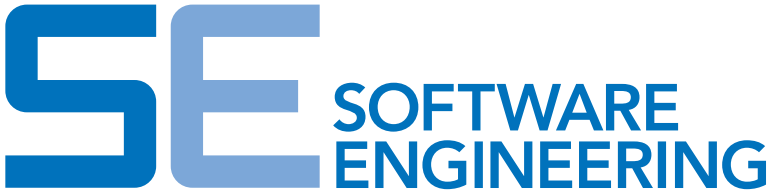
Rheinisch Westfälische Technische Hochschule Aachen
Lehrstuhl für Software Engineering



Masterarbeit

Framework für die Migration von XML zu problemadäquaten, lesbaren DSLs

Peter Damm

Aufgabenstellung: Prof. Dr. B. Rumpe
Betreuer: Dipl. Inf. (FH) M. Sc. Carsten Kolassa

Aachen, den 18. September 2014

Erklärung

Ich versichere, die vorliegende Arbeit selbständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Aachen, den 18. September 2014

Kurzfassung

In vielen Softwaresystemen werden XML-Formate zur Datenhaltung eingesetzt. Um eine effiziente Verarbeitung von Daten durch Menschen und Werkzeuge zu ermöglichen, ist ein adäquates Verhältnis zwischen Auszeichnungselementen und Daten wichtig. Durch ihren extrem hohen Anteil an Auszeichnungselementen, der oftmals 50 Prozent des Inhalts eines Dokumentes übersteigt, ist die XML nicht gut zur Datenhaltung geeignet. Im Rahmen dieser Arbeit wurde aus diesem Grund ein Framework zur Ablösung von XML-Formaten durch domänenspezifische Sprachen entwickelt. Diese sind kleiner und leichter verständlich. Die XML-Grammatiken werden durch das Framework in UML/P-Klassendiagramme und die XML-Dokumente in UML/P-Objektdiagramme transformiert. Auf Basis der Klassendiagramme erzeugt das Framework Java-Klassen und ermöglicht die Bindung der Daten aus den Objektdiagrammen an diese. Dadurch wird eine Schnittstelle zu bestehenden und neuen Softwaresystemen geschaffen. Die abschließende Evaluierung des Migrationsframeworks anhand eines industriell verwendeten XML-Formats zeigt, dass das Framework eine effiziente Ablösung vorhandener XML-Datenmodelle ermöglicht. Neben den Vorteilen im Kontext der Datenhaltung gegenüber XML-Formaten, lassen sich die erzeugten UML/P-Modelle außerdem sehr gut in modellgetriebene Softwareentwicklungsprozesse integrieren.

Abstract

Many software systems employ XML in order to describe data structures. Efficient processing of data within tools or by humans requires a good proportion between markup code and data. Due to its extremely high ratio of markup code, that often exceeds 50 percent of a document's content, the XML is not well suited for data storage. In this thesis a framework for replacement of XML-based languages by domain-specific languages, that are of reduced scope and thus more comprehensible, is introduced. It transforms existing XML grammars into UML/P class diagrams and the associated XML documents into UML/P object diagrams. Based on the class diagrams, the framework generates Java classes and allows their binding to the data embodied by the object diagrams. This way, a convenient interface for existing and future software systems is provided. The concluding evaluation of the framework, by migrating an XML format used in industry, shows that the framework allows the efficient replacement of existing XML data models that are used for data storage. As an additional benefit, the generated UML/P models can be directly integrated into model-driven software developement processes.

Aufgabenstellung

Der Einsatz von XML-Formaten zur Datenhaltung ist in der Industrie weit verbreitet. Ursprünglich zum Datenaustausch zwischen und zur Verarbeitung durch Computersystemen konzipiert, wird XML auch als Schnittstelle zwischen Menschen und Computern verwendet, etwa um Konfigurationsdaten zu pflegen. Obwohl XML menschenlesbar ist, ist die Lesbarkeit in den meisten Fällen schlecht. Das liegt an dem hohen Anteil von Auszeichnungselementen und der hierarchischen Struktur der Daten. Bei komplexeren Formaten verschlechtern Referenzierungen innerhalb dieser Struktur die Lesbarkeit zusätzlich.

Da XML eine sehr weitläufige Metasprache ist, werden mit Hilfe von Schemasprachen, wie zum Beispiel XSD, anwendungsspezifische XML-Formate definiert. Dies geschieht durch strukturelle und inhaltliche Einschränkungen, die zwar die Fehleranfälligkeit bei der manuellen Erstellung durch Syntaxprüfung reduzieren, die sprachbedingte, schlechte Lesbarkeit jedoch nicht verbessern.

Eine Alternative zu XML-Formaten stellen domänenspezifische Sprachen (Domain Specific Languages - DSLs) dar, die, ähnlich wie spezielle XML-Formate, für bestimmte Anwendungsfelder entworfen werden. Im Vergleich zu XML-Formaten sind DSLs aufgrund ihres begrenzten Umfangs (weniger Redundanz, weniger technischer Code) für Menschen jedoch leichter zu erlernen und anzuwenden. Auch die Erstellung von Werkzeugen zur Sprachverarbeitung von DSLs ist einfacher und unabhängig von weiteren Frameworks.

Im Rahmen der Arbeit soll ein Framework konzipiert und erstellt werden, das die automatische Generierung von DSL-Grammatiken aus vorhandenen XSD-Schemata erlaubt. Es soll die vollständige Ablösung von XML-Formaten durch DSLs, sowie die Migration der in XML vorhandenen Daten in Dokumente der DSL ermöglichen. Um eine komfortable Schnittstelle, beispielsweise für die Erstellung von Werkzeugen, zur Verfügung zu stellen, sollen die als DSL-Modelle vorhandenen Daten außerdem als Java-Objekte bereitgestellt werden.

Dazu werden die vorhandenen XSD-Schemata zunächst in UML/P-Modelle transformiert und in einem weiteren Schritt aus diesen Modellen entsprechende DSL-Grammatiken generiert. Abschließend soll das Framework anhand verschiedener, im industriellen Einsatz befindlicher und zur Ablösung diskutierter, XML-Dialekte evaluiert werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Kapitel 1

Einleitung

Seit ihrer Standardisierung im Jahr 1998 hat sich die XML aufgrund ihrer Anwendungsvielfalt in vielen Bereichen als Auszeichnungssprache durchgesetzt [www14t]. XML-Formate kommen dabei auch in den verschiedenen Schichten von Softwaresystemen sowie an diversen Stellen des Entwicklungsprozesses zum Einsatz. Die prominentesten Anwendungsszenarien sind die Verwendung als domänen- und plattformunabhängiges Format zur Datenpräsentation [Bos98], zum Datenaustausch [Cer02] und zur Datenverwaltung [CZR03].

Problematisch ist der Einsatz von XML-Formaten in Bereichen, in denen die XML-Daten direkt von Menschen bearbeitet werden und somit als eine Benutzerschnittstelle des jeweiligen Systems dienen. XML-Dokumente sind zwar menschenlesbar und in der Regel selbstbeschreibend, dennoch sind sie im Vergleich zu Dokumenten domänenspezifischer Sprachen umfangreicher. Das führt zu einer schlechteren Verständlichkeit, die wiederum die Handhabung erschwert. Die Erstellung von korrekten XML-Dokumenten durch einen Menschen ist daher ebenso unpräzise und fehleranfällig wie die Erstellung eines reinen Textdokumentes, da die Daten sowie Auszeichnungselemente manuell erstellt werden müssen.

Ein weiterer Nachteil der XML ist, dass ihre Menschenlesbarkeit erst durch die umfangreiche Auszeichnung der Daten erreicht [Law04] und damit eine effiziente Speicherung, Übertragung und Verarbeitung [LDL08] der Daten erschwert wird. Die Fehleranfälligkeit bei der Verwendung als Benutzerschnittstelle lässt sich zwar durch die Verwendung von speziellen Editoren und XML-Grammatiken auf die Korrektheit der Daten beschränken, die Schwächen bei Lesbarkeit und Verbosität werden dadurch jedoch nicht kompensiert.

Die Herausforderung dieser Arbeit besteht darin, ein Migrationsframework zu konzipieren, das die vollständige Ablösung von XML-Formaten, die zur Datenhaltung verwendet werden, durch domänenspezifische Sprachen ermöglicht. Dabei sollen alle Informationen der Grammatik und alle XML-Daten erhalten bleiben sowie durch die Bindung der Dokumente der domänenspezifischen Sprache an Java-Objekte eine Anwendungs- und Werkzeugschnittstelle entstehen.

1.1 Verwandte Arbeiten

In diesem Abschnitt werden verschiedene Frameworks und weitere Arbeiten vorgestellt, die im Themenbereich dieser Arbeit angesiedelt sind. Dabei soll festgestellt werden, ob die existierenden Ansätze alle oder zumindest Teile der Ziele des angestrebten Migrationsframeworks umsetzen können und, ob es sinnvoll ist, Konzepte dieser Frameworks bei der Implementierung des Migrationsframeworks aufzugreifen.

Da keine Ansätze existieren, die eine vollständige Ablösung von XML ermöglichen und anschließend Schnittstellen zur Verwendung des neuen Datenformats bereitstellen, werden in den folgenden Unterabschnitten verwandte Arbeiten zu diesen beiden Hauptfunktionen des Migrationsframeworks getrennt betrachtet.

1.1.1 Migration von XML-Grammatiken

Das primäre Ziel des Frameworks ist die Migration von bestehenden XML-Grammatiken und den dazugehörigen XML-Daten in anwendungsspezifische, besser lesbare Datenformate. Es existieren zahlreiche Frameworks, die verschiedene Datenformate anhand bestehender XML Schema Definitionen erzeugen. Es erfüllen jedoch nicht alle die Anforderungen einer besseren Lesbarkeit und Weiterverarbeitung.

Migration in UML-Diagramme

Wie in Abschnitt 2.2.1 anhand eines Beispiels gezeigt wird, sind UML-Klassendiagramme prinzipiell geeignet, um die Informationen aus XML-Grammatiken darzustellen. Die zugehörigen XML-Daten können dann in Form von Objektdiagrammen gespeichert werden. Die Lesbarkeit von UML-Modellen hängt dabei in erster Linie von der Art ihrer textuellen Repräsentation ab. Visualisiert sind sie aufgrund des höheren Abstraktionsniveaus verständlicher als graphische Darstellungen von XML-Grammatiken und Daten.

Zur Transformation von XML-Grammatiken in UML-Modelle existieren verschiedene Ansätze [BKK04, RBG02] und Implementierungen [www14w]. Problematisch ist dabei jedoch die Weiterverwendung der erzeugten Modelle im Sinne der Aufgabenstellung. Verglichen mit Modellen der UML/P (vgl. Abschnitt 2.2.1) erlauben die jeweiligen Ansätze durch die Verwendung von speziell zu diesem Zweck entworfenen Sprachprofilen nur eine eingeschränkte und weniger flexible Weiterverwendung. Einige sehen sogar die textuelle Notation der Modelle in Form von XML-Dokumenten vor. Eine derartige Transformation bietet im Kontext des angestrebten Migrationsframeworks keinen Mehrwert zum Ausgangszustand und ist daher nicht im Sinne der Aufgabenstellung dieser Arbeit.

Nicht alle der Ansätze zur Migration in UML-Diagramme sehen eine Migration der eigentlichen XML-Daten vor. Für die Bindung der migrierten Daten an Java-Objekte wäre außerdem der Einsatz weiterer Frameworks erforderlich. Dieser wird durch die Verwendung der zuvor erwähnten, spezifischen Sprachprofile zusätzlich erschwert. Dennoch können die vorhandenen Ansätze nützliche Konzepte für die Transformation in Modelle der UML/P enthalten. Vorteilhaft ist außerdem, dass die erzeugten UML-Modelle im Rahmen modellgetriebener Entwicklungsprozesse verwendet werden können.

Migration in JSON Schema

Die *JavaScript Object Notation* (JSON) [www14l] ist ein Datenaustauschformat, das in diesem Anwendungsbereich mit der XML konkurriert. Sie ist zwar nicht so vielfältig einsetzbar wie die XML, bietet aufgrund ihrer Fokussierung auf den Datenaustausch als Anwendungsdomäne jedoch eine bessere Lesbarkeit bei geringerem Overhead, da die Auszeichnung der Werte einfach gehalten ist. Aus diesen Gründen ist JSON ein möglicher Kandidat als Zielsprache für das Migrationsframework.

JSON Schema [www14m] erlaubt die Definition von anwendungsspezifischen JSON-Formaten, indem Art und Struktur der Daten vorgeschrieben werden. Anhand der so erstellten Grammatik lassen sich JSON-Dokumente validieren, um eine fehlerfreie Verarbeitung durch entsprechende Anwendungen zu ermöglichen.

[Nog13] stellt einen Ansatz zur Transformation von XML-Grammatiken in JSON-Grammatiken vor. Dabei werden jedoch keine Lösungen zur konformen Transformation der XML-Daten in JSON-Daten beziehungsweise zu deren anschließender Datenbindung vorgestellt. Der Ansatz von [Lee11] implementiert die bidirektionale Transformation von Grammatiken und Daten. Durch die Verwendung von *XSLT* [www14x] als Transformationssprache werden jedoch neue XML-Artefakte erzeugt, die für die Migration der XML-Daten erforderlich sind. Eine vollständige Ablösung von XML ist damit durch diesen Ansatz ebenfalls nicht möglich.

Migration in relationale Datenbankschemata

Ein weit verbreitetes Vorgehen bei der Ablösung von XML-Formaten ist die Erzeugung von relationalen Datenbankschemata. Dazu existieren verschiedene Ansätze [PLMC03, BFRS02], die auch die Migration der XML-Daten in relationale Daten ermöglichen. Außerdem sind seit der Etablierung von Objektrelationalen Abbildungen [BS98] bereits Schnittstellen zu vielen Programmiersprachen vorhanden. Der Nachteil bei diesem Ansatz ist, dass die migrierten Datensätze nicht in Dateiform vorliegen und somit eine Verarbeitung ähnlich der XML-Daten nicht möglich ist. Das erschwert beispielsweise den Datenaustausch und macht eine effiziente Weiterverwendung vorhandener Anwendungssoftware unmöglich.

1.1.2 Datenbindung

Neben der Migration von XML-Formaten soll auch die Bindung der transformierten XML-Daten an Java-Objekte durch das Framework implementiert werden. Für die direkte XML-Datenbindung gibt es verschiedene Ansätze und Frameworks, die unterschiedlich stark von den verwendeten XML-Formaten abstrahieren.

Die geringste Abstraktion bieten dabei Schnittstellen, die über das *Document Object Model* [www14d] Zugriff auf XML-Dokumente ermöglichen. Andere Ansätze ermöglichen die direkte Integration von XML-Abfragen in Anwendungscode, indem Programmiersprachen durch entsprechende Schnittstellen erweitert werden [HRS+05, MBB06]. Die Datenbindung durch die Generierung von Klassenrepräsentationen, die nach einem festen Schema aus dem XML-Format abgeleitet werden, bietet ein gutes Abstraktionsniveau und wird von weit verbreiteten Frameworks umgesetzt [www14v, www14n]. Das *JiBX*-Framework

[www14k] erlaubt außerdem eine benutzerdefinierte Datenbindung, da die Zuordnungen zwischen XML-Strukturen und Datenbindungsklassen durch den Anwender definiert werden können.

Da XML-Formate zumeist über eine, durch ein Datenbindungsframework erzeugte, Klassenrepräsentation in Anwendungen integriert werden, ist es wichtig durch das Migrationsframework ein ähnliches Abstraktionsniveau bei der Bindung der Daten zu erreichen.

1.1.3 Fazit

Die Analyse von verwandten Arbeiten und Implementierungen hat gezeigt, dass bisher kein ganzheitlicher Ansatz zu Ablösung von XML-Formaten im Sinne der Aufgabenstellung existiert. Als weiteres Ergebnis wurde festgestellt, dass die Migration von Grammatik und Daten in UML-Modelle die Integration vorhandener Datenmodelle in modellgetriebene Entwicklungsprozesse ermöglichen würde. Bei der Datenbindung ist auf ein geeignetes Abstraktionsniveau zu achten, das die effiziente Migration von bereits bestehenden Anwendungen ermöglicht.

1.2 Überblick über die Arbeit

Die Arbeit gliedert sich in vier Teile, deren Inhalt im Folgenden kurz beschrieben wird.

Im vorhergehenden Abschnitt dieses Kapitels wurde das Ziel der Arbeit anhand verschiedener, existierender Ansätze zur XML-Datenbindung und -Migration motiviert. Anschließend werden in Kapitel 2 die benötigten Grundlagen aufbereitet und die im Rahmen der Arbeit verwendeten Technologien und Standards vorgestellt, bevor die ergebnisorientierten Kapitel der Arbeit folgen.

In Kapitel 3 wird die Aufgabenstellung analysiert und in verschiedene Teilaufgaben unterteilt. Aus diesen werden Anforderungen an das Migrationsframework ermittelt und formuliert, anhand derer im weiteren Verlauf der Arbeit Lösungen für die Aufgabenstellung erarbeitet werden. Kapitel 4 analysiert mögliche Lösungsansätze für die geforderten Funktionen und Eigenschaften des Frameworks, die aus den Anforderungen resultieren. Dabei wird ein Gesamtkonzept zum Aufbau des Frameworks entwickelt sowie Strategien für die Implementierung der einzelnen Komponenten erarbeitet.

Kapitel 5 beschreibt die im Rahmen der Arbeit entstandene Implementierung des Migrationsframeworks. Zum Abschluss des Kapitels wird anhand eines Beispiels beschrieben, wie XML-Formate mit Hilfe des Frameworks durch eine problemadäquate, lesbare DSL abgelöst werden können und wie die Datenbindung funktioniert.

Der letzte Teil der Arbeit befasst sich mit der Bewertung und Zusammenfassung der Ergebnisse. In Kapitel 6 werden die Ergebnisse von Konzeption und Implementierung anhand eines, im industriellen Einsatz befindlichen, XML-Dialekts evaluiert. Kapitel 7 fasst die Ergebnisse der vorhergehenden Kapitel mit dem Ziel, Themenbereiche und Aufgaben für weiterführende Arbeiten aufzuzeigen, zusammen.

Kapitel 2

Grundlagen

In diesem Kapitel werden die der Arbeit zu Grunde liegenden Technologien vorgestellt und in deren Kontext verwendete Begriffe erläutert.

2.1 Extensible Markup Language

Die *Extensible Markup Language* (XML) [www14e] ist eine Auszeichnungssprache, welche die Darstellung von hierarchisch strukturierten Daten in menschen- und maschinenlesbaren Textdokumenten ermöglicht. Die erste Version des XML-Standards wurde ab 1996 entwickelt, wobei Einfachheit und Allgemeingültigkeit als Entwurfsziele im Vordergrund standen [www14g]. Anfang 1998 erreichte XML den Status einer W3C-Empfehlung [www14f].

XML-Syntax

Quelltext 2.1 zeigt beispielhaft die Darstellung einer Rechnung in Form eines XML-Dokumentes. Die einzelnen Textzeichen eines XML-Dokumentes werden in Auszeichnungssprache und Inhalt unterteilt, woraus sich die logische Struktur des Dokumentes ableiten lässt.

Ein XML-Dokument sollte mit einer XML-Deklaration beginnen, welche die verwendete Version des XML-Standards sowie die Zeichenkodierung des Dokumentes festlegt. Die XML-Deklaration beginnt mit den Zeichen `<?xml` und wird durch die Zeichen `?>` geschlossen. Sie ist jedoch kein zwingender Bestandteil eines Dokumentes. Ist keine XML-Deklaration vorhanden, wird Version 1.0 für den XML-Standard und UTF-8 als Zeichenkodierung angenommen. Die XML-Deklaration für das Beispieldokument befindet sich in Zeile 1.

Die wichtigsten Auszeichnungskonstrukte sind XML-Tags. Sie kennzeichnen die zentralen Struktureinheiten eines XML-Dokumentes, die XML-Elemente. Jedes Tag beginnt mit einem < Zeichen und endet mit einem > Zeichen. Es wird zwischen drei Arten von Tags unterschieden:

- *Start-Tags* zeichnen den Beginn eines XML-Elements aus. Nach dem < Zeichen folgt der Name des Elements. Optional enthält ein Start-Tag nach dem Namen eine durch

Leerzeichen getrennte Liste von Eigenschaftszuweisungen, sogenannten Attributen. Eine Eigenschaftszuweisung besteht aus dem Namen des Attributs gefolgt von einem = Zeichen und dem zuzuweisenden Wert in Anführungszeichen. Danach wird das Start-Tag durch ein > Zeichen geschlossen. Ein Beispiel hierfür ist das Start-Tag des *artikel*-Elements in Zeile 15, welches das Attribut *artikelnummer* enthält. Auf das Start-Tag folgt der Inhalt des XML-Elements, welcher aus beliebigen Auszeichnungskonstrukten besteht. Bei dem *artikel*-Element besteht der Inhalt aus vier weiteren XML-Elementen.

- *End-Tags* zeichnen das Ende eines XML-Elements aus. Sie beginnen mit der Zeichenfolge </. Darauf folgt der Name des Elements sowie das > Zeichen, welches das End-Tag schließt. Das End-Tag des *artikel*-Elements befindet sich in Zeile 22.

