Westsächsische Hochschule Zwickau, University of Applied Sciences

Bachelorarbeit

Überarbeiten des Typemappings vom Genesez Projekt

Huster, Peter

geboren am 22. Mai 1984 in Zwickau

Studiengang Informatik - Systeminformatik

Westsächsische Hochschule Zwickau
Fachbereich Physikalische Technik / Informatik
Fachgruppe Informatik

Betreuer: Prof. Dr. rer. nat. habil. W. Golubski
Einrichtung: Westsächsische Hochschule Zwickau
Abgabetermin: 04. Februar 2009

Huster, Peter
Matrikelnummer: 052079/34
Kreisigstrasse 35
08056 Zwickau

Autorenreferat

Die vorliegende Bachelorthesis analysiert den Sachverhalt des Typemappings in der modellgetriebenen Softwareentwicklung näher. Das Typemapping befasst sich hierbei mit dem Abbilden von UML-Typen auf programmiersprachenspezifische Datentypen. In dem Transformationsprozess, welcher als Resultat den generierten Quelltext aufweist, wird bei der Modell-zu-Quelltext-Transformation das Typemapping aufgerufen. Dieser Mechanismus ist bereits im vorliegenden GeneSEZ Projekt implementiert, genügt aber den aktuellen Anforderungen nicht mehr. Das Typemapping basiert auf XML-Dateien, welche verarbeitet werden. Ziel ist es, weitestgehend die Struktur zu überarbeiten und unter Verwendung einer besseren XML-Verarbeitungs-API, die Mechanismen, wie das Einbinden beziehungsweise referenzieren weiterer Typemapping Dateien um einen Multi-Include Mechanismus zu erweitern und kleinere Änderungen am Verhalten der bis dato eingesetzten Lösung vorzunehmen. Desweiteren ist es notwendig, eine Validierung der XML-Dateien zu implementieren, mit der Folge dass ebenfalls eine dagegen zu validierende Definitionsdatei entwickelt werden muss. Die Verwendung aktueller Technologien und Entwicklungsprozesse ist weitestgehend Bestandteil, um eine zukunftssichere, sowie verständliche und leicht erweiterbare Lösung zur Verfügung zu stellen. Mögliche Anforderungen, welche nach Abschluss der Thesis anfallen, können so zeitnah und mit geringem Aufwand umgesetzt werden.

Inhalt

1 Einleitung

1.1 Einordnung der Arbeit

Folgend wird auf das Einsatzgebiet und die Verwendung der hier abgehandelten Thesis zur Sensibilisierung bezüglich des Themas eingegangen. Ziel und Aufgabenstellung werden in den folgenden Kapiteln in der chronologischen Relevanz näher betrachtet. Die hier beschriebene Vorgehensweise stimmt aufgrund des evolutionären Charakters von Programmierung und Recherche von Technologien nicht exakt überein.

1.2 Modellgetriebene Softwareentwicklung

Die Modelgetriebene Softwareentwicklung (MDSD) ist eine junge Disziplin im Bereich der Softwareentwicklung, aufbauend auf dem Prinzip dass die Modelle von Software maßgeblich an der Qualität der resultierenden Software beteiligt ist und deswegen auch in den Entwicklungsprozess integriert wird. MDSD legt großen Wert auf die exakte Terminologie, um die verschiedenen Konzepte auseinanderzuhalten und grenzt sich von Computer Aided Software Engineering (CASE) oder Sprachen der vierten Generetion (4GL) ab, indem es einen domänenspezifischen Ansatz verfolgt und bei diesem aber die Modellierungssprache, Transformation und Plattform frei wählbar sind. Mit der Einführung des Notationsstandard Unified Modelling Language (UML) allein konnte zwar die graphischen Notationen der objektorientierten Systeme vereinheitlicht werden, aber erst mit Einzug der MDSD ist es möglich geworden die Modelle zum Teil der Software zu machen und somit zur Steigerung der Entwicklungsgeschwindigkeit, Softwarequalität, Wiederverwendung und Portabilität sowie zur Vermeidung von Redundanzen beizutragen. Modellgetriebene Software wird anhand von Modellen beschrieben. Um selbige zu beschreiben ist eine Sprache notwendig, da es sich um formale Modelle handelt, die einem Metamodell genügen. Weiterführend sind grundlegende Begriffe definiert.
"Ein Modell ist eine abstrakte Repräsentation von Struktur, Funktion oder Verhalten eines Systems."[1] In der MDSD legt man sich zuerst auf ein Metamodell fest. Hier gibt es die Wahl zwischen Domänenspezifische Sprache (DSL), welche in der Regel von Domänenspezialisten, aber auch von den Entwicklern kreiert werden, mit Bezug auf die Domäne, oder alternativ der

[1] [SV05, S. 20]

UML. Der Vorteil der DSL liegt in der Nähe zur Domäne und damit in ihrer Verständlichkeit gegenüber dem Kunden sowie ihrer geringen Komplexität. Deshalb eignet sie sich hervorragend zur Generierung von Quellcode aus Modellen. Der gravierende Nachteil ergibt sich allerdings aus dem Konzept der Domäne, da die DSL in der MDSDmeist nicht standardisiert ist und Domänen nicht zwingend gleich sind beziehungsweise es bei ähnlichen Domänen durchaus Unterschiede in der Definition der Komponenten der DSL geben kann. Damit müssten die Entwickler bei jedem neuen Projekt möglicherweise eine weitere DSL erlernen. Hier sticht der Vorteil der UML hervor, da selbige standardisiert und somit eindeutig, aufgrund ihrer Komplexität allerdings ungeeignet zur Codegenerierung ist.

1.3 GeneSEZ

Generative Softwareentwicklung Zwickau (GeneSEZ) verfolgt hierbei einen Ansatz, der die Vorzüge beider Varianten wählt. Indem ein vereinfachtes Metamodell basierend auf UML verwendet wird, welches als Domäne die Generierung von Code als solches in Betracht zieht und somit eine Art von Middleware-Ansatz in den MDSD Prozess eingliedert. Basierend auf dem Generator Framework open Architecture Ware (oAW) können die Modell-zu-Modell bzw. die Modell-zu-Text-Transformationen durchgeführt werden. "Transformationen bilden Modelle auf die jeweils nächste Ebene - weitere Modelle oder Sourcecode ab."[2] Hier werden wiederverwendbare Templates und Skripte genutzt, um Infrastrukturcode zu generieren. Im Falle von GeneSEZ wird auf Forward Engineering gesetzt, aber implementierter Quelltext bleibt von der Neugenerierung der Infrastruktur verschont um das Modell mit dem Generat konsistent zu halten, da Designänderungen auch nur im Design geändert werden sollten. Durch die sequentiellen Transformationen, welche über den Generator Workflow gesteuert werden, wird basierend auf dem Anwendungsmodell der Quellcode generiert. Genau zwischen diesem Schritt greift das GeneSEZ Framework und überführt mittels Modelladapter das Anwendungsmodell in das GeneSEZ Metamodell. Dieses Modell wird validiert und nach erfolgreicher Validierung modifiziert. Diese Modifikationen, zusammen mit der Modell-zu-Text-Transformationen binden unter anderem Namenskonventionen, Pakete, Abbildung der Datentypen und vieles mehr an das Generat. Mit der Wahl der Plattform legt man schließlich fest, in welcher Sprache der Quelltext erzeugt wird.

[2] [SV05, S. 23]

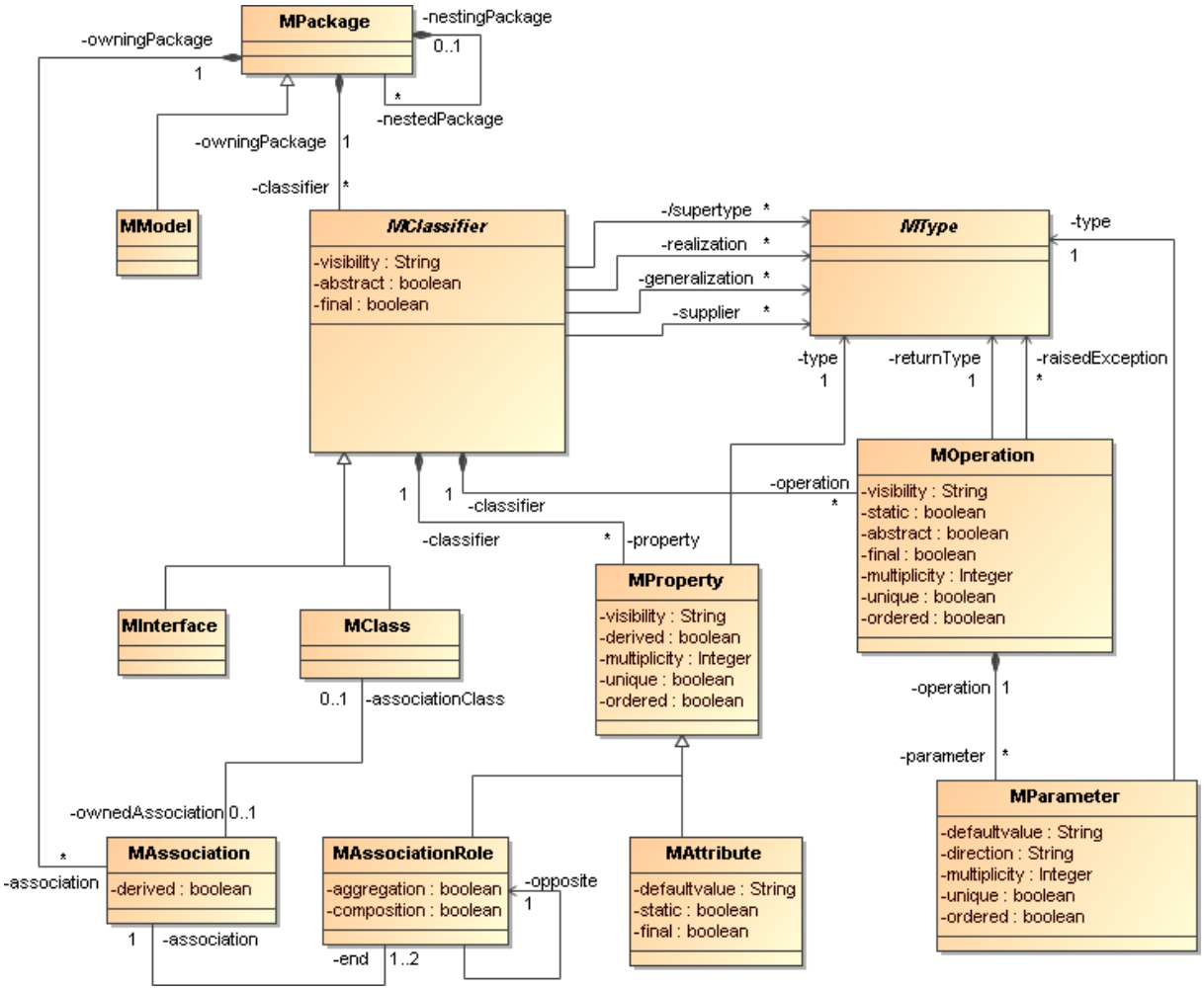

Abbildung 1.1: Metamodell des GeneSEZ Kerns - objektorientes Konzept

1.4 Typemapping

"... when a team is picking the angle of attack and choosing what languages and technology to use. These decisions about the foundations of a piece of software, which might appear at first to be lightweight and reversible, turn out to have all the gravity and consequence of poured concrete"[3]. Mit der Funktionalität des Typemappings kann dieser Sachverhalt partiell entschärft werden. Es ist für das Abbilden der UMLTypen auf die programmspezifischen Datentypen zuständig, um das Gerüst in der jeweiligen Programmiersprache zu generieren. Als Teil des Generator Workflows wird es im Abschnitt der Modell-zu-Text-Transformation angewandt.

