensätze zu löschen.
Die Detailwartung wird von der Tabellenwartung aufgerufen, wenn ein neuer Datensatz angelegt oder geändert werden soll.

Attribut

Für jedes Attribut wird eine Spalte in der Tabellenwartung generiert. In der Detailwartung wird jeweils ein Eingabefeld definiert. Da das Werkzeug im Rahmen dieser Arbeit erstmal prototypisch entwickelt werden soll, werden alle Attribute gleich behandelt. Das bedeutet konkret, dass ausschließlich Textfelder generiert werden, auch wenn es sich um numerische Attribute handelt. Es erfolgt zum jetztigen Zeitpunkt noch keine Plausibilierung der Eingabewerte, diese könnte allerdings in einer späteren Version dieses Werkzeugs implementiert werden.

Assoziation

Auch für Assoziationsenden mit der Kardinalität von 1 werden Eingabefelder in der Detailwartung generiert. In der Eingabemaske der generierten Detailwartung muss der Key der assoziierten Klasse in diese Eingabefeld eingegeben werden, damit wird die Verbindung zwischen den assoziierenden Klassen hergestellt. Ausserdem wird ebenfalls eine Spalte in der Tabellenwartung generiert.

6.3 Erzeugen der Datenbankschicht

Da die Daten, die im Prototypen erfasst werden, auch persistiert werden sollen, muss eine Datenbankschicht generiert werden. Als Basis dient das Hibernate-Framework mit einer HSQL-Datenbank.

6.3.1 Hibernate Cartridge

Zum Generieren der Datenbankschicht wird ein bereits vorgefertigtes Hibernate Cartridge verwendet. Das Cartridge beinhaltet eine Reihe von Templates zum Generieren von Klassenrümpfen und den Konfigurationsdateien für das Hibernate Framework. Außerdem verfügt das Cartridge über ein Metamodell, mit dem es möglich ist, UML-Klassendiagramme zu modellieren, welche später von dem Cartridge verarbeitet werden können. Durch dieses Metamodell werden die Stereotypen wie die *Entity* der UML zur Verfügung gestellt. In dieser Arbeit wird die Version 1.3.0 verwendet, welche unter der folgender URL [Kamann (2007)] zum Download steht, wo auch ein Tutorial und ein Handbuch angeboten werden.

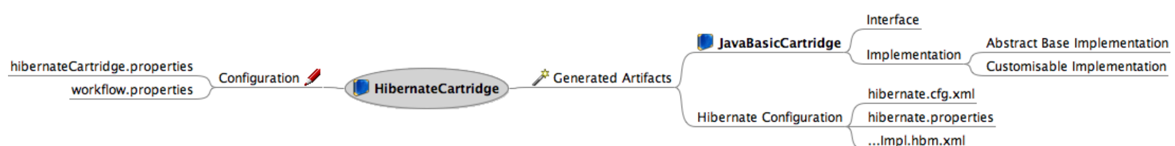

Abbildung 6.2: Funktionsweise vom Hibernate Cartridge [Kamann (2007)]

Wie in der Abbildung 6.2 zu erkennen ist, können über zwei Properties-Files einige Konfigurationen vorgenommen werden. Welche genau wir später in diesem Kaptitel erläutert. Das Hibernate Cartridge besteht aus zwei Teilen, zum einem aus einem JavaBasicCartridge, welches Interfaces und die Implementierungsklassen generiert, zum anderen einen Generator für die Hibernate Konfigurationsdateien.

6.3.2 Installation

Um das Hibernate Cartridge zu installieren, muss zu Beginn ein *openArchitectureWare* Projekt angelegt werden. Danach sollten einige Verzeichnisse angelegt werden, in denen die .jar-Dateien und die später generierten Sourcen abgegelegt werden sollen. Nun werden die laut Handbuch benötigten Dateien in die jeweils dafür vorgesehenen Verzeichnisse kopiert. Wurden alle Dateien wie im Handbuch beschrieben angelegt, kann mit der Modellierung begonnen werden.

6.3.3 Konfiguration

Für die Konfiguration von Hibernate Cartridge stehen zwei properties-Dateien zur Verfügung. Zum einem die *workflow.properties* und zum anderen die *hibernateCartridge.properties*. Letztere ist nur optional und muss nicht angegeben werden, in diesem Fall werden die Defaulteinstellungen aus dem Cartridge geladen.