- *Empty-Element-Tags* stellen eine syntaktische Alternative dar, um leere XML-Elemente auszuzeichnen. Anstatt direkt aufeinander folgender Start- und End-Tags, wie bei dem *zusatz*-Element in Zeile 9, kann ein Empty-Element-Tag verwendet werden. Wie beim Start-Tag folgt auf das < Zeichen der Elementname und optional eine Attributliste. Geschlossen wird es durch die Zeichenfolge />. Ein Beispiel ist das *versand*-Element in Zeile 23.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <rechnung rechnungsnummer="2014-9102" datum="19.09.2014">
3    <kunde>
4      <vorname>Max</vorname>
5      <nachname>Mustermann</nachname>
6      <anschrift>
7        <strasse>Musterweg</strasse>
8        <hausnummer>321</hausnummer>
9        <zusatz></zusatz>
10       <plz>12345</plz>
11       <stadt>Musterstadt</stadt>
12     </anschrift>
13   </kunde>
14   <posten>
15     <artikel artikelnummer="0IC1300">
16       <anzahl>4</anzahl>
17       <einzelpreis>105.00</einzelpreis>
18       <gesamtpreis>420.00</gesamtpreis>
19       <beschreibung>
20         Musterartikel XYZ, EAN Nummer: 4043006189288
21       </beschreibung>
22     </artikel>
23     <versand versandart="DHL" kosten="0.00"/>
24   </posten>
25 </rechnung>
```

Quelltext 2.1: Rechnungsdaten als XML-Dokument

XML-Struktur

Abbildung 2.1 zeigt die durch XML-Auszeichnungskonstrukte definierte Struktur der Daten aus Quelltext 2.1. Durch die Verschachtelung von Elementen wird eine Hierarchie der Daten

erzeugt, welche sich als Baum darstellen lässt. Aus dieser Art der Darstellung werden Begriffe abgeleitet, um die Beziehung zwischen verschiedenen XML-Elementen auszudrücken. Als *Wurzelelement* wird das Element auf der obersten Ebene des Dokumentes bezeichnet. Die unmittelbar untergeordneten Elemente eines Elements werden *Kinder* genannt. Hierzu zählen auch die Attribute oder Textinhalte eines Elements. So hat das *kunde*-Element in Zeile 3 die Kindelemente *vorname*, *nachname* und *anschrift*. Umgekehrt ist das *kunde*-Element das Elternelement dieser drei Elemente. Das *vorname*-Element wiederum hat nur ein Text-Element als Kind. Außer dem Wurzelelement verfügt jedes Element über genau ein Elternelement. Alle übergeordneten Elemente eines Elements werden als Vorfahren, alle untergeordneten Elemente als Nachkommen bezeichnet. So ist jedes Element ein Nachkomme des Wurzelelements.

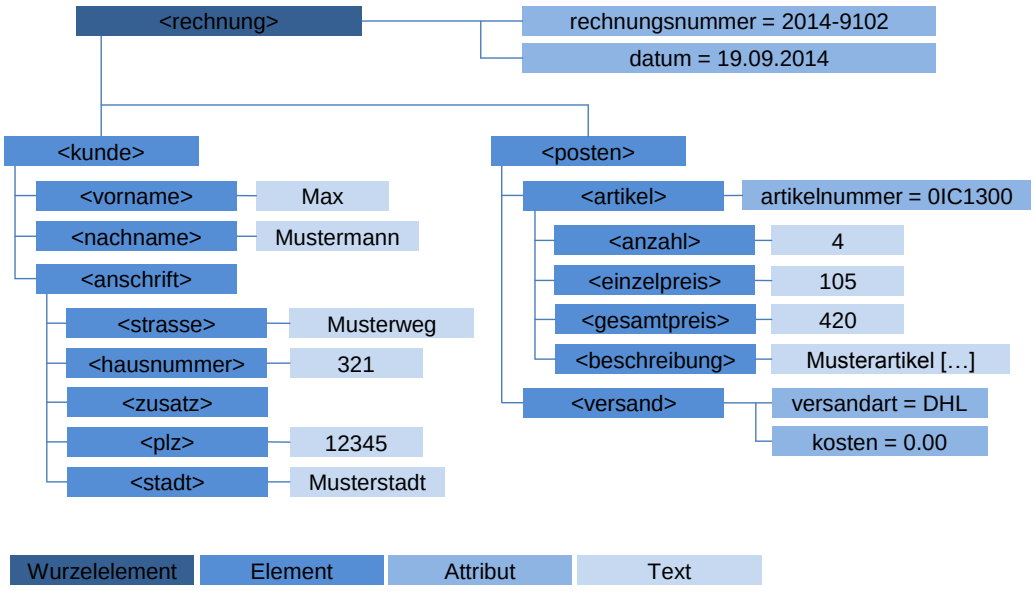


Abbildung 2.1: Baumdarstellung des in Quelltext 2.1 definierten XML-Dokumentes

Wohlgeformtheit

Der XML-Standard macht keine Vorgaben zum Inhalt oder der Hierarchie und den verwendeten Attributen der ausgezeichneten Elemente eines XML-Dokumentes. Er definiert lediglich Regeln zum korrekten Aufbau von XML-Dokumenten. Ein XML-Dokument, das allen Regeln des Standards genügt, wird wohlgeformt genannt.

Das XML-Dokument in Quelltext 2.2 enthält beispielhafte Auszeichnungskonstrukte, die gegen einige dieser Regeln verstoßen. So besitzt es zunächst kein eindeutiges Wurzelelement, sondern gleich drei Elemente auf der obersten Ebene. Zwei dieser Elemente sind außerdem nicht korrekt aufgebaut. Das Start- und End-Tag des *b*-Elements sind in unterschiedlicher Groß- und Kleinschreibung notiert und das *c*-Start-Tag verfügt über kein korrespondierendes End-Tag. Die Kindelemente *a1* und *a2* des *a*-Elements sind nicht ebenentreu-paarig verschachtelt. Entweder ist *a2* ein Kind von *a1*, dann müsste das *a2*-End-Tag vor dem *a1*-End-Tag stehen, oder beide sind Kinder von *a*, dann müsste das *a1*-End-Tag vor dem

```
<?xml version="1.0" encoding="UTF-8"?>
<a>
  <a1>
  <a2>
  </a1>
  </a2>
  <a3 f="200" g="300" f="400"/>abcd1234</a3>
</a>
<b>
</B>
<c>
```

Quelltext 2.2: Nicht-wohlgeformtes XML-Dokument

a2-Start-Tag ausgezeichnet werden. Das *a3*-Element beginnt mit einem Empty-Element-Tag, hat jedoch trotzdem einen Textinhalt und ein entsprechendes End-Tag. Außerdem verfügt es über zwei Attribute mit dem Namen *f*, was ebenfalls nicht zulässig ist.

Ohne Vorgaben zum Inhalt und den verwendeten XML-Elementen ist die automatische Verarbeitung eines XML-Dokumentes sowie dessen Erstellung durch einen Menschen nicht wesentlich einfacher oder präziser als die Verwendung von reinen Textformaten. Wäre beispielsweise die Struktur des XML-Dokumentes aus Quelltext 2.1 nicht festgelegt, hätte die Auszeichnung der Daten durch XML-Tags und -Attribute nur einen geringen Mehrwert für deren automatische Extrahierung und Überführung in Objektinstanzen einer Buchhaltungssoftware. Selbst wenn die Struktur durch eine Spezifikation festgelegt ist, können bei der Bearbeitung durch einen Menschen, wegen der umfangreichen Auszeichnung, leicht Fehler wie falsche Attribut- oder Elementnamen entstehen.

Aus diesem Grund lassen sich mit Hilfe sogenannter Schemasprachen Grammatiken für XML-Dokumente erstellen. Durch sie können unter anderem die zu verwendenden XML-Elemente, deren Verschachtelung und Attribute sowie die Art des Inhalts von Elementen und Attributen festgelegt werden. Beispiele hierfür sind die innerhalb des XML-Standards beschriebene *Dokumententypdefinition* (DTD) und die, im Rahmen dieser Arbeit betrachtete, *XML Schema Definition*.

2.1.1 XML Schema Definition

Mit der *XML Schema Definition* (XSD) [www14u] lassen sich Struktur, Inhalt und Semantik von XML-Dokumenten definieren. Eine XML Schema Definition wird selbst in Form eines XML-Dokumentes definiert und erlaubt im Vergleich zu DTD eine detailliertere Spezifikation der Dokumentenstruktur und verwendeten Datentypen.

Quelltext 2.3 zeigt ein XSD-Schema, das Rechnungsdokumente in der Form des Beispiels aus Quelltext 2.1 definiert. Das Schema beginnt, als XML-Dokument, zunächst mit einer XML-Deklaration. Das Wurzelelement jedes XSD-Schemas ist ein *schema*-Element, durch dessen Attribute hauptsächlich die Namensräume des Schemas verwaltet werden. Im Beispiel ist nur ein Attribut vorhanden. Es gibt an, dass die innerhalb der Schemadeklaration verwendeten Elemente und Datentypen aus dem Namensraum `http://www.w3.org/2001/XMLSchema` stammen und als Präfix `xs:` tragen.

Die Kindelemente des *schema*-Elements definieren die Elemente und Datentypen des Sche-

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="rechnung" type="rechnungtype"/>
  <xs:complexType name="rechnungtype">
    <xs:sequence>
      <xs:element name="kunde" type="kundetype"/>
      <xs:element name="posten" type="postentype"/>
    </xs:sequence>
    <xs:attribute name="rechnungsnummer" type="xs:string"/>
    <xs:attribute name="datum" type="xs:string"/>
  </xs:complexType>
  <xs:complexType name="kundetype">
    <xs:sequence>
      <xs:element name="vorname" type="xs:string"/>
      <xs:element name="nachname" type="xs:string"/>
      <xs:element name="anschrift" type="anschrifttype"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="postentype">
    <xs:sequence>
      <xs:element name="artikel" type="artikeltype"
        ↪ maxOccurs="unbounded"/>
      <xs:element name="versand" type="versandtype"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="anschrifttype">
    <xs:sequence>
      <xs:element name="strasse" type="xs:string"/>
      <xs:element name="hausnummer" type="xs:int"/>
      <xs:element name="zusatz" type="xs:string"/>
      <xs:element name="plz" type="xs:int"/>
      <xs:element name="stadt" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="artikeltype">
    <xs:sequence>
      <xs:element name="anzahl" type="xs:int"/>
      <xs:element name="einzelpreis" type="xs:decimal"/>
      <xs:element name="gesamtpreis" type="xs:decimal"/>
      <xs:element name="beschreibung" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="artikelnummer" type="xs:string"/>
  </xs:complexType>
  <xs:complexType name="versandtype">
    <xs:attribute name="versandart" type="xs:string"/>
    <xs:attribute name="kosten" type="xs:decimal"/>
  </xs:complexType>
</xs:schema>
```

Quelltext 2.3: Beispielhafte XML Schema Definition für Rechnungsdokumente im Format des Beispiels aus Quelltext 2.1

mas, also Struktur und Inhalt der XML-Dokumente, die dem Schema genügen. Dazu beinhaltet der XML Schema Namensraum eine große Anzahl von unterschiedlichen Element- und Datentypen, die auf verschiedene Arten verschachtelt werden können.

Zentral sind dabei die *element*-Elemente. Jedes dieser Elemente definiert ein erlaubtes Element in Dokumenten des Schemas. Über das Attribut *name* wird der Elementname festgelegt, über das Attribut *type* der Inhaltstyp. Falls ein *element*-Element das einzige *element*-Kindelement des *schema*-Elements ist, beschreibt es das Wurzelelement für die XML-Dokumente des Schemas. Ein solches *element*-Element befindet sich in Zeile 3 des Beispielschemas aus Quelltext 2.3. Die Wurzelelemente des Schemas tragen den Elementnamen *rechnung* und haben den Inhaltstyp *rechnungtype*.

Die unterschiedlichen Arten von Inhalts- beziehungsweise Datentypen, die auf verschiedene Weisen definiert werden können, werden in den nächsten beiden Unterabschnitten vorgestellt.

Einfache Inhaltstypen

Einfache Inhaltstypen definieren atomare XML-Elemente, die weder eigene XML-Kindelemente, noch XML-Attribute besitzen dürfen. Der einzig zulässige Inhalt ist ein Text-Element, dessen Wert den Einschränkungen des entsprechenden Datentyps genügt. Durch XML-Schema stehen einige, auch in anderen Typsystemen verbreitete, grundlegende Datentypen sowie von diesen abgeleitete Datentypen zur Verfügung. Ein Beispiel für einen grundlegenden Datentypen ist der Datentyp *xs:decimal*, der die Teilmenge der Reellen Zahlen, die durch Dezimalzahlen dargestellt werden können, als Wert zulässt.

Ein Beispiel für einen abgeleiteten Datentypen ist der Typ *xs:int*, der in Quelltext 2.3 unter anderem als Inhaltstyp für *anzahl*-Elemente definiert wird (Zeile 36). Aus dem Typ *xs:decimal* leitet sich zunächst der Typ xs:integer ab, indem die maximale Anzahl von Nachkommastellen auf 0 beschränkt und die Verwendung von Dezimaltrennzeichen verboten wird. Aus *xs:integer* leitet sich der Typ *xs:long* ab, dessen Wertebereich auf das Intervall [-9223372036854775808; 9223372036854775807] eingeschränkt ist. Durch eine weitere Einschränkung auf das Intervall [-2147483648; 2147483647] wird letztendlich der Typ *xs:int* abgeleitet.

Außer durch Einschränkungen des Wertebereichs lassen sich einfache Inhaltstypen auch durch Auflistungen und Vereinigungen ableiten. Bei der Ableitung durch Auflistung ist der zulässige Inhalt des Text-Elements eine endliche Sequenz von Werten, die den Einschränkungen des Datentyps, von dem abgeleitet wurde, entsprechen. Durch Vereinigung von mehreren Datentypen kann ein neuer Datentyp abgeleitet werden, indem ihre Wertebereiche durch den Wertebereich des neuen Datentyps vereinigt werden.

Komplexe Inhaltstypen

Komplexe Inhaltstypen definieren XML-Elemente, die neben Text-Elementen auch andere XML-Elemente als Kinder beinhalten sowie XML-Attribute besitzen dürfen. Ab Zeile 4 in Quelltext 2.3 wird der komplexe Inhaltstyp *rechnungtype*, der, wie bereits erwähnt, den Inhaltstypen des Wurzelelements darstellt, definiert. Kindelemente von komplexen Inhaltstypen können auf verschiedene Arten kombiniert werden. Über das *xs:sequence*-Element wird eine Auflistung von Kindelementen definiert, die in der gegebenen Reihenfolge auftreten müssen.

Über die Attribute *minOccurs* und *maxOccurs* kann die Anzahl der Elemente in der Auflistung vorgeschrieben werden. Sind diese Attribute nicht vorhanden, muss das Element genau einmal vorkommen. Elemente vom Typ *rechnungtype* haben also jeweils ein Kindelement mit Elementnamen *kunde* und Elementtyp *kundetype* und ein Kindelement mit Elementnamen *posten* und Elementtyp *postentype*. Über *xs:choice*-Elemente wird eine Liste von möglichen Kindelementen definiert, von denen jedoch nur eines auftreten darf. Außerdem verfügen Elemente vom Typ *rechnungtype* über die Attribute *rechnungsnummer* und *datum*, welche beide Werte des Typs *xs:string* annehmen dürfen.

Elementspezifische Attribute

Grundsätzlich können die XML-Elemente innerhalb einer XML Schema Definition beliebige Attribute tragen. Für jeden XSD-Elementtyp gibt es jedoch Attribute mit einer semantischen Bedeutung für das definierte Schema. Dies wurde im letzten Abschnitt beispielsweise an den *name*- und *type*-Attributen der *element*-Elemente deutlich. Sie haben im weiteren Verlauf der Arbeit, aufgrund ihrer semantischen Bedeutung, einen hohen Stellenwert bei der Analyse der XML Schema Definitionen und werden unter dem Begriff *elementspezifische Attribute* zusammengefasst.

Validität

Neben der Wohlgeformtheit gibt es im XML-Standard noch eine weitere Eigenschaft, die XML-Dokumente erfüllen können, die Validität. Ein XML-Dokument ist valide, wenn es:

- wohlgeformt ist,
- auf eine Grammatik, also beispielsweise auf eine XML Schema Definition, verweist,
- und den Regeln dieser Grammatik genügt.

Die Validität ist daher die Kondition, über welche die Zugehörigkeit eines XML-Dokumentes zu einer bestimmten XML-Grammatik evaluiert werden kann.

2.2 Domänenspezifische Sprachen

Eine domänenspezifische Sprache (Domain Specific Language, DSL) [Vö11] ist eine formale Sprache, die mit dem Ziel, Aufgaben eines spezifischen Anwendungsgebietes lösen zu können, entwickelt wird. Im Gegensatz zu universellen Programmiersprachen soll eine DSL durch ihre Problemspezifität, auch ohne ausgeprägtes Zusatzwissen im Softwarebereich, von Experten des jeweiligen Anwendungsgebietes verwendet werden können. Daher ist ein grundlegendes Ziel beim Entwurf von DSLs, den Umfang der Sprache auf die Elemente der Sprache zu begrenzen, die zur Darstellung der Probleme der Domäne benötigt werden [vDKV00].

Aufgrund des begrenzten Sprachumfangs ergeben sich neben der leichten Erlernbarkeit weitere Vorteile wie eine bessere Lesbarkeit der Dokumente der DSL, keine oder nur sehr

geringe Anteile technischen Codes sowie die Möglichkeit, Werkzeuge für die Sprache zu generieren. Beispiele für Werkzeuge, die dem Domänenexperten die Arbeit mit der Sprache weiter erleichtern, sind sprachspezifische Editoren, welche Dokumente der Sprache validieren und wiederkehrende Aufgaben der Sprachverarbeitung automatisieren sowie Werkzeuge zur Visualisierung von Dokumenten der Sprache. Im folgenden Unterabschnitt wird die UML/P, als eine Sammlung von domänenspezifischen Sprachen, vorgestellt.

2.2.1 UML/P

Die UML/P ist ein Sprachprofil der UML [www14r], das zur Unterstützung der Tätigkeiten Entwurf, Implementierung und Weiterentwicklung im Softwareentwicklungsprozess entworfen wurde [Rum11, Rum12]. Sie erlaubt die Integration von Java-Code in verschiedene Modelltypen der UML, deren Semantik und Umfang im Hinblick auf die oben genannten Aktivitäten optimiert wurden.

Die einzelnen Modelltypen beziehungsweise Teilsprachen der UML/P stellen somit, wenn auch recht umfangreiche, domänenspezifische Sprachen für die Lösung von Modellierungsproblemen im Rahmen der jeweiligen Tätigkeit im Softwareentwicklungsprozess dar.

In [Sch12] wurde eine textuelle Fassung der UML/P mit umfassender Werkzeuginfrastruktur entwickelt, die eine praktische Nutzung der UML/P in Softwareprojekten ermöglicht und auf die bei der Implementierung des Migrationsframeworks zurückgegriffen wurde. Im Rahmen dieser Arbeit sind zwei Modelltypen der UML/P von Relevanz, die im Folgenden als Beispiele domänenspezifischer Sprachen vorgestellt werden.

Klassendiagramme

Mit Hilfe von Klassendiagrammen lässt sich die Struktur von Softwaresystemen und Datenmodellen charakterisieren. Innerhalb der so definierten Struktur lassen sich mit Hilfe der anderen Diagrammtypen der UML/P Eigenschaften, Verhalten und Daten des Systems beschreiben.

Die Notation von Klassendiagrammen entspricht dabei der Sichtweise der objektorientierten Programmierung auf das System. Mit ihrer Hilfe werden Attribute als Klassen zusammengefasst und gekapselt, die die gültige Struktur der Objekte des Systems definieren. Das System selbst wird als Gesamtmenge dieser Objekte gesehen.

Im Rahmen der Arbeit werden nicht alle Sprachelemente und -konzepte der UML/P-Klassendiagramme benötigt, da die modellierten Datenformate alle eine ähnliche Struktur aufweisen. Im Folgenden werden daher anhand eines Beispiels die verwendeten textuellen Sprachelemente vorgestellt. Quelltext 2.4 zeigt die textuelle Darstellung eines UML/P-Klassendiagramms, das die hierarchische Struktur der Daten des XML-Dokumentes aus Quelltext 2.1 repräsentiert.

Durch das Schlüsselwort `class`, gefolgt von einem eindeutigen Namen, werden verschiedene Klassen, die Teile der Datenhierarchie abbilden, deklariert. Es folgt eine von geschweiften Klammern umfasste und durch Semikolon getrennte Liste von Attributen, die jeweils aus einem Typen und einem Attributnamen bestehen.

```
package example;

import java.math.BigDecimal;

classdiagram Rechnung{

  class Rechnungtype{
    String rechnungsnummer;
    String datum;
  }
  composition [1] Rechnungtype -> (kunde) Kundetype [1] ;
  composition [1] Rechnungtype -> (posten) Postentype [1] ;

  class Kundetype{
    String vorname;
    String nachname;
  }
  composition [1] Kundetype -> (anschrift) Anschrifttype [1] ;

  class Anschrifttype{
    String strasse;
    int hausnummer;
    String zusatz;
    int plz;
    String stadt;
  }

  class Postentype{
  }
  composition [1] Postentype -> (artikel) Artikeltype [1..*] ;
  composition [1] Postentype -> (versand) Versandtype [1] ;

  class Artikeltype{
    String artikelnummer
    int anzahl;
    BigDecimal einzelpreis;
    BigDecimal gesamtpreis;
    String beschreibung;
  }

  class Versandtype{
    String versandart;
    BigDecimal kosten;
  }

}
```

Quelltext 2.4: Textuelle Darstellung eines Klassendiagramms, das die Datenstruktur des XML-Dokumentes aus 2.1 repräsentiert

Durch das Schlüsselwort `composition` können unidirektionale Beziehungen zwischen Klassen modelliert werden, die eine Existenzabhängigkeit der Instanzen der Klassen implizieren. Eine solche Beziehung drückt aus, dass sich der Typ auf der linken Seite der Komposition aus den Typen der rechten Seite zusammensetzt. Außerdem lässt sich für

jede Seite der Beziehung ein Rollenname und eine Kardinalität festlegen. Daher lassen sich Kompositionen auch nutzen, um die Beziehung zwischen Eltern- und Kindknoten von XML-Elementen zu modellieren. Die Kardinalität des linken Typs ist dabei immer eins, da ein Kindknoten genau einen Elternknoten hat. Für die rechte Seite ist die Kardinalität abhängig davon, ob mehrere XML-Elemente der gleichen Art als Kind des Elternelements existieren können. Als Rollenname bietet sich der Name des XML-Kindelementes an.

In dem für dieses Beispiel gewählten Modellierungsansatz wurden die Klassen, Attribute und Kompositionen, ausgehend vom Wurzelknoten, auf folgende Art und Weise erzeugt:

- Für das Wurzelelement wird eine Klasse erzeugt, deren Name der Elementname kombiniert mit dem Suffix `type` ist.

Für jede erzeugte Klasse:

- Für XML-Attribute des XML-Elements, von dem die Klasse abgeleitet wird, wird jeweils ein Attribut erzeugt, das einen geeigneten Typen hat und den Namen des XML-Attributes trägt.
- Für einfache Kindelemente mit Textinhalt wird ein Attribut erzeugt, das einen geeigneten Typen hat und den Elementnamen als Attributnamen trägt.
- Für Kindelemente mit eigenen Kindelementen wird eine Klasse erzeugt, die nach dem selben Schema aufgebaut und anschließend über eine Komposition mit der Klasse des Elternelements verknüpft wird. Hierbei werden unterschiedliche Kardinalitäten entsprechend berücksichtigt und modelliert.

So haben Objekte der Klasse *Rechnungtype* die Attribute *rechnungsnummer* und *datum* und setzen sich außerdem aus genau einem Objekt vom Typ *Kundentype* und *Postentype*, die über die Rollennamen *kunde* beziehungsweise *posten* referenziert werden, zusammen.

