Einsatz von Model Driven Architecture bei der Entwicklung von Product Lifecycle Management Services

Masterarbeit an der Universität Ulm

Fakultät für Informatik

vorgelegt von

Stefan Schreinert

April 2004

1. Gutachter: Prof. Dr. Partsch
2. Gutachter: Prof. Dr. Schweiggert

Inhaltsverzeichnis

I Einführung **10**

1 Einleitung **11**

1.1 Die konventionelle Software-Entwicklung 11

1.1.1 Der tatsächliche Software-Entwicklungs-Zyklus 11

1.1.2 Das Problem der Portabilität 12

1.2 Die MDA-gestützte Software-Entwicklung 13

1.3 Zielsetzung dieser Arbeit 13

1.4 Aufbau dieser Arbeit 14

2 Prinzip der Model Driven Architecture **15**

2.1 Überblick über Model Driven Architecture 15

2.1.1 Bewertung der groben Architektur 16

2.1.2 Erweiterung um Interoperabilität 16

2.2 Transformationen für den MDA-Prozess 17

2.2.1 Aufbau einer Transformation 17

2.2.2 Kategorien von Transformationstools 18

2.3 Zusammenfassung 19

3 Verwendung weiterer Standards **20**

3.1 XML Metadata Interchange 20

3.2 Java Metadata Interface 21

3.3 Object Constraint Language 22

3.4 Zusammenfassung 23

4 Prinzip des Product Lifecycle Management Service **24**

4.1 Anforderungen an den PLM Service 25

4.2 Produktdaten 26

4.2.1 Strukturdaten eines Produktes 26

4.2.2 Produktstruktur einer Stückliste 27

4.3 Zusammenfassung 27

II Prototyp-Implementierung **28**

5 Entwurf der Transformationen **29**

6 Vorverarbeitung **31**

6.1 Manipulation des Namensraums 31

6.2 Markieren der Klassen und Attribute 32

6.3 Erzeugen des Primary Keys 33

6.4 Manipulation der Enumerations 34

6.5 Hinzufügen von Mehrsprachigkeit 35

6.6 Zusammenfassung 35

7 Transformationen der Daten-Schicht 36

7.1 Aufgaben und Anforderungen 36

7.2 PIM-nach-PSM – Transformation 37

7.2.1 Identifizierung der benötigten Entitäten 37

7.2.2 Umwandlung der Attribut-Typen in Datenbank-Typen 38

7.2.3 Auflösen der Vererbungen 39

7.2.4 Kürzen der Attribut-Namen 43

7.2.5 Vereinfachung des SQL-PSMs für die Weiterverarbeitung 45

7.3 PSM-nach-SQL – Transformation 45

7.4 Zusammenfassung 47

8 Transformationen der Service-Schicht 48

8.1 Aufgaben und Anforderungen 48

8.2 PIM-nach-PIM – Transformationen 49

8.2.1 Festlegen des Klassengerüsts 49

8.2.2 Erzeugen der Verteiltheit 51

8.3 Verknüpfung von Modell-Elementen 53

8.3.1 Erzeugung der Zugriffsmethoden 54

8.3.2 Verbinden der Klassen 54

8.3.3 Rückverfolgung vom PSM zum PIM 55

8.4 PIM-nach-PSM – Transformation 55

8.4.1 Anpassen der Client-Klassen 55

8.4.2 Einbauen des RemoteException-Handlings 56

8.4.3 Anpassen der Service-Klassen 57

8.4.4 Einführen eines Manager-Objekts 58

8.4.5 Auflösen von Entwurfs-Mustern 58

8.5 PSM-nach-Java – Transformation 59

8.6 Brücke zur Daten-Schicht 60

8.7 Zusammenfassung 62

9 Transformationen der Client-Schicht 64

9.1 Generierung des Servlet-Codes 64

9.2 Generierung des Beispiel-Templates 65

9.3 Zusammenfassung 67

III Bewertung 69

10 Bewertung des Prototypen 70

10.1 Vorteile der Model Driven Architecture 70

10.1.1 Geringerer Programmieraufwand 70

10.1.2 Änderungen an einer zentralen Stelle 71

10.1.3 Code-Qualität in jeder Klasse gleich 71

10.1.4 Wissen ist in Transformationen gekapselt 71

10.1.5 Transformationen sind wiederverwendbar 72

10.2 Nachteile der Model Driven Architecture 72

10.2.1 Sehr abstraktes Programmieren der Transformationen 72

10.2.2 PIM enthält nicht alle Informationen 73

10.2.3 Vorgehen bei der Transformations-Entwicklung 73

10.2.4 Definition der Transformations-Schnittstellen 73

10.3 Problemzonen der Model Driven Architecture 74

10.3.1 Vergleich des Code-Umfangs 74

10.3.2 Fehlersuche und -behebung 74

10.3.3 Wiederverwendbarkeit der Transformationen 76

10.3.4 Eignung für Prototypen-Entwicklung 77

10.3.5 Nachvollziehbarkeit des Transformations-Prozesses 77

10.4 Zusammenfassung 78

11 Transformationen formalisieren **79**

11.1 Template Engine 79

11.2 OCL als Transformations-Sprache 81

11.3 Zusammenfassung 85

12 Zusammenfassung und Ausblick **86**

12.1 Model Driven Architecture 86

12.2 MDA in CASE-Tools 87

12.3 Dynamische Aspekte von MDA 87

A Starten des beiligenden Programmes **91**

Literaturverzeichnis **95**

Abbildungsverzeichnis

1.1 Prozess zur Wartung von Software 12

2.1 Der MDA-Entwicklungs-Prozess 16

2.2 Interoperabilität mit Brücken 17

4.1 Produktstruktur für eine Baugruppe 26

6.1 Klasse „`Item`" mit Stereotypen nach der Vorverarbeitung 32

6.2 Mehrsprachigkeit erzeugen 35

7.1 Problem der Vererbung 39

7.2 Generalisierungen auflösen — Variante 1 40

7.3 Generalisierungen auflösen — Variante 2 40

7.4 Generalisierungen auflösen — Variante 3 41

7.5 Kompositionen zwischen *Item*, *Version*, *Design Discipline* und *Item Instance* 43

8.1 Klassengerüst für `Item` 51

8.2 Erweiterung von `Item` für die Verteilung 52

9.1 View-Servlet für `Item_version` 67

9.2 List-Servlet für `Item` 68

10.1 Komposition zwischen den Klassen der Produktstruktur 75

12.1 Sequenz-Diagram für `beginTransaction()` 88

12.2 Aktivitätsdiagramm für *unchanged* nach *changed* 89

Liste der Beispiel-Programme

1 Manipulation des Namensraums 32
2 Setzen von Stereotypen 33
3 Identifizierung aller Entitäten - Teil 1 37
4 Identifizierung aller Entitäten - Teil 2 38
5 Auflösen der Generalisierungen 42
6 Kommunikation mit der Service-Schicht (Beispiel) 53
7 Methode `handleRequest` des erzeugten `ItemViewServlets` 66

Abkürzungsverzeichnis

BoM	Bill of Materials, Stückliste
CORBA	Common Object Request Broker Architecture (http://www.corba.org/)
DMU	DMU - Digital Mockup
EJB	Enterprise JavaBean (http://java.sun.com/products/ejb/)
ERD	Entity-Relationship-Diagramm
GUI	Graphical User Interface
IDL	Interface Definition Language
ISO	International Organization for Standardization (http://www.iso.ch/)
JMI	Java Metadata Interface (http://java.sun.com/products/jmi/)
JNDI	Java Naming and Directory Interface (http://java.sun.com/products/jndi/)
MDA	Model Driven Architecture (http://www.omg.org/mda/)
MDR	Metadata Repository Project (http://mdr.netbeans.org/)
OCL	Object Constraint Language (http://www.omg.org/)
OEM	Original Equipment Manufacturer
OMG	Object Management Group (http://www.omg.org/)
PDM	Product Data Mangagement
PIM	Platform Independent Model
PLM	Product Lifecycle Management
PSM	Platform Specific Model
RMI	Remote Method Invocation (http://java.sun.com/products/jdk/rmi/)
SQL	Structured Query Language
STEP	Standard for the Exchange of Product Model Data
UDDI	Universal Description, Discovery and Integration (http://www.uddi.org/)
UML	Unified Modelling Language (http://www.omg.org/uml/)
WSDL	Web Services Description Language (http://www.w3.org/TR/wsdl)
XMI	XML Metadata Interchange (http://www.omg.org/technology/xml/)
XML	Extensible Markup Language (http://www.w3.org/XML/)

Zusammenfassung

Diese Arbeit gibt einen praktischen Einblick in eine aufkommende Software-Entwicklungs – Methodik namens *Model Driven Architecture*. Dieses Verfahren zwingt den Entwickler, alle notwendigen Informationen in einem Plattform-unabhängigen Modell formal zu beschreiben und auch nur in diesem zu pflegen. Dafür wird der Entwickler damit belohnt, dass sich Änderungen im Modell automatisiert in lauffähigen Programm-Code umwandeln lassen. Hierfür kommen *Transformationen* zum Einsatz, die auf dem gegebenen Modell arbeiten und dieses in ein anderes Modell überführen. Dadurch wird das Plattform-unabhängige Modell zu einem Modell speziell für eine Plattform umgewandelt und dieses letztendlich in ausführbaren Programm-Code. Diese Arbeit zeigt die Methodik der Model Driven Architecture anhand des neuen Standards *Product Lifecycle Management Service* aus dem Automobilbereich. Dazu beschreibt sie alle notwendigen Transformationen, um aus dem vorgegebenen Modell eine Drei-Schichten-Architektur zu generieren, welche den Anforderungen des Standards entspricht. Abschließend werden Erkenntnisse und Verbesserungen diskutiert.

Teil I

Einführung

Kapitel 1

Einleitung

Bereits in den sechziger Jahren erkannten die Software-Unternehmen, wie komplex es ist, Software zu entwickeln.[1] Die Struktur der Programme und die steigende Zahl von Code-Zeilen führte dazu, dass der Code sehr schnell unüberschaubar und damit unwartbar wurde. In dieser Zeit wurde erstmals der Begriff des *Software-Engineerings* geprägt. Ab sofort sollte die Software-Programmierung ingenieurmäßig entwickelt werden.

1.1 Die konventionelle Software-Entwicklung

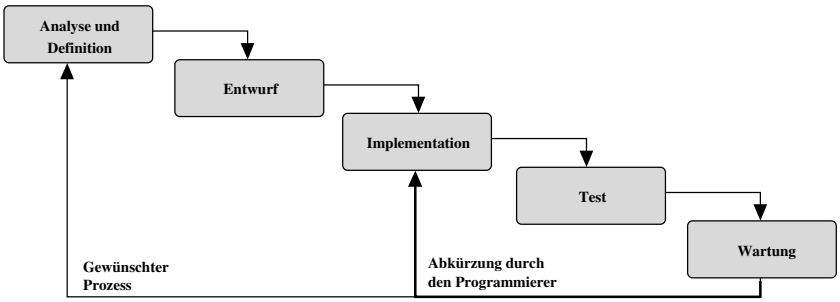
Es entstanden mehrere Prozessmodelle, wie Software entwickelt werden soll. Vertreter hierfür sind das *Phasenmodell*[2], das *Spiralmodell*[3] und die *formale Software-Entwicklung*[4]. Bei den Modellen durchläuft dabei der Entwicklungs-Prozess in etwa die gleichen Phasen: die *Analyse*, der *Entwurf*, die *Implementierung* und die *Wartung*. Die Modelle unterscheiden sich hauptsächlich in den Übergängen der einzelnen Phasen. So startet beim *Wasserfall-Modell*[5] eine Phase nach der anderen, ohne dabei zu einer früheren Phase zurückzukehren. Das *iterative Phasenmodell*[6] hingegen erlaubt es, zu jeder beliebigen, früheren Phase zurückzukehren. Beispielsweise von der Wartungs-Phase in die Analyse-Phase. Auch ist der Sprung zurück zur Implementierung erlaubt.

1.1.1 Der tatsächliche Software-Entwicklungs-Zyklus

Dieser direkte Sprung zurück zur Implementierung ist durchaus sinnvoll, sollen Fehler der Implementierung korrigiert werden. Für Programm-Erweiterungen und neue Versionen muss aber immer bis zur Analyse-Phase zurückgesprungen werden, denn dies ist der Ausgangspunkt jeder Software. Nur hier

[1] Vgl. [Par98, S. 2].
[2] Vgl. [Par98, S. 2f].
[3] Vgl. [Par98, S. 5f].
[4] Vgl. [Par98, S. 7f].
[5] Vgl. [Par98, S. 2].
[6] Vgl. [Par98, S. 3f].

lässt sich bestimmen, was der Kunde benötigt. Im Endeffekt bedeutet dies, dass die Analyse-Phase sämtliche relevanten Informationen halten muss. Während der Entwurfsphase werden diese in eine andere Darstellung gebracht, etwa in ein UML-Diagramm. Die Implementierung wiederum verfeinert den Entwurf, in dem es das Diagramm in Programm-Code umwandelt. Aber genau hier setzt das Problem an, dass Entwickler gerne essentielle Änderungen direkt im Programm-Code während der Implementierungsphase einbauen.[7] Das Ergebnis ist, dass Informationen in allen früheren Phasen fehlen. Die Abbildung 1.1 zeigt den gewünschten Prozess, wenn die Software erweitert werden soll zusammen mit dem Weg den der Programmierer bevorzugt.



Quelle: Nach [KWB03, S. 3]

Abbildung 1.1: Prozess zur Wartung von Software

Besonders negativ ist dies, wenn zu einem späteren Zeitpunkt herausgefunden werden soll, warum ein bestimmtes Programm-Modul entwickelt wurde. Hier lässt sich nur schwer die Anforderung ausmachen. Zugleich wird auch der modellierte Entwurf obsolet, da auch diesem Informationen fehlen. Damit ist dieses Modell ungeeignet, um die Software auf abstraktem Niveau zu diskutieren. Zudem ist es ungeeignet, darin zukünftige Änderungen einzupflegen. Dies muss weiterhin während der Implementierung erfolgen. Damit wird mit der Zeit die beschriebene, sinnvolle Rückkehr von der Wartungs-Phase zur Analyse-Phase unmöglich.

1.1.2 Das Problem der Portabilität

Daneben besteht das Problem, dass die Software oftmals nur für eine spezielle Plattform entwickelt wird.[8] Beispielsweise lässt sich eine in C++ entwickelte Software für Windows nur mit großen Anpassungen auf Unix portieren. Aber auch wenn Sprachen für virtuelle Maschinen verwendet werden, etwa Java, gibt es Probleme. Beispielsweise dann, wenn Software-Module mit anderen Modulen kommunizieren müssen. Hier gibt es etwa die Kommunikations-Protokolle *Remote Method Invocation* (RMI), *Common Object Request Broker Architecture* (CORBA) oder *Web Services*. Aber selbst wenn ein Kommunikations-Protokoll gefunden ist, so wird es sich über die Zeit weiterentwickeln. So bieten die *Enterprise JavaBeans* (EJB)[9] in der Version 2 zahlreiche Vorteile gegenüber der früheren Version, etwa die *container-managed relationships* oder die *EJB query language*.[10]

[7] Vgl. [KWB03, S. 2ff].
[8] Vgl. [KWB03, S. 4].
[9] Vgl. [DP02] und [Mic03].
[10] Vgl. [Ste01] und [DP02].

Der Aufwand, um portable Software zu entwickeln, lässt sich reduzieren, indem die Entwickler eingeschränkt werden. So sollten die Entwickler der grafische Benutzerschnittstelle (GUI) solche Bibliotheken verwenden, die auch auf anderen Systemen laufen. Beispielsweise würde mit QT der Firma trolltech[11] die gleiche GUI sowohl unter Windows als auch unter Linux erscheinen, ohne dass Programm-Code verändert wird.

Allerdings ist diese Lösung nicht für alle Portabilitäts-Probleme einsetzbar. So gibt es unter C/C++ Datentypen, die von System zu System unterschiedlich definiert sind — einmal vorzeichenbehaftet und einmal nicht. Hier muss der Entwickler entsprechend restriktiert werden, so dass er andere Datentypen verwenden soll.

Als Alternative zur Restriktion lässt sich der erstellte Programm-Code nachträglich für ein anderes System umschreiben, sprich portieren. Dazu müsste aber entweder der jeweilige Entwickler die Portierung durchführen oder aber ein anderer Entwickler dafür zuständig sein. Beides hat den Nachteil, dass solch ein Entwickler für beide Domänen ein Experte sein müsste. Der Portierer muss die zu portierende Software genau verstehen, während der Anwendungs-Entwickler die Eigenarten der Zielplattform kennen muss. Somit lässt sich die Portabilität nur mit einem deutlich erhöhten Aufwand betreiben.

1.2 Die MDA-gestützte Software-Entwicklung

Der Ansatz der *Model Driven Architecture* (MDA) versucht, die Probleme der konventionellen Software-Entwicklung zu vermeiden. Dies soll dadurch erreicht werden, dass die Software aus den Anforderungen *generiert* wird und der Entwickler somit keinen Code mehr programmieren muss. Die Aufgabe ist es nun, alle relevanten Informationen in ein wohl-definiertes Modell zu packen und auch nur dort zu pflegen.

MDA lässt dem Entwickler dabei die Wahl, was für ein Modell er verwenden möchte. Die Problemstellung muss folglich nicht zwangsweise in der *Unified Modelling Language* (UML)[12] modelliert sein, etwa dann, wenn ein Datenbank-Schema abgebildet werden soll. Hier gibt es andere Repräsentationen, etwa des *Entity-Relationship-Diagramm*. MDA setzt lediglich ein wohl-definiertes Modell voraus. Im folgenden sind die Modelle in UML modelliert, da sich diese Modellierungssprache für vielseitige Problemstellungen einsetzen lässt.

1.3 Zielsetzung dieser Arbeit

Die vorgestellte Arbeit beschäftigt sich mit der relativ neuen Art der Software-Entwicklung: die *Model Driven Architecture*. Ziel ist es, ein lauffähiges Programm dadurch zu erreichen, dass die Software aus einem Modell erzeugt wird. Dazu soll aufgezeigt werden, unter welchen Restriktionen es möglich ist, Software automatisiert erzeugen zu lassen.

[11] `http://www.trolltech.com/`, zuletzt besucht am 20.04.2004.
[12] Vgl. etwa [JRH+03] oder [Fow03].

Als Beispiel dienen die *Product Lifecycle Management Service* (PLM Service)[13], die sich derzeit in den letzten Phasen des Standardisierungsprozesses der OMG[14] befinden. Dieser Standard beschreibt Schnittstellen, über die sich unter anderem Produktdaten der Automobilindustrie abfragen und damit austauschen lassen.

Diese Ausarbeitung modelliert dazu einen Teil dieses Standards nach und zeigt auf, wie dieses vorgegebene Modell zu Programm-Code umzuwandeln ist. Dabei wird gezielt darauf verzichtet, auf verwandte MDA-Projekte zurückzugreifen, etwa dem AndroMDA-Projekt[15]. So gibt es bereits Software-Tools, die Teile eines Modells in Code umwandeln, etwa für die weiter unten vorgestellte PSM-nach-Code – Transformation[16]. Auch könnten einige Probleme von anderen Standards gelöst werden, etwa durch *Enterprise JavaBeans*[17]. Aber auch auf dies wurde verzichtet, um den Umwandlungs-Prozess genauer darstellen zu können.

Zuletzt soll diese Arbeit den MDA-Entwicklungsprozess bewerten. Es soll gezeigt werden, auf welche Eigenarten beim produktiven Einsatz von MDA zu achten ist. Zudem sollen Schwachpunkte von MDA aufgedeckt und konkrete Vorschläge geliefert werden. Spätere Tools können somit den MDA-Prozess effizienter werden lassen.

1.4 Aufbau dieser Arbeit

Diese Arbeit untergliedert sich in drei Teile. Der erste Teil beschäftigt sich mit den *Grundlagen* für den MDA-Prozess (Kapitel 2) und dem *PLM Service* (Kapitel 4). Außerdem werden relevante *Technologien* angesprochen, um dem Leser einen Einblick in diese zu vermitteln (Kapitel 3).

Der zweite Teil ist der Hauptteil der Ausarbeitung. Dieser stellt den für die Arbeit entwickelten Prototypen vor und zeigt, wie das Modell in lauffähigen Programm-Code zu transferieren ist. Dabei werden die Prozesse für jede Ebene einer Drei-Schichten-Architektur vorgestellt (Kapitel 7 bis 9).

Der dritte Teil arbeitet die Erfahrungen aus dem zweiten Teil heraus, bewertet diese und versucht, bessere Alternativen zu finden (Kapitel 10). Dazu wird versucht, die fest-verdrahteten Transformationen aus dem zweiten Teil durch formalere Verfahren zu verbessern (Kapitel 11).

[13] Vgl. [FHH+03].
[14] `http://www.omg.org`, besucht am 20.04.2004.
[15] `http://www.andromda.org/`, besucht am 20.04.2004. Siehe auch Abschnitt 2.2.2 auf Seite 19 und 11.1 auf Seite 79.
[16] Siehe etwa 7.3 auf Seite 45 und 8.5 auf Seite 59.
[17] Vgl. [DP02] und [Mic03].

Kapitel 2

Prinzip der Model Driven Architecture[1]

Dieses Kapitel soll den Leser in die Prinzipien des Model Driven Architecture einführen. Er erfährt dabei das grundsätzliche Vorgehen und lernt die verwendeten Begriffe kennen. Zudem gibt das Kapitel einen Einblick, wie Transformationen aufgebaut sind und wie sie arbeiten. Im späteren Verlauf der Arbeit werden die hier vorgestellten Grundlagen weiter vertieft.

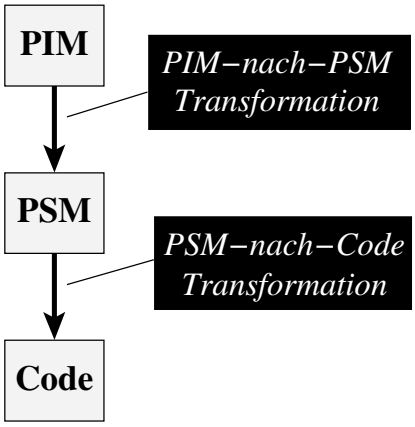
2.1 Überblick über Model Driven Architecture

Model Driven Architecture beschreibt ein Verfahren, um aus einem Modell einen lauffähigen Programm-Code zu generieren. Dazu enthält dieses Modell alle für das Programm relevanten Informationen, ohne jedoch auf Eigenschaften einer konkreten Plattform einzugehen. Daraus ergibt sich der Name *Platform Independent Model* (PIM). Der Grund für die Plattform-unabhängigkeit liegt darin, dass aus dem Modell Implementierungen für verschiedene Plattformen zu erzeugen sind. Dies bedeutet einen Vorteil gegenüber der konventionellen Software-Entwicklung.[2]

Bevor der eigentliche Programm-Code erzeugt wird, findet eine Transformation des PIMs in ein weiteres Modell statt, das sogenannte *Platform Specific Model* (PSM). Während dieser Transformation werden dem Modell alle Informationen zugefügt, die für die ausgewählte Plattform notwendig sind. Für die Ziel-Plattform EJB würde das PSM zusätzlich die Home-Interfaces zu den Beans enthalten. Auch wären die Beans entsprechend ihrem Typ markiert, etwa als EntityBean oder als SessionBean.

Um vom PIM ins PSM zu kommen, muss folglich das PIM überführt werden. Diese Aufgabe übernimmt eine Transformation, in diesem Fall eine *PIM-nach-PSM – Transformation*. Um aus dem PSM ein lauffähiges Programm zu erzeugen, ist eine weitere Transformation notwendig: die *PSM-nach-Code – Transformation*. Die Abbildung 2.1 auf der nächsten Seite zeigt das grundsätzliche Vorgehen.

[1] Vgl. [KWB03, S. 8] und [MM, S. 2-1ff]
[2] Vgl. 1.1.2 auf Seite 12.



Quelle: Nach [KWB03, S. 8 u. 10]

Abbildung 2.1: Der MDA-Entwicklungs-Prozess

2.1.1 Bewertung der groben Architektur

Auf den ersten Blick mag es wenig sinnvoll erscheinen, ein weiteres Modell zu erzeugen, da sich auch gleich aus dem PIM der Code erzeugen lassen würde. Der Vorteil der MDA-Vorgehensweise ist, dass die PIM-nach-PSM – Transformation auf eine Problemdomäne angepasst ist. Die PSM-nach-Code – Transformation hingegen ist unabhängig von speziellen Problemen. Sie erzeugt aus jedem PSM den dazugehörigen Programm-Code. Damit gibt es mehrere PIM-nach-PSM – Transformationen aber nur eine PSM-nach-Code pro Plattform.

Während dieser Arbeit hat sich zudem gezeigt, dass selbst zwei Transformationen zu wenig sind.[3] Es ist sinnvoller, für jedes Problem eine Transformation einzusetzen. Beispielsweise verändert die erste Transformation das PIM soweit, dass sich die Software verteilt einsetzen lässt. Die zweite Transformation reichert dieses Modell anschließend um Informationen an, die von einem konkreten Kommunikations-Protokoll zwischen den Programm-Modulen benötigt werden. Eine dritte Transformation sorgt dafür, dass die Kommunikation verschlüsselt verläuft.

2.1.2 Erweiterung um Interoperabilität

Sollen nun aus einem Modell verschiedenen Implementationen erzeugt werden, so sind hierfür auch verschiedene Transformationen nötig. Der später vorgestellte Prototyp ist beispielsweise eine dreischichtige Anwendung, wobei der Client auf die Service-Schicht und diese auf die Daten-Schicht zugreifen muss. Jede dieser Schicht kann auf unterschiedlichen Plattformen laufen, z.B. die Daten-Schicht als Datenbank und die Service-Schicht als Web Services. Damit müssen aus dem PIM im ersten Schritt drei PSMs erzeugt werden, eines für die Daten-Schicht, eines für die Service-Schicht und eines für die Client-Schicht. Abbildung 2.2 zeigt dieses Vorgehen.

[3] Vgl. auch [MM, S. 6-2ff].

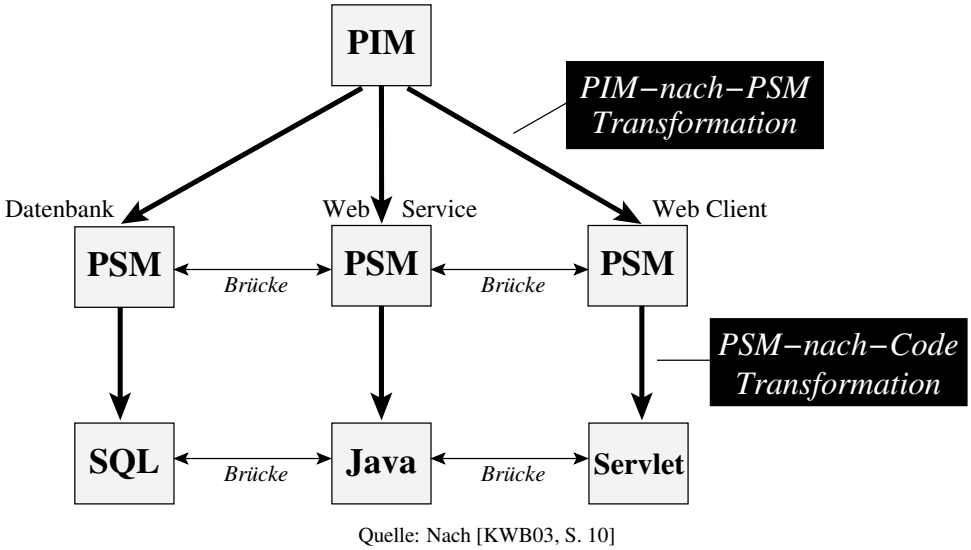


Abbildung 2.2: Interoperabilität mit Brücken

Dabei stellen die Brücken die Verbindung zwischen den einzelnen Schichten dar. Auch für den Prototypen sind diese Brücken von enormer Bedeutung. Sie erlauben dem Client Zugriff auf die Services — unabhängig von der tatsächlichen Implementierung der Service-Schicht. Die Brücke muss dazu beispielsweise das Kommunikations-Protokoll verbergen.

2.2 Transformationen für den MDA-Prozess

Ein wichtiger Bestandteil des MDA-Prozesses sind Transformationen. Erst sie ermöglichen es, ein Modell in ein anderes umzuwandeln, etwa um aus einem PIM ein PSM zu erzeugen. Eine einfache Transformation ist beispielsweise, aus einer gegebenen Klasse alle öffentlichen Attribute durch Getter- und Setter-Methoden zu ersetzen.

Neben dem zu transformierenden Modell muss eine Transformation weitere Informationen erhalten. Diese *Transformations-Parameter* geben etwa an, dass `String` in `VARCHAR(255)` umzuwandeln ist. Für diese Arbeit enthält eine Konfigurations-Datei unter anderem die Angaben, ob Debug-Anweisungen erzeugt werden sollen, ob bei Web Service – Anfragen die Kommunikation zu verschlüsseln ist oder welcher Datentyp als Container zu verwenden ist. Der Aufbau dieser Datei ist dabei beliebig. Für den Prototypen wurde eine Properties-Datei mit einfachen Schlüssel/Wert – Paaren eingesetzt.

2.2.1 Aufbau einer Transformation

Grundsätzlich unterliegen Transformationen keinerlei Restriktionen. Ihre Aufgabe ist nur ein Modell in ein anders umzuwandeln. Wie sie dies vornehmen, bleibt vollständig ihnen überlassen. Für den Pro-

totypen ist jede Transformation ein gewöhnliches Java-Programm, welches über das JMI-Interface[4] das Modell manipuliert.