[3] Siehe [Ros08, S. 58]

2 Der Typemapping-Mechanismus

Typemapping ist wie schon erwähnt Teil des Transformationsprozesses von Software nach der MDSD. Dabei werden im Modell verwendete Typen auf Typen einer Programmiersprache abgebildet. Aus konzeptioneller Sicht realisiert das Typemapping zum einen ein so genanntes Namensmapping, welches die Namen der Typen auf die der Zielsprache abbildet, zum anderen das Abbilden der Typen[1], welches die Modifikatoren des Typs auswertet. Das Klassendiagramm in

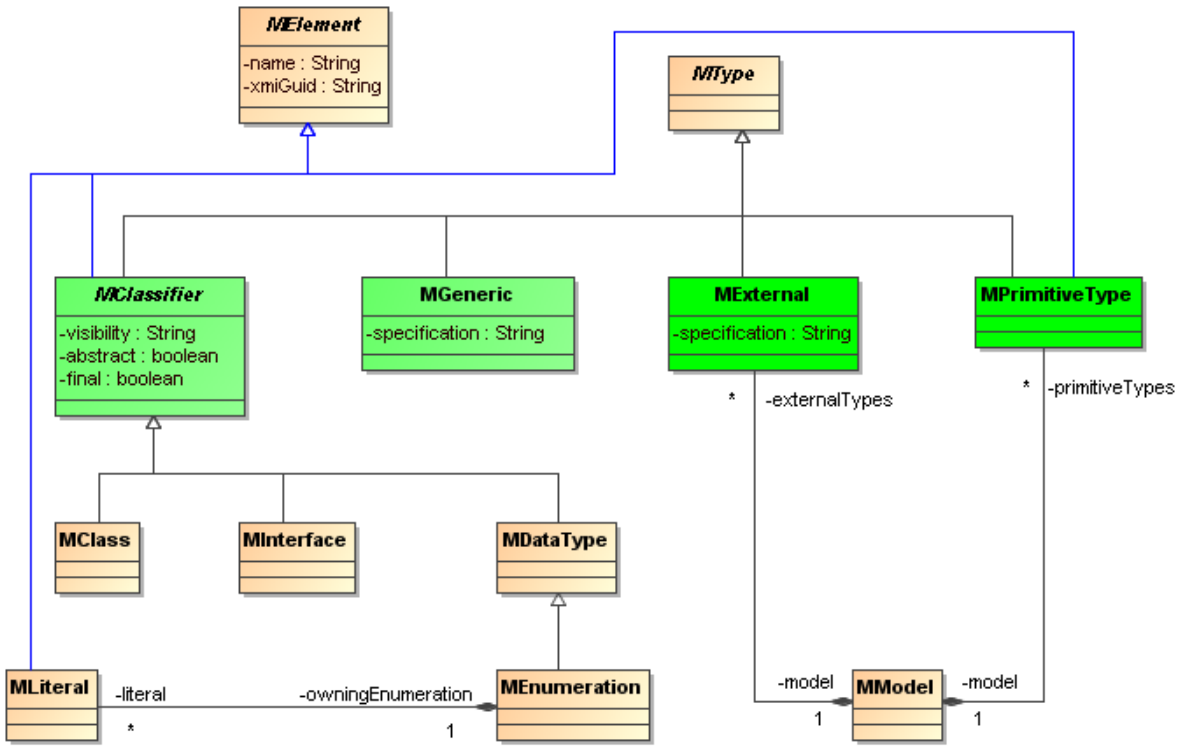

Abbildung 2.1: GeneSEZ Typsystem

Abbildung 2.1, sowie der Ausschnitt aus der Abbildung des GeneSEZ Kerns[2] veranschaulichen noch einmal die relevanten Typen. `MGeneric`, `MClassifier`, `MPrimitiveType` und `MExternal` werden im Typemapping berücksichtigt. `MGeneric` steht für einen generischen Typ zur Parametrisierung einer Klasse und wird auf seine Spezifikation gemappt. Bei `MClassifier` handelt es sich um einen selbst erstellten Typen des Modells und wird deswegen auf seinen Namen gemappt. `MPrimitiveType` bezeichnet, wie der Name schon sagt, die primitiven bzw. nativen Datentypen, die in der Regel in der Programmiersprache verankert sind, wie zum Beispiel `int`,

[1] so genanntes Typenmapping
[2] Abbildung 2.2

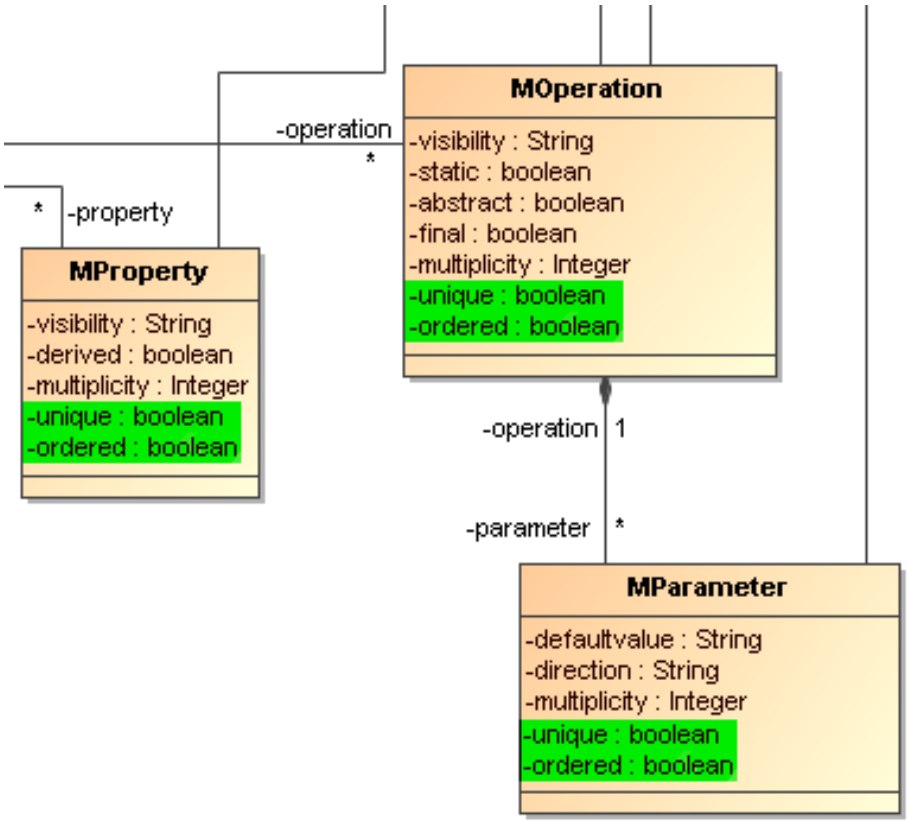

Abbildung 2.2: MultiValue Klassen

`float` oder `char` in Java. `MExternal` stellt Datentypen aus nicht schon in der Programmiersprache enthaltenen Typen dar, das heißt aus Bibliotheken, Assemblies sowie Standardbibliotheken welche mittels `import`, `include`, `using` oder ähnlichen Anweisungen erst bekannt gemacht werden müssen.

Die auszuwertenden Typmodifikatoren betreffen immer das Element des Typs, das heißt beispielweise ein Attribut, welches den Typ aufweist. So sagt der Modifikator `multiplicity` etwas über die Anzahl der zu speichernden Objekte aus. Die Abbildung 2.3 zeigt beispielhaft das Setzen der Multiplizität in UML. Ist Eins angegeben so handelt es sich um einen der oben genannten Typen, ist die Zahl allerdings größer oder ist der Stern gesetzt, so handelt es sich um einen `MultiValued`-Typ, wie zum Beispiel ein `Array` oder eine `Collection`. Bei `MultiValued` -Typen werden die Typmodifikatoren `unique` und `ordered` ausgewertet, welche jeweils einen Wahrheitswert repräsentieren. Ist `unique` auf wahr gesetzt, enthält der Typ nur eindeutige Elemente bzw. keine gleichen Elemente mehrfach, wie bei einem `Set` der Programmiersprache Java. Ist `ordered` auf wahr gesetzt, sind die Elemente in einer festgelegten Reihenfolge oder nach einem bestimmten Algorithmus sortiert, wie es bei einem `Array` üblich ist.

Des Weiteren können Kontexte der einzelnen Mappings angegeben werden. Diese Kontexte

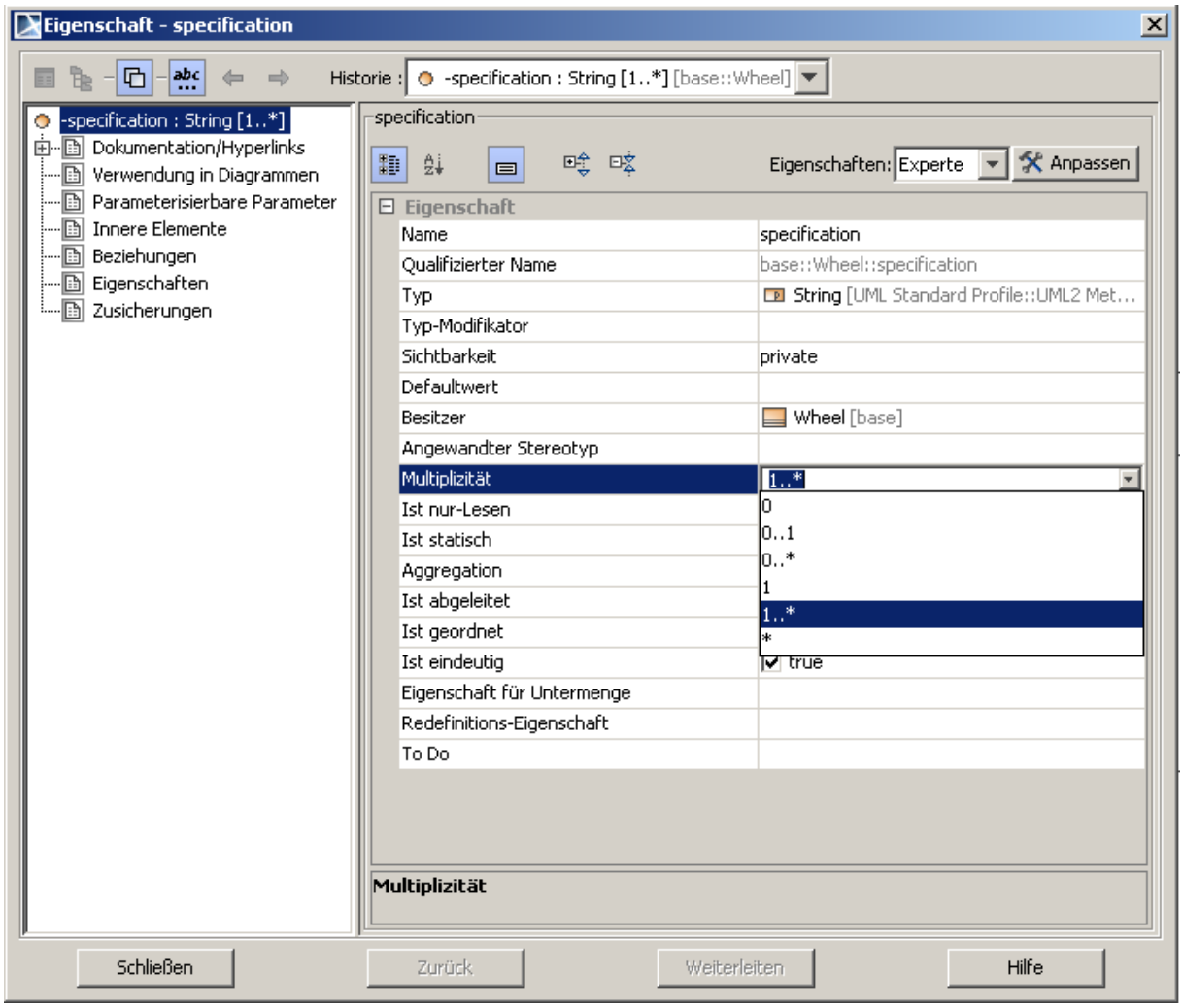

Abbildung 2.3: Wahl der Multiplizität für ein Attribut in MagicDraw UML

sind abhängig von ihrer Verwendung, wie zum Beispiel der Implementierung eines `Sets` als `java.util.HashSet` in Java. Die Mappingvorschriften werden als XML-Datei repräsentiert wird.

```
1 <typeMapping>
2  <header>
3   <superMappingFile>de/genesez/platforms/common/typemapping/
       typeMapping.xml</superMappingFile>
4  </header>
5  <collectionTypes>
6   <mappingElement unique="false" ordered="false">
7    <defaultType>java.util.List</defaultType>
8     <contextType>
```

```
9           <context>Implementation</context>
10          <type>java.util.ArrayList</type>
11      </contextType>
12   </mappingElement>
13  </collectionTypes>
14  <primitiveTypes>
15   <mappingElement>
16              <map>int</map>
17              <defaultType>int</defaultType>
18              <contextType>
19                  <context>Wrapper</context>
20                  <type>Integer</type>
21              </contextType>
22          </mappingElement>
23  </primitiveTypes>
24  <externalTypes>
25     <!--ähnlich den primitiveTypes-->
26  </externalTypes>
27 </typeMapping>
```

Listing 2.1: Ausschnitte aus der java.typemapping.xml auf der CD unter Sourcecode/former.type.mappings zu finden

Ein möglicher Aufbau ist in Listing 2.1 zu sehen. In der Typemapping-Datei können im Header mögliche mit einzubeziehende Typemapping Dateien referenziert werden. Im Tag `collectionTypes` werden die Typmodifikatoren für UML Typen mit einer `multiplicity` größer Eins angegeben. Im Tag `primitiveTypes` sind die Mappings für die nativen Typen enthalten und im Tag `externalTypes` Mappings für Typen von externen Bibliotheken. Diese Tags enthalten wiederrum einzelne Mapping Tags, welche jeweils einen Typ mit ihrem Standardmapping und ihren möglichen Kontexten repräsentieren. Die Funktionalität des Typemappings wird durch 2 Funktionen eines Xtend Skriptes realisiert, die jeweils für die Abbildung der Namen oder Typen zuständig sind. Das Ergebnis wird dann im Xpand-Template eingebunden und kann fortan verwendet werden. Die notwendigen Informationen für das Typemapping sind folglich die UMLTypen, welche über das Modell verfügbar sind, die Typemapping Datei, welche die Map-

pings von UML auf programmiersprachenspezifische Typen abbildet und die oben erwähnten Skripte bzw. Templates, um eine Modell-zu-Code Transformation durchzuführen.

3 Gegenüberstellung der gegebenen Umsetzung und der geforderten Funktionalität

Die Codegenerierungnutzte vorerst einen Typemapping Mechanismus mit teilweise eingeschränkter Funktionalität. Ferner wurden neue Anforderungen an das Typemapping gestellt, welche eine komplexe Restrukturierung des Quelltextes voraussetzten.

