Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

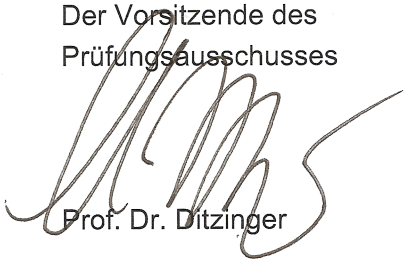
Fakultät für Informatik und Wirtschaftsinformatik

Bachelor-Thesis

Name:	Tobias Wink
Thema:	Generierung der Persistenzschicht durch Erstellung einer domänenspezifischen Sprache
Arbeitsplatz:	msgGillardon AG, Bretten
Referent:	Prof. Dr.-Ing. Vogelsang
Korreferent:	Prof. Dr. Pape
Abgabetermin:	14.01.2015

Karlsruhe, 15.10.2014

Der Vorsitzende des
Prüfungsausschusses



Prof. Dr. Ditzinger

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Bretten, den 9. Januar 2015 Tobias Wink

Zusammenfassung

Das interne Application Server Framework (ASF) Version 1 der msgGillardon AG wird in verschiedenen Produkten des Unternehmens verwendet. Es stellt den nutzenden Anwendungen zentrale Dienste bereit, unter anderem den PersistenceService. Teil des PersistenceService ist der POG2, welcher auf Basis von UML-Klassendiagrammen die Persistenzschicht generiert.

Aufgrund des Dateiformats der Klassendiagramme ist die sinnvolle Zusammenführung zweier unterschiedlicher Versionen eines Klassendiagramms nur unter hohem Aufwand möglich. Durch diese Einschränkung ist die parallele Bearbeitung nicht möglich und somit benötigt die Bearbeitung der Diagramme innerhalb des Projektteams einen erhöhten Koordinationsaufwand.

Ziel dieser Arbeit ist die Erstellung einer textuellen domänenspezifischen Sprache (DSL) auf Basis von Xtext. Dadurch wird versucht, die vorhandenen Einschränkungen des POG2 zu beseitigen und so den gesamten Arbeitsablauf für die Generierung der Persistenzschicht effizienter zu gestalten.

Dazu werden zunächst die für das Verständnis der Arbeit notwendigen Grundlagen vermittelt. Anschließend wird der bestehende Workflow analysiert und die Anforderungen an die textuelle DSL festgelegt. Im nachfolgenden Kapitel wird die erstellte Sprache mit ihren Elementen und Konzepten vorgestellt. Das Kapitel Implementierung gibt einen Überblick über die Realisierung und beleuchtet technisch interessante Vorgehensweisen. Im Anschluss daran werden die bestehende und die neu erarbeitete Lösung miteinander verglichen. Abschließend soll ein Ausblick die Möglichkeiten zur Erweiterung und Verbesserung der Arbeit aufzeigen.

Abstract

The internal Application Server Framework (ASF) version 1 of msgGillardon AG is used by several products of this company. It provides the utilising applications with central services, among other things the persistence service. Part of the persistence service is the POG2 which generates the persistence layer on the basis of UML class diagrams.

Due to the data format of the class diagrams the reasonable consolidation of two different versions requires great efforts. Because of this limitation the parallel processing is not possible and so the editing of the diagrams within the project team means increased coordination efforts.

The target of this thesis is the creation of a textual domain-specific language (DSL) based on Xtext. Thereby the existing restrictions of the POG2 are intended to be eliminated and thus the complete work sequence for the generation of the persistance layer shall get more efficient.

For that purpose the basic knowledge for the understanding of this thesis is imparted first of all. Afterwards, the existent work flow is analysed and the demands on the textual DSL are determined. Following is the presentation of the created language with its elements and concepts. The next chapter gives an overview about the realisation and it highlights technical interesting procedures. Subsequently, the existing and the new solution are compared. Finally, an outlook shall point out the prospects of extension and improvement of this work.

Inhaltsverzeichnis

Abkürzungsverzeichnis

Bezeichnung	Beschreibung
ANTLR	Another Tool for Language Recognition
API	Application Programming Interface
ASF	Application Server Framework
AST	Abstract Syntax Tree
BPMN	Business Process Model Notation
CVS	Concurrent Versions System
DDL	Data Definition Language
DSL	domain-specific language
EBNF	Erweiterte Backus-Naur Form
EMF	Eclipse Modeling Framework
IDE	Integrated Development Environment
JPA	Java Persistence API
JSP	Java Server Pages
JVM	Java Virtual Machine
MWE	Modeling Workflow Engine
oAW	openArchitectureWare
POG	Peristent Object Generator
RC	Release Candidate
RCS	Revision Control System
SCCS	Source Code Control System
SQL	Structured Query Language
SVN	Subversion
SysML	Systems Modeling Language

Bezeichnung	Beschreibung
TMF	Textual Modeling Framework
UML	Unified Modeling Language
UPDM	Unified Profile for DoDAF/MODAF
VCS	Version Control System
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

KAPITEL 1

Einleitung

1.1 Umfeld

Abb. 1.1: msgGillardon Logo

Neben der Beratung ihrer namhaften Kunden aus der Finanzdienstleistungsbranche in den D-A-CH-Ländern bietet die msgGillardon AG kundenspezifische Lösungen, sowohl auf Basis der eigenen Softwareprodukte als auch auf Basis marktüblicher Standardsoftware. Sie entstand 2008 durch den Zusammenschluss des Geschäftsbereichs Finanzdienstleistungen der msg systems ag und der GILLARDON AG financial software. Der Hauptsitz befindet sich in Bretten. Außerdem unterhält die msgGillardon AG noch weitere Standorte in Ismaning, Eschborn, Passau, Hürth, Berlin und Wien [vgl. AG14].

Die eigene Software wird sowohl mit C++ als auch mit Java entwickelt. Die auf Java basierenden Produkte nutzen das interne Framework ASF, welches aktuell in den Versionen 1 und 2 existiert und verwendet wird. Durch diese Frameworks werden den nutzenden Anwendungen nicht nur alle benötigten Abhängigkeiten bereitgestellt, sondern ebenfalls zentrale Dienste. Thema dieser Arbeit ist ein Teil des PersistenceServices von ASF1, der POG2.

1.2 Motivation

Die Persistenzschicht wird momentan auf Basis von UML-Klassendiagrammen erstellt. Diese Klassendiagramme werden mit MagicDraw erstellt und im XMI-Format gespeichert. Wie der Name vermuten lässt, basiert das XMI-Format auf XML und wird als Austauschformat zwischen Software-Entwicklungswerkzeugen benutzt [vgl. Wik13c]. Durch die fehlende Fixierung auf die menschliche Lesbarkeit dieses Datenformats lassen sich nur sehr aufwendig Unterschiede zwischen zwei Versionen einer Datei feststellen. Aus dieser schlechten Differenzierbarkeit resultieren die folgenden Probleme:

Keine parallele Bearbeitung möglich, da auftretende Konflikte nur unter hohem Aufwand aufgelöst werden können.

Mangelhafte Branch-Fähigkeit, da die Branches (Entwicklungszweige) nicht ohne enormen Aufwand wieder zusammengeführt werden können.

Ein weiteres Problem ist, dass der Umfang der UML-Spezifikation für Klassendiagramme viel größer ist, als es für die Generierung der Persistenzschicht notwendig wäre. Dadurch erhöht sich der Einarbeitungsaufwand, da der Entwickler wissen muss, welche Teile der Spezifikation für die Generierung benutzt werden dürfen.

Die Nutzung von UML-Klassendiagrammen für die Persistenzschicht birgt im Vergleich zu der direkten Implementierung den Vorteil, dass man sich damit schnell einen Überblick über die Zusammenhänge verschaffen kann.

Es gibt bereits OpenSource-Lösungen, beispielsweise Sculptor[1], die textuelle domain-specific languages (DSLs) zur Generierung der Persistenzschicht zur Verfügung stellen. Sculptor unterstützt sogar die Möglichkeit, direkt eine dazugehörige Serviceschicht zu generieren. Ein Umstieg auf eine bereits verfügbare Lösung würde jedoch einen erhöhten Migrationsaufwand bedeuten, da die von einer solchen Lösung generierte Persistenzschicht nicht alle Funktionalitäten so bereitstellen würde, wie es die bisherige Lösung tut. Das hätte zur Folge, dass alle auf dem Framework basierenden Anwendungen angepasst werden müssten. Ein weiterer Kritikpunkt ist, dass durch die bereits verfügbaren Lösungen, ähnlich wie bei der bestehenden Lösung, mehr Funktionalität zur Verfügung gestellt werden würde als für das interne Framework benötigt wird. Dadurch würde die Komplexität unnötig erhöht werden, da man bei der Benutzung wissen müsste, welche Funktionalitäten genutzt werden sollen und welche nicht.

Aus diesem Grund soll im Rahmen dieser Arbeit eine textuelle DSL für einen alternativen Workflow erarbeitet werden.

1.3 Ziel und Aufbau dieser Arbeit

Ziel dieser Arbeit ist die Konzeption und Realisierung einer textuellen DSL, welche die Generierung der Persistenzschicht innerhalb von ASF1 übernimmt. Dabei soll die DSL sowohl das Problem der Differenzierbarkeit lösen als auch nur den Sprachumfang besitzen, der tatsächlich benötigt wird. So soll parallele Bearbeitung ermöglicht und ein möglichst geringer Einarbeitungsaufwand erreicht werden.

Im Detail beinhaltet die Umsetzung die folgenden Arbeitsschritte:

- Analysieren der Anforderungen
- Konzeption der Grammatik

1 `http://sculptorgenerator.org/`

- Implementierung der Grammatik in Xtext
- Generierung der Persistenzschicht

Zu Beginn werden zunächst die für das Verständnis der Arbeit entscheidenden Grundlagen vermittelt.

In der Analyse werden anhand der aktuellen Lösung die Musskriterien für eine DSL-basierte Lösung erarbeitet.

Im nachfolgenden Kapitel wird die erstellte Sprache mit ihren Elementen und Konzepten vorgestellt.

Das Kapitel Implementierung soll einen Überblick über die Realisierung geben und beleuchtet technisch interessante Vorgehensweisen.

Im Anschluss daran werden die bestehende und die neu erarbeitete Lösung miteinander verglichen.

Abschließend soll ein Ausblick die Möglichkeiten zur Erweiterung und Verbesserung der Arbeit aufzeigen.

KAPITEL 2

Grundlagen

2.1 Der Softwareentwicklungsprozess bei msgGillardon

Der Alltag eines Entwicklers bei der msgGillardon AG besteht in erster Linie aus der Benutzung der folgenden drei Systeme: des Entwickler-PCs, des Versionsverwaltungssystems und des Buildsystems. Diese Systeme sollen im Folgenden etwas genauer erläutert werden. Das Zusammenspiel dieser drei Systeme wird in Abbildung 2.1 nochmals schematisch dargestellt.

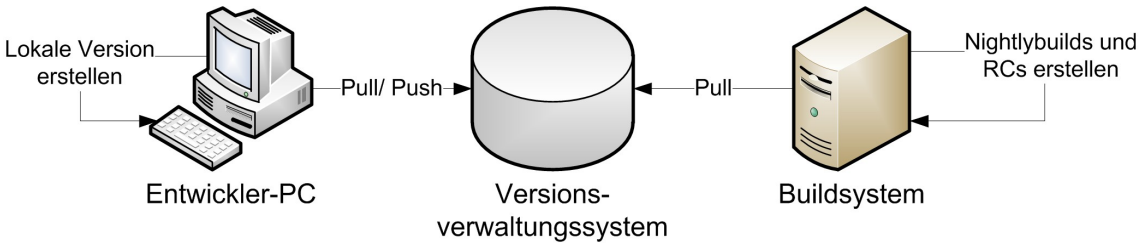


Abbildung 2.1: Softwareentwicklungsprozess

Entwickler-PC

Der Entwickler-PC ist der lokale Arbeitsplatz jedes Entwicklers. Entwickler besitzen für die Konfiguration ihrer lokalen Systeme große Freiheiten. Aus diesem Grund werden Softwareversionen, die auf dem Entwickler-PC erstellt wurden, nur zu Entwicklungs- und Testzwecken verwendet und stellen keine Kundenversion dar.

Versionsverwaltungssystem

Das Versionsverwaltungssystem dient der zentralen Ablage des Quellcodes für die verschiedenen Projekte. Es wird in den Abschnitten 2.11 und 2.12.2 ausführlich behandelt.

Buildsystem

Das Buildsystem stellt eine standardisierte Umgebung für die Softwareerstellung bereit. Dort wird in der Regel jede Nacht die jeweils aktuelle Version der verschiedenen Projekte erstellt, welche „Nightlybuild“ genannt werden. Zu dieser Erstellung gehört nicht nur die reine Kompilierung des Quellcodes, sondern ebenfalls das Ausführen von Tests und das Sammeln von Informationen für den zentralen SonarQube[1]-Server. Über das Buildsystem lassen sich ebenfalls sogenannte Releasekandidaten (RC) erstellen, welche anschließend zu einem Release erklärt werden können.

Das Buildsystem basiert dabei auf Jenkins und Ant, worauf in den Abschnitten 2.12.4 bzw. 2.12.3 genauer eingegangen wird.

2.2 Persistenzschicht

Persistenz in der Softwareentwicklung bedeutet, dass Daten dauerhaft gespeichert werden. Gerade bei größeren Datenmengen bieten sich aus Gründen der Performance Datenbanken an, welche klassischerweise relational arbeiten.

Dieses Konzept beruht auf der relationalen Algebra des Briten Edgar F. Codd aus dem Jahr 1970. Eine Relation besteht aus Attributen und Tupeln. Ein Attribut beschreibt den Typ eines möglichen Attributwertes und bezeichnet ihn mit einem Attributnamen. Ein Tupel stellt eine konkrete Kombination von Attributwerten dar [vgl. Wik13b].

Relationale Datenbanken bilden diese Relationen üblicherweise durch Tabellen ab. Attribute entsprechen den jeweiligen Spaltenköpfen innerhalb der Tabelle und die Attributwerte den in den Spalten vorhandenen Einträgen. Ein Tupel entspricht dabei einer Zeile einer Tabelle.