In den *workflow.properties* werden die Verzeichnisse für die Codegenerierung festgelegt, also in welche Verzeichnisse die Klassen und Konfigurationsdateien generiert werden sollen. Drei Verzeichnisse werden für die Ausgabe des Generierungsprozesses benötigt. Ein Verzeichnis für die genierten Java-Klassen, welche nicht manuell geändert werden sollen. Ein zweites Verzeichnis für die Hibernate Konfigurationsdateien und ein weiters für Java-Klassen, die nur einmal durch den Generierungsprozess erzeugt werden und durch weitere Generierungen nicht mehr überschrieben werde. Diese können für die individuelle Implementierung verwendet werden. Außerdem ist noch die Angabe eines weiteren Verzeichnisses nötig. In diesem stehen die Templates, der Workflow und die Modelle, welche als Eingabe für die Codegenerierung dienen.

In der *hibernateCartridge.properties* können Hibernate spezifische Einstellungen vorgenommen werden, wie zum Beispiel SQL Dialekt, Datenbanktreiber, Location der Datenbank, Zugangsdaten zur Datenbank und ob die Datenbank bei jedem Neustart neu erstellt werden soll.

6.3.4 Modellierung

Die Modellierung erfolgt, wie bereits erwähnt, mit dem UML-Tool MagicDraw. Um mit der Modellierung beginnen zu können, muss zu erst die DSL für das Hibernate Cartridge in das neu anzulegende Modell geladen werden. Dies ist nötig, damit die für das Cartridge benötigten speziellen Stereotypen zur Verfügung stehen. Auf die wichtigsten wird im folgenden näher eingegangen.

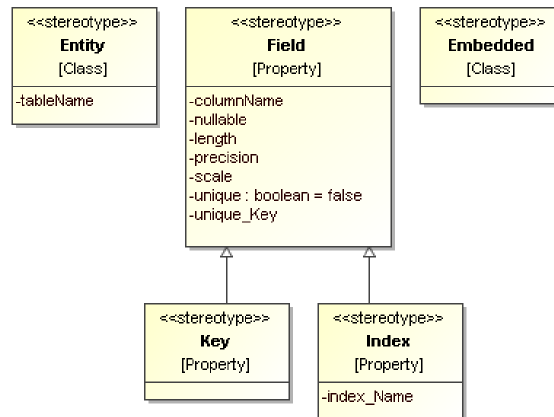

Abbildung 6.3: Persistenz Modell

Die Abbildung 6.3 zeigt das Modell, welches für Modellierung in das MagicDraw Projekt geladen werden muss. Im wesentlichen sind die Stereotypen *Entity*, *Field* und *Key* nennen. Auf diese wird im Folgenden noch näher eingegangen. Durch das Hinzufügen dieses Modells in das MagicDraw Projekt, wird das UML Standard Profil um diese Prototypen erweitert und können verwendet werden.

Entity

Alle Klassen die später persistiert werden sollen, müssen mit dem Stereotypen *Entity* versehen werden. Zusätzlich zum Klassennamen, kann ein abweichender Tabellenname angegeben werden, wie im Beispiel in der Abbildung 6.4 zu sehen ist.

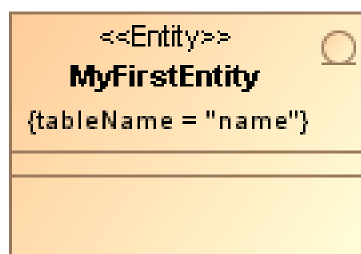

Abbildung 6.4: Modellierung einer Entity-Klasse

Zu einer Entity können folgende Parameter angegeben werden:

- Name der Entity

- Package zu dem die Entity zugeordnet ist
- Ein abweichender Tabellenname
- Attribute
- Beziehungen zu anderen Klassen
- Operationen

Mindestens sollten der Name der Entity und das Package angegeben werden.

Attribute

Entities können Attribute haben. Attribute haben wiederum Namen und Typen. Die Namen sollten den Java Namenskonventionen entsprechen. Es werden alle UML Basis Typen unterstützt, außerdem werden noch zusätzlich Hibernate Datentypen unterstützt, auf diese wird später noch weiter eingegangen. Möchte man ein Attribut als Primärschlüssel deklarieren, kann man dies durch die Angabe des Stereotypen *«key»*. Leider sind zusammengesetzte Schlüssel in dieser Version noch nicht möglich. Durch die Angabe des Stereotypen *«field»* können zu einem Attribut noch weitere Paramater angegeben werden.

Folgende Parameter stehen zur Verfügung:

- columnName
- nullable
- length
- precision
- scale
- unique:boolean = false
- unique_key

Jedes Entity muss einen Primärschlüssel haben, ist keiner explizit angegeben, wird ein Default-Schlüssel hinzugefügt.