3.1 gegebene Umsetzung

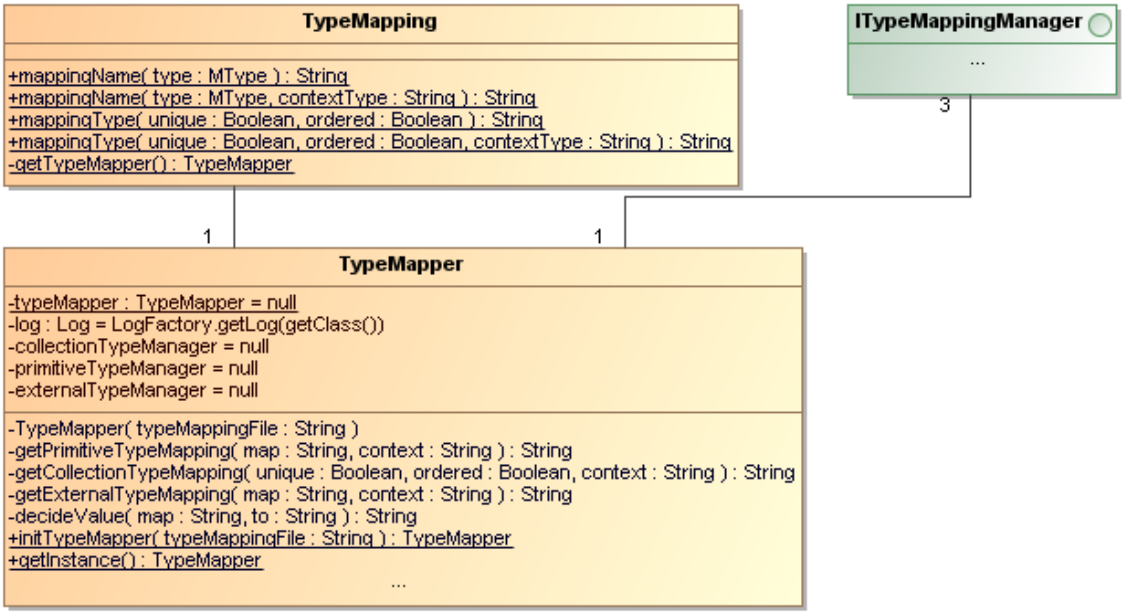

Abbildung 3.1: Fassade des Typemappings für den Transformationsprozess

Die Abbildung 3.1 und 3.2 zeigen die Struktur des Typemappings und Abbildung 3.3 das Klassendiagramm der Mappingtypen. Die Lösung setzt auf einen Singleton[1] Ansatz, der Anfragen vom Transformationsprozess umsetzt. Der Transformationsprozess übergibt der `Typemapping` Klasse den UMLTyp, respektive die Parameter für einen `MultiValued`-Typ. Dieser delegiert die Daten an den Singleton-`Typemapper` weiter. Anhand der Parameter entscheidet der `Typemapper` , an welchen `MappingManager` die Daten weitergegeben werden müssen. Sind die Informationen beim Manager angelangt, sucht selbiger in einer `Map` aus `String`- und `MappingType`-Objekten nach dem Typ. Ist der `String` enthalten wird, falls - ein Kontext mit übergeben wurde - beim

[1]Siehe [GHJV96, S. 139], von einer Klasse kann nur ein einziges Objekt instanziiert werden

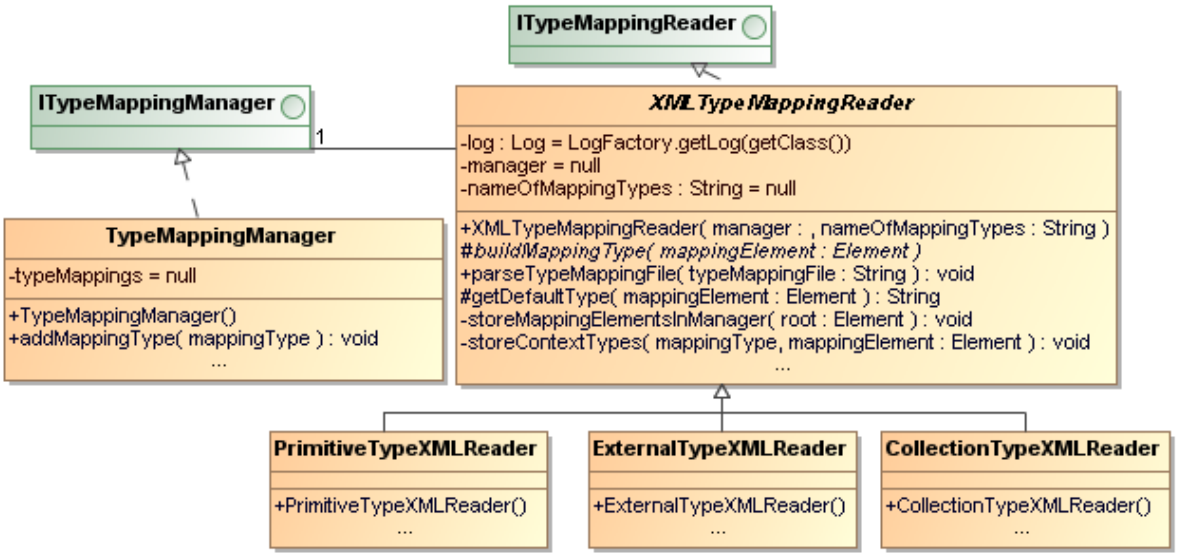

Abbildung 3.2: Klassenstruktur der vorangengenen Umsetzung des Typemappings

`MappingType` in einem weiteren assoziativen Array nach dem Kontext gesucht und zurückgegeben, ansonsten wird nur das gesuchte Mapping zur Verfügung gestellt. Beim Instanziieren des `Typemappers` wird genau eine Typemapping-Datei übergeben; wenn nicht, so wird eine Default-Typemapping-Datei verwendet. Anhand dieser Datei nutzt der `Typemapper` interne `XMLReader`-Klassen, die die jeweiligen Manager der `MappingTypes` basierend auf ihren Implementierungen füllen. Handelt es sich beispielweise um den Manager für primitive Datentypen, parst der Reader nur die Elemente innerhalb der `primitiveTypes`-Tags. Der `XMLReader` sucht, basierend auf dom4j, zuerst nach einer übergeordneten Mapping-Datei und arbeitet sich dann rekursiv durch die Struktur. Die Folge daraus ist ein erhöhter Speicherbedarf, der aufgrund der relativ kleinen XML-Dateien kaum ins Gewicht fällt. Der Speicherbedarf kommt daher zustande, da für jeden Manager einzeln die Typemapping-Datei als Baumstruktur geladen und geparst werden muss. Die Typemapping-Dateien nutzen weder Document Type Definition (DTD) noch XML Schema Definition (XSD) zur Validierung und sind beispielhaft im Anhang angefügt[2].

3.2 Anforderungskatalog

Zusammen mit dem im Trac des GeneSEZ Projektes spezifizierten Anforderungen, als auch der Mindmap, welche in Abbildung 3.4[3] zu sehen ist, kristallisierten sich folgende Direkti-

[2]Siehe Quelltextlisting unter Sourcecode/former.solution auf der CD
[3]Bereitgestellt durch Tobias Haubold, unter Documents auf der CD zu finden

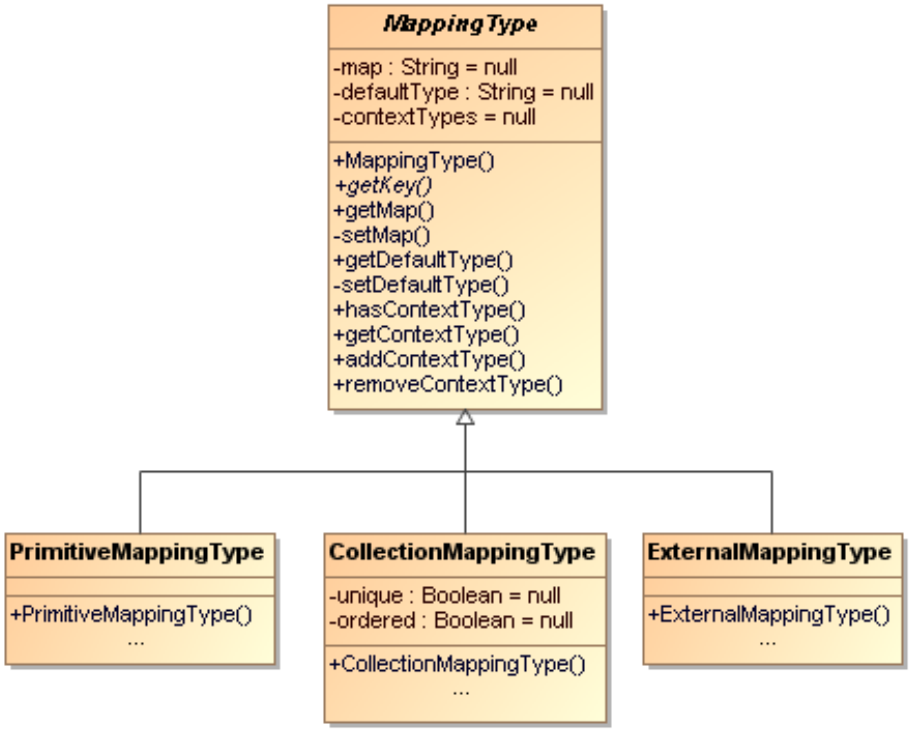

Abbildung 3.3: Repräsentation der Mapping Typen

ven heraus: Das genannte Vererbungskonzept des Angebens einer Basis-Typemapping-Datei,

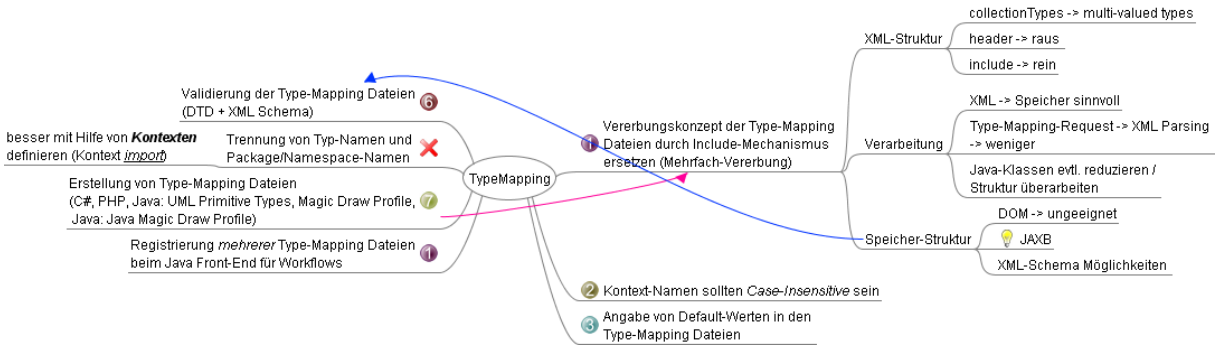

Abbildung 3.4: Mindmap zur Spezifikation der Anforderungen

welche spezialisiert wird, schränkt den Mechanismus in seiner Flexibilität ein. Es werden wie auch in objektorientieren Sprachen wie Java oder Smalltalk durch neue Definitionen weitere Mappings beschrieben, respektive andere überschreiben. Durch den Einsatz eines sogenannten Mult-Include-Mechanismus, ähnlich dem `include`-Mechanismus bekannt aus C/C++, soll es dem Entwickler möglich sein, eine Hierarchie von Typemapping-Dateien[4] mit mehreren zu

[4]Die Prioritäten der referenzierten Dateien werden durch eine Breitensuche realisiert, das heisst das die Prioritäten ebenenweise abnehmen

referenzierenden Dateien anzugeben, wie es in Abbildung 3.5 zu sehen ist. Nicht nur in den Typemapping-Dateien, sondern ebenfalls im Transformationsprozess selbst soll es möglich, sein mehrere Dateien zu referenzieren. Des Weiteren soll es möglich sein, Standardwerte für das

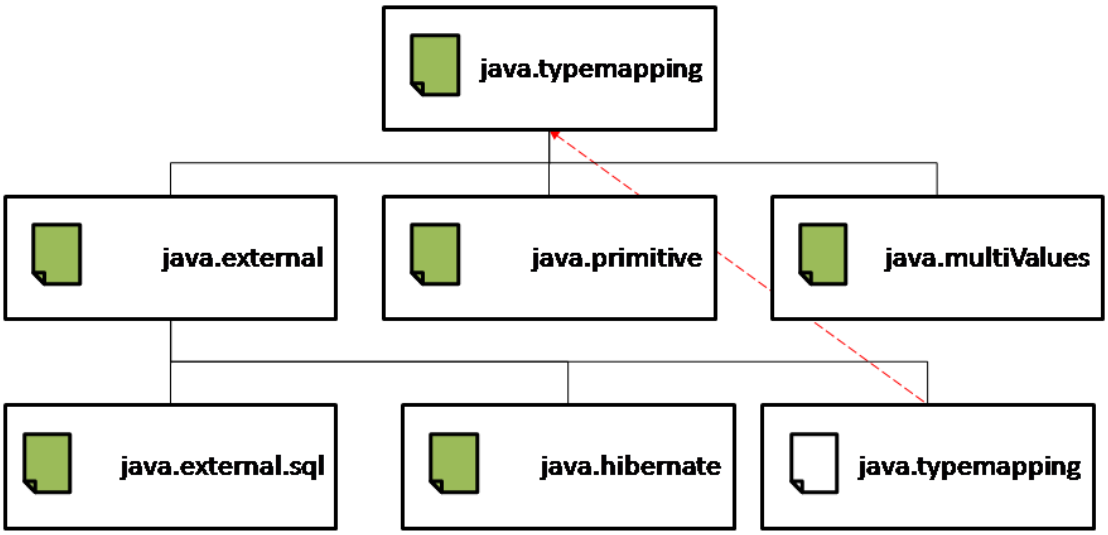

Abbildung 3.5: Multi-Include Konzept von Typemapping Dateien

Default Return-Statement in der Typemapping-Datei festzulegen, welches bislang von einem eXtend-Skript übernommen wird[5]. Es stellte sich heraus das die Trennung von Paket und Typnamen für externe Typen sinvoller durch Kontexte wiedergegeben werden kann und somit die Aufgabe nicht mehr im Anforderungskatalog enthalten ist. die Forderung nach case-insensitiven Kontexten der einzelnen Mapping Typen dagegen aufgenommen. Als letzter wichtiger Punkt ist die Validierung der Typemapping-Dateien zu nennen, das heißt zum einen die Erstellung und Einbindung einer DTD oder alternativ einer XSD zum anderen valide Typemapping-Dateien, die auch der Struktur des Schemas entsprechen. In der Mindmap ist ebenfalls schon der Ansatz zur weiteren Vorgehensweise aufgezeigt. So wird sich voraussichtlich die Struktur der XML-Dateien verändern, als auch die Java Klassen des Typemappings. Das Parsen ist weitestgehend zu minimieren, respektive vollständig zu entfernen. Auf der Mindmap ist auch der Verweis auf die Java Architecture for XML Binding (JAXB) gesetzt als Alternative zur Verarbeitung von XML-Daten. Neben dieser Möglichkeit werden folgend auch weitere Technologien analysiert, um die bestmögliche für die definierten Anforderungen auszuwählen.