In einer Applikation ist es typischerweise sinnvoll, Klassen nach Funktionsbereich zu organisieren. Bei Verwendung solcher Schichten gilt die folgende Regel:

Schichten kommunizieren von oben nach unten. Eine Schicht besitzt Abhängigkeiten nur zu der Schicht direkt unter ihr und kennt außer dieser auch keine andere Schicht [vgl. Chr07, S. 19].

Eine häufig anzutreffende Grundform der Schichtenarchitektur ist in Abbildung 2.2 dargestellt. Sie besteht aus drei Schichten:

- **Präsentationsschicht:** Ist für die Benutzerinteraktion zuständig.
- **Business-Schicht:** Beinhaltet den eigentlichen Anwendungskern, die Geschäftslogik.
- **Persistenzschicht:** Beinhaltet alle notwendigen Klassen und Komponenten, um Daten dauerhaft zu speichern und auszulesen.

1 SonarQube ist eine OpenSource Qualitätsmanagement Plattform für die Softwareentwicklung. Weitere Informationen unter `http://www.sonarqube.org`

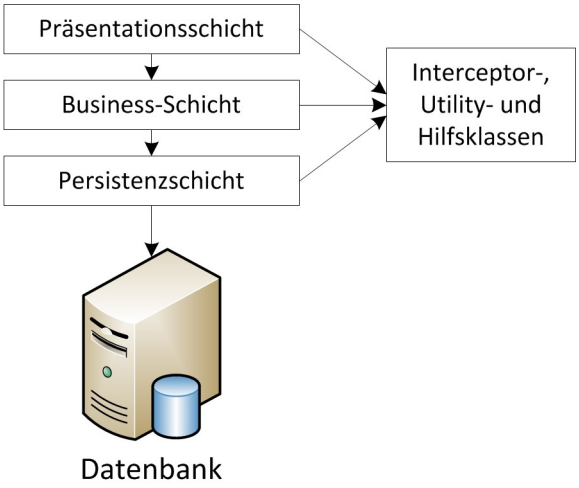


Abbildung 2.2: Schichtenarchitektur [Chr07]

- **Datenbank:** Ist für das dauerhafte Speichern der Daten auf dem Datenträger zuständig. Die Vielfalt reicht von einer einfachen Textdatei bis zu einem vollständigen Datenbank-Management-System.
- **Interceptor-, Utility- und Hilfsklassen:** Stellen allgemein verfügbare Komponenten dar, die von allen Schichten verwendet werden dürfen, z. B. Exception-Klassen.

2.3 Hibernate

Abb. 2.3: Hibernate Logo

Da Datenbanken oftmals mit Relationen arbeiten und Java, wie viele andere Programmiersprachen, objektorientiert ist, muss bei dem Zugriff auf die Datenbank zwischen Objekten und Relationen konvertiert werden. Man spricht bei dieser Konvertierung vom sogenannten Mapping. Diese Arbeit kann von Hibernate, genauer gesagt Hibernate ORM, übernommen werden.

Das Hibernate-Projekt wurde 2001 von Gavin King zusammen mit Cirrus Technologies als Alternative zu EJB2 Entity-Beans gestartet. Das Ziel war es, bessere Persistenzunterstützung im Vergleich zu EJB2 zu erreichen, indem die Komplexität reduziert und die von der Community geforderte Funktionalität bereitgestellt wurde. Später stellte JBoss, Inc. (mittlerweile Teil von Red Hat) die leitenden Hibernate-Entwickler ein, um die Entwicklung des Projektes zu fördern. Hibernate ist eine zertifizierte Implementierung der JPA 2.0 Spezifikation (ab Version 3.5) bzw. der JPA 2.1 Spezifikation (ab Version 4.3).

Um Hibernate zu nutzen, müssen entsprechend die Mapping-Informationen hinterlegt werden. Dafür werden drei Arten unterstützt:

1. JPA2 Annotationen
2. JPA2 XML-Deployment-Deskriptoren
3. Hibernate XML-Dateien, bekannt als `hbm.xml`

In dieser Arbeit werden Annotationen verwendet.

2.4 Unified Modeling Language (UML)

Die Unified Modeling Language, kurz UML, ist eine grafische Modellierungssprache zur Spezifikation und Dokumentation von Software-Komponenten und anderen Systemen. Sie wird von der Object Management Group entwickelt und ist sowohl von ihr als auch von der ISO standardisiert. UML definiert die meisten bei einer Modellierung wichtigen Begriffe und legt mögliche Beziehungen zwischen diesen fest. UML definiert außerdem grafische Notationen für diese Begriffe sowie für Modelle statischer Strukturen und dynamischer Abläufe, die man mit diesen Begriffen formulieren kann. Insgesamt werden von UML2 14 verschiedene Diagrammtypen spezifiziert, welche in die Kategorien Strukturdiagramme und Verhaltensdiagramme eingeordnet sind. Für diese Arbeit ist vor allem das Klassendiagramm von Interesse, welches zur ersten Kategorie gehört [vgl. Wik14f].

Abb. 2.4: UML Logo

Ein Klassendiagramm ist ein Strukturdiagramm der UML zur grafischen Darstellung von Klassen, Schnittstellen und deren Beziehungen [vgl. Wik14c].

2.5 MagicDraw

MagicDraw ist ein grafischer Editor von No Magic, Inc., der neben UML noch die folgenden weiteren Sprachen unterstützt: SysML, BPMN und UPDM. Diese werden nur der Vollständigkeit halber erwähnt, da sie für das Verständnis der Arbeit irrelevant sind.

Abb. 2.5: MagicDraw Logo

MagicDraw unterstützt sowohl Round-Trip-Engineering für die J2EE, C#, C++, CORBA IDL, .NET, XML-Schema und WSDL, als auch Datenbankschema-Modellierung und DDL-Generierung [vgl. Wik14d].

2.6 Domain-specific language (DSL)

Eine domänenspezifische Sprache (im Englischen domain-specific language, kurz DSL) ist eine formale Sprache, die speziell für ein bestimmtes Problemfeld (Domäne) entworfen und implementiert wird. Beim Entwurf wird man bemüht sein, einen

hohen Grad an Problemspezifität zu erreichen. Dies bedeutet, dass die Sprache in der Lage sein soll, alle Probleme der Domäne darzustellen. Sie soll dagegen nichts darstellen können, was außerhalb der Domäne liegt. Dadurch sollte sie durch Domänenspezialisten ohne besonderes Zusatzwissen leicht bedienbar sein.

Das Gegenteil einer domänenspezifischen Sprache ist eine universell einsetzbare Programmiersprache, wie C oder Java oder aber eine universell einsetzbare Modellierungssprache, wie UML [vgl. Wik14a].

2.7 openArchitectureWare (oAW)

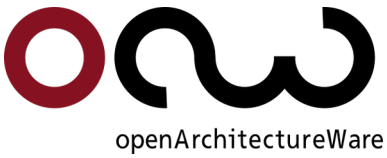


Abb. 2.6: oAW Logo

Hinter dem Begriff openArchitectureWare, kurz oAW, verbirgt sich eine Plattform für die modellgetriebene Softwareentwicklung und das ebenfalls modellgetriebene Testen.

Im Wesentlichen bietet oAW die Möglichkeit, Codegeneratoren für verschiedenste Modelle zu verarbeiten. Zu diesen Modellen gehören EMF-Modelle, fast alle mit UML-Werkzeugen erstellten Modelle, aber auch Microsoft Visio-Modelle oder textuelle Spezifikationen. Aus den Modellquellen kann beliebiger Quellcode generiert werden. Umfangreiche Möglichkeiten für Modellvalidierungen und einfache Modelltransformationen stehen ebenfalls zur Verfügung.

Durch die Integration in das Eclipse Galileo Release (2009) wurde oAW unter dem Namen MWE Teil des Eclipse Modeling Projects und wird seitdem nicht mehr eigenständig weiterentwickelt [vgl. Wik13a].

2.8 Xtext

Abb. 2.7: Xtext Logo

Xtext ist ein OpenSource-Framework für die Entwicklung von Programmiersprachen und ein Teil des EMF-Projekts. Im Gegensatz zu vielen anderen Parsergeneratoren erzeugt Xtext nicht nur einen Parser, sondern stellt zudem die folgenden Komponenten zur Verfügung:

- Ein Klassenmodell für den typsicheren abstrakten Syntaxbaum
- Einen in Eclipse integrierten Texteditor
- Die notwendige Infrastruktur für die Implementierung einer modernen Entwicklungsumgebung

Die erste Version von Xtext wurde im Jahre 2006 im Rahmen von oAW veröffentlicht. Seit Anfang 2008 wird Xtext innerhalb des Eclipse Modeling Project im TMF weiterentwickelt. Am 3. September 2014 erschien die Version 2.7.0, welche für die Umsetzung dieser Arbeit verwendet wird [vgl. Wik14i].

Xtext nutzt MWE2, um aus der erstellten Grammatik den Parser, den Serializer und Xtend-Stubs für Syntaxhervorhebung, Validierung und Autovervollständigung zu generieren[vgl. Xte14b]. Zur Code-Generierung werden aktuell zwei Wege zur Verfügung gestellt, die in Kapitel 5.3 genauer erläutert werden.

2.9 Xpand

Xpand ist eine statisch typisierte Templatesprache. Sie wurde ursprünglich als Teil der oAW entwickelt. Mit der Integration von oAW in Eclipse wurde Xpand eine Eclipse-Komponente [vgl. Fou14c].

Abb. 2.8: Xpand Logo

Xpand unterstützt dabei unter anderem:

- Polymorphes Aufrufen von Templates
- Aspektorientierte Programmierung
- Funktionale Erweiterungen
- Flexible Abstraktion des Typsystems
- Modelltransformation
- Modellvalidierung

2.10 Xtend

Xtend ist eine Programmiersprache für die JVM. Sie stammt ursprünglich aus dem Xtext-Projekt, wurde aber 2012 in ein eigenes Eclipse-Projekt ausgelagert. Der Quellcode ist unter der Eclipse Public License frei erhältlich.

Abb. 2.9: Xtend Logo

Syntaktisch und semantisch hat Xtend seine Wurzeln in der Programmiersprache Java. Im Vordergrund stehen allerdings eine kompaktere Syntax und zusätzliche Funktionalitäten wie Typinferenz, Extension-Methoden und Operatorüberladung. Xtend ist primär objektorientiert, ermöglicht aber auch funktionale Programmierung, z. B. durch Lambda-Ausdrücke. Xtend ist statisch typisiert und benutzt das Typsystem von Java ohne Änderungen. Es wird auf Java-Code kompiliert und integriert sich daher nahtlos in existierende Java-Bibliotheken [vgl. Wik14h].

Xpand wurde bei der Code-Generierung durch Xtend abgelöst, da Xtend im Vergleich unter anderem die folgenden Vorteile bietet [vgl. Eff13]:

- Durch die Übersetzung von Xtend in Java-Code läuft der Xtend-Code in der JVM. Das Ausführen der kompilierten Generierung kann im Vergleich zur Interpretierung des Xpand-Templates bis zu zehnmal schneller sein.

- Durch die Übersetzung von Xtend in Java-Code ist er als solcher ebenso debugbar. Ein weiterer Vorteil ist, dass durch die gute Eclipse-Integration sogar direktes Debuggen des Xtend-Codes möglich ist.

2.10.1 Kurzeinführung in Xtend

Da Xtext in der genutzten Version für viele Komponenten Xtend als Sprache vorsieht, soll an dieser Stelle ein kurzer Einblick in die Sprache gegeben werden und die wichtigsten Unterschiede im Vergleich zu Java aufgezeigt werden [vgl. Xte14a]:

Variablendeklaration

```
<var|val> [<Datentyp>] <Name> [= <Initialwert>]
```

Listing 2.1: Grundlegende Syntax einer Xtend-Variablendeklaration.

Listing 2.1 zeigt die grundlegende Syntax der Variablendeklaration in Xtend. Wie zu erkennen ist, können Variablen mit den Schlüsselwörtern `var` und `val` eingeleitet werden, wobei das Schlüsselwort `val` die Variable als `final` deklariert. Wird die Variable direkt mit einem Wert initialisiert, so ist die Angabe des Datentyps optional und als Datentyp wird entsprechend der Typ des Wertes übernommen. Bei Attributen erweitert sich die Syntax entsprechend um die aus Java bekannten Sichtbarkeitsoperatoren.

Methoden

```
[<Sichtbarkeitsoperator>] <def|override> [<Rückgabetyp>] <Name> ([<
↪ Datentyp> <Name>[, <Datentyp> <Name>...]]) { ... }
```

Listing 2.2: Grundlegende Syntax einer Xtend-Methode.

Listing 2.2 zeigt die grundlegende Syntax der Methodendeklaration in Xtend. In den überwiegenden Fällen werden Methoden mit dem Schlüsselwort `def` eingeleitet, eine Angabe der Sichtbarkeit ist optional möglich, der Standardwert ist `public`. Methoden, die geerbte Methoden überschreiben, müssen mit dem Schlüsselwort `override` eingeleitet werden. Der Rückgabetyp ist optional und wird, falls nicht anders angegeben, automatisch erkannt. Falls keine `return`-Anweisung im Methodenrumpf vorhanden ist und der Rückgabetyp nicht explizit auf `void` gesetzt wurde, so wird die letzte Anweisung als `return`-Anweisung benutzt. Bei abstrakten und rekursiven Methoden ist der Rückgabetyp eine Pflichtangabe.

Erweiterungsmethoden

```
"hello".toFirstUpper() // calls StringExtensions.toFirstUpper("hello")
```

Listing 2.3: Erweiterter Methodenaufruf [Xte14a].

Mit Erweiterungsmethoden können vorhandene Klassen um neue Methoden erweitert werden. Dabei findet in Xtend keine richtige Erweiterung der Klasse statt, sondern es wird ermöglicht, dass die Methode auf dem ersten Übergabeparameter der Methode aufgerufen wird, was in Listing 2.3 anhand eines einfachen Beispiels gezeigt wird. Durch diesen Mechanismus soll eine erhöhte Lesbarkeit des Quellcodes erreicht werden.