Beziehungen

Wie normale Klassen können auch Entities in Beziehung zu einander stehen.

Datentypen

Es werden alle UML2 Datentypen vom Hibernate Cartridge unterstützt. Es ist allerdings auch möglich, Java spezifische Datentypen zu benutzen, allerdings müssen diese manuell dem Modell hinzugefügt werden. Dies kann über den Stereotyp *«DataType»* erfolgen.

6.3.5 Generierung

Die Generierung lässt sich in drei Bereiche einteilen. Zum einem die Generierung von Java-Klassen, die Generierung der Hibernate-Konfigurationsdatei und der Erstellung der O/R Mapping-Files.

Für jede modellierte Entity werden drei Artefakte generiert. Als erstes ist das Interface zu nennen, welches alle getter- und setter-Methoden beinhaltet. Außerdem wird eine abstrakte Klasse generiert, diese implementiert das oben genannte Interface. Beim ersten Generierungsprozess wird zusätzlich noch eine weitere Klasse erzeugt, welche von der abstrakten erbt. Diese ist für die individuelle Implementierung gedacht, und wird bei allen folgenden Generierungen nicht überschrieben, so bleibt dieser Programmcode erhalten.

Für das Hibernate Framework wird außerdem noch die Konfigurationsdatei *hibernate.hbm.xml* erzeugt. Im wesentlichen werden hier nur die bereits in der hibernateCartridge.properties ein getragenen Einstellungen übernommen. Zusätzlich werden nur noch die Mappingsfiles definiert.

Genau diese Mappingfiles sind der dritte Bereich in der Generierung.

6.4 Erzeugen der Anpassungschicht

Zu jeder Entity soll ein DAO (Data Access Object) generiert werden. Um dies zu realisieren müssen eigene Templates geschrieben werden.

6.5 Erzeugen des User-Interfaces

Das Generieren des User-Interfaces ist der schwierigste und aufwendigste Teil des Generierungsprozesses. Es sollen für jede Entity zwei GUIs generiert werden. Die erste ist die Tabellenwartung, welche auch als Einstieg ins Wartung einer Entity dient. Die zweite ist die Detailwartung. Hier ist es möglich, Daten eine Entity zu ändern oder anzulegen.

6.5.1 Tabellenwartung

Die erste Klasse für die GUI die aus einer Entity generiert wird, ist die Tabellenwartung. Diese dient als Einstieg für die Wartung der Entity. Jede Tabellenwartung beinhaltet eine JTable. Für jedes Attribut der Entity wird eine Spalte in der JTable erzeugt. Die Überschriften werden mit den Namen der Attribute versehen. Außerdem werden noch drei Buttons erzeugt. Diese Buttons dienen zum Löschen und Öffnen eines ausgewählten Satzes und zum Anlegen eines neuen Satzes. Beim Starten der Tabellenwartung wird die Tabelle aus der Datenbank gefüllt.

Die folgende Abbildung 6.5 zeigt eine generierte GUI, die aus der Entity *Kunde* aus der Beispielanwendung erzeugt wurde.

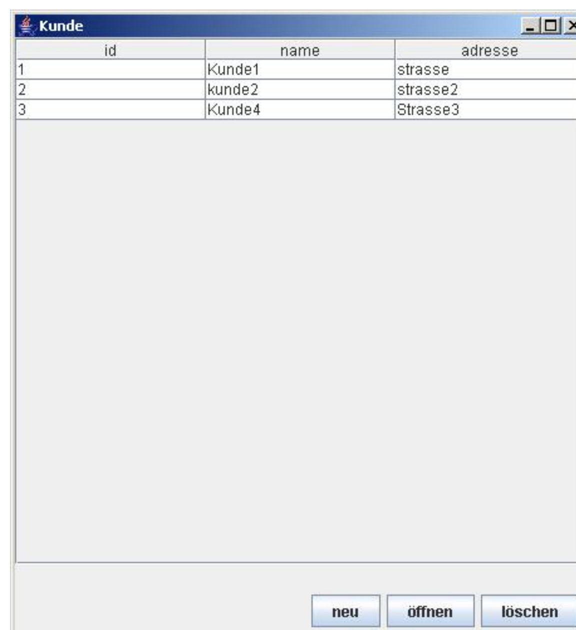

Abbildung 6.5: Generierte Tabellenwartung für Entity Kunde

6.5.2 Detailwartung

Die Detailwartung dient zum Editieren eines Datenbanksatzes. Es wird für jedes Attribut ein Eingabefeld vom Typ *JText* generiert. Außerdem werden für jeden Key der assoziierten Klasse, wenn das Assoziationende an der assoziierten Klasse eine Kardinalität von 1 aufweist, ein Eingabefeld erzeugt. Anhand des hier eingegeben Wertes wird eine Instanz der assoziierten Klasse mit Werten aus der Datenbank geladen und in der eigentlichen Entity abgelegt.