[5]Diese Anforderung wurde nicht umgesetzt, auf den Grund wird in Kapitel 6 noch einmal

4 Realisierungsmöglichkeiten zur Verarbeitung von XML Daten

Da die Typemapping-Daten als XML-Datenstrom vorliegen, ist es unabdingbar, eine passende Application Programming Interface (API) zur Verarbeitung von XML-Daten zu wählen. Die vorhandene Umsetzung setzt, wie beschrieben die Low-Level XML-API dom4j ein, welche allerdings den neuen Anforderungen teilweise nicht gewachsen ist. Folgend wird auf die einzelnen Arten der XML-Verarbeitung in Java eingegangen um die bestmögliche API auszuwählen.

4.1 XML Processing

Seit dem Aufkommen sogenannter Auszeichnungssprachen ist der Bedarf selbige computergesteuert zu verarbeiten immer weiter gewachsen. Aufgrund der Standardisierung durch das World Wide Web Consortium (W3C) weisen alle XML-Datenströme dieselbe Struktur auf und können trotzdem je nach Bedarf angepasst werden. Grundlage ist die Wohlgeformtheit[1] sowie die Gültigkeit[2], das heißt dass die Dokumente einer DTD, dem später aufkommenden XSD oder einer alternativen Definition gegenüber valide sind. Durch diese Spezifikationen sind XML-Datenströme weit verbreitet und werden bevorzugt für Konfigurationsdaten sowie zur Beschreibung von Daten verwendet. Java bietet dem Programmierer verschiedene Möglichkeiten solche XML-Daten zu verarbeiten. Die ältesten APIs sind XML-Processing-APIs. Sie werden aufgrund ihrer Nähe zum eigentlichen XML-Datenstrom auch Low-Level APIs genannt. Die Simple API for XML (SAX) war die erste verfügbare API und gilt immer noch als die performanteste. Als ereignisgesteuerte API verarbeitet diese den XML-Datenstrom sequentiell. Trifft der Parser auf einen gesetzten Event, übergibt er die Kontrolle an die Anwendung, welche erst nach erfolgreicher Ereignisbehandlung dem Parser die Kontrolle zurückgibt. Allerdings muss der Programmierer der Anwendung stets die Position im Datenstrom bezüglich des Handlers verwalten. Es handelt sich um das sogenannte Push-Prinzip[3]. Seiner Schnelligkeit stehen die enorme Komplexität sowie der proportional steigende Speicherbedarf bei zu validierenden Datenströmen gegenüber. Alternativ zum SAX kam dom4j als baumorientierte

[1] Siehe [McL02, S. 11], jedes öffnende Tag hat ein zugehöroges schliessendes Tag und die Verschachtelung der Tags darf nicht verletzt werden
[2] Siehe [McL02, S. 11], XML Daten sind valide wenn sie einer Definition genügen
[3] Siehe [VV06, S. 36ff]

API auf. Dadurch, dass XML-Dokumente immer einen Baum abbilden, ist es möglich diese in das Document Object Model (DOM) zu transformieren. Nach dem Einlesen der Daten werden diese als Dokument-Objekt zur Verfügung gestellt um sie ohne weiteres zu traversieren und auswerten zu können. Die Komplexität gegenüber SAX nimmt deutlich ab, mit dem Verlust an Performance und dem erhöhten Bedarf an Speicher, da der Datenstrom immer vollständig geladen werden muss. Eine Symbiose aus den beiden Technologien stellt Streaming API for XML (StAX) dar, welches auf dem Pull-Prinzip[4] basiert. Ein Cursor wird im Dokument positioniert und von der Anwendung gesteuert. Ist die gesuchte Stelle erreicht, kann die Anwendung die gesuchten Informationen vom Parser anfordern. Performance und Minderung der Komplexität sind die Folge dieser Vorgehensweise. Diese drei APIs zur XML-Verarbeitung bilden die Grundlage von Java API for XML Processing (JAXP) und mit der aktuellen Version 1.3 (beziehungsweise Maintenance Release 1.4) spezifiziert nach JSR-205[5].

4.2 XML Binding

Mit dem steigenden Bedarfs der XML-Verarbeitung, der Umständlichkeit der Low-Level APIs und der damit einhergehenden Fehleranfälligkeit wurden sogenannte High-Level APIs entwickelt, welche auf XML-Processing aufbauen, aber dem Programmierer mühselige und wiederkehrende Aufgaben abnehmen. Damit einher geht der Verlust von Flexibilität und Geschwindigkeit bei der Verarbeitung, allerdings in einem vertretbaren Rahmen, da diese APIs in der Regel optimierte Low-Level-Zugriffe nutzen. Die Reduktion von Quellcode und der vereinfachte Zugriff auf die Daten senken die Fehlerrate und der Prozess des Parsens und Validierens entfällt größtenteils, da dieser hintergründig durch die API stattfindet. Neben den Binding-APIs gibt es speziell zur effizienten Verarbeitung großer XML-Dateien "nicht extrahierende" XML-APIs wie Virtual Token Descriptor for eXtensible Markup Language (VTD-XML)[6], welche allerdings aufgrund ihres spezifischen Einsatzgebietes nicht in die nähere Auswahl der High-Level APIs kamen. Übliche High-Level APIs nutzen den Data Binding-Ansatz, welcher die persistenten Daten als Objektgraph zur Verfügung stellt. Diese Objektgraphen basieren, im Falle von Java, auf JavaBeans, welche die Definition der XML-Daten repräsentieren. Liegt eine DTD[7], XSD[8] oder möglicherweise Regular Language Description for XML New Generation (RELAX

[4] Siehe [VV06, S. 37ff]
[5] Siehe [SWK]
[6] http://vtd-xml.sourceforge.net/
[7] http://www.w3.org/TR/REC-xml/
[8] http://www.w3.org/XML/Schema

NG)[9] vor, können daraus - soweit es die Binding API zulässt - JavaBeans erstellt oder gegebenenfalls generiert werden. XMLBeans[10] ist eine Binding API von Apache, die von Beginn an vollen XML-Schema Support zur Verfügung stellte. Basierend auf den zugrundeliegenden XMLObject, XMLCursor und SchemaType APIs wird dem Programmierer ein pragmatischer Ansatz zur Verarbeitung von XML-Daten zur Verfügung gestellt[11]. Die XMLCursor API bietet hervorragende Möglichkeiten mittels Cursor, sowie mit XQuery zu navigieren. XMLObject repräsentieren die JavaBeans, und die SchemaType API bietet die Unterstützung für verschiedene Schemata.
JiBX[12], welches unter der BSD Lizenz steht, ist eine der performantesten Binding-APIs für Java, welche allerdings auf einen eigenen Parser setzt und einige Einschränkungen bei der Validierung von XML-Daten mit sich bringt. JAXB wird direkt von SUN angeboten und ist aktuell spezifiziert nach JSR-222[13]. Mittlerweile ist es Bestandteil von Java6. In der Revision 1.x waren die Möglichkeiten stark begrenzt, gewisse Vorgänge setzten Workarounds voraus[14]. Mit dieser Version gab es ausschließlich Support für DTDs. Mit dem Revisionssprung auf 2.0 änderte sich ein Großteil der API und es wurde der Support für weitere Schemata hinzugefügt. Der Validierungsprozess wurde automatisiert und die API um die Funktion des bidirektionalen Mappings erweitert. Dieses lässt es dem Programmierer offen, ob er nun mit dem Objektgraphen oder mit den XML Daten an sich arbeiten will. Weitere Binding APIs sind entweder nicht den Anforderungen entsprechend beziehungsweise kaum bekannt oder etabliert genug um ausreichend Support Zukunftsicherheit zu gewährleisten, oder Spezialformen der oben genannten mit der Optimierung auf bestimmte Anwendungsgebiete[15].

4.3 Auswahl von JAXB

Die Wahl der einzusetzenden API viel aus folgenden Gründen auf JAXB:

- High Level API
- direkter Zugriff auf den Objektgraph

[9] http://relaxng.org/
[10] http://xmlbeans.apache.org/
[11] Vergleiche [VV06, S. 185ff]
[12] http://jibx.sourceforge.net/
[13] Siehe [Kaw]
[14] Vergleiche [McL02, S. 121ff]
[15] Vergleiche JaxME2, EMF...

- Integrierte Validierung der XML-Daten
- Generierung von JavaBeans aus der XSD
- Entwickelt und bereitgestellt von SUN, sowie Bestandteil von Java6
- spezifiziert nach JSR-222

Wenn keine spezifischen Anforderungen an die Binding API gestellt werden, wird der Einsatz von JAXB empfohlen[16]. Diese von Sun spezifizierte und entwickelte Binding-API gewährleistet zum einen Zukunftssicherheit im Bereich der Weiterentwicklung und Optimierung, aber auch Zuverlässigkeit, da sie auf JAXP aufbaut. Performanz sowie eine optimierte Navigation im XML-Dokument stehen für das Typemapping nicht im Vordergrund. Wichtiger ist die Lesbarkeit und eine geringe Komplexität um Fehler zu vermeiden und weitere Direktiven problemlos in den Typemapping Mechanismus aufzunehmen.

[16] Vergleiche [VV06, S. 185]

5 Funktionalität von JAXB

Wie im vorigen Kapitel erwähnt fiel die Wahl der API zum Arbeiten mit XML-Daten in Java auf JAXB. Aus diesem Grund werden folgend die Funktionalitäten, wie sie im späteren Projektverlauf eingesetzt werden, näher erläutert und zusätzlich werden notwendige Kenntnisse für das spätere Verständnis geschaffen.
Die Voraussetzungen für JAXB 2.0, wie es hier verwendet wird, sind Java Standard Edition 5 zusammen mit der JAXB Referenzimplementierung, welche die notwendigen Bibliotheken enthält. Alternativ kann Java Standard Edition 6 verwendet werden in welcher JAXB schon enthalten ist. Zusätzlich ist Apache ANT für die Generierung von JavaBeans oder Schemata notwendig. Die eingesetzte Entwicklungsumgebung ist im vorliegenden Fall Eclipse 3.3. Die JAXB Referenzimplementierung liefert folgende notwendige Java-Archive mit:

- jaxb-api.jar: Dieses Archiv enthält die Grundlegenden API-Klassen der JAXB Spezifikation
- jaxb-impl.jar: Bei den hier enthaltenden Klassen handelt es sich um die JAXB Referenzimplementierung
- jsr173_1.0_api.jar: Diese Bibliothek bringt die XML Streaming API mit
- activation.jar: Dieses Archiv enthält die Abhängigkeiten zur JavaBeans Activation API
- jaxb-xjc.jar: Zur Generierung von Schemata aus JavaBeans bzw. JavaBeans aus Schemata wird diese Bibliothek benötigt

Ein Einführungsbeispiel[1] soll verdeutlichen, wie man mit JAXB arbeitet. Die folgenden Beispiele dienen allein der Veranschaulichung und behandeln deswegen nicht das Typemapping.
JAXB setzt in der Regel immer ein JavaBean voraus.

```
1 @XmlRootElement
2 public class Messenger {
3     private String message;
4
5     public String getMessage() {
6         return message;
```

[1] Vergleiche. Listing 5.1, 5.2, 5.3

```
7       }
8
9       public void setMessage(String message) {
10          this.message = message;
11      }
12  }
```

Listing 5.1: Ausschnitt aus einer einfachen JavaBean für JAXB - Siehe Messenger.java Zeile 13-24 unter Sourcecode im de.genesez.examples Projekt

Die Annotation `@XmlRootElement` veranlasst JAXB dazu, dieses JavaBean als Wurzelelement der XML Datei zu betrachten.

```
1 JAXBContext context = JAXBContext.newInstance(Messenger.class);
2 Marshaller marshaller = context.createMarshaller();
3 Messenger messenger = new Messenger();
4 messenger.setMessage("Erstellen einer XML Datei mittels JAXB 2.0"
     );
5 marshaller.marshal(messenger,new FileWriter("messenger.xml"));
```

Listing 5.2: Unter Verwendung der Bean wird eine persistente Datei geschrieben - Siehe Introduction.java Zeile 19-24 unter Sourcecode im de.genesez.examples Projekt

JAXB erfordert neben einem Kontext, der die Grundlage für jegliche Arbeit mit XML-Daten ist, in diesem Fall auch einen Marshaller, welcher Objektgraphen in XML-Datenströme wandeln kann.

```
1 <messenger>
2  <message>Erstellen einer XML-Datei mittels JAXB 2.0</message>
3 </messenger>
```