Objektdiagramme

Objektdiagramme ermöglichen die Definition von konkreten Daten- und Objektzuständen eines Systems. Grundsätzlich sind sie zu der durch ein Klassendiagramm vorgegebenen Struktur konform [Rum11]. Syntaktisch sind Objektdiagramme Klassendiagrammen ähnlich, unterscheiden sich jedoch semantisch in wesentlichen Punkten und stellen daher einen eigenen Diagrammtypen innerhalb der UML/P dar [Rum12].

Erneut wird nur eine Teilmenge der Sprachelemente und -konzepte der UML/P-Objektdiagramme benötigt, da die modellierten Datenzustände, bedingt durch die ähnliche Struktur der zugrunde liegenden Klassendiagramme, alle eine ähnliche Struktur aufweisen. Zentrale Elemente eines Objektdiagramms sind Objekte, die durch einen Objektnamen gefolgt von einem Doppelpunkt und anschließender Typangabe, deklariert werden. Objekte beinhalten wie Klassen Attribute, die jedoch zusammen mit konkreten Werten angegeben werden. Ein weiterer Unterschied zu Klassendiagrammen ist, dass mehrere Objekte derselben Klasse auftreten können.

```
package example;

objectdiagram Rechnung{

  r:Rechnungtype{
    rechnungsnummer = "2014-9102";
    datum = "19.09.2014";
  }
  composition r -> (kunde) k;
  composition r -> (posten) p;

  k:Kundetype{
    vorname = "Max";
    nachname = "Mustermann";
  }
  composition k -> (anschrift) a1;

  a1:Anschrifttype{
    strasse = "Musterweg";
    hausnummer = 321;
    plz = 12345;
    stadt = "Musterstadt" ;
  }

  p:Postentype{
  }
  composition p -> (artikel) a2;
  composition p -> (versand) v;

  a2:Artikeltype{
    artikelnummer = "0IC1300";
    anzahl = 4;
    einzelpreis = 105.00;
    gesamtpreis = 420.00;
    beschreibung = "Musterartikel XYZ, EAN Nummer: 4043006189288";
  }

  v:Versandtype{
    versandart = "DHL";
    kosten = 0.00;
  }

}
```

Quelltext 2.5: Textuelle Darstellung eines Objektdiagramms, das die Daten des XML-Dokumentes aus 2.1 konform zu dem Klassendiagramm 2.4 repräsentiert

Durch das Schlüsselwort `composition` kann die unidirektionale, existenzabhängige Kompositionsbeziehung zwischen zwei Objekten festgelegt werden. Die Objekte werden anhand ihrer Objektnamen referenziert und der Rollenname des Teilobjekts wird in runden Klammern angegeben. Aufgrund der Existenzabhängigkeit ist es nicht erlaubt, dass ein Objekt im Objektdiagramm gleichzeitig den rechten Teil von mehr als einer Kompositionsbeziehungen darstellt.

Das Objektdiagramm in Quelltext 2.5 stellt die Daten des XML-Dokumentes aus Abschnitt 2.1 mit Hilfe der zuvor beschriebenen Sprachelemente für Objektdiagramme dar. Dabei stellen die Objekte Instanzen der im Klassendiagramm aus Quelltext 2.4 definierten Klassen dar und sind somit mit dessen Struktur konform. Das Objekt `r` vom Typ *Rechnungtype* entspricht dem Wurzelelement des XML-Dokumentes. Es besitzt die Attribute *rechnungsnummer* und *datum*, deren konkrete Werte angegeben sind, und setzt sich außerdem über Kompositionsbeziehungen aus den Objekten `k` und `p` zusammen, welche den Kindelementen des Wurzelelements entsprechen. Nach dem selben Muster sind alle weiteren Daten innerhalb des Objektdiagramms deklariert.

2.3 MontiCore Sprachframework

Weite Teile des im Rahmen dieser Arbeit implementierten Migrationsframework bauen auf dem *MontiCore* Sprachframework [Kra12, CHC+13] auf und verwenden mit Hilfe von MontiCore erzeugte Softwarekomponenten. Daher werden an dieser Stelle die grundlegenden, zum Verständnis benötigten Konzepte von MontiCore erläutert.

Das MontiCore Framework hilft bei der schnellen und effektiven Erstellung von DSLs beziehungsweise von Komponenten zu deren Verarbeitung. MontiCore erzeugt diese Komponenten anhand textueller Grammatikbeschreibungen der domänenspezifischen Sprachen. [CHC+13]

Im folgenden Unterabschnitt wird deren Aufbau anhand eines Beispiels beschrieben.

2.3.1 Grammatikbeschreibungen für DSLs

Als Beispiel zeigt Quelltext 2.6 vereinfachte Auszüge der im Rahmen von [Sch12] entwickelten Grammatikbeschreibung für die bereits aus Abschnitt 2.2.1 bekannte Sprache zur textuellen Beschreibung von Klassendiagrammen. Sie wurde auf die Sprachelemente, die in Abschnitt 2.2.1 verwendet werden, reduziert.

Die Zeilen der Grammatik in Quelltext 2.6 haben die folgende Bedeutung:

1 Das Package, zu dem die Grammatik gehört. Unter diesem werden die von MontiCore generierten Komponenten wie beispielsweise der Parser für die Dokumente der DSL abgelegt.

3 Der Name der Grammatik, von dem die Namen der generierten Komponenten abgeleitet werden.

5 Definition der Tokenklasse *Name*. Da MontiCore Antlr [PQ95, Par07, www14a] integriert, können Reguläre Ausdrücke mit entsprechender Syntax verwendet werden, um Terminalsymbole zu beschreiben. Gültige *Name*-Literale beginnen mit Klein- oder Großbuchstaben, einem Unterstrich oder einem `$` Zeichen, worauf beliebig viele Klein- oder Großbuchstaben, Unterstriche, Ziffern und `$` Zeichen folgen.

9 Die Regel für das Nichtterminal *CDDefinition* beschreibt, dass ein Klassendiagramm mit dem Schlüsselwort `classdiagram`, gefolgt von einem *Name*-Token und einer öffnenden, geschweiften Klammer beginnt. Darauf folgt eine beliebige Anzahl von

```
1  package mc.umlp.cd;
2
3  grammar CD {
4
5   token Name
6    ( 'a'..'z' | 'A'..'Z' | '_' | '$' )
7    ( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$' )*;
8
9   CDDefinition =
10  " classdiagram " Name
11  "{" ( CDClass | CDComposition )* "}";
12
13  CDClass =
14  " class " Name
15  ( ";" | "{" CDAttribute * "}" );
16
17  CDAttribute = Type Name ";";
18
19  CDComposition =
20  " composition "
21  leftCardinality : Cardinality ?
22  leftReferenceName : Name
23   " ->"
24  ( "(" rightRole : Name ")" )?
25  rightReferenceName : Name
26  rightCardinality : Cardinality ? ";";
27  }
28
29 }
```

Quelltext 2.6: MontiCore Grammatikbeschreibung für Klassendiagramme (vereinfacht)

CDClass- und *CDComposition*-Nichtterminalen und schließlich eine schließende, geschweifte Klammer. Alternativ zu einem `*` hinter einem Block, der die beliebige Wiederholung bedeutet, kann auch ein `+` verwendet werden, um ein mindestens einmaliges Auftreten des Blocks zu fordern.

13 Ein *CDClass*-Nichtterminal beginnt mit dem Schlüsselwort `class`, gefolgt von einem *Name*-Token. Anschließend kann es durch ein Semikolon oder eine, durch geschweifte Klammern eingefasste, beliebige Anzahl von *CDAttribute*-Nichtterminalen abgeschlossen werden.

17 *CDAttribute*-Nichtterminale bestehen aus einem *Type*- und einem *Name*-Token, gefolgt von einem Semikolon.

19 Die Regel für *CDComposition*-Nichtterminale beschreibt, dass eine Kompositionsbeziehung mit dem Schlüsselwort `composition` beginnt. Darauf folgen eine optionale Kardinalitätsangabe der linken Seite sowie ein *Name*-Terminal, über das die Klasse des Kompositums referenziert wird. Die Optionalität wird durch das `?` beschrieben, welches auch hinter geklammerten Ausdrücken verwendet werden kann. Über das Schlüsselwort `->` wird der Beginn der rechten Seite der Kompositionsbeziehung deklariert. Darauf folgt optional in runden Klammern ein *Name*-Token, das den Rollennamen der Komponente angibt. Anschließend folgt ein weiteres *Name*-Token, über das die Klasse der Komponente referenziert wird. Am Ende stehen eine optionale

Kardinalitätsangabe der rechten Seite und ein Semikolon, das die Kompositionsbeziehung abschließt.

Abstrakte Syntaxbäume

Zu den von MontiCore zur Sprachverarbeitung aus der Grammatikbeschreibung erzeugten Komponenten zählt in erster Linie ein Parser für Dokumente der implementierten DSL. Als Ergebnis der Verarbeitung eines Eingabedokuments erzeugt der von MontiCore generierte Parser unter anderem einen abstrakten Syntaxbaum (Abstract Syntax Tree, AST) [Jon03] in Form von Java-Objekten. Der AST repräsentiert den syntaktischen Aufbau des Eingabemodells in Form einer Baumstruktur.

Dabei enthält die Sprachbeschreibung durch das MontiCore Grammatikformat ausreichend Informationen, um AST-Klassen zu erzeugen, die dem Parser erlauben, einen typisierten und heterogenen AST aufzubauen [CHC+13]. Die grundlegende Idee ist, dass aus jeder Regel der Grammatik eine AST-Klasse erzeugt wird.

Abbildung 2.2 zeigt das Klassendiagramm der aus der *CDComposition*-Regel der MontiCore Grammatikbeschreibung aus Quelltext 2.6 erzeugten AST-Klasse. Der Klassenname wird aus dem Namen der Regel abgeleitet, die einzelnen Attribute der Klasse aus den Komponenten der rechten Seite der Regel. Die Regelkomponenten bestehen in dem gewählten Beispiel jeweils aus zwei, durch einen Doppelpunkt getrennten, Bezeichnern. Der erste Bezeichner bestimmt den Attributnamen, während der zweite Teil auf ein weiteres Nichtterminal-Symbol verweist. Ist dieses Nichtterminal ein Token (im Beispiel *Name*), erhält das erzeugte Attribut der AST-Klasse den Typ *String*. Verweist es auf eine weitere Regel (im Beispiel *Cardinality*), so ist der Typ des Attributs, die AST-Klasse, die aus der referenzierten Regel generiert wird.

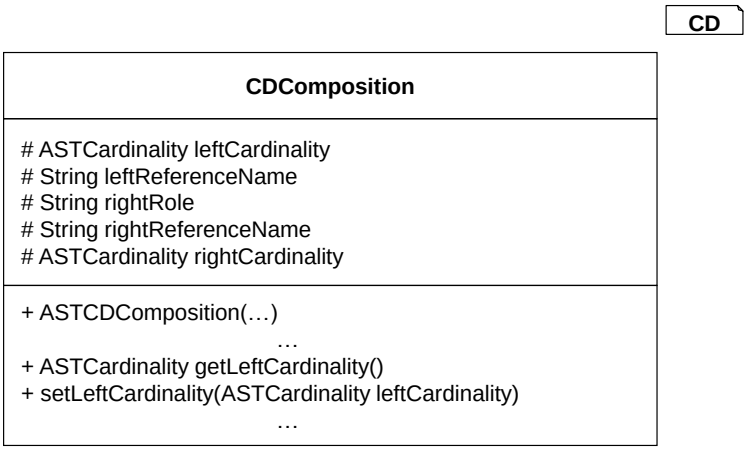


Abbildung 2.2: Klassendiagramm der aus der Regel *CDComposition* erzeugten AST-Klasse

Bei Regelkomponenten ohne einen, durch Doppelpunkt getrennten, ersten Bezeichner wird der Attributname vom Klassennamen abgeleitet. So enthält die AST-Klasse *ASTAttribute* das Attribut *name* vom Typ *String*. Bei Produktionen, die Alternativen beziehungsweise optionale Regelkomponenten enthalten, werden für alle Regelkomponenten entsprechende Attribute erzeugt.

Für wiederholt auftretende Regelkomponenten, also solche auf die direkt ein $\star$ oder + folgt oder die innerhalb eines Blocks mit $\star$ oder + auftreten, wird neben der AST-Klasse eine Listenklasse erzeugt, die Instanzen dieser AST-Klasse aufnehmen kann. Die Listenklasse wird dann als Typ des Attributs der zu der Regel gehörenden AST-Klasse verwendet. So enthält beispielsweise die Klasse *ASTCDDefinition* die Attribute *CDClasses* und *CDCompositions* vom Typ *ASTCDClassList* beziehungsweise *ASTCDCompositionList*.

Durch den Aufbau der AST-Klassen wird die Repräsentation der Syntax als Baumstruktur deutlich. Ebenfalls ist ersichtlich, dass diese Baumstruktur durch Änderungen an der Grammatik vom Entwickler der DSL gezielt beeinflusst werden kann.

Mit Hilfe der AST-Klassen können:

- Geparste Dokumente der DSL analysiert werden,
- AST-Repräsentationen von Dokumenten der DSL neu aufgebaut werden,
- AST-Repräsentationen einer DSL in ASTs der selben oder einer anderen DSL transformiert werden.

Durch die Typisierung des ASTs in MontiCore lassen sich diese Tätigkeiten mit beliebigem Java-Code, etwa durch die Verwendung objektorientierter Entwurfsmuster oder vorhandener Bibliotheken, komfortabel realisieren. Der AST stellt daher die zentrale Schnittstelle für die Verarbeitung der Dokumente einer DSL dar.

AST-Regeln

Neben der Möglichkeit die Struktur der AST-Klassen über die Produktionen der Grammatik zu beeinflussen, bietet MontiCore die Möglichkeit die Grammatik um sogenannte *AST-Regeln* zu erweitern. Durch sie können die AST-Klassen um beliebige Attribute und Methoden ergänzt werden, die in keinem Kontext zu der Bedeutung der Produktion im Sinne der DSL stehen müssen.

Schnittstellenproduktionen

Bei umfangreichen Grammatiken gibt es oft Gemeinsamkeiten zwischen Produktionen, die durch die zuvor beschriebene Erzeugung der AST-Klassen nicht widergespiegelt werden. MontiCore ermöglicht dem Sprachentwickler deshalb über das Schlüsselwort `interface` sogenannte Schnittstellenproduktionen zu erzeugen. Produktionen können diese Schnittstelle durch das `implements`-Schlüsselwort implementieren. Die so definierten Schnittstellen und Implementierungsbeziehungen werden von MontiCore bei der Erzeugung der AST-Struktur berücksichtigt und haben auch einen entsprechenden Einfluss auf die Syntax der Sprache. Die Schnittstellenproduktion kann wie eine einfache Produktion innerhalb der Grammatik verwendet werden und überall, wo sie auftritt, können Objekte aller Klassen auftreten, die sie implementieren. Außerdem ist es möglich, die aus Schnittstellenproduktionen erzeugten AST-Klassen, wie im vorherigen Absatz beschrieben, um Methoden und Attribute zu erweitern. Das hat durch die in die AST-Struktur übertragenen Implementierungsbeziehungen entsprechenden Einfluss auf die AST-Klassen der implementierenden Produktionen.

Syntaktische Prädikate

Der von MontiCore aus einer Grammatikschreibung generierte Parser arbeitet nach dem LL(k)-Verfahren [ASU86], also mit einem konstanten Lookahead [Kra12]. Das heißt, der Parser betrachtet nur eine bestimmte Anzahl von Wörtern, die auf die aktuelle Position folgen, um die nächste Produktion auszuwählen. Dieses Vorgehen kann dazu führen, dass ein Entscheidungskonflikt zwischen alternativen Produktionen einer Grammatik besteht, da die zur Unterscheidung benötigte Anzahl von Wörtern größer als der verwendete Lookahead des Parsers ist.

Die Verwendung von syntaktischen Prädikaten [Par07] erlaubt dem Parser trotz der Mehrdeutigkeit die richtige Regel auszuwählen. Ein syntaktisches Prädikat besteht aus den zur Unterscheidung relevanten Teilen einer Produktion in runden Klammern und einem => Symbol, gefolgt von der zu wählenden Regel. In [CHC+13] ist die Verwendung von syntaktischen Prädikaten zur Auflösung von Mehrdeutigkeiten anhand eines Beispiels erklärt.

2.3.2 Workflows und DSL-Tools

Um die automatisierte Verarbeitung von DSLs zu erleichtern, stellt MontiCore das *DSLTool-Framework* zur Verfügung. Es ermöglicht die Erstellung von generativen Werkzeugen, deren Konfiguration und Ablaufsteuerung entkoppelt sind [Kra12]. So ist es möglich, durch Angabe von Parametern verschiedene Aufgaben mit einem einzigen Werkzeug auszuführen.

Mit Hilfe des DSLTool-Frameworks erstellte Werkzeuge lassen sich flexibel nutzen und kombinieren, da Aufruf und Parametrisierung an verschiedenen Stellen erfolgen können:

- Programmatisch an beliebigen Stellen im Programmcode.
- Als Maven [www14b] Ausführungsschritt mit Hilfe des MontiCore-Maven-Plugins.
- Durch die Kommandozeile und Übergabe von Kommandozeilenparametern.

Ein Werkzeug wird durch eine von *DSLTool* abgeleitete Klasse implementiert. In ihr werden die Typen der zu verarbeitenden Eingabedateien sowie verschiedene Ausführungsschritte des Werkzeugs definiert. Die Ausführungsschritte sind Algorithmen, die als sogenannte *Workflows* in eigenen Klassen gekapselt werden. Der erste Arbeitsschritt eines *DSLTools* ist in der Regel das Parsen des Dateiinhalts.

2.3.3 Codegenerierung

Neben der Möglichkeit Dokumente über die AST-Repräsentation manuell zwischen verschiedenen DSLs zu transformieren, bietet MontiCore auch die Möglichkeit, auf Basis von Templates, Dokumente anderer Sprachen aus einem AST zu erzeugen. Dieser Ansatz ist vor allem für die Erzeugung von ausführbarem Code geeignet, da dieser hohe Anteile von fixem, technischen Code wie Kontrollstrukturen hat [Sch12]. Dabei ist jedes Template für die Transformation einer bestimmten AST-Klasse verantwortlich.

Innerhalb eines Templates lassen sich die statischen Anteile direkt in der Syntax der Zielsprache formulieren, während andere Teile aus Werten des ASTs der Ausgangssprache

bestehen oder auf deren Basis berechnet werden können. Zur Umsetzung dieses Ansatzes verwendet MontiCore die Template-Engine *FreeMarker* [www14h].

FreeMarker-Templates haben die Dateiendung *.ftl* und können neben den Fragmenten der Zielsprache Variablen, Operatoren und Kontrollstrukturen beinhalten, die den Ablauf der Codegenerierung beeinflussen. In [Sch12] wird ein ausführlicher Überblick über diese gegeben. Die Erklärungen zu den für die in Abschnitt 5.3 vorgestellten Templates relevanten Kontrollstrukturen sind in Tabelle 2.1 aufgeführt. Außerdem bietet [www14h] eine vollständige Dokumentation.

Tag	*Beschreibung*
`<#assign` *`Name=Wert`*`>`	Wertzuweisung an lokale Variablen, wobei auch mehrere durch Leerzeichen getrennte "Name = Wert"-Paare als Parameter angegeben werden können. Nicht existierende Variablen werden gleichzeitig angelegt.

`<#if` *`Bedingung 1`*`> ...`
`<#elseif` *`Bedingung 2`*`> ...`
`...`
`<#elseif` *`Bedingung N`*`> ...`
`<#else> ...`
`</#if>`
	Führt nur den ersten zwischen den Tags stehenden Teil des Templates aus, dessen davor angegebene Bedingung als boolescher Wert *true* ausgewertet wird. Trifft dies für keine Bedingung zu, wird der Teil nach dem *else*-Tag ausgeführt. Es können mehrere *elseif* nach dem *if*-Tag angegeben werden, wobei *elseif* und *else* jeweils optional sind.

`<#list` *`Liste`* `as` *`Element`*`>`
`...`
`</#list>`
	Iteriert über eine Liste von Elementen. In jeder Iteration kann auf das aktuelle Element über den für *Element* vergebenen Namen zugegriffen werden.

Tabelle 2.1: Übersicht der wichtigsten FreeMarker-Kontrollstrukturen nach [Sch12]

Außer den explizit angelegten Variablen sind durch das MontiCore Framework in jedem Template die Variablen *op* und *ast* verfügbar. Die *ast*-Variable enthält die Instanz der AST-Klasse des aktuellen Knotens und erlaubt so den Zugriff auf das Dokument der Quellsprache.

Die Variable *op* ermöglicht den Zugriff auf eine Instanz der Klasse *TemplateOperator*, welche Methoden für die Definition von Variablen, den Aufruf von Hilfklassen und die Einbindung von weiteren Templates in das aktuelle Template zur Verfügung stellt:

- Über die Methode *instantiate* können Hilfsklassen instanziiert werden und beispielsweise über eine FreeMarker *assign*-Anweisung in einer Variablen gespeichert werden.
- Die Methode *setValue* stellt eine Möglichkeit für die Wertzuweisung an globale Variablen beziehungsweise zu deren Erzeugung dar.
- Mit Hilfe der Methode *callTemplate* kann ein Template aufgerufen werden. Das Ergebnis des Templateaufrufs wird dabei in einer eigenen Datei abgelegt. Dazu wird der Name des aufzurufenden Templates, der zu verwendende AST-Knoten und der gewünschte Name der Zieldatei an die Methode übergeben.

- Die Methode *includeTemplates* ermöglicht die Einbettung weiterer Templates. Das Ergebnis der aufgerufenen Templates wird an der Stelle des Aufrufs eingefügt. Dazu werden der Methode der Name des einzubettenden Templates sowie der zu verwendende AST-Knoten als Parameter übergeben. Es ist auch möglich, Listen von AST-Knoten zu übergeben, für die dann jeweils das angegebene Template aufgerufen wird.

Templates, die über die Methode *callTemplate* aufgerufen werden, und deren Ergebnis in einer eigenen Datei abgelegt wird, werden als *Haupttemplates* bezeichnet. Templates, die in andere Templates eingebettet werden, werden als *Subtemplates* bezeichnet.

Kapitel 3

Anforderungen an das Migrationsframework

In diesem Kapitel werden durch die Aufgabenstellung implizierte, grundlegende Anforderungen an das Migrationsframework formuliert. Zum Erreichen des Gesamtziels des Frameworks, die vollständige Ablösung von vorhandenen XML-Formaten zu ermöglichen, gibt es eine Vielzahl möglicher Wege. Sie setzen sich aus unterschiedlichen Teillösungen zusammen, deren Anzahl bereits an dieser Stelle, durch die konkrete Ausformulierung der Anforderungen, reduziert werden soll.