Um Instanzmethoden einer Klasse in Erweiterungsmethoden umzuwandeln, muss das Attribut, die lokale Variable oder die Parameter-Deklaration mit dem Schlüsselwort `extension` eingebunden werden.

Templateausdrücke

Xtend unterstützt mit den Templateausdrücken eine an Xpand angelehnte Templatesprache. Templateausdrücke werden von `'''` umschlossen und können mehrere Zeilen umspannen. Innerhalb dieser Ausdrücke können nicht nur einfache Zeichenketten enthalten sein, sondern innerhalb von «» können weitere Ausdrücke stehen, die dynamisch ausgewertet werden, vergleichbar mit den JSP-Ausdrücken: `<%= Ausdruck %>`. Die Templateausdrücke werden sehr stark bei der templatebasierten Generierung genutzt, die in Abschnitt 5.3.1 genauer beschrieben wird.

2.11 Versionsverwaltung

Versionsverwaltung im Allgemeinen dient der Verfolgung von Veränderungen an Dateien und Ordnern. Dafür werden alle Versionen der Dateien und Ordner in einem Archiv (Repository) mit Zeitstempel und Benutzerkennung gespeichert und können so wiederhergestellt werden. Um Speicherplatz zu sparen, werden meist nur die Veränderungen zwischen den einzelnen Versionen gespeichert. Aus diesem Grund ist die Versionsverwaltung in erster Linie auf textbasierte Datenformate ausgelegt.

Die folgenden Punkte zählen zu den Hauptaufgaben eines Versionsverwaltungssystems [vgl. Wik14g]:

- **Änderungsprotokollierung**, um jederzeit nachvollziehen zu können, wer wann was geändert hat.
- **Wiederherstellung früherer Versionen**, wodurch versehentliche Änderungen jederzeit wieder rückgängig gemacht werden können.
- **Archivierung** ermöglicht jederzeit Zugriff auf alle Versionen.
- **Koordinierung** des gemeinsamen Zugriffs von mehreren Entwicklern auf die Dateien.
- **Gleichzeitige Entwicklung** mehrerer Entwicklungszweige (Branches) eines Projektes.

Daher werden Versionsverwaltungssysteme typischerweise in der Softwareentwicklung eingesetzt.

Insgesamt werden drei Arten von Versionsverwaltungssystemen unterschieden:

- **Lokale Versionsverwaltung:** Bei der lokalen Versionsverwaltung wird oft pro Datei eine eigene Version verwaltet. Diese Variante wurde mit Werkzeugen wie SCCS und RCS umgesetzt. Sie findet auch heute noch Verwendung in Büroanwendungen, die Versionen eines Dokumentes in der Datei des Dokuments selbst speichern. Aber auch Entwicklungsumgebungen beherrschen oft eine lokale Versionsverwaltung von Quelltextdateien.
- **Zentrale Versionsverwaltung:** Diese Art ist als Client-Server-System aufgebaut, wodurch der Zugriff auf ein Repository auch über das Netzwerk erfolgen kann. Durch eine zentrale Rechteverwaltung werden die Zugriffsrechte für jedes Archiv geregelt. Die Versionsgeschichte ist hierbei nur auf dem Server vorhanden. Dieses Konzept wurde vom OpenSource-Projekt CVS populär gemacht, mit SVN neu implementiert und wird von vielen kommerziellen Anbietern verwendet.
- **Verteilte Versionsverwaltung:** Die verteilte VCS verwendet kein zentrales Repository mehr. Jeder, der an dem verwalteten Projekt arbeitet, hat sein eigenes Archiv und kann dieses mit jedem beliebigen anderen Repository abgleichen. Die Versionsgeschichte ist dadurch genauso verteilt. Änderungen können also lokal verfolgt werden, ohne eine Verbindung zu einem Server aufbauen zu müssen. Weit verbreitet sind dabei die verteilten Versionsverwaltungssysteme git und Mercurial.

  Systembedingt bieten verteilte Versionsverwaltungen keine Locking-Mechanismen. Da wegen der höheren Zugriffsgeschwindigkeit die Granularität der gespeicherten Änderungen viel kleiner sein kann, können sie sehr leistungsfähige, weitgehend automatische, Merge-Mechanismen zur Verfügung stellen.

2.12 Bestehende Infrastruktur bei msgGillardon

2.12.1 Die Entwicklungsumgebung Eclipse

Abb. 2.10: Eclipse Logo

In der heutigen Zeit lassen sich Softwareentwickler in den meisten Fällen durch IDEs unterstützen. Diese vereinen die Funktionalität eines Texteditors mit einer Dateiverwaltung, einer mächtigen Suchfunktion und meist speziell auf die verwendete Sprache zugeschnittenen Möglichkeiten wie Syntaxhervorhebung oder direkter Anzeige von unterstützender Dokumentation. Auch bieten die meisten IDEs einen leistungsfähigen Debugger, welcher den Entwickler bei der Fehlersuche im Programm unterstützt. So werden viele Schritte während der Programmierung eingespart oder erleichtert. Eclipse ist eine solche IDE.

Initial entwickelt wurde Eclipse für Java von der Firma IBM, die es im Jahr 2001 als OpenSource der Allgemeinheit zur Verfügung gestellt hat. Im Januar 2004 wurde die Eclipse Foundation gegründet, um die Entwicklung der IDE zu organisieren und voranzutreiben. Sie ist eine unabhängige, gemeinnützige Organisation, zuständig für die technische Infrastruktur, das IP Management[1], den Entwicklungsprozess und das „Eclipse Ecosystem“. Dieses beinhaltet unter anderem die Entwicklung von Marketingstrategien, die Organisation von Trainings und Weiterbildungen sowie Pressearbeit [vgl. Fou14b].

Die Entwicklungsumgebung Eclipse basiert auf einem Plug-in Konzept, das es ermöglicht, jegliche Art von Erweiterung in die IDE zu integrieren. Weil viele Plug-ins auch für andere Sprachen, wie zum Beispiel C++ existieren, wird Eclipse längst über das Java-Umfeld hinaus verwendet. Nicht nur in Bezug auf Programmiersprachen ist Eclipse erweiterbar. Zusätzlich gibt es Plug-ins für viele Anwendungsfälle aus dem alltäglichen Leben des Entwicklers. So werden zum Beispiel verschiedene Erweiterungen für die Integration von Versionsverwaltungssystemen angeboten, darunter Add-ons für Subversion oder Mercurial. Auch für ganz andere Bereiche der Softwareentwicklung, wie das Erstellen von Benutzungsoberflächen mit grafischen Editoren, existieren verschiedene Lösungen. Diese Vielfalt und die Möglichkeiten zur Erweiterung der IDE nach den eigenen Bedürfnissen machen Eclipse zu einem sehr mächtigen und erfolgreichen Werkzeug in der Softwareentwicklung.

2.12.2 Versionsverwaltung mit Mercurial

Mercurial ist ein plattformunabhängiges, verteiltes Versionsverwaltungssystem. Es ist ein Open-Source-Projekt und wurde im April 2005 von Matt Mackall initiiert. Mercurial hat eine weite Verbreitung in der Welt [vgl. Wik14e].

Abb. 2.11: Mercurial Logo

Zur Unterstützung der Arbeit mehrerer Entwickler an einer Codebasis gibt es mit Rhodecode einen Server, auf dem die gemeinsam genutzten Repositories abgelegt werden und bei welchem die Zugriffsrechte verwaltet werden können. Die Entwickler verbinden sich dann mit Mercurial-Clients auf diesen Server, um die Versionsstände miteinander abzugleichen. Für die Verwaltung auf dem lokalen Entwicklerrechner besitzt Mercurial ein Kommandozeilenprogramm, mit der TortoiseHg-Workbench steht aber auch ein Werkzeug mit grafischer Benutzungsoberfläche zur Verfügung. Eine spezielle Integration in die verbreitetsten Entwicklungsumgebungen existiert ebenfalls.

1 Verwaltung des geistigen Eigentums der OpenSource-Entwickler

2.12.3 Build-Management mit Apache Ant

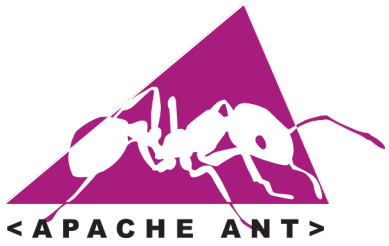


Abb. 2.12: Apache Ant Logo

Apache Ant ist ein sogenannter Task-Runner. So werden Programme bezeichnet, welche bestimmte Aufgaben (Tasks) in einer konfigurierbaren Reihenfolge abarbeiten. Er wird als Java-Bibliothek und als Kommandozeilenwerkzeug bereitgestellt. Hauptsächlich findet er in der Java-Entwicklung Verwendung. Apache Ant gilt als ein sehr flexibles Werkzeug, das keine Vorgaben bezüglich Programmier- oder Projektstruktur-Konventionen macht, wie es zum Beispiel bei Apache Maven der Fall ist. Zum Bewerkstelligen gängiger Anwendungsfälle in einem Build-Prozess wird eine große Anzahl an Bibliotheken geboten, die von den Benutzern verwendet werden können [vgl. Fou14a].

Der Build-Vorgang wird in Ant als Projekt beschrieben. Tasks sind ständig wiederkehrende Aufgaben, die von einer Bibliothek bereitgestellt werden. Die Entwicklung eigener Tasks ist zum Beispiel in Java möglich. Falls die vorhandenen Bibliotheken nicht ausreichen sollten, kann man außerdem eigene Tasks direkt in der XML-Datei eines Projektes in verschiedenen Skript-Sprachen, wie zum Beispiel Groovy, implementieren. Ein Build-Schritt wird in Ant als sogenanntes Target beschrieben. Innerhalb der Targets können dann die notwendigen Aktionen durchgeführt werden. Zusätzlich kann es Abhängigkeiten zwischen Targets geben, so dass vor dem eigentlich aufgerufenen Target zuerst andere aufgerufen werden, um beispielsweise initiale Aktionen auszuführen.

Listing 2.4 zeigt ein sehr einfaches Beispiel eines Ant-Projektes. Zunächst wird, wie in XML üblich, die verwendete XML-Version angegeben. Daraufhin beginnt der eigentliche Ant-Quelltext. Das Wurzelelement ist das `project`-Tag. Es definiert ein Build-Projekt mit einem Namen und einem Default-Target. Dieses wird, falls beim Aufruf nicht anders angegeben, bei der Ausführung zuerst aufgerufen. In der Abbildung ist es das Target `print.helloWorld`. Es wird definiert mit dem `target`-Tag und einem Namen. Das Target enthält den Quelltext, der beim Aufruf ausgeführt werden soll. In diesem Beispiel ist dies eine Ausgabe mit dem Task `echo`. Der Task

```
<?xml version="1.0"?>
<project name="HelloWorld" default="print.helloWorld">
  <target name="print.helloWorld">
    <echo message="Hello World!"/>
  </target>
</project>
```

Listing 2.4: Ein einfaches Build-Projekt in Apache Ant.

empfängt als Parameter eine `message` vom Typ String und gibt diese Zeichenkette auf der Konsole aus. Dieser Quelltext wird in einer Datei, standardmäßig `build.xml`, gespeichert und kann dann von Ant ausgeführt werden. Als Ergebnis würde bei der Ausführung die Nachricht „Hello World!“ auf der Konsole ausgegeben. Der Build eines Java-Programms besteht unter Umständen aus sehr vielen Build-Schritten, wovon einer die Kompilierung ist. Diese könnte mit dem Task `javac` bewältigt werden.

msgGillardon setzt Apache Ant primär für die folgenden Aufgaben ein:

- Checkout der benötigten Abhängigkeiten aus dem Apache Ivy-Repository.
- Kompilieren des Quellcodes und Ausführen der Tests.
- Speichern der Artefakte.

2.12.4 Jenkins

Bei der Bereitstellung eines zentralen Buildsystems, siehe Abschnitt 2.1, ist darauf zu achten, dass auch die damit verbundenen Aufgaben durchgeführt werden können. Dazu wird bei der msgGillardon AG die Software Jenkins verwendet. Sie ist der Nachfolger des Systems Hudson. Aufgrund von Namensrechten musste die Software nach einer Firmenübernahme umbenannt werden. Ursprünglich gedacht war das System für die kontinuierliche Integration von Java-Programmen. Durch die einfache Erweiterbarkeit der Software mittels Plug-ins ist es allerdings auch sehr einfach möglich, Projekte mit anderen Technologien wie C++, PHP oder Ruby zu verwalten [vgl. Wik14b].

Abb. 2.13: Jenkins Logo

In Jenkins werden Continuous-Integration-Vorgänge für verschiedene Projekte in sogenannten Jobs verwaltet. Diese Jobs können neben der manuellen Steuerung ebenfalls ereignis- und zeitbasiert gestartet werden. Manuell gesteuerte Jobs können von einem Entwickler selbst oder auch mittels eines API-Aufrufs ausgelöst werden. Ereignisbasierte Jobs werden bei definierten Vorfällen, etwa einer Änderung im Versionsverwaltungssystem, ausgelöst. Zeitgesteuerte Jobs werden regelmäßig, zum Beispiel nachts (Nightlybuild), gestartet und durchgeführt.

Ein Job besteht aus verschiedenen Build-Schritten. Diese können bei Jenkins frei definiert werden. Das folgende Beispiel soll eine solche Schrittfolge verdeutlichen. Hierbei ist zu beachten, dass die einzelnen Schritte vereinfacht formuliert sind. Es wird angenommen, dass der Build zeitgesteuert ausgelöst wird.

1. Lade den Quellcode aus dem Versionsverwaltungssystem.

2. Baue das Projekt mittels eines vorgegebenen Buildscripts (Ant-Skript).
3. Führe die Tests für das gebaute Projekt aus.
4. Stelle das gebaute Produkt als Archiv bereit.
5. Informiere die hinterlegten Entwickler per E-Mail über das Ergebnis.

Diese Schrittfolge kann bei einer großen Codebasis sehr lange dauern, da unter Umständen sehr große Datenmengen verarbeitet werden müssen. Für die Skalierbarkeit in großen Unternehmen mit vielen Teams und großen Projekten bietet Jenkins ein Master-Slave-Prinzip. Dies erlaubt die Verteilung der Jobs auf mehrere Server. So ist es möglich, dass viele Projekte auf dem gleichen Jenkins-Server verwaltet werden können, sich aber in der Performanz nicht gegenseitig einschränken, da sie auf eigenen Rechnern ausgeführt werden.