Listing 5.3: Ausschnitt aus der erzeugten XML-Datei - Siehe messenger.xml Zeile 2-4 unter Sourcecode im de.genesez.examples Projekt

Das Listing 5.3 stellt den Inhalt des resultierenden Ergebnisses dar. Das aufgeführte Einführungsbeispiel soll für JAXB sensibilisieren und die offensichtlichen Vorzüge aufzeigen. Die Abbildung

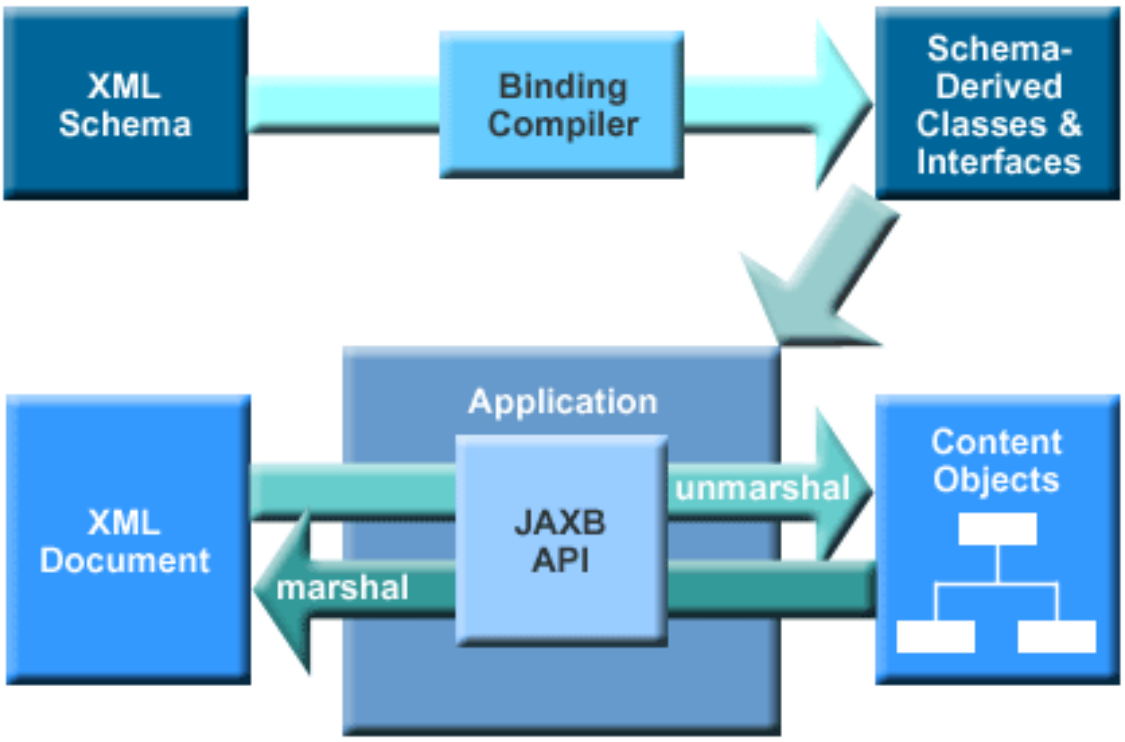

Abbildung 5.1: Arbeitsweise von JAXB

5.1[2] stellt noch einmal grafisch dar, wie JAXB arbeitet. Grundlage bilden wie eben beschrieben die JavaBeans und JAXB. Marshalling kann als spezielle Serialisierungsart angesehen werden, bei welcher aus einem Objektgraphen ein XML-Datenstrom erstellt wird. Der Gegenpart dazu ist das sogenannte Unmarshalling, welche eine Spezialform der Deserialisierung darstellt. Es wird aus einer persistenten XML-Datei ein Objektgraph erzeugt, der mit der JavaBean harmoniert. Grundlage für diesen Prozess ist ein XML-Schema, zu welchem die XML Datei valide sein muss.

5.1 Das Erzeugen von JavaBeans

JAXB bietet verschiedenste Möglichkeiten, um XML-Daten zu verarbeiten. Im vorliegenden Fall wurde das Generieren von JavaBeans aus einem selbst definierten XML-Schema mittels ANT Task bevorzugt, weil die Vorteile daraus überwiegen. Das Erstellen der JavaBeans entfällt und damit wird auch das Testen der Beans obsolet und verlagert sich auf das Schema. Solange das Schema nicht geändert wird, wird sich das Generat immer gleich verhalten. Die Anfor-

[2] http://java.sun.com/developer/technicalArticles/WebServices/jaxb/

derungen, die im Schema gesetzt werden, brauchen bloß an dieser Stelle gesetzt werden und jegliche Änderungen am Schema können sofort nach dem Kompilieren der neuen JavaBeans im Quelltext angewendet werden. Nachteil an diesem Vorgehen ist zum einen der Verlust der Flexibilität beim Erstellen der JavaBeans, zum anderen ein Mehraufwand durch das Erstellen von Bindungsdeklarationen um die generierten JavaBeans anzupassen. Erstellt man die JavaBeans selbst und lässt sich das Schema daraus generieren, kann es unter Umständen zu Veränderungen des generierten Schemas kommen, wenn verschiedene Revisionen der JAXB Bibliotheken genutzt werden. Dies wiederrum hat zur Folge, dass die XML Dateien gegebenenfalls angepasst werden müssten. Der Verlust der Flexibilität kann fernerhin mit neueren Revisionen der Binding API kompensiert werden, da schon spezielle Parameter zur Kennzeichnung generierter Daten beim Schema Compiler angegeben werden können. Leider ist es mit der aktuellen Version nicht möglich den ANT Task so zu steuern, dass nur generierte Elemente, oder gar Quelltextteile gelöscht werden. Ein Beispiel für die Flexibilität durch eigenhändig erstellte JavaBeans wäre der Einsatz assoziativer Arrays anstelle von `ArrayList` in den JavaBeans. Folgend wird ein Beispielschema[3] zur Veranschaulichung verwendet. Es definiert eine Liste aus Kunden, die ein oder mehrere Adressen besitzen können. Es ist beim Entwurf des Schemas darauf zu achten, die Kardinalitäten an den Elementen statt an den Type Compositors[4] zu setzen, da es sonst zu Inkonsistenzen mit den Bindungsdeklarationen und die generierten JavaBeans das erwartete Verhalten nicht aufweisen können. In der zugehörigen XML-Datei[5] sind die persistenten Daten enthalten. Die durch JAXB 2.0 eingeführten Bindungsdeklarationen können noch spezifische Änderungen am später resultierenden Generat annektieren. Diese Bindungsdeklarationen werden über die Annotationstags des XML-Schemas eingefügt und können sowohl inline als auch extern definiert werden. Aufgrund der Lesbarkeit und der Wiederverwendbarkeit ist es sinnvoll externe Definitionen einzusetzen, mit dem Nachteil dass man mit der XML Path Language (XPath) Ausdrücke zu den jeweiligen Schematags navigieren muss.

```
1 <jaxb:bindings node="//xs:element[@name='Customers']">
2  <jaxb:class name="CustomerContainer" />
3   <jaxb:bindings node=".//xs:element[@name='Customer']">
4      <jaxb:property name="CustomerList" generateIsSetMethod="true
          " ></jaxb:property>
5    </jaxb:bindings>
```

[3]Siehe customers.xsd unter Sourcecode im de.genesez.examples Projekt
[4]zum Beispiel `xs:sequence`
[5]Siehe customers.xml auf der cd

```
6 </jaxb:bindings>
```

Listing 5.4: Bindungsdeklaration für das Wurzelelmement Customers - Siehe customers.xjb Zeile 14-19 unter Sourcecode im de.genesez.examples Projekt

Der Ausschnitt des Listings 5.4 aus der Bindungskonfiguration übermittelt dem Schema Compiler die Information, das Wurzelelement Customers als CustomerContainer zu generieren und das enthaltende Element Customer in CustomerList umzubenennen. Selbigem soll weiterhin eine IsSet Methode angefügt werden um direkt über die Bean ermitteln zu können, ob Kunden enthalten sind. Der durch JAXB mitgelieferte Schema Compiler xjc kann via Java-Quelltext, mittels Kommandozeilenbefehle oder durch einen ANT Task gerufen und gesteuert werden. Da das Generieren der JavaBeans bei etwaigen Änderungen am Schema oder an den Bindungsdeklarationen benötigt wird, ist es sinnvoll, einen ANT Task, wie er in Listing 5.5 aufgezeigt wird, bereitzustellen. Ausserdem ist es mit dem ANT Task möglich den Prozess zu automatisieren.

```
1 <xjc destdir="src" binding="data/de/genesez/examples/customers.
    xjb"
2  removeOldOutput="yes" extension="true"
3  package="de.genesez.examples.generated">
4     <produces dir="src/de/genesez/examples/generated" includes="
        **/*" />
5     <schema dir="data/de/genesez/examples">
6         <include name="customers.xsd" />
7     </schema>
```

Listing 5.5: Der Aufruf des Schema Compilers um die JavaBeans zu generieren siehe build.xml Zeile 13-18 unter Sourcecode im de.genesez.examples Projekt

Der Aufruf des Schema Compilers erfolgt chronologisch vor dem eigentlichen Build, da der Programmcode Bezug auf die generierten JavaBeans nehmen wird. Einzig die Angabe des Schemas und des Zieles sind Prämissen. Die daraus generierten Java Beans[6] können unmittel-

[6] Siehe CustomerContainer.java, Customer.java und Address.java unter Sourcecode im de.genesez.examples Projekt

bar für das eigentlichen Binding verwendet werden. Das Erstellen des Objektgraphen aus der persistenten Datei[7] erfolgt durch den Aufruf des Unmarshallers.

5.2 Unmarshalling

Der schon erwähnte Deserialisierungsprozess wird in JAXB als Unmarshalling bezeichnet und aufgrund seiner Eigenschaften das Parsen von XML-Daten weitestgehend ablösen. Im Gegensatz zum Marshalling gibt es für den Deserialisierungsprozess keine weiteren Konfigurationsmöglichkeiten. Eine typische Anwendung des Unmarshallings ist in Listing 5.6 zu sehen.

```
1 JAXBContext context = JAXBContext.
2  newInstance("de.genesez.examples.generated");
3 Unmarshaller unmarshaller = context.createUnmarshaller();
4 CustomerContainer customers=(CustomerContainer)unmarshaller.
5 unmarshal(new File("data/de/genesez/examples/customerlist.xml"));
```

Listing 5.6: Unmarshalling der customerlist.xml - sample01.java Siehe Zeile 25-37 unter Sourcecode im de.genesez.examples Projekt

Das Eingabeformat kann in verschiedenen Datenströmen[8] vorliegen, wie zum Beispiel als Datei oder DOM-Teilbaum. Der Unmarshaller gibt ein Objekt mit dem Verhalten eines `JAXBElement` zurück, welches dann zum spezifischen Typ umgewandelt werden kann. Im Beispiel ist der `CustomerContainer` mit `@XMLRootElement` annotiert, was die Klasse dazu privilegiert direkt genutzt werden zu können. Baut man hierarchische XML-Schemata auf, ist es möglich valide XML-Dokumente zu binden die kein Wurzelelement aufweisen[9]. Eine weitere Variante ist es, nur einen Teil des XML-Datenstroms zu binden um Elemente heraus zu extrahieren die ebenfalls die `@XMLRootElement` Annotation nicht aufweisen und nur die gesuchten Daten zur Verfügung stellen, welches das Listing 5.7 verdeutlicht.

```
1 Node addressNode = (Node)xpath.evaluate("//Customer/ex:Address",
     document,XPathConstants.NODE);
2 JAXBElement<Address> elem = unmarshaller.unmarshal(addressNode,
     Address.class);
```

[7] Siehe sample01.java unter Sourcecode im de.genesez.examples Projekt
[8] Vergleiche [MS06, S. 88]
[9] Vergleiche [MS06, S. 92]

```
3 Address address = (Address) elem.getValue();
```

Listing 5.7: Unmarshalling der customerlist.xml - siehe sample03.java Zeile 44-48 unter Sourcecode im de.genesez.examples Projekt

Unter Verwendung von JAXP 1.3 wird der Dokumentbaum erstellt und mittels XPath an die gesuchte Stelle navigiert. Der Unmarshaller erstellt anhand des Knotens in Kombination mit der JavaBean den Objektgraphen, welcher vorerst nur als `JAXBElement` vorliegt. Ein Typecast gewährleistet anschließend das Weiterarbeiten mit dem von nun an gebundenen Element. Der Unmarshaller, wie auch die Evaluierungsmethode des XPathAusdruckes können mit invaliden Dokumenten umgehen, allerdings ist es möglich, dass es zu fehlerhaften oder unvollständigen Objektgraphen kommen kann. Aus diesem Grund sollte man die in JAXB integrierte Validierungsfunktionalität nutzen.