3.1 Ablösung von XML Schema Definitionen durch Dokumente einer besser lesbaren Sprache

Wie in Kapitel 2.1.1 erläutert, lassen sich XML-Formate mit Hilfe von Schemasprachen beschreiben. Für komplexere Formate, die verschiedene Datentypen abstrahieren, ist XSD eine gut geeignete Schemasprache [www14c]. Auch wenn es vom Standpunkt der formellen Sprachtheorie betrachtet ausdrucksstärkere Sprachen gibt [MLMK05], ist die Verbreitung der XSD aufgrund ihrer Standardisierung durch das W3C [www14s] im industriellen Kontext sehr hoch.

Da die Grammatiken sämtliche Informationen über das abzulösende XML-Format beinhalten und XSD selbst ein XML-Dialekt ist, ist es notwendig diese Informationen durch das Framework zu extrahieren, damit es die migrierten Daten auch nach Ablösung von XML validieren kann. Außerdem wäre es vorteilhaft, diese Informationen auch in einer besser lesbaren Form zu pflegen.

Damit ergeben sich folgende Anforderungen an die Art der Ablösung von XML Schema Definitionen:

1. Das Migrationsframework ist in der Lage mit XSD beschriebene XML-Formate abzulösen.
2. Die in der XSD-Grammatik enthaltenen Informationen müssen durch das Framework extrahiert werden können.

3. Die in der XSD-Grammatik enthaltenen Informationen sollten zwecks vollständiger Ablösung von XML und XSD sowie zur einfacheren Weiterverarbeitung in einem besser lesbaren Format gespeichert werden.

3.2 Migration der XML-Daten in Dokumente einer geeigneten DSL

Die primäre Motivation dieser Arbeit ist es, XML-Daten in Dokumente einer weniger verbosen, problemadäquaten und dadurch für Anwender besser verarbeitbaren Sprache zu migrieren. In Kapitel 2.2 wurden bereits einige Vorteile von domänenspezifischen Sprachen herausgearbeitet, die sie für den Einsatz in diesem Szenario besonders geeignet erscheinen lassen.

Grundsätzlich gibt es zwei verschiedene Möglichkeiten die Migration durch den Einsatz domänenspezifischer Sprachen zu realisieren, zwischen denen jedoch an dieser Stelle noch nicht entschieden werden soll. Ein Ansatz ist, für jedes XML-Format, anhand der Informationen der XSD-Grammatik, eine eigene DSL zu erzeugen. Alternativ kann für alle XML-Formate eine einzige DSL genutzt werden, deren Domäne die Modellierung von Daten umfasst.

Beide Ansätze haben Vor- und Nachteile, welche im nächsten Kapitel, im Rahmen der Konzeption des Frameworks, diskutiert werden. Hinsichtlich der Datenmigration werden die folgenden Anforderungen definiert:

4. Die XML-Daten müssen von dem Framework in Dokumente einer geeigneten, gut lesbaren DSL transformiert werden.

5. Die transformierten Daten müssen vom Framework analog zur bisherigen Validierung über die XSD-Grammatik validiert werden können.

3.3 Bereitstellung der migrierten Daten als Java-Objekte

Um einen komfortablen Zugriff auf die Daten des neuen Formats zu ermöglichen, wird im Rahmen der Aufgabenstellung gefordert, dass die Daten über das Framework in Form von Java-Objekten verfügbar gemacht und nach deren Manipulation wieder persistiert werden können. Die konkrete Umsetzung ist jedoch von dem zur Umsetzung der Anforderungen des vorherigen Abschnitts gewählten Ansatz abhängig.

Es werden daher die folgenden Anforderungen definiert:

6. Das Framework muss in der Lage sein, Daten des neuen Formats, nach Migration oder Neuerstellung, als Java-Objekte mit geeigneten Zugriffsmethoden bereitzustellen.

7. Das Framework muss in der Lage sein, entsprechende Java-Objekte nach Manipulation in das neue Datenformat zurückzuführen.

3.4 Verwendung des MontiCore Sprachframeworks

In Abschnitt 2.3 wurden das MontiCore Sprachframework und einige seiner Funktionen vorgestellt. Dabei wurde deutlich, dass alle sich durch die in den vorherigen Abschnitten gestellten Anforderungen ergebenden Aufgaben der Spracherzeugung und -verarbeitung mit Hilfe von MontiCore bewältigt werden können. Auch das am Lehrstuhl Software Engineering verfügbare Expertenwissen bei der Verwendung des MontiCore Frameworks sprechen für dessen Einsatz im Rahmen dieser Arbeit.

Somit ergibt sich folgende Anforderung:

8. Aufgaben zur Erstellung und Verarbeitung von DSLs sollen mit Hilfe des MontiCore Sprachframeworks gelöst werden.

3.5 Größtmögliche Unabhängigkeit von weiteren Frameworks

Aus Gründen der Wartbarkeit und Erweiterbarkeit ist es ratsam, die Anzahl der verwendeten externen Softwarebibliotheken und -frameworks gering zu halten. Nötige Änderungen, Erweiterungen und die Beseitigung eventueller Fehler lassen sich, auch wenn der Quellcode einer verwendeten externen Software verfügbar ist, in eigenem Code im Normalfall leichter durchführen.

Viele der teilweise recht komplexen Teilaufgaben des Migrationsframeworks, für die vorhandene externe Softwarebibliotheken eingesetzt werden könnten, lassen sich beispielsweise mit Hilfe des MontiCore Frameworks effizient lösen.

Daher wird die folgende Anforderung formuliert:

9. Bei Konzeption und Entwicklung des Migrationsframeworks soll Wert auf eine hohe Unabhängigkeit von externer Software gelegt werden.

3.6 JiBX-Kompatibilität

In der Aufgabenstellung ist eine Evaluierung des Migrationsframeworks anhand im industriellen Einsatz befindlicher und zur Ablösung diskutierter XML-Formate vorgesehen. Für den praktischen Einsatz dieser Formate wird größtenteils das verbreitete JiBX-Framework [www14k] zur Datenbindung genutzt.

Auf den von JiBX generierten Klassen basieren entsprechend große Mengen produktiv genutzter Software, die erst die domänenspezifische, problembezogene Verarbeitung der Daten implementiert. Für die effiziente Ablösung der in diesen Fällen genutzten XML-Formate ist es also erforderlich, dass der bereits vorhandene Produktivcode mit geringem zusätzlichen Aufwand weiter eingesetzt werden kann.

Daraus ergibt sich die folgende Anforderung:

10. Die zur Erfüllung von Anforderung 6. und 7. vom Migrationsframework erzeugten Java-Objekte beziehungsweise Klassen sollen kompatibel zu den durch das JiBX-Framework erzeugten Artefakten sein.

Kapitel 4

Konzeption

In diesem Kapitel werden die im Rahmen der Arbeit entstandenen Konzepte für die Umsetzung der Anforderungen an das Migrationsframework aus Kapitel 3 vorgestellt. Die Konzepte stellen den Rahmen für die in Kapitel 5 beschriebene Implementierung des Migrationsframeworks dar.

4.1 Übersicht

Zunächst ist es nötig, sich für konkrete Ansätze zur Lösung der gestellten Anforderungen zu entscheiden, um nachfolgend eine geeignete Komponentenarchitektur des Frameworks sowie konkrete Konzepte für die Implementierung der einzelnen Komponenten entwerfen zu können.

In Abschnitt 3.2 wurde die Form der Verwendung domänenspezifischer Sprachen noch offen gelassen. Zur Diskussion stehen der Ansatz, für jedes XML-Format eine eigene DSL zu erzeugen, und der Ansatz, eine gemeinsame, zur Datenmodellierung geeignete DSL für alle XML-Formate zu verwenden. In den folgenden Unterabschnitten sollen Form der Verwendung und Art der genutzten DSLs konkretisiert werden.

4.1.1 Form der Verwendung von DSLs

Der Einsatz der gleichen DSL für alle XML-Formate hat den Nachteil, dass die Validierung von Dokumenten gegenüber den Einschränkungen der einzelnen XML-Formate separat implementiert werden muss. Würde eine spezifische DSL aus jeder XSD-Grammatik erzeugt, wäre keine dedizierte Validierung erforderlich, da die DSL-Grammatik so generiert werden kann, dass nur gültige Datensätze Dokumente der DSL sind.

Da bei diesem Ansatz jedoch alle Informationen zum Datenformat in Form der DSL-Grammatik anstelle der XSD-Grammatik vorhanden sind, ist Anforderung 3. nicht erfüllt. Die Lesbarkeit der Dokumente der DSL ist zwar besser als die von XML-Dateien, die einer DSL-Grammatik im Vergleich zu einer XSD-Grammatik jedoch nicht. Es ist im Gegenteil sogar damit zu rechnen, dass sie deutlich schlechter ist.

Auch bei dem Ansatz individueller DSLs würden die Sprachen alle eine ähnliche Struktur aufweisen, da die Grammatiken automatisiert aus den entsprechenden XSD-Grammatiken erzeugt werden müssten. Daher hat die Entscheidung über die Nutzungsform von domänenspezifischen Sprachen nur unwesentlichen Einfluss auf den Aufbau und damit die entscheidenden, im Rahmen von Anforderung 4. geforderten, Eigenschaften der Dokumente der DSL.

Zu Gunsten von Anforderung 3. sollen daher gemeinsame DSLs für alle XML-Formate verwendet werden. Im folgenden Unterabschnitt sollen dazu geeignete DSLs konzipiert beziehungsweise ausgewählt werden.

4.1.2 Auswahl geeigneter DSLs

In Kapitel 2.2.1 wurde bereits die in [Sch12] mit Hilfe von MontiCore entwickelte, textuelle Fassung der UML/P vorgestellt. Anhand der gewählten Beispieldokumente der Sprachen für Klassen- und Objektdiagramme wurde deutlich, dass diese gut geeignet sind, um die hierarchische Datenstruktur von XML-Formaten, beziehungsweise die Daten selbst, zu repräsentieren.

Da diese Sprachen sowie eine umfassende Werkzeuginfrastruktur zu deren Verarbeitung bereits zur Verfügung stehen, stellen sie eine gute Option für die Verwendung innerhalb des Migrationsframeworks dar. Auch die in Kapitel 3 gestellten Anforderungen an Problembezogenheit und Lesbarkeit werden von den beiden Teilsprachen der UML/P erfüllt, weshalb sie gegenüber der Implementierung einer oder mehrerer neuer domänenspezifischer Sprachen bevorzugt werden.

Ein weiteres Argument für diese Entscheidung ist die Möglichkeit, vorhandene XML-Datenmodelle auf diese Weise in modellgetriebene Entwicklungsprozesse [Sch12] zu integrieren. Die Vorteile die daraus erwachsen werden in [BKK04] genannt.

Nachdem die Entscheidung über die Form der Verwendung und die Art von domänenspezifischen Sprachen innerhalb des Migrationsframeworks getroffen ist, werden in den nächsten Abschnitten der Entwurf für die Zergliederung des Frameworks in einzelne Komponenten sowie Konzepte zu deren Implementierung vorgestellt.

4.2 Aufbau des Frameworks

Die einzelnen Teilfunktionen des Migrationsframeworks sind vielfältig. Daher ist die Aufteilung des Frameworks in einzelne Teilkomponenten ein essentieller Teil der Konzeption. Nach der Entscheidung für die textuelle Form der UML/P als einzusetzende domänenspezifische Sprache im letzten Abschnitt wird die Modularisierung nun vorgenommen.

Abbildung 4.1 zeigt den Aufbau des Frameworks in Form eines T-Diagramms [Ter96]. Ausgehend von den Artefakten eines zu migrierenden XML-Formats ist jede Komponente des Frameworks als T-Bauteil illustriert. Die seitlichen Blöcke bezeichnen das Format beziehungsweise die Sprache der Ein- und Ausgabedaten der jeweiligen Komponente. Der mittlere Block trägt den Namen der Komponente und der untere Block die Implementierungssprache.

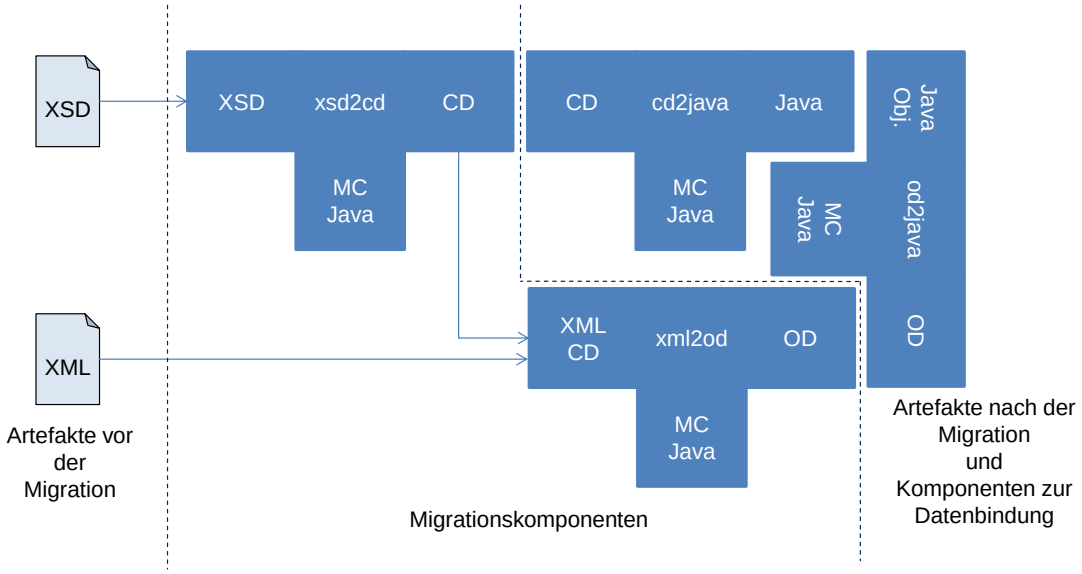


Abbildung 4.1: Aufteilung des Frameworks in Komponenten als T-Diagramm dargestellt

Somit gliedert sich das Framework in die folgenden Komponenten mit ihren entsprechenden Transformationsaufgaben:

- *xsd2cd*: Die Transformation von XSD-Grammatiken in Klassendiagramme wird von der *xsd2cd*-Komponente implementiert.
- *xml2od*: Die *xml2od*-Komponente wandelt ein XML-Dokument in ein Objektdiagramm um. Dazu steht zusätzlich das durch die *xsd2cd*-Komponente aus der dazugehörigen XSD-Grammatik erzeugte Klassendiagramm als Eingabe zur Verfügung.
- *cd2java*: Aus den Klassendiagrammen, die von der *xsd2cd*-Komponente erzeugt werden, generiert die *cd2java*-Komponente Java-Klassen, die zur Datenverarbeitung genutzt werden können.
- *od2java*: Die *od2java*-Komponente hat zwei Aufgaben. Zum einen instanziiert sie die von der *cd2java*-Komponente generierten Java-Klassen und befüllt diese mit Daten aus Objektdiagrammen. Zum anderen ist sie in der Lage, Objektdiagramme aus vorhandenen Java-Instanzen zu erzeugen.

Durch die Trennlinien in der Abbildung werden die verschiedenen Stufen der Migration verdeutlicht. Vor der Migration sind ausschließlich die XSD-Grammatik und eventuell Daten in Form von XML-Dokumenten verfügbar. Nach den Transformationen durch die Komponenten *xsd2cd* und *xml2od* ist die eigentliche Migration abgeschlossen. Die XSD-Grammatik und die XML-Dokumente können verworfen werden und das Datenmodell auf Basis der Klassen- und Objektdiagramme weiterentwickelt und -verwendet werden.

Dieser Punkt kann natürlich auch unabhängig von vorhandenen XML-Formaten erreicht werden, indem bereits vorhandene Klassen- und Objektdiagramme genutzt oder solche neu erstellt werden.

Aus den Klassendiagrammen können letztendlich Java-Klassen durch die *cd2java*-Komponente generiert werden und mit Hilfe der *od2java*-Komponente mit Daten aus Objektdiagrammen befüllt werden sowie deren Instanzen in Objektdiagramme überführt werden. Somit kann das Datenmodell auch nach der Transformation jederzeit weiterentwickelt werden.

4.2.1 Einsatz des MontiCore Sprachframeworks

Wie aus Abbildung 4.1 ersichtlich wird, werden alle Komponenten mit Hilfe von MontiCore und Java implementiert, da jeweils zumindest die Parser und AST-Klassen zum Einlesen und Erzeugen der Ein- und Ausgabedateien durch MontiCore generiert werden.

In den folgenden Abschnitten werden detaillierte Umsetzungskonzepte für die einzelnen Komponenten des Migrationsframeworks im Hinblick auf die Anforderungen aus Kapitel 3 vorgestellt.

4.3 xsd2cd - Umwandlung von XSD-Grammatiken in Klassendiagramme

Die Umwandlung von existierenden XSD-Grammatiken in Klassendiagramme stellt den ersten Schritt beim Einsatz des Migrationsframeworks dar. Um XSD-Dateien einlesen zu können, muss zunächst eine entsprechende MontiCore Grammatikbeschreibung erstellt werden, aus der ein Parser und die benötigten AST-Klassen generiert werden können.

Der beim Parsen entstandene AST kann dann mit Hilfe von Java-Code in einen AST für UML/P-Klassendiagramme transformiert und anschließend in eine Datei ausgegeben werden. Die dazu benötigten AST-Klassen und Werkzeuge sind als Ergebnis von [Sch12] bereits verfügbar.

Da der Java-Code von der *cd2java*-Komponente auf Basis der Klassendiagramme generiert wird, muss Anforderung 10. bereits bei der Erzeugung der Klassendiagramme berücksichtigt werden, um die JiBX-Kompatibilität der vom Migrationsframework erzeugten Java-Klassen zu gewährleisten. Sollte diese Anforderung in einem anderen Szenario wegfallen oder durch Andere ersetzt werden, kann, dank der Komponentenarchitektur, einfach eine andere Komponente für die Erzeugung von Klassendiagrammen entwickelt und in das Framework eingebunden werden.

4.4 cd2java - Generierung von Java-Klassen aus Klassendiagrammen

Zur Umsetzung der Anforderungen 6. und 7. werden Java-Klassen benötigt, die den Zugriff auf die Daten ermöglichen. Grundsätzlich gibt es zahlreiche Gestaltungsmöglichkeiten bei

der Generierung derartiger Klassen.

Im Rahmen des *DEx*-Projekts [www14p] sind bereits Generatoren vorhanden, die aus Klassendiagrammen komplette Anwendungen zur Datenverwaltung erzeugen. Der Aufbau der Klassen zur Datenhaltung verstößt jedoch gegen Anforderung 10., weshalb die Codegeneratoren des *DEx*-Projektes höchstens die Grundlage für die im Rahmen der Arbeit zu Implementierenden darstellen können. Dennoch könnten sie aufgrund der Komponentenarchitektur in einem anderen Kontext im Rahmen des Frameworks eingesetzt werden.

Die Gestaltung der Java-Klassen wird also durch Anforderung 10. geprägt, da zumindest alle Attribute, Methoden und inneren Klassen verfügbar sein müssen, um die Weiterverwendung von existierenden JiBX-basierten Java-Anwendungen zu ermöglichen.

Zur Codegenerierung bietet sich der in [Sch12] entwickelte und in Abschnitt 2.3.3 vorgestellte, templatebasierte Ansatz an. Er empfiehlt die Gliederung der Templates in Haupttemplates zur Dateierzeugung und Subtemplates zur Kapselung von wiederkehrenden Blöcken der Zielsprache. Außerdem werden für komplexe Berechnungen in Java implementierte Hilfsklassen eingesetzt. Das erlaubt, den Inhalt der Templates in großen Teilen auf die Zielsprache zu beschränken, was eine effiziente Entwicklung ermöglicht und eine gute Lesbarkeit der Templates gewährleistet.

4.5 xml2od - Umwandlung von XML-Dokumenten in Objektdiagramme

Den zweiten Schritt der Migration stellt die Transformation der vorhandenen XML-Datensätze in Objektdiagramme dar. Die *xml2od*-Komponente erhält XML-Dokumente sowie das aus der dazugehörigen XSD-Grammatik erzeugte Klassendiagramm als Eingabe.

Es existiert bereits eine MontiCore Grammatikbeschreibung für XML-Dokumente und damit Parser und AST-Klassen, die zum Einlesen der Daten verwendet werden können. Wie in Abschnitt 4.3 bereits erwähnt, sind auch für Klassen- und Objektdiagramme die zur Sprachverarbeitung benötigten Werkzeuge und Klassen bereits vorhanden.

Der vom Parser erzeugte AST eines XML-Dokuments kann von der *xml2od*-Komponente analysiert werden. Mit Hilfe der Informationen aus dem Klassendiagramm kann sie aus den XML-Daten den AST eines Objektdiagramms aufbauen und in eine Datei ausgeben.

4.6 od2java - Erzeugung von Java-Objekten aus Objektdiagrammen sowie Erzeugung von Objektdiagrammen aus Java-Objekten

Nach Abschluss der Migration von XML-Grammatik und -Daten sowie der Erzeugung der benötigten Java-Klassen zur Datenbindung ist die *od2java*-Komponente der Teil des Frameworks, der in den Anwendungscode eingebunden werden muss, um die Verwendung der migrierten Daten zu ermöglichen.

Zunächst muss die *od2java*-Komponente in der Lage sein, die von der *cd2java*-Komponente erzeugten Klassen zu instanziieren und die so erzeugten Java-Objekte anschließend mit den Daten aus einem Objektdiagramm zu befüllen. Dabei ist es wichtig, dass die dazu benötigten Methodenaufrufe denen zum Einlesen von XML-Dokumenten des JiBX-Frameworks entsprechen, um Anforderung 10. gerecht zu werden. Die Aufrufe des JiBX-Frameworks innerhalb von existierendem Code können dann einfach und ohne zusätzlichen Aufwand durch identische Aufrufe der *od2java*-Komponente ersetzt werden.

Die Herausforderung bei der Implementierung liegt darin, dass die Klassen- beziehungsweise Objektstrukturen der *od2java*-Komponente gar nicht oder mit Hilfe von Reflexion [FFI+04, www14i] erst zur Laufzeit bekannt sind. Prinzipiell gibt es zwei verschiedene Ansätze um dieses Problem zu lösen:

- Neben den Klassen zur Datenhaltung wird von der *cd2java*-Komponente auch Code zu deren Instanziierung und Befüllung anhand entsprechender Objektdiagramm-Fragmente generiert. Diese Klassen oder Methoden können dann von der *od2java*-Komponente mit Teilen des Objektdiagramms aufgerufen werden, ohne dass ihr die Klassenstruktur dazu bekannt sein muss.
- Alternativ können die Datenbindungsklassen per Reflexion zur Laufzeit analysiert werden und anschließend ebenso per Reflexion entsprechende Objekte erzeugt und mit den Daten aus dem Objektdiagramm befüllt werden.