Während der Realisierung des Prototypens hat sich aber gezeigt, dass die Transformationen aufgespalten werden sollten. So gibt es Aufgaben innerhalb der Transformation, welche unabhängig von dem konkret zu lösenden Problem sind. Um beispielsweise für jedes öffentliche Attribut die Accessoren zu erzeugen, müssen zuerst alle öffentlichen Attribute gefunden werden. Diese Aufgabe fällt auch bei anderen Transformationen an, etwa wenn ein IDL-Interface für CORBA generiert werden soll.

Aus diesem Grund sollten die Transformationen getrennt werden in Transformations-Tools und Transformations-Definition bzw. -Regeln. Das Transformationstool enthält Routinen, um ein Modell zu lesen und zu manipulieren. Die Transformations-Regeln bestimmen, was und wie zu transformieren ist. Das Kapitel 11 auf Seite 79 beschreibt diese Formalisierung genauer.

2.2.2 Kategorien von Transformationstools

Um MDA für ein Software-Projekt einsetzen zu können, sind grundsätzlich zwei Dinge notwendig. Zum einen muss ein korrektes Modell des Problems erstellt und gepflegt werden. Zum anderen sind Tools notwendig, welche aus dem Modell letztendlich einen ausführbaren Programm-Code erzeugen. In diesem Abschnitt sind verschiedene Arten von Transformationstools beschrieben und werden — basierend auf [KWB03, S. 36ff] — kategorisiert.

PIM-nach-PSM – Transformationstools

Diese Art von Tools ist sehr stark von einem konkreten Problem abhängig. Es ist davon auszugehen, dass kurz- und mittelfristig diese Transformationstools noch für jedes Projekt neu zu schreiben sind. Alternativ müssen bestehende Tools angepasst werden. Es besteht die Möglichkeit, dass hier Problem-spezifische Transformationstools das PIM so transformieren, dass „weniger" Problem-spezifische Transformationstools darauf aufbauen können. So müsste nicht das gesamte Transformationstool neu entwickelt werden.

PSM-nach-Code – Transformationstools

Bei den PSM-nach-Code – Transformationentools sieht es anders aus, wie bei den Tools für PIM-nach-PSM. Für diese Art existieren bereits mehrere Transformationstools. Dabei enthalten diese Tools zumeist eine feste, eingebaute Transformationsregel, die sich nur minimal anpassen lässt. Dafür transformieren sie ein fest vorgegebenes Modell in ein genau definiertes Code-Modell.[5] Beispielsweise kann Rational Rose 2000 aus einem UML-Modell ein SQL-Datenbank-Schema erzeugen, wenn auch nur mit großen Restriktionen. CASE-Tools wie dieses fallen ebenfalls in diese Rubrik der Transformationstools, da ihr Ausgangsmodell fest definiert ist.

[4] Siehe Abschnitt 3.2 auf Seite 21.
[5] Vgl. [KWB03, S. 37].

PIM-nach-Code – Transformationstools

Auch hierfür gibt es bereits Tools. Der Abschnitt 2.1.1 auf Seite 16 zeigt, warum dies nur bedingt sinnvoll ist. Es ist aber jedem selber überlassen, solche Tools für seine Aufgaben einzusetzen. So kann das AndroMDA-Projekt[6] ebenfalls in diese Kategorie aufgenommen werden. AndroMDA erzeugt aus einem PIM sofort ein Klassen-Gerüst. Allerdings fehlen die dynamischen Aspekte, so dass der Entwickler die Methoden-Körper noch konventionell mit Programm-Code füllen muss.

Anpassbare Transformationstools

Transformationstools mit einem fest verdrahteten Regelwerk funktionieren zwar sehr gut, sind aber nur für spezielle Problemstellungen einsetzbar. Abhilfe wollen Tools liefern, bei denen die Regeln etwa per Skriptsprache veränderbar sind.[7] Mangels fehlender Standardisierung einer solchen Skriptsprache sind diese Tools-spezifisch. Es ist folglich aufwändig, eine Reihe von Tools unterschiedlicher Hersteller zu bedienen.

Transformations-Regel Tools

Da es sehr komplex ist, anpassbare Transformationstools zu justieren, bieten sich Tools an, die einem diese Aufgabe abnehmen. Aber auch für diese ist es notwendig, eine standardisierte Definitionssprache zu bekommen. Nur so können damit anpassbare Tools auch wirklich sinnvoll angepasst werden.

2.3 Zusammenfassung

Das Kapitel hat gezeigt, dass die Model Driven Architecture einen relativ einfachen Aufbau hat. So befinden sich alle relevanten Informationen in einem Plattform-unabhängigen Modell, dem PIM. Transformationstools setzen dabei beim PIM an und transformieren das Modell in ein Plattform-spezifisches Modell, dem PSM. Dieses Modell ist damit — im Gegensatz zum PIM — um alle notwendigen Informationen bereichert, die für die Zielplattform notwendig sind. Damit lässt sich das PSM in einer weiteren Transformation zu ausführbaren Programm-Code umwandeln.

Es hat sich gezeigt, dass die Transformationen derzeit noch fest verdrahtet aufgebaut sind. Es empfiehlt sich jedoch, Transformationen aufzuspalten in Tools und in Definitionen bzw. Regeln. Somit lässt sich eine Transformation leicht auf neue Situationen einstellen. Dies setzt jedoch eine einheitliche, standardisierte Definitionssprache voraus, welche derzeit noch fehlt.

[6] `http://www.andromda.org/`, besucht am 20.04.2004. Siehe auch Abschnitt 11.1 auf Seite 79.
[7] Vgl. [KWB03, S. 37].

Kapitel 3

Verwendung weiterer Standards

Die Model Driven Architecture ist eine Methodik, wie Software entwickelt werden kann. Dabei beruht MDA nicht auf konkreten Tools oder Standards. Dennoch gibt es hiervon einige, die sinnvoll im Umgang mit MDA sind, welche in diesem Kapitel kurz erläutert werden sollen.

3.1 XML Metadata Interchange

Damit die graphischen UML-Modelle auch von den Transformationstools verarbeitet werden können, müssen sie in einer Repräsentation vorliegen, die sich automatisiert verarbeiten lässt. Hierzu hat sich ein Standard gebildet, der genau diesen Austausch von UML-Modellen ermöglichen soll: *XML Metadata Interchange* (XMI)[1].

Dazu liegen sämtliche Modelle, Grafiken und Kommentare in einer XML-Datei vor. Die nachfolgenden XML-Zeilen zeigen auszugsweise eine Klasse namens `Item` mit dem öffentlichen Attribut `description`. Der Typ des Attributs ist dabei ein Datentyp, hier identifiziert mit der ID `a003`.

```
<UML:Class xmi.id = 'a001' name = 'Item' visibility = 'public'
     isSpecification = 'false' isRoot = 'false' isLeaf = 'false'
     isAbstract = 'false' isActive = 'false'>
  <UML:Classifier.feature>
    <UML:Attribute xmi.id = 'a002' name = 'description'
       visibility = 'public' isSpecification = 'false'
       ownerScope = 'instance' changeability = 'changeable'>
      <UML:StructuralFeature.type>
        <UML:DataType xmi.idref = 'a003'/>
      </UML:StructuralFeature.type>
    </UML:Attribute>
  </UML:Classifier.feature>
</UML:Class>
```

[1] Vgl. `http://www.omg.org/technology/documents/formal/xmi.htm`, besucht am 20.04.2004. Sowie [OMG02] und [OMG03b].

Hierüber lassen sich sämtliche UML-Objekte und -Diagramm abbilden. Damit ist es auch möglich, Attribute als statisch oder unveränderlich zu kennzeichnen und Klassen Stereotypen oder Tagged-Values zu vergeben. Deshalb sind die XMI-Dateien derart für MDA interessant ist.

Auch wenn XMI standardisiert ist, so ist es nur bedingt für den Datenaustausch geeignet. Während dieser Arbeit hat sich gezeigt, dass ein im CASE-Tool Poseidon von gentleware[2] erstelltes UML-Modell weder in Together[3] geladen wurde, noch in Rational-Rose[4] mit dem XMI-Plugin von Unisys[5]. Somit muss für den produktiven Einsatz der MDA-Methodik vorab genauestens geklärt werden, welcher XMI-"Standard" verwendet wird und somit welches CASE-Tool und welche Transformationstools.

3.2 Java Metadata Interface

Java bietet mit dem *Java Metadata Interface* (JMI) eine Schnittstelle, um relativ einfach auf die XMI-Daten zugreifen zu können. Zwar wäre dies auch mit einem normalen XML-Parser möglich, jedoch liegen sämtliche Modell-Elemente dank JMI in einer entsprechenden Klassen-Repräsentation vor. Somit entspricht eine Klasse im XMI-Code einem konkreten Java-Objekt, welches über Methoden manipuliert werden kann.

Nachdem das XMI-Modell eingelesen ist, kann auf die Daten über die Startklasse `UmlPackage` zugegriffen werden, wobei diese Management-Klasse unglücklicherweise den selben Namen hat, wie das aus UML bekannte Package. Von der `UmlPackage`-Klasse aus kann nun beispielsweise auf Zustandsdiagramme und verschiedene Zusammenarbeitsdiagramme zugegriffen werden. Für den statisch UML-Teil gibt es das `CorePackage`, welches ebenfalls über `UmlPackage` erreichbar ist. Hierin finden sich die UML-Klassen, die Packages, die Attribute und Methoden, sowie alle weiteren statischen Modell-Elemente.

Während dieser Arbeit hat sich gezeigt, wie überaus nützlich und arbeitserleichternd das JMI-Framework ist. Jedoch treten zahlreiche Tücken und Fallstricke während des Gebrauchs auf. So kann beispielsweise das selbe Datentyp-Objekt `DataType` für ein weiteres Attribut vergeben werden. Multiplizitäten für Assoziationen oder Attribute hingegen sind vor Gebrauch zu kopieren. Dieses inkonsistente Verhalten ist auf die XMI-Definition zurückzuführen. Datentypen sind innerhalb ihrer Verwendung als Referenz auf die eigentliche Definition angelegt, während Multiplizitäten direkt zum zugehörigen Modell-Element definiert sind (siehe den XMI-Code im Abschnitt auf der vorherigen Seite). Somit ist es hilfreich, sich trotzt JMI mit XMI auseinanderzusetzen.

Weiterhin hat sich gezeigt, dass häufiger der Bedarf vorhanden ist, Modell-Elemente zu kopieren oder zu löschen. Dies ist aber von JMI so nicht ohne weiteres möglich, weshalb für diese Arbeit eine nützliche Helper-Klasse erstellt wurde. In dieser befinden sich Methoden, um beispielsweise Klassen zu kopieren oder Assoziationen zu löschen. Darüberhinaus enthält die Helper-Klasse noch weniger komplexe, aber dennoch hilfreiche Methoden. Beispielsweise muss während der Code-Generierung der

[2] `http://www.gentleware.com/`, besucht am 21.04.2004.
[3] `http://www.borland.com/together/`, besucht am 22.04.2004.
[4] `http://www.rational.com/`, besucht am 21.04.2004.
[5] `http://www.unisys.com/`, besucht am 22.04.2004.

komplette Klassen-Pfad angegeben werden. Auch hierfür wurde für diese Arbeit eine entsprechende Methode entwickelt.

Das Java Metadata Interface bietet somit eine sehr hilfreiche Methodik an, um bequem auf ein XMI-Modell zuzugreifen. Für den konkreten Einsatz gibt es mehrere Implementierungen zum Interface. Für diese Arbeit kam das *Metadata Repository Project*-Tool (MDR)[6] von NetBeans[7] zum Einsatz. Trotz der Arbeitserleichtung ist dringend zu empfehlen, sich für den Gebrauch des Frameworks eine Helper-Klasse anzulegen, um immer wiederkehrende, aufwändige Arbeitsschritte zu kapseln.

3.3 Object Constraint Language

Mithilfe der *Object Constraint Language*[8] lässt sich ein statisches UML-Modell um dynamische Aspekte erweitern. So können, wie der Namen schon verrät, Bedingungen angegeben werden, die erfüllt sein müssen. Hierzu gibt es pro Klasse beziehungsweise pro Methode drei elementare Bereiche: `inv`, `pre` und `post`. Der Bereich `inv` gibt an, dass ein Klassen-Attribut die gesamte Zeit über erfüllt sein muss, beispielsweise soll eine Mengenangabe immer größer als 0 und kleiner als 10 sein:

```
context demoClass
   inv : self.amount > 0 and self.amount < 10
```

Die Bedingung im Bereich `pre` muss erfüllt sein, bevor die Methode startet. Bei `post` gilt, dass die Bedingung am Ende der Methode erfüllt sein muss.

So lässt sich beispielsweise definieren, dass eine Methode zum Setzen eines Wertes nur einmal aufgerufen werden darf. Die `pre`-Bedingung gibt an, dass das zu setzende Attribut noch undefiniert sein muss. Die `post`-Methode sagt aus, dass das Attribut nun den Wert des Methoden-Parameters hat:

```
context demoClass:setMethod( value : Object )
  pre : self.storedValue = oclVoid // is unintialized
  post : self.storedValue = value
```

Diese Bedingungen können nun von einem Constraint-Compiler ausgewertet werden. Dieser Compiler wäre dann in der Lage, einen entsprechenden Programm-Code zu erzeugen, um die Einhaltung der Bedingungen zu gewährleisten.

Hierfür gibt es beispielsweise den *dresden-ocl-Compiler*[9] der Universität Dresden. Dieser ist derzeit in der Lage, Constraints für Java und Datenbank-Schemata zu erzeugen. Beim programmierten beziehungsweise erzeugten Java-Quelle-Code ist lediglich die OCL-Bedingung als JavaDoc-Tag über die betroffene Methode anzugeben. Anschließend wird der OCL-Compiler gestartet, der daraufhin

[6] `http://mdr.netbeans.org/`, besucht am 20.04.2004.
[7] `http://www.netbeans.org`, besucht am 20.04.2004.
[8] Vgl. [WK03] oder [OMG03a].
[9] `http://dresden-ocl.sourceforge.net`, besucht am 20.04.2004.

die Klassen um die Abfragen, Prüfungen und gewünschten Exceptions erweitert. Der so manipulierte Programm-Code lässt sich per Java-Compiler übersetzen.

Mit der Version 2 von OCL sind weitere Sprachelemente hinzugekommen. Somit ist OCL mehr als nur ein Constraint-Prüfer. Jetzt lässt sich via OCL auch das UML-Modell manipulieren. Hierzu gibt es die Bereiche `let` und `def`, über die neue Attribute und Methoden hinzugefügt werden können. Darüber hinaus ist der Bereich `body` hinzugekommen. Über diesen lässt sich der Körper einer Query-Methode definieren. Ein Compiler wäre somit in der Lage, die Methode in einen gewünschten Programm-Code zu übersetzen. Dies gilt dabei nur für Abfrage-Methoden, wie Getter. Setter-Methoden lassen sich damit nicht definieren.

Hierzu bedarf es eines komplexeren Compilers. Dieser müsste in der Lage sein, einen Code zu generieren, der für die Einhaltung der Bedingungen sorgt. Beispielsweise lautet die obige `post`-Bedingung, dass ein bestimmtes Attribut am Ende der Methode den Wert des Parameters annehmen soll. Hieraus ließe sich noch relativ einfach ein entsprechender Code generieren: das Attribut müsste einfach mit dem Parameter initialisiert werden.

Die Object Constraint Language hat sich bereits bewährt, und das nicht nur für den MDA-Einsatz.[10] Dank ihr ist es möglich, verschiedene UML-Diagrammtypen um dynamische Aspekte zu bereichern.[11] Sie ist standardisiert, was einen entscheidenden Vorteil gegenüber anderen Sprachen bietet. Zudem ist der Umfang der Sprache weitaus mächtiger, wie hier dargestellt werden kann.

3.4 Zusammenfassung

Dieses Kapitel führte in wichtige Standards und Tools ein, die für den Einsatz von MDA hilfreich sind. So dient der XML Metadata Interchange-Standard als Verarbeitungsformat für die Transformationen. Hierüber können sie ein Modell lesen und manipulieren. Problematisch ist nur, dass XMI derzeit nicht wirklich für den Datenaustausch zwischen den CASE-Tools geeignet ist. Das Java Metadata Interface bietet dem Transformations-Entwickler eine hilfreiche Schnittstelle an, um auf die XMI Daten zuzugreifen. Jedoch empfiehlt es sich, zusätzliche Helper-Klassen zu schreiben, die den Zugriff auf JMI erleichtern. Die Object Constraint Language erweitert das UML-Modell sowohl um Bedingungen, als auch um dynamische Aspekte. Mit ihr lässt sich Plattform-neutral Verhalten spezifizieren, etwa dass am Ende einer Methode ein gewünschter Zustand erreicht werden soll.

[10] Vgl. [BH03] und [Har01].
[11] Vgl. [WK03, S. 50ff].

Kapitel 4

Prinzip des Product Lifecycle Management Service

In der Automobil-Industrie entsteht ein hoher Bedarf danach, mit anderen Zulieferern (OEMs), Herstellern und Partnern zusammenzuarbeiten. Der Austausch der Produktdaten ist dazu eine wesentliche Grundlage für diese Zusammenarbeit. Ein einfaches Beispiel ist der OEM, der für den Automobil-Konzern eine Achse eines PKW-Fahrgestells plant, konstruiert, produziert und pflegt. Um dieser Aufgabe nachgehen zu können, muss der OEM in seinem System die kompletten Daten der Achse besitzen.

Aber auch der Automobil-Hersteller benötigt diese Daten, damit er seine Produktstruktur um die Achse vervollständigen kann. Nur so kann er beispielsweise per DMU prüfen, ob die Achse auch in sein Fahrgestell passt, ohne sie tatsächlich einbauen zu müssen. Zudem sind die Daten auch notwendig, um das Gesamtgewicht des Fahrzeugs zu berechnen.

Für diese Zwecke existieren bereits eingesetzte Standards. Ein Vertreter ist der *PDMEnablers* Standard[1], welcher den Zugriff auf die Daten über eine standardisierte CORBA-Schnittstelle[2] ermöglicht. Die teilnehmenden Partner müssen diese Schnittstelle so implementieren, dass die gewünschten Daten aus ihrem *Product Data Management System* (PDM) beziehungsweise *Product Lifecycle Management System* (PLM) geliefert werden. Ein weiterer Standard ist *PDTnet*[3]. Dieser beschreibt den Datenaustausch über XML-Dateien. Für PDTnet gibt es zwei wesentliche Anwendungsfälle:[4] Den Upload und den Download von Produktdaten im XML-Format. Letztendlich gibt es den hier vorgestellten Standard des *Product Lifecycle Management Service* (PLM Service)[5]

[1] Vgl. [OMG00].
[2] Vgl. etwa [GT00, S. 164ff].
[3] Vgl. [Hab03].
[4] Vgl. [Hab03, Seite 5].
[5] Vgl. [FHLT03] und [FHH+03].

4.1 Anforderungen an den PLM Service

Der *Product Lifecycle Management Service* (PLM Service) befindet sich derzeit in den letzten Phasen der Standardisierung durch die OMG. Er basiert auf PDTnet und PDMEnablers und hat als Anforderungen:[6]

- Ein Plattform-unabhängiges Referenz-Modell (PIM) zu erzeugen
- Eine Verknüpfung mit ISO 10303-AP214 (STEP) herzustellen
- Kompatible Implementierungen auf unterschiedlichen Plattformen zu unterstützen
- Die Komplexität auf der Client-Seite zu reduzieren
- Implementierungen zu erlauben, die gegenüber PDMEnablers performanter sind.

Diese Anforderungen machen ihn besonders interessant, um die Implementierung über Model Driven Architecture zu realisieren. So existiert bereits ein wohl-definiertes PIM, auf dem die Transformationen aufsetzen können. Daraus resultiert jedoch, dass das Modell nicht geändert werden kann, auch dann nicht, wenn es den Transformations-Prozess erleichtern würde.

Die Verknüpfung mit dem *ISO STEP*[7] Standard bedeutet, dass das PIM einen großen Anteil an Daten-Elementen besitzt und somit stark datenorientiert ist. Damit enthält das PIM größtenteils statische Modelle. Zudem bietet ISO STEP eine Unmenge an Daten-Entitäten, die allesamt ähnlich oder gleich zu übersetzen sind.

MDA ist zudem bestens geeignet, kompatible Implementierungen auf unterschiedlichen Plattformen zu erzeugen. Hierfür kommen je nach Plattform unterschiedliche Transformationen zum Einsatz. Der zweite Teil dieser Arbeit zeigt beispielsweise, dass die Implementierungen trotz unterschiedlicher Kommunikations-Protokolle kompatibel bleiben können.

Auch die Komplexität auf der Client-Seite kann reduziert werden. Diese Arbeit unterstellt, dass der Client auf konventionelle Weise entwickelt wird. Zwar wird im Kapitel 9 ein lauffähiger Client generiert, doch dient dieser nur zur Vervollständigung dieser Arbeit. Aus diesem Grund wird dem Client-Entwickler eine einfache und stabile Schnittstelle zur Service-Schicht geboten. Zudem versteckt die Schnittstelle das eingesetzte Kommunikations-Protokoll, so dass dieses sehr leicht durch ein performanteres austauschbar ist.

Lediglich die letzte Anforderung bekommt für diese Ausarbeitung eine niedrige Priorität. Die Transformationen sind für den Prototypen nicht auf performanten Code getrimmt. Sie sollen hier nur zeigen, wie MDA funktioniert. Grundsätzlich ist es jedoch möglich, auch dieser Anforderung gerecht zu werden.[8]

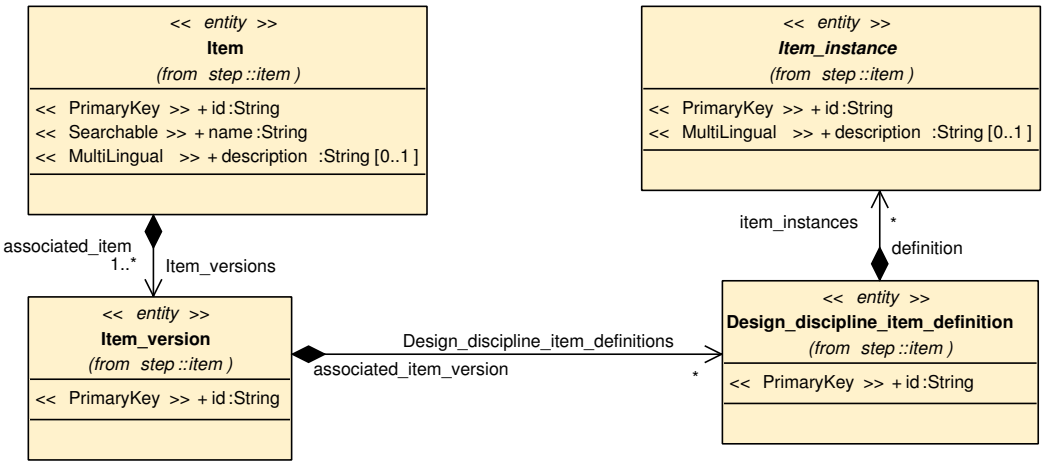
[6] Vgl. [FHH+03, Seite 9].
[7] Vgl. [ISO01].
[8] Vgl. [Dub03, S. 3ff].

4.2 Produktdaten

Nachdem bisher lediglich von Daten gesprochen wurde, die ausgetauscht werden, soll nun kurz genauer auf diese Daten eingegangen werden. Es ist zwingend notwendig, alle angefallenen Daten für ein Produkt zu verwalten, zum Beispiel die oben erwähnte Achse. Das dies in einem Computersystem zu erfolgen hat, ist seit längeren unbestritten. Dieser Abschnitt stellt zwei wesentliche Bestandteile der Produktdaten vor, welche später für den Prototypen verwendet werden. Es sei nochmal ausdrücklich darauf hingewiesen, dass dies nur ein sehr kleiner Ausschnitt aus dem gesamten Spektrum der Produktdaten ist.

4.2.1 Strukturdaten eines Produktes

Informationen über eine Baugruppe, etwa dem Achsgehäuse, sind Inhalt der Produktstruktur. So interessieren neben der Teilenummer etwa das Material, die Ausmaße und das Gewicht. Da ein Bauteil im Laufe seines Lebens weiterentwickelt und verbessert wird, ist es auch notwendig, es zu versionieren. Letztendlich ergibt sich für eine Baugruppe die in Abbildung 4.1 dargestellte Struktur.



Quelle: In Anlehnung an [FHLT03, Seite 17]

Abbildung 4.1: Produktstruktur für eine Baugruppe

Hierzu ist die abstrakte Baugruppe in `Item` gespeichert. Die Versionierung erfolgt dabei über die Klasse `Item_version`. Das tatsächlich vorhandene und einbaubare Teil ist in `item_instance` definiert. Eine Besonderheit ist die Klasse `Design_discipline_item_definition`, über die sich verschiedene Design-Disziplinen abbilden lassen. So wird etwa bei einem teureren Fahrzeug eine teurere Schraube verbaut, als bei einem günstigeren Fahrzeug. Im After-Sales jedoch wäre es zu teuer, sowohl die günstigere als auch die teurere Schraube vorrätig zu halten. Deshalb gibt es im After-Sales für beide Fahrzeuge nur noch eine Schraube. Dies erklärt die Notwendigkeit der Klasse `Design_discipline_item_definition`.

4.2.2 Produktstruktur einer Stückliste

Neben den reinen Daten über ein Bauteil, ist es notwendig, Stücklisten zu verwalten (engl. *Bill of Materials*, BoM). Hierüber lässt sich angeben, dass eine Achse etwa aus einem Achsgehäuse, einer Achswelle und einem Achsschaft besteht. Diese Angabe macht es möglich, eine konkrete Achse zu verwenden und dabei implizit alle enthaltenen Bauteile mit zu verwenden. Auch lässt sich somit das Gesamtgewicht einer Achse berechnen.

Die Stückliste ist jedoch weitaus komplexer, wie hier dargestellt werden kann. So besteht eine Achse aus verschiedenen Varianten, etwa mit einer Welle für höhere Umdrehungszahlen und eine für geringere Umdrehungszahlen. Zudem gibt es die Möglichkeit, ein alternatives, gleichwertiges Bauteil eines anderen Zulieferers zu verwenden, sollte der Hauptlieferant nicht liefern können. Diese Beschreibungen werden unter dem Begriff des Konfigurations-Managements vereinigt. Eine noch komplexere Aufgabe ist es, eine vom Kunden gewählte Konfiguration daraufhin zu prüfen, ob diese überhaupt baubar ist.

Dies sind aber Probleme, mit denen sich bereits andere Projekte mehr oder weniger erfolgreich widmen. Diese Arbeit verzichtet auf die Lösung dieses Problems, zumal es vom PLM Service Standard zu lösen wäre. Auch verwendet diese Arbeit nur einen Auszug aus dem umfangreichen Sortiment der Produktstrukturdaten. Eine umfassende Beschreibung befindet sich in der über 3.500 Seiten starken ISO STEP – Spezifikation [ISO01] und daran angelehnt [Hab03]. [Sta99] bietet eine abstraktere Sichtweise auf dieses Thema.

4.3 Zusammenfassung

Der Product Lifecycle Management Service Standard ist zwar ein neuer Standard, jedoch basiert er auf existierenden Standards. Als MDA-Beispiel eignet er sich jedoch hervorragend. So ist es ein praxisnaher Anwendungsfall, bei dem bereits das PIM schon definiert ist. Durch die Verknüpfung mit ISO STEP ist es ein datenorientierter Ansatz mit vielen, ähnlich zu behandelnden Objekten. Auch können weitere Anforderungen leicht erfüllt werden, etwa eine kompatible Implementierung auf verschiedenen Plattformen und die Reduzierung von Client-Code. Als Beispiel für den Prototypen dient ein Auszug aus dem umfangreichen Spektrum. Hier soll die Generierung für die Produktstruktur erfolgen.

Teil II

Prototyp-Implementierung

Kapitel 5

Entwurf der Transformationen

Nachdem das Modell bereits definiert ist und nur durch die OMG geändert werden kann, werden nun die Transformationsregeln zum Modell erstellt. Dabei sollen jedoch vier Randbedingungen gesetzt werden, um ein erhöhtes Maß an Flexibilität zu erreichen.

Die erste Bedingung ist, dass die Software einer Drei-Schichten-Architektur entspricht und sich somit aus den Schichten Datenverwaltung, Service-Funktionen und Client zusammensetzt. Alle drei Schichten sollen dabei strikt voneinander getrennt sein und jederzeit ausgetauscht werden können. Sollte also der Anwender statt einer Datenbank sein Legacy-System als Datenspeicher anbinden wollen, so darf dies keine Auswirkungen auf die Transformations-Regeln der Service- und Client-Schicht haben. Gleiches gilt für den Fall, dass die Service-Funktionen statt via RMI über Web Services angesprochen werden sollen.

Erschwert wird dies von einer zweiten Bedingung. Es soll nämlich noch immer möglich sein, einzelne Schichten von Hand programmieren zu können — auch wenn dies nicht gänzlich dem MDA Gedanken folgt. Hierbei soll es vor allem den Client-Entwickler unberührt lassen, über welches Protokoll die Service-Schicht anzusprechen ist. Wird das Protokoll ausgetauscht, so darf sich am Client nichts ändern.