5.3 Validierung

Mit dem Revisionssprung von JAXB 1.0 auf JAXB 2.0 hat sich der Validierungsprozess gewandelt. Die Validierung kann nicht mehr direkt beeinflusst werden, sondern findet während des Marshalling bzw. Unmarshalling Prozesses statt[10]. Bei nicht wohlgeformten XML-Dokumenten wird stets eine Ausnahme geworfen; sind die Dokumente dagegen nicht valide, greift das gesetzte Schema oder eine Instanz des `ValidationEventhandler`.

```
1 SchemaFactory sf = SchemaFactory.
2  newInstance(javax.xml.XMLConstants.W3C_XML_SCHEMA_NS_URI);
3 Schema schema  = sf.newSchema(
4  new File("data/de/genesez/examples/customers.xsd"));
5 unmarshaller.setSchema(schema);
```

Listing 5.8: Setzen des Schemas für den Unmarshallingprozess - siehe sample01.java Zeile 28.32 unter Sourcecode im de.genesez.examples Projekt

Ist das Schema gesetzt, werfen alle Verstöße dagegen eine Ausnahme. Speziell Einschränkungen, die den Typ des Elements betreffen, werden durch diese Methode validiert. Eine flexiblere

[10] Vergleiche Listing 5.8, 5.9, beide Varianten aktivieren einen Validierungsmechanismus für den Deserialisierungsprozess

Art der Validierung ist der Einsatz eines benutzerspezifischen `ValidationEventHandler` , welcher granularere Abstufungen der einzelnen Fehlerstufen realisieren kann[11].

```
1 unmarshaller.setEventHandler(new MyValidationEventHandler());
```

Listing 5.9: Setzen eines ValidationEventhandlers für den Unmarshallingprozess - siehe sample01.java Zeile 34 unter Sourcecode im de.genesez.examples Projekt

Über einen `ValidationEventCollector` ist es möglich, mehrere solcher Handler zu registrieren. Wichtig ist es, den Objektgraphen mittels dieser Handler unter keinen Umständen zu verändern ,da dies zu unerwarteten Ergebnissen führen kann. Die Dokumentation sieht vor, eine der beiden Varianten zu implementieren um die XML-Daten vor dem Binden zu validieren. Aus Performanzgründen ist es möglich die Validierung zu unterlassen, allerdings besteht wie schon oben erwähnt die Gefahr von inkonsistenten Daten. Die Dokumentation stimmt im Falle der Validierung nicht ganz mit der realen Implementierung überein, denn der `ValidationEventHandler` behandelt nur die Fehler, welche auf den JavaBeans beruhen. Verwendet man das Schema, werden alle, auch das Schema betreffende Validierungsprobleme behandelt. So wird zum Beispiel die Einschränkungen auf eine bestimmte Länge einer Zeichenkette vom `ValidationEventHandler` weder abgefangen noch behandelt. Durch das validieren mittels Schema befindet man sich auf der sicheren Seite, da selbige Validierung das Fail Fast-Prinzip realisiert. Zusätzlich kann man noch `ValidationEventHandler` registrieren, um diverse Ereignisse zu protokollieren. Eine detailliertere Erläuterung, welche für das Typemapping nur von marginaler Relevanz ist, bietet das JAXB-HowTo unter Documents auf der CD.

[11] Siehe MyValidationEventHandler.java unter Sourcecode im de.genesez.examples Projekt

6 Umsetzung

Nachdem alle notwendigen Spezifikationen bekannt waren, konnte umgehend mit der Entwicklung des Typemappings auf XML-Binding-Basis begonnen werden.

6.1 Entwurf und Evolution der XML Schema Definition

Vor dem Schreiben von Java-Quelltext war es notwendig, basierend auf einer Beispiel XML-Datei, welche in Zusammenarbeit mit dem GeneSEZ Team erstellt wurde, eine XSD zu definieren. Das Erstellen eines XML-Schemas kann auf verschiedene Weisen realisiert werden. Im angewandten Fall wurde mittels Liquid XML Studio ein Schema ohne eindeutigem Wurzelelement erzeugt, das heißt jedes Element ist Kind der Schemawurzel, wobei die zugewiesenen Typen der Elemente die korrekte Struktur[1] enthalten. Der Vorteil dieses Aufbaus ist die Flexibilität und Erweiterbarkeit. Miteinander einher geht der Verlust der Lesbarkeit sowie der im späteren Verlauf festgestellte Missstand, dass der Schema-Compiler kein Element mit `@XMLRootElement` annotiert. Diese Art des Schemaaufbaus findet sich ebenfalls bei Schemata von Microsoft, wie zum Beispiel Microsoft Office 2007 oder eXtensible Application Markup Language (XAML) wieder. Der grundlegende Aufbau bezüglich der vorangegangen Lösung wurde um ein include Tag erweitert, welches die zu referenzierenden XML-Dateien enthalten kann. Die Tags für die verschiedenen Mapping Typen wurden nur marginal verändert. Weiterhin wurde das fertiggestellte Schema teilweise angepasst. Die Kardinalitäten wurden in die referenzierten `element` Tags verschoben aufgrund der Inkompatibilität von JAXB beim Generieren der JavaBeans. Weiterhin wurden die Restriktionen erst im Nachhinein hinzugefügt, da selbige das Generat nicht beeinflussen. Abbildung 6.1 zeigt einen Ausschnitt aus dem komplexen Typ `typeMappingType`, welcher den Typ des Wurzelelements repräsentiert. Folgend werden die Vor- und Nachteile der Vorgehensweise aufgelistet:

- Vorteile
  - sehr flexibel
  - problemlos erweiterbar

[1] Siehe typemapping.xsd innerhalb des Sourcecode Ordners im de.genesez.typemapping Projekt

- Nachteile
  - schwer zu lesen
  - Schema Compiler annotiert kein Element mit `@XMLRootElement`

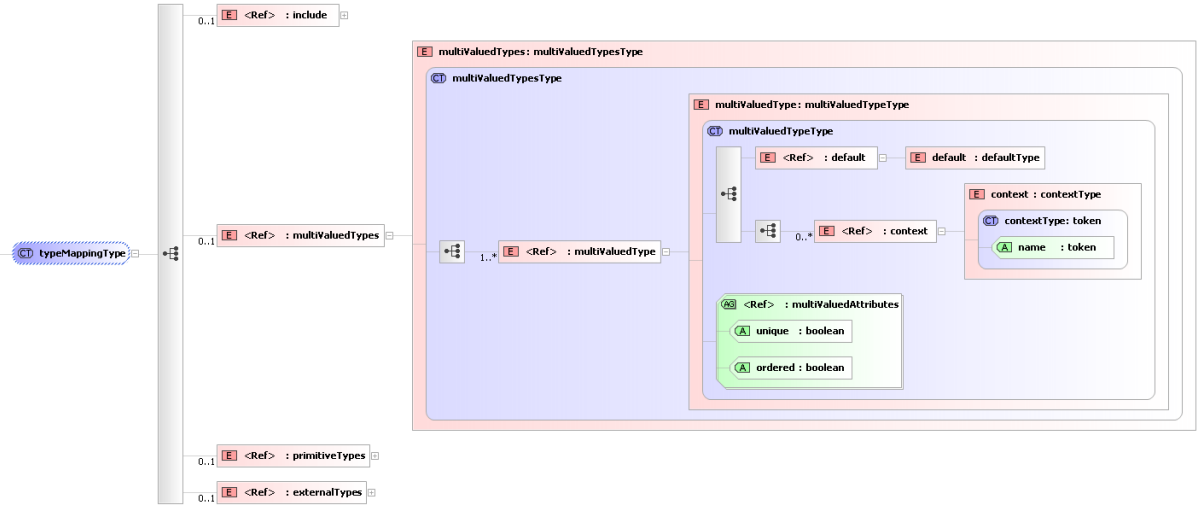

Abbildung 6.1: Ausschnitt aus dem komplexen Typ für das eigentliche Wurzelelement der Typemapping-Dateien

6.2 Generieren der JavaBeans für das Typemapping

Natürlich ist es nach dem Schemaentwurf unmittelbar möglich die JavaBeans generieren zu lassen, allerdings würden die vorgegebenen Standards des Schema-Compilers verwendet. Die Folge daraus sind unter anderem unpassende Klassen- und Attributnamen. Aus diesem Grund, und auch um weitere Anpassungen durchzuführen, wurde eine Bindungskonfiguration zur Anpassung der JavaBeans erstellt[2].

```
1 <jaxb:bindings node="//xs:complexType[@name='typesType']">
2  <jaxb:class name="SingleValuedTypeContainer" />
3  <jaxb:bindings node="..//xs:element[@ref='tm:type']">
4   <jaxb:property name="MappingList" />
5  </jaxb:bindings>
6 </jaxb:bindings>
7
8 <jaxb:bindings node="//xs:complexType[@name='multiValuedTypeType
    ']">
```

[2]Siehe typemapping.xjb innerhalb des Sourcecode Ordners im de.genesez.typemapping Projekt

```
9  <jaxb:class name="MultiValuedType" />
10 <jaxb:bindings node=".//xs:element[@ref='tm:default']">
11  <jaxb:property name="destinationMapping" generateIsSetMethod="
       true" />
12 </jaxb:bindings>
```

Listing 6.1: Bindungsdeklarationen zur Anpassung des Generats - Siehe Zeile 46-57 typemapping.xjb

Das Quelltextlisting 6.1 bewirkt Folgendes. Die Namen komplexer Typen, welche wiederrum Elemente mit einer `multiplicity` größer Eins enthalten, erhalten den Suffix `Container` und die Elemente darin, welche dann zu Attributen werden, rezipieren den Suffix `List`, da alle Collections als `ArrayList` abgebildet werden. Des Weiteren wurden optionale Elementen um eine `IsSet` Methode zur simplifizierten Auswertung ergänzt. Das Generieren der JavaBeans wurde dann durch einen ANT Task[3] realisiert, welcher beim Verändern des Schemas oder einer Bindungsdeklaration erneut durchgeführt werden kann und automatisiert das generierte `Package` neu erstellt. Aufgrund der hierarchischen Struktur von XML Daten sind die JavaBeans ebenfalls hierarchisch aufgebaut und setzen den Composite Pattern[4] um. Als Composite Pat-

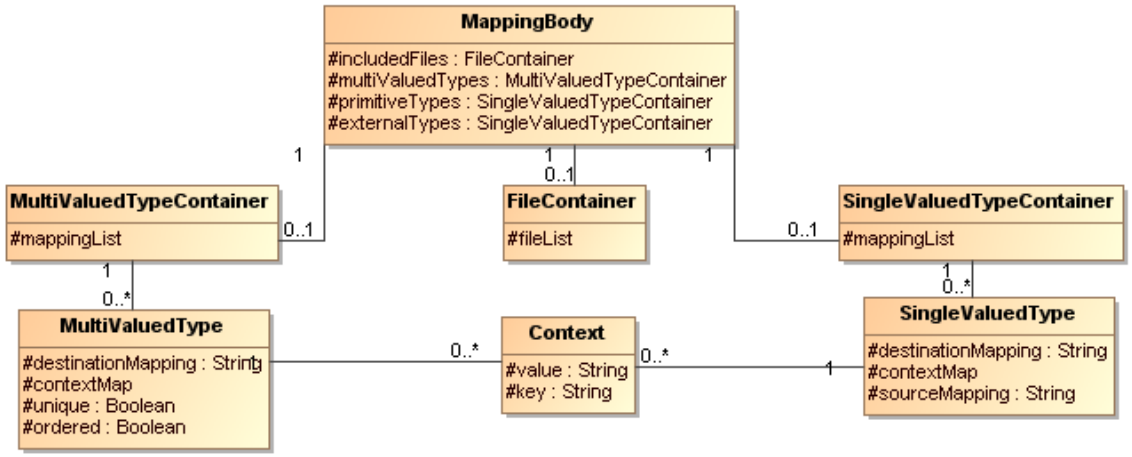

Abbildung 6.2: Klassendiagramm der aus dem Schema generierten mappingtypes

tern bezeichnet man die Abbildung einer Baumstruktur beziehungsweise die Repräsentation einer Teil-Ganzes-Hierarchie. Jedoch handelt es sich hier nicht direkt um eine Teil-Ganzes-Hierarchie, da die einzelnen JavaBeans unabhängig voneinander existieren können. Indes wird

[3]Siehe build.xml innerhalb Sourcecode Ordners im de.genesez.typemapping Projekt
[4]Vergleiche [GHJV96, S. 213], Ein Strukturmuster um Teile-Ganzes-Hierarchien oder Baumstrukturen zu repräsentieren

der Antipattern einer Nachrichtenkette[5] implementiert. Gegen diese Code Smell wurde via "Hide Delegate"[6] Refactoring entgegengewirkt. Das Generat ist im `package de.genesez.typemapping.typemappingtypes` zu finden.

6.3 XML Validierung und Binding

Da die `Typemapper`-Klasse stetig gewachsen ist und immer mehr längere Methoden[7] aufwies, entstand unter Verwendung von Delegation die Klasse `BindingDelegator`. Als Vermittler[8] stellt sie den anderen Klassen nur die notwendigsten JAXB Funktionalitäten zur Verfügung um eine weitestgehend lose Kopplung zu realisieren. Diese Delagationsklasse enthält das Binding und die Validierung der XML-Daten. Durch den `Typemapper` wird die Klasse initialisiert und aktiviert die jeweiligen Validierungsoptionen, wie das Listing 6.2 zeigt.

```
1 Schema schema = SchemaFactory.newInstance(
2     javax.xml.XMLConstants.W3C_XML_SCHEMA_NS_URI).newSchema(
3         new File("src/typemapping.xsd"));
4         unmarshaller.setSchema(schema);
5         unmarshaller
6             .setEventHandler(new
                 ErrorLoggingValidationEventHandler());
```

Listing 6.2: Aktivieren der Validierungsoptionen für das Typemapping - siehe BindingDelegator.java Zeile 70-75 innerhalb des Sourcecode Ordners im de.genesez.typemapping Projekt

Als einzige Klasses welche direkt auf den `Unmarshaller` zugreift, behandelt sie ebenfalls das Binden. Durch die Delegation werden die XML-Dateien an den `BindingDelegator` gereicht, welcher dann die Objektgraphen zurückgibt, wie in Listing 6.3 zu sehen ist.

```
1 public MappingBody getMappingBody(String mappingFile) {
2         JAXBElement<MappingBody> element;
```

[5] Vergleiche [FBB+99, Kapitel 3], Das anfragende Objekt fragt nach einem Objekt des Objekts... Indikator sind verkettete Aufrufe von getObject Methoden

[6] Vergleiche [FBB+99, Kapitel 7], Kapseln der Delegate in einer Methode

[7] Vergleiche [FBB+99, Kapitel 3], Grosse Klassen sind der Indikator für eine Klasse die zuviel Verantwortung trägt und die Gefahr des duplizierten Quelltextes birgt, lange Methoden sind schwer zu erfassen

[8] Vergleiche [GHJV96, S. 345], steuert das Verhalten von Objekten. Der Vermittler kapselt die Kommunikation zwischen Objekten

```
3         try {
4             element = (JAXBElement) unmarshaller.unmarshal(
                  ClassLoader
5                       .getSystemResourceAsStream(mappingFile));
6             return element.getValue();
```

Listing 6.3: durch die Delegation findet das Binding transparent statt - siehe BindingDelegator.java Zeile 126-133 innerhalb Sourcecode Ordners im de.genesez.typemapping Projekt

Diese Vorgehensweise kommt dem "Gesetz von Demeter"[9] zur Vermeidung von Kopplung nach.