Durch das Betätigen des Save-Button wird die Entity gespeichert. Die folgende Abbildung 6.6 zeigt die GUI für die Detailwartung der Entity *Bestellposition*.

Abbildung 6.6: Generierte Detailwartung für Entity Bestellposition

6.6 Machbarkeitsbeweis

Es konnte gezeigt werden, dass es möglich ist, aus einem bereits vorläufigen UML-Klassendiagramm ein lauffähigen Prototypen zu generieren. Es wurde eine Benutzeroberfläche erzeugt, mit welcher es möglich ist, Daten zu erfassen, zu editieren und zu löschen. Außerdem wurde eine Datenbankanbindung automatisch erzeugt, die es möglich macht, die eingebenden Daten dauerhaft zu persistieren.
Durch die Generierung von abstrakten Klassen und Implementierungsklassen wird gewährleistet, dass bereits individuell programmierter Code erhalten bleibt und nicht durch das Wiederholen des Generierungsprozesses überschrieben wird. Damit ist dieses Werkzeug auch für iterative und evolutionäre Projekte geeignet.

7 Schluss

7.1 Zusammenfassung

Diese Arbeit hat sich mit der modellgetriebenen Softwareentwicklung beschäftigt. Es wurde ein Werkzeug geschaffen, mit dem es möglich ist, einen Prototypen mit GUI, einer Anpassungschicht und Datenbankanbindung aus einem UML-KLassendiagramm zu erzeugen. Nach dem ein Anforderungsprofil für das in dieser Arbeit zu entwickelnde Werkzeug erstellt wurde, wurden einige Werkzeuge der MDA vorgestellt. Nachdem sich für die verwendeten Werkzeuge entschieden wurde, werden die Grundlagen, die für die Bearbeitung nötig sind, näher vorgestellt. Darauf folgt eine Betrachtung auf die Architekturen. Dazu zählte die fachliche und technische Architektur des Prototypen und der Entwicklungsumgebung. Zum Schluss wurde auf die Einzelheiten der Realisierung eingegangen. Dabei wurde auf die Besonderheiten der Modellierung näher eingegangen. Nachdem ein Klassendiagramm für die zu erzeugende Anwendung erstellt wurde, wurden die Transformationsregeln für die Generierung der Datenbankschicht, Anpassungsschicht, Business Logik und des User-Interfaces ausgestellt.

7.2 Grenzen des Werkzeuges

Es ist mit Sicherheit nicht möglich, mit einem solchen Werkzeug eine komplexe Anwendung komplett ohne individuelle Programmierung zu Generieren. Es wird immer nur eine Art lauffähige Rumpfanwendung zu generieren sein. Besonders in der Business Logik trifft ein solches Werkzeug auf seine Grenzen. Zudem verfügt der hier generierte Prototyp über keine Validierung der Eingabewerte. In der jetzigen Version wird leider keine zusammenhängende Anwendung generiert, sondern nur einzelne unabhängige Wartungsprogramme.

7.3 Weiterentwicklungsmöglichkeiten

Für das Werkzeug gibt es allerdings auch noch diverse Weiterentwicklungsmöglichkeiten. Zum Einen könnten die Eingabewerte in der Detailwartung plausibilisiert werden. Zudem könnten auch individuell modellierte Datentypen berücksichtigt werden. In der jetzigen Version werden nur die in Java bekannten Datentypen verwendet. Außerdem könnte die Möglichkeit geschaffen werden, mehrere Klassen zu einer GUI zusammenzufassen. Z.B. könnte eine Klasse Kunde modelliert werden mit dem Attribut Adresse, welches wiederum eine eigene Klasse ist. Man möchte zum Beispiel die Adresse direkt im Kunden pflegen können und nicht in eine extra Anwendung wechseln.