Die zweite Bedingung liegt darin begründet, dass einzelne Aufgaben nur mit erhöhtem Aufwand per MDA einzubinden sind. So ist eine benutzerfreundliche Oberfläche ein wesentlicher Grundsatz der Software-Entwicklung. Die exakte Position eines Eingabefeldes per MDA zu bestimmen, ist zwar möglich, doch viel zu abstrakt. Zudem würden sich so zwei Spezialgebiete miteinander vermischen: die *Software-Modellierung* und das *Design der Oberfläche*. Ein Lösungsansatz, wie dieses Problem doch per MDA gelöst werden kann, zeigt das Kapitel 9 auf Seite 64 für die Client-Transformation.

Ein weiterer Grund, warum Teile der Software doch nicht mit diesem MDA Ansatz entwickelt werden, liegt darin, dass teilweise bereits Schnittstellen und Wrapper zum Legacy-System bestehen. Damit kann die Service-Schicht auf die vorhandene Software zurückgreifen.

Die dritte Bedingung, die zu beachten ist, soll verschiedene Implementierungen erlauben. So soll die Service-Schicht sowohl in Java als auch in C#/.Net entwickelt werden können. Damit können mit dem

modellierten PIM auch jene Kunden mit Software beliefert werden, die unterschiedlichen Hardware- oder Software-Restriktionen unterliegen und somit unterschiedliche Implementierungen benötigen.

Die vierte Bedingung fordert, dass möglichst viele Transformationen auch für andere Plattformen wiederverwendet werden können. Dies betrifft natürlich nur die PIM-nach-PSM – Transformationen, da die Code-generierenden zu speziell sind.

Kapitel 6

Vorverarbeitung

Das zu verwendende PIM ist von der OMG festgelegt und kann nicht ohne weiteres verändert werden, nur um den Transformations-Prozess zu erleichtern. Aus diesem Grund kommt vor den eigentlichen Transformationen eine Vorverarbeitung. Diese Kapitel nennt einige vorverarbeitende Transformationen, die

- das PIM daraufhin anpassen, das nachfolgende Transformationen einsetzbar sind,
- es dem PIM-Designer erlauben, Vereinfachungen beim Design vorzunehmen oder
- den nachfolgenden Transformationen die Weiterverarbeitung erleichtern.

6.1 Manipulation des Namensraums

Die einfachste Vorverarbeitung betrifft den Namensraum der Applikation. Der Namensraum innerhalb des UML Modells kann anders sein, als der im späteren Programmcode. Dies erleichtert die Übersichtlichkeit im PIM und stellt dennoch sicher, dass die spätere Applikation einen eindeutigen Namensraum hat.

Die zuständige Transformations-Regel muss dazu den gewünschten Namensraum im UML Modell anlegen. Beispielsweise sollen alle UML Packages in den Namensraum `org.schreinert.masterthesis.step` verschoben werden. Hierzu müssen zuerst alle Sub-Domänen angelegt werden, also ein Package „`org`" im Wurzelverzeichnis, gefolgt von „`schreinert`" als Sub-Package von „`org`" und so weiter.

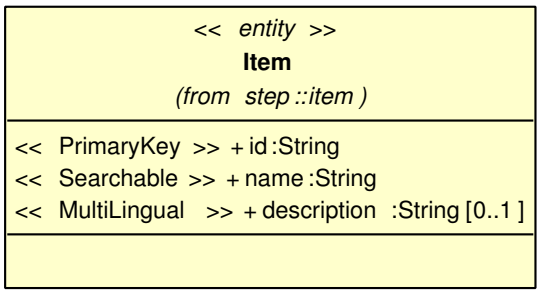
Beispiel-Programm 1 Manipulation des Namensraums

Zuerst muss das Root-Package gefunden werden. Dies kann dadurch erfolgen, dass ein beliebiger Classifier genommen wird. Die Methode `getNamespace():Namespace` liefert daraufhin den Namensraum des Classifiers zurück. Wird nun diese Methode auf jedem zurückgelieferten Namensraum ausgeführt, bis die Methode `null` zurückliefert, so ist der zuletzt gelieferte Namensraum der gesuchte Root-Namensraum.
Neue Packages werden über die statische Methode `UmlPackageClass.createUmlPackage(..)` angelegt.
Anschließend muss das `UmlPackage` mit dem Datenmodell in das gewünschte `UmlPackage` verschoben werden. Dies geschieht dadurch, dass die Methode `setNamespace(Namespace)` auf dem betroffenen `UmlPackage` aufgerufen wird.

6.2 Markieren der Klassen und Attribute

Es wird unterstellt, dass der PLM Service Standard nur ein Teil des UML-Modells ist und sich noch andere Klassen im Modell befinden. Da die folgenden Transformationen nur für das PLM Service-Modell angewandt werden sollen, sind die PLM Service Klassen als solche zu markieren.

Die einfachste Art ist es, jede PLM Service Klasse mit einem Stereotypen zu kennzeichnen. In den nachfolgenden Modellen werden diese mit «`entity`» markiert, wie in Grafik 6.1 zu sehen ist. Außerdem werden die Klassenattribute ebenfalls um passende Stereotypen erweitert. So erhält das Attribut `Item::id` den Stereotypen «`PrimaryKey`», welcher kennzeichnen soll, dass `id` als primäres Attribut eines Objektes dient. Das Attribut `Item::name` wird mit dem Stereotypen «`Searchable`» markiert, was bedeutet, dass nach diesem Attribut gesucht werden kann. Die Client-Transformation weiß somit, dass hierfür z.B. eine Suchfunktion bereitzustellen ist. Der Stereotyp «`MultiLingual`» stellt eine Hilfe für den PLM Service Designer dar und markiert das Attribut `Item:description` als mehrsprachig.



Quelle: Eigene Grafik

Abbildung 6.1: Klasse „`Item`" mit Stereotypen nach der Vorverarbeitung

Die Vorverarbeitung könnte dabei von Hand vorgenommen werden, indem der Designer das UML Modell entsprechend erweitert. Alternativ lässt sich das Modell auch maschinell anpassen, indem

etwa aus einer Properties-Datei die Werte ausgelesen werden. Die Properties-Datei muss natürlich vom Entwickler gepflegt werden.

Beispiel-Programm 2 Setzen von Stereotypen

Über JMI ist es relativ einfach, einem Modell-Element einen Stereotypen zuzuordnen. Das nachfolgende Beispiel zeigt, wie die Klasse `entity` den Stereotypen mit Namen `"entity"` erhält. Die Helper-Methode `getStereotype(String):Stereotype` liefert dazu den Stereotypen zum angegebenen Namen. Ist der Stereotype nicht vorhanden, wird er erzeugt.

```
getCore().getAStereotypeExtendedElement().add(
      getHelper().getStereotype("entity"), entity);
```

Alternativ zu den Stereotypen, ließen sich die Klassen und Objekte auch anderweitig markieren. So könnte jede PLM Service Klasse eine Tagged-Value erhalten, der die Klasse als Entität definiert. Gleiches lässt sich mit den Attributen machen, so dass es etwa eine Tagged-Value namens „`isPrimaryKey`" mit den Werten `true` oder `false` gibt. Der Vorteil dieses Vorgehens wäre, dass keine mehrfachen Stereotypen vergeben werden müssen. Denn ist beispielsweise ein Attribut sowohl «`Searchable`», als auch «`MultiLingual`», dann würde es im obigen Ansatz zwei Stereotypen erhalten. Dies ist vor allem für UML 1.x sehr ungewöhnlich. Mit zunehmender Anzahl Stereotypen pro Element leidet zudem auch die Übersichtlichkeit. Innerhalb des JMI-Frameworks bedarf es jedoch eines erhöhten Aufwands, Tagged-Values zu einem Element aufzufinden.

Eine weitere Alternative wären leere Interfaces. So implementieren alle Klassen des PLM Service Standards dieses methodenlose Interface, womit die Klassen wie gewünscht markiert wären. Aber auch hier gilt, dass es deutlich aufwändiger ist, PLM Service Klassen zu finden und weiter zu verarbeiten, als der zuerst aufgeführte Lösungsansatz. Außerdem ist dieser Ansatz nur mit großen Einschränkungen auf Attribute und ihren Datentypen anwendbar und daher weitestgehend sinnfrei.

Ebenso der Ansatz, Modell-Elemente per Constraint zu kennzeichnen. Zum einen ist dies nur eingeschränkt möglich, etwa um primäre Attribute zu definieren — jedoch ist dieses Verfahren für durchsuchbare Attribute nicht möglich. Zum anderen bringt die automatische Constraint-Programmierung nur dann einen Vorteil, wenn entsprechende Compiler vorhanden sind. Solch ein Compiler müsste dann Programm-Code erzeugen, der etwa die Searchable-Eigenschaft sicherstellt. Hier ist es bedeutend einfacher und unabhängiger, in den eigenen Transformationen diese Eigenschaft sicherzustellen, indem der entsprechende Stereotyp ausgewertet wird.

6.3 Erzeugen des Primary Keys

Die konkreten Objekte, etwa eines `Items`, müssen sich vom Client genau identifizieren lassen. Somit ist es nötig, primäre Attribute jeder Klasse zu kennzeichnen. Zwar werden die Attribute durch den Entwickler bzw. Designer markiert, jedoch besteht die Gefahr, dass eine Klasse kein primäres Attribut besitzt, etwa wenn sie attributlos ist. Aus diesem Grund stellt diese Transformation sicher, dass jedes Objekt eindeutig identifizierbar ist.

Hierzu ist über alle Klassen zu iterieren, die mit «`entity`» als PLM Service Klasse gekennzeichnet sind. Eine Iteration über alle enthaltenen Attribute gibt darüber Aufschluss, ob ein Primary Key definiert ist. Im negativen Fall ist ein neues Attribut zu erzeugen und mit dem Stereotyp «`PrimaryKey`» zu kennzeichnen.

Dieses Attribut wäre jedoch kein Bestandteil des PLM Service Standards. Das bedeutet, dass der Wert des Attributs nirgends definiert bzw. gesetzt wird. Deshalb ist es nötig, dieses Attribut zusätzlich mit dem Stereotyp «`AutoID`» zu markieren. Diese Information benutzt etwa die Transformation der Daten-Schicht, indem das CREATE-TABLE SQL-Skript das Attribut als `auto_increment` markiert. Damit stellt das Datenbanksystem die Eindeutigkeit der Datensätze sicher. Aber auch der Client nutzt diese Information. So versteckt er das AutoID - Feld vor dem Anwender, da dieser mit dem Wert nur wenig anfangen kann.

Alternativ zu dem neu eingefügten Attribut, könnte die Primary Key-Eigenschaft auch später geprüft werden. Beispielsweise ließe sich auch dann ein AutoID Attribut anlegen, wenn tatsächlich das SQL-Skript erzeugt wird. Das hätte aber zur Folge, dass auch in anderen Transformationen diese Primary Key-Eigenschaft zugesichert werden muss, da diese konkrete Objekte referenzieren müssen. Dies bedeutet, dass die selbe bzw. sehr ähnliche Funktionalität an verschiedenen Stellen zu implementieren und zu warten wäre. Ebenso erhöht diese Alternative die Anzahl von Ausnahme-Bedingungen, da immer wieder die Primary Key-Eigenschaft zu prüfen wäre.

6.4 Manipulation der Enumerations

Enumerations sind Klassen, die im PIM mit dem Stereotypen «`Enumeration`» gekennzeichnet sind und lediglich Attribute enthalten. Dabei ist der Typ der Attribute unwichtig und wird auch nicht weiter beachtet. Lediglich die Attribut-Namen sind von Interesse. Enumerations werden später anders behandelt, als normale Klassen. Für das Datenbank-Skript werden Enumerations etwa in `ENUM ('attribute1', 'attribute2', '...')` transformiert. Für die Service-Schicht wird hierfür zum Beispiel eine Java-Klasse generiert, mit der sich ein Wert selektieren und abfragen lässt. In der Client-Schicht werden diese Objekte nicht als Text-Box dargestellt, sondern als Auswahl-Box (Combo-Box / SELECT-Feld).

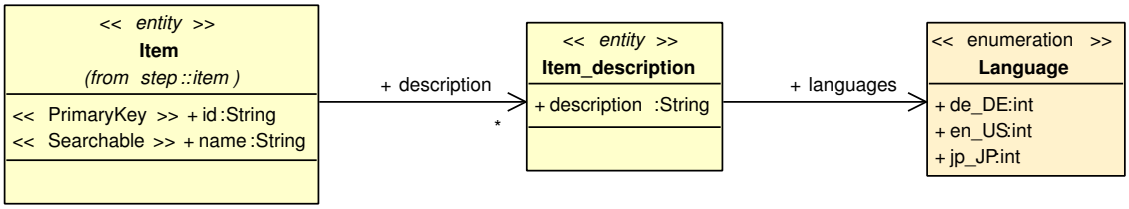
Für den Prototypen werden die Enumerations in eine normale Klasse umgewandelt. Jedes Attribut wird dabei in die neue Klasse übernommen, wobei der Namen noch anzupassen ist. Letzteres liegt daran, dass Enumerations auch Leerzeichen und ähnliches im Namen enthalten dürfen. Dies soll aber nicht zugelassen werden. Als Datentyp wird `Integer` verwendet und jedes Attribut erhält einen einmaligen, initialen, unveränderlichen Wert.

Zusätzlich erhält die neue Klasse ein Attribut, in dem das aktuell ausgewählte Element gespeichert wird. Über die Zugriffsmethoden lässt sich das selektierte Token des Enumerations lesen und verändern. Es ist jedoch immer noch nötig, diese Klasse als Enumeration zu kennzeichnen, da eben das Datenbank-Skript Enumerations anders behandelt. Für die Service-Schicht ist das Enumeration jedoch eine normale Klasse und kann wie diese transformiert werden.

6.5 Hinzufügen von Mehrsprachigkeit

Eine Erleichterung für den Modellierer bietet sich dadurch an, eine Art Makro bzw. Prä-Prozessor bereitzustellen. So sind im PLM Service Standard einige Attribute als mehrsprachig definiert, etwa das `description`-Feld der Klasse `Item`. Anstatt diese Mehrsprachigkeit im UML Diagramm für jede betroffene Klasse aufwändig zu modellieren, können diese Attribute als «`MultiLingual`» gekennzeichnet werden.

Statt dem einen Attribut `description` gibt es nun eine neue Klasse, welche die Sprachinformationen enthält. Als Namen der Klasse empfiehlt sich, den ursprünglichen Klassennamen mit dem Attribut-Namen zu kombinieren, etwa `Item_description` oder `Item_description_lang`. Der Datentyp ist der selbe, wie vom ursprünglichen Attribut. Die neue Klasse enthält als Attribut den dazugehörigen Text `description`, sowie einen Verweis auf die Sprache, wie in Abbildung 6.2 zu sehen ist.



Quelle: Eigene Grafik

Abbildung 6.2: Mehrsprachigkeit erzeugen

Das Attribut für die aktuelle Sprache könnte vom Typ `String` sein und die entsprechende Sprachen-ID enthalten. Alternativ ist dieses Attribut vom Typ `Language`, einer neu zu erzeugenden Klasse. `Language` ist dabei ein Enumeration und enthält alle möglichen Sprachen-IDs.

6.6 Zusammenfassung

Dieses Kapitel zeigte erste, kleinere Transformationen am PIM. Diese dienen dazu, das Modell für die nachfolgenden Transformationen zu prüfen und anzupassen. Zudem kann so dem Designer Arbeit abgenommen werden. So lässt sich der Namensraum entsprechend der zu erstellenden Applikation verändern, die Klassen und Attribute können gekennzeichnet werden oder es wird bei Bedarf ein primäres Attribut erzeugt. Auch denkbar ist es, Enumerations in normale Klassen umzuwandeln, um sie weitestgehend wie normale Klassen transformieren zu können.

Kapitel 7

Transformationen der Daten-Schicht

Nachdem im ersten Schritt das Modell grundlegend geprüft und angepasst wurde, kann nun die eigentliche Transformation geschehen. Diese erste Transformation ist relativ einfach durchzuführen und zeigt dennoch die grundlegenden Möglichkeiten des MDA-Ansatzes. In diesem Kapitel soll dazu gezeigt werden, wie aus dem PIM ein fertiges SQL-Schema erzeugt wird. Der folgende Abschnitt definiert dazu die Aufgaben und Anforderungen dieser Transformation. Die diesem folgenden zwei Abschnitte zeigen die PIM-nach-PSM – und PSM-nach-SQL – Transformation.

7.1 Aufgaben und Anforderungen

Da das Ausgangsmodell schon ein Datenmodell ist — nämlich das für den Product Lifecycle Management Service —, muss nicht über Umwege ein Datenbank-Schema entworfen werden. Es ist jedoch zu prüfen, ob das in UML modellierte Modell in eine Datenbank-typische Modellierungssprache zu überführen ist.

Hier bietet sich beispielsweise das Entity-Relationship-Diagramm (ERD) oder verwandte Dialekte an. Vorteile eines solchen Modells wären unter anderem, dass Datenbank-Experten das Modell schneller verstehen und validieren können. Außerdem gibt es Tools, die aus einem ER-Diagramm fertigen SQL-Code erzeugen können. Solche Tools sind bereits heute sehr ausgereift und unterstützen den Datenbank-Designer mit zahlreichen Hilfsmitteln, etwa indem sie SQL-Code für unterschiedliche Datenbanksystem erzeugen oder indem sie Verbesserungen am Schema vorschlagen.

Diese Arbeit verzichtet jedoch gezielt darauf, den UML-Weg zu verlassen, auch wenn dies durch MDA durchaus angedacht ist. Außerdem sollen so wenig Tools wie möglich verwendet werden, um die Möglichkeiten der MDA besser demonstrieren zu können.

Aber auch wenn das vorliegende Ausgangs-Modell primär ein Datenmodell ist, sind noch zahlreiche Änderungen vorzunehmen. Zur einfacheren Demonstration wird darauf verzichtet, das Datenbank-Schema zu optimieren. Wie bereits angeführt, gibt es Tools, welche Datenbank-Entwürfe optimieren können. Für einen produktiven Einsatz kann auf solche Routinen zurückgegriffen werden. Damit bleiben im Wesentlichen die folgenden Aufgaben zu bearbeiten:

1. Alle benötigten Entitäten identifizieren und markieren
2. Die PIM-Attribut-Typen in Datenbank-verträgliche Datentypen umzuwandeln
3. Die Relationen zwischen den Klassen aufzulösen
4. Vererbungen im Klassen-Modell aufzulösen
5. Attribut-Namen SQL-konform zu setzen

7.2 PIM-nach-PSM – Transformation

Dieser Abschnitt stellt die nötigen Transformationen vor, die zum Erzeugen des Datenbank-Schemata notwendig sind.

7.2.1 Identifizierung der benötigten Entitäten

Auf dem ersten Blick scheint klar zu sein, für welche Klassen die Datenbank-Tabellen entstehen müssen. Immerhin sind alle Klassen, die zum PLM Service Modell gehören als solche markiert (siehe Abschnitt 6.2 auf Seite 32). Jedoch referenzieren einige PLM Service-Klassen auf Klassen, die nicht als solche gekennzeichnet sind.

Beispielsweise die Klasse `numerical_value`, welche eine Zahl mit einer Einheit darstellt. Diese Klasse ist nicht Teil des PLM Service Modells[1], wird aber von der Klasse `Item_instance` als Datentyp verwendet. Somit muss auch für diese Klasse eine Datenbank-Tabelle erzeugt werden. Gleiches gilt für die in `numerical_value` referenzierten Klassen.

Aus diesem Grund ist über alle PLM Services-Klassen zu iterieren. Dabei sind alle verwendeten Attribut-Typen entsprechend zu prüfen. Die Code-Beispiele 3 und 4 zeigen die Methodik.

Beispiel-Programm 3 Identifizierung aller Entitäten - Teil 1

Das Array `entities` enthält alle Modell-Elemente, die mit dem Stereotypen «`entity`» markiert sind, wobei hierfür nur `Classifier` interessant sind. Diese werden der privaten Methode `identifyAllEntities` (Code-Beispiel 4 auf der nächsten Seite) übergeben.

```
Object[] entities = getCore().getAStereotypeExtendedElement()
                 .getExtendedElement(entityStereo).toArray();
for ( int elemCounter=0 ; elemCounter<array.length ; elemCounter++ ) {
   ModelElement elem = (ModelElement)entites[elemCounter];
   if ( !( elem instanceof Classifier ) ) {
      continue;
   }
   Classifier entity = (Classifier)elem;
   identifyAllEntities(entity);
}
```

[1] Aussage gilt nur für den vorliegenden Initial Submission des PLM Services Standards (vgl. [FHLT03]).

Beispiel-Programm 4 Identifizierung aller Entitäten - Teil 2

Die Methode `identifyAllEntities` betrachtet alle Attribute und Assoziationen eines UML-Classifiers. Interessant sind dabei nur solche, die auf einen anderen `Classifier` verweisen, der nicht mit «`entity`» markiert sind.
Gefundene `Classifier` werden daraufhin mit dem Stereotypen «`table`» markiert. Anschließend werden auch diese neu identifizierten Klassen auf jene Referenzen überprüft.

```
Object[] array = getHelper().getAttributes(entity).toArray();
for ( int objCounter=0 ; objCounter<array.length ; objCounter++ ){
   Attribute elem = (Attribute)array[objCounter];
   Classifier type = elem.getType();
   if ( type instanceof DataType ) {
      continue;
   }
   if ( type.getStereotype().contains(entityStereo) ||
        type.getStereotype().contains(tableStereo) ) {
      continue;
   }
   getCore().getAStereotypeExtendedElement().add(tableStereo, type);
   // identify recursive
   identifyAllEntities(type);
}
```

Damit sind sämtliche Klassen im SQL-PSM entweder mit «`entity`» markiert oder aber mit «`table`». Die nachfolgenden Operationen und Transformationen beziehen sich nur auf solche Klassen.

7.2.2 Umwandlung der Attribut-Typen in Datenbank-Typen

Im nächsten Schritt müssen die Datentypen des PIM in SQL-verträgliche Datentypen umgewandelt werden. So wird beispielsweise aus dem Datentyp `int` oder `Integer` der SQL-Typ `INTEGER`. Komplizierter wird es aber bei dem Datentypen `String`. Hierfür bietet sich der Datentyp `VARCHAR` an. Es ist jedoch unbekannt, wie lang der Text sein kann und wie lang folglich das Datenfeld sein muss. Aus diesem Grund verwendet die hier gezeigte Transformation ein dreistufiges Modell.

Im ersten Schritt durchsucht das Transformationstool eine Properties-Datei. In dieser ist der Datentyp festgelegt, der für ein spezifisches Attribut einer spezifischen Klasse zu verwenden ist. So ist etwa definiert, dass das Attribut `id` der Klasse `Item` zu `VARCHAR(13)` wird und `name` zu `VARCHAR(50)`.

In einem zweiten Schritt lassen sich die SQL-Datentypen einzig anhand des Attribut-Namens in Kombination mit einem Datentypen bestimmen. So sollen alle Attribute mit Namen `description` und Typ `String` nach `TEXT` umgewandelt werden.

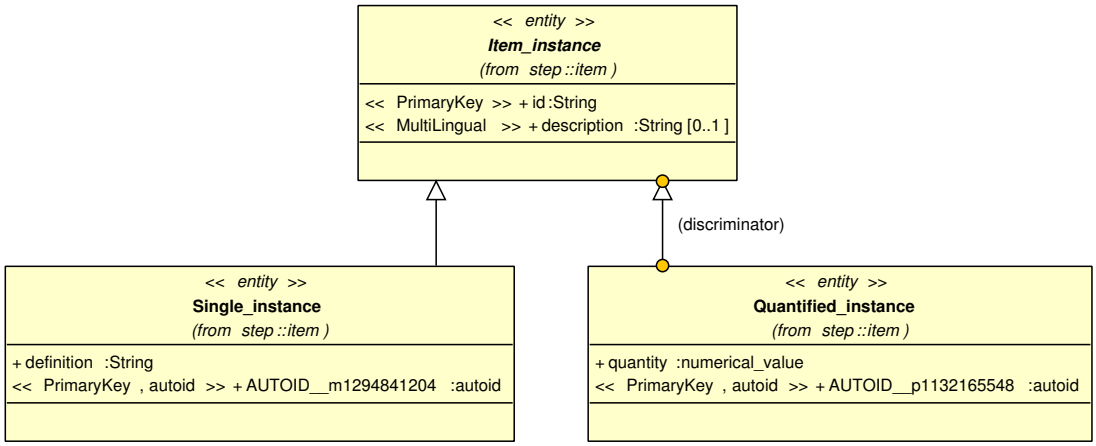
Der letzte Schritt wandelt alle verbliebenen Datentypen pauschal um. Beispielsweise wird der Typ `String` in `VARCHAR(255)` umgewandelt und `Double` in `DOUBLE(10,10)`.

Für die Umsetzung muss zuerst für jeden SQL-Datentyp ein entsprechender Datentyp im JMI erzeugt werden. Da der Zugriff auf einen Datentypen ähnlich läuft, wie der auf einen Stereotypen, gibt es

auch hierfür eine Helper-Methode. Wie beim Stereotyp (siehe auf Seite 33) liefert die Methode einen `DataType` entsprechend dem angegeben Namen. Ist der gewünschte Datentyp nicht vorhanden, so wird er erzeugt.

7.2.3 Auflösen der Vererbungen

Nach den beiden relativ einfachen Transformationen ist nun das Datenmodell im größeren Umfang zu ändern. Der Product Lifecycle Management Service Standard arbeitet natürlich auch mit Vererbungen. Zum Beispiel generalisiert die abstrakte Klasse `Item_instance` die Klassen `Single_instance` und `Quantified_instance`, wie in Abbildung 7.1 dargestellt.



Quelle: Eigene Grafik

Abbildung 7.1: Problem der Vererbung

Relationale Datenbanksysteme, wie etwa die zur Demonstration verwendete hypersonic[2] beziehungsweise MySQL[3], beherrschen jedoch keinerlei Vererbung. Aus diesem Grund müssen die Vererbungen aufgelöst werden. Es existieren hierfür mehrere Lösungsansätze, die kurz vorgestellt werden sollen.

Die erste Möglichkeit ist, alle Attribute der spezialisierten Klassen in die generelle Klasse zu übernehmen. Damit erhielte die Klasse `item_instance` im obigen Beispiel zusätzlich die Attribute `definition` von `Single_instance` und `quantity` von `Quantified_instance` (Abbildung 7.2 auf der nächsten Seite). Die spezialisierten Klassen sind anschließend ohne Bedeutung und können aus dem SQL-PSM entfernt werden. Die zur spezialisierten Klasse gezogenen Attribute müssten jedoch als optional gekennzeichnet werden, unabhängig von ihrer vorherigen Kardinalität. Denn ist ein Datenbank-Tupel eigentlich vom Typ `Single_instance`, so bliebe das Feld `quantity` ungesetzt, also `NULL`.

Je nach Anzahl der spezialisierten Klassen und deren Attribute würde somit die Datenbank-Tabelle zahlreiche nutzlose Felder speichern müssen. Ebenso fällt es schwer, die Datenbank-Integrität mit den Datenbank-Constraints und Foreign Keys sicherzustellen. Denn beispielsweise bleibt es dem Anwender bzw. der Anwendung überlassen, ob ein Feld mit einem Wert oder mit `NULL` belegt wird.

[2] `http://hsqldb.sourceforge.net/`, besucht am 22.04.2004.
[3] `http://www.mysql.com`

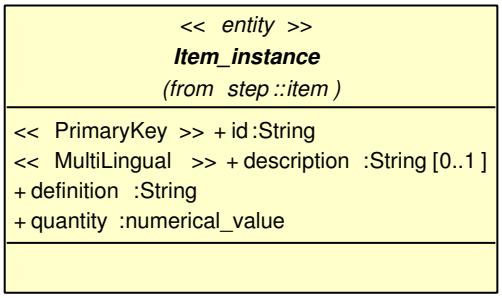


Abbildung 7.2: Generalisierungen auflösen — Variante 1

Eine weitere Alternative geht den umgekehrten Weg. Hier erhalten die spezialisierten Klassen die Eigenschaften der Eltern-Klasse. Im Beispiel würden demnach die Klassen `Single_instance` und `Quantified_instance` jeweils die Attribute `id` und `description` von `Item_instance` übernehmen und `Item_instance` könnte gelöscht werden (Abbildung 7.3). Damit wäre das Problem gelöst, dass Datenfelder zwangsweise optional sein müssten. Jedoch treten Probleme auf, wenn andere Klassen auf die generalisierte Klassen referenzieren. Diese Referenz müsste geändert werden auf entweder `Single_instance` oder `Quantified_instance`. Da dies zum jetzigen Zeitpunkt nicht feststeht, generiert diese Alternative weitere Probleme.

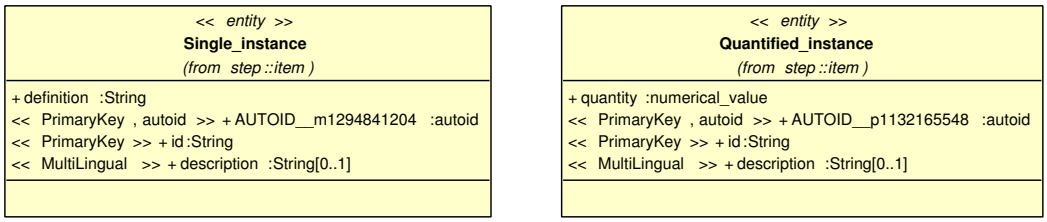


Abbildung 7.3: Generalisierungen auflösen — Variante 2

In einer dritten Variante bleiben alle Klassen und Referenzen erhalten. Es werden lediglich die Generalisierungen durch Kompositionen ersetzt. Damit ist beispielsweise die Klasse `Single_instance` Teil von `Item_instance` (vgl. Abbildung 7.4 auf der nächsten Seite) . Die Komposition stellt zudem sicher, dass die Referenz der Sub-Klassen auf `Item_instance` zum primären Schlüssel für die Sub-Klassen wird. Für die spätere SQL-Beschreibung bedeutet dies, dass in der Tabelle `Single_instance` eine nicht-`NULL` Referenz auf `Item_instance` enthalten ist. Über diese Referenz sind die Daten der Super-Klasse zugreifbar.