6.4 Multi-Include Mechanismus

Die komplexeste Anforderung stellte der Multi-Include Mechanismus dar. Basierend auf der generierten Klasse `FileContainer` ist es möglich, die referenzierten Dateien im Objektgraph verfügbar zu machen. Durch das Binden von Teilbäumen wird über den `BindingDelegator` der Objektgraph an den Teilbaum gebunden. Die JavaBean `FileContainer` repräsentiert in diesem Fall den Teilbaum in Java. Das Listing 6.4 nutzt einen XPathAusdruck zur Navigation und gibt den Objektgraph an den `MappingFileCollector`.

```
1 Document typeMappingDoc = documentBuilder.parse(ClassLoader
2     .getSystemResourceAsStream(mappingFile));
3 Node includeNode = (Node) xPath.evaluate(xPathExpression,
4     typeMappingDoc, XPathConstants.NODE);
5 if (includeNode != null) {
6     JAXBElement<FileContainer> element;
7     element = unmarshaller.unmarshal(includeNode,
8         FileContainer.class);
9         return element.getValue();
```

Listing 6.4: partielles Unmarshalling basierend auf einem Teilbaum - siehe BindingDelegator.java Zeile 157-165 innerhalb Sourcecode Ordners im de.genesez.typemapping Projekt

[9] Siehe [HT99, S.140], auch "Principle of Least Knowledge" es besagt das Objekte nur mit in der Nähe befindlichen Objekten kommuzieren sollen um Kopplung möglichst gering zu halten

Eine elementare Rolle spielt der Algorithmus zur Zusammenstellung der zu referenzierenden Typemapping-Dateien. Basierend auf der Wurzelmapping-Datei werden die Dateien rekursiv traversiert. Das Parsen erfolgt, wie bei der Breitensuche, ebenenweise um die geforderte Reihenfolge der Mapping Dateien zu gewährleisten. Das Listing 6.5 stellt den Ausschnitt des Algorithmus in Java dar.

```
1  if (fileContainer != null) {
2      for (String file : fileContainer.getFileList()) {
3          if (!(fileSet.contains(file))) {
4              fileSet.add(file);
5                  workList.addLast(file);
6          }
7      }
8  }
9  if (!workList.isEmpty()) {
10     aggregateMappingFiles(workList.removeFirst());
11 }
```

Listing 6.5: Parsen und Aggregieren der zu referenzierenden Typemapping Dateien - siehe MappingFileCollector.java Zeile 78-90 innerhalb Sourcecode Ordners im de.genesez.typemapping Projekt

Ist die Liste vollständig aggregiert, wird diese in der `Typemapper`-Klasse zum Füllen der Manager für die einzelnen Mappings genutzt. Dateien, welche am Anfang der Liste stehen, besitzen eine höhere Priorität, das heißt, dass die enthaltenen Mappings nicht von Mappings aus minder priorisierten Typemapping-Dateien überschrieben werden, sondern nur nicht vorhandene angehangen werden. Kreisreferenzen und Referenzen auf schon enthaltene Elemente wird entgegengewirkt, indem - bevor Elemente hinzugefügt werden - geprüft wird, ob das jeweilige Element enthalten ist. Beim Füllen der Manager für die Mappings wird das gesamte XML-Dokument gebunden und den Managern die passenden `MappingList` der Typencontainer zur Verfügung gestellt. Das Listing 6.6 illustriert diesen Vorgang.

```
1  if (mappingBody.isSetExternalTypes()) {
2      externalTypeManager.appendMappingMap(mappingBody
3          .getExternalTypes().getMappingList());
4  }
5  if (mappingBody.isSetPrimitiveTypes()) {
```

```
6       primitiveTypeManager.appendMappingMap(mappingBody
7           .getPrimitiveTypes().getMappingList());
8   }
9   if (mappingBody.isSetMultiValuedTypes()) {
10      multiValuedTypeManager.appendMappingMap(mappingBody
11          .getMultiValuedTypes().getMappingList());
12  }
```

Listing 6.6: Füllen der Manager zur Verwaltung der Mappings - siehe Typemapper.java Zeile 95-106 im de.genesez.typemapping Projekt

An dieser Stelle kommen unter anderem die Bindungsdeklaration zur Generierung von `isSet` Methoden zum Einsatz, um die Verfahrensweise lesbarer zu gestalten. Das Managerkonzept wurde von der vorangegangen Lösung übernommen aufgrund ihres Vermittler-Charakters. Wie schon im Listing 6.6 zu sehen ist, kann durch das Kompositum der generierten JavaBeans nur bedingt der Antipattern "Message Chain" vermieden werden, durch die Manager werden aber diese Ketten für die einzelnen Mappings weitestgehend verborgen.

6.5 Weitere Anforderungen

Folgend wird auf die restlichen Anforderungen eingegangen:

- Parsen der XML-Dateien reduzieren
- Standardrückgabewerte in den Typemappings
- Case-insensitive Kontexte
- Erstellen von Typemapping-Dateien die valide zum Schema sind
- Registrieren mehrere Typemapping-Dateien beim Transformationsprozess

Anhand des Einsatzes von JAXB entfällt das XML-Parsen weitestgehend und weicht dem Einsatz der Objektgraphen. Die anfangs geforderten Standardrückgabewerte der Typen entfielen. Die Ursache liegt in der Art wie kompilierbarer Quelltext erzeugt wird. Durch das Xpand Skript wurden die Standard Rückgabewerten innerhalb der Operation des kompilierbaren Codes generiert. Dieses Verfahren erschwert eine mögliche Fehlersuche. Aus diesem Grund ist auf ein

Exceptionkonzept ausgewichen worden. Dadurch kann beispielsweise angezeigt werden das eine Methode nicht implementiert wird, aber der Quelltext bleibt weiterhin kompilierbar. Um die Kontexte der Typemappings unabhängig von Groß-/Kleinschreibung zu machen, wird der gesuchte Typ unabhängig von besagter Schreibweise mit den Mappings verglichen, wie das Listing 6.7 veranschaulicht. Gleichwohl ergab sich die Frage ob zusätzliche Kontexte, welche an Typemappings hängen die, die aufgrund ihrer geringeren Priorität nicht gesetzt wurden, an die Typen angehangen werden sollten. Da vorerst kein Bedarf danach besteht wurde diese Variante nicht implementiert.

```
1 for(Context c : specificType.getContextMap()){
2     if(c.getKey().equalsIgnoreCase(context)){
3         return c.getValue();
4     }
5 }
```

Listing 6.7: Case-insensitive Kontexte - siehe TypemappingManager.java Zeile 48-51 innerhalb Sourcecode des Ordners im de.genesez.typemapping Projekt

An der Stelle der Manager wurde das Refactoring "Extract Interface"[10] mit teilweise mäßigem Erfolg eingesetzt. Aufgrund der bis jetzt noch nicht integrierbaren Vererbungshierarchie in die JavaBeans[11] in das Generat ist es weiterhin erforderlich, zwei verschiedene Implementierungen der Manager zur Verfügung zu stellen, obwohl diese sich im Aufbau stark ähneln und somit den Verdacht bezüglich des Code Smell "Duplicated Code"[12] erhärten. Ähnlich verhält es sich mit den beiden Methoden des `BindingDelegator` die ein spezialsiertes `JAXBElement` zurückgeben[13]. An dieser Stelle könnte man versuchen durch eine generische Methode den anfragenden Klassen nur diese Methode zur Verfügung zu stellen. Das Registrieren mehrerer Mapping Dateien direkt beim Workflow lies sich problemlos über die in Java5 eingeführte Funktionalität der Ellipse[14] realisieren.

```
1 public static TypeMapper initTypeMapper(String...
     typeMappingFiles) {
2     if (typeMapper == null) {
3         typeMapper = new TypeMapper(typeMappingFiles);
```

[10] Vergleiche [FBB+99, Kapitel 11]
[11] bezogen auf die generierten Klassen `SingleValuedType` und `MultiValuedType` auf der cd
[12] Siehe [FBB+99, Kapitel 3]
[13] Es handelt sich um `getMappingBody` und `getFileContainerByXPath`
[14] Vergleiche Listing 6.8

```
4       }
5       return typeMapper;
6   }
```

Listing 6.8: Registrieren mehrerer Typemapping-Dateien via Workflow - siehe TypeMapper.java Zeile 206-211 innerhalb des Sourcecode des Ordners im de.genesez.typemapping Projekt

Die in der Ellipse enthaltenen Strings werden anschliessend sequentiell an den `MappingFileCollector` übergeben und jeweils gesammelt. Nachdem der Mechanismus mit vollem Funktionsumfang implementiert war, konnten die schon existierenden Mapping-Dateien auf das jetzt vorhandene Schema angepasst werden, wie Listing 6.9 veranschaulicht. Im Vergleich zu den bisherigen Dateien ist dieser Aufbau durch die Angabe des Namespaces und des Schemas etwas komplexer, allerdings sind diese nicht zwingend erforderlich, werden aber vom W3C empfohlen.

```
1  <tns:typeMapping xmlns:tns="http://www.genesez.de/typemapping"
2      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:schemaLocation="http://www.genesez.de/typemapping ../
           typemapping.xsd">
4    <tns:include>
5      <tns:file>de/genesez/platforms/java/typemapping/typemapping.
           xml</tns:file>
6    </tns:include>
7    <tns:externalTypes>
8    <!-- we used the type 'Color' and declared it as 'external'
        within the uml model -->
9      <tns:type from="Color">
10         <!-- default mapping should return the type name -->
11         <tns:to>Color</tns:to>
12         <!-- used to generate imports -->
13         <tns:context name="Import">java.awt.Color</tns:context>
14     </tns:type>
15   </tns:externalTypes>
```

```
16 </tns:typeMapping>
```

Listing 6.9: Beispiel einer aktualisierten Mapping-Datei - siehe java.car.example.typemapping.xml Zeile 9-24 innerhalb des Sourcecode Ordners im de.genesez.typemapping Projekt

6.6 Tests

Obwohl es sinnvoller ist, die Tests vor dem eigentlichen Quelltext zu schreiben[15], wurden im vorliegenden Projekt die Tests basierend auf JUnit4 in Kombination mit EasyMock im Nachhinein angelegt. Da die Anforderungen bekannt waren und im bereits vorhandenen Projekt keine Tests angelegt wurden, war es nicht zwingend erforderlich die Tests so früh wie möglich zu definieren, obschon es den Entwicklungsprozess immens beschleunigt hätte. Der beibehaltene Singleton[16] `Typemapper` lässt sich allerdings nur durch einen Workaround seiteneffektfrei testen. Dabei ist es notwendig den Pattern aufzuweichen und den Konstruktor `protected` zu setzen, um in einer für den Test angelegten Klasse Selbigen explizit aufrufen zu können[17]. Der Singleton Pattern sollte bewusst vermieden werden aufgrund dieser Tatsache, ist aber an jener Stelle gerechtfertigt, da explizit nur eine Instanz für den Workflow zur Verfügung stehen soll. Das Testen der Dateireferenzierungen und die Aggregation der Dateien werden durch präparierte XML-Dateien simuliert. So wird zum Beispiel der Fall der zirkulären Referenz durch den Test in Listing 6.10 abgedeckt.

```
1 EasyMock.expect(primitiveMock.getName()).andReturn("circle");
2 EasyMock.replay(primitiveMock);
3 utm = new UnitTypeMapper(
4     "de/genesez/typemapping/testmappings/circularReference1.xml")
        ;
5 assertEquals("ellipse", utm.getMappingName(primitiveMock));
```

Listing 6.10: Auszug aus der Klasse FileReferencingTest - Siehe FileReferencingTest.java Zeile 79-85 innerhalb des Sourcecode Ordners im de.genesez.typemapping Projekt

[15] Vergleiche [Wes06, S. 51ff]
[16] Siehe [GHJV96, S. 139], Es wird nur ein Objekt der Klasse instanziiert
[17] Siehe UnitTypeMapper.java auf der cd

Tests für die generierten JavaBeans wurden nicht entwickelt, da Selbige bisher nicht von Entwicklern manipuliert werden sollten. Die vorliegenden Unittests zielen weniger auf Code Coverage, als vielmehr auf das korrekte Verhalten der Klassen ab. Der Ansatz der verhaltensorientierten[18] Tests bietet den Vorteil, dass man die Tests als Beschreibung der Klasse lesen und verstehen kann. In diesem Zusammenhang lässt sich auch der Einsatz von EasyMock erklären, da diese API zum einen klassenunabhängiges und zum anderen seiteneffektfreies Testen zulässt. EasyMock setzt nicht die Kenntnis der Beziehungen zwischen den Klassen voraus und erleichtert den Einstieg in die Klassenstruktur.