Beide Vorgehensweisen haben Vor- und Nachteile. Sauberer ist der erste Ansatz, da die Verwendung von Reflexion unter anderem schlechte Wartbarkeit des Codes mit sich bringt [www14i] und die Prinzipien objektorientierter Programmierung verletzt.

Kapitel 5

Implementierung

Anhand der in Kapitel 4 erarbeiteten Konzepte wurde das Migrationsframework *xsd2dsl* implementiert, welches in diesem Kapitel vorgestellt wird. Es werden die Projektstruktur sowie die zur Umsetzung des Frameworks entworfenen domänenspezifischen Sprachen, Transformationsworkflows und Codegeneratoren beschrieben. Zum Abschluss des Kapitels wird die Verwendung des Frameworks anhand eines Beispiels erläutert.

Die Implementierung des Migrationsframeworks ist in Java und mit Hilfe des MontiCore Frameworks nach einem testgetriebenen Ansatz [Mad10] entwickelt worden. Dabei wurde Apache Maven [www14b] als Build-Management-Werkzeug und JUnit [www14o] als Testframework verwendet.

5.1 Aufteilung des Frameworks in Teilprojekte

Die Aufteilung der Implementierung in verschiedene Teilprojekte wurde auf Basis der in Abschnitt 4.2 konzipierten Modularisierung der Migrationsframeworks vorgenommen. Das Hauptprojekt *xsd2dsl* befindet sich im Namensraum *de.monticore* und setzt sich aus den folgenden Maven-Modulen zusammen:

- *common*: Softwarebibliothek mit gemeinsam genutzten Klassen und Methoden.
- *xsd-lang*: Implementierung der DSL zum Verarbeiten von XSD-Dokumenten.
- *xsd2cd*: Werkzeug zur Transformation von XSD-Dokumenten in Klassendiagramme.
- *xml2od*: Werkzeug zur Transformation von XML-Dokumenten in Objektdiagramme.
- *cd2java*: Codegenerator zur Erzeugung von JiBX-kompatiblen Datenbindungsklassen aus Klassendiagrammen.
- *cd2java-use*: Projekt zum Aufruf des Codegenerators.
- *od2java*: Projekt zur Datenbindung von Objektdiagrammen.

In den folgenden Abschnitten wird die Implementierung der einzelnen Komponenten des Frameworks beschrieben.

5.2 Umwandlung von XSD-Grammatiken in Klassendiagramme

Abbildung 5.1 zeigt ein Aktivitätsdiagramm, das den Ablauf des *Xsd2CdTools*, um eine XSD-Grammatik in ein Klassendiagramm zu transformieren, beschreibt. Zunächst wird die zu transformierende XSD-Datei geparst und in einen entsprechenden XSD-AST transformiert.

Anschließend wird der in Unterabschnitt 5.2.2 beschriebene Transformationsworkflow gestartet, der anhand des XSD-ASTs einen CD-AST erzeugt. Der XSD-AST wird zu Beginn der Transformation der XSD-Elemente auf referenzierte Schemadateien analysiert, die, sofern vorhanden, ebenfalls geparst werden. Bevor mit der Transformation des ursprünglichen Schemas fortgefahren wird, wird der Transformationsprozess auf dem AST des referenzierten Schemas gestartet und dieser ebenfalls zunächst auf referenzierte Schemadateien untersucht.

Dieser Vorgang wird wiederholt, bis alle Referenzen erfolgreich aufgelöst sind. Entsprechend werden zuerst die Elemente aller referenzierten Schemata transformiert. Durch dieses Vorgehen wird gewährleistet, dass alle Elemente aus referenzierten Schemadateien bereits transformiert sind, bevor das referenzierende Schema verarbeitet wird.

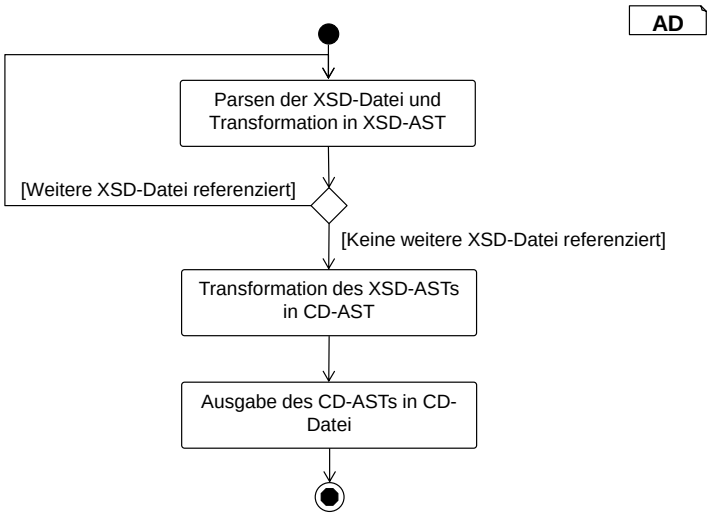


Abbildung 5.1: Aktivitätsdiagramm der Transformation von XSD-Dateien in CD-Dateien

5.2.1 DSL zur Verarbeitung von XSD-Dokumenten

Wie in Abschnitt 4.3 beschrieben muss zunächst eine MontiCore Grammatikbeschreibung für XSD-Dokumente entworfen werden, um einen Parser zu erhalten, der den benötigten AST aufbaut. Die Implementierung dieser DSL ist in das *xsd-lang* Projekt ausgelagert und als Maven-Abhängigkeit in das *xsd2cd* Projekt eingebunden. In diesem Unterabschnitt werden der Aufbau der Grammatik, ihre wesentlichen Elemente sowie sprachspezifische Besonderheiten beschrieben.

Die MontiCore XSD-Grammatik ist in die Dateien *XSDLiterals.mc* und *XSD.mc* aufgeteilt. *XSDLiterals.mc* enthält alle Definitionen von Literalen, während sich die Produktionen der Grammatik in der Datei *XSD.mc* befinden.

Aufbau der MontiCore XSD-Grammatik

Das Fundament jeder Grammatik stellen die von ihr verwendeten Literale und Schlüsselwörter dar. In der Datei *XSDLiterals.mc* sind die folgenden Literale definiert:

- Literale für XML-Auszeichnungszeichen (z.B. `</` oder `<!-`)
- Literal für XML-Identifier
- Literale für Tagnamen der unterschiedlichen XSD-Elemente
- Literal für Leerzeichen und Zeilenumbrüche
- Literale für Textinhalt (z.B. Attributwerte oder Elementinhalt)

Neben den Literalen besteht die Grammatik für XSD-Dokumente im Wesentlichen aus den Regeln für die über vierzig verschiedenen XML-Elementtypen, die im XSD-Standard definiert sind. Dabei bilden sie die mögliche Verschachtelung der Elemente ab, indem sie für jeden Elementtyp die als Kindelemente erlaubten Elementtypen definieren. Um die einzelnen Regeln möglichst einfach beschreiben zu können und trotz ihrer großen Anzahl insgesamt eine gute Lesbarkeit der Grammatik zu erreichen, wurden die AST-Klassen ähnlicher Elementtypen an geeigneten Stellen durch Implementierung von Interfaces zusammengefasst.

Abbildung 5.2 zeigt ein vereinfachtes Klassendiagramm der von MontiCore aus der *XSD*-Grammatikbeschreibung erzeugten AST-Klassen, um deren Vererbungshierarchie zu visualisieren.

Jede AST-Klasse wird automatisch durch das MontiCore Framework von der Klasse *ASTNode* abgeleitet. Darüber hinaus besteht die Möglichkeit, über die Grammatikbeschreibung weitere Beziehungen zwischen den AST-Klassen zu erzeugen. In der *XSD*-Sprache gibt es das Interface *ASTXSDElementContent*, das, direkt oder indirekt, von allen Produktionen außer der Startproduktion implementiert wird. Es fasst alle möglichen Inhaltstypen für XSD-Elemente zusammen. Das können je nach Elementtyp Text, Markup-Code wie Kommentare oder XSD-Elemente sein. Textinhalt wird durch die AST-Klasse *ASTXSDText* repräsentiert, die verschiedenen Arten Markup-Code durch die Klassen *ASTXSDComment*, *ASTXSDProcessingInstruction* und *XSDCDATA*.

XSD-Elemente

Um die gemeinsamen Eigenschaften aller XSD-Elemente an zentraler Stelle bündeln zu können, implementieren alle Element-Produktionen das Interface *ASTXSDElement*. Über AST-Regeln innerhalb der Grammatikbeschreibung werden der AST-Klasse des Interfaces jeweils ein *String*-Attribut für den Namen und den Namensraum des Elements sowie eine Liste mit XML-Attributen hinzugefügt. Das *ASTXSDElement* Interface kann außerdem

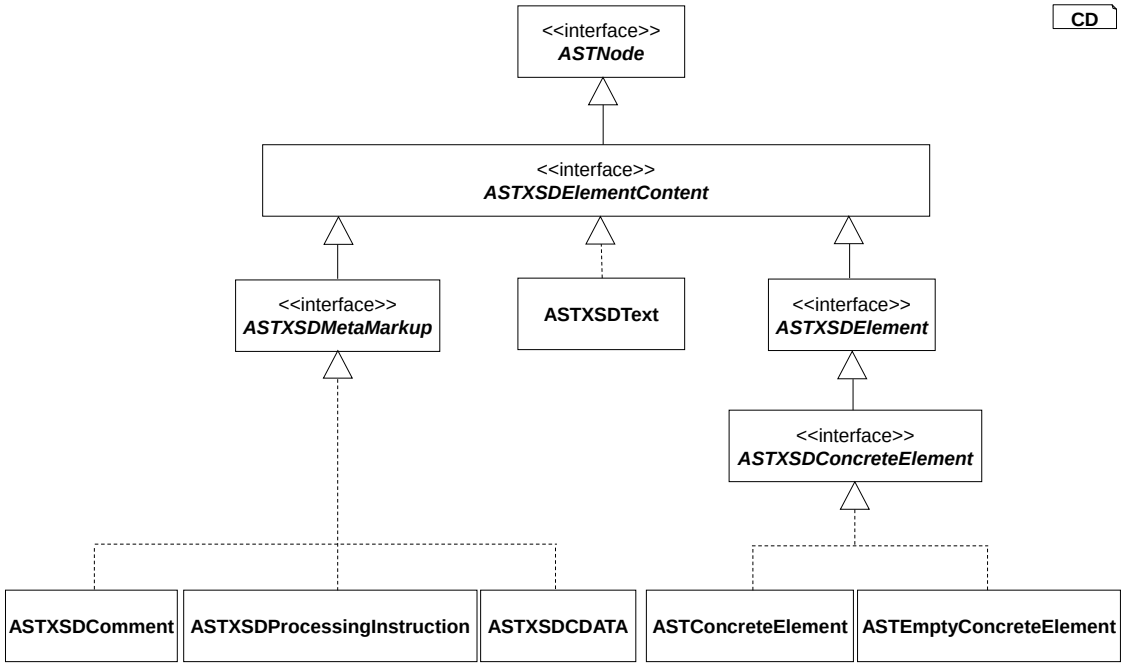


Abbildung 5.2: Klassendiagramm zur Darstellung der Hierarchie der AST-Klassen der MontiCore XSD-Grammatik (vereinfacht)

als Regelkomponente eingesetzt werden, wenn alle XSD-Elemente eine gültige Auswahl darstellen.

Für jeden Typ von XSD-Elementen existiert ein weiteres Interface sowie jeweils eine Regel für leere und nicht-leere Elemente dieses Typs. Die Unterteilung in leere und nicht-leere Elemente ist erforderlich, da diese sich syntaktisch unterscheiden. Die AST-Klasse des Interfaces trägt den Namen *ASTXSD*<Elementname>*Element*, die AST-Klassen der Regeln heißen *AST*<Element-name>*Element* beziehungsweise *ASTEmpty*<Elementname>*Element*.

Der Aufbau und die Funktionsweise der Regeln und damit der erzeugten AST-Klassen sind für alle Elementtypen identisch und werden daher im Folgenden am Beispiel der bereits in Quelltext 2.3 verwendeten *attribute*-Elemente erläutert.

Quelltext 5.1 zeigt die Deklaration des *XSDAttributeElement*-Interfaces für *attribute*-Elemente. Das Interface erlaubt die Einbindung leerer und nicht-leerer Elemente über eine einzige Regelkomponente sowie die Definition von zu implementierenden Zugriffsmethoden für die elementspezifischen Attribute durch AST-Regeln.

```
1 interface XSDAttributeElement extends XSDElement;
```

Quelltext 5.1: Definition eines gemeinsamen Interfaces für leere und nicht-leere XSD-Elemente des selben Typs am Beispiel von *attribute*-Elementen

AST-Regeln

In Quelltext 5.2 ist beispielhaft die AST-Regel für AST-Klassen, die das *XSDAttributeElement*-Interface implementieren aufgeführt.

```
1 ast XSDAttributeElement =
2     method public String getDefault() {}
3     method public String getFixed() {}
4     method public String getForm() {}
5     method public String getId() {}
6     method public String getNameAttribute() {}
7     method public String getRef() {}
8     method public String getType() {}
9     method public String getUse() {};
```

Quelltext 5.2: Definition von erforderlichen Methoden für Interfaces über AST-Regeln am Beispiel des *XSDAttributeElement*-Interfaces

Für jedes elementspezifische XML-Attribut des Elementtyps wird eine Zugriffsmethode deklariert, die von den AST-Klassen implementiert werden muss. Die *attribute*-Elemente treten, bedingt durch den Regelaufbau, als Kindelemente in Form einer Liste von *ASTXSDAttributeElement*-Objekten auf. Die Aufnahme der Methoden in das Interface erlaubt den Abruf der elementspezifischen Attribute, ohne zuvor eine Typumwandlung der Objekte in Instanzen der durch die Produktionen erzeugten AST-Klassen für leere oder nicht-leere Elemente vornehmen zu müssen. Das vereinfacht die Verarbeitung des ASTs, etwa bei der Transformation in ein Klassendiagramm.

Grammatikproduktionen und Syntaktische Prädikate

Quelltext 5.3 zeigt die Produktionen der *XSD*-Grammatik für leere und nicht-leere *attribute*-Elemente. Die Bedeutung der einzelnen Zeilen wird im Folgenden erläutert. d

1 und 12 Definieren die Namen der Produktionen. Aus ihnen werden die Namen der AST-Klassen abgeleitet.

2+5 und 13+16 Sorgen für die Implementierung des *XSDAttributeElement*-Interfaces.

3+4 und 14+15 Beinhalten die syntaktischen Prädikate zur Unterscheidung der Element-Regeln durch den Parser. Die Unterscheidung wird anhand des Zeichens getroffen, welches auf das letzte XML-Attribut folgt. Ist es das Schlusszeichen eines einfachen Start-Tags (>), wird die Regel *AttributeElement* verwendet. Ist es die Schlusszeichenfolge eines Empty-Element-Tags (/>), wird die Regel *EmptyAttributeElement* verwendet.

6-10 Stellen die rechte Seite der *EmptyAttributeElement*-Produktion dar. Sie beschreibt den Aufbau eines leeren *attribute*-Elements. Es beginnt mit einem *TAG_L*-Literal (<), gefolgt von einem optionalen *XSDIDENT*-Literal, das den Namensraum des Elements definiert. Anschließend muss das Schlüsselwort *attribute* als Tagname folgen, es wird durch das Literal *TAG_NAME_ATTRIBUTE* repräsentiert. Anschließend

```
1  / EmptyAttributeElement
2      implements
3      (TAG_L! (XSDIDENT ":")? TAG_NAME_ATTRIBUTE
4        XSDAttribute* TAG_R_SLASH!) =>
5      XSDAttributeElement =
6    TAG_L!
7      (namespace:XSDIDENT ":")?
8      name:TAG_NAME_ATTRIBUTE
9      attributes:XSDAttribute*
10   TAG_R_SLASH!;
11
12 / AttributeElement
13     implements
14     (TAG_L! (XSDIDENT ":")? TAG_NAME_ATTRIBUTE
15       XSDAttribute* TAG_R!) =>
16     XSDAttributeElement =
17   TAG_L!
18     (namespace:XSDIDENT ":")?
19     name:TAG_NAME_ATTRIBUTE
20     attributes:XSDAttribute*
21   TAG_R!
22     ( XSDMetaMarkup | XSDWS )*
23     XSDAnnotationElement?
24     ( XSDMetaMarkup | XSDWS )*
25     XSDSimpleTypeElement?
26     ( XSDMetaMarkup | XSDWS )*
27   TAG_L_SLASH!
28     (closingTagNamespace:XSDIDENT ":")?
29     closingTagName:TAG_NAME_ATTRIBUTE
30   TAG_R!;
```

Quelltext 5.3: Grammatikregeln für XSD-Elemente am Beispiel der *attribute*-Elemente

folgt eine Liste von XML-Attributen, welche durch die *XSDAttribute*-Produktion geparst werden. Das leere *attribute*-Element wird durch ein *TAG_R_SLASH*-Literal (/>) geschlossen.

17-30 Beinhalten die rechte Seite der *AttributeElement*-Produktion, welche den Aufbau eines nicht-leeren *attribute*-Elements definiert. Bis Zeile 21 sind die Regelkomponenten mit denen für leere *attribute*-Elemente identisch. In Zeile 21 wird das Start-Tag des nicht-leeren *attribute*-Elements durch ein *TAG_R*-Literal (>) geschlossen. Darauf folgen in Zeile 22 bis 26 die Regelkomponenten, welche den möglichen Inhalt des *attribute*-Elements definieren. An beliebigen Stellen von Markup-Code, Leerzeichen und Zeilenumbrüchen umschlossen, stellen ein optionales *annotation*-Element, gefolgt von einem ebenfalls optionalen *simpleType*-Element den erlaubten Inhalt dar. Ab Zeile 27 wird das End-Tag beschrieben. Es beginnt mit einem *TAG_L*-Literal, gefolgt von einem optionalen *XSDIDENT*-Literal, das den Namensraum des Elements definiert. Anschließend muss das Schlüsselwort *attribute* als Tagname folgen, es wird durch das Literal *TAG_NAME_ATTRIBUTE* repräsentiert. Geschlossen wird das End-Tag und damit das *attribute*-Element durch ein *TAG_R*-Literal.

Konflikte zwischen Identifiern

Im vorherigen Abschnitt wurde bereits die Auflösung der Mehrdeutigkeit zwischen den Produktionen für leere und nicht-leere Elemente des selben Typs mit Hilfe von syntaktischen Prädikaten beschrieben. Ein weiteres Problem stellen Überschneidungen zwischen den Wortmengen verschiedener Identifier dar. Durch die Integration von Antlr (vgl. Abschnitt 2.3.1) ist es möglich Einfluss auf die lexikalische Analyse der Eingabedokumente zu nehmen.

Konkret stellen einige der als Literale definierten Tagnamen Präfixe von möglichen elementspezifischen Attributen dar, welche über einen gemeinsamen Regulären Ausdruck für alle Identifier als das Literal *XSDIDENT* definiert sind. Dieses ist über die Antlr Option *testLiterals* so konfiguriert, dass der Lexer zunächst alle anderen Literale testet und somit bevorzugt. Für den Fall, dass ein Attributname einen Tagnamen als Präfix hat, wird das Literal für den Tagnamen verwendet, was zu einer ungültigen Tokenfolge für den Parser führt.

Der Zustandsautomat in Abbildung 5.3 illustriert, wie dieser Konflikt mit Hilfe der Antlr-Variable *waitingForTagName* aufgelöst wurde. Die Variable wird mit *false* initialisiert, entsprechend befindet sich der Automat zu Beginn der lexikalischen Analyse im Zustand *Kein Tagname erwartet*. Sobald durch ein < oder </ ein Element geöffnet wird, wird die Variable auf *true* gesetzt, der Automat wechselt in den Zustand *Tagname erwartet*. Durch die Abfrage der Variable auf den Wert *true* in den Definitionen der Literale für Tagnamen ist gewährleistet, dass sie nur von diesem Zustand aus verwendet werden können und damit gegenüber allgemeinen Identifiern bevorzugt werden. Ist das der Fall, wird die Variable anschließend auf *false* gesetzt, sodass der Automat wieder in den Zustand *Kein Tagname erwartet* wechselt und Attributnamen gelesen werden können.

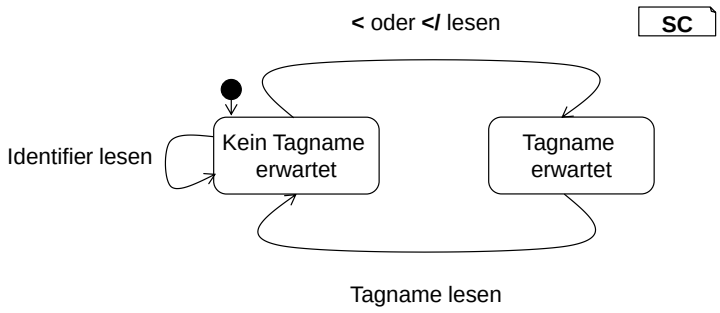


Abbildung 5.3: Zustandsautomat zur Verwendung der Antlr-Variable *waitingForTagName*

Erzeugte AST-Klassen

Abbildung 5.4 zeigt den AST, der beim Parsen des leeren *attribute*-Elements aus Zeile 45 von Quelltext 2.3 erzeugt wird, in Form eines Objektdiagramms.