Problematisch ist es, wenn Referenzen von anderen Klassen auf eine Super-Klasse vorhanden sind. Gäbe es nur Referenzen auf die tatsächlichen Klassen, etwa `Single_instance`, so könnten alle Informationen ohne Probleme gelesen werden. Lediglich die Daten aus der Super-Klasse wären etwa per JOINT zu lesen. Ein Verweis auf die Super-Klasse bedeutet jedoch, dass zuerst die tatsächliche Implementierung gefunden werden muss.

Ein weiterer Ansatz verschönert die dritte Alternative. Da die Informationen der Super-Klasse nur per

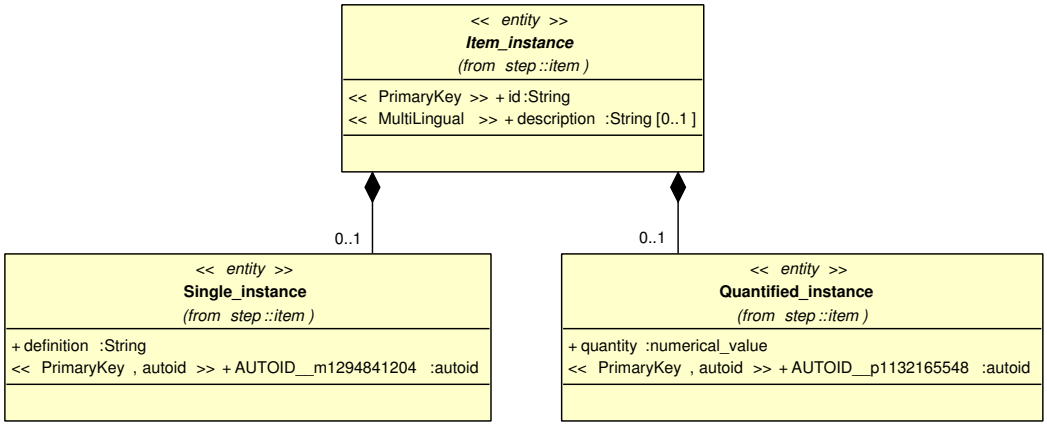


Abbildung 7.4: Generalisierungen auflösen — Variante 3

Referenz zugreifbar sind, müssten diese Informationen entweder per zweiter SELECT-Abfrage geschehen oder aber per JOINT. Um diese Logik aus der Service-Schicht zu nehmen, könnte zusätzlich für jede Sub-Klasse eine View auf diese eingerichtet werden. In dieser View wären dann auch die Informationen der Super-Klasse direkt sichtbar. Referenzen auf die Sub-Klasse würden dann auf die View umgebogen werden.

Die hier vorgestellte Transformation des PIMs in das SQL-PSM bedient sich der dritten Alternative, wie im Beispiel-Code 5 auf der nächsten Seite verdeutlicht. Auf die Verschönerung durch die View wurde verzichtet, da neben dieser noch weitere Verbesserungen am Datenbank-Schema möglich wären. Solche Optimierungen sollen jedoch nicht Aufgabe dieser Arbeit sein. Diese können von Tools beziehungsweise von deren Routinen erledigt werden, wie in der Einleitung zum Abschnitt 7.1 auf Seite 36 bereits angeführt wurde.

Beispiel-Programm 5 Auflösen der Generalisierungen

Es werden zuerst alle Generalisierungen gesucht und geprüft, ob diese zu Classifiern verweisen.

```
Object[] generalization = entity.getGeneralization().toArray();
for ( int generalizationCounter=0 ;
      generalizationCounter<generalization.length ;
      generalizationCounter++) {
    Generalization gen = (Generalization)
           generalization[generalizationCounter];
    if ( !( gen.getParent() instanceof Classifier) ) {
        continue;
    }
    Classifier child = (Classifier)gen.getParent();
```

Für jede gefundene Generalisierung wird ein neue Assoziation erzeugt und mit „`specialization`" benannt.

```
    UmlAssociation asso = getCore().getUmlAssociation()
                   .createUmlAssociation();
    asso.setName("specialization");
```

Für beide Klassen wird jeweils ein Endpunkt für die Assoziation erzeugt. Der Endpunkt für die Sub-Klasse ist dabei eine Komposition mit der Kardinalität 1.

```
    AssociationEnd from = getCore().getAssociationEnd()
                   .createAssociationEnd();
from.setAggregation(AggregationKindEnum.AK_COMPOSITE);
    from.setMultiplicity(getHelper().createMultiplicity(1,1));
    from.setNavigable(false);
    from.setVisibility(VisibilityKindEnum.VK_PUBLIC);
    from.setParticipant(entity);
```

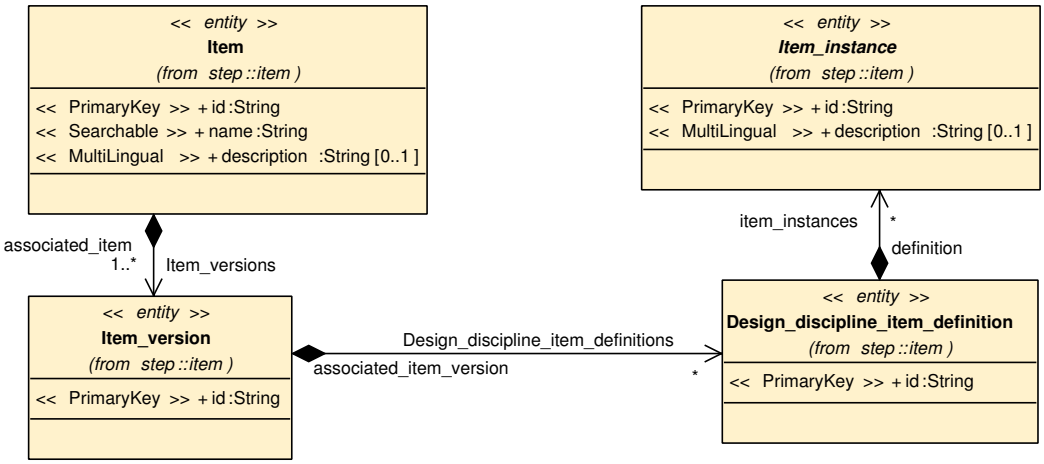
Für die Super-Klasse ist der Endpunkt optional und navigierbar.

```
    AssociationEnd to = getCore().getAssociationEnd()
                .createAssociationEnd();
    to.setAggregation(AggregationKindEnum.AK_NONE);
    to.setMultiplicity(getHelper()
                .createMultiplicity(0,1));
    to.setNavigable(true);
    to.setVisibility(VisibilityKindEnum.VK_PUBLIC);
    to.setParticipant(child);
    getCore().getAAssociationConnection().add(asso, from);
    getCore().getAAssociationConnection().add(asso, to);
```

Anschließend wird die Generalisierung entfernt und eine Dependency gesetzt. Letztere soll späteren Transformationsschritten weitere Informationen liefern. Zudem ist dies ein Weg dahin, dass ein Entwickler den Weg vom PSM zurück zum PIM verfolgen kann, um so Fehler leichter zu finden (siehe 10.3.5 auf Seite 77).

7.2.4 Kürzen der Attribut-Namen

Die vorliegende Version des PLM Service Standards verwendet zahlreiche Kompositionen zwischen einzelnen Klassen. Und dies aus gutem Grund. Beispielsweise bezieht sich eine `Item_instance` immer auf eine `Design_discipline_item_definition`, diese wiederum auf eine `Item_version` und diese letztendlich auf ein `Item` (vgl. Abbildung 7.5). Dies bedeutet, dass etwa eine `Item_version` nur dann existieren kann, wenn es ein entsprechendes `Item` gibt.



Grafik: Nach [FHLT03, S. 17]

Abbildung 7.5: Kompositionen zwischen *Item*, *Version*, *Design Discipline* und *Item Instance*

Für den Datenbank-Entwurf bedeutet dies, dass der primäre Schlüssel der Klasse `Item_version` zusätzlich `Item` enthält. Im konkreten Beispiel heißt dies, dass der primäre Schlüssel aus den Attributen `id` von `Item_version` und `id` von `Item` besteht, da beide Attribute mit dem Stereotypen «`PrimaryKey`» gekennzeichnet sind. Damit muss das Datenbank-Schema etwa folgendermaßen in SQL ausschauen:

```
CREATE TABLE item_version (
   -- the id as part of item_version
   id VARCHAR(3) NOT NULL,
   -- the primary key of item (here it is 'id')
   -- the association to item is named associated_item
   -- so we named it associated_item__id
   associated_item__id VARCHAR(13) NOT NULL
                       REFERENCES item(id),
   -- TODO: ... FOREIGN KEY ...
   PRIMARY KEY ( id, associated_item__id )
)
```

Das gezeigte SQL-Statement soll primär zeigen, dass eine Referenz auf die assoziierte Klasse besteht (`associated_item__id`). Dabei muss ein eindeutiger Attributname dafür gefunden werden, wie das

primäre Attribut `id` der assoziierten Klasse `Item` in das Schema für `Item_version` aufzunehmen ist. Da die Beziehung zu `Item` mit `associated_item` benannt ist, könnte dieser Name verwendet werden. Im angeführten Beispiel würde dies auch problemlos funktionieren. Wären jedoch zwei Attribute primärer Schlüssel einer Klasse, so müsste der Name erweitert werden. Im Beispiel wurde daher der Attributname `associated_item__id` verwendet — er ist zusammengesetzt aus dem Namen der Assoziation `associated_item` und dem Namen des primären Attributs `id` der Klasse `Item`.

Für die Klasse `Design_discipline_item_definition` hätte dies zur Folge, dass die Referenz auf `Item_version` inzwischen zwei Attribute enthielte. Für das Attribut `id` in `Item_version` wäre das SQL-Schema:

```
associated_item_version__id VARCHAR(3) NOT NULL
                            REFERENCES item_version(id)
```

Hierbei ist `associated_item_version` der Name der Assoziation zu `Item_version`. Auch das zweite, primäre Attribut von Item_version erhält diesen Präfix, jedoch ist der Name dieses Attributs in Item_version `associated_item__id`, wodurch sich der relativ lange Name `associated_item-_version__associated-_item__id` ergibt. Das SQL-Schema für diesen Eintrag wäre:

```
associated_item_version__associated_item__id VARCHAR(13) NOT NULL
                    REFERENCES item_version( associated_item__id )
```

Mit zunehmender Anzahl zusammenhängender Kompositionen tritt demnach die Eigenschaft auf, dass die Namen der Attribute immer länger werden. So erreicht die Länge sehr schnell ein Limit, welches vom Datenbankmanagementsystem gesetzt ist, nämlich die maximale Länge des Attribut-Namens.

Aus diesem Grund ist es erforderlich, von dem oben beschriebenen Verfahren wegzukommen und stattdessen kürzere IDs zu verwenden. Deshalb werden Hash-Werte der Namen verwendet. Aus `associated_item_version__associated_item__id` würde somit z.B. `associated_item_version-__p253489508`. Auch dies sorgt für Probleme. Es ist nämlich nicht der Hash-Wert für `associated-_item__id` zu bilden. Denn auch dieser Attribut-Name wurde auf den Hash-Wert `associated_item-__p33554` gekürzt.

Das genaue Vorgehen hierfür ist, zuerst den Hash-Wert von `id` zu berechnen und damit `associated-_item__p3355` in der Klasse `Item_version` zu erhalten. Für diesen Namen wird dann für `Design-_discipline_item_definition` erneut der Hash-Wert erzeugt, wodurch das Attribut als `associated_item_version__p253489508` benannt wird. Dies muss immer wieder für jeden Schritt berechnet werden, was bei der späteren automatischen Code-Erzeugung zu großen Problemen führte. Damit ist es aus dem UML-Modell heraus sehr schwer zu sagen, zu welcher Verknüpfung ein Attribut gehört.

In der Beispiel-Implementierung für diese Transformation wurde deshalb darauf verzichtet, die Referenzen im SQL-PIM zu kürzen. Dies hat jedoch den Nachteil, dass alle Referenzen zu einem späteren Zeitpunkt mit IDs zu versehen war, was bei dem verwendeten Verfahren nicht einfach war.

Eine bessere Lösung wäre gewesen, wären die Attribute doch im SQL-PSM gekürzt gewesen. Das Problem mit der fehlenden Zuordnung könnte mit Assoziationen zwischen den Attributen gelöst werden. Ein Endpunkt kann dazu ein konkretes Attribut als Teilnehmer benennen, worüber sich diese Abbildung darstellen lässt.

7.2.5 Vereinfachung des SQL-PSMs für die Weiterverarbeitung

Mit dem vorliegendem SQL-PSM könnte bereits problemlos SQL-Code erzeugt werden. Jedoch greifen spätere Transformationen auf dieses Datenschema zu, so dass es für diese vereinfacht werden soll. Ein Problem der Transformationen ist nämlich, dass Operationen auf Attribute ebenfalls auf Assoziationen bezogen sein können, etwa um Zugriffsmethoden zu erzeugen. Dies bedeutet doppelte Programmierarbeit für den Transformations-Entwickler. Um diese redundante Arbeit einzusparen, werden deshalb Assoziationen in Attribute aufgelöst. Somit brauchen nur noch die Attribute betrachtet werden.

Hierzu muss lediglich für jede Assoziation bestimmt werden, in welcher Klasse die Relation gespeichert wird. In dieser Klasse wird dann das entsprechende Attribut erzeugt. Die Entscheidung, in welcher Klasse das Attribut zu erzeugen ist, hängt von zwei Dingen ab. Primär werden Kompositionen in der abhängigen Klasse gespeichert. So ist die Komposition zwischen `Item` und `Item_version` in `Item_version` gespeichert, da `Item_version` ohne `Item` nicht existieren kann. Dies geschieht so, obwohl die Assoziation eigentlich in Richtung `Item_version` gerichtet ist. Alle anderen Assoziationen werden entsprechend ihrer Richtung gespeichert.

7.3 PSM-nach-SQL – Transformation

Nachdem alles vorbereitet wurde und alle notwendigen Informationen vorhanden sind, kann aus dem PSM ein SQL-Skript erzeugt werden. Hierzu muss jedoch das eingesetzte Transformationstool den entsprechenden Dialekt des Datenbankmanagementsystems sprechen. Aber auch wenn z.B. SQL98-konformer SQL-Code erzeugt wird, können einige Systeme nicht damit umgehen, da diese nur ein eingeschränktes Vokabular besitzen. Da der Prototyp mit einer MySQL-Datenbank laufen soll, erzeugt die PSM-nach-Code – Transformation SQL-Anweisungen speziell für die MySQL-Datenbank. Tools für andere Datenbanksysteme unterscheiden sich aber nur unwesentlich von dem hier gezeigten.

Um den Programm-Code zu erzeugen, muss zuerst über einen Transformations-Parameter der Name der Datei gelesen werden, in die das fertige SQL-Skript geschrieben werden soll. Anschließend sind für jede Klasse, die als «`table`» oder «`entity`» gekennzeichnet sind (siehe Abschnitt 6.2 auf Seite 32), die folgenden Schritte zu unternehmen.

Für jede Klasse wird zuerst der im PIM (und damit auch im PSM) eingetragene Kommentar zu der Klasse ausgegeben — selbstverständlich mit führenden Kommentar-Zeichen. Es ist darauf zu achten, dass in dem Kommentar Zeilenumbrüche vorhanden sein können, so dass für jede neue Zeile Kommentar-Zeichen voranzusetzen sind. Ebenfalls sollte nach beispielsweise 80 Zeichen ein Zeilenumbruch erzwungen werden, damit das SQL-Skript leichter verständlich ist. Dann erfolgt die Ausgabe `CREATE TABLE` gefolgt vom Klassennamen.

Nun müssen die Attribute der Klasse ausgegeben werden. Dazu muss jedoch unterschieden werden, ob es sich hierbei um eine Referenz, eine Enumeration oder ein primitives Attribut handelt. Ein primitives Attribut, wie etwa eines vom Typ `VARCHAR` oder `INTEGER`, kann einfach nach SQL-Syntax ausgegeben werden. Schließlich ist noch auszuwerten, ob das Attribut optional ist. Ist die Angabe erzwungen, so wird zusätzlich `NOT NULL` ausgegeben, sonst `NULL`.

Enumerations lassen sich ähnlich abarbeiten. Hierzu folgt nach dem Attribut-Namen das Wort `ENUM` und anschließend eine durch Kommata getrennte, in Klammern eingeschlossene Liste aller Attribute der Enumeration-Klasse.

Aufwändiger sind Referenzen auf andere Klassen. Diese Referenzen verweisen derzeit noch auf Klassen, etwa verweist `Item_version` auf `Item`. Für die Datenbank muss jedoch auf konkrete Attribute verwiesen werden, etwa auf `id` in `Item`. Aus diesem Grund müssen hier die primären Attribute der referenzierten Klasse ermittelt und gekürzt werden. Für jedes gefundene primäre Attribut muss ein Attribut in der SQL-Tabelle erzeugt werden.

Nachdem alle Attribute in das SQL-Skript geschrieben sind, müssen noch Foreign Keys und Primary Keys angegeben werden. Die Foreign Keys können dabei leicht mitprotokolliert werden, während die Referenzen wie oben genannt behandelt werden. Ähnliches gilt für die Primary Keys. Bei diesem ist für jedes Attribut zusätzlich zu prüfen, ob es das Stereotyp «`PrimaryKey`» enthält. Ist dies so, wird auch dieses Attribut in einer Liste vermerkt.

Am Ende werden dann alle mitprotokollierten Foreign Keys ausgegeben, gefolgt von der Liste mit den Primary Keys. Nachfolgend ist die generierte Tabelle für `Item_version` dargestellt.

```
-- An Item_version is a version of an Item and serves as the collector of
-- the data characterizing a physically realizable object in various
-- application contexts.
-- NOTE An Item_version may be produced, consumed, used to produce other
-- Item_version objects, or offered to the market.
-- NOTE The collection of defining information may be incomplete, i.e., not
-- all of the Design_discipline_item_definition objects needed to define an
-- Item_version are associated with the Item_version.
-- NOTE The set of Item_version objects of an Item represents the history
-- of the Item within a particular life cycle stage or over its complete
-- life cycle.
-- NOTE An Item_version does not have to be referenced by any
-- Design_discipline_item_definition.
-- NOTE In the case, where two Item_version objects are related to each
-- other by an Item_version_relationship with a relation_type of value
-- 'supplied part', the relating Item_version is the version carrying the
-- necessary information, e.g., shape and properties, and the related is
-- the version either supplied by or supplied for another company.
--
CREATE TABLE item_version (
  -- The id specifies the identifier of the Item_version. The id shall be
  -- unique within the scope of the associated Item.
  -- NOTE A particular object is identified by the id of Item_version that
  -- serves to identify the particular version coupled with the id of the
  -- associated Item that serves to identify the generic object.
  id VARCHAR(3) NOT NULL,
```

```
  -- The associated_item specifies the Item with which the Item_version is
  -- associated.
  associated_item__p3355 VARCHAR(13) NOT NULL REFERENCES item( id ),
  FOREIGN KEY ( associated_item__p3355 ) REFERENCES item(id),
  PRIMARY KEY ( id, associated_item__p3355 )
);
```

7.4 Zusammenfassung

Dieses Kapitel beschrieb die benötigten Schritte, die für eine erfolgreiche Transformation in ein SQL-Datenbank-Schema notwendig sind. Dazu müssen alle Klassen identifiziert werden, die persistent sein sollen. Auch müssen die PIM-Datentypen in SQL-konforme Datentypen umgewandelt werden. Zwei große Probleme sind dabei zum einen die Vererbung, die aufzulösen ist und die Attribut-Namen, die zu kürzen sind. Zudem ist es möglich, das Datenschema für nachfolgende Transformationen zu vereinfachen. Den eigentlichen SQL-Code zu generieren ist vergleichsweise einfach. Etwas aufwändiger ist, die Referenzen zu anderen Entitäten in die SQL-Datei auszugeben.

Kapitel 8

Transformationen der Service-Schicht

Die Aufgabe der Service-Schicht ist es, Anfragen der Clients entgegen zu nehmen, sie zu bearbeiten und zu beantworten. Für den Prototypen bedeutet jede Anfrage zumeist einen Zugriff auf die Daten-Schicht. Das Prinzip soll dabei dem der Enterprise JavaBeans (EJB) entsprechen, auch wenn die hier vorgestellte Implementierung keineswegs an EJB herankommt. Für den produktiven Einsatz könnten die Transformationen jedoch entsprechend mächtiger werden.

8.1 Aufgaben und Anforderungen

Das Kapitel 5 auf Seite 29 zählte bereits wichtige Bedingungen auf, unter denen die Beispiel-Implementierung zu entwickeln ist. So sind alle drei Schichten der Software strikt voneinander zu trennen. Es soll sogar erlaubt sein, dass einzelne Schichten konventionell entwickelt werden, etwa um der Client-Schicht eine benutzerfreundliche Oberfläche zu bieten. Auch sollen verschiedene Implementierungen erlaubt sein, etwa um ein Legacy-System anstatt einer Datenbank anzubinden oder um sowohl Java/EJBs als auch .Net oder CORBA zu verwenden. Dieser Abschnitt soll Anforderungen an die zu generierende Software definieren und diese mit den genannten Voraussetzungen abgleichen.

Die erste Anforderung an die zu generierende Software ist, dass sie verteilt lauffähig sein soll. Sowohl Datenbank, Service-Schicht als auch die Clients dürfen auf verschiedenen Rechnern liegen. Dies hat zur Folge, dass zwischen den einzelnen Schichten Kommunikations-Protokolle einzusetzen sind. Für die Daten-Schicht bietet sich zumeist der JDBC bzw. ODBC-Standard an, sofern mit einem Datenbankmanagement-System kommuniziert wird. Dem Client stehen verschiedene Protokolle zur Verfügung, mit denen er mit der Service-Schicht kommunizieren kann, etwa per CORBA, Web Services oder RMI.

Diese Anforderung stellt die Software-Generierung vor zwei Probleme: Zum einen soll es erlaubt sein, die Client-Schicht auf konventionelle Weise zu entwickeln und zum anderen dürfen verschiedene Kommunikations-Protokolle eingesetzt werden. Aus diesem Grund soll der Prototyp der Client-Schicht eine Schnittstelle bieten, die unabhängig vom eingesetzen Protokoll ist.

Die zweite Anforderung richtet sich an den Client. Für die prototypische Implementierung ist ein lauffähiger Client zu generieren. Dieser soll per Web-Browser ansprechbar sein (Web Interface). Dabei soll es nachträglich möglich sein, das Layout der Web-Seiten anzupassen, ohne dazu die Transformationen anpassen zu müssen. Außerdem soll ein Layout-Experte ohne Programmier-Kenntnisse das Aussehen der Seiten gestalten können.

Die dritte und wichtigste Anforderung ist, dass ein Großteil der Transformationen wiederverwendbar sein soll — unabhängig von den tatschlichen Protokollen oder Implementierungen. Beispielsweise besagt die erste Anforderung, dass die Anwendung verteilt lauffähig sein soll. Damit muss das Modell um entsprechende Verteilungsaspekte erweitert werden, etwa um eine Service- und eine Client-Klasse. Dies geschieht unabhängig vom verwendeten Kommunikations-Protokoll und soll somit auch nur in einer Transformation enthalten sein. Ziel ist es, verschiedene Transformationen anzubieten, die der Software-Designer nach Bedarf verwenden kann.

8.2 PIM-nach-PIM – Transformationen

Der Begriff PIM-nach-PIM – Transformation wurde bisher zwar noch nicht definiert, doch wurde bereits erwähnt, dass es mehr als eine Transformation zwischen dem PIM und dem PSM gegeben kann und sogar sollte. Eine solche Transformation erzeugt aus einem PIM wiederum ein PIM, was den gewählten Begriff erklärt (siehe Absatz 2.1.1 auf Seite 16). Für den Prototypen sind diese die Transformationen, um das Klassengerüst und die Verteilung zu erzeugen.

8.2.1 Festlegen des Klassengerüsts

Das Kapitel 7 auf Seite 36 zeigte bereits, wie aus dem UML-Modell ein SQL-Schema erzeugt wurde. Eine Aufgabe der Service-Schicht ist es nun, genau diese Daten zu verwalten. So sind die Daten aus der Daten-Schicht zu lesen, in das korrekte Format zu bringen und an den Client weiterzureichen. Ändert der Anwender über den Client die Daten, so müssen diese überprüft und an die Daten-Schicht übermittelt werden.

Es existieren hierfür bereits standardisierte Verfahren, welche diese Funktionalität bereitstellen. Ein Vertreter ist beispielsweise die *Enterprise JavaBeans* (EJB) Spezifikation. Diese nehmen die Datenverwaltung auf Wunsch vollständig dem Entwickler ab und stellen die Persistenz der Daten in den Enterprise Beans sicher. Für diesen Prototypen wurde jedoch gezielt auf EJBs verzichtet. Zum einen stellen EJBs immer noch das Standard-Beispiel für MDA dar, so dass hier auf andere Quellen verwiesen sei[1]. Zum anderen soll gezeigt werden, dass sich die Vorteile der EJBs auch für andere Plattformen bereitstellen und wiederverwenden lassen. Dennoch orientiert sich diese Beispiel-Implementierung sehr stark an dem EJB-Standard, um bekannte Konzepte zu demonstrieren.

Für diese Datenverwaltung sind unter anderem Klassen erforderlich, welche die Daten in sich tragen und verwalten. Dies sind die im PIM modellierten Klassen, wie etwa `Item` oder `Item_instance`. Jedes Attribut, welches als nicht-privat definiert ist, erhält Zugriffsmethoden je nach Sichtbarkeit des

[1] Vgl. etwa AndroMDA (`http://www.andromda.org/`), [KWB03, S. 43ff].

Attributs. Dies ist sicher keine große Kunst, da die meisten CASE-Tool diese Funktionalität bereitstellen. Jedoch kommt hier eine Besonderheit hinzu: Attribute, welche mehrere Werte annehmen können, erhalten keine Getter- und Setter-Methoden. Diese Attribute werden zu Containern und benötigen deshalb Methoden, um Werte hinzuzufügen und zu löschen. Aus diesem Grund erhalten diese Attribute die Accessoren `add` und `remove`, sowie `get` um den gesamten Container abzurufen.

Damit sind die Daten in einem Objekt verfügbar. Offen bleibt, wie die Daten in das Objekt geladen werden und wie Änderungen an den Werten persistent gemacht werden. Hier erhält jede Entity-Klasse eine zusätzliche Management-Klasse. Sie bietet alle Funktionen, um das Daten-Objekt mit der Daten-Schicht zu laden und zu synchronisieren.

Der Prototyp bietet hierzu die minimal notwendigen Methoden an, um einen Datensatz zu finden (`findByPrimaryKey`), einen neuen Satz zu erzeugen (`create`) und eine Liste aller Objekte zu erhalten (`list`). Eine weitere denkbare Methode wäre `remove`, um Datensätze zu löschen, sowie Methoden, die nach weiteren Attributen suchen. So könnten zusätzliche Such-Methoden für solche Attribute angelegt werden, die mit dem Stereotyp «`Searchable`» markiert sind, etwa `findByName` (siehe Abschnitt 6.2 auf Seite 32).

Möchte ein Client auf ein bestimmtes Objekt zugreifen, so muss dieses die Management-Klasse kontaktieren und die Methode `findByPrimaryKey` aufrufen. Als Rückgabewert erhält der Aufrufer dann das konkrete Daten-Objekt, etwa das `Item` mit der ID "`M100`". Da die konkrete Implementierung der Management-Klasse von der Daten-Schicht abhängig ist, wird hier nur ein Interface definiert. Die Implementierung folgt dann über eine Brücke zur Daten-Schicht, welche genau weiß, wie auf die Daten zugegriffen werden muss (siehe 8.6 auf Seite 60).

Da der Zugriff auf die Daten-Schicht ausschließlich über die Management-Klasse geschieht, kann diese sicherstellen, dass pro physikalischem Datensatz nur ein Service-Objekt existiert. Sollten mehrere Clients gleichzeitig den selben Datensatz lesen wollen, so erhalten beide auch das selbe Daten-Objekt. Somit sind Änderungen eines Clients sofort in den anderen Clients sichtbar.

Soll dies nicht gewünscht sein, so ließe sich ein Transaktions-Verfahren einführen. Beispielsweise würde der Beginn einer Transaktion bedeuten, dass ein neues, eigenes Objekt erzeugt wird. Alle Änderungen an diesem bleiben den anderen verborgen. Erst beim Commit wird das eigentliche Daten-Objekt gesperrt und aktualisiert.