KAPITEL 3

Analyse

Die Analyse beinhaltet die Einarbeitung in das fachliche Umfeld, um einen möglichst genauen Einblick in den vorhandenen Entwicklungsprozess und die Vorgehensweisen der Entwickler bezogen auf die Datenmodellgenerierung zu erlangen. Dazu wird zunächst der bisherige Prozess analysiert und eventuelle Schwachstellen aufgezeigt, um daraufhin eine möglichst präzise Erarbeitung der Anforderungen zu ermöglichen.

Die Durchführung des nachfolgend beschriebenen Prozesses führt zu einem Grundgerüst der Persistenzschicht. Dieses setzt sich üblicherweise zusammen aus jeweils einem Java-Interface und einem Grundgerüst der Java-Klasse für jede modellierte Tabelle. Das Interface fungiert als JPA-`MappedSuperclass` und enthält einerseits grundlegende Informationen über die Attribute, z. B. Spaltenname der Tabelle, und andererseits Zugriffsmethoden auf die Attribute. Die Java-Klasse dient als persistente Klasse und enthält somit die Attribute und die Mapping-Informationen als JPA-Annotationen.

Der bisherige Prozess gliedert sich in drei bzw. vier Teilprozesse:

1. Klassendiagramm mit MagicDraw 11 erstellen und im XMI-Format speichern.
2. Aufruf des Ant-Targets:
   a) `pog2.oawgenmetainfo`, wenn Meta-Informationen erzeugt werden sollen, welche beispielsweise zur Erzeugung von DDL-Anweisungen benutzt werden.
   b) `pog2.oawgenpersclasses`, wenn die eigentliche Persistenzschicht erzeugt werden soll.
3. Durch das vorige Ant-Target wird die oAW Workflow Engine aufgerufen und generiert aus dem Klassendiagramm die gewünschten Dateien.
4. Falls Meta-Informationen erzeugt wurden, können anschließend durch das Ant-Target `pog2.gendbadminsql` DDL-Anweisungen für das msgGillardon eigene DBAdminTool erzeugt werden.

Abbildung 3.1 visualisiert den bisherigen Prozess.

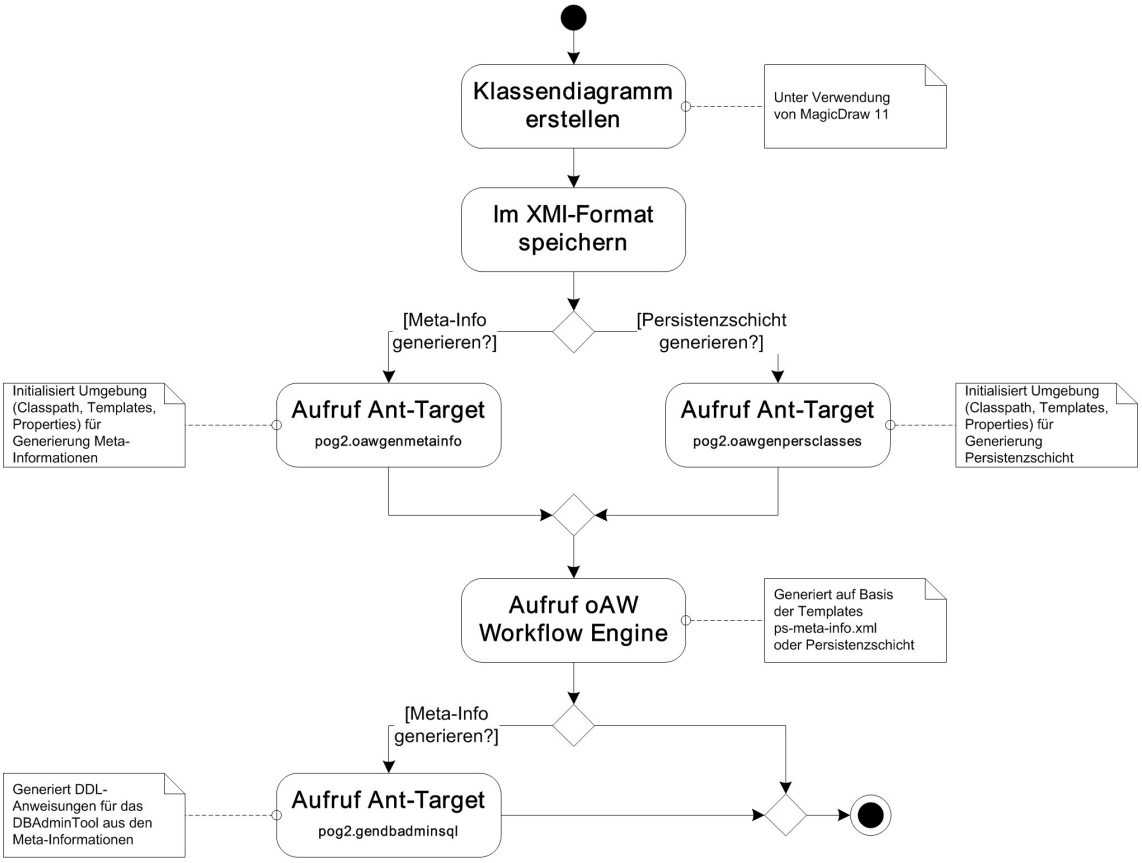


Abbildung 3.1: Bisheriger Prozess zur Generierung der Persistenzschicht

3.1 Klassendiagramm erstellen

Das Klassendiagramm wird mithilfe von MagicDraw 11 erstellt. Dafür wurde ein benutzerdefinierter Diagrammtyp namens PersistenceService erstellt, welcher auf der UML2 Notation basierend die folgenden Situationen abbilden kann:

- **Klassen**
- **Vererbung**
- **1:1-Beziehungen**
- **1:N-Beziehungen**
- **N:M-Beziehungen**
- **Kompositionen** beschreiben „abhängige Objekte“, welche automatisch gelöscht werden, sobald die Verbindung zu ihrem Parent erlischt.
- **Datenbankeigenschaften**

- **Eingebettete Klassen**
- **Typsichere Enumerationen**, aber keine Java-Enumerationen, da der bisherige Prozess vor der Einführung offizieller Java-Enumerationen (Java 5) entwickelt wurde.
- **Bottom-Up Mapping**: Das bedeutet, dass vorhandene Datenbanktabellen auf Klassen abgebildet werden können.
- **Benutzerdefinierte Methoden**: Eine angegebene Methode wird nur als Methodenrumpf, somit ohne Implementierung, hinzugefügt.

3.1.1 Klassen

Daten, die persistiert werden sollen, werden als Klasse mit dem Stereotyp `«persistent»` modelliert. Die zugehörigen Datenbankeigenschaften werden über Eigenschaften des Stereotyps in MagicDraw konfiguriert. Zu diesen gehört beispielsweise der Tabellenname.

Alle Attribute einer persistenten Klasse sind persistent. Über das optionale Stereotyp `«pAtt»` samt zugehöriger Eigenschaften kann das Datenbankmapping beeinflusst werden, z. B. Primärschlüssel oder Spaltenname.

3.1.2 Eingebettete Klassen

Eingebettete Klassen existieren in Java als eigene Klassen, allerdings nicht unabhängig, sondern immer nur in Form einer Komposition einer übergeordneten Klasse. In der Datenbank erhalten eingebettete Klassen keine eigene Tabelle. Stattdessen werden die Spalten in die Tabelle der übergeordneten Klasse aufgenommen. Sie werden als Klasse mit dem Stereotyp `«embedded»` modelliert.

3.1.3 Typsichere Enumerationen

Typsichere Enumerationen werden als Klasse mit dem Stereotyp `«enum»` modelliert. Die Werte der Enumeration werden als Attribute angegeben. Die Attribute müssen vom Typ `int` und statisch sein. Die Ordinaten werden dem Attribut direkt zugewiesen.

3.1.4 Beziehungen

Beziehungen zwischen Klassen werden über UML2-Assoziationen modelliert.

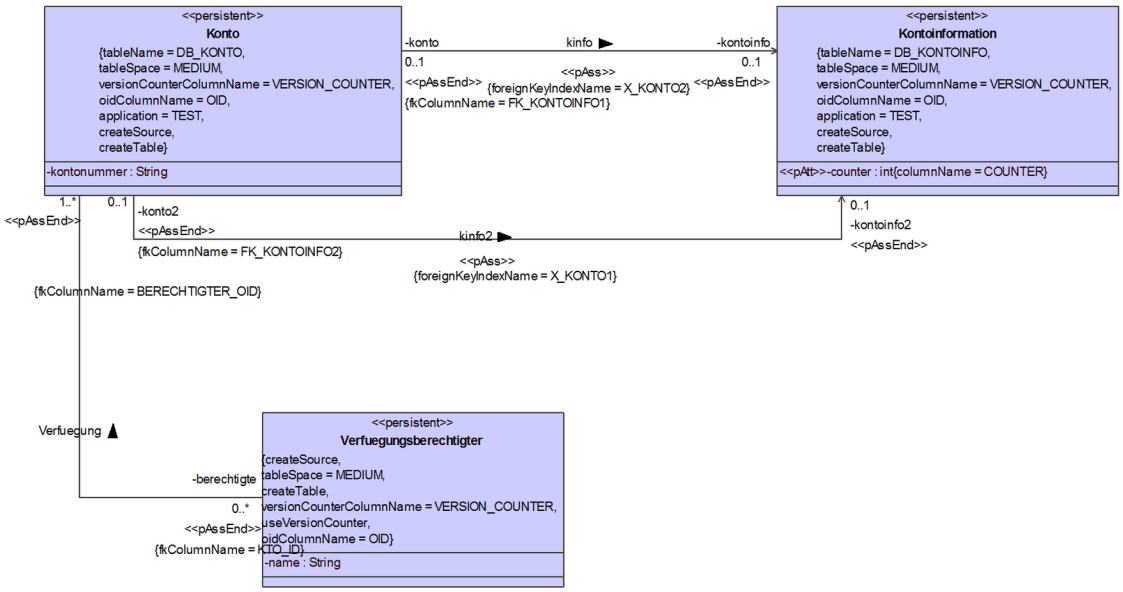


Abbildung 3.2: MagicDraw Klassendiagramm

3.2 pog2.oawgenmetainfo

Diesem Ant-Target werden das Klassendiagramm im XMI-Format und ein Ziel-Verzeichnis übergeben. Über optionale Parameter lassen sich noch weitere Optionen setzen, beispielsweise kann über den Parameter `includeFilter` festgelegt werden, dass nur bestimmte Java-Pakete generiert werden sollen. Mit diesen Eingabedaten und dem Template zur Metadaten-Generierung wird dann die oAW Workflow Engine aufgerufen. Das Ergebnis ist ein XML-Dokument, welches Meta-Informationen zu dem entsprechenden Datenmodell enthält. Es wird einerseits vom PersistenceService des Frameworks benötigt und andererseits werden daraus DDL-Anweisungen für das DBAdminTool generiert, siehe Abschnitt 3.5.

3.3 pog2.oawgenpersclasses

Diesem Ant-Target werden das Klassendiagramm im XMI-Format und ein Ziel-Verzeichnis übergeben. Über optionale Parameter lassen sich noch weitere Optionen setzen, beispielsweise kann über den Parameter `includeFilter` festgelegt werden, dass nur bestimmte Java-Pakete generiert werden sollen. Mit diesen Eingabedaten und dem Template zur Generierung der Persistenzschicht wird dann die oAW Workflow Engine aufgerufen. Das Ergebnis sind die generierten Klassen des Klassendiagramms, welche die Persistenzschicht bilden. Sie beinhalten spezielle Attribute und Methoden, die vom PersistenceService des Frameworks instrumentiert werden.

3.4 oAW Workflow Engine

Die oAW Workflow Engine generiert, abhängig vom übergebenen oAW-Template, aus der übergebenen XMI-Datei die gewünschten Daten. Um die Persistenzschicht aus den Informationen der XMI-Datei zu generieren, werden Xpand-Templates befüllt, die die jeweiligen Klassengrundgerüste enthalten.

3.5 pog2.gendbadminsql

In diesem Ant-Target werden aus den durch Abschnitt 3.2 vorhandenen Meta-Informationen DDL-Anweisungen generiert. Dafür werden zunächst die Meta-Informationen in ein Meta-Modell übertragen.

Anschließend werden die für das DBAdminTool notwendigen Skripte mit den entsprechenden DDL-Anweisungen erzeugt.

Das DBAdminTool ist ein Hilfsmittel der msgGillardon AG, um die Datenbankadministration für die verschiedenen Projekte zu vereinfachen. Hierzu bietet es unter anderem folgende Funktionalitäten:

- Anlage und Update des Datenmodells
- Ausführen von einzelnen Statements oder Skripten
- Reorganisation und das Erstellen von Statistiken
- Export von Tabelleninhalten

Dabei bietet das DBAdminTool den Vorteil, die herstellerspezifischen Syntaxen der unterstützten Datenbanksysteme so zu abstrahieren, dass der jeweilige Anwendungsentwickler unabhängig vom verwendeten Datenbanksystem entwickeln kann, da das Spezialisieren der SQL-Anweisungen auf das verwendete Datenbanksystem vom DBAdminTool übernommen wird. Dabei muss sich der Entwickler auf eine festgelegte Untermenge von Anweisungen beschränken.