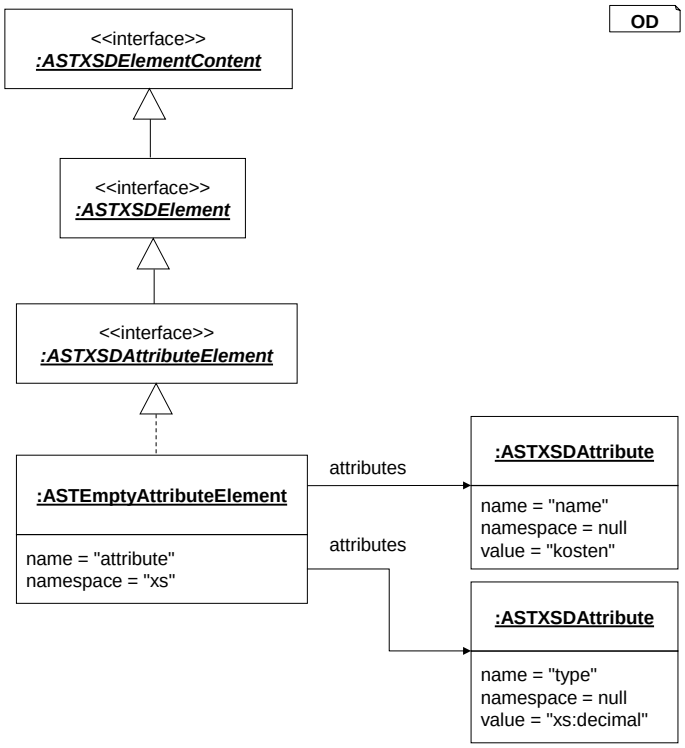


Abbildung 5.4: Ausschnitt des ASTs eines *EmptyAttributeElements* als Objektdiagramm

Der Parser erzeugt ein *ASTEmptyAttributeElement*-Objekt sowie ein *ASTXSDAttribute*-Objekt für jedes XML-Attribut, das das leere *attribute*-Element enthält. Das *ASTEmptyAttributeElement*-Objekt implementiert das Interface *ASTXSDAttributeElement* und damit dessen Oberinterfaces *ASTXSDElement* und *ASTXSDElementContent*.

Das Interface *ASTXSDElement* definiert verschiedene Zugriffsmethoden für die Verarbeitung von XML-Attributen, die unabhängig vom Elementtyp sind. Die wichtigsten sind *getAttribute*, welche einen Attributnamen als String entgegen nimmt und ein Objekt von Typ *ASTXSDAttribute* zurück gibt und *getAttributeValue*, die ebenfalls den Attributnamen als String empfängt und deren Rückgabewert der Wert des entsprechenden Attributs als String ist.

Da die Implementierung der Zugriffsmethoden unabhängig vom Elementtyp ist, stellt die Hilfsklasse *XSDASTHelper* entsprechende Methoden zur Verfügung, die von den eigentlichen AST-Klassen aufgerufen werden. Quelltext 5.4 zeigt die Implementierung der Methoden *getAttribute* und *getAttributeValue* in der Klasse *XSDASTHelper*.

Die Methode *getAttribute* erhält als Parameter ein *ASTXSDElement*-Objekt und den Attributnamen als String, um das entsprechende *ASTXSDAttribute*-Objekt des *ASTXSDElements*, falls vorhanden, zurückzugeben. Dazu werden ab Zeile 3 über die Methode *getAttributes* des *ASTXSDElements* alle im AST des Elements als *ASTXSDAttribute*-Objekte vorhandenen XML-Attribute abgerufen und in einer Schleife durchlaufen. Wie in Abbildung 5.4 ersichtlich, haben *ASTXSDAttribute*-Objekte die Attribute *name*, *namespace* und *value*, für die von MontiCore die entsprechenden Zugriffsmethoden generiert werden. In Zeile

```
1  static ASTXSDAttribute getAttribute(ASTXSDElement element, String
       ↪ attributeName) {
2    ASTXSDAttribute attributeOfName = null;
3    for (ASTXSDAttribute attribute : element.getAttributes()) {
4      if (attribute.getName().equals(attributeName)) {
5        attributeOfName = attribute;
6        break;
7      }
8    }
9    return attributeOfName;
10 }
11
12 static String getAttributeValue(ASTXSDElement element, String
       ↪ attributeName) {
13   String value = null;
14   ASTXSDAttribute attribute = XSDASTHelper.getAttribute(element,
         ↪ attributeName);
15   if (attribute != null) {
16     value = attribute.getValue();
17   }
18   return value;
19 }
```

Quelltext 5.4: Hilfsmethoden für den Zugriff auf Attribute von *ASTXSDElementen* in der Klasse *XSDASTHelper*

4 wird über die Zugriffsmethode *getName* das *name*-Attribut jedes *ASTXSDAttribute*-Objektes ausgelesen und mit dem an die Methode übergebenen Attributnamen verglichen. Sind die Namen identisch handelt es sich bei dem aktuellen *ASTXSDAttribute*-Objekt um das gesuchte. Es wird in der Variable *attributeOfName* gespeichert (Zeile 5) und der Schleifendurchlauf abgebrochen (Zeile 6). Nach Abbruch oder Beendigung der Schleife wird der Wert der Variablen *attributeOfName* in Zeile 9 zurückgegeben. Wurde kein Attribut mit dem gesuchten Namen gefunden ist der Rückgabewert, entsprechend der Initialisierung der Variablen in Zeile 2, *null*.

Die Methode *getAttributeValue* erhält als Parameter ein *ASTXSDElement*-Objekt und den Attributnamen als String, um den Wert des entsprechenden *ASTXSDAttribute*-Objekts zurückzugeben. In Zeile 14 werden dazu die Parameter an die zuvor beschriebene Methode *getAttribute* weitergegeben und der Rückgabewert in der Variable *attribute* gespeichert. Falls ein Attribut mit entsprechendem Namen gefunden wurde, wird in Zeile 16 über die Zugriffsmethode *getValue* dessen Wert ausgelesen und in der Variable *value* gespeichert. In Zeile 18 wird der Wert der Variablen zurückgegeben. Wurde von der Methode *getAttribute* kein Attribut mit dem beim Aufruf übergebenen Namen gefunden, ist der Rückgabewert, entsprechend der Initialisierung von *value* in Zeile 13, *null*.

Durch die Implementierung des Interfaces *XSDAttributeElement* muss die AST-Klasse *ASTEmptyAttributeElement* die über die AST-Regeln in Quelltext 5.2 definierten Zugriffsmethoden, auf die für *attribute*-Elemente spezifischen Attribute, zur Verfügung stellen. Quelltext 5.5 zeigt beispielhaft die Implementierung einer solchen Zugriffsmethode mit Hilfe der zuvor beschriebenen Hilfsmethoden der Klasse *XSDASTHelper*.

```
1 @Override
2 public String getAttributeValue(String attributeName) {
3   return XSDASTHelper.getAttributeValue(this, attributeName);
4 }
5
6 public String getType() {
7   return getAttributeValue("type");
8 }
```

Quelltext 5.5: Implementierung der Zugriffsmethoden für elementspezfische Attribute am Beispiel der AST-Klasse *ASTEmptyAttributeElement*

Die Zeilen 1-4 zeigen die Implementierung der vom Elementtyp unabhängigen Funktion *getAttributeValue* für das *ASTXSDElement*-Interface. Diese besteht aus einem Aufruf der *getAttributeValue*-Methode der Klasse *XSDASTHelper*, bei dem die Instanz der *ASTEmptyAttributeElement*-Klasse sowie der Attributname übergeben wird. Als Beispiel zeigen Zeile 6-8, anhand des *type*-Attributes, wie die elementspezifischen Methoden für den Attributzugriff implementiert werden. Durch die Implementierung der zuvor beschriebenen Zugriffsmethoden genügt ein Aufruf der *getAttributeValue*-Funktion mit dem Namen des entsprechenden Attributs. Ist das XML-Attribut in dem entsprechenden *attribute*-Element vorhanden, wird sein Wert, ansonsten *null*, zurückgegeben.

Für das *attribute*-Element, dessen AST in Abbildung 5.4 illustriert ist, würde die Methode *getName kosten* zurückgeben, die Methode *getType xs:decimal* und alle anderen elementspezifischen Zugriffsmethoden wie *getDefault null*, da die entsprechenden Attribute im *attribute*-Element aus Quelltext 2.3 nicht vorhanden sind.

Tests

Wie zu Beginn des Kapitels erwähnt wurde die Implementierung des Frameworks testgetrieben entwickelt. Die JUnit-Tests für die MontiCore XSD-Grammatik sind in der Klasse *XSDGrammarTests* implementiert. Die Klasse enthält die Methoden *testValid* und *testInvalid*, die mit Hilfe der MontiCore Klasse *GrammarTestParser* den Inhalt eines Verzeichnisses mit gültigen beziehungsweise ungültigen XSD-Dateien durch den von MontiCore, anhand der *XSD*-Grammatikbeschreibung, erzeugten Parser parsen.

Die Klasse *GrammarTestParser* stellt die Methoden *succeededParsingAllModels* und *failedParsingAllModelFiles* zu Verfügung, über die ermittelt werden kann ob alle Eingabedateien erfolgreich beziehungsweise fehlerhaft geparst wurden. Der Test *testValid* ist erfolgreich wenn alle gültigen XSD-Dateien erfolgreich geparst wurden, kommt es beim Parsen von mindestens einer Datei zu einem Fehler schlägt der Test fehl. Umgekehrt schlägt der Test *testInvalid* fehl, sobald eine der ungültigen XSD-Dateien erfolgreich geparst wurde und ist in allen anderen Fällen erfolgreich.

5.2.2 Transformationsworkflow

Nachdem aus dem XSD-Schema mit Hilfe des *xsd-lang*-Projektes ein XSD-AST erzeugt wurde, beginnt das *Xsd2CdTool*, wie im Aktivitätsdiagramm in Abbildung 5.1 beschrieben, mit dessen Transformation in ein Klassendiagramm. Dazu wird ein CD-AST durch

Einsatz der in [Sch12] implementierten MontiCore Grammatik für Klassendiagramme erzeugt. Deren Aufbau ist dort ausführlich beschrieben und wird daher an dieser Stelle nicht genauer erläutert. Über die Klasse *CDNodeFactory* können alle für den Aufbau eines Klassendiagramms benötigten AST-Knoten erzeugt werden.

Das *Xsd2CdTool* ruft nach dem Parsen des XSD-Schemas durch den *XSDParsingWorkflow* den *TransformationWorkflow* auf. Dieser instanziiert die Klasse *TransformationVisitor* welche die Transformation des ASTs in Form des Visitor-Entwurfsmusters [GHVJ09] implementiert. Am Ende der Transformation kann über die Methode *getResult* der *TransformationVisitor*-Klasse der AST des erzeugten Klassendiagramms als ein *ASTMCCompilationUnit* -Objekt abgerufen werden. Anschließend verwendet der Workflow einen Pretty-Printer [Sch12] um das Klassendiagramm in eine Datei auszugeben.

In den Folgenden Unterabschnitten wird zunächst der durch das Visitor-Entwurfsmuster definierte Ablauf der Transformation beschrieben und anschließend die Umwandlung der zentralen XSD-Elementtypen in Komponenten eines Klassendiagramms anhand des Beispielschemas aus Quelltext 2.3 erläutert. Eine schriftliche Darstellung aller von der Klasse *TransformationVisitor* durchgeführten Transformationen ist aufgrund von Menge und Umfang an dieser Stelle nicht möglich, jedoch in Form einer Javadoc [www14j] Dokumentation innerhalb von Anhang C vorhanden.

Es gibt eine Vielzahl von Möglichkeiten das Datenmodell eines XSD-Schemas in Klassendiagramme beziehungsweise Java-Klassen abzubilden. Im Rahmen der Konzeption wurde in Abschnitt 4.3 bereits festgestellt, dass die geforderte Kompatibilität zum JiBX-Framework bei der Erzeugung der Klassendiagramme berücksichtigt werden muss, da die *cd2java*-Komponente auf deren Basis die Java-Klassen erzeugt. Dadurch bedingt ist die Implementierung des Transformations-Visitors sehr komplex. Um den Umfang dieses Abschnitts zu begrenzen und die Nachvollziehbarkeit zu erhöhen werden daher nur für die Erläuterung der Beispiele relevante Ausschnitte aus Methoden der Klasse *TransformationVisitor* vorgestellt. Eine Bewertung der durch JiBX vorgegebenen Umwandlungsstrategien ist im Rahmen dieser Arbeit nicht vorgesehen, weshalb im Folgenden nur deren Umsetzung beschrieben und nicht auf mögliche Nachteile eingegangen wird.

Ablauf der Transformation

Wie zuvor erwähnt ist der Ablauf der Transformation durch das verwendete Visitor-Entwurfsmuster geprägt. Das Entwurfsmuster erlaubt die automatisierte Verarbeitung des ASTs indem dieser in einer bestimmten Reihenfolge traversiert wird. Dabei wird zum Besuchszeitpunkt jedes Knotens eine typabhängige Methode aufgerufen, die im Falle des XSD-ASTs eine vom Elementtyp abhängige Transformation erlaubt.

Der AST wird beginnend vom Wurzelknoten in der Reihenfolge einer Tiefensuche durchlaufen, was bei den XSD-Elementen und dazugehörigen AST-Klassen des Beispielschemas der folgenden Reihenfolge entspricht:

1. schema-Element, ASTSchemaElement (Zeile 2)

2. complexType-Element, ASTComplexTypeElement (Zeile 4)

3. attribute-Element, ASTEmptyAttributeElement (Zeile 9)

4. attribute-Element, ASTEmptyAttributeElement (Zeile 10)

5. sequence-Element, ASTSequenceElement (Zeile 5)

6. element-Element, ASTEmptyElementElement (Zeile 6)

7. element-Element, ASTEmptyElementElement (Zeile 7)

8. complexType-Element, ASTComplexTypeElement (Zeile 12)

9. sequence-Element, ASTSequenceElement (Zeile 13)

10. element-Element, ASTEmptyElementElement (Zeile 14)

11. element-Element, ASTEmptyElementElement (Zeile 15)

12. element-Element, ASTEmptyElementElement (Zeile 16)

[. . .]

33. element-Element, ASTElementElement (Zeile 3)

Bedingt durch diese Reihenfolge kann es während der Transformation nötig sein vom aktuellen Knoten aus, neben Eltern- und Kindelementen, auch weitere Elemente des ASTs, wie beispielsweise referenzierte Inhaltstypen, zu untersuchen. Die Erzeugung des CD-ASTs findet trotzdem dem Ablaufschema des Visitor-Entwurfsmusters entsprechend statt.

Beim Instanziieren der Klasse *TransformationVisitor* durch den *TransformationWorkflow* wird zunächst ein *ASTCDDefinition*-Objekt erzeugt, welches die Wurzel des CD-ASTs darstellt. Als Name für das Klassendiagramm wird der Name der XSD-Datei verwendet, wobei eventuell kleine Änderungen vorgenommen werden um den UML/P-Konventionen für die Benennung von Klassendiagrammen gerecht zu werden.

Anschließend startet der *TransformationWorkflow* den Durchlauf des *TransformationVisitors* mit Hilfe der MontiCore Klasse *AbstractVisitor*. Diese traversiert den XSD-AST in der zuvor beschriebenen Reihenfolge und ruft die typabhängigen Transformationsmethoden des *TransformationVisitors* auf. Diese besitzen alle den Namen *visit* und einen Parameter des entsprechenden Objekttypen. Für jeden Knoten des ASTs überprüft der *AbstractVisitor* ob eine passende *visit*-Methode existiert und ruft diese gegebenenfalls auf. Die folgenden Unterabschnitte beschreiben die Umwandlung von Datentypen und die, für die Transformation des Beispiels, benötigten Teile der *visit*-Methoden.

Umwandlung von *schema*-Elementen

Wie in der Einleitung von Abschnitt 5.2 erläutert, ist es erforderlich zunächst alle referenzierten Schemadateien zu parsen und transformieren, damit alle dort definierten Typen während der Transformation des referenzierenden Schemas verfügbar sind. Das *schema*-Element stellt den Wurzelknoten jedes XSD-ASTs dar und wird daher immer als erstes behandelt, weshalb sich die *visit*-Methode für *ASTSchemaElement*-Objekte für diese Tätigkeit anbietet.

Als erster Schritt der Transformation wird die Packagebezeichnung für das Klassendiagramm aus dem *targetNamespace*-Attribut des *schema*-Elements abgeleitet. Anschließend werden über die Methode *getXSDIncludeElement* des *ASTSchemaElement*-Objekts alle *include*-Elemente des Schemas abgerufen und in einer Schleife behandelt. Enthält das *include*-Element ein *schemaLocation*-Attribut mit einen korrekten Dateipfad als Wert, wird die referenziert Schemadatei geparst und mit dem Besuch ihres *schema*-Elements fortgefahren.

Abbildung 5.5 zeigt das *schema*-Element des Beispielschemas und das Klassendiagramm nach dessen Transformation. Außer der Einbindung des *xs*-Namensraumes über ein *xmlns*-Attribut enthält das *schema*-Element keine weiteren Informationen, die die Erzeugung des Klassendiagramms beeinflussen. Daher werden Name und Packagebezeichnung aus dem Dateinamen (*Rechnung.xsd*) abgeleitet.

```
1 <xs:schema xmlns:xs=
    ↪ "http://www.w3.org/2001/
    ↪ XMLSchema">
2 [...]
3 </xs:schema>
```

XSD

```
1 package Rechnung;
2
3 classdiagram Rechnung{
4 }
```

CD

Abbildung 5.5: Ergebnis der Transformation des Beispielschemas: Transformation des *schema*-Elements

Umwandlung von XSD-Standardtypen

Für XSD-Standardtypen wie *xs:dateTime* und *xs:decimal* existieren keine eigenen Elemente innerhalb des Schemas. Sie treten nur als Typangaben innerhalb anderer Elemente auf und müssen daher in den *visit*-Methoden für diese Elemente einheitlich in Java-Typen transformiert werden.

Aus diesem Grund wurde im *common*-Projekt innerhalb der Klasse *Constants* eine Map angelegt. Jeder Eintrag der Map besteht aus einem Schlüssel und einem zugehörigen Wert, die beide vom Typ *String* sind. Der Schlüssel ist der Name des XSD-Standardtyps und der Wert der zugeordnete Java-Typ. Anhang A enthält eine Tabelle mit diesen (JiBX-kompatiblen) Typzuordnungen.

Innerhalb jeder Komponente des Migrationsframeworks können so über die Methoden *containsKey* und *get* der Map für referenzierte Typen überprüft werden, ob sie ein Standardtyp sind und gegebenenfalls der entsprechende Java-Typ ermittelt werden. Innerhalb der *TransformationVisitor*-Klasse der *xsd2cd*-Komponente wird der Zugriff auf die Map unter anderem durch die Methoden *isStandardType* und *getUMLPType* gekapselt.

Umwandlung von einfachen Inhaltstypen

Benutzerdefinierte, einfache Inhaltstypen werden durch *simpleType*-Elemente von Standardtypen abgeleitet. In dem hier verwendeten Beispielschema wurde aufgrund des Um-

fangs auf sie verzichtet. Dennoch soll an dieser Stelle kurz die Transformationsstrategie beschrieben werden, da einfache Inhaltstypen in allen größeren XML-Schema Definitionen zum Einsatz kommen.

Der Umgang mit *simpleType*-Elementen ist abhängig von der Art der Ableitung, die sie vorschreiben:

- *Einschränkungen* werden mit Ausnahme von Enumerationen ignoriert. Stattdessen wird der Typ verwendet, von dem abgeleitet werden soll. Weitere Ableitungen übergeordneter Typen werden ebenfalls eliminiert, bis der zugrunde liegende Standardtyp und der ihm zugeordnete Java-Typ ermittelt ist.
- *Einschränkungen* durch Enumeration der zulässigen Werte werden in Enum-Klassen transformiert, deren Name aus dem *name*-Attribute des *simpleType*-Elements abgeleitet werden.
- *Auflistungen* werden dem Typ *String* zugeordnet.
- *Vereinigungen* werden ebenfalls dem Typ *String* zugeordnet.

Die Ermittlung des zugrunde liegenden Standardtyps wird von der Methode *getBaseTypeFromSimpleType* gekapselt. Zur Erzeugung der Enum-Klasse verwendet die *visit*-Methode für *ASTSimpleTypeElement*-Objekte die Prozedur *createASTCDEnum*, die aus einem übergebenen Namen und einer Liste von *ASTXSDEnumerationElement*-Objekten ein *ASTCDEnum*-Objekt mit entsprechenden Konstanten erzeugt und dem Wurzelknoten des CD-ASTs hinzufügt. Diese Enum-Klasse wird an allen Stellen, an denen das zugehörige *simpleType*-Element referenziert wird, als Zieltyp verwendet.

Umwandlung von komplexen Inhaltstypen

Die zentralen XSD-Elemente zur Erzeugung von Klassen sind die *complexType*-Elemente. Sie definieren komplexe XML-Inhaltstypen mit verschiedenen Attributen und Kindelementen, die auf eine entsprechende Klassenstruktur abgebildet werden müssen.

Sie können an verschiedenen Stellen eines Schemas auftreten und erfordern nicht immer die Erzeugung einer eigenen Klasse. Wenn es sich beispielsweise um ein *complexType*-Element innerhalb eines *element*-Elements handelt, wird die Klasse aus dem *element*-Element erzeugt und um die aus der Definition des komplexen Inhaltstyps resultierenden Attribute und Assoziationen erweitert.

Quelltext 5.6 zeigt die *visit*-Methode für *ASTComplexTypeElement*-Objekte. In Zeile 2 wird mit Hilfe der Methode *isInline* überprüft ob aus dem *complexType*-Element eine eigene Klasse erzeugt werden muss. Ist das der Fall, wird die ab Zeile 7 beschriebene Methode *createASTCDClass* aufgerufen. Sie erhält als Parameter ein *ASTNode*-Objekt und den gewünschten Klassennamen als String.

Als Klassenname wird von der *visit*-Methode der Wert des *name*-Attribute des *complexType*-Elements übergeben. In Zeile 8 wird dieser durch die Methode *getUMLPIdentifier* in einen gültigen und JiBX-kompatiblen Identifier transformiert. Da die Methode nicht nur

```
1 public void visit(ASTComplexTypeElement element) {
2   if (!isInline(element)) {
3     createASTCDClass(element, element.getNameAttribute());
4   }
5 }
6
7 private ASTCDClass createASTCDClass(ASTNode element, String name) {
8   name = getUMLPIdentifier(name, true);
9   ASTCDClass cdClass = CDNodeFactory.createASTCDClass();
10  cdClass.setName(name);
11  classDiagram.getCDClasses().add(cdClass);
12  mapNode(element, cdClass);
13  mapClass(cdClass);
14  return cdClass;
15 }
```

Quelltext 5.6: Visitor-Methode für die Behandlung von *ASTCompleyTypeElement*-Knoten und Methode zur Erzeugung von *ASTCDClass*-Knoten

in diesem Kontext verwendet wird, kann über den zweiten Parameter angegeben werden, ob der zurückgegebene Identifier mit einem Großbuchstaben beginnen muss. Das ist bei Klassennamen der Fall.

Anschließend wird in den Zeilen 9 bis 11 mit Hilfe der *CDNodeFactory* ein *ASTCDClass*-Knoten erzeugt, mit dem an die Methode übergebenen Klassennamen versehen und dem AST des Klassendiagramms hinzugefügt.