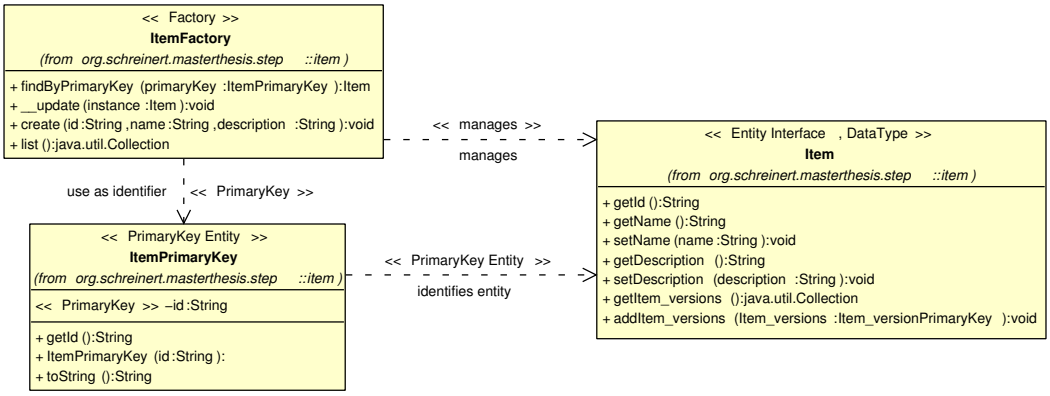
Bleibt noch zu definieren, wie Änderungen an den Objekten zur Daten-Schicht zu propagieren sind. Im Prototypen wurde eine recht simple Lösung eingesetzt. Jede Methode im Daten-Objekt, welche die Daten manipulieren kann, informiert sein Management-Objekt darüber, dass es Änderungen gibt. Das Management-Objekt kann dann entscheiden, ob die Änderung entweder sofort zur Daten-Schicht gemeldet wird oder erst zu einem späteren Zeitpunkt. Das sofortige Aktualisieren hat den Nachteil, dass hierdurch eventuell (kurzfristig) Inkonsistenzen in der Datenbank entstehen. Datenbank-Constraints könnten hier greifen und diese Änderung verhindern, was nicht im Sinne des Programmes wäre.

Es gibt noch weitere Möglichkeiten, wie Änderungen zur Daten-Schicht gemeldet werden können. Beispielsweise würde jedes Objekt intern vermerken, wenn es manipuliert wurde. Das Management-Objekt könnte z.B. in bestimmten Intervallen die Objekte nach Änderungen abfragen bzw. beim Vernichten des Objektes den Status prüfen. Die ständige Abfrage der Objekte ist jedoch besonders bei hoher Last ein zusätzlicher Rechenaufwand. Wird hingegen gewartet, bis das Objekt vernichtet wird,

besteht die Gefahr, Daten durch einen Absturz zu verlieren. Aus diesem Grund empfiehlt sich für eine produktive Umsetzung ein Transaktions-Verfahren. Solange kein Commit erfolgt, sind alle Manipulationen am Daten-Objekt privat für den jeweiligen Client. Mit dem Commit muss zwangsweise die Daten-Schicht aktualisiert werden, um im Falle eines Absturzes die Daten-Schicht konsistent zu halten.

Weiterhin wird eine Primary Key - Klasse benötigt. Sie trägt die Information, welche Attribute primär sind. Für die Klasse `Item` enthielte `ItemPrimaryKey` eine Methode `getId():String`, sowie einen Konstruktor mit `id:String` als Parameter. `Item_version` enthielte zusätzlich `getItem():Item` und den zusätzlichen Parameter `item:Item` im Konstruktor.

Somit existieren pro Entity-Klasse drei Klassen: Die ursprüngliche *Entity*-Klasse, die *Management*-Klasse und die *Primary Key*-Klasse. Für das Beispiel von `Item` ergibt sich das Datenmodell, wie in Abbildung 8.1 gezeigt. Dabei ist `ItemFactory` die Management-Klasse und `ItemPrimaryKey` die Primary Key Klasse. Die Namen der anderen Klassen folgen der selben Konvention.



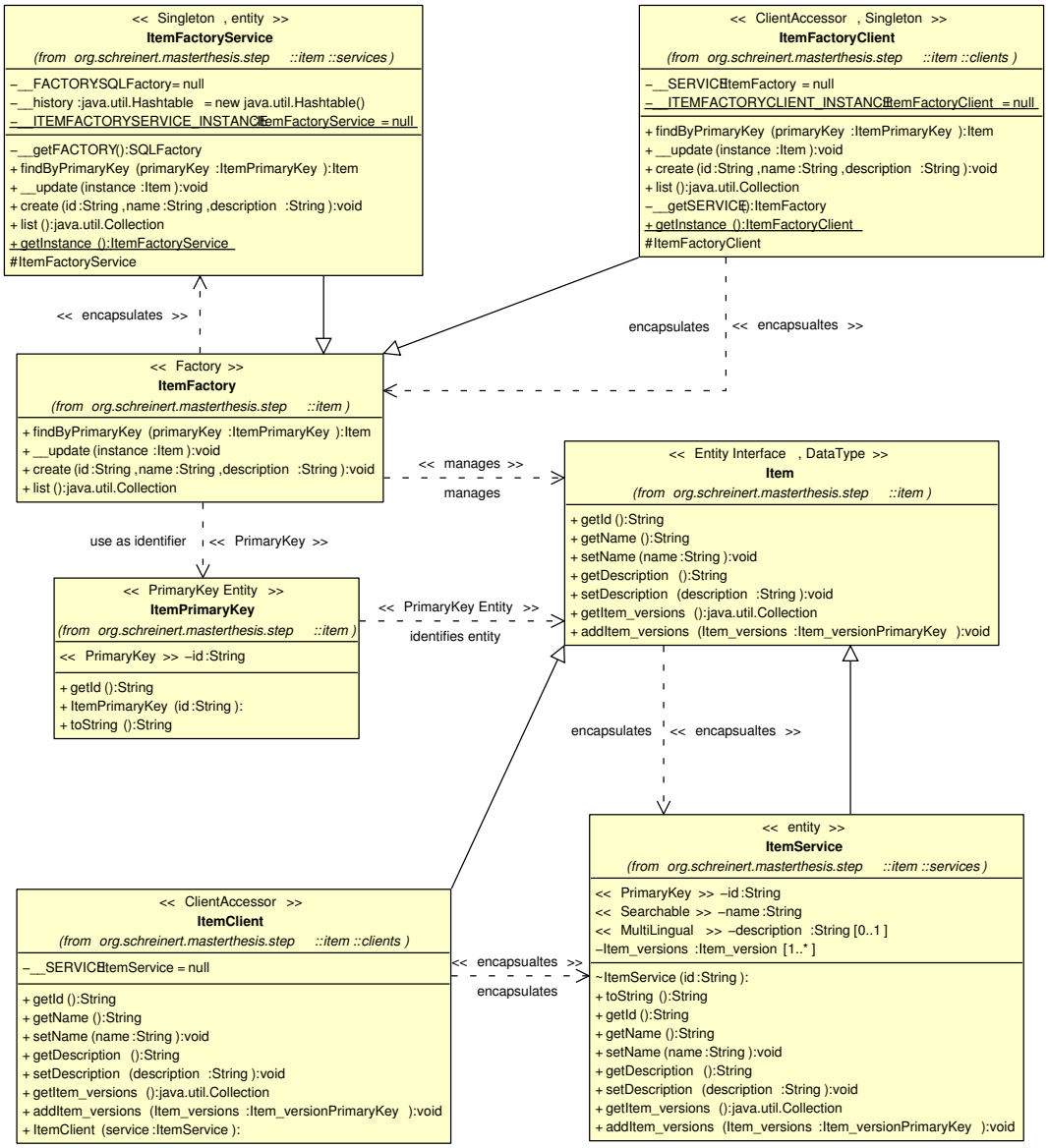
Quelle: Eigene Grafik

Abbildung 8.1: Klassengerüst für `Item`

8.2.2 Erzeugen der Verteiltheit

Eine große Herausforderung ist es, den Client mit der Service-Schicht zu verbinden und dabei die erste und zweite Anforderung bezüglich der Verteiltheit zu erfüllen (siehe Seite48). Dazu soll dem Client das verwendete Kommunikations-Protokoll verborgen bleiben. Aus diesem Grund wird jede Entity-Klasse und jede Management-Klasse aufgeteilt in eine *Service*- und eine *Client*-Klasse. Lediglich die Primary Key-Klasse bleibt unverändert, da auf diese Objekte nicht remote zugegriffen werden muss. Letztendlich ergibt es die in Abbildung 8.2 auf der nächsten Seite dargestellte Struktur für die Klasse `Item`.

Hierbei müssen größere Veränderungen am Modell vorgenommen werden, was am Beispiel der Klasse `Item` verdeutlicht werden soll. Zuerst wird die Klasse `Item` in `ItemService` umbenannt. Würde die Client-Schicht direkt auf dieses Service-Objekt zugreifen wollen, so müsste sie das konkrete Kommunikations-Protokoll kennen. Da dies verhindert werden soll, gibt es eine weitere Klasse



Quelle: Eigene Grafik

Abbildung 8.2: Erweiterung von `Item` für die Verteilung

namens `ItemClient`, welche als lokales Objekt im Client läuft. In dieser sind alle Signaturen der nicht-privaten Methoden von `ItemService` nachgebildet. Die Methoden werden in einer späteren Transformation so mit Code ergänzt, dass die eigentliche Methode am `ItemService`-Objekt aufgerufen wird — und dies via dem festgesetzten Kommunikations-Protokoll. Somit enthält die Klasse `ItemClient` ausschließlich den Code, um mit dem `ItemService`-Objekt zu kommunizieren. Dieses Objekt wird dem Client-Entwickler gestellt, so dass er immer mit diesem arbeitet — unabhängig von seiner Implementation.

Dieses Vorgehen macht jedoch ein Interface notwendig. Die Methode `getAssociated_item()` in

`Item_version` liefert ein Objekt vom Typ `ItemService` zurück. Richtig wäre jedoch ein Objekt vom Typ `ItemClient`. Aus diesem Grund wird ein Interface eingeführt, welches (wie beim `ItemClient`) alle nicht-privaten Methoden der `ItemService`-Klasse enthält. Die Klassen `ItemService` und `ItemClient` implementieren dieses Interface. Zudem werden alle betroffenen Methoden so modifiziert, dass sie ab sofort das Interface `Item` als Rückgabewert haben.

Der Client erhält noch eine weitere Methode. Diese unterscheidet sich nach einem Client für die Entity-Klasse (`ItemClient`) und einem Client für die Management-Klasse (`ItemFactory`). Letztere erhält eine zusätzliche private Methode namens `__getService()`, welche die Adresse des Service-Objekts ermittelt. Dies geschieht beispielsweise per *CORBA-NamingService*, *Java Naming and Directory Interface* (JNDI) oder *Universal Description Discovery and Integration* (UDDI) in Verbindung mit der *Web Services Description Language* (WSDL). Die Entity-Klasse hingegen erhält eine Referenz auf das Service-Objekt, über welches die eigentliche Service-Methode aufgerufen wird.

Beispiel-Programm 6 Kommunikation mit der Service-Schicht (Beispiel)

Das nachfolgende Beispiel zeigt, wie einfach der Client auf die Service-Schicht zugreifen kann. Dieser Code ist — in einer Methode implementiert — wie abgebildet lauffähig.

Zuerst wird das Management-Objekt für `Item` benötigt.

```
ItemFactoryClient itemFactory = ItemFactoryClient.getInstance();
```

Anschließend wird ein neues `Item` erzeugt.

```
itemFactory.create("Q138", "Steering Wheel", "Steering Wheel for Product P23");
```

Das Objekt wird abgefragt via:

```
ItemPrimaryKey pk = new ItemPrimaryKey("Q138");
Item item = itemFactory.findByPrimaryKey(pk);
```

Eine Liste mit `ItemPrimaryKey`s aller vorhandenen `Item`s liefert dieser Aufruf:

```
Collection items = itemFactory.list();
```

Der Beispiel-Code 6 soll zeigen, wie einfach es ist, auf Objekte der Service-Schicht zuzugreifen. Sollte die Client-Schicht doch auf konventionelle Art entwickelt werden, so muss sich der Entwickler nicht um das Protokoll oder die Implementierung der Services kümmern. Seine Schnittstelle zur Service-Schicht bleibt stabil, er verwendet immer das Client-Objekt mit dem dazugehörigen Interface.

8.3 Verknüpfung von Modell-Elementen

Die zwei zuvor beschriebenen Transformationen haben das Modell grundlegend verändert, indem sie neue Klassen eingeführt haben und bestehende Methoden-Signaturen geändert haben (Rückgabewert). Nachfolgende Transformationen benötigen aber beispielsweise die Information, zu welcher

Service-Klasse eine Client-Klasse gehört. Daneben gibt es noch weitere Informationen, die im Modell vorhanden sein müssen. Dieser Abschnitt zeigt Möglichkeiten auf, wie diese Informationen abgebildet werden können.

8.3.1 Erzeugung der Zugriffsmethoden

Für den Fall der erzeugten Zugriffsmethoden bietet die JMI-Spezifikation mehrere Varianten an. So lassen sich diese Zugriffsmethoden mit einem Stereotypen kennzeichnen, etwa «`Accessor`». Es fehlt aber die Information, auf welches Attribut sich die jeweilige Methode bezieht. Ein im Prototypen häufig und erfolgreich verwendetes Verfahren, solche Beziehungen zu verwalten, sind Abhängigkeiten (Dependencies). Hierzu wird eine Dependency von den Zugriffsmethoden zum betroffenen Attribut aufgebaut. Zusätzlich sollte die Dependency mit einem Stereotyp markiert werden, z.B. «`Accessor`», damit sie von anderen Dependencies unterscheidbar bleibt.

Eine weitere Möglichkeit wäre eine Informations-Fluss-Referenz (`Flow`). Es ist möglich, den Fluss eines Wertes in ein Attribut zu modellieren. Für die Getter-Methode wäre folglich ein Informations-Fluss anzulegen, der das Klassen-Attribut in den Rückgabe-Parameter der Getter-Methode fließen lässt. Ähnlich beim Setter. Hier würde der Parameter in das Attribut fließen.

Beide Varianten müssen jedoch zum Zeitpunkt der Implementierung (PSM-nach-Code) ausgewertet werden, so dass hierdurch keine von beiden einen großen Vorteil besitzt. Eine dritte Möglichkeit hingegen kann später von externen Tools ausgewertet werden. Hierzu erhalten die Zugriffsmethoden OCL-Markierungen. Beispielsweise soll bei der Getter-Methode der Rückgabe-Wert das entsprechende Attribut sein. Hierfür ist etwa nachfolgende Bedingung anzugeben:

```
context Item:getId():String
  body : self.concreteAttribute
```

Letztendlich könnte die Transformation, welche die Zugriffsmethoden erzeugt, auch gleich Code für die gängigsten Programmiersprachen angegeben. Die PSM-nach-Java – Transformation würde dann den gespeicherten Programm-Code für Java ausgeben. Dieser Ansatz hat jedoch den Nachteil, dass neue oder weniger populäre Sprachen nicht unterstützt wären. Außerdem wäre es schwerer, den Methoden zusätzlichen Code unterzuschieben, etwa für Debugging- oder Zugriffsschutz-Zwecke.

8.3.2 Verbinden der Klassen

Im Abschnitt 8.2.2 auf Seite 51 über die Verteilung wurden zusätzliche Klassen und Interfaces generiert. Diese Modell-Elemente stehen dabei immer noch in Verbindung miteinander. Während späteren Transformationen ist es notwendig, die passende Service-Klasse für eine gegebene Client-Klasse zu finden. Zwar könnte dies über die Namenskonvention entdeckt werden, doch ist dies eine zu unsichere Methodik. Auch hierfür eignen sich die bereits angesprochenen Dependencies. Im Beispiel würde dies bedeuten, dass es eine Dependency zwischen `Item` und `ItemInstance` gibt. Auch hier empfiehlt sich, der Dependency einen eindeutigen Stereotypen zuzuweisen, da mit einer Klasse verschiedene Abhängigkeiten verbunden sein können.

8.3.3 Rückverfolgung vom PSM zum PIM

Es ist auch für den Software-Entwickler hilfreich, jede Klasse im PSM auf ein konkretes Objekt im PIM zurückverfolgen zu können. Dies erleichtert es dem Entwickler zu verstehen, warum eine Klasse auf eine bestimmte Art verändert wurde. So kann etwa die Methode `getInstance()` der Management-Klasse darauf zurückgeführt werden, dass diese Klasse *Singleton* sein soll.

Auch lässt sich so leichter effizienterer Code erstellen. Der Entwickler wäre in der Lage, die Transformationen über die Transformations-Parameter optimal anzupassen. Er könnte so erkennen, welchen Parameter er für ein gewünschtes Ergebnis anpassen müsste.

Angedacht ist auch, dass der Entwickler einen umgekehrten Weg gehen kann. So möchte er etwa erkennen, welche Auswirkungen eine Änderung im Programm-Code auf das PIM hätte. Auf diese Weise könnte er den erzeugten Code so verändern, dass dieser performanter läuft. Dabei sollen jedoch die Anforderungen aus dem PIM unverändert bleiben.

8.4 PIM-nach-PSM – Transformation

Im nächsten Schritt muss eine konkrete Implementierungs-Plattform gewählt werden. Dies betrifft besonders die Frage, welches Kommunikations-Protokoll zu verwenden ist. So muss an dieser Stelle festgelegt werden, wie der Client das Service-Objekt anspricht.

8.4.1 Anpassen der Client-Klassen

Für die Implementierung via *Remote Method Invocation* (RMI) müssen die bestehenden Klassen angepasst werden. Der RMI-Client benötigt zuerst das eigentliche, entfernt liegende Service-Objekt. Dazu wurde die Client-Klasse im Abschnitt 8.2.2 auf Seite 51 bereits um ein privates Attribut erweitert. Dieses Attribut muss spätestens dann initiiert werden, wenn über den Client eine Service-Methode aufgerufen werden soll.

Die Methode `getId():String` der Klasse `Item` würde beispielsweise in Java wie folgt implementiert sein:

```
public String getId() { // vereinfacht (ohne Exception-Handling)
    return this.__service.getId();
}
```

Nun muss das Attribut `__service` noch initiiert werden. Da der Prototyp nicht darauf getrimmt ist, performat und Ressourcen-sparend zu sein, wird das Service-Objekt zu beginn referenziert und zwischengespeichert. Dies geschieht so lange, bis der Garbage-Collector das Client-Objekt eliminiert. Für RMI sieht die Methode etwa wie folgt aus:

```
private ItemService __getService() throws java.rmi.RemoteException {
   if ( this.__service == null ) {
      ItemService service = (ItemService)java.rmi.Naming.
                            lookup("//host/ItemService");
      this.service = service;
   }
   return this.__service;
}
```

Die in diesem Abschnitt beschriebene Transformation ist für die tatsächliche Implementierung der Methode eigentlich nicht zuständig. Die Client-Klassen müssen nur als solche gekennzeichnet und um die Methode `__getService():ItemService` erweitert werden. Das private Attribut `__service: ItemService` und alle Methoden-Signaturen sind ebenfalls um die entsprechende Exception zu erweitern. Das RMI-Beispiel ist fest auf eine Java-Implementierung fixiert, so dass die Methoden gleich mit dem Java-Code belegt sein können. Alternativ müsste dies von der PSM-nach-Java – Transformation nachgezogen werden.

8.4.2 Einbauen des RemoteException-Handlings

Im Beispiel fehlt noch ein Exception-Handling, da `java.rmi.RemoteException` geworfen werden kann. Hierzu könnte diese konkrete Exception in die Signatur der Methode einfließen, etwa

```
public String getId() throws java.rmi.RemoteException {
  // ..
}
```

Allerdings müsste somit an einer schwer festzusetzenden Stelle „im Hauptprogramm" diese Exception abgefangen werden. Alternativ könnte diese Exception auch ignoriert werden:

```
public String getId() {
  try {
    // ..
  } catch( java.rmi.RemoteException excp ) {
  }
}
```

Ebenfalls möglich wäre die Exception `java.lang.Exception` zu werfen. Damit würde sich das Error-Handling jedoch auf ein reines Abfangen von Fehlern beschränken. Deshalb empfiehlt es sich, eigene Exception-Klassen einzuführen und diese der Client-Schicht sichtbar zu machen. Es sei noch angemerkt, dass dies eigentlich in der Transformation erfolgen müsste, welche die Verteilung und vor allem das Interface erzeugt (siehe 8.2.2 auf Seite 51). Der Übersichtlichkeit wegen folgt die Einführung erst an dieser Stelle.

Der Client hat somit die Möglichkeit, eine RemoteException auszuwerten, ohne hier auf die spezielle RMI-Exception einzugehen. Stattdessen wird eine Exception für alle Kommunikations-Protokolle verwendet. Die Methode sähe damit wie folgt aus:

```
public String getId() throws mytool.RemoteException {
    try {
       return this.__service.getId();
    } catch ( java.rmi.RemoteException excp ) {
       throw new mytool.RemoteException( excp );
    }
}
```

Die hier beschriebene Exception `mytool.RemoteException` wird dabei von dieser Transformation erzeugt. Der Package-Name `mytool` ist dabei als Platzhalter für das tatsächliche Package zu verstehen.

Selbstverständlich ist es auch erlaubt, dass ein Service-Objekt eine weitere, gegebenenfalls eigene Exception werfen kann, beispielsweise `mytool.ConfigException` oder `java.text.ParseException`. Dies ist durch das PIM bereits in der Service-Klasse vorhanden und wurde beim Anlegen des Clients und des Interfaces in beiden ebenfalls mit angelegt. Je nach Protokoll müsste diese Exception unter Umständen noch angepasst werden.

8.4.3 Anpassen der Service-Klassen

Im Vergleich zu den Client-Klassen sind an den Service-Klassen weitergehende Änderungen durchzuführen. Um das Beispiel mit der RMI-Implementierung fortzuführen, muss das Service-Objekt unter anderem von einer RMI-Klasse erben. Im allgemeinen genügt es, wenn die Service-Klasse die Klasse `java.rmi.server.UnicastRemoteObject` erweitert.[2] Da unter Java nur einfache Vererbungen unter Klassen möglich sind, muss jene Klasse die `UnicastRemoteObject`-Klasse erweitern, die `java.lang.Object` erweitert.

Es ist zudem für RMI-Implementierungen üblich, ein Interface für das Service-Objekt bereitzustellen. Da dieses Interface — aus anderen Gründen — bereits durch eine frühere Transformation erstellt wurde, muss es noch die Generalisierung zu `java.rmi.Remote` erhalten. Im obigen Beispiel ist dies das Interface `Item`, welches von `ItemService` und `ItemClient` realisiert wird.

```
public interface Item extends java.rmi.Remote {
   // ...
}
```

Neben den Client-Klassen müssen auch die Methoden-Signaturen sowohl des Interfaces als auch der Service-Klasse angepasst werden. Dies geschieht genauso, wie bei der Client-Klasse. Damit kann jede Methode ab sofort RemoteExceptions werfen. Bleibt noch sicherzustellen, dass die Klasse einen Konstruktor besitzt, der ebenfalls Remote-Exceptions werfen kann. Dies ist für RMI erforderlich.

```
public class ItemService extends java.rmi.server.UnicastRemoteObject {
   public ItemService() throws java.rmi.RemoteException {
   }
```

[2] Vgl. [HR98, S. 582].

```
    public String getId() throws java.rmi.RemoteException {
       return this.id();
    }
    // ...
}
```

8.4.4 Einführen eines Manager-Objekts

Letztendlich bleibt noch das Problem, wie der Client sein Service-Objekt finden kann. Der Abschnitt 8.4.1 auf der Seite 55 ist als Beispiel aufgeführt, dass dies etwa per `JavaNaming` – Schnittstelle geschieht. Jedoch muss der Dienst einmalig registriert werden. Hierzu steht die Methode `Naming.bind (String name,Remote obj)` beziehungsweise `Naming.rebind(String name,Remote obj)` aus dem Package `java.rmi` bereit, um eine mögliche vorherige Bindung zu überschreiben.

Da ein Service-Objekt erst dann initiiert wird, wenn es per `Naming.lookup(..)` gesucht wurde, kann diese Bindung nicht im Service-Objekt geschehen. Dazu empfiehlt sich ein weiteres Management-Objekt, welches alle Service-Objekte initiiert und an einen Namen bindet. Dieses Management-Objekt existiert nur einmal für alle Klassen und muss während des Programm-Starts einmalig aufgerufen werden.

Dazu muss dem Management-Objekt eine Liste mit allen zu initiierenden Service-Klassen bereitstehen. Diese Liste ließe sich zwar leicht per Transformationsregel in den Code des Management-Objekts eintragen, jedoch wäre diese Liste damit fest verdrahtet. Besser wäre es, wenn die Liste per Konfiguration geladen wird. Damit kann der Administrator nicht benötigte Service-Klassen abschalten oder gar eigene Implementierungen bereitstellen und angeben.

Damit besteht das Problem, dass dem Management-Objekt auch bekannt sein muss, an welchen Namen es ein Service-Objekt binden muss. Zwar könnte auch dies in eine Liste eingetragen werden, doch müssten dann auch die Client-Objekte auf diese Namens-Liste zugreifen. Deshalb empfiehlt es sich, die Namenswahl mit dem Service-Objekt beziehungsweise dem Service-Interface zu verknüpfen. Jedes Interface, etwa `Item`, erhält damit eine Konstante mit Namen `RMINAME`. Als Wert ist dann der Service-Name enthalten, an den das Management-Objekt das entsprechende Remote-Objekt binden soll. Sowohl Client, als auch Management-Objekt erfahren den Namen über die genannte Konstante.

8.4.5 Auflösen von Entwurfs-Mustern

Während der Entwickler sein Programm modelliert, verwendet er dabei häufiger bestimmte, festdefinierte Muster. Beispielsweise arbeitet er mit Singletons, Proxy-Objekte oder mit Enumerations. Dazu sei es dem Entwickler gestattet, solche Entwurfs-Muster auch als solche zu kennzeichnen. Beispielsweise markiert er eine Klasse als Singleton, indem er dieser Klasse den Stereotyp «`Singleton`» gibt. Gleiches gilt etwa für die Enumerations.

Um Klassen die Singleton-Eigenschaft zu verleihen, braucht es weiterer Modell-Elemente. Um zu verhindern, dass mehr als eine Instanz der Klasse existiert, müssen zuerst alle Konstruktoren mindestens auf protected, besser auf private gesetzt werden. Sollte kein Konstruktor vorhanden sein und damit der Default-Konstruktor greifen, muss einer erzeugt werden. Damit kann eine Instanz nur noch aus der Klasse heraus erzeugt werden.

Genauer gesagt, wird die einzige Instanz von der statischen Methode `getInstance()` erzeugt. Diese erzeugt beim ersten Aufruf eine Instanz und gibt diese dem Aufrufer zurück. Jedoch notiert sich die Methode die Referenz auf die Instanz und gibt bei zukünftigen Aufrufen diese gespeicherte Referenz zurück. Für diese Referenz ist noch ein Attribut zu erzeugen, welches nur privat sichtbar ist und das aus einer statischen Methode heraus zugreifbar ist.[3]

Die Behandlung von Enumerations wurde aus Gründen der Programm-Vereinfachung bereits in Abschnitt 6.4 auf Seite 34 beschrieben. An dieser Stelle sei nur erwähnt, dass auch diese Enumeration als Muster gesehen werden können.

8.5 PSM-nach-Java – Transformation

Die letzte Transformation der Service-Schicht wandelt nun alle Klassen in Java-Code um. Hierzu bekommt jedes Package einen eigenen Ordner im Dateisystem und jede Klasse eine eigene Datei im jeweiligen Ordner. Für jede Klasse gibt die Transformation dazu zuerst die Dokumentation aus, wie sie im PIM vorhanden ist oder während der Transformationen hinzugefügt wurde. Für Java-Code erfolgt anschließend die Ausgabe des Package-Namens:

```
/**
 *  An Item is either a single object or a unit in a group of objects. It
 * collects the information that is common to all versions of the object.
 * An Item shall always be classified as 'part', 'tool', or 'raw material'
 * [...]
 */
package org.schreinert.masterthesis.step.item.services;
```

Normalerweise würden nun im Programm-Code alle verwendeten Klassen per `import`-Angabe eingebunden werden. Der Prototyp verwendet jedoch keine Imports, da die benutzen Klassen nur mit erhöhtem Aufwand ermittelt werden können. Zudem besteht die Gefahr, dass in unterschiedlichen Packages Klassen mit gleichen Namen auftreten. Deshalb wird pro Referenz geprüft, ob das jeweilige Objekt im gleichen Package wie die Klasse liegt. Ist dies der Fall, wird nur der Objekt-Name ausgegeben, ansonsten ist immer der volle Pfad zum Objekt angegeben.

Die Klassensignatur lässt sich relativ leicht ermitteln. Es muss zuerst je nach Sichtbarkeit z.B. `public` ausgegeben werden, gefolgt von `class` oder `interface` und dem Klassen-Namen. Anschließend sind die Erweiterungen und Implementationen anzugeben.

[3] Vgl. [GHJV01, S. 157ff].

```
public class ItemService implements org.schreinert.masterthesis.step.item.Item
                        extends    java.rmi.server.UnicastRemoteObject  {
```

Nun werden alle Attribute angegeben. Auch ist wieder die Sichtbarkeit zu beachten und es muss gegebenenfalls ein Default-Wert angegeben werden. Zudem muss geprüft werden, ob das Attribut final oder static ist.

```
/**
  * The id specifies the identifier of the Item. For the id, an owner shall
* be specified by a Person_organization_assignment with role 'id owner'.
* [...]
*/
private String id;
```

Auch die Methoden-Signatur lässt sich ähnlich ermitteln. Ein Sonderfall ist der Konstruktor, der keinen Rückgabe-Wert definiert. Eine Liste mit Methoden-Parametern mit Namen und Typen steht ebenso bereit, wie der Typ des Rückgabewertes. Auch Exceptions lassen sich aus dem Modell ermitteln.

```
/**
 * Print content of this Object of type ItemService
 * @seeItemService
 * @return content of object as String
 */
 public String toString( ) {
```

Die große Herausforderung liegt im eigentlichen Programm-Code. Während Getter- und Setter-Methoden noch relativ einfach mit Code gefüllt werden können (siehe Abschnitt 8.3.1 auf Seite 54), stellt die Implementierung des Kommunikations-Protokolls höhere Anforderungen. Jedoch lässt sich auch dies lösen, wie im Abschnitt 8.4.1 auf Seite 55 gezeigt.