3.6 Einschränkungen

Wie bereits in Abschnitt 1.2 angesprochen hat der bisherige Prozess ein paar Einschränkungen:

- **Keine parallele Bearbeitung**: msgGillardon arbeitet in Teams mit bis zu 20 Personen, welche sich wiederum in sogenannte Feature-Teams aufteilen. Durch diese vertikale Unterteilung wird eine hohe Parallelisierbarkeit auf allen Schichten benötigt. Fehlende Parallelisierbarkeit erfordert einen hohen Koordinationsaufwand zwischen den Feature-Teams.
- **Fehlende Branch-Fähigkeit**

- **Aufwendiges Update**: Das XMI-Format der Klassendiagramme unterscheidet sich bei den verschiedenen Versionen von MagicDraw. Dies hat zur Folge, dass bei einem Update von MagicDraw die Skripte zur Verarbeitung der Klassendiagramme angepasst werden müssen.
- **Zu großer Sprachumfang** ⇒ erhöhter Einarbeitungsaufwand
- **Hohe Turn-Around-Zeiten**: Diagramm in MagicDraw erstellen/ bearbeiten ↪ Aufruf des Ant-Targets ↪ Aufruf der oAW Workflow Engine durch Ant-Target ⇒ erhöhte Entwicklungszeit

KAPITEL 4

Konzeption

4.1 Anforderungen

Die Anforderungen an die zu erstellende Lösung ergeben sich direkt aus der Analyse. Es gilt, die bereits bestehenden Möglichkeiten ebenfalls zu unterstützen, die vorhandenen Einschränkungen nach Möglichkeit zu entfernen und dabei keine neuen Einschränkungen hinzuzufügen.

Das bedeutet, dass die Unterstützung folgender Möglichkeiten von der Sprache und dem generierten Java-Quellcode unterstützt werden müssen:

- Klassen
- Vererbung
- Beziehungen (1:1, 1:n, n:1, n:m)
- Kompositionen
- Datenbankeigenschaften
- Eingebettete Klassen
- Bottom-Up Mapping
- Benutzerdefinierte Methoden

Für die Einbettung in den bestehenden Build-Prozess muss die Möglichkeit bestehen, die Generierung über Apache Ant ansteuern zu können.

Um die vorhandene Lösung ablösen zu können, werden neben den reinen Java-Klassen der Persistenzschicht auch noch die Meta-Informationen für den Persistence-Service benötigt. Jedoch ist die Generierung der Meta-Informationen aus Zeitgründen nicht Bestandteil dieser Arbeit.

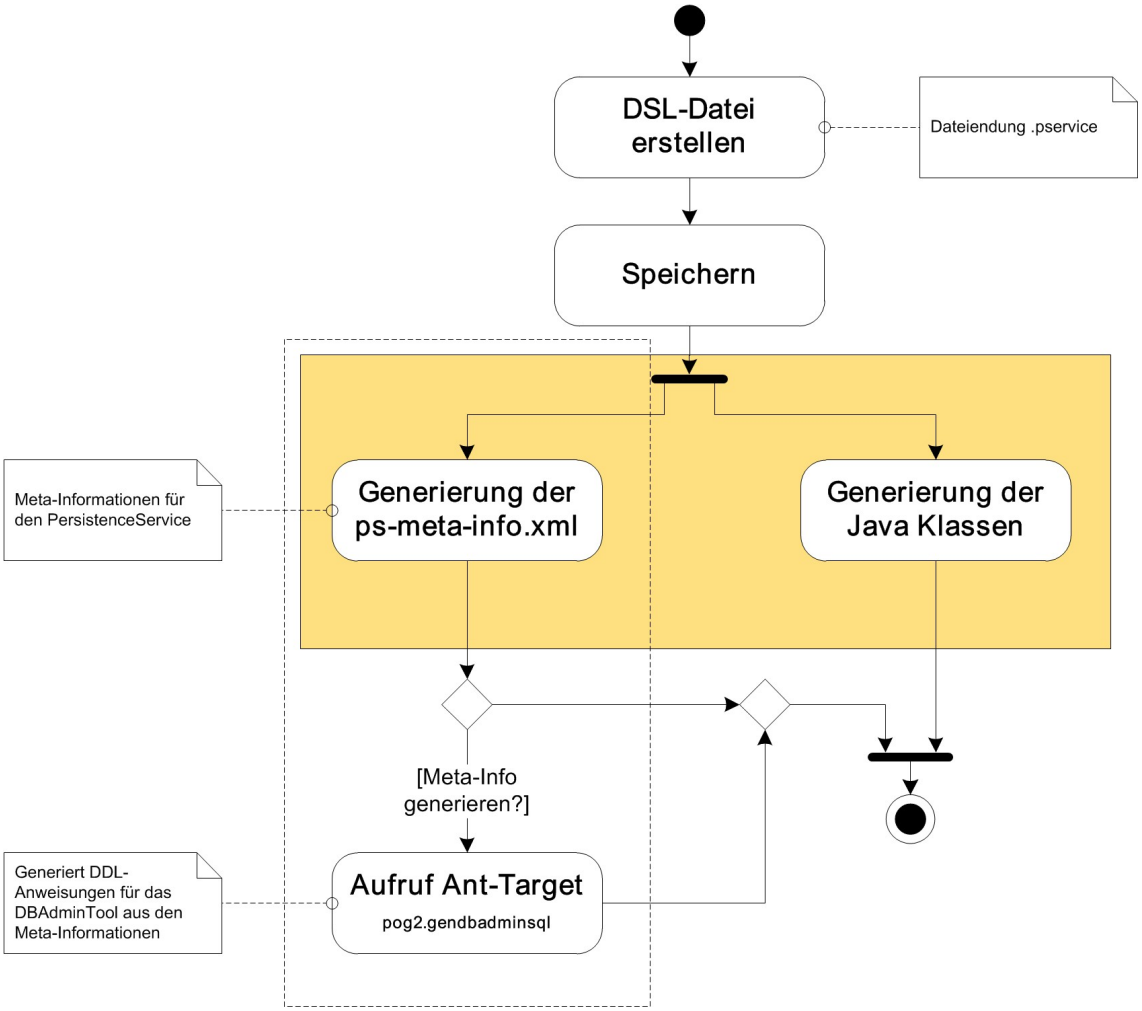


Abbildung 4.1: Planung neuer Prozess zur Generierung der Persistenzschicht

4.2 Neuer Prozess

Abbildung 4.1 zeigt die Planung des neuen Prozesses zur Generierung der Persistenzschicht. Der gelbe Kasten kennzeichnet dabei die Aktivitäten, die entweder automatisch vom erstellten Eclipse-Plugin nach dem Speichern der entsprechenden DSL-Datei ausgeführt werden oder aber über die angebotene Ant-Schnittstelle außerhalb von Eclipse getriggert werden können. Die gestrichelte Umrandung zeigt, welche Aktivitäten des Prozesses nicht Teil dieser Arbeit sind und somit zu einem späteren Zeitpunkt noch umgesetzt werden müssen.

Man kann erkennen, dass sich durch die Integration der Generierung in das Eclipse-Plugin die Turn-Around-Zeiten für den Anwendungsentwickler deutlich verkürzen, da die generierten Klassen direkt nach dem Speichern zur Verfügung stehen.

4.3 Die Sprache

Um eine möglichst intuitive und für Java-Entwickler leicht erlernbare Sprache zu entwickeln, wurde versucht, die Syntax möglichst an Java anzulehnen.

4.3.1 Dateiaufbau

Der grundlegende Dateiaufbau orientiert sich an dem Aufbau von Java-Klassen. Der Unterschied ist jedoch , dass es möglich ist, innerhalb einer Datei nicht nur eine Klasse zu erstellen, sondern alle Klassen desselben Pakets.

Zu Beginn erfolgt also die Paketdefinition mithilfe des Schlüsselworts `package`. Nachfolgend werden über die Importsektion Referenzen aus anderen Paketen importiert, wobei, wie bei Java, die Klassen des Pakets java.lang automatisch importiert sind. Anschließend können beliebig viele Entitäten (`entity`) und eingebettete Klassen (`embedded`) definiert werden. Listing 4.1 verdeutlicht den Dateiaufbau anhand eines Beispiels.

```
package de.pservice.beispiel.adresse;

embedded Ort {
    String plz;
    String ort;
}

entity Adresse {
    String strasse;
    Ort ort nullable;
}
```

Listing 4.1: Ein einfaches Beispiel in der DSL.

4.3.2 Eingebettete Klassen

Eingebettete Klassen werden durch das Schlüsselwort `embedded` eingeleitet und definieren eine Klasse, welche anschließend als Komposition von anderen Klassen aus genutzt werden kann. Der Rest der Syntax orientiert sich an Java und wird in Listing 4.2 aufgezeigt.

```
embedded <NAME> [extends <OBERKLASSE>] [implements <INTERFACE>[, <
↪ INTERFACE>]...] {
  [[FEATURE]...]
  [[METHODE]...]
}
```

Listing 4.2: Grundlegende Syntax einer eingebetteten Klasse.

4.3.3 Entitäten

Eine Entität entspricht einer zu persistierenden Java-Klasse, welche eine eigene Datenbanktabelle erhält. Eingeleitet wird die Definition einer Entität durch das Schlüsselwort `entity`. Wie man im Vergleich der Listing 4.2 und 4.3 erkennen kann, enthält eine Entität im Gegensatz zur eingebetteten Klasse Metadaten, um allgemeine Tabelleneigenschaften definieren zu können.

```
entity <NAME> [extends <OBERKLASSE>] [implements <INTERFACE>[, <INTERFACE
↪ >]...] {
  [[ENTITY_METADATEN]...]
  [[FEATURE]...]
  [[METHODE]...]
}
```

Listing 4.3: Grundlegende Syntax einer Entität.

Entitätsmetadaten

Über die Metadaten können verschiedene Eigenschaften für die ganze Entität und somit für die gesamte Datenbanktabelle festgelegt werden. Alle Metadaten sind optional. Tabelle 4.1 zeigt die unterstützten Metadaten auf.

4.3.4 Features

Feature ist der Oberbegriff für Attribute und Relationen. Features unterstützen ebenfalls die Möglichkeit, über Metadaten verschiedene Einstellungen zu treffen. Tabelle 4.2 zeigt die unterstützten Metadaten auf.

Attribute

Ein Attribut entspricht einem zu persistierenden Klassenattribut und somit einer Datenbankspalte. Listing 4.4 zeigt die grundlegende Syntax für Attribute.

```
[@AttributeOverrides(...)]
<Datentyp> <NAME>[= <Initialwert>] [[FEATURE_METADATEN]...];
```

Listing 4.4: Grundlegende Syntax eines Attributs.

Wie man erkennen kann, ist die Möglichkeit vorhanden, die Annotation `AttributeOverrides` zu nutzen. Diese Annotation wird für eingebettete Klassen unterstützt, um die Spaltennamen überschreiben zu können. Ansonsten führt die mehrfache Benutzung einer eingebetteten Klasse in einer Entität zu Konflikten, weil die Standardspaltennamen bereits durch die erste eingebettete Klasse belegt sind. Außerdem wird durch diesen Mechanismus die Anforderung des Bottom-Up-Mappings realisiert.

Relationen

Eine Relation entspricht einer Beziehung zwischen zwei Entitäten. Dabei sind die folgenden Ausprägungen möglich: 1:1, 1:n, n:1, n:m. Die Syntax innerhalb der DSL

Tabelle 4.1: Unterstützte Metadaten für Entitäten

Schlüsselwort	Wertebereich	Bedeutung
oidColumnName	String	Legt den Spaltennamen für den technischen Primärschlüssel fest; Standardwert: OID.
primaryKeyIndexName	String	Wird für die spätere Unterstützung des DBAdminTools benötigt.
tableName	String	Legt den Namen für die Datenbanktabelle fest; Standardwert: Entitätsname in Großbuchstaben.
entityName	String	Legt den Namen der Entität fest.
tableSpace	SMALL, MEDIUM, BIG	Wird primär für die spätere Unterstützung des DBAdminTools benötigt. Legt für die generierten Klassen fest, wieviele Schlüsselwerte vom Generator vorgehalten werden sollen; Standardwert: MEDIUM.
useVersionCounter		Falls gesetzt, wird ein zusätzliches Klassenattribut generiert, welches für das optimistische Locking des PersistenceServices benötigt wird.
versionCounterColumnName	String	Legt den Spaltennamen für das mit `useVersionCounter` angelegte Attribut fest; Standardwert: VERSION_COUNTER.

ist vom Typ der Relation abhängig, es wird zwischen `toOne`- (Listing 4.5) und `toMany`-Relationen (Listing 4.6) unterschieden. Bei `toMany`-Relationen kann der zu verwendende Typ von `Collection` beeinflusst werden.

```
[@JoinColumns(...)]
<oneToOne|manyToOne> <Entity> <NAME> [[FEATURE_METADATEN]...] [[
↪ RELATION_METADATEN]...];
```

Listing 4.5: Grundlegende Syntax einer toOne-Relation.

Tabelle 4.2: Unterstützte Metadaten für Features

Schlüsselwort	Wertebereich	Bedeutung
required		Legt fest, dass das Feature zwingend benötigt wird und somit beim Speichern nicht `NULL` sein darf.
primaryKey		Legt fest, dass dieses Feature als Primärschlüssel verwendet wird.
columnName	String	Legt den Namen für den Spaltennamen fest; Standardwert: <Featurename in Großbuchstaben>.
length	int	Legt die Maximallänge datenbankseitig für Strings und Blobs fest; Standardwert: 50.
converterClass	Java Klasse	Legt bei Java-Enumerationen die JPA 2.1 Converterklasse (siehe Abschnitt 5.4) fest.

Durch die Annotation `JoinColumns` wird dem Benutzer die Möglichkeit gegeben, eine benutzerdefinierte Abbildung für den Fremdschlüssel anzugeben. Wird diese Annotation benutzt, so wird der Standardmechanismus zur Generierung der entsprechenden Annotation umgangen und die benutzerdefinierte Annotation wird übernommen.

```
[@JoinColumns(...)|@JoinTable(...)]
<oneToMany|manyToMany> <Entity> <NAME> [CollectionType = <Collection|Map|
↪ Set>] [[FEATURE_METADATEN]...] [[RELATION_METADATEN]...];
```

Listing 4.6: Grundlegende Syntax einer toMany-Relation.

Bei `manyToMany`-Relationen muss über die Annotation `JoinTable` die Abbildung für die notwendige Join-Tabelle angegeben werden. Dieser Zwang wurde bewusst gewählt, da solche Relationen in den Framework-Anwendungen nur sehr vereinzelt auftreten und sich der Entwickler bewusst werden soll, dass es sich hierbei um einen seltenen Sonderfall handelt.