7.4 Fazit

Es ist mit dieser Arbeit erfolgreich gezeigt worden, dass es durchaus möglich ist, aus einem UML-Klassendiagramm eine lauffähige Anwendung zu generieren, ohne das zusätzlich programmiert werden muss. Nachdem die Templates implementiert wurden, können theoretisch beliebig unterschiedlich Anwendungen erzeugt werden. Auch die benutzen Werkzeuge haben meine Erwartungen erfüllt. Man hatte mit dem openArchtiktureWare schnell erste Erfolge zu verzeichnen und konnte sich so langsam durch die Problematiken tasten. Besonders die Integration in Eclipse hat sehr geholfen, da man sich nicht erst in eine neue Entwicklungumgebung einarbeiten musste. Auch die Autovervollständigung, die man auch aus der Javaentwicklung kennt, war sehr hilfreich. Diese hat in vielen Fällen das Lesen von Java-Docs und Dokumentationen erspart. Das Programmieren war sehr angenehm, da man sich auf das Programmieren der Referenzanwendung konzentrieren konnte.

Literaturverzeichnis

[Apache 2007] APACHE: *Apache Maven Project*. 2007. – URL http://maven.apache.org/

[ArcStyler 2007] ARCSTYLER: *Hompage von ArcStyler*. 2007. – URL http://www.interactive-objects.com/products/arcstyler/support-documentation

[Bauer und King 2004] BAUER, Christian ; KING, Gavin: *Hibernate in Action*. Manning, 2004. – ISBN 1-932394-15-X

[Eclipse 2007] ECLIPSE: *GMT Project*. 2007. – URL http://www.eclipse.org/gmt/

[Efftinge 2007] EFFTINGE, Sven: *Xpand Language Reference*. Referenzhandbuch. : , 2007

[Form4 2007] FORM4: *Model Driven Architecture (MDA)*. 2007. – URL http://www.form4.de/technologie/uml-und-mda/model-driven-architecture/model-driven-architecture-mda/

[Fornax 2007] FORNAX: *Homepage Fornax*. 2007. – URL http://www.fornax-platform.org/

[Goossens u. a. 2000] GOOSSENS, Michel ; MITTELBACH, Frank ; SAMARIN, Alexander: *Der LaTeX -Begleiter*. Addison-Wesley, 2000. – ISBN 3-8273-1689-8

[Hansen 1998] HANSEN, H. R.: *Wirtschaftsinformatik I*. UTB für Wissenschaft, 1998. – ISBN 9-783825-208028

[Kamann 2007] KAMANN, Thorsten: *Hibernate Cartridge*. 2007. – URL http://www.fornax-platform.org/cp/display/fornax/Hibernate+%28CHB%29

[Kecher 2006] KECHER, Christoph: *UML 2.0 - Das umfassende Handbuch*. Galileo Computing, 2006. – ISBN 3-89842-738-2

[Köhler 2003] KÖHLER, Kristian: *XDoclet - MDA für Praktiker?* 2003. – URL http://www.oio.de/xdoclet.pdf

[MagicDraw 2007] MAGICDRAW: *Homepage MagicDraw*. 2007. – URL http://www.magicdraw.com/

[OMG 2007] OMG: *OMG MDA*. 2007. – URL http://www.omg.org/mda/

[OpenMDX 2007] OPENMDX: *Hompage von OpenMDX*. 2007. – URL http://www.openmdx.org/index.html

[Peak und Heudecker 2006] PEAK, Patrick ; HEUDECKER, Nick: *Hibernate Quickly*. Manning, 2006. – ISBN 1-932394-41-9

[Schulz 2005] SCHULZ, Daniel: *MDA-Frameworks:AndroMDA*, Uni Münster, Studienarbeit, 2005. – URL http://www.wi.uni-muenster.de/pi/lehre/ws0506/seminar/02_andromda.pdf

[Softwarekompetenz 2007] SOFTWAREKOMPETENZ: *Softwarekompetenz*. 2007. – URL http://www.softwarekompetenz.de/?26981

[Stahl u. a. 2007] STAHL, Thomas ; VÖLTER, Markus ; EFFTINGE, Sven ; HAASE, Arno: *Modellgetriebene Softwarentwicklung*. **dpunkt**.verlag, 2007. – ISBN 3-89864-448-8

[Voelter 2007] VOELTER, Markus: *oAW 4 Installation*. 2007. – URL http://www.eclipse.org/gmt/oaw/doc/4.0/10_installation.pdf

[Warmer 2005] WARMER, Jos: *Introduction to OCL*. 2005. – URL http://www.klasse.nl/ocl/ocl-introduction.html

[XDoclet 2007] XDOCLET: *XDoclet*. 2007. – URL http://xdoclet.sourceforge.net/xdoclet/index.html

[Zukowski 2005] ZUKOWSKI, John: *The Definitive Guide to Java Swing*. Apress, 2005. – ISBN 1-59059-447-9

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 14. September 2007	
Ort, Datum	Unterschrift