8.6 Brücke zur Daten-Schicht

Es fehlt noch eine konkrete Implementierung, wie die Service-Schicht auf die Daten-Schicht zugreifen kann. Eine Möglichkeit wäre etwa die JDBC- oder ODBC-Schnittstelle zu verwenden, die von den meisten Datenbankmanagementsystemen bereitgestellt werden. Es sollte auch möglich sein, andere Daten-Quellen anzubinden, beispielsweise ein *Product Lifecycle Management System* (PLM) beziehungsweise ein *Product Data Management System* (PDM).

Für diese genannten Legacy-Systeme muss sehr wahrscheinlich die Brücke auf konventionelle Weise geschrieben werden. Zwar ließe sich auch hierfür ein MDA-Vorgehen definieren, allerdings wäre es einfacher, einen Experten diese Schnittstelle konventionell realisieren zu lassen. Denn dies wäre nicht die erste Anbindung, die für ein System realisiert würde. Außerdem hätte der MDA-Prozess

einen großen Overhead, der sich bei der Größe des Problems nicht lohnen sollte (siehe Kapitel 10 auf Seite 70).

Um unabhängig von der Daten-Schicht zu sein, wurde ein Klassengerüst erstellt, welches eine Factory-Klasse pro Daten-Objekt bereitstellt, beispielsweise `ItemFactory`. Diese Fabrik hat die Aufgabe, Service-Objekte zu generieren und diese dabei mit den Daten aus der Daten-Schicht zu füllen und Änderungen im Service-Objekt an die Daten-Schicht zu propagieren. Dazu wurden für den Prototypen drei Methoden festgesetzt: `create`, `findByPrimaryKey` und `list` (siehe Abschnitt 8.2.1 auf Seite 49).

Diese Methoden wurden bisher nur als Klassen-Signatur definiert. Die konkrete Implementierung ist noch zu definieren. Als Beispiel soll dazu eine klassische Datenbank dienen, genauergesagt jene Datenbank, die im Kapitel 7 auf Seite 36 erzeugt wurde. Da die Transformation Zugriff auf das zuvor erzeugte Datenschema hat, können alle Attribute im Service-Objekt auf die entsprechenden Attribute im Datenbank-Schema abgebildet werden.

Für das Beispiel der Klasse `Item` wird eine Anweisung der Art `INSERT INTO item (<Attribut Liste>) VALUES(<Werte Liste>)` erzeugt, um ein neues Objekt anzulegen (`create`). Die *Attribute Liste* ist dem Datenbank-Schema zu entnehmen. Die *Werte Liste* ist mit den Werten des Service-Objekts zu befüllen — und zwar in der Reihenfolge, wie in der *Attribute Liste*. Damit ist die Methode `create` vollständig implementiert.

Auch die Methode `list` ist relativ einfach zu implementieren. Hierzu müssen alle Datensätze aus der Datenbank ausgelesen werden, was mit einer SQL-Anweisung derart geschieht: `SELECT <Primary Keys> FROM item`. Die Liste mit den primären Schlüsseln kommt ebenfalls aus dem Datenbank-Schema. Das SQL-Statement liefert somit zu allen Objekten der Entität `item` die primären Schlüssel zurück. Um an die vollständigen Daten zu gelangen, ist die Factory-Methode `findByPrimaryKey` aufzurufen und dabei das entsprechende Primary Key-Objekt zu übergeben.

Die Methode `list` könnte alternativ auch vollständige Objekte mit allen Daten zurückliefern, jedoch treten dabei Probleme auf. So würden nämlich für alle Datenbank-Entitäten auch fertig initialisierte Service-Objekte erzeugt werden. Und dies nicht nur von der angefragten Klasse `Item`, sondern auch von allen referenzierten Klassen. Soll beispielsweise ein Objekt vom Typ `Item_version` erzeugt werden, so hält dieses Objekt eine Referenz auf ein Objekt vom Typ `Item`. Damit würde ein einfacher `list` Aufruf einen Großteil der Datenbestände auslesen und in Service-Objekten initialisieren.

Dieses Problem besteht auch bei dem einfachen Aufruf von `findByPrimaryKey`, weswegen es doch beachtet werden muss. Eine Lösung ist, ein konkret angefragtes Objekt, z.B. `Item_version`, zu initiieren. Enthält das Objekt ein Verweis auf ein anderes Objekt, z.B. zu `Item`, so wird dieses weitere Objekt noch nicht erzeugt. Erst wenn erstmals über die Getter-Methode in `Item_version` auf das Objekt `Item` zugegriffen werden soll, wird das Objekt tatsächlich erzeugt.

Damit wird aber nur verhindert, dass schlagartig ein Großteil der Datenbestände ausgelesen wird. Mit der Zeit hingegen existieren immer mehr Objekte im System. Deshalb muss es erlaubt sein, Objekte wieder zu deaktivieren. Dazu darf ein Client-Objekt nicht mehr direkt mit dem Service-Objekt verbunden sein, sondern nur über ein Stellvertreter-Objekt. Dieses Objekt ist einmal pro Typ vorhanden und sorgt dafür, dass ein tatsächliches Service-Objekt bei Bedarf vorhanden ist. Damit kann ein

Service-Objekt auch deaktiviert bzw. terminiert werden. Sobald der Client wieder darauf zugreifen will, wird dieses transparent vom Stellvertreter-Objekt initiiert.

Bleibt noch, dieses Konzept mit dem in Abschnitt 8.2.1 auf Seite 49 beschriebenen Transaktionen zu kombinieren. Da beim Prototypen pro konkreten Daten-Objekt maximal eine Instanz existiert, arbeiten Clients eventuell gemeinsam auf diesem Objekt. Alle Änderungen an einem Datensatz stehen damit unverzüglich den anderen Clients zur Verfügung. Damit könnte der Stellvertreter das Service-Objekt jederzeit speichern und eliminieren. Beim nächsten Zugriff auf das Service-Objekt würde er es ja wieder erzeugen und den gespeicherten Zustand auslesen.

Anders ist es, wenn jeder Client ein eigenes Service-Objekt erhält, wie bei den oben beschriebenen Transaktionen genannt. Dann könnte das Stellvertreter-Objekt zwar immer noch ein Service-Objekt jederzeit terminieren, jedoch müsste zuvor der Zustand gespeichert werden. Beim nächsten Zugriff des Clients müsste dieser eine Id, vergleichbar mit einer Session-Id, dem Stellvertreter-Objekt mitgeben. Dieses würde dann entweder das passende Service-Objekt zurückliefern oder aber ein neues Service-Objekt erzeugen und mit dem Zustand des ursprünglichen Objektes versehen.

Der hier beschriebene Ansatz weißt starke Ähnlichkeiten mit dem der *Enterprise JavaBeans* (EJB) auf. Das Stellvertreter-Objekt entspricht hier dem Home-Interface. Der auch für Transaktionen geeignete Entwurf mit den Sitzungs-IDs entspricht den Session-Beans und der zustandslose etwa den MessageBeans.[4] Der EJB-Ansatz ist vielfach im Einsatz und hat sich durchaus bewährt, wie die zahlreichen auf EJB basierenden kommerziellen Anwendungen zeigen.

Für die Implementierung dieses Management-Objekts ist eines der oben beschriebenen Verfahren zu wählen. Erst dann kann die Methode `findByPrimaryKey` implementiert werden. Dazu ist neben dem Stellvertreter-Objekt lediglich eine SELECT-Anfrage zu starten, die alle Attribute des Datensatzes ausliest — entsprechend dem Primary Key-Objekt. Anschließend ist ein Service-Objekt zu erzeugen und mit den gelesenen Daten zu befüllen.

Im Prototypen ist noch eine weitere Methode angedacht. Über diese können die Service-Objekte ihrem Management-Objekt mitteilen, wenn der Client die Daten geändert hat. Damit kann die Factory diese Änderungen auch sofort an die Daten-Schicht propagieren, welche dann die Daten persistent speichert.

8.7 Zusammenfassung

Dieses Kapitel zeigte die zahlreichen Transformationen, die für die Service-Schicht notwendig sind. Dabei galt es, drei wesentliche Bedingungen einzuhalten: die Software muss verteilt laufen, der Client kann leicht konventionell entwickelt werden und Transformationen sollen für möglichst viele Plattformen wiederverwendbar sein. Dazu musste das Klassen-Modell dahingehend angepasst werden, dass es Service- und Client-Klassen gibt, die ein gemeinsames Interface implementieren. Auch wurde eine Architektur entwickelt, mit der Zugriffe auf die Daten-Schicht gekapselt werden können. Dazu wurden weitere Klassen eingeführt, die ein konkretes Objekt erzeugen, aus der Daten-Schicht laden

[4] Vgl. [DP02, S. 42ff].

oder alle Objekte auflisten. Der Zugriff auf ein konkretes Objekt erfolgt dabei über ein Primary Key-Objekt. Ein weiteres Problem betrifft die Abhängigkeiten unter den einzelnen Modell-Elementen. So muss gespeichert werden, auf welches Attribut sich Zugriffsmethoden beziehen. Auch soll jedes Modell-Element im PSM leicht auf eine Anforderung im PIM zurückverfolgt werden können.

Kapitel 9

Transformationen der Client-Schicht

Letztendlich ist noch der Client zu implementieren. Wie bereits erwähnt, ist es ausdrücklich erlaubt, den Client auf konventionelle Weise zu entwickeln. Dieses Kapitel stellt dennoch ein Verfahren vor, mit dem sich Clients generieren lassen. Außerdem lässt sich das Layout des Clients auch so noch nachträglich durch einen Designer anpassen.

Als Client soll hier ein Web-Browser dienen, welcher durch Servlets erzeugte HTML-Seiten anzeigt. Über in den Seiten enthaltenen Links können andere Servlets und damit andere Funktionen aufgerufen werden. Die Servlets werden für den Prototypen durch eine Transformation erzeugt.

9.1 Generierung des Servlet-Codes

Entsprechend dem vorher vorgestellten Management-Objekt, welches jede Entity-Klasse besitzt, gibt es pro Klasse drei Servlets[1]. Das erste zeigt den gesamten Datensatz an (*View-Servlet*), das zweite erzeugt einen neuen Eintrag (*Create-Servlet*) und das dritte listet alle enthaltenen Objekte auf (*List-Servlet*). Dazu greifen die Servlets per definierten Client-Klassen auf die Service-Objekte zu.

Nun könnte der Code der Servlet-Klasse eine komplette HTML-Seite enthalten und diese vom Browser anzeigen lassen. Damit müsste aber auch das Layout der Seiten im Servlet-Code fest verdrahtet sein. Änderungen könnten nur durch Code-Anpassung erfolgen. Deshalb erzeugt das Servlet keine HTML-Seite, sondern beliefert eine Layout-Engine mit den benötigten Daten. Hierbei kommt die *Velocity-Template-Engine*[2] aus dem *Jakarta*-Projekt[3] zum Einsatz.

Das Servlet stellt dazu der Engine zur Laufzeit die benötigten Daten bereit. Die Template-Engine wertet anschließend das vom Servlet gewählte Template aus und übergibt diesem alle angeforderten Daten. Das Template sollte dabei von einem erfahrenen Designer erstellt werden. Für den Prototypen wurde auch dieses Template (als Vorschlag) erzeugt.

[1] Vgl. `http://java.sun.com/products/servlet/`, besucht am 22.04.2004.
[2] `http://jakarta.apache.org/velocity/`, besucht am 21.03.2004.
[3] `http://jakarta.apache.org/`, besucht am 22.04.2004.

Für die Servlet-Klasse gibt es zwei Anforderungen, möchte sie als Velocity-Servlet auftreten. Zum einen muss sie die Klasse `VelocityServlet` aus dem Package `org.apache.velocity.servlet` erweitern. Zum anderen muss eine Methode mit der nachfolgenden Signatur angelegt werden:

```
public Template handleRequest(
       HttpServletRequest request,
       HttpServletResponse response,
       Context context )
```

In dieser Methode stellt das Servlet der Template-Engine alle benötigten Daten über das Objekt `context` bereit. Das Servlet muss hierin eine Referenz auf die Factory des jeweiligen Typs erzeugen, etwa auf `ItemFactory`. Dieses Factory-Objekt genügt der Template-Engine, um per Template-Anweisungen auf alle Methoden des Service-Objekts zuzugreifen.

Für einen produktiven Einsatz empfiehlt es sich, auch weitere Informationen zu übergeben. Beispielsweise ist von Bedeutung, von welchem Typ ein Attribut ist. Somit lassen sich Integer-Werte rechtsbündig darstellen, anstatt linksbündig wie bei Strings. Lange Felder können so in HTML als TEXTAREA angelegt werden und zeigen somit mehrere Zeilen pro Attribut-Wert an. Auch Enumerations lassen sich so gesondert darstellen, etwa per Combo-Box oder via Radio-Buttons.

Ebenfalls interessant für den Designer sind Zusatzinformationen. Beispielsweise kann das Servlet eine Beschreibung der Attribute liefern. Diese Beschreibung kann dem Anwender als Hilfe bereitgestellt werden.

Letztendlich wird ein Servlet einen sehr simplen Aufbau haben, wie das Beispiel 7 auf der nächsten Seite zeigt. Die Komplexität liegt im Template, da hier die Position und das Aussehen jedes Attributs beschrieben ist. Das Servlet liefert lediglich die benötigten Daten.

9.2 Generierung des Beispiel-Templates

Die Template-Datei zur jeweiligen Klasse kann ebenfalls erzeugt werden, auch wenn dies nur begrenzt sinnvoll ist. Beim View-Template zum Anzeigen eines kompletten Datensatzes muss das Template zuerst das konkret anzuzeigende Objekt ermittelt. An diese Information gelangt der Client etwa dadurch, dass er zuvor eine Liste mit allen enthaltenen Objekten anzeigt oder aber per Suchfunktion das gewünschtes Objekt liefert. Über den Link zum View-Template kann hier das gewünschte Objekt als Parameter übergeben werden.

Hat das Template das gewünschte Objekt per Primary Key-Objekt identifiziert, kann es das Objekt erzeugen. Hierzu muss über die per `context` übergebene `Factory` gegangen werden. Die Velocity-Anweisung lautet hierfür:

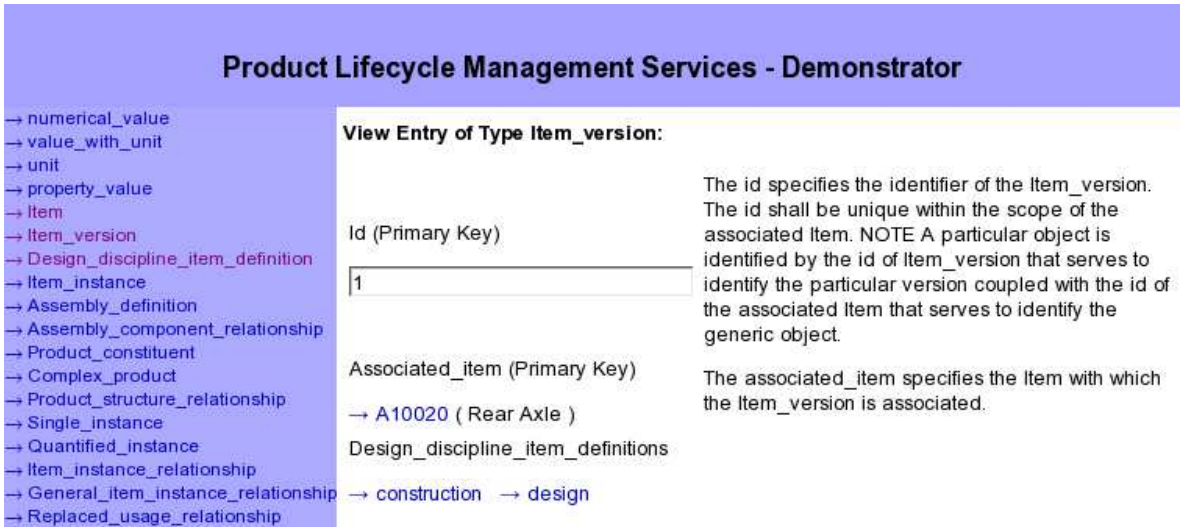
```
$entity = $factory.findByPrimaryKey( $primaryKeyObject ).
```

Beispiel-Programm 7 Methode `handleRequest` des erzeugten `ItemViewServlets`

```
public Template handleRequest(
   HttpServletRequest request,
   HttpServletResponse response,
   Context context ) {

   // get and store Factory for Item
   org.schreinert.masterthesis.step.item.clients.ItemFactoryClient
       factory = org.schreinert.masterthesis.step.item.clients
         .ItemFactoryClient.getInstance();
   context.put( "factory", factory );

   // store all attributes with types
   // and store the description of all attributes
   java.util.Hashtable parameters = new java.util.Hashtable();
   java.util.Hashtable help = new java.util.Hashtable();

   parameters.put("id", "String");
   help.put("id", new String("<p class='comment'><br>The id specifies ..."));

   parameters.put("name", "String");
   help.put("name", new String("<p class='comment'><br>The name ..."));

   parameters.put("description", "String");
   help.put("description", new String("<p class='comment'><br>The ... "));
   parameters.put("Item_versions", "Item_version");

   context.put("help", help);
   context.put("parameters", parameters);

   // generate template
   Template template = null;
   try
   {
       template =  getTemplate("templates/screens/ItemViewServlet.vm");
   }
   // stripped out detailed exception handling
   catch( Exception excp )
   {
     throw new RuntimeException(excp);
   }
   return template;
}
```

Ab sofort befindet sich hinter der Velocity-Variable `$entity` das gewünschte `Item`-Objekt . Der Zugriff auf die Attribute erfolgt dabei per `$entity.id` beziehungsweise `$entity.getId()`.

Das Ergebnis für das List-Servlet zeigt die Abbildung 9.1 auf der nächsten Seite. Im linken Bereich der Web-Seite ist eine Liste mit allen möglichen Service-Objekten. Ein Klick auf den jeweiligen Link führt zum passenden List-Servlet. Im rechten Teil der Seite befinden sich die Informationen der



Quelle: Eigene Abbildung.

Abbildung 9.1: View-Servlet für `Item_version`

aktuellen Anwendung. In der Abbildung sind dies die Werte der Attribute `Id`, `Associated_item` und `Design_discipline_item_definitions`, wobei die primären Attribute mit dem entsprechenden Hinweis versehen sind („Primary Key"). Das Attribut `Associated_item` hat als Wert einen Link auf das entsprechende `Item`-Objekt, hier mit der Id A10020. Zusätzlich wird der Name des Items mit ausgegeben. Hinter dem Wert von `Design_discipline_item_definions` befinden sich ebenfalls Links auf die Objekte, wobei hier mehrere Objekte aufgelistet sein können. Im Beispiel bedeutet dies, dass für das Teil `A10020` in der Version `1` die zwei Disziplinen `construction` und `design` vorhanden sind.

Der Bildschirm für die Liste aller z.B. Items sieht ähnlich aus, wie in Abbildung 9.2 auf der nächsten Seite zu sehen ist.

9.3 Zusammenfassung

Für den Prototypen wurde ein lauffähiger Web-Client generiert. Hierzu kam die Template-Engine Velocity zum Einsatz. Sie ermöglicht es, nachträglich das Aussehen des Clients zu verändern, ohne dabei Code anpassen zu müssen. Velocity benötigt dazu zwei Objekte: ein konkretes Template und ein Velocity-Servlet. Das Template enthält das Aussehen der zu erzeugenden Web-Seite. Alle anzuzeigenden Daten, wie etwa die Id oder der Name, werden zur Laufzeit in das Template eingebunden. Das Servlet dient dabei nur dem Zweck, das Template mit den benötigten Daten zu beliefern. Für den Prototypen gibt es pro Entity-Klasse drei Servlets, ein View-Servlet für den gesamten Datensatz, ein List-Servlet für eine Liste aller Objekte und ein Create-Servlet, um ein neues Objekt zu erzeugen.

Product Lifecycle Management Services - Demonstrator

→ numerical_value
→ value_with_unit
→ unit
→ property_value
→ Item
→ Item_version
→ Design_discipline_item_definition
→ Item_instance
→ Assembly_definition
→ Assembly_component_relationship
→ Product_constituent
→ Complex_product
→ Product_structure_relationship
→ Single_instance
→ Quantified_instance
→ Item_instance_relationship
→ General_item_instance_relationship
→ Replaced_usage_relationship

List of all Entities of Type Item:

	Id	Name	Description	Item_versions→Id
View	A10020	Rear Axle	Rear Axle 10.000 RPM	→ 1 → 2 → 3
View	A10030	Front Axle	Front Axle 10.000 RPM	→ 1
View	A12010	Axle House		→ 1
View	A12020	Axle SHAFT	Axle shaft for Z-Axle	→ 1

Quelle: Eigene Abbildung.

Abbildung 9.2: List-Servlet für `Item`

Teil III

Bewertung

Kapitel 10

Bewertung des Prototypen

Für den Prototypen wurde erstmals der komplette MDA-Prozess durchlaufen und es konnten zahlreiche Eindrücke gewonnen werden. Diese sollen in der folgenden Bewertung erörtert werden. Dazu gibt es sowohl positives, als auch negatives zu berichten, was die beiden folgenden Abschnitte zeigen. Zudem wurden Problemzonen identifiziert.

10.1 Vorteile der Model Driven Architecture

10.1.1 Geringerer Programmieraufwand

Wäre der PLM Service Standard auf konventionelle Weise zu entwickeln, so müsste der Programmierer zahlreiche, sich wiederholende Tipp-Arbeit erledigen. Beispielsweise müssen alle Attribute eine Getter- und eine Setter-Methode erhalten oder Assoziationen sind entsprechend ihren Endpunkten aufzulösen. Zudem müssten die Factory- und Primary Key-Klassen von Hand erstellt und implementiert werden. Gleiches gilt für die Anforderung, dass die Software verteilt laufen soll. Alle Schritte wären für jede Entity-Klasse zu wiederholen — und das für jedes Protokoll.

Für den MDA-Prozess muss der Entwickler hingegen nur sagen, wie die Entity-Klassen umzuwandeln sind. Dazu definiert er eine oder mehrere Transformations-Regeln, die anschließend auf alle Entity-Klassen angewandt werden. Dabei ist es unerheblich, auf wie viele Klassen die Transformation angewandt wird: Der Aufwand bleibt gleich groß, auch mit zunehmender Anzahl von Klassen. Bei der konventionellen Software-Entwicklung würde jede neue Klasse zusätzliche Arbeit bedeuten.

Ähnliches gilt auch für die notwendige Code-Dokumentation. Hier hat der Entwickler die zumeist lästige Aufgabe, seinen Programm-Code zu dokumentieren. Da auch bei konventioneller Software-Entwicklung eine Dokumentation im Entwurf vorhanden sein sollte, kann diese entsprechend übernommen werden. Dies bedeutet eventuell Copy-and-Paste – Arbeit. Der MDA-Prozess kann ebenfalls auf diese Dokumentation zugreifen und zudem sein erzeugtes Programm um weitere Erklärungen anreichern. Somit ließe sich jede generierte Zeile Code im Methoden-Körper dokumentieren.

10.1.2 Änderungen an einer zentralen Stelle

Wie bereits im vorigen Unterabschnitt genannt, sind die Transformationsregeln unabhängig von der Anzahl Klassen im PIM. Sie filtern die gewünschten Modell-Elemente heraus und führen auf ihnen die Transformation aus. Dies spart nicht nur Programmierarbeit, sondern erlaubt es auch, Programmänderungen einfach vorzunehmen.

Als Beispiel soll dienen, dass in einer ersten Software-Implementierung die Getter- und Setter-Methoden auf ihre einfachen Zugriffs-Aufgaben beschränkt sind: Der Getter liefert demnach lediglich das Attribut zurück und der Setter manipuliert es. Während der Wartungs-Phase des Software-Prozesses kommt die neue Anforderung auf, ein Sicherheits-Modell einzuführen, das den Zugriff auf die Attribute je nach Berechtigung gewährt oder ablehnt.

Bei konventioneller Software-Entwicklung bedeutet dies eine enorme Umprogrammierarbeit, da alle betroffenen Klassen geändert werden müssen. Anders beim MDA-Prozess. Hier würde die Transformation entsprechend erweitert werden. Zudem ließe sich über einen Transformations-Parameter entscheiden, ob die Sicherheits-Prüfung etwa zu Test-Zwecken abzuschalten ist oder welche Sicherheits-Richtlinie zu implementieren ist.

Aber auch wenn sich etwas am PIM ändert, hat dies kaum Auswirkungen auf die Transformationsregeln. Hat sich etwa der Klassen- oder Attribut-Name geändert oder ist eine Signatur verändert, so erzeugen die Transformationen entsprechend veränderten Programm-Code. Anders bei konventioneller Software-Programmierung. Hätte sich hier ein Methoden-Name geändert, müsste jede Klasse, in der diese Methode aufgerufen wird, gefunden und angepasst werden.

10.1.3 Code-Qualität in jeder Klasse gleich

Da die Logik für eine Zeile Code immer einen festen Ursprung hat, ist der erzeugte Code an jeder Stelle von gleich guter oder gleich schlechter Qualität. Somit wären etwa alle Klammern, die einen Methoden-Körper einleiten, an einer Stelle: entweder in der gleichen Zeile wie die Methoden-Signatur oder in der nachfolgenden Zeile. Dies erleichtert es einem Code-Reviewer, sich in den Programm-Text einzulesen und ihn leichter zu verstehen.

Es besteht auch weniger Gefahr, dass eine Problematik auf verschiedene Arten in verschiedenen Klassen gelöst wird: Ein Problem wird immer gleich gelöst und implementiert. Zwar könnte die Lösung dennoch inkorrekt sein, doch wäre es damit an allen Stellen inkorrekt. Damit wäre der Fehler bei einem Code-Review leichter zu entdecken.

10.1.4 Wissen ist in Transformationen gekapselt

Einer der größten Vorteile liegt in der Tatsache, dass sich Wissen über ein spezielles Problem kapseln lässt. So entwickelt ein Datenbank-Experte die Transformation für die Daten-Schicht und bestimmt zudem die Brücke zwischen Daten- und Service-Schicht. Der Web Service-Spezialist hingegen implementiert die Umsetzung für eine Web Services Architektur. Dazu muss dieser keinerlei Kenntnisse von den Experten aus anderen Gebieten haben.

Dabei ist hier sogar noch zu unterscheiden, wie konkret eine Transformation ist. Im Prototypen wird die Service-Schicht in einer der ersten Transformation dafür vorbereitet, verteilt aufgerufen zu werden (siehe 8.2.2 auf Seite 51). Erst später folgt die Festsetzung, welches konkrete Protokoll für die Kommunikation verwendet wird. Gibt es eine Entscheidung für EJBs, so würde eine weitere Transformation den Einsatz dieses Protokolls vorbereiten, jedoch ohne konkreten Programm-Code zu erzeugen. Vielmehr erzeugt die Transformation beispielsweise Home-Interfaces und markiert die Klassen als EJB-Objekte. Damit braucht dieser Entwickler auch nicht zu wissen, welche konkrete Version von EJB eingesetzt wird. Diese Kenntnis besitzt ein weiterer Experte.

10.1.5 Transformationen sind wiederverwendbar

Ein zusätzlicher Aspekt der Kapselung von Wissen ist, dass die meisten Entwickler das zu implementierende Software-Produkt weder verstehen noch kennen müssen. Sie entwickeln lediglich ihre Transformation, ohne zu wissen, für was konkret sie eingesetzt wird.

Dieser Umstand macht es möglich, eine einmal definierte Transformation für verschiedene Plattformen einzusetzen. So kann die Transformation, die für die Verteilung sorgt, sowohl für RMI als auch für CORBA unverändert eingesetzt werden.

Darüberhinaus können teilweise Transformationen auch für andere Projekte verwendet werden, da sie Problem-neutral sind. Dies gilt besonders für PIM-nach-Code – Transformationen. Sie arbeiten auf einem Modell (PSM), welches das eigentliche Problem bereits gelöst hat und alle Informationen für die Ziel-Plattform bereit hält. Somit kann ohne weiteres Wissen der Programm-Code erzeugt werden.

Für den kommerziellen Einsatz hat dies den großen Vorteil, dass Tranformationen sowohl verkauft, als auch eingekauft werden können. Somit muss nicht jedes Unternehmen für Ihre Projekte eine PSM-nach-Code – Transformation entwickeln. Aber auch für besondere Problemstellungen, bei denen Experten-Wissen vorhanden sein muss, kann es lohnender sein, die Transformation einzukaufen.

10.2 Nachteile der Model Driven Architecture

10.2.1 Sehr abstraktes Programmieren der Transformationen

Computer-Software zu entwickeln ist auch bei konventioneller Entwicklungsweise ein sehr abstrakter Prozess. Es ist jedoch noch weit komplexer und abstrakter, Transformationen zu schreiben. Dies resultiert daraus, dass der Entwickler beispielsweise nicht mehr mit der konkreten Klassen `Item` und seinen Attributen `id` und `name` arbeitet. Vielmehr definiert er abstrakt, dass es eine Klasse gibt, die Attribute enthalten kann. Auf eine solche Klasse wird eine Transformationsregel angewandt. Der Entwickler arbeitet somit nicht mehr auf Modell-Ebene, sondern auf Meta-Ebene.[1]

Zudem haben die meisten Transformationen die Aufgabe, eine kleine Änderung am Modell vorzunehmen. Damit ist das Ergebnis aus Sicht des Entwicklers nur von kleinen Nutzen, da er mit seiner Transformation keine sofort lauffähige Software generieren kann. Dies mindert das Erfolgserlebnis des Entwicklers.