Relation-Metadaten Um die Relationen genauer zu konfigurieren, werden entsprechende Metadaten bereitgestellt, welche in Tabelle 4.3 aufgezeigt werden.

4.3.5 Methoden

```
public <Rückgabetyp> <Name> ([<Datentyp> <Name>[, <Datentyp> <Name>...]]);
```

Listing 4.7: Grundlegende Syntax einer Methode.

Tabelle 4.3: Unterstützte Metadaten für Relationen

Schlüsselwort	Wertebereich	Bedeutung
dependant		Gibt an, ob die Instanz der abhängigen Klasse gelöscht werden soll, wenn die assoziierte Instanz der anderen Klasse gelöscht wird.
fkColumnName	String	Legt den Namen für die JoinColumn-Annotation fest; Standardwert: FK_-<Entitätsname in Großbuchstaben>.
notFoundIgnore		Wenn eine Entität aus einer Beziehung nicht gefunden wird, weil die id aus der Fremdschlüsselspalte nicht existiert, wird typischerweise eine Exception geworfen. Wenn das Flag gesetzt wird, wird das Element ignoriert.
foreignKeyIndexName	String	Name des foreignKey-Index, wird für die DBAdminTool Unterstützung benötigt.
mappedBy	String	Legt fest über welches Feature des zugehörigen Beziehungspartners die Beziehung definiert wird.
notNull		Legt fest, dass die Relation nicht `NULL` sein darf.

Sowohl Entitäten als auch eingebettete Klassen können Methodendeklarationen beinhalten. Die grundlegende Syntax dieser Methoden wird in Listing 4.7 verdeutlicht. Wie zu erkennen ist, muss jede Methode `public` sein und es wird nur die Angabe der Methodensignatur unterstützt. Dies hat zur Folge, dass Klassen, die benutzerdefinierte Methoden besitzen, als abstrakte Klassen generiert werden. Die konkrete Klasse wird in diesem Fall durch den Anwendungsentwickler implementiert.

KAPITEL 5

Implementierung

5.1 Grammatik

In diesem Abschnitt soll der Aufbau der Grammatik aufgezeigt werden.

Xtext-Grammatiken beginnen immer mit einem Header, wie in Listing 5.1 gezeigt. Zuerst wird dort über das Schlüsselwort `grammar` der Name der Grammatik festgelegt. Der Name muss den Java-Kriterien für Klassen folgen. Das bedeutet, die Datei muss sich in dem über den Namen festgelegten Paketpfad befinden und den festgelegten Namen besitzen, die geforderte Dateiendung ist `xtext`. Grammatiken können, mithilfe des Schlüsselworts `with` gefolgt vom Namen der abzuleitenden Grammatik, von anderen Grammatiken abgeleitet werden. In unserem Fall wird von der XBase-Grammatik abgeleitet. Der Grund dafür wird in Abschnitt 5.3 genauer erläutert. Nach der Deklaration der Grammatik folgt die Generierungsanweisung des EMF Ecore-Modells. Ein Ecore-Modell ist ein Objektgraph, welcher während des Parsens von Xtext erstellt wird. Über das Ecore-Modell wird die Struktur der instantiierten Objekte beschrieben. Ecore-Modelle bestehen aus einem `EPackage`, welches wiederum `EClasses`, `EDataTypes` und `EEnums` beinhaltet. In Zeile 4 wird dann über die optionale `import`-Anweisung das Ecore-Modell um ein zusätzliches Modell erweitert. In diesem Fall wird die Grammatik um Regeln für grundlegende Java-Datentypen erweitert.

Nach dem Header folgt die oberste Regel. Es handelt sich dabei um eine Parser-Regel und sie ist nach dem Namen des Ecore-Modells benannt. Diese Regel stellt die Basis der DSL dar und regelt somit den Aufbau des gesamten Quellcodes.

Anschließend folgen die eigentlich Grammatikregeln. Dabei wird grundsätzlich zwischen *Lexing*- und *Parsing*-Regeln unterschieden.

Lexing-Regeln werden während der Lexing-Phase ausgewertet und liefern atomare EDataTypes zurück. Sie werden mit dem Schlüsselwort `terminal` eingeleitet. Nachfolgend kommt der Name, welcher nach der Konvention nur aus Großbuchstaben besteht. Zwischen „:“ und „;“ stehen dann die eigentlichen Bedingungen. Listing 5.2

```
grammar de.msggillardon.ps.pog3.PersistenceService with org.eclipse.xtext.
↪ xbase.annotations.XbaseWithAnnotations

generate persistenceService "http://www.gillardon.de/ps/pog3/
↪ PersistenceService"
import "http://www.eclipse.org/xtext/common/JavaVMTypes" as jvmTypes

persistenceService:
    package=PackageDeclaration
    importSection=XImportSection?
    (elements+=AbstractElement)*;
```

Listing 5.1: Xtext Grammatik Header mit oberster Regel.

zeigt eine einfache Regel zur Erkennung des Anweisungsendes durch ein „;“. Ohne weitere Angaben liefert eine Regel einen Wert vom Typ ecore::EString zurück. Für abweichende Rückgabetypen muss dieser mithilfe der Anweisung `returns` gefolgt vom entsprechenden Datentyp nach dem Regelnamen angegeben werden, wie in Listing 5.3 gezeigt. Lexing-Regeln werden generell in EBNF angegeben. Bei der Definition von Lexing-Regeln ist die Reihenfolge sehr wichtig, da sich Regeln gegenseitig überschreiben können und immer nur die neueste Regel gilt.

```
terminal EOL:
    ';';
```

Listing 5.2: Einfache Lexing-Regel.

```
terminal INT returns ecore::EInt:
    '0'..'9' ('0'..'9'|'_')*;
```

Listing 5.3: Lexing-Regel mit definiertem Rückgabewert aus der Xbase-Grammatik.

Die Ergebnisse der Lexing-Phase werden während der Parser-Phase mithilfe von Parser-Regeln ausgewertet. Im Gegensatz zu Lexing-Regeln liefern Parser-Regeln keine atomaren Datentypen zurück, sondern einen AST. Dieser Baum besteht aus den Variablen der Parser-Regeln. Beispielsweise wird nach dem Parsen der obersten Regel aus Listing 5.1 im AST der Knoten `package` aus einem Knoten vom Typ `PackageDeclaration` bestehen, welcher durch die Parser-Regel aus Listing 5.4 definiert wird.

```
PackageDeclaration:
    'package' packageName=QualifiedName EOL;
```

Listing 5.4: Parser-Regel für die Paketdeklaration.

Auch Parser-Regeln unterstützen Teile der EBNF, nämlich Gruppen, Alternativen, Schlüsselwörter und Regelaufrufe. Listing 5.5 zeigt die Parser-Regel für Attribute und wendet dabei viele der Möglichkeiten für Parser-Regeln an.

```
Attribute:
    (annotation = XAnnotation)?
    type=JvmTypeReference name=ValidID ('=' value=XExpression)? (
↪ attributeProperties+=FeatureProperties)*;
```

Listing 5.5: Parser-Regel für die Attribute.

Neben Parser-Regeln werden während der Parsing-Phase auch noch die folgenden Regeln ausgewertet:

- *Datentyp-Regeln*: Diese Regeln geben keinen AST zurück, sondern EDataType-Instanzen. Sie erfüllen also einen ähnlichen Zweck wie die Lexing-Regeln, aber im Gegensatz zu diesen sind sie kontextsensitiv und erlauben dem Benutzer die Nutzung von *versteckten Tokens*. Versteckte Tokens sind Tokens, also Resultate der Lexing-Phase, die semantisch nicht von Interesse sind, beispielsweise Kommentare. Sie werden durch das Konzept der versteckten Tokens vor den Parser-Regeln verborgen. Ein weiterer Unterschied zu Lexing-Regeln ist, dass sich Datentyp-Regeln nicht gegenseitig überdecken können. Listing 5.6 zeigt die Datentyp-Regel für den `QualifiedName`, welcher in Listing 5.4 verwendet wurde.
- *Enum-Regeln*: Enum-Regeln sind eine spezielle Form von Datentyp-Regeln. Sie liefern generell Strings zurück und bieten somit eine einfache Art, typsichere Werte anzubieten, welche sehr leicht validiert werden können. Listing 5.7 zeigt die Regel, welche die möglichen Collection-Typen für toMany-Relationen festlegt, die in Abschnitt 4.3.4 vorgestellt wurden.

```
QualifiedName:
  ValidID (=>'.' ValidID)*;
```

Listing 5.6: Datentyp-Regel für qualifizierte Namen.

```
enum CollectionType:
    Collection | Map | Set;
```

Listing 5.7: Enum-Regel für Collection-Typ.

Ist die Grammatik vollständig, werden über einen MWE2-Workflow aus der Grammatik das Ecore-Modell, die ANTLR-Grammatik, eine Klassenhierarchie für den AST und verschiedene Xtend-Stubs erzeugt.

5.2 Validatoren

Da es sehr aufwendig sein kann, alle Restriktionen der DSL über die Grammatik festzulegen, unterstützt Xtext das Konzept von Validatoren. Mit Validatoren ist es möglich, die Freiheiten, die die Grammatik bietet, nachträglich einzuschränken. Beispielsweise erlaubt die Grammatik für Attribute aus Listing 5.5, dass oberhalb der eigentlichen Attributdefinition jegliche Art von `XAnnotation`[1] erlaubt ist. Aber wie in Abschnitt 4.3.4 bereits dargestellt, wird nur die Annotation `AttributeOverrides` unterstützt und dies nur bei der Verwendung von eingebetteten Klassen. Diese Restriktion wird durch Validatoren umgesetzt. Dafür wird von Xtext eine Xtend-Klasse, welche von `AbstractPersistenceServiceValidator` erbt, bereitgestellt. Innerhalb dieser Klasse können Methoden erstellt werden, welche mit der Annotation `org.eclipse.xtext.validation.Check` automatisch während des Generierungsprozesses aufgerufen werden. Diesen Methoden werden Sprachkomponenten übergeben und sie überprüfen dann, ob gewünschte Bedingungen der Sprache eingehalten werden.

Die vorher erwähnte Einschränkung, nur die Annotation `AttributeOverrides` und diese wiederum nur bei eingebetteten Klassen zuzulassen, wird durch zwei Validatorregeln umgesetzt. Die erste Regel überprüft, dass nur festgelegte Annotationen verwendet werden und wird in Listing 5.8 gezeigt.

```
public static val NOT_PERMITTED_ANNOTATION = "de.msggillardon.ps.pog3.
↪ persistenceService.notPermittedAnnotation"

@Check
def checkOnlyPermittedAnnotationsAreUsed(Feature feature) {
  if (feature.annotation != null && !(
    feature.annotation.annotationType.identifier == "javax.persistence.
↪ AttributeOverrides" ||
    feature.annotation.annotationType.identifier == "javax.persistence.
↪ JoinColumns" ||
    feature.annotation.annotationType.identifier == "javax.persistence.
↪ JoinTable"
  )) {
    error(
      "Es sind nur die Annotationen javax.persistence.AttributeOverrides,
↪ javax.persistence.JoinColumns und javax.persistence.JoinTable gestattet
↪ !",
      PersistenceServicePackage.Literals::FEATURE__ANNOTATION,
      NOT_PERMITTED_ANNOTATION
    )
  }
}
```

Listing 5.8: Validatorregel, dass nur zugelassene Annotationen benutzt werden.

1 XAnnotation ist eine Parser-Regel von Xbase, die Java-Annotationen zulässt.

Die zweite Regel (Listing 5.9) überprüft zuerst, ob das übergebene `Feature` eine Annotation hat. Anschließend wird validiert, ob es sich bei dem Feature um eine Relation handelt. Sollte dem so sein, wird geprüft, ob es sich dabei um die zu überprüfende Annotation handelt. Ist das der Fall, so wird mit der Methode `error` ein entsprechender Fehler generiert.

Ist das Feature ein Attribut, so wird überprüft, ob es sich dabei um die Annotation `AttributeOverrides` handelt und ob der Datentyp des Attributs keine eingebettete Klasse ist. Wenn es sich bei dem Datentyp um keine eingebettete Klasse handelt, wird abermals ein entsprechender Fehler generiert.

```
public static val ANNOTATION_ATTRIBUTE_OVERRIDES = "de.msggillardon.ps.
↪ pog3.persistenceService.annotationAttributeOverrides"

@Check
def checkAnnotationAttributeOverridesOnlyWithEmbedded(Feature feature) {
  if (feature.annotation != null) {
    if (feature instanceof Relation &&
      feature.annotation.annotationType.identifier == "javax.persistence.
↪ AttributeOverrides") {
      error(
        "Die Annotation javax.persistence.AttributeOverrides ist nur bei
↪ eingebetteten Klassen gestattet!",
        PersistenceServicePackage.Literals::FEATURE__ANNOTATION,
        ANNOTATION_ATTRIBUTE_OVERRIDES
      )
    } else if (feature instanceof Attribute &&
      feature.annotation.annotationType.identifier == "javax.persistence.
↪ AttributeOverrides" &&
      ((feature as Attribute).type.type as JvmGenericType).annotations.
↪ filter(a|a.annotation.identifier == "javax.persistence.Embeddable").
↪ size == 0) {
      error(
        "Die Annotation javax.persistence.AttributeOverrides ist nur bei
↪ eingebetteten Klassen gestattet!",
            PersistenceServicePackage.Literals::FEATURE__ANNOTATION,
            ANNOTATION_ATTRIBUTE_OVERRIDES
      )
    }
  }
}
```

Listing 5.9: Validatorregel für Attribut Annotationen.

Neben der Möglichkeit, Fehler zu erzeugen, besteht natürlich ebenfalls die Unter-

stützung, Warnungen und Informationen zu generieren. Im einfachsten und in den Listings auch benutzten Fall nehmen all diese Methoden als ersten Parameter einen String für die Meldung und als zweiten Parameter ein `EStructuralFeature` für die Meldungsposition entgegen.

5.3 Generierungsprozess

Xtext unterstützt in der aktuellen Version zwei Möglichkeiten zur Generierung von Java-Quellcode.