[18] Siehe [Nor]

7 Resultat

7.1 Abschluss

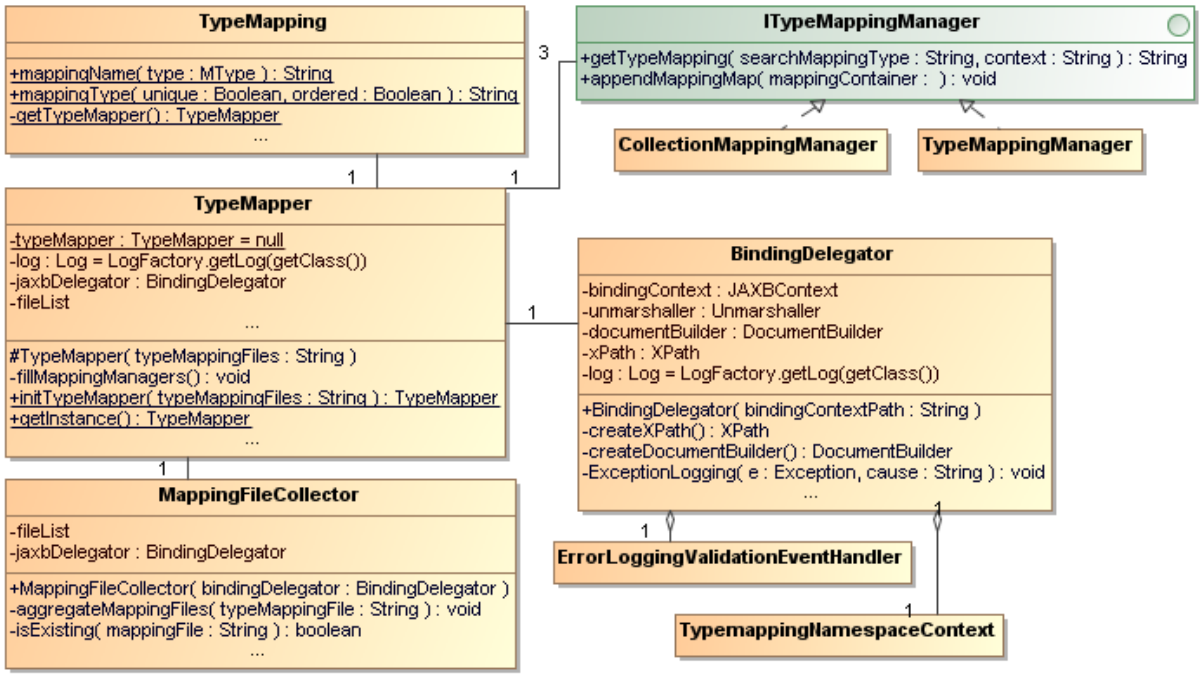

Abbildung 7.1: Resultierendes Klassendiagramm des Typemapping-Mechanismus

Einen Überblick gibt das resultierende Klassendiagramm in der Abbildung 7.1. Die Struktur der Lösung weist weiterhin Parallelen zur vorgegangenen Umsetzung auf. Speziell der Einsatz von Design Patterns ist aufgrund ihrer Vorteile übernommen worden. Der gravierendste Unterschied ist der Technologie zur Verarbeitung von XML-Daten geschuldet. JAXB bietet dem Programmierer bedeutend mehr Lösungsmöglichkeiten, die weniger Aufwand erfordern. Der implementierte Multi-Include-Mechanismus schließt sich daran an.

7.2 Fazit

Um XML-Daten zu verarbeiten, gibt es mittlerweile Unmengen an APIs, welche sich größtenteils in ihren Anwendungsgebieten unterscheiden. High-Level APIs werden auch in Zukunft Low-Level APIs nicht vollständig ablösen können, da diese weiterhin ihre Daseinsberechtigung besitzen, wie zum Beispiel SAXmit seiner Performanz. Allerdings werden Binding-APIs ein breiteres Spektrum abdecken, da Quelltext eingespart und somit Fehler vermieden werden können.

Generierte JavaBeans, wie sie von den meisten Binding-APIs zur Verfügung gestellt werden, bieten weiterhin den Vorteil, dass sie nicht von Programmierern direkt erzeugt werden müssen und somit ein fehlerfreies Generat darstellen. Natürlich ist unabhängig von den eingesetzten APIs die Verwendung von Design Patterns, Refactorings und Tests zwingend erforderlich, um die Qualität des resultierenden Produktes zu gewährleisten und es weiterhin les- und änderbar zu gestalten. Die Umsetzung des Typemappings verhält sich nach außen weiterhin wie zuvor, so dass der Workflow weitestgehend unbeeinflusst bleibt.

7.3 Ausblick

Neben dem eigentlichen Anforderungskatalog, welcher die Grundlage dieser Thesis bildet, gibt es weiterhin noch Möglichkeiten der Verbesserung oder Anpassung. Mit dem Einsatz von JAXB ist es problemlos möglich, einen Plugin für Eclipse zu erzeugen, um Typemapping Dateien basierend auf dem Schema zu erstellen. Da Binding APIs schemabasiert vorgehen, ist es möglich während des Serialisierens die zu erzeugende Datei zu validieren um zu prüfen, ob sie den Anforderungen gerecht wird. Die bisherigen Einschränkungen durch den Schema Compiler in Kombination mit ANT machen es noch nicht möglich Anpassungen am Generat vorzunehmen, wie es zum Beispiel im GeneSEZ Projekt möglich ist. Allerdings ist mit den zukünftigen Revisionen damit zu rechnen, dass diese Funktionalität realisiert wird. Basierend darauf wäre es möglich, mit `@XMLRootElement` annotierte JavaBeans auch aus dem vorliegenden Schema zu generieren, sowie das Implementieren von assoziativen Arrays zur Vereinfachung des Kontexthandlings. Des Weiteren ist durch das vorliegende JAXB-HowTo möglich, diese oder andere speziell JAXB betreffende Aufgaben ohne enormen Rechercheaufwand zu entwickeln.

Verzeichnis wichtiger Begriffe und Formate

Code Coverage beliebte Vorgehensweise und Indikator für Tests, welche daran gemessen wird wieviel Quelltext von den gesamten Tests abgedeckt wird

Code Smell Quelltext der Designfehler aufweist bzw. zum Fehlverhalten der Software führen kann oder einen Antipattern aufweist. Auch Lesbarkeit und Änderbarkeit können durch einen Code Smell leiden. Durch Refactorings können diese Quelltextstellen bereinigt werden

Fail Fast Fail Fast beschreibt die Eigenschaft von Komponenten, aufgetretene Probleme sofort zu melden und gegebenfalls den Programmablauf zu stoppen

Forward Engineering Entwicklung vom Modell zum eigentlichen Produkt.

GeneSEZ Generative Softwareentwicklung Zwickau. Projektgruppe des Fachbereichs Informatik der WHZ. Es geht darum den Ansatz der modellgetriebenen Softwareentwicklung für den Einsatz in der Praxis verfügbar zu machen.

JavaBean Wiederverwendbare Java Komponente. Sie bildet eine Art Container für Daten ab und weist immer öffentliche Getter und Setter Methoden, sowie mindestens einen öffentlichen Konstruktor auf und ist Serialisierbar. Das Konzept der Beans ist durch Sun spezifiziert

Mapping Abbilden eines Sachverhaltes auf einen anderen unter spezifischen Voraussetzungen

Middleware Anwendungsneutrale Programme die als Vermittler oder Verteiler agieren

Seiteneffekt	ungewollte Zustandsänderung, als Nebenwirkung einer Manipulation von einem oder mehreren Objekten oder Klassen
Trac	Webbasiertes Projektmanagementwerkzeug zur Softwareentwicklung
Typemapping	Abbilden von UML Typen auf programmiersprachenspezifische Datentypen
XML Binding	Mittels spezieller Serialisierer/Deserialisierer werden XML Daten als Objektgraphen repräsentiert und umgekehrt um direkt in der Programmierumgebung genutzt werden zu können
XML Processing	Parserbasierende XML Verarbeitungsart. In Java gibt es die Java API for XML Processing welche SAX, DOM, StAX sowie XSLT umfasst
Xpand	Statisch typisierte Templatesprache des open Architecture Ware Frameworks
XQuery	Stark typisierte Abfragesprache für XML Daten
Xtend	Funktionelle Sprache des open Architecture Ware Frameworks zum ändern und erweitern des Metamodelles

Verzeichnis der Abkürzungen

4GL	Fourth generation language
API	Application Programming Interface
CASE	Computer Aided Software Engineering
DOM	Document Object Model
DSL	domain-specific language
DTD	Document Type Definition
GeneSEZ	Generative Softwareentwicklung Zwickau
JAXB	Java Architecture for XML Binding
JAXP	Java API for XML Processing
MDSD	Model-Driven Software Development
oAW	open Architecture Ware
RELAX NG	Regular Language Description for XML New Generation
SAX	Simple API for XML
StAX	Streaming API for XML
UML	Unified Modelling Language
VTD-XML	Virtual Token Descriptor for eXtensible Markup Language

W3C World Wide Web Consortium

XAML eXtensible Application Markup Language

XPath XML Path Language

XSD XML Schema Definition

Literaturverzeichnis

[Dau08] Daum, Berthold: *Java-Entwicklung mit Eclipse 3.3*. dpunkt.verlag GmbH, 2008.

[Eub06] Eubanks, Brian D.: *Echt cooles Java*. Hanser Fachbuchverlag, 2006.

[FBB+99] Fowler, M., K. Beck, J. Brant, W. Opdyke und D. Roberts: *Refactoring*. Addison Wesley, 1999. Improving the Design of Existing Code.

[For07] Forbrig, Peter: *Objektorientierte Softwareentwicklung mit UML*. Hanser Fachbuchverlag, 2007.

[FPB08] Frederick P. Brooks, Jr.: *The Mythical Man-Month*. Addison-Wesley, 2008.

[GBB03] Grässle, P., H. Baumann und P. Baumann: *UML projektorientiert*. Galileo Press, 2003.

[GHJV96] Gamma, E., R. Helm, R. Johnson und J. M. Vlissides: *Entwurfsmuster*. Addison Wesley, 1996. Elemente wiederverwendbarer objektorientierter Software.

[HT99] Hunt, A. und D. Thomas: *The Pragmatic Programmer*. Addison Wesley, 1999.

[Kaw] Kawaguchi, Kohsuke: *Java Architecture for XML Binding (JAXB) 2.0*. http://jcp.org/en/jsr/detail?id=222,(16.01.2009).

[McC94] McConnell, Steve: *Code Complete*. Microsoft GmbH, 1994.

[McL02] McLaughlin, Brett: *Java and XML Data Binding*. O'Reilly Media, 2002.

[MS06] Michealis, S. und W. Schmiesing: *JAXB 2.0*. Hanser Fachbuchverlag, 2006. Ein Programmiertutorial für die Java Architecture for XML Binding.

[Nor] North, Dan: *DanNorth.net - Introducing BDD*. http://dannorth.net/introducing-bdd,15.01.2009.

[Ros08] Rosenberg, Scott: *Dreaming in Code*. Three Rivers Press, 2008.

[SV05] Stahl, T. und M. Völter: *Modellgetriebene Softwareentwicklung*. dpunkt.verlag GmbH, 2005. Techniken, Engineering, Management.

[SWK] Suttor, Jeff, Norman Walsh und Kohsuke Kawaguchi: *Java API for XML Processing (JAXP) 1.3*. http://www.jcp.org/en/jsr/detail?id=206,17.01.2009.

[Ull] Ullenboom, Christian: *Java ist auch eine Insel*. http://openbook.galileocomputing.de/javainsel7/,(11.12.2008).

[VV06] Vohra, A. und D. Vohra: *Pro XML Development with Java Technology*. Apress, 2006.

[Wes06] Westphal, Frank: *Testgetriebene Entwicklung mit JUnit und FIT*. dpunkt.verlag GmbH, http://www.frankwestphal.de, 2006. Wie Software änderbar bleibt.

Versicherung an Eides statt

Hiermit versichere ich, die vorliegende Bachelorthesis selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfmittel geschrieben zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Zwickau, 04. Februar 2009

Peter Huster

Thesen

Thema der Arbeit: Überarbeiten des Typemappings vom Genesez Projekt
Bearbeiter: Huster, Peter

- Durch den Typemapping-Mechanismus werden dem Entwickler basierend auf einem UML Modell mehrere programmierspezifische Klassengerüste zur Verfügung gestellt.

- Das Konzept der Mehrfachvererbung lässt sich neben Programmiersprachen auch auf das Einbinden von Konfigurationsdateien anwenden.

- Werden in einem Softwareprojekt XML-Daten herangezogen ist das Validieren unabdingbar.

- zur Validierung vom XML-Daten ist eine XSD einer DTD vorzuziehen.

- Die automatisierte Generierung von Quelltext kann zur Einschränkung der Flexibilität führen. Allerdings kann der ersparte Aufwand und die Zuverlässigkeit des Generats diesen Nachteil kompensieren.

- Das Anwenden von Design Patterns und Refactoring sollte schon während der Umsetzung des Projektes stattfinden - mit der Trennung zwischen Refactoring und der eigentlichen Entwicklung.

- Testgetriebene Entwicklung kann den Prozess der Softwareentwicklung beschleunigen und verbessern.