[1] Vgl. [KWB03, S. 85ff].

10.2.2 PIM enthält nicht alle Informationen

Es ist fester Bestandteil des MDA-Prozesses, dass lediglich das PIM alle Informationen zum Problem enthält. Alle Plattform-spezifischen Angaben sind entweder implizit in den Transformations-Regeln enthalten oder aber explizit über die Transformations-Parameter angegeben. Und genau dies stellt ein Problem dar.

Ändern sich für eine neue Software-Generation die Anforderungen, so muss das Plattform-unabhängige PIM angepasst werden. Hier kann es jedoch vorkommen, dass diese Änderungen den Angaben der früher festgelegten und validierten Transformations-Parametern widersprechen.

Dies ist auch deshalb negativ, da das zukünftige UML in der Version 2 die Modelle verifizieren und sogar ausführen können soll.[2] Da die Parameter außerhalb des Modells liegen, können diese weder mit-verifiziert werden, noch entspricht die per UML „ausgeführte" Software dem später generierten Programm.

10.2.3 Vorgehen bei der Transformations-Entwicklung

Design-Fehler innerhalb einer Transformation können zur Folge haben, dass zahlreiche Transformationen ebenfalls unter dem Fehler leiden und angepasst werden müssen. Ein Beispiel ist, dass eine der letzten Transformationen im MDA-Prozess eine Dependency von einem Attribut auf die Getter-Methode benötigt. Diese Dependency hat eine frühere Transformation zwar geliefert, jedoch gibt es exakt die selbe Ahängigkeit für die Setter-Methode — beide sind mit «`Accessor`» markiert. Damit kann die Transformation die gesuchte Getter-Methode nicht eindeutig bestimmen — lediglich über die Methoden-Signatur oder -Namen, was hier nicht betrachtet werden soll. Die Folge wäre, dass die erste Transformation die beiden Dependencies differenziert kennzeichnen müsste, etwa mit «`Getter Accessor`» und «`Setter Accessor`». Damit müssten zwangsweise auch alle Transformationen zwischen den beiden angepasst werden.

10.2.4 Definition der Transformations-Schnittstellen

Der hier gezeigte Prototyp wurde von einer Person entwickelt und alle Transformationen sind Eigenschöpfungen. Somit war es kein Problem, eine Transformation auf einer anderen aufbauen zu lassen. Dies wäre jedoch nicht so einfach gewesen, wenn Transformationen zugekauft wären oder mehrere Entwickler mitgearbeitet hätten.

Eine eingekaufte Transformation muss zwingend definieren, aufgrund welcher Eigenschaft sie ein Modell-Element bearbeitet. So muss die PSM-nach-Code – Transformation für EJBs angeben, dass EJB-Home-Interfaces mit einem definierten Stereotypen zu kennzeichnen sind. Gleiches gilt dann auch für die Bean-Typen, etwa für die Unterscheidung nach EntityBean und SessionBean.

Auch ist anzugeben, wie die durch die Transformation manipulierten Modell-Elemente zu erkennen sind. Dies ist besonders dann wichtig, wenn die Transformation ein weiteres Modell erzeugt. Denn

[2] Vgl. [Boy03].

hier gibt es weitere Transformationen, die mit den Ergebnissen der ersten arbeiten müssen. Im Prototyp wäre dies beispielsweise die Transformation, welche die Verteiltheit herstellt. Die Transformation für ein konkretes Kommunikations-Protokoll muss wissen, welche Klassen die Clients und welche die Services sind. Folglich müsste die erste Transformation diese Modell-Elemente so kennzeichnen, wie es die eingekaufte Transformation benötigt.

10.3 Problemzonen der Model Driven Architecture

10.3.1 Vergleich des Code-Umfangs

Für den Prototypen waren mit ca. 7.000 Zeilen Code gut doppelt so viel zu programmieren, wie die damit generierten ca. 3.500 Zeilen Java-Code. Wird die deutlich höhere Komplexität und Abstraktheit zum Code-Verhältnis hinzugezogen, wäre dies ein klares Argument gegen MDA. Immerhin bedeutet jede geschriebene Programmzeile Kosten. Kurzfristig gesehen, wäre damit der konventionelle Ansatz beim Prototypen vorteilhafter.

Jedoch implementiert dieser Prototyp nur einen Bruchteil des eigentlichen PLM Service Standards. Jede neu hinzukommende Klasse hätte keinerlei Auswirkungen auf die Transformationsregeln. Es würde folglich keine neue Zeile Programm-Code in den Transformationen hinzukommen. Somit wäre die Umsetzung des vollständigen Standards deutlich vorteilhafter — gemessen an der Code-Größe.

Hieraus folgt, dass der Einsatz von MDA grundsätzlich nicht für jedes Projekt wirtschaftlich ist. Kleinere Projekte, bei denen die Änderungen und Wartungsaufgaben gering sind, mag die konventionelle Software-Programmierung weiterhin kostengünstiger sein. MDA lohnt sich besonders bei Projekten, bei denen häufig die ähnliche Programmierarbeit anfällt oder zahlreiche Änderungen im PIM zu erwarten sind.

10.3.2 Fehlersuche und -behebung

Es lässt sich nicht pauschal sagen, ob Fehler dank MDA leichter gefunden werden können oder eher schwerer. MDA bietet die Möglichkeit, Debugging-Code in den generierten Methoden-Code hinzuzufügen. Zwar ist dies auch bei konventioneller Entwicklung möglich, doch kann bei MDA ein Transformations-Parameter bestimmen, ob Debugging-Code eingefügt wird und in welchem Umfang. Nach einem erneuten Durchlauf aller Transformationen wäre das Programm bereit zur Fehlersuche.

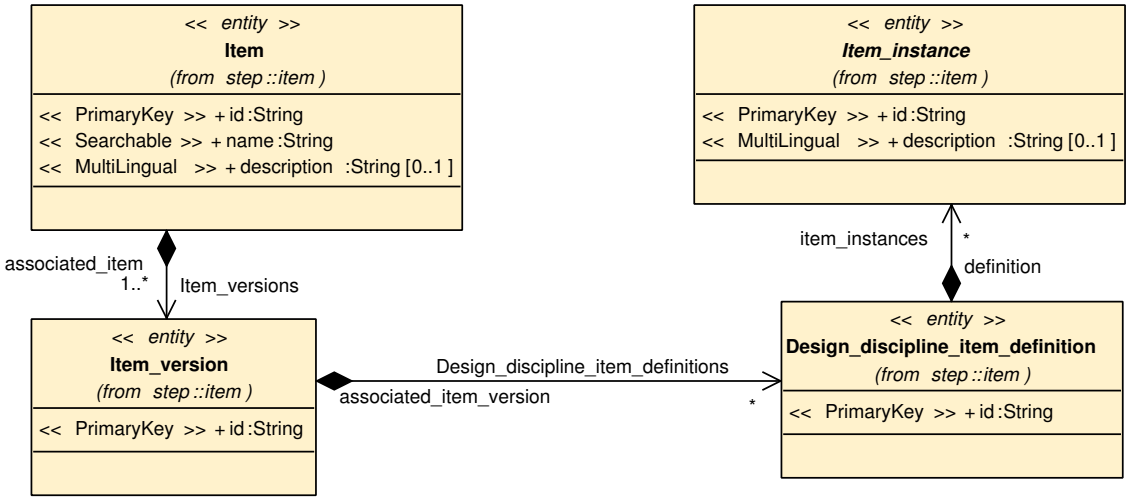
Auch lässt sich Debugging-Code wieder vollständig entfernen, etwa bei einem neuen Release. Dazu muss nur der Parameter geändert werden. Ähnliches ist zwar durch diverse Tools bei konventioneller Programmierung realisierbar, jedoch bleiben die Code-Zeilen später vorhanden. Es wird lediglich der Debugging-Output zur Laufzeit per Parameter abgeschaltet. Beispiele hierfür sind etwa *log4j* für Java oder *log4net* für C#.[3] Bei MDA lässt sich das Logging auch komplett aus dem Code entfernen.

Erschwert wird hingegen das Debugging der Transformationen. Hier kann ein fehlerhaft erzeugter Programm-Code entweder durch das PIM verursacht sein oder aber — was wahrscheinlicher ist —

[3] `http://logging.apache.org/`, besucht am 21.04.2004.

durch Fehler in den Transformationen. Während des Prototypens sind zahlreiche solcher Fehler aufgetreten und die Fehlersuche war häufig sehr aufwändig. So hat das reine Fehlerbeseitigen über einen Personenmonat in Anspruch genommen.

Es ist nicht immer klar, in welcher Transformation der Fehler auftritt. Ein Problemfall war, dass Kompositionen inkorrekt aufgelöst wurden. Für das Datenbank-Schema bedeutet die Komposition zwischen `Item` und `Item_version` beispielsweise, dass `Item_version` eine Referenz auf `Item` enthält, obwohl es im Modell anders abgebildet ist (siehe Abbildung 10.1).



Quelle: Nach [FHLT03, S. 17]

Abbildung 10.1: Komposition zwischen den Klassen der Produktstruktur

Damit folgt auch, dass beim Zugriff der Service-Schicht auf die Daten-Schicht die Komposition hierfür genauso aufzulösen ist. Anders in der Service-Klasse. Hier muss `Item` einen Container mit den `Item_versions` enthalten und zusätzlich muss `Item_version` auf ein `Item` referenzieren. Da es eine komplexe Aufgabe ist, Assoziationen zu bearbeiten, wurde für beide Schichten die gleiche Helper-Methode zum Auflösen verwendet. Dies führte zu Problemen, die nur schwer gelöst werden konnten.

Besonders große Probleme gab es jedoch bei der Brücke zwischen Service- und Daten-Schicht. Soll beispielsweise ein `Item_version` aus der Daten-Schicht gelesen werden, so ist natürlich ein Objekt vom Typ `Item_version` in der Service-Schicht zu erzeugen. Jedoch befindet sich in der Datenbank zusätzlich zu deren Attributen eine Referenz auf das dazugehörige `Item` — und zwar hier als Referenz auf die Id. Somit muss beim Lesen des `Item_version`-Datensatzes die Referenz auf `Item` anders behandelt werden. Der Aufruf sieht dann etwa so aus:

```
new Item_versionService("id", "name", "beschreibung",
    new ItemService("id", "name") )
```

Der Aufruf von `Item_instance` ist entsprechend komplexer, da hierzu alle „Zwischenklassen" bis zum `Item` zu initiieren sind.

```
new Item_instanceService( (String)rs.getObject("id"), clients.
   Design_discipline_item_definitionFactoryClient.getInstance().findByPrimaryKey(
   new Design_discipline_item_definitionPrimaryKey(
   (String)rs.getObject("definition__p3355"),
   clients.Item_versionFactoryClient.getInstance().findByPrimaryKey(
   new Item_versionPrimaryKey( (String)rs.getObject("definition__p1736608683"),
   clients.ItemFactoryClient.getInstance().findByPrimaryKey(
   new ItemPrimaryKey( (String)rs.getObject("definition__p576027949") ) ) ) ) )
)
```

Von der Datenbank wird lediglich eine flache Liste mit allen Werten geliefert. Die Brücken-Software muss dann selbst entscheiden, welcher Wert zum `Item` und welcher zu `Item_version` gehört. Hierbei kam es zu großen Problemen bei der Kommata-Setzung und beim Klammern-Schließen. Bei konventioneller Software-Programmierung wäre diese Aufgabe mühelos zu lösen. Beim Prototypen nahm dieses Problem zwei Personentage in Anspruch.

10.3.3 Wiederverwendbarkeit der Transformationen

Das eigentliche MDA-Schema sieht zwei Transformationen pro Plattform vor: eine PIM-nach-PSM und eine PSM-nach-Code –Transformation. Wie bereits geschrieben, kann die PSM-nach-Code – Transformation ohne weiteres für verschiedene Problemstellungen wiederverwendet werden.

Anders sieht es bei der PIM-nach-PSM – Transformation aus. Diese Transformation ist auf eine konkrete Problemdomäne zugeschnitten und kann von daher höchstens für ähnlich gelagerte Projekte wiederverwendet werden. Beim Prototypen hat sich gezeigt, dass mehr als eine PIM-nach-PSM – Transformation angebracht ist.

So wurde für den Prototypen ein weiteres Kommunikations-Protokoll testweise implementiert. Das Ergebnis war, dass lediglich eine neue Transformationsregel für dieses Protokoll zu implementieren war. Alle anderen Transformationen blieben hingegen komplett unberührt.

Dies war nur möglich, da konsequenter Weise sämtliche Logik in jeweils eine eigene Transformation gepackt wurde. Wie gesagt, gibt es beim obigen Beispiel eine Transformation, welche die Verteilung erzeugt und jeweils eine pro Protokoll. Die nachfolgenden Transformationen konnten damit auf einem abstrakten Modell der Client/Server-Architektur arbeiten, ohne sich um das Protokoll zu kümmern.

Für zukünftige Projekte bedeutet dies, dass auf eine strikte Trennung der Logiken zu achten ist. Somit sollten Transformationen — ähnlich den Programm-Modulen — eine feine aber sinnvolle Granularität besitzen.

Hier setzt jedoch ein bereits beschriebener Nachteil an. Für jede eigenständige Transformation hat dies zur Folge, dass das Eingabe-Modell und das Ausgabe-Modell genau zu spezifizieren ist. Anderweitig besteht die Gefahr, dass vorangehende Transformationen das Modell so manipulieren, dass eine Transformation die geforderte Struktur nicht vorfindet und fehlerhaft arbeitet. Somit wird die Wiederverwendbarkeit dadurch erkauft, dass die Transformations-Entwickler restriktiert werden und die Schnittstellen der Transformationen sehr formal zu beschreiben sind.

10.3.4 Eignung für Prototypen-Entwicklung

Müssen die Transformationen für eine zu erstellende Anwendung selber entwickelt werden, so vergeht sehr viel Zeit, bis sich eine erste lauffähige Software generieren lässt. Beim Prototypen vergingen ca. 3,5 Personen*monate* mit Programmierarbeit, bis der Java-Compiler zum ersten Mal fehlerfrei durchlief. Um die Transformationen anschließend so zu ändern, das die erzeugte Software wie gewünscht lief, vergingen nur 2 Personen*tage*.

Dieses Beispiel soll zeigen, dass es mit MDA sehr lang braucht, bis erstmals eine lauffähige Software erzeugt werden kann. Dem Kunden kann somit erst sehr spät ein Prototyp demonstriert werden, was eventuell benötigte Änderungen teuer werden lässt.

Anders jedoch, wenn der Markt eine große Anzahl von Transformationen zum Kauf anbietet. Hier lässt sich aus dem Modell sehr schnell ein Prototyp erstellen. Im optimalen Fall könnte dem Kunden sogar sofort ein Prototyp gezeigt werden. So kann zusammen mit dem Kunden das PIM erstellt beziehungsweise vervollständigt werden. Dies ist sinnvoll, da der Kunde weiß, was er haben will. Jedoch werden seine Wünsche von den Entwicklern zumeist anders verstanden, wie eigentlich gemeint.[4] Dank MDA können jedoch Entwickler und Kunden sofort einen lauffähigen Prototypen erzeugen. Somit fällt es den Entwicklern leichter, die Bedürfnisse des Kunden zu verstehen. Dies aber nur unter der Annahme, dass entsprechende Transformationen vorhanden sind und nicht erst für die Anwendung entwickelt werden müssen.

10.3.5 Nachvollziehbarkeit des Transformations-Prozesses

Ein Vorteil des MDA-Prozesses ist, dass jedes Element im PSM und jede Zeile Programm-Code auf eine Anforderung im PIM zurückzuführen sei (siehe Abschnitt 8.3.3 auf Seite 55). Diese eigentlich sehr positive Eigenschaft ist nur dann korrekt, wenn die Transformationen diese Verbindungen auch sinnvoll anlegen.

Beim Prototypen wurden nur Verbindungen angelegt, die für nachfolgende Transformationen notwendig waren, etwa die zur Client-Klasse gehörige Service- oder Factory-Klasse. Diese Verbindungen erhielten zwar sinnvolle Namen, allerdings kann nur der Transformations-Entwickler wirklich etwas damit anfangen. Hier muss ein Spagat gefunden werden, damit sich Informationen für den späteren Anwender von denen unterscheiden, die für die Transformationen nötig sind.

Auch haben die prototypischen Transformationen lediglich die Modell-Elemente manipuliert. Für die Übersichtlichkeit empfiehlt es sich, neue Diagramme im XMI-Modell zu erzeugen und hier zusammenhängende Informationen verständlich abzubilden. Dies erleichtert es dem Anwender, das PSM genau nachverfolgen zu können und zu erkennen, warum und wie eine Klasse manipuliert wurde.

Im erzeugten Programm-Code lassen sich diese Verknüpfungen nur durch Kommentare abbilden. Hier ist etwa denkbar, für jeden zusammenhängenden Code-Block die Existenzgründe für diesen aufzuzählen. Auch denkbar ist, die Transformation anzugeben, welche den Code eingebaut beziehungsweise veranlasst hat. Dies erleichtert außerdem das Debugging der Transformationen, bedarf dafür aber eines erhöhten Aufwands, um die Transformation zu entwickeln. Denn so muss der Entwickler neben dem eigentlichen Programm-Code noch eine gute Dokumentation generieren.

[4] Stichwort *Kommunikationsproblem*, vgl. [Par98, S. 9ff].

10.4 Zusammenfassung

Die Methodik der Model Driven Architectures bietet zahlreiche Vorteile. So hat der Prototyp gezeigt, dass häufige, gleiche Programmierarbeit mit Hilfe einer Transformation schnell erledigt werden kann. Auch lassen sich dank MDA Änderungen an einer Stelle vornehmen, nämlich in der Transformation. Besonders interessant ist, dass sich komplexes Wissen in einer Transformation kapseln und einfach benutzen lässt. Die Wiederverwendbarkeit von Transformationen macht MDA mittel- und längerfristig sehr interessant. Dagegen stehen jedoch auch Nachteile, etwa dass die Schnittstellen der Transformationen exakt zu definieren sind. Zudem ist es sehr abstrakt, Transformationen zu entwickeln. Vor- oder nachteilhaft kann MDA in den Problemzonen sein — je nach Situation. So könnte die Anzahl der Code-Zeilen der Transformationen den der zu generierten Software übersteigen. Auch ist die Fehlersuche teils einfacher, etwa im erzeugten Programmcode. Anders die Fehlersuche in den Transformationen. Diese kann sehr aufwändig sein. Auch bedürfen wiederverwendbare Transformationen eine exakte Definition über die Ein- und Ausgabemodelle. Letztendlich besteht die Gefahr, dass die Prozesse nur sehr schwer nachvollziehbar sind, wenn die Transformationen keine detaillierten Informationen und Abhängigkeiten anbieten.

Kapitel 11

Transformationen formalisieren

Das Kapitel 5 auf Seite 29 führte bereits an, dass Transformationen zu trennen sind zwischen den Methoden und den eigentlichen Transformations-Regeln. Dieses Kapitel geht dazu besonders auf den zweiten Teil ein und versucht, die Transformations-Regeln zu formalisieren. Die Methoden um auf das Modell zuzugreifen, hängen dabei von der Sprache der Regeln ab. Der Prototyp hat bereits gezeigt, dass ein Modell sehr umfangreich manipuliert werden kann. Jetzt gilt es, die Regeln aus den prototypischen Transformationen zu extrahieren.

Dieses Kapitel zeigt zwei Möglichkeiten, wie eine Transformation über Regeln angegeben werden kann: per *Template Engine* oder per *Object Constraint Language*. Die Template Engine eignet sich besonders für PSM-nach-Code – Transformationen, während der Schwerpunkt von OCL auf PIM-nach-PSM liegt.

11.1 Template Engine

Ein Open-Source Projekt, das sich der Model Driven Architecture widmet, ist AndroMDA[1]. Das als ant-Task[2] realisierte Transformations-Tool bedient sich dazu der *Velocity Template Engine*. Der Prototyp hat diese Template Engine ebenfalls eingesetzt, hier jedoch um den Servlets ein veränderbares Aussehen zu ermöglichen (siehe Kapitel 9 auf Seite 64). Hier soll sie dazu verwendet werden, das Modell in Programm-Code zu transferieren.

AndroMDA bietet hierzu der Template Engine das Objekt `$class` mit der zu implementierenden Klasse, das UmlPackage `$model` wie von JMI angeboten (vgl. 3.2 auf Seite 21), ein Helper-Objekt `$transform`, sowie das aktuelle Datum in `$date` und `$str`, um Strings zu manipulieren.

Als Beispiel soll die PSM-nach-SQL – Transformation betrachtet werden. Die zu erzeugende SQL-Datei soll über ein Velocity-Template generiert werden.

[1] `http://www.andromda.org`, besucht am 20.04.2004.
[2] `http://ant.apache.org/`, besucht am 21.04.2004.

Wie im Abschnitt 7.3 auf Seite 45 nachzulesen ist, enthält die zu generierende SQL-Datei alle Klassen, die mit dem Tag «`entity`» gekennzeichnet sind. Dazu muss das Template zuerst vom Transformations-Tool alle entsprechenden UML-Klassen anfordern:

```
## set name of entity stereotype we are looking for
#set ( $entityStereoName = "entity" )

## iterato over all ModelElements
#foreach ( $modelElement in $transform.modelElements )
  #set ( $isEntity = false )

  ## iterate over all Steretypes the ModelElement contains
  #foreach ( $stereotype in $modelElement.getStereotype() )

    ## we are looking for $entityStereName
    #if ( stereotype.name == $entityStereoName )
      ## yep! found it!
      #set ( $isEntity == true )
    #end
  #end

  ## so if it is an entity we will go on
  #if( $isEntity == true )

     ## remember: we provide the macro performTransformation
     performTransformation( modelElement )

  #end
#end
```

Der obige Code-Auszug ist relativ aufwändig im Vergleich zur Aufgabe. Grund ist, dass AndroMDA hierfür keine komfortable Methode bereitstellt. Dies könnte durch Macros oder zusätzliche Cartridges (Transformationstools) vereinfacht werden.

Nun sind alle Entity-Klassen gefunden und es kann das Datenbank-Schema ausgegeben werden. Der Template-Code für diese Transformation wurde der Übersichtlichkeit halber in ein Macro namens `performTransformation( ModelElement )` gepackt.

```
#set ( $commentChar = "#" )
#macro performTransformation( modelElement )

  ## call a macro that is printing documentation
  ## parameters $modelElement = the concrete ModelElement
  ## parameter $commentChar = the line prefix (comment character)
  #printDocumentation( $modelElement, $commentChar )

  CREATE TABLE $modelElement.name (

    ## now print all attributes ...
```

```
        #foreach ( $attribute in $modelElement.ownedElement )
           #printDocumentation( $attribute, $commentChar )
           $attribute.name $attribute.type.name
           #printMultiplicty( $attribute )
           #printReferences( $attribute )
        #end

        #printPrimaryKey( $modelElement )

        #printForeignKey( $modelElement )

      );

    #end
```

Die im obigen Template vorhandenen weiteren Macros werden hier nicht weiter definiert. Im Grunde muss hierfür auf das JMI-Interface zugegriffen werden. Somit unterscheidet sich diese Implementierung der Transformations-Regel nur unwesentlich von der im Prototypen. Hierfür könnten angepasste Helper-Objekte oder Cartridges bereitgestellt werden.

Auch ist zu erwähnen, dass die prototypischen Transformationen auf eine Implementierung in Java ausgelegt sind. Es wäre alternativ auch möglich, Modelle während der PIM-nach-PSM – Transformationen zu erzeugen beziehungsweise zu markieren, die leichter von AndroMDA bearbeitet werden können.

So bietet AndroMDA eine Methode an, um aus einem Java-Datentypen den entsprechenden SQL-Datentyp zu finden. Somit ließe sich die Aufgabe im Abschnitt 7.2.2 auf Seite 38 schneller und einfacher erledigen. Auch ist der Zugriff auf Tagged-Values wesentlich einfacher, als auf Abhängigkeiten oder Stereotypen. Es empfiehlt sich somit, frühere Transformationen dahingehend zu trimmen, dass das Modell von nachfolgenden Transformationen leichter zu benutzen ist.

11.2 OCL als Transformations-Sprache

Für die PIM-nach-PSM – Transformation eignet sich besser eine andere, formalere Sprache. Hierzu kann die bereits erwähnte *Object Constraint Language* weiterhelfen. Sie ist in der Version 2 sogar dafür entwickelt worden, den MDA-Prozess gezielt zu unterstützen. Hierzu bietet sie etwa die Befehle `let` und `def` an, über die sich neue Attribute und Methoden erzeugen lassen.

Diese Konstrukte sind jedoch ungenügend, um damit komplexe Transformationen abzubilden. Aus diesem Grund wird ein neues Schema erzeugt, welches OCL zur Transformationssprache erweitert. Der Aufbau entspricht dabei dem in [KWB03, Seite 93ff].

Das grobe Gerüst ähnelt dabei einer Methoden-Definition. Eine Transformations-Regel beginnt mit dem Wort `Transformation` gefolgt von einem Namen. Der Name hat für die Transformation keine Bedeutung, er dient nur dem besseren Verständnis. Außerdem folgen zwei Typnamen in Klammern, welche die Quell- und Zielsprache spezifizieren. Nachfolgend ein Beispiel, um öffentliche Attribute privat zu machen und ihnen Zugriffsmethoden zu geben.

```
Transformation PublicToPrivate(UML, UML) {
  //...
}
```

Im Körper der Definition befinden sich verschiedene Bereiche. Der Bereich `params` bestimmt Parameter für die Transformation. Hier lässt sich u.a. der Prefix für die Getter- und Setter-Methoden angeben.

```
params
   setterPrefix : String = "set";
   getterPrefix : String = "get";
```

Nun muss das Meta-Objekt bestimmt werden, welches zu transformieren ist. Dies geschieht im Bereich `source`. Der Bereich `target` gibt die Objekte an, die durch die Transformation entstehen beziehungsweise verändert werden.

```
source
   sourceAttribute : UML::Attribute;
target
   targetAttribute : UML::Attribute;
   getter          : UML::Operation;
   setter          : UML::Operation;
```

Das hier vorgestellte Prinzip sieht vor, dass Transformationen nicht explizit aufgerufen werden. Vielmehr werden sie ausgeführt, wenn eine bestimmte Bedingung erfüllt ist. Diese Bedingung findet sich im Bereich `source condition`. Im Beispiel wäre diese Bedingung, dass das betroffene Attribut nicht privat ist, also etwa öffentlich oder geschützt (protected)

```
source condition
   sourceAttribute.visibility <> VisibilityKind::private;
```

Nun folgt der gewünschte Zielzustand. Dieser wird im Bereich `target condition` beschrieben und entspricht der bekannten OCL-Semantik. Aufgabe des dazugehörigen Transformationstools ist es, das Modell so zu transformieren, dass die `target condition` erfüllt ist. Für das Beispiel muss die Sichtbarkeit des Attributs auf privat gesetzt werden und es ist ein ein Getter, sowie ein Setter zu erzeugen.

```
target condition
   targetAttribute.visibility = VisibilityKind::private and
   targetAttribute.owner = sourceAttribute.owner and
   setter.name = setterPrefix.concat( targetAttribute.name ) and
   setter.type = oclVoid and
   setter.parameters->exists( param | param.name = targetAttribute.name and
                                      param.name = targetAttribute.type ) and
```

```
setter.owner = sourceAttribute.owner and
getter.name = getterPrefix.concat( targetAttribute.name ) and
getter.type = targetAttribute.type and
getter.parameters->isEmpty() and
getter.owner = sourceAttribute.owner
```

Nun folgt je nach Transformation das Schlüsselwort `bidirectional` oder `unidirectional`. Dieses gibt an, ob die Transformation in beide Richtung anwendbar ist und somit das PSM auf das PIM zurückzuführen ist.

Der letzte Bereich mit Namen `mapping` verweist auf bestehende Transformationsregeln. Hier wird gesagt, dass es eine andere Transformation gibt, welche ein aufgeführtes Quell-Objekt in ein Ziel-Objekt überführt. Dazu gibt es einen neuen infix Operator `<~>`, der angibt, dass der linke Ausdruck in den rechten überführt werden kann. Bei Bidirectionalität gilt auch der Rückweg.

```
mapping
   sourceAttribute.name <~> targetAttribute.name;
   sourceAttribute.type <~> sourceAttribute.type;
```

Diese Mapping-Regel ist notwendig, da beispielsweise ein UML-Ausgangsmodell in ein SQL-Zielmodell zu transformieren ist. Hierbei unterliegt die Namenskonvention in SQL anderen Regeln, als die in z.B. Java. Auch gibt es in SQL andere Datentypen.

Das nachfolgende Beispiel demonstriert, wie für das SQL-Schema eine vorhandene Generalisierung aufgelöst wird. Dabei wird das selbe Verfahren verwendet, wie im Abschnitt 5 auf Seite 42 beschrieben ist. Diese Transformation erzeugt vorerst noch ein UML-Modell. Erst in einer weiteren Transformation erfolgt die Umwandlung in SQL. Hierfür sei auf [KWB03, S. 101ff] verwiesen.

```
Transformation derivedClassToTable(UML, UML) {
   params
      entityStereoName : String = "Entity";
   source
      child    : UML::Classifier;
      parent   : UML::Classifier;
      model    : METADATA::UmlPackage;
   target
      aeParent : SQL::AssociationEnd;
      aeChild  : SQL::AssociationEnd;
      asso     : SQL::Association;
```

Im oben dargestellten Teil der Transformation befindet sich unter anderem der Parameter `entityStereoName`. Dieser enthält den Namen des Stereotypes, der die Entity-Klassen markiert. Nur solche Klassen sind für das SQL-Schema relevant. Als `source` sind drei Objekte anzugeben: `child` beschreibt die zu transformierende UML-Klasse, `parent` enthält den Super-Typen und `model` ermöglicht den Zugriff auf das Meta-Modell. Als Ziel entsteht eine Assoziation `asso` mit den zwei Endpunkten `aeChild` und `aeParent`.