Die Klasse *TransformationVisitor* pflegt verschiedene Maps, in denen die erzeugten Klassendiagrammkomponenten nach unterschiedlichen Kriterien katalogisiert werden um den Zugriff auf sie zu erleichtern. In der Map *src2target* wird jedem AST-Knoten des Quell-ASTs, also des XSD-ASTs, eine Liste aus Knoten des Ziel- also CD-ASTs zugeordnet, die aus ihm erzeugt wurden. Die Methode *mapNode* legt, falls noch kein Eintrag für den Quellknoten existiert, einen solchen an und fügt der Liste den, als zweiten Parameter übergebenen, Zielknoten hinzu. In der Map *classMap* werden Verweise auf die erzeugten *ASTCDClass*-Knoten über die Methode *mapClass*, mit ihrem Namen als Schlüssel, abgelegt.

Die entsprechenden Methoden zur Katalogisierung der erzeugten Klasse werden von der Methode *createASTCDClass* in den Zeilen 12 und 13 aufgerufen.

Abbildung 5.6 zeigt das Ergebnis der Transformation anhand der relevanten XSD-Elemente und den daraus erzeugten Teilen des Klassendiagramms. Aus dem *complexType*-Element in Zeile 4 des Beispielschemas wird auf die zuvor beschriebene Weise von der Visitor-Methode die Klasse *Rechnungtype* erzeugt. In den folgenden Abschnitten wird beschrieben, wie sie durch die Visitoren der Kindelemente um Attribute und Assoziationen ergänzt wird.

```
<xs:complexType
    ↪ name="rechnungtype">
[...]
</xs:complexType>
```

XSD

```
class Rechnungtype{}
```

CD

Abbildung 5.6: Ergebnis der Transformation des Beispielschemas: Transformation eines *complexType*-Elements

Umwandlung von Attributen

Quelltext 5.7 zeigt den für das Beispielschema relevanten Ausschnitt der Implementierung der *visit*-Methode für *ASTXSDAttributeElement*-Objekte. Sie wird von den Visitoren für *ASTAttributeElement*- und *ASTEmptyAttributeElement*-Knoten aufgerufen, da die Transformationsstrategie für leere und nicht-leere *attribute*-Elemente gleich ist. Die Implementierung des gemeinsamen Interfaces macht diese Vorgehensweise möglich.

```
public void visit(ASTXSDAttributeElement element) {
  if (isParentComplexType(element){
    if (element.getType() != null) {
      if (isStandardType(element.getType())) {
         String attributeName = element.getNameAttribute();
         addAttributeToClass(getParentClass(element), attributeName,
            ↪  getUMLPType(element.getType()), isChoice);
      }
    }
  }
}

private void addAttributeToClass(ASTCDClass cdClass, String name,
    ↪ String type) {
  ASTCDAttribute cdAttribute = createASTCDAttribute(name, type);
  cdClass.getCDAttributes().add(cdAttribute);
}

private ASTCDAttribute createASTCDAttribute(String name, String type
    ↪ ) {
  name = getUMLPIdentifier(name, false);
  ASTCDAttribute cdAttribute = CDNodeFactory.createASTCDAttribute();
  cdAttribute.setName(name);
  ASTSimpleReferenceType cdType = TypesNodeFactory.
     ↪ createASTSimpleReferenceType();
  List<String> listNames = new ArrayList<String>();
  listNames.add(type);
  cdType.setName(listNames);
  cdAttribute.setType(cdType);
  return cdAttribute;
}
```

Quelltext 5.7: Visitor-Methode für die Behandlung von *ASTXSDAttributeElement*-Knoten und Methoden zum Hinzufügen von *ASTCDAttribute*-Knoten zu *ASTCDClass*-Knoten

In den Zeilen 2-4 wird überprüft ob das *attribute*-Element eine Kombination aus Eigenschaften erfüllt um daraus ein Attribut im Klassendiagramm zu erzeugen. Im Quelltext sind nur die für die Elemente aus dem Beispielschema relevanten Abfragen und Pfade enthalten. Insgesamt gibt es wesentlich mehr mögliche Konstellationen aus Eigenschaften, die aufgrund der JiBX-Kompatibilität alle eine unterschiedliche Behandlung erfordern. So beträgt der Umfang der *visit*-Methode für *ASTXSDAttributeElemente* tatsächlich 142 anstatt der 10 hier aufgeführten Zeilen.

Für die *attribute*-Elemente in Zeile 9 und 10 des Beispielschemas treffen folgende Eigenschaften zu, so dass jeweils die Zeilen 5 und 6 ausgeführt werden:

- Das Elternelement ist ein *complexType*-Element, was von der Methode *isParentComplexType* überprüft wird. (Zeile 2)
- Das *type*-Attribut ist vorhanden und enthält einen Wert. (Zeile 3)
- Der über das *type*-Attribut referenzierte Typ ist ein XSD-Standardtyp, was von der bereits im Unterabschnitt *Umwandlung von XSD-Standardtypen* erwähnten Methode *isStandardType* verifiziert wird. (Zeile 4)

Da das Elternelement vom Typ *complexType* ist, existiert bereits eine Klassenrepräsentation im AST des erzeugten Klassendiagramms, der anhand der Informationen des *attribute*-Element ein Attribut hinzugefügt werden kann. Dies wird über den Aufruf der ab Zeile 12 definierten Methode *addAttributeToClass* in Zeile 6 erreicht. Die Methode *getParentClass* liefert mit Hilfe der *src2target*-Map das aus dem Elternelement erzeugte *ASTCDClass*-Objekt zurück, welches zusammen mit dem Namen und dem durch die Methode *getUMLPType* bestimmten Java-Typ des Attributs an die *addAttributeToClass*-Methode übergeben wird.

In Zeile 13 ruft diese zunächst die ab Zeile 17 definierte Methode *createASTCDAttribute* auf. Diese kapselt die aufgrund der Struktur des CD-ASTs recht umfangreiche Instanziierung von *ASTCDAttribute*-Knoten anhand eines übergebenen Namens und Typs. Das so erzeugte Objekt wird in Zeile 14 dem von der *visit*-Methode übergebenen *ASTCDClass*-Knoten hinzugefügt, wodurch die entsprechende Klasse im Klassendiagramm ein neues Attribut erhält.

```
1 <xs:complexType
    ↪ name="rechnungtype">
2   <xs:attribute
      ↪ name="rechnungsnummer"
      ↪ type="xs:string" />
3   <xs:attribute
      ↪ name="datum"
      ↪ type="xs:string" />
4 </xs:complexType>
```

XSD

```
1 class Rechnungtype{
2   String rechnungsnummer;
3   String datum;
4 }
```

CD

Abbildung 5.7: Ergebnis der Transformation des Beispielschemas: Transformation von *attribute*-Elementen

Abbildung 5.7 zeigt die beiden *attribute*-Kindelemente des im letzten Abschnitt behandelten *complexType*-Elements und die aus ihm erzeugte Klasse nach der Transformation der

attribute-Elemente. Wie zuvor beschrieben wird der AST-Knoten der die Klasse repräsentiert von der *visit*-Methode ausfindig gemacht und um die entsprechenden *ASTCDAttribute*-Knoten erweitert. Die Klasse verfügt somit nach der Transformation über die beiden Attribute *rechnungsnummer* und *datum* vom Typ *String*.

Umwandlung von Elementen

Der dritte, für die Transformation des Beispielschemas relevante, Elementtyp sind *element*-Elemente. In Quelltext 5.8 sind die entsprechende *visit*-Methode und die von ihr zur Erweiterung des CD-ASTs verwendete Methode *addCompositionToClassdiagramm* dargestellt. Wie im vorherigen Abschnitt handelt es sich dabei nur um die für die Umwandlung des Beispielschemas benötigten Codeteile.

```
1  public void visit(ASTXSDElementElement element) {
2    if (isParentSequence(element)) {
3      if (element.getType() != null) {
4        type = getUMLPIdentifier(element.getType(), true);
5        addCompositionToClassdiagram(element.getMinOccurs(), element.
             ↪ getMaxOccurs(), getParentClass(element).getName(),
             ↪ element.getNameAttribute(), type);
6      }
7    }
8    else if (isParentSchema(element)) {
9      if (isRootElement(element)) {
10       if (element.getType() != null) {
11         getClass(element.getType()).setModifier(createASTModifier("
               ↪ Element"));
12     }
13   }
14 }
15
16 private void addCompositionToClassdiagram(String minOccurs, String
       ↪ maxOccurs, String parentName, String childName, String
       ↪ childType) {
17     ASTCDAssociation cdAssociation = createASTCDAssociation(
           ↪ minOccurs, maxOccurs, parentName, childName, childType);
18     classDiagram.getCDAssociations().add(cdAssociation);
19 }
```

Quelltext 5.8: Visitor-Methode für die Behandlung von *ASTXSDElementElement*-Knoten und Methode zur Erzeugung von *ASTCDAssociation*-Knoten

Die *visit*-Methode ist ähnlich der Methode für *ASTXSDAttributeElemente* aufgebaut. Sie besteht aus verschiedenen Pfaden die, abhängig von den Eigenschaften des *element*-Elements, AST-Knoten erzeugen und so das Klassendiagramm erweitern. Auf die *element*-Elemente in Zeile 6 und 7 treffen die beiden folgenden Eigenschaften zu:

- Das Elternelement ist ein *sequence*-Element, was von der Methode *isParentSequence* überprüft wird. (Zeile 2)
- Das *type*-Attribut ist vorhanden und enthält einen Wert. (Zeile 3)

Für diese Elemente werden daher die Zeilen 4 und 5 ausgeführt, in denen zunächst das *type*-Attribute abgerufen und in einen gültigen Java-Identifier überführt wird. Anschließend wird die ab Zeile 16 definierte *addCompositionToClassdiagramm*-Methode aufgerufen. Ihr werden die Werte der Attribute *minOccurs*, *maxOccurs*, *name*, der Name der über die Methode *getParentClass* ermittelten, aus dem übergeordneten *complexType*-Element erzeugten Klasse und der zuvor bestimmte Typen-Identifier übergeben.

Die Methode *addCompositionToClassdiagramm* gibt ihre Parameter an die Methode *createASTCDAssociation* weiter, welche ein *ASTCDAssociation*-Objekt mit den entsprechenden Eigenschaften erzeugt. Da die AST-Struktur für Assoziationen aufgrund der verschiedenen Assoziationstypen komplex ist, ist diese Methode sehr umfangreich und es wird daher an dieser Stelle auf ihre Darstellung verzichtet. Da für die Darstellung der in XML Schema Definitionen möglichen Beziehung Kompositionen ausreichen, haben die erzeugten Assoziationen immer den Typ *Composition*. Anhand der *minOccurs*- und *maxOccurs*-Attribute wird die Kardinalität der Komponente bestimmt, anhand des *name*-Attributs der Rollenname.

Abschließend wird in Zeile 18 dem Klassendiagramm der *ASTCDAssociation*-Knoten hinzugefügt.

Das *element*-Element in Zeile 3 besitzt die beiden folgenden Eigenschaften:

- Das Elternelement ist das *schema*-Element, was von der Methode *isParentSchema* überprüft wird. (Zeile 8)
- Es ist das einzige Kindelement des *schema*-Elements vom Typ *element*, was von der Methode *isRootElement* überprüft wird. (Zeile 9)
- Das *type*-Attribut ist vorhanden und enthält einen Wert. (Zeile 10)

Aus der Kombination dieser Eigenschaften folgt, dass das *element*-Element das Wurzelelement in der durch das Schema beschriebenen XML-Sprache ist (siehe Abschnitt 2.1.1). In der Sprache der Klassendiagramme existiert kein entsprechendes Konstrukt. Um diese Information während der Transformation dennoch zu erhalten, wird in Zeile 11 anhand des *type*-Attributs mit Hilfe der *getClass*-Methode über die *classMap*-Map die Klasse des Elements bestimmt und ihr ein, durch die Methode *createASTModifier* erzeugter, *«Element»*-Stereotyp hinzugefügt.

Abbildung 5.8 zeigt das Ergebnis der Transformation nach der Umwandlung der für die Klasse *Rechnungtype* relevanten *element*-Elemente anhand der daraus erzeugten Teile des Klassendiagramms. Die Klasse wurde um den «Element»-Stereotyp erweitert und dem Klassendiagramm die Kompositionen *RechnungtypeToKundetype* und *RechnungtypeToPostentype* hinzugefügt. Die Klassen *Kundetype* und *Postentype* werden analog zur Klasse *Rechnungtype* aus den *complexType*-Elementen in Zeile 12 und 19 des Beispielschemas erzeugt.

Umwandlung weiterer Elementtypen

In diesem Abschnitt wurde die Umwandlung von XML-Schema Deklarationen in Klassendiagramme durch die in Form des Visitor-Entwurfsmusters implementierte AST-Transfor-

```xml
<xs:element name="rechnung" type="rechnungtype"/>
<xs:complexType name="rechnungtype">
  <xs:sequence>
    <xs:element name="kunde" type="kundetype" />
    <xs:element name="posten" type="postentype" />
  </xs:sequence>
</xs:complexType>
```

XSD

```
<<Element>> class Rechnungtype{
  String rechnungsnummer;
  String datum;
}
composition RechnungtypeToKundetype [1] Rechnungtype -> (kunde)
    ↪ Kundetype [1];
composition RechnungtypeToPostentype [1] Rechnungtype -> (posten)
    ↪ Postentype [1];
```

CD

Abbildung 5.8: Ergebnis der Transformation des Beispielschemas: Transformation von *element*-Elementen

mation anhand eines Beispiels erläutert. Das vollständige Klassendiagramm ist in Anhang B.1 als Diagramm vorhanden. Obwohl nur ein kleiner Anteil der tatsächlich implementierten *visit*-Methoden in reduzierter Form dargestellt wurde, erlauben die Ausführungen, aufgrund der einheitlichen Struktur, die übrigen Teile der Klasse *TransformationVisitor* anhand der Javadoc Dokumentation in Anhang C nachzuvollziehen.

Tests

Die Transformation von XML-Schema Definitionen in Klassendiagramme wird ebenfalls automatisiert getestet. Die Klasse *TransformationWorkflowTests* enthält Tests die den gesamten *xsd2cd*-Workflow auf verschiedenen XSD-Dateien aufrufen und im Fall eines Fehlers während der Transformation fehlschlagen.

Die Klasse *TransformationVisitorTests* implementiert einen Test zu Überprüfung der durch die Klasse *TransformationVisitor* durchgeführten Transformation ganzer XSD-Dateien in CD-Dateien. Dazu wurden verschiedene Testfälle angelegt, die jeweils aus XSD- und CD-Dateien mit dem selben Dateinamen bestehen. Die Testmethode *testTransformation* parst zunächst alle im Testordner vorhandenen XSD-Dateien. Anschließend wird für jeden so erzeugten XSD-AST der folgende Testablauf durchgeführt:

1. Die *TransformationVisitor*-Klasse wird instanziiert und der XSD-AST mit ihrer Hilfe in einen CD-AST überführt.
2. Die zugehörige CD-Datei wird mit Hilfe des in [Sch12] entwickelten *CDTool* eingelesen und dadurch ebenfalls in einen CD-AST überführt.
3. Über die *toString*-Methode der *ASTCDDefinition*-Objekte, welche die Wurzel der CD-ASTs darstellen, werden das durch Transformation erzeugte Klassendiagramm

und das in der CD-Datei definierte erwartete Klassendiagramm in Strings umgewandelt.

4. Die beiden Strings werden über die JUnit Methode *assertEquals* miteinander verglichen. Sind die Strings nicht identisch, schlägt der Test fehl.

Der Vergleich der textuellen Darstellung der ASTs hat den Vorteil, dass eventuelle Unterschiede als Ausgabe des Tests direkt in menschenlesbarer Form vorliegen.

5.3 Generierung von Java-Klassen aus Klassendiagrammen

Der Codegenerator zur Erzeugung der Datenbindungsklassen aus Klassendiagrammen ist im *cd2java*-Projekt implementiert und wird durch das *cd2java-use*-Projekt aufgerufen. Das *cd2java*-Projekt ist in vier Packages aufgeteilt, deren Inhalt im Folgenden beschrieben wird.

Das Package *xsd2dsl.cd2java.configure* enthält die von *UMLPTool* abgeleitete Klasse *Cd-2JavaGenerator*, die aufgerufen werden kann um den Generierungsprozess zu starten. Außerdem beinhaltet sie das Starttemplate *StartAllOutput.ftl*. Dieses instanziiert mit Hilfe der ebenfalls im Package enthaltenen Klasse *ConfigureCd2JavaGenerator* einige Hilfsklassen und ruft anschließend die Haupttemplates zur Erzeugung von Klassen- und Enum-Dateien auf.

Alle weiteren Templates sind im Package *xsd2dsl.cd2java.core* enthalten und greifen auf die Hilfsklassen und Kalkulatoren aus dem Package *xsd2dsl.cd2java.helper* zu. Sie werden in Abschnitt 5.3.1 beschrieben. Im Package *xsd2dsl.cd2java.cocos* sind einige Kontextbedingungen [Sch12] definiert, die von Klassendiagrammen erfüllt werden müssen um die Generierung von kompilierbarem Java-Code zu garantieren. Sie wurden aus dem *DEx*-Projekt [www14p] übernommen.

5.3.1 Templates

In diesem Abschnitt werden einige der zur Codegenerierung entwickelten Templates vorgestellt und dabei die Struktur der von der *cd2java*-Komponente erzeugten Java-Klassen beschrieben. Wie bereits erwähnt werden durch das Starttemplate verschiedene Hilfsklassen instanziiert und in FreeMarker-Variablen gespeichert, sodass sie in allen Subtemplates verfügbar sind. Tabelle 5.1 zeigt eine Übersicht der Hilfsklassen, ihrer Aufgaben und der Variablennamen.

Anhand des CD-ASTs werden im Haupttemplate über die Methode *getCDClasses* alle im Klassendiagramm enthaltenen Klassen ausgelesen und durch die Methode *includeTemplates* des Template-Operators jeweils mit dem im folgenden Abschnitt beschriebenem Template *ClassMain.ftl* aufgerufen, um daraus Java-Code zu erzeugen.

Generierung von Klassen

Quelltext 5.9 zeigt das Haupttemplate zur Transformation von Klassen. Auf das aus dem *StartAllOutput*-Template übergebene Klassenobjekt vom Typ *ASTCDClass* kann über die

Hilfsklasse	*Aufgabe*	*FreeMarker-Variable*
TypeHelper	Analyse der Typen eines Klassendiagramms.	typeHelper
GetterSetter-Helper	Kapselung der Berechnung von Namen für Zugriffsmethoden.	getterSetterHelper
Model2Target-Converter	Kapselung der Umwandlung von Klassendiagrammfragmenten in Java-Code. Aktuell zählt dazu nur die Transformation von Import-Statements aus Klassendiagrammen in Java Import-Statements.	m2tConv
*Details	AST-Klassen-spezifische Berechnungen.	-

Tabelle 5.1: Hilfsklassen des Codegenerators und ihre Aufgaben

ast-Variable zugegriffen werden. In Zeile 1 wird zunächst die Hilfsklasse *ClassDetails* durch Übergabe des *ASTCDClass*-Objektes instanziiert und der Variable *helper* zugewiesen.

```
1 <#assign helper = op.instantiate("xsd2dsl.cd2java.helper.
    ↪ ClassDetails", [ast])>
2 <#assign isInner = helper.isInner()>
3
4 <#if !isInner>
5   ${op.defineValue("package", helper.getPackage())}
6   ${op.callTemplate("xsd2dsl.cd2java.core.Class", package + "." +
      ↪ ast.printName(), ast)}
7 </#if>
```

Quelltext 5.9: Freemarker-Template *ClassMain.ftl* zur Erzeugung von Java-Klassen

Da UML/P-Klassendiagramme keine dedizierten Sprachkomponenten für die Behandlung oder Kennzeichnung von inneren Klassen haben, verwendet das *xsd2dsl*-Framework das in Java übliche Namensschema [Fla96]. Innere Klassen tragen den Namen der äußeren Klasse, getrennt durch ein `$` Zeichen, als Präfix. Deshalb erfolgt bei der Codegenerierung zunächst keine unterschiedliche Behandlung von inneren Klassen, da sie ebenfalls in der durch die *getCDClasses*-Methode abgerufenen Klassenliste enthalten sind. Über den Aufruf der in Quelltext 5.10 gezeigten Methode *isInner* der Hilfsklasse wird ermittelt, ob es sich bei der zur Transformation an das Template übergebenen Klasse um eine innere Klasse handelt.

Die Methode *getOuterClass* versucht für das *ASTCDClass*-Objekt, mit dem die *ClassDetails* Klasse instanziiert wurde, den Namen der äußeren Klasse zu bestimmen. Falls der Klassenname ein `$` Zeichen enthält wird über eine Hilfsmethode aus dem *common*-Projekt der Teil hinter dem letzten `$` Zeichen, und damit der eigentliche Klassenname, abgetrennt. So wird der Name der äußeren Klasse in Zeile 8 der Variable *sReturn* zugewiesen und in Zeile 10 zurückgegeben. Enthält der Klassenname kein `$` Zeichen wird entsprechend der Initialisierung der *sReturn*-Variable *null* zurückgegeben.

So genügt es in der Methode *isInner* in Zeile 2 den Rückgabewert der *getOuterClass*-Methode zu prüfen, um zu bestimmen ob es sich um eine innere Klasse handelt. Ist das

```
1  public boolean isInner() {
2    return getOuterClass() != null;
3  }
4
5  public String getOuterClass() {
6    String sReturn = null;
7    if (ast.getName().contains("$")) {
8      sReturn = HelperMethods.getOuterClassName(ast.getName());
9    }
10   return sReturn;
11 }
```

Quelltext 5.10: Hilfsmethoden zu Behandlung von inneren Klassen

der Fall, wird durch das if-Statement in Zeile 4 von Quelltext 5.9 ausgeschlossen, dass für diese Klasse eine eigene Java-Datei erzeugt wird. Die Codeerzeugung über diesen Pfad wird somit beendet, da keine weiteren Templates mehr aufgerufen werden. Im weiteren Verlauf wird gezeigt, wie die inneren Klassen an der benötigten Stelle gefunden und Code aus ihnen erzeugt wird.

Handelt es sich bei der Klasse nicht um eine innere Klasse, wird in Zeile 5 über die Methode *getPackage* der Hilfsklasse die Packagebezeichnung der Klasse ermittelt und in Zeile 6 mit Hilfe der *callTemplate*-Methode das Template *Class.ftl* aufgerufen und eine entsprechende Java-Datei erzeugt.