5.3.1 Templatebasierte Generierung

Die templatebasierte Generierung ist der Universalansatz. Durch die in Xtend integrierte Templatesprache, welche auch schon bei Xpand benutzt wurde, ist es möglich, jedwede Art von Text zu generieren. Diese Möglichkeit birgt aber ein ebenso großes Risiko, da keinerlei Validierung des Templatetextes stattfinden kann.

Deshalb wird dieser Generierungsprozess nur für alle Sprachen außer Java von Xtext empfohlen. Die Ursache dafür ist, dass da für die Nutzung der Xbase-Erweiterung der API-basierte Ansatz gewählt werden muss, damit auch die selbst erstellten Klassen (Entitäten bzw. Eingebettete Klassen) ebenfalls als mögliche Java-Referenzen erkannt werden.

Aus diesem Grund fiel die Entscheidung gegen diesen Ansatz für die Generierung der Persistenzklassen. Bei der Erweiterung dieser Arbeit um die Generierung der in Abschnitt 3.2 erwähnten Meta-Informationen ist dieser Ansatz jedoch das Mittel der Wahl.

5.3.2 API-basierte Generierung

Die vom Xtext-Projekt propagierte Lösung zur Erstellung von Java-Klassen ist die API-basierte Generierung. Dieser Ansatz ist auch der Richtige, wenn man Xbase nutzen möchte. Xbase ist ein Unterprojekt von Xtext und erweitert die Grammatik um einige im Javaumfeld weit verbreitete Datentypen, wie beispielsweise die Unterstützung für eine Importsektion, Annotationen oder Java-Referenzen als Datentypen.

Nutzt man Xbase, so wird durch den MWE2 Workflow automatisch eine Xtend-Klasse generiert, die den `AbstractModelInferrer` implementiert. Innerhalb dieser Klasse werden dann für alle Elemente, aus denen Java-Klassen generiert werden sollen, Methoden erstellt. Diese Methoden werden dann während der Generierung automatisch mit den entsprechenden Daten des Abstract Syntax Trees (ASTs) aufgerufen.

Da im Rahmen dieser Arbeit Java-Quellcode generiert werden soll und zur Vereinfachung an verschiedenen Stellen die Erweiterungen von Xbase verwendet werden können, fiel die Entscheidung auf den API-basierten Ansatz. Abbildung 5.1 zeigt die Zusammenhänge der Generatorenstruktur. Wie man erkennen kann, besitzt jedes

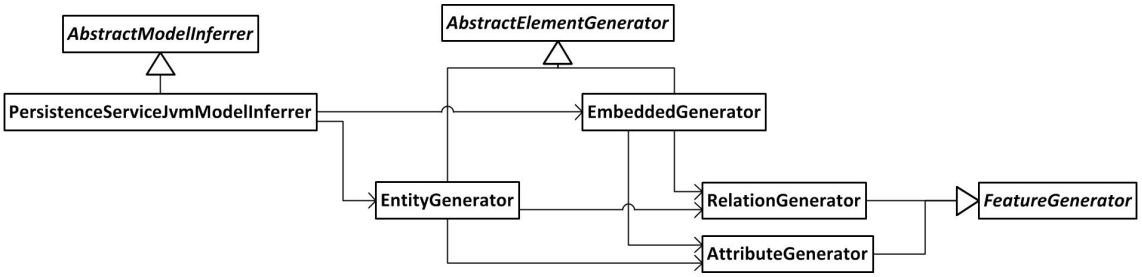


Abbildung 5.1: Klassenhierarchie Generierung

Element der Sprache[1] eine eigene Generatorklasse. Dort wird für den jeweiligen AST-Knoten der Java-Quellcode über entsprechende API-Aufrufe generiert. Einstiegspunkt für die Generierung ist die Klasse `PersistenceServiceJvmModelInferrer`, welche zwei `infer`-Methoden zur Verfügung stellt, eine für Entitäten und eine für eingebettete Klassen, in denen der jeweilige Generator aufgerufen wird. Listing 5.10 zeigt die `generate`-Methode der Klasse `AbstractElementGenerator`. An dieser Stelle wird sehr stark von der in Abschnitt 2.10.1 erklärten Technik der Erweiterungsmethoden Gebrauch gemacht. Die Zuweisungen von `fileHeader` und `documentation` sind eigentlich Aufrufe entsprechender Setter der Klasse `JvmTypesBuilder`. Die nachfolgende Zuweisung von `abstract` hingegen ruft die zugehörige Set-Methode des Objekts `it` auf. Auch alle weiteren Aufrufe der Generierungsmethoden nutzen diese Technik und rufen Methoden innerhalb des `AbstractElementGenerators` auf.

```
@Inject protected extension JvmTypesBuilder
@Inject protected extension Utils
@Inject protected extension AttributeGenerator
@Inject protected extension RelationGenerator

protected def generate(AbstractElement element, JvmGenericType it,
↪ JvmTypeReferenceBuilder typeReferenceBuilder) {
  fileHeader = JAVADOC_HEADER
  documentation = element.documentation
  abstract = element.methods.size > 0
  element.generateSupertypes(it, typeReferenceBuilder)
  for (method : element.methods) {
    method.generate(it)
  }
  element.generateGetterAndSetter(it)
}
```

Listing 5.10: Generatormethode der Klasse AbstractElementGenerator.

1 Die Methodengenerierung befindet sich aufgrund ihrer Simplizität innerhalb der Klasse `AbstractElementGenerator`.

5.4 Java-Enum Unterstützung

Standardmäßig unterstützt die JPA Java Enumerationen durch die Annotation `@Enumerated`, mit welcher konfiguriert werden kann, ob Enumerationen mit ihrem Ordinalwert oder mit ihrem Stringwert in der Datenbank gespeichert werden.

Bei der Speicherung der Ordinalwerte, die auch dem Standardwert der Annotation entspricht, besteht die Problematik, dass sich die Ordinalwerte verändern können, wenn man die Werte der Enumeration umsortiert. Die Speicherung der Stringwerte birgt das Risiko, dass eine Umbenennung das entsprechende Mapping bricht.

Mit der Version 2.1[1] der JPA werden sogenannte Converter-Klassen für Enumerationen unterstützt. Converter-Klassen implementieren die Schnittstelle `javax.persistence.AttributeConverter<X, Y>`, wobei X der Java Enumerationsklasse und Y dem Datentyp der Datenbankspalte entspricht. Diese Schnittstelle deklariert die folgenden zwei Methoden [vgl. Ora14]:

1. `Y convertToDatabaseColumn(X attribute)`: Konvertiert den Wert der Enumeration in den entsprechenden Wert für die Datenbankspalte.
2. `X convertToEntityAttribute(Y dbData)`: Konvertiert den Wert der Datenbankspalte in den entsprechenden Wert der Enumeration.

Genutzt werden diese Converter-Klassen dann, indem in der Klasse, die Werte einer Java Enumeration verwendet, über die Annotation `@Convert(converter = <Converter>.class)` die entsprechende Converter-Klasse konfiguriert wird. Die automatische Konvertierung zwischen Enumerationswert und Datenbankwert wird danach automatisch durch die entsprechende JPA-Implementierung sichergestellt.

Da der Weg über die Converter-Klassen die oben genannten Nachteile der bisherigen JPA-Lösung wettmacht, wurde beschlossen, diesen Ansatz auch durch die zu erstellende DSL zu unterstützen. Dazu kann in der DSL bei Enumeration der Wert, der einer solchen Converter-Klasse zugewiesen werden soll, über die Eigenschaft `converterClass` konfiguriert werden. Dadurch wird bei der Generierung die entsprechende `@Convert`-Annotation erstellt.

5.5 Tests

Xtext unterstützt JUnit-Tests auf allen Ebenen. Vom Parser über die Validatoren bis hin zur Generierung lassen sich alle Komponenten testen. Sogar die UI-Komponenten des Eclipse-Plug-ins können getestet werden, beispielsweise Code-Completion. Im Folgenden soll etwas genauer auf die Tests für Parser und Validatoren eingegangen werden.

1 Die JPA 2.1 Spezifikation wurde am 2. April 2013 veröffentlicht. [vgl. Gro14]

5.5.1 Parsertests

Mit Parsertests wird versucht zu überprüfen, ob die Grammatik nur das Gewünschte akzeptiert. Es handelt sich dabei um Positivtests. Man sollte, wie bei JUnit-Tests üblich, immer nur einen Aspekt überprüfen und den Test so einfach wie möglich halten. Das bedeutet, dass man bei Parsertests zuerst den zu überprüfenden Quellcode mithilfe der Klasse `ParseHelper` parst. Als Rückgabewert bekommt man den geparsten AST, welcher dann anschließend mit den JUnit-üblichen `assert`-Methoden überprüft wird.

Mithilfe der Klasse `ValidationTestHelper` erhält man zusätzlich noch die Möglichkeit sicherzustellen, dass während des Parsens keine Fehler auftraten. Listing 5.11 zeigt einen einfachen Test, der die Paketdeklaration überprüft. Diese entspricht der kleinsten gültigen Quellcodemenge der DSL.

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(PersistenceServiceInjectorProvider))
class PersistenceServiceParserTest {

  @Inject extension ParseHelper<persistenceService>
  @Inject extension ValidationTestHelper

  @Test
  def void testParsingPackage() {
    val model = '''
      package test.de.msggillardon.ps.pog3;
    '''.parse

    model.assertNoErrors

    assertEquals("test.de.gillardon.ps.pog3", model.package.packageName)
  }
}
```

Listing 5.11: Parsertest

5.5.2 Validatortests

Während es bei den Parsertest darauf ankommt, dass das Parsen fehlerfrei funktioniert, ist es bei den Validatortests genau umgekehrt. Es handelt sich um Negativtests. Aufgabe von Tests dieser Kategorie ist es, die erstellten Validatorregeln zu überprüfen und dadurch Warnungen und Fehler zu provozieren. Die generelle Vorgehensweise ändert sich dabei natürlich nicht. Zuerst wird Quellcode, welcher die gewünschte Warnung bzw. den gewünschten Fehler erzeugen soll, geparst. Anschließend wird mithilfe der Klasse `ValidationTestHelper` mit der Methode `assertWarning` bzw.

`assertError` sichergestellt, dass der gewünschte Effekt durch den Validator ausgelöst wurde. Listing 5.12 soll diese Vorgehensweise veranschaulichen. In dem dort gezeigten Test wird die aus Listing 5.8 bekannte Validatorregel überprüft, in welcher sichergestellt wird, dass nur erlaubte Annotationen benutzt werden.

```
@RunWith(typeof(XtextRunner))
@InjectWith(typeof(PersistenceServiceInjectorProvider))
class PersistenceServiceParserTest {

  @Inject extension ParseHelper<persistenceService>
  @Inject extension ValidationTestHelper

  @Test
  def void testParsingPackage() {
    '''
      package test.de.msggillardon.ps.pog3;

      import javax.persistence.Transient;

      entity TestEntity {
        @Transient
        String transient;
      }
    '''.parse
    .assertError(PersistenceServicePackage.eINSTANCE.feature,
↪ PersistenceServiceValidator.NOT_PERMITTED_ANNOTATION,
    "Es sind nur die Annotationen javax.persistence.AttributeOverrides,
↪ javax.persistence.JoinColumns und javax.persistence.JoinTable gestattet
↪ !"
  }
}
```

Listing 5.12: Validatortest

5.6 Syntaxhervorhebung

Die Syntaxhervorhebung ist ein Teil des späteren Eclipse-Plugins. Es besteht aus drei Komponenten:

1. der Definition von Text-Konfigurationen
2. der tokenbasierten Hervorhebung
3. der semantischen Hervorhebung

Text-Konfigurationen werden in einer Klasse definiert, die die Schnittstelle `IHighlightingConfiguration` implementiert. Über diese Schnittstelle wird die Methode `void configure(IHighlightingConfigurationAcceptor acceptor)` bereitgestellt. In dieser werden die bereitzustellenden Konfigurationen innerhalb des `IHighlightingConfigurationAcceptor`s mithilfe von eindeutigen Codes hinterlegt. Über Text-Konfigurationen können verschiedene Stile definiert werden, welche dann von den Hervorhebungsdefinitionen verwendet werden können. Für jeden Stil lassen sich Textauszeichnungen (fett, kursiv), Schrift- und Hintergrundfarbe festlegen. Listing 5.13 zeigt eine solche Konfiguration, welche für die Auszeichnung von Schlüsselwörtern verwendet wird. Das Resultat ist eine fettgedruckte purpurne Schrift, die der Standardauszeichnung von Eclipse für Java-Schlüsselwörter entspricht.

```
public TextStyle keywordTextStyle() {
  TextStyle textStyle = defaultTextStyle();
  textStyle.setColor(new RGB(127, 0, 85));
  textStyle.setStyle(SWT.BOLD);
  return textStyle;
}
```

Listing 5.13: Text-Konfiguration für Schlüsselwörter.

Die tokenbasierte Hervorhebung wird für Quellcodeteile verwendet, die sich auf Tokens des generierten ANTLR-Parsers zurückführen lassen. Das bedeutet, dass sich darüber die Auszeichnung für jegliche Schlüsselwörter und Lexing-Regel-Entsprechungen steuern lässt. Um eine solche Hervorhebung zu realisieren, muss die Klasse von `AbstractAntlrTokenToAttributeIdMapper` erben und die Methode `calculateId(String tokenName, int tokenType)` implementiert werden. Diese Methode wird nach jedem Tastendruck für jeden gefunden Token aufgerufen und sollte daher möglichst effizient implementiert werden. Als Rückgabewert liefert sie den Code einer Text-Konfiguration zurück, die für die Auszeichnung verwendet werden soll.

Die semantische Hervorhebung wird asynchron im Hintergrund ausgeführt und somit kann der Nutzer teilweise eine kurze Verzögerung feststellen, bis die darüber konfigurierten Auszeichnungen angezeigt werden. Für die semantische Hervorhebung wird die Schnittstelle `ISemanticHighlightingCalculator` bereitgestellt, welche die Methode `provideHighlightingFor(XtextResource, IHighlightedPositionAcceptor)` bereitstellt. Dieser Methode wird die aktuelle XtextResource übergeben, über welche man an das Ergebnis des Parsens, den AST, kommt. Durch diesen Baum lässt sich dann anschließend iterieren und die semantischen Knoten, welche ausgezeichnet werden sollen, entsprechend hervorheben. Listing 5.14 zeigt die Implementierung einer solchen semantischen Hervorhebung am Beispiel von Entitätsmetadaten. Wie zu erkennen ist, kann es vorkommen, dass Teile der übergebenen `XtextResource` `null` sein können. Dieser Fall kann eintreten, wenn es während des Parsens zu Fehlern kommt.