Diese Transformation betrifft nur solche Klassen, die mit dem Stereotypen «`Entity`» markiert sind und zudem eine weitere Klasse spezialisieren.

```
source condition
   child.stereotypes->exists( s | s.name = entityStereoName ) and
   model.generalizations->exists( g | g.child = child and
       g.parent = parent and
       g.child.stereotypes->exists( s | s.name = entityStereoName) )
```

Der gewünschte Zielzustand gibt zuerst an, dass die vorher existierende Generalisierung verschwunden sein muss. Zudem wird die Assoziation `asso` mit den beiden Endpunkten `aeChild` und `aeParent` verknüpft. Dabei ist `aeParent` eine Komposition und ist mit dem `parent` in Form einer 1:1-Beziehung verankert. Der Endpunkt `aeChild` ist mit `child` verbunden und ist optional (0:1-Beziehung).

```
target condition
   not model.generalizations->exists( g | g.child = child and
           g.parent = parent ) and
   aeParent.association = asso and
   aeParent.aggregation = AggregationKind::composite and
   aeParent.multiplicity.range.lower = 1 and
   aeParent.multiplicity.range.upper = 1 and
   aeParent.participant = parent and
   aeChild.association = asso and
   aeChild.aggregation = AggregationKind::none and
   aeChild.multiplicity.range.lower = 0 and
   aeChild.multiplicity.range.upper = 1 and
   aeChil.participant = entity
```

Der letzte Teil der Transformation besagt, dass das Mapping nur in eine Richtung erfolgen kann. Dies liegt daran, dass bereits vor einer Transformation Assoziationen vorhanden sein können, die dem selben Erscheinungsbild entsprechen. Somit fehlt die Information für die Rück-Transformation. In `mapping` wird unterstellt, dass es eine Transformation gibt, die einen Klassen-Namen in einen Rollen-Namen eines Endpunktes umwandelt.

```
   unidirectional;
   mapping;
      child.name  <~> aeChild.role;
      parent.name <~> aeParent.role;
}
```

Eine weitere Transformation soll noch aufgezeigt werden. Die Aufgabe von MapToVarchar ist es, alle Zeichenketten aus dem Modell in den Datenbank-konformen Datentype `VARCHAR` zu transformieren. Der `source` - Type ist dazu entweder vom Typ `String` oder `chars`. Durch die Transformation entsteht ein neuer Datentyp mit dem Namen `VARCHAR(100)`.

```
Transformation MapToVarchar(UML,SQL) {
   params
      length : int = 100;
      stringTypes : Bag = {"String", "chars" }
```

```
        varcharTypeName : String = "VARCHAR"
    source
        sourceType : UML::DataType;
    target
        targetType : SQL::DataType;
    source condition
        stringTypes->exists( t | t.name = sourceType.name )
    target condition
        targetType.name = varcharTypeName.concat("(").concat( length ).concat(")")
    unidirectional;
    mappings;
}
```

Diese formale Transformation hat auch Nachteile. Es muss die Frage geklärt werden, in welcher Reihenfolge die Zielbedingungen abzuarbeiten sind. So entfernt die Transformation *derivedClassToTable* zuerst die Generalisierung zwischen zwei Klassen. Offen ist, ob damit die Generalisierung zu jedem Zeitpunkt im `target condition` Bereich entfernt ist oder ob in einer weiter links oder rechts vorkommenden Bedingung noch auf diese zugegriffen werden kann.

Ein weiteres Problem betrifft die Reihenfolge, in der die Transformationen gestartet werden. Die Transformation *MapToVarchar* wandelt sämtliche `String`s in `VARCHAR`s um und gibt diesem die Länge 100. Sollen spezielle Attribute eine andere Länge erhalten, so kann dies in einer weiteren Transformation angegeben werden. Die `source condition` grenzt hierzu die gewünschten Attribute ein. Nun aber muss ein Compiler entscheiden, welche der Transformationen zuerst auszuführen ist, da beide erfüllt sind.

11.3 Zusammenfassung

Dieses Kapitel zeigte zwei Möglichkeiten, wie Transformationen aufgespalten werden können in Tools und Regeln. Dazu wurde die Velocity-Template-Engine in Verbindung mit AndroMDA und der Object Constraint Language betrachtet. Velocity/AndroMDA eignet sich besonders für die PSM-nach-Code – Transformationen. Dabei bietet AndroMDA der Template-Engine primär ein Objekt vom Typ `UmlPackage` aus dem JMI-Framework und eine Helper-Klasse. Somit ist der Zugriff auf die Daten weitestgehend wie der per Java. Allerdings lassen sich Cartridges und Velocity-Macros entwickeln, die den Prozess vereinfachen können. Die Object Constraint Language ließe sich zu einer Transformationssprache erweitern. Hierzu definiert sie unter anderem die Quell- und Ziel-Objekte und nennt Bedingungen, wann die Transformationen ausgeführt werden sollen. Der interessante Bereich steckt in den Ziel-Bedingungen. Hier wird per OCL definiert, welches Ergebnis nach der Transformation vorliegen soll.

Kapitel 12

Zusammenfassung und Ausblick

12.1 Model Driven Architecture

Der für diese Arbeit entwickelte Prototyp konnte zeigen, welche Vorteile die Model Driven Architecture der Software-Entwicklung bringt. Dabei wurden einige Aspekte erwähnt, wie z.B. die leichte Wartbarkeit der Anwendung. So spielt die Anzahl der Klassen für die Transformationen keine Rolle. Damit könnte das PIM von der OMG größere Modifikationen erfahren, ohne dass die Transformationen neu zu schreiben wären. Es könnten lediglich geringe Anpassungen notwendig werden, etwa um die Kompositionen anders aufzulösen.

Als sehr interessant hat sich die MDA-Methodik dafür herausgestellt, Programm-Code für verschiedene Plattformen zu erzeugen. Zwar traten für den Prototypen nur verschiedene Kommunikations-Protokolle als Plattformen in Erscheinung, aber hier hat es sich gezeigt, wie leicht RMI etwa durch Web Services ausgetauscht werden kann.

Dabei lassen sich Spezialgebiete fast ohne Berühungspunkte in Transformationen kapseln. So kann ein CORBA-Spezialist den verteilten Zugriff ermöglichen, während ein Datenbank-Experte die Transformation für das SQL-Schema bearbeitet.

MDA ist aber nicht die Lösung aller Probleme aus dem Software-Engineering. So steckt die automatisierte Software-Generierung noch in den Kinderschuhen, was an den fehlenden Tools erkennbar ist. Für diese Arbeit kam nur ein CASE-Tool in Frage, nämlich Poseidon von gentleware[1]. Dieses Tool unterstützt bereits ansatzweise die UML-2.0 Philosophie. Es unterstützt unter anderem Tagged-Values, die Vergabe mehrerer Stereotypen für ein Modell-Element und ermöglicht die Benutzung von Constraint-Sprachen, darunter die Object Constraint Language. Der wichtigste Grund ist aber, dass die Poseidon-Dateien problemlos von mdr[2] verarbeitet werden können. Zudem kann auch AndroMDA[3] mit diesen Dateien umgehen.

[1] `http://www.gentleware.com/`, besucht am 21.04.2004.
[2] `http://mdr.netbeans.org/`, besucht am 20.04.2004. Siehe auch 3.2 auf Seite 21.
[3] `http://www.andromda.org/`, besucht am 20.04.2004. Siehe auch Abschnitt 2.2.2 auf Seite 19 und 11.1 auf Seite 79.

Auch die Zahl der verfügbaren Transformationen muss zunehmen. Zwar gibt es einige PSM-nach-EJB – Transformationen, wie AndroMDA, jedoch fehlen wiederverwendbare PIM-nach-PSM – Transformationen. Hier muss der Transformations-Architekt darauf achten, die Transformationen auf eine hohe Wiederverwendbarkeit zu trimmen. Dies kann schon durch einfache Maßnahmen erzielt werden, wie diese Arbeit gezeigt hat (siehe z.B. Abschnitt 8.2.2 auf Seite 51). Dabei stellte sich heraus, dass die Transformationen nur eine bestimmte Aufgabe erfüllen sollen und dabei ein fest definiertes PSM liefern müssen.

Erschwerend kommt hinzu, dass Transformationen zumeist fest in einer Sprache programmiert sind, etwa Java oder eine Tool-spezifische Skriptsprache. Hier bedarf es einer formalen, fest definierten *Transformation Definition Language*. Der Abschnitt 11.2 auf Seite 81 führte in eine solche Sprache ein. In [KWB03] befinden sich weitere Ausführungen zu dieser Sprache.

12.2 MDA in CASE-Tools

Als kurzfriste Lösung empfiehlt es sich, die Model Driven Architecture in CASE-Tools einzubauen. Grundlegende Ansätze sind bereits in zahlreichen Tools vorhanden. So generieren Tools wie Rational Rose, Poseidon oder Together zum gezeichneten Modell die Klassen mit den Attribut- und Methoden-Signaturen. Auch versuchen diese Programme, Modell-Änderungen im Code nach zu ziehen. Auch der umgekehrte Weg wird teilweise unterstützt.

Als Erweiterung hierzu sind etwa Patterns beziehungsweise Templates möglich, welche die Code-Generation beeinflussen. Es ließe sich etwa angeben, dass alle Setter-Methoden einen zusätzlichen Programm-Code generiert bekommen, um die Zugriffe zu protokollieren. CASE-Tools, die solche Patterns erlauben, fallen in die Kategorie der anpassbaren Transformationen, wie im Abschnitt 2.2.2 auf Seite 18 beschrieben. Die Patterns entsprechen somit den Transformations-Regeln. Rational Rose XDE von IBM/Rational unterstützen bereits rudimentäre Patterns.

Neben den Getter- und Setter-Methoden sind weitere Operationen denkbar. Das Kapitel 6 auf Seite 31 beschreibt hierzu vorverarbeitende Modell-Transformationen. Hierüber ließe sich beispielsweise die Mehrsprachigkeit einzelner Attribute leichter modellieren. Der mit PLM Service verknüpfte ISO STEP Standard enthält zahlreiche solcher Attribute. Somit bleibt es dem Designer erspart, jede Klasse entsprechend zu modellieren. Stattdessen vergibt er nur einen entsprechenden Stereotyp.

Aber auch die angesprochene Verteiltheit lässt sich vergleichsweise einfach in ein CASE-Tool einbauen. So müsste der Designer die Klassen bzw. die Komponenten kennzeichnen, die als Remote-Service ansprechbar sein sollen. Anschließend erstellt das CASE-Tool ein entsprechendes Klassengerüst. Dies kann bereits heute mit AndroMDA realisiert werden — auch wenn es kein CASE-Tool ist. AndroMDA würde dazu eine Cartridge erhalten, welche die Transformations-Regel für die Verteilung enthält. Das erzeugte Klassengerüst wäre noch auf konventionelle Weise mit Code zu füllen.

12.3 Dynamische Aspekte von MDA

Bisher wurden für den MDA-Ansatz nur statische UML-Elemente ausgewertet. Dies ist bedingt durch den PLM Service Standard, der stark datenorientiert ist und zahlreiche Klassen beschreibt. Nun soll

ein Ausblick für dynamische Aspekte gegeben werden. Als Beispiel dient ein im Abschnitt 8.2.1 auf Seite 49 angedachter Transaktions-Mechanismus.

Dabei arbeiten alle Clients, die den gleichen Datensatz bearbeiten, weiterhin auf einem Objekt. Jedoch können die Clients ein privates Objekt anfordern, indem sie die Methode `beginTransaction()` aufrufen. Für das Objekt vom Typ `Item_instance` muss dazu eine neue Klasse mit Namen `Item_instanceAtom` angelegt werden. Diese erweitert `Item_instance` für Transaktionen. Dazu muss nach `beginTransaction()` das Haupt-Objekt per `setItem_instance()` übergeben werden. Das neue Atom-Objekt speichert diese Refererenz und initialisiert sich mit den Werten von `Item_instance`. Abbildung 12.1 zeigt das Sequenz-Diagramm dieses Vorgehens.

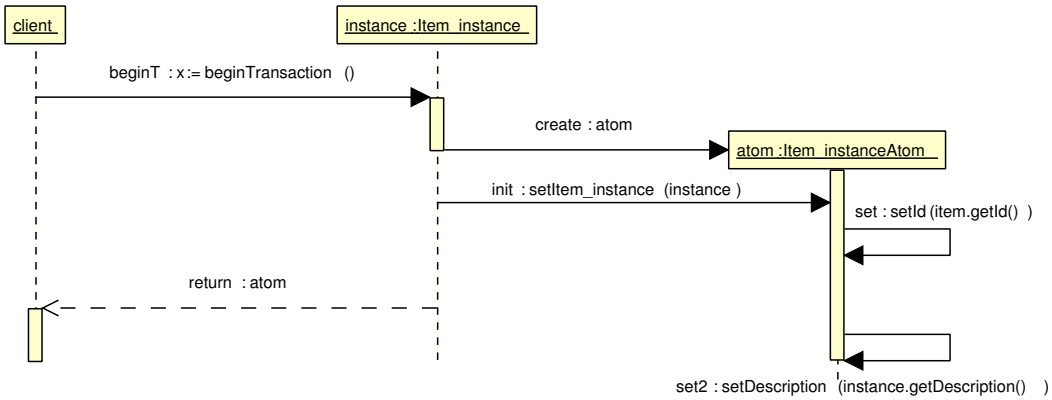


Abbildung 12.1: Sequenz-Diagram für `beginTransaction()`

Eine entsprechende Transformation könnte aus dem Sequenz-Diagramm einen Programm-Code erzeugen, der dem Modell entspricht. Die Transformation kann hiermit erkennen, was beim Aufruf von `beginTransaction()` zu erfolgen hat. Dazu ist ein neues Objekt vom Typ `Item_instanceAtom` zu erzeugen, wobei sich die Referenz auf das Objekt im Attribut `atom` befindet. Anschließend erfolgt der Aufruf der Methode `setItem_instance()` mit der `Item_instance` (this) als Parameter. Das Atom-Objekt initialisiert sich über die entsprechenden Methoden. Somit wäre die Transformation in der Lage, auch die Körper zu den Methoden zu erzeugen, aber nur in einem gewissen Umfang. Die gezeigte Methode `setItem_instance()` initialisiert zwar sowohl `id` als auch `description`, es sollte dennoch auch die Referenz auf `Item_instance` gespeichert sein. Dies lässt sich im Sequenz nicht angeben.

Neben dem Sequenz-Diagramm können auch aus dem Aktivitätsdiagramm Informationen gewonnen werden. Die Abbildung 12.2 auf der nächsten Seite zeigt die zwei Zustände *unchanged* und *changed*. Zu Beginn einer Transformation befindet sich das private Objekt vom Typ `Intem_instanceAtom` im Zustand unchanged, da keine Werte verändert wurden. Dies ändert sich, sobald eine Setter-Methode aufgerufen wird, sprich `setDescription()` oder `setId()`. Über die Methode `commitTransaction()` oder `abortTransaction()` wird das Objekt terminiert, bei `commitTransaction()` wird je nach Zustand das eigentliche Objekt aktualisiert.

Das dargestellte Aktivitätsdiagramm ist nur bedingt nutzbar, da den beiden Zuständen keine Aktionen zugeordnet sind. Beide müssen folglich noch mit einem Attribut verknüpft werden. Eine Variante

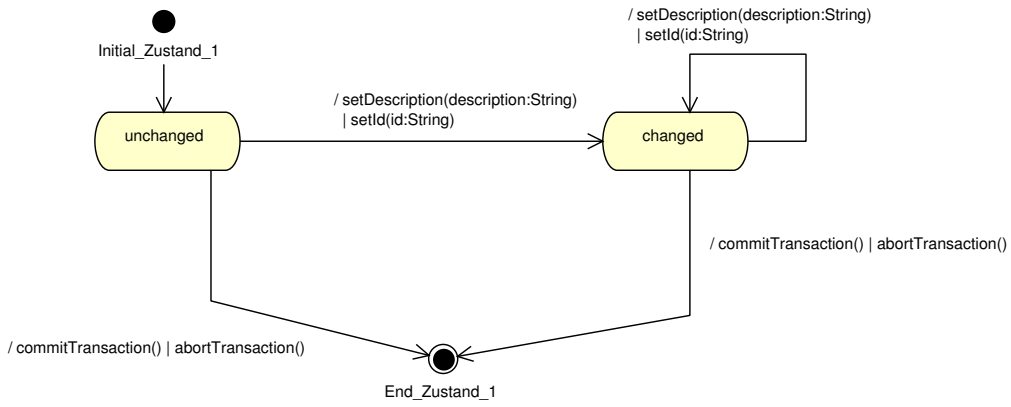


Abbildung 12.2: Aktivitätsdiagramm für *unchanged* nach *changed*

ist, die Zustände in `update=false` und `update=true` umzubenennen. Dabei ist `update` ein Attribute in `Item_instanceAtom` und vom Typ Boolean. Alternativ dazu könnten auch OCL-Constraints angegeben werden, etwa

```
context Item_instanceAtom
   inv : oclInState(unchanged) implies self.update=false and
         oclInState(changed) implies self.update=true
```

Damit könnte ein zusätzlicher OCL-Compiler die Werte von `update` entsprechend setzen. Alternativ zum Aktivitätsdiagramm ließe sich dieses Vorgehen auch komplett in OCL beschreiben. Dazu müsste jede Setter-Methode als `post`-Bedingung implizieren, dass `update` auf true steht.

Anhang

Anhang A

Starten des beiligenden Programmes

Der für diese Arbeit erstellte Prototyp befindet sich auf der beiliegenden CD-ROM. Dieser Anhang zeigt, wie der Prototyp zu generieren ist.

Am schnellsten geschieht dies, wenn im Verzeichnis `Programm` das ant-Build-Tool[1] aufgerufen wird. In diesem Verzeichnis befindet sich die Datei `build.xml`, welche die eigentlichen Build-Dateien im Unterverzeichnis `Transformationen` und `PLM-Service-Prototyp` aufruft. Das Resultat befindet sich anschließend im Verzeichnis `Programm/PLM-Service-Prototyp/build`. Darin befinden sich sowohl die generierten Java-Source-Code – Klassen als auch das SQL-Skript. Dieses Verzeichnis hat außerdem das Unterverzeichnis `build`, in dem sich die übersetzten Class-Dateien der generierten Anwendung befinden.

Somit enthält das Verzeichnis `Programm/PLM-Service-Prototyp/build/build` die übersetzten Java-Class-Dateien des lauffähigen Prototypen. Diese können nun etwa per jar als Bibliothek bereit gestellt werden. Mit diesem jar-Archiv könnte dann der Client-Entwickler arbeiten.

Alternativ können auch alle dort befindliche Klassen in einen Applikationsserver kopiert werden, etwa jakarta-tomcat[2]. Als Start-Servlet kann das List-Servlet der Klasse Item verwendet werden. Der Servlet-Name ist `org.schreinert.masterthesis.step.item.servlets.ItemListDialog`. Als Voraussetzung für den Prototypen muss eine MySQL-Datenbank mit Namen plms vorhanden sein. Außerdem wird der JDBC-Treiber[3] für MySQL benötigt. Als Benutzer dient Standard-mäßig root mit einem leeren Kennwort. Die Werte lassen sich durch Umgebungsvariablen ändern:

- `sql.connection.driver`
  Der Namen des JDBC-Treibers, z.B. `com.mysql.jdbc.Driver`.
- `sql.connection.username`
  Der Name des Benutzers um sich mit der Datenbank zu verbinden.

[1] `http://ant.apache.org/`, besucht am 24.04.2004.
[2] `http://jakarta.apache.org/tomcat/`, besucht am 24.04.2004.
[3] Z.B. `http://dev.mysql.com/downloads/connector/j/3.0.html`, besucht am 25.04.2004.

- `sql.connection.password`
  Das Kennwort passend zum obigen Benutzer, um sich mit der Datenbank zu verbinden.

- `sql.connection.url`
  Die URL zum Datenbankmanagementsystem und zur Datenbank. Für die URL sollte ein JDBC-Treiber vorhanden sein (siehe `sql.connection.driver`). Für die MySQL-Datenbank mit Namen plms ist die URL `jdbc:mysql://localhost/plms`.

Anbei ist ein erweitertes Programm aus dem Beispiel 6 auf Seite 53 abgebildet, mit dem direkt auf die Service-Schicht zugegriffen werden kann. Das Programm liegt im `PLM-Service-Prototyp` - Verzeichnis auf der CD als `test.java`.

```
import java.util.*;
import org.schreinert.masterthesis.step.item.*;
import org.schreinert.masterthesis.step.item.clients.*;
public class test {
  public static void main(String[] args) {

        System.setProperty("sql.connection.driver", "com.mysql.jdbc.Driver");
        System.setProperty("sql.connection.username", "root");
        System.setProperty("sql.connection.password", "");
        System.setProperty("sql.connection.url", "jdbc:mysql://localhost/plms");

        //
        // Create Item
        //
        ItemFactoryClient factory = ItemFactoryClient.getInstance();
        factory.create("0816", "Test-Item", "Dies ist ein Test");
        ItemPrimaryKey pk = new ItemPrimaryKey("0816");
        Item item = factory.findByPrimaryKey(pk);
        System.out.println("Item.id = "+item.getId()+" name="+item.getName());

        //
        // Create Item_version
        //
        Item_versionFactoryClient.getInstance().create( "1", item );
        Item_version version = Item_versionFactoryClient.getInstance()
                              .findByPrimaryKey(
                 new Item_versionPrimaryKey("1", item) );

        System.out.println("Version: "+version.getId()+" item="+
                           version.getAssociated_item().getId());

        //
        // Create Design_discipline_item_definition
        //
        Design_discipline_item_definitionFactoryClient.getInstance()
                 .create( "production", version );
        Design_discipline_item_definitionFactoryClient.getInstance()
                 .create( "design", version );
```

```
Design_discipline_item_definitionFactoryClient.getInstance()
        .create( "construction", version );
Design_discipline_item_definitionFactoryClient.getInstance()
        .create( "maintainance", version );
Design_discipline_item_definition ddid =
        Design_discipline_item_definitionFactoryClient.getInstance()
        .findByPrimaryKey(
        new Design_discipline_item_definitionPrimaryKey
                ( "production", version ));
System.out.println("DDID: "+ddid.getId()+"  version="
        +ddid.getAssociated_item_version().getId());

//
// List all Design_discipline_item_definition
//
System.err.println("All DDIDS:");
java.util.Collection list =
        Design_discipline_item_definitionFactoryClient.getInstance()
        .list();
for ( Iterator it = list.iterator() ; it.hasNext() ; ) {
   Object obj = it.next();
   Design_discipline_item_definitionPrimaryKey dPK =                                    (
   Design_discipline_item_definition d =
        Design_discipline_item_definitionFactoryClient.getInstance()

   System.out.println("  - id="+d.getId()
        +" item_version="+d.getAssociated_item_version().getId()
        +" item="+d.getAssociated_item_version().getAssociated_item()
        .getId()
        +" name="+d.getAssociated_item_version().getAssociated_item()
        .getName()
        +" descr="+d.getAssociated_item_version().getAssociated_item()
        .getDescription());
}

//
// Create Item_instance
//
Item_instanceFactoryClient.getInstance()
        .create("Instance 01", "Beschreibung", ddid);
Item_instance instance = Item_instanceFactoryClient.getInstance()
        .findByPrimaryKey(
        new Item_instancePrimaryKey("Instance 01", ddid )
);
   System.out.println("Instance id="+instance.getId()+" descr="+
        +" item_version="+instance.getDefinition()
        .getAssociated_item_version().getId()
        +" item="+instance.getDefinition()
        .getAssociated_item_version().getAssociated_item().getId()
        +" name="+instance.getDefinition()
        .getAssociated_item_version().getAssociated_item().getName()
        +" descr="+instance.getDefinition()
        .getAssociated_item_version().getAssociated_item().getDescription());
```

```
        //
        // List Item_instance
        //
        list = Item_instanceFactoryClient.getInstance().list();
        for ( Iterator it = list.iterator() ; it.hasNext() ; ) {
           Object obj = it.next();
           Item_instancePrimaryKey dPK = (Item_instancePrimaryKey)obj;
           Item_instance d = Item_instanceFactoryClient.getInstance()

           System.out.println("  - id="+d.getId()+" descr="+d.getDescription()
                +" item_version="+d.getDefinition().getAssociated_item_version()
                .getId()
                +" item="+d.getDefinition().getAssociated_item_version()
                .getAssociated_item().getId()
                +" name="+d.getDefinition().getAssociated_item_version()
                .getAssociated_item().getName()
                +" descr="+d.getDefinition().getAssociated_item_version()
                .getAssociated_item().getDescription());
        }
    }

}
```

Literaturverzeichnis

[BH03] Richard Bubel and Reiner Hähnle. Integration of informal and formal development of object-oriented safety-critica. *Electronic Notes in Theoretical Computer Science 80 (2003)*, 2003. URL `http://www.elsevier.nl/locate/entcs/volume80.html`.

[Boy03] Gerry Boyd. Executable uml: Diagrams for the future. 5. Februrary 2003. URL `http://www.devx.com/uml/Article/10717`, besucht am 21.04.2004.

[DP02] Stefan Denninger and Ingo Peters. *Enterprise JavaBeans 2.0*. Addison-Wesley, München, 2nd edition, 2002.

[Dub03] Carolyn K. Dub. Accelerating embedded software development with a model driven architecture. September 2003. URL `http://www.omg.org/mda/mda_files/MDA_overview.pdf`, besucht am 22.04.2004.

[FHH+03] Michael Feltes, Peter Habel, Ullrich Haug, Lutz Lämmer, Steffen Nowacki, and Günter Staub. *Product Lifecylce Management Service – Initial Submission*. DaimlerChrysler AG, PROSTEP AG, ZGDV, PDTec GmbH, 01. Januar 2003.

[FHLT03] Michael Feltes, Ullrich Haug, Lutz Lämmer, and Carsten Theis. *Product Lifecylce Management Service – Initial Submission*. DaimlerChrysler AG and PROSTEP AG, 12. Dezember 2003.

[Fow03] Martin Fowler. *UML Distilled*. Addison-Wesley Professional, Boston, 3rd edition, 2003.

[GHJV01] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster — Elemente wiederverwendbarer objektorientierter Software*. 2001.

[GT00] Volker Gruhn and Andreas Thiel. *Componentenmodelle — DCOM, JavaBeans, Enterprise JavaBeans, CORBA*. Addison Wesley, München, 2000.

[Hab03] Peter Habel. *Derivation of the PDTnet Schema from AP214*. PROSTEP AG, Darmstadt, Januar 2003.

[Har01] Paul Harmon. Building advanced applications — uml models e-business. *Software Magazine*, April/May 2001. URL `http://www.softwaremag.com/archive/2001apr/umlmodelsebiz.html`, besucht am 22.04.2004.

[HR98] Philip Heller and Simon Roberts. *Java 1.2 Developer's Handbook*. Sybex International, San Francisco, 1998.

[ISO01] ISO, Geneva. *Industrial automation systems and integration — Product data representation and exchange — Part 214: Application protocol: Core data for automotive mechanical design processes*, 2001.

[JRH+03] Mario Jeckle, Chris Rupp, Jürgen Hahn, Barbara Zengler, and Stefan Queins. *UML 2 glasklar*. Hanser Fachbuchverlag, München/Wien, 2003. URL `http://www.uml-glasklar.de`.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Addison Wesley Professional, Boston, 2003.

[Mic03] Sun Microsystems. Enterprise javabeans technology. Homepage, 2003. URL `http://java.sun.com/products/ejb/`, besucht am 20.04.2004.

[MM] Joaquin Miller and Jishnu Mukerji. Mda guide version 1.0. URL `http://www.omg.org/docs/omg/03-06-01.pdf`, besucht amt 03.08.2004.

[OMG00] OMG. *Product Data Management Enablers Specification*, Version 1.3, November 2000.

[OMG02] OMG. *OMG XML Metadata Interchange (XMI) Specification v1.2*, Januar 2002. URL `http://www.omg.org/cgi-bin/doc?formal/2002-01-01`.

[OMG03a] OMG. *Object Constraint Language Specification*, March 2003.

[OMG03b] OMG. *OMG XML Metadata Interchange (XMI) Specification*, May 2003. URL `http://www.omg.org/cgi-bin/doc?formal/2003-05-02`.

[Par98] Helmuth Partsch. *Requirements-Engineering systematisch. Modellbildung für softwaregestützte Systeme*. Springer Verlag, Heidelberg, 1998.

[Sta99] David Starzyk. *STEP and OMG Product Data Management Specifications — A Guide for Decision Makers*. OMG, September 1999.

[Ste01] Beth Stearns. Enterprise javabean 2.0 specification changes. 2001. URL `http://java.sun.com/developer/technicalArticles/ebeans/ejb20/`, besucht am 20.04.2004.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language - Getting Your Models Ready for MDA*. Addison Wesley Professional, Boston, 2nd edition, 2003.

Erklärung

Ich erkläre, dass ich die Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, 26.04.2004, _____
(Ort, Datum, Unterschrift)