```
public void provideHighlightingFor(XtextResource resource,
↪ IHighlightedPositionAcceptor acceptor) {
  if (resource == null || resource.getParseResult() == null|| resource.
↪ getParseResult().getRootASTElement() == null)
    return;

  INode root = resource.getParseResult().getRootNode();
  for (INode node : root.getAsTreeIterable()) {
    if (node.getSemanticElement() instanceof EntityMetaData) {
      acceptor.addPosition(node.getOffset(), node.getLength(),
↪ PersistenceServiceHighlightingConfiguration.ENTITY_METADATA);
      }
    }
  }
```

Listing 5.14: Implementierung der semantischen Hervorhebung am Beispiel der Entitätsmetadaten.

KAPITEL 6

Bewertung und Vergleich

6.1 Bewertungskriterien

Die in Abschnitt 4.1 beschriebenen Anforderungen entsprechen den Musskriterien, die eine Lösung zu erfüllen hat, damit sie überhaupt in Betracht gezogen werden kann. Sie werden somit sowohl von der bisherigen als auch von der neu erstellten Lösung erfüllt.

Um aber darüber hinaus die Lösungen miteinander vergleichen zu können, wurden die folgenden Kriterien festgelegt:

- **Parallele Bearbeitung**: Hiermit wird die Branch-/ Diff-/ Mergebarkeit zweier Versionen bewertet, also wie leicht sich die Unterschiede zweier Versionen möglichst automatisiert feststellen und vereinen lassen.
- **Sprachumfang**: Ein zu großer Sprachumfang erhöht die Einarbeitungszeit und macht die Benutzung unnötig kompliziert.
- **Turn-Around-Zeiten**: Hiermit wird die Zeit bewertet, die benötigt wird, um vom erstellten Persistenzmodell zur fertig generierten Java-Persistenzschicht zu gelangen.
- **Übersichtlichkeit**: Hiermit wird bewertet, wie gut sich, vor allem größere, Modelle überblicken lassen. Dafür ist es von Vorteil, wenn sich Teile des Modells ausblenden lassen.
- **Einarbeitungszeit**: Hiermit wird die geschätzte Einarbeitungszeit eines Java-Entwicklers bewertet. Dabei wird davon ausgegangen, dass grundlegende Kenntnisse von UML-Klassendiagrammen vorhanden sind.
- **Erweiterbarkeit**: Hiermit wird bewertet, wie gut sich eine Lösung um neue Aspekte erweitern lässt.
- **Wartbarkeit**: Hiermit wird bewertet, wie gut sich eine Lösung warten lässt.

Teil der Wartung ist es beispielsweise, die Lösung auch für zukünftige Betriebssysteme und Java-Versionen bereitzustellen.

Die Bewertung basiert auf dem deutschen Schulnotensystem. Somit entspricht der Wertungsbereich 1 - 6, wobei 1 der besten Wertung „sehr gut“ und 6 der schlechtesten Wertung „ungenügend“ entspricht.

6.2 Gegenüberstellung beider Lösungen

Tabelle 6.1 stellt beide Lösungen nach den festgelegten Kriterien gegenüber. Es ist erkennbar, dass die neue Lösung in fast allen Kriterien der bestehenden Lösung überlegen ist. In der Übersichtlichkeit ist die bestehende Lösung, durch Nutzung der grafischen Darstellung, der neuen textbasierten Lösung überlegen.

Tabelle 6.1: Gegenüberstellung beider Lösungen

Kriterium	**Bestehende Lösung**	**Neue Lösung**
Parallele Bearbeitung	6	2
Sprachumfang	3	1
Turn-Around-Zeiten	4	2
Übersichtlichkeit	2	4
Einarbeitungszeit	3	2
Erweiterbarkeit	3	2
Wartbarkeit	3	2

KAPITEL 7

Fazit und Ausblick

Ziel dieser Arbeit war die Konzeption und Realisierung einer textuellen DSL, um die Generierung der Persistenzschicht innerhalb von ASF1-Projekten produktiver zu gestalten. Dabei sollte Xtext zum Einsatz kommen, da es eine gute Eclipse-Integration bietet und Eclipse die Standard IDE der msgGillardon AG in diesem Umfeld ist.

Die in Abschnitt 4.3 beschriebene Sprache wurde über mehrere Iterationen erarbeitet und immer wieder verbessert. Der praktische Teil dieser Arbeit erfüllt alle geforderten Musskriterien. Dabei wurden die folgenden Komponenten realisiert:

- Xtext-Grammatik, die die grundlegende Sprachspezifikation unterstützt.
- Validatoren, um die Grammatik der Spezifikation entsprechend einzuschränken.
- Eclipse-Plugin mit auf die erstellte DSL bezogenen Features:
  - Rudimentäre Quellcodevervollständigung
  - Syntaxhervorhebung
  - Quellcodeformatierer
  - Generierung von Java-Klassen
- Ansteuerung der Generierung über Apache Ant.
- Testfälle zur Validierung der Lösung.
- Migration eines Testprojekts zur Validierung der generierten Klassen.

Wie in Abschnitt 6.2 dargestellt, wird die bestehende Lösung in fast allen definierten Bewertungskriterien übertroffen. Zukünftig ließe sich durch Integration von EuGENia[1]

1 EuGENia ist ein Werkzeug zur automatischen Generierung von für GMF-Editoren notwendigen Dateien über ein Ecore-Model [vgl. Eps14].

der DSL-basierte Ansatz um eine grafische Ansicht ergänzen, um dadurch eine höhere Übersichtlichkeit zu erreichen.

Es ist festzuhalten, dass Xtext sehr viel Komfort bei der Erstellung von textuellen DSLs bietet und so qualitativ hochwertige Lösung mit relativ geringem Aufwand ermöglicht, wenn man den Aufwand der Bereitstellung entsprechender IDE-Unterstützung mit einbezieht.

Für einen Produktiveinsatz muss die Generierung der Meta-Informationen nachgerüstet werden. Außerdem sollte für die Umstellung bestehender Projekte ein entsprechender Konverter erstellt werden, der die XMI-Dateien bestehender Klassendiagramme in die DSL konvertiert.

Literaturverzeichnis

[AG14] AG, msgGillardon: *Geschichte*. msgGillardon AG. 2014. URL: http://www.msg-gillardon.de/unternehmen/unterseiten/unternehmen/geschichte.html (besucht am 15. 09. 2014) (siehe S. 1).

[Chr07] Christian Bauer, Gavin King: *Java Persistence mit Hibernate*. 2. Aufl. HANSER, 2007 (siehe S. 6, 7).

[Eff13] Efftinge, Sven: *sven efftinge's blog: Five good reasons to port your code generator to Xtend*. Juni 2013. URL: http://blog.efftinge.de/2013/06/five-good-reasons-to-port-your-code.html (besucht am 24. 09. 2014) (siehe S. 10).

[Eps14] Epsilon: *EuGENia GMF Tutorial*. The Eclipse Foundation. Dez. 2014. URL: http://eclipse.org/epsilon/doc/articles/eugenia-gmf-tutorial/ (besucht am 30. 12. 2014) (siehe S. 47).

[Fou14a] Foundation, The Apache Software: *Welcome*. The Apache Software Foundation. 2014. URL: http://ant.apache.org/ (besucht am 15. 09. 2014) (siehe S. 15).

[Fou14b] Foundation, The Eclipse: *About the Eclipse Foundation*. The Eclipse Foundation. 2014. URL: http://www.eclipse.org/org/ (besucht am 15. 09. 2014) (siehe S. 14).

[Fou14c] Foundation, The Eclipse: *Eclipse Modeling - M2T (Home)*. The Eclipse Foundation. 2014. URL: http://www.eclipse.org/modeling/m2t/?project=xpand (besucht am 24. 09. 2014) (siehe S. 10).

[Gro14] Group, Java Persistence 2.1 Expert: *JSR 338: JavaTM Persistence API, Version 2.1*. Oracle. Nov. 2014. URL: http://download.oracle.com/otndocs/jcp/persistence-2_1-fr-eval-spec/index.html (besucht am 30. 11. 2014) (siehe S. 40).

[Ora14] Oracle: *AttributeConverter (Java(TM) EE 7 Specification APIs)*. Oracle. Nov. 2014. URL: https://docs.oracle.com/javaee/7/api/javax/persistence/AttributeConverter.html (besucht am 30. 11. 2014) (siehe S. 40).

[Wik14a] Wikipedia: *Domänenspezifische Sprache*. Wikipedia - Die freie Enzyklopädie. 2014. URL: http://de.wikipedia.org/wiki/Dom%C3%83%C2%A4nenspezifische_Sprache (besucht am 15. 09. 2014) (siehe S. 9).

[Wik14b] Wikipedia: *Jenkins (Software)*. Wikipedia - Die freie Enzyklopädie. 2014. URL: http://de.wikipedia.org/wiki/Jenkins_(Software) (besucht am 24. 09. 2014) (siehe S. 16).

[Wik14c] Wikipedia: *Klassendiagramm*. Wikipedia - Die freie Enzyklopädie. 2014. URL: http://de.wikipedia.org/wiki/Klassendiagramm (besucht am 15. 09. 2014) (siehe S. 8).

[Wik14d] Wikipedia: *MagicDraw*. Wikipedia - The Free Encyclopedia. 2014. URL: http://en.wikipedia.org/wiki/MagicDraw (besucht am 15. 09. 2014) (siehe S. 8).

[Wik14e] Wikipedia: *Mercurial*. Wikipedia - Die freie Enzyklopädie. 2014. URL: http://de.wikipedia.org/wiki/Mercurial (besucht am 15. 09. 2014) (siehe S. 14).

[Wik13a] Wikipedia: *openArchitectureWare*. Wikipedia - Die freie Enzyklopädie. 2013. URL: http://de.wikipedia.org/wiki/OpenArchitectureWare (besucht am 15. 09. 2014) (siehe S. 9).

[Wik13b] Wikipedia: *Relation (Datenbank)*. Wikipedia - Die freie Enzyklopädie. 2013. URL: http://de.wikipedia.org/wiki/Relation_(Datenbank) (besucht am 15. 09. 2014) (siehe S. 6).

[Wik14f] Wikipedia: *Unified Modeling Language*. Wikipedia - Die freie Enzyklopädie. 2014. URL: http://de.wikipedia.org/wiki/Unified_Modeling_Language (besucht am 15. 09. 2014) (siehe S. 8).

[Wik14g] Wikipedia: *Versionsverwaltung*. Wikipedia - Die freie Enzyklopädie. 2014. URL: http://de.wikipedia.org/wiki/Versionsverwaltung (besucht am 15. 09. 2014) (siehe S. 12).

[Wik13c] Wikipedia: *XML Metadata Interchange*. Wikipedia - Die freie Enzyklopädie. 2013. URL: http://de.wikipedia.org/wiki/XML_Metadata_Interchange (besucht am 29. 09. 2014) (siehe S. 1).

[Wik14h] Wikipedia: *Xtend*. Wikipedia - Die freie Enzyklopädie. 2014. URL: http://de.wikipedia.org/wiki/Xtend (besucht am 15. 09. 2014) (siehe S. 10).

[Wik14i] Wikipedia: *Xtext*. Wikipedia - Die freie Enzyklopädie. 2014. URL: http://de.wikipedia.org/wiki/Xtext (besucht am 15. 09. 2014) (siehe S. 9).

[Xte14a] Xtend: *Xtend Documentation*. The Eclipse Foundation. Sep. 2014. URL: http://eclipse.org/xtend/documentation/2.7.0/Xtend%20User%20Guide.pdf (besucht am 15. 09. 2014) (siehe S. 11, 57).

[Xte14b] Xtext: *Xtext Documentation*. The Eclipse Foundation. Sep. 2014. URL: http://www.eclipse.org/Xtext/documentation/2.7.0/Xtext%20Documentation.pdf (besucht am 15. 09. 2014) (siehe S. 10).

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Danksagung

Zunächst möchte ich mich an dieser Stelle bei all denjenigen bedanken, die mich während der Anfertigung dieser Bachelor-Thesis unterstützt und motiviert haben.

Im Besonderen möchte ich mich bei meiner Frau Andrea bedanken, ohne die ich weder mein Studium begonnen noch durchgehalten hätte. Sie schaffte es immer, mich in den Momenten zu motivieren, in denen ich kurz davor war, es abzubrechen. Dazu kommt, dass sie durch das ständige Korrekturlesen diese Ausarbeitung nun sicherlich besser kennt als ich. Ohne dich hätte ich dies hier wohl nie beendet.

Weiterhin möchte ich mich bei Herrn Prof. Dr.-Ing. Vogelsang für die gute Betreuung während der verschiedenen Arbeiten meines Studiums und insbesondere während dieser Arbeit bedanken.

Mein Dank gilt ebenfalls Hans-Georg Siegel für seine moralische Unterstützung und Motivation. Er hat mich dazu gebracht, über meine Grenzen hinaus zu denken. Vielen Dank für die Geduld und Mühen.

Auch meine Vorgesetzten und Kollegen bei der msgGillardon AG haben maßgeblich dazu beigetragen, dass diese Bachelor-Thesis nun so vorliegt. Vielen Dank, dass Sie mir die Möglichkeit gegeben haben, bei Ihnen zu forschen und zu arbeiten.

Ich möchte auch die Möglichkeit nutzen, mich an dieser Stelle bei Markus Hoek zu bedanken. Er gab mir eine Chance, als es kein anderer tat und ohne ihn wäre diese Arbeit wohl nie entstanden.

Last but not least möchte ich mich bei meinen Eltern und meinen Geschwistern bedanken. Sie gaben mir schon früh die Möglichkeit, mich mit meinen Interessen auseinanderzusetzen und so meinen Weg im Leben zu finden.