Quelltext 5.11 zeigt das Template *Class.ftl* zur Generierung von Java-Klassendateien. In Zeile 1-5 werden die Hilfsklasse *ClassDetails* instanziiert und mit ihrer und der Hilfe der *TypeHelper*-Klasse verschiedene für die Gestaltung der Java-Klasse relevante Eigenschaften aus dem AST berechnet. Handelt es sich nicht um eine innere Klasse werden durch Zeile 7 die Package-Deklaration und in Zeile 9 und 10 für alle Klassen gemeinsame Import-Statements erzeugt. Von Zeile 11 und 12 werden die klassenspezifischen Import-Statements erzeugt. Mit Hilfe der *Model2TargetConverter*-Klasse werden die Klassendiagramm-Import-Statements in Java-Import-Statements transformiert.

In Zeile 13 wird die Java-Klassendeklaration erzeugt. Aufgrund der JiBX-Kompatibilität tragen alle Klassen den Modifier *public* und innere Klassen den Modifier *static*. Auf die Modifier folgt das *class*-Schlüsselwort und der Klassenname. Falls die Klasse eine Oberklasse hat, wird ein entsprechendes *extends*-Statement mit dem durch die *TypeHelper*-Klasse ermittelten Klassennamen erzeugt.

Ab Zeile 15 wird durch das Einbinden weiterer Subtemplates der Klassenrumpf erzeugt. Die in Zeile 15 und 16 eingebundenen Templates zur Erzeugung von Attributen beziehungsweise Assoziationen werden in den beiden folgenden Abschnitten näher betrachtet. In Zeile 17 werden über die Methode *getInnerClasses* der Klasse *ClassDetails* alle inneren Klassen der aktuellen Klasse anhand ihres Namens bestimmt. Es werden alle Klassen des Klassendiagramms aufgerufen, deren Name als Präfix den Namen der aktuellen Klasse und ein `$` Zeichen trägt und anschließend keine weiteren `$` Zeichen mehr beinhaltet. Für jede dieser Klassen wird in Zeile 18 rekursiv das Template *Class.ftl* aufgerufen, das, wie in diesem Abschnitt beschrieben, Deklaration und Rumpf der inneren Klassen erzeugt. Nach

```
1  <#assign helper = op.instantiate("xsd2dsl.cd2java.helper.
       ↪ ClassDetails", [ast])>
2  <#assign superClass = typeHelper.getSuperClassName(ast)>
3  <#assign isInner = helper.isInner()>
4  <#assign className = helper.getName()>
5  ${op.setValue("inherited", typeHelper.hasSuperClass(ast))}
6  <#if !isInner>
7  package ${package};
8
9  import java.util.List;
10 import java.util.ArrayList;
11 <#list ast.printImportList() as i>import ${m2tConv.convertImport(i)}
       ↪ ;</#list>
12 </#if>
13 public <#if isInner>static</#if> class ${className}<#if inherited>
       ↪ extends ${superClass}</#if>
14 {
15   ${op.includeTemplates("xsd2dsl.cd2java.core.Attribute", ast.
         ↪ getCDAttributes())}
16   ${op.includeTemplates("xsd2dsl.cd2java.core.Association", ast.
         ↪ getCDAssociations())}
17   <#list helper.getInnerClasses() as inner>
18       ${op.includeTemplates("xsd2dsl.cd2java.core.Class", inner)}
19   </#list>
20   <#list helper.getInnerEnums() as inner>
21       ${op.includeTemplates("xsd2dsl.cd2java.core.Enum", inner)}
22   </#list>
23 }
```

Quelltext 5.11: FreeMarker-Template zur Erzeugung von Java-Klassen

dem selben Schema werden in Zeile 20 innere Enumerationsklassen bestimmt und durch Einbindung des Templates *Enum.ftl* in Zeile 21 generiert.

Generierung von Attributen

Quelltext 5.12 zeigt das Template *Attribute.ftl*, das den Java-Code für *ASTCDAttribute*-Objekte erzeugt. Über die Hilfsklasse *GetterSetterHelper* werden in Zeile 1 und 2 zunächst die Namen der Zugriffsmethoden bestimmt und in FreeMarker-Variablen gespeichert. In Zeile 4 wird eine private Java-Klassenvariable mit dem Typ und Namen des ASTCDAttribute-Objekts erzeugt. In Zeile 6 bis 8 wird eine Abfragemethode mit dem zuvor bestimmten Namen erzeugt, die den Wert der in Zeile 4 angelegten Variable zurück gibt. In Zeile 10 bis 12 wird analog eine Änderungsmethode erzeugt, die den Wert der Variablen auf den des übergebenen Parameters setzt.

Generierung von Assoziationen

Die Erzeugung von Java-Code aus *ASTCDAssociation*-Objekten ist durch das Template *Association.ftl* gekapselt, welches von Quelltext 5.13 gezeigt wird. Zunächst wird die

```
<#assign getterName = getterSetterHelper.getPlainGetter(ast)>
<#assign setterName = getterSetterHelper.getPlainSetter(ast)>

private ${ast.printType()} ${ast.printName()};

public ${ast.printType()} ${getterName}() {
  return ${ast.printName()};
}

public void ${setterName}(${ast.printType()} ${ast.printName()}) {
  this.${ast.printName()} = ${ast.printName()};
}
```

Quelltext 5.12: FreeMarker-Template zur Erzeugung von Java-Code aus Attributen

Hilfsklasse *AssociationDetails* instanziiert um mit Hilfe ihrer Methoden verschiedene Informationen über die Assoziation zu berechnen und in FreeMarker-Variablen zu speichern. Dazu zählen die Kardinalität der Komponente, ihr Typ und der gewünschte Attribut- sowie Rollenname.

Da in den von der *xsd2cd*-Komponente erzeugten Klassendiagrammen ausschließlich Kompositionen verwendet werden muss bei der Erzeugung des Java-Codes nur zwischen Kompositionen mit einfacher und mehrfacher Kardinalität der Komponente unterschieden werden.

In Zeile 2 wird in der FreeMarker-Variable *targetCardinalityMultiple* ein entsprechen-

```
1 ${op.setValue("assocDetails", op.instantiate("xsd2dsl.cd2java.helper
    ↪ .AssociationDetails", [ast, op]))}
2 ${op.setValue("targetCardinalityMultiple", assocDetails.
    ↪ getTargetCardinalityMultiple())}
3 ${op.setValue("targetType", assocDetails.getTargetType())}
4 ${op.setValue("targetAttrName", assocDetails.getTargetAttrName())}
5 ${op.setValue("targetRole", assocDetails.getTargetRole())}
6 <#if targetCardinalityMultiple>
7   ${op.includeTemplates("xsd2dsl.cd2java.core.Composition", ast)}
8 <#else>
9   ${op.includeTemplates("xsd2dsl.cd2java.core.
      ↪ CompositionCardinalityMultiple", ast)}
10 </#if>
```

Quelltext 5.13: FreeMarker-Template zur Erzeugung von Java-Code aus Assoziationen

der boolescher Wert gespeichert, der mit Hilfe der Methode *getTargetCardinalityMultiple* der Hilfsklasse *AssociationDetails* aus dem AST-Objekt ermittelt wird. Ab Zeile 9 wird danach, abhängig vom Wert dieser Variable, das Template *Composition.ftl* oder *CompositionCardinalityMultiple.ftl* eingebunden.

Quelltext 5.14 zeigt das Template *Composition.ftl* zur Behandlung von Kompositionen mit einfacher Kardinalität.

Es erzeugt, ähnlich dem Template *Attribute.ftl* aus dem letzten Abschnitt, eine Klassenvariable und entsprechende Zugriffsmethoden. Der einzige Unterschied ist, dass die benötigten Identifier und Typen nicht direkt aus dem AST-Objekt ermittelt werden können, sondern wie zuvor beschrieben durch die Hilfsklasse berechnet werden.

```
1 private ${targetType} ${targetAttrName};
2
3 public ${targetType} get${targetRole}() {
4   return ${targetAttrName};
5 }
6
7 public void set${targetRole}(${targetType} ${targetAttrName}) {
8   this.${targetAttrName} = ${targetAttrName};
9 }
```

Quelltext 5.14: FreeMarker-Template zur Erzeugung von Java-Code aus Kompositionen

Quelltext 5.15 zeigt das Template *CompositionCardinalityMultiple.ftl*. Es ist zuständig für die Erzeugung von Java-Code aus Kompositionsbeziehungen mit mehrfacher Kardinalität der Komponente. In Zeile 1 wird eine Klassenvariable erzeugt, deren Typ eine generische Liste vom Typen der Komponente ist. Ihr wird eine neue Instanz der Klasse *ArrayList* mit entsprechendem Typparameter zugewiesen. Ab Zeile 3 und 7 werden Entsprechende Abfrage- und Änderungsmethoden erzeugt.

```
1 private List<${targetType}> ${targetAttrName}List = new ArrayList<${
    ↪ targetType}>();
2
3 public List<${targetType}> get${targetRole}List () {
4   return ${targetAttrName}List;
5 }
6
7 public void set${targetRole}List (List<${targetType}> list) {
8   ${targetAttrName}List = list;
9 }
```

Quelltext 5.15: FreeMarker-Template zur Erzeugung von Java-Code aus Kompositionen mit mehrfachen Kardinalitäten

Generierung weiterer Entitäten

Die Erzeugung von Enum-Klassen aus *ASTCDEnum*-Objekten des CD-ASTs läuft nach dem selben Schema wie die Generierung regulärer Klassen ab. Daher wird an dieser Stelle auf die explizite Darstellung der verwendeten Templates verzichtet.

Die in den vorherigen Abschnitten gezeigten Templates sind, ebenfalls aus Platzgründen, teilweise vereinfacht.

5.3.2 Verwendung des Generators

Um neben den Java-Klassen und den Templates auch die Verwendung des Generators in das *xsd2dsl*-Projekt einzubinden wurde dieses um das *cd2java-use*-Projekt erweitert. Das *cd2java-use*-Projekt nutzt das MontiCore-Maven-Plugin um das *Cd2JavaGenerator*-Tool zu parametrisieren und zu starten. Dabei wird das Generator-Tool so konfiguriert, dass

für alle Klassendiagramme im Verzeichnis *models* Java-Code generiert wird. Die erzeugten Java-Dateien werden entsprechend der Packetdeklaration der Klassendiagramme in Unterverzeichnissen des Ordners *gen* abgelegt. Als weitere Build-Schritte des *cd2java-use*-Projekts wird der generierte Code kompiliert und in einem Java Archive zusammengefasst. Die so erzeugte Softwarebibliothek kann in andere Projekte eingebunden werden.

5.4 Umwandlung von XML-Dokumenten in Objektdiagramme

Für die Transformation von XML-Dateien in Objektdiagramme wurde innerhalb des *xml2od*-Projektes das *Xml2OdTool* entwickelt. Wie in Abschnitt 4.5 erwähnt ist bereits eine MontiCore Grammatikbeschreibung für XML-Dokumente vorhanden, die an dieser Stelle eingesetzt werden kann. Durch das *xml-lang*-Projekt ist im Namensraum *mc.lang.xml* die Klasse *XMLParsingWorkflow* vorhanden, welche als erster Ausführungsschritt des *Xml2OdTools* verwendet wird.

5.4.1 Transformationsworkflow

Nach dem Parsen der XML-Datei wird wie im *Xsd2CdTool* die Transformation als zweiter Ausführungsschritt gestartet. Sie ist im *xml2od*-Projekt ebenfalls in einer Klasse namens *TransformationWorkflow* gekapselt. Zusätzlich zu den Standard-Parametern kann dem *Xml2OdTool* über den Parameter *cdpath* der Pfad zu dem Klassendiagramm übergeben werden, welcher an den *TransformationWorkflow* bei seiner Instanziierung weitergegeben wird. Wird der Parameter nicht angegeben, verwendet der *TransformationWorkflow* für jede XML-Datei das Klassendiagramm mit dem entsprechenden Dateinamen.

Ist der Pfad zum Klassendiagramm ermittelt instanziiert der *TransformationWorkflow* die Klasse *TransformationVisitor*, die das bereits bekannte Visitor-Entwurfsmuster implementiert. Der Konstruktor erzeugt zunächst den Wurzelknoten vom Typ *ASTODDefinition* des OD-ASTs und gibt dem Objektdiagramm, über dessen Methode *setName*, den Namen der XML-Datei. Anschließend wird das zugehörige Klassendiagramm geparst und in einer Klassenvariable als *ASTCDDefinition*-Objekt gespeichert.

Erzeugung von Objektdiagrammkomponenten

Für den Aufbau des OD-ASTs verfügt die *TransformationVisitor*-Klasse über verschiedene Hilfsmethoden, die von den *visit*-Methoden aufgerufen werden. Aufgaben und Ablauf der Hilfsmethoden werden in den folgenden Absätzen beschrieben.

createObject erzeugt ein *ASTODObject*-Objekt und fügt es dem OD-AST hinzu. Dazu erhält die Methode als Parameter ein *ASTCDClass*-Objekt und ein *ASTXMLElement*. Das XML-Element enthält die Daten, mit denen das Objekt gefüllt werden soll und das *ASTCDClass*-Objekt aus dem Klassendiagramm die benötigten Informationen über den Aufbau, den das erzeugte Objekt haben soll. Zunächst wird mit Hilfe der *ODNodeFactory* ein *ASTODObject*-Knoten instanziiert und anhand der Informationen aus dem *ASTCDClass*-Objekt sein Typ festgelegt. Außerdem werden alle direkten und geerbten Attribute

der Klasse ermittelt und dem Objekt hinzugefügt. Durch die Methode *getObjectId* wird ein eindeutiger Identifier bestimmt und anschließend dem Objekt zugewiesen. Ist der Aufbau des Objekts abgeschlossen werden alle XML-Attribute des übergebenen XML-Elements traversiert und ihr Wert dem entsprechenden Attribut des *ASTODObject*-Objekts zugewiesen. Abschließend wird das Objekt dem Objektdiagramm hinzugefügt und genau wie die Klassen im *xsd2cd*-Projekt innerhalb verschiedener Maps der *TransformationVisitor*-Klasse katalogisiert. Um in der aufrufenden Methode eine direkte Weiterbehandlung des erzeugten Objekts zu ermöglichen wird das *ASTODObject*-Objekt zurückgegeben.

Die in Quelltext 5.16 gezeigte Methode *setLinkOrAttribute* erzeugt anhand eines *ASTXMLElement*-Knoten einen Attributwert oder eine Komponente für ein *ASTODObject*-Objekt. Dazu wird in Zeile 2 zunächst der Elementname in einen entsprechenden UMLP-Identifier transformiert und in Zeile 3 das dem *ASTODObject*-Objekt zugehörige *ASTCDClass*-Objekt ermittelt.

Ab Zeile 5 wird die Klasse analysiert. Verfügt sie über ein Attribut dessen Name dem aus dem Elementnamen berechneten Identifier entspricht, wird der Wert des Attributs im *ASTODObject*-Knoten in Zeile 7, durch den Aufruf der Methode *setAttributeValue*, auf den Textinhalt des XML-Elements gesetzt. Ist kein gültiger Textinhalt vorhanden wird von der *setAttributeValue*-Methode mit Hilfe der Methode *removeAttribute* das Attribut, falls vorhanden, aus dem Objekt entfernt, da keine Wertzuweisung möglich ist. Durch das Entfernen der Attribute ohne Wertzuweisung wird die Lesbarkeit des Objektdiagramms erhöht.

Wurde kein passendes Attribut gefunden, wird in Zeile 12 durch die Methode *getAssociation* im Klassendiagramm nach einer Kompositionsbeziehung mit entsprechendem Typen und Rollennamen gesucht. In Zeile 14 wird der Klassenname und in Zeile 16 das *ASTCDClass*-Objekt der Komponente ermittelt. Aus ihm und dem *ASTXMLElement*-Knoten wird in Zeile 18-22 mit Hilfe der zuvor beschriebenen *createObject*-Methode ein neuer *ASTODObject*-Knoten erzeugt, dem Objektdiagramm hinzugefügt und falls erforderlich mit Daten aus dem XML-Element befüllt. Anschließend wird in Zeile 23 mit Hilfe der *createLink*-Methode ein *ASTODLink*-Knoten erzeugt, der die Kompositionsbeziehung zwischen den beiden *ASTODObject*-Knoten ausdrückt und dem OD-AST hinzugefügt.

Transformation des Wurzelelements

Die Transformation des XML-ASTs beginnt mit dem Aufruf des Visitors für das *ASTXMLDocument*-Objekt, das die Wurzel des ASTs ist. Durch die *getRootClass*-Methode wird anhand des *«Element»*-Stereotyps das *ASTCDClass*-Objekt der zugehörigen Klasse des Klassendiagramms ermittelt, welche die oberste Ebene im Datenmodell darstellt. Mit Hilfe der *createObject*-Methode wird das erste Objekt erzeugt und dem Objektdiagramm hinzugefügt. Um diese Information auch im Objektdiagramm zu erhalten wird ihm der Stereotyp «Root» hinzugefügt.

Entsprechend dem Visitor-Entwurfsmuster wird das XML-Dokument elementweise durchlaufen und durch die im folgenden Abschnitt beschriebene *visit*-Methode der OD-AST weiter aufgebaut.

```
1  private void setLinkOrAttribute(ASTODObject targetObject,
       ↪ ASTXMLElement element) {
2    String elementName = getUMLPIdentifier(element.getName(), false);
3    ASTCDClass targetClass = objectToClass.get(targetObject);
4    if (targetClass != null) {
5      ASTODAttribute targetAttribute = getAttribute(targetObject.
         ↪ getODAttributes(),elementName);
6      if (targetAttribute != null) {
7        setAttributeValue(targetAttribute, getValueFromElement(element
           ↪ ));
8      }
9      else {
10       String linkedClassName = null;
11       ASTCDClass linkedClass = null;
12       ASTCDAssociation targetAssociation = getAssociation(
           ↪ classDiagram.getCDAssociations(),targetClass.getName(),
           ↪ elementName);
13       if(targetAssociation != null){
14           linkedClassName = targetAssociation.
               ↪ printRightReferenceName();
15       }
16       linkedClass = getClass(linkedClassName);
17       if (linkedClass != null) {
18         ASTODObject linkedObject = createObject(linkedClass, element
             ↪ );
19         targetAttribute = getAttribute(linkedObject.getODAttributes
             ↪ (),elementName);
20         if (targetAttribute != null) {
21           setAttributeValue(targetAttribute, getValueFromElement(
               ↪ element));
22         }
23         objectDiagram.getODLinks().add(createLink(targetObject,
             ↪ linkedObject, elementName));
24       }
25     }
26   }
27 }
```

Quelltext 5.16: Hilfsmethode für die Transformation von XML-Elementen in Objektdiagrammfragmente

Transformation der übrigen XML-Elemente

Der vom Parser erzeugte XML-AST besteht im wesentlichen aus leeren und nicht-leeren XML-Elementen.

Quelltext 5.17 zeigt die Implementierung der *visit*-Methode für XML-Elemente, die sowohl für leere als auch für nicht-leere XML-Elemente aufgerufen wird. In Zeile 2 und 3 wird zunächst das Elternelement und der daraus erzeugte *ASTODObject*-Knoten mit Hilfe der *elementToObject*-Map ermittelt.

Abhängig von der Struktur des dem Klassendiagramm zugrunde liegenden XML-Schemas ist es möglich, dass aus dem direkten Elternelement gar kein *ASTODObject*-Knoten erzeugt worden ist. Daher wird in den Zeilen 4 bis 12 durch eine Schleife der XML-AST

```
1  public void visit(ASTXMLElement element){
2    ASTXMLElement parentElement = element.getParentElement();
3    ASTODObject parentObject = elementToObject.get(parentElement);
4    while (parentObject == null) {
5      parentObject = elementToObject.get(parentElement);
6      if (parentElement != null) {
7        parentElement = parentElement.getParentElement();
8      }
9      else {
10       break;
11     }
12   }
13   if (parentObject != null) {
14     setLinkOrAttribute(parentObject, element);
15   }
16 }
```

Quelltext 5.17: Visitor für leere und nicht-leere XML-Elemente

aufwärts durchsucht, bis ein Element mit Objektrepräsentation gefunden wurde oder das Wurzelelement erreicht wurde. Wurde ein Objekt gefunden, werden durch die *setLinkOrAttribute*-Methode in Zeile 14 abhängig von Element- und Objektstruktur ein Attributwert gesetzt oder aus dem XML-Element ein *ASTODObject*-Knoten erzeugt und das Objektdiagramm um eine entsprechende Kompositionsbeziehung erweitert.

Über die Methode *getResult* der *TransformationVisitor*-Klasse kann am Ende der Transformation der OD-AST als ein *ASTMCCompilationUnit*-Objekt abgerufen werden. Die Dateiausgabe erfolgt mit Hilfe eines Pretty-Printers als letzter Schritt des *Transformation-Workflows*.

Anhang B.2 zeigt das durch das *Xml2OdTool* anhand des Klassendiagramms B.1 aus dem XML-Dokument in Quelltext 2.1 erzeugte Objektdiagramm.

Tests

Die Transformation von XML-Dokumenten in Objektdiagramme wird durch die Klasse *TransformationVisitorTests* des *xml2od*-Projekts getestet. Die Testfälle bestehen jeweils aus einem XML-Dokument, einem Klassendiagramm und einem Objektdiagramm. Die Testmethode *testTransformation* parst zunächst alle im Testordner vorhandenen XML-Dateien. Für jeden so erzeugten XML-AST wird der folgende Testablauf durchgeführt:

1. Die *TransformationVisitor*-Klasse wird instanziiert und der XML-AST mit ihrer Hilfe anhand des Klassendiagramms in einen OD-AST überführt.

2. Die zugehörige OD-Datei wird mit Hilfe des in [Sch12] entwickelten *ODTools* eingelesen und dadurch ebenfalls in einen OD-AST überführt.

3. Über die *toString*-Methode der *ASTODDefinition*-Objekte, welche die Wurzel der OD-ASTs darstellen, werden das durch Transformation erzeugte Objektdiagramm und das in der OD-Datei definierte, erwartete Objektdiagramm in Strings umgewandelt.

4. Die beiden Strings werden über die JUnit Methode *assertEquals* miteinander verglichen. Sind die Strings nicht identisch, schlägt der Test fehl.

Genau wie bei den Tests des *xsd2cd*-Projekts wird der Vergleich der textuellen Darstellung der ASTs bevorzugt, da eventuelle Unterschiede der Objektdiagramme als Ausgabe der Tests direkt in menschenlesbarer Form vorliegen.

5.5 Datenbindung zwischen Java-Klassen und Objektdiagrammen

Die Befüllung der vom *cd2java*-Projekt erzeugten Datenbindungsklassen mit Daten aus Objektdiagrammen sowie das Erzeugen von Objektdiagrammen aus Instanzen der Datenbindungsklassen ist innerhalb des *od2java*-Projektes implementiert. Wie in Abschnitt 4.6 erkannt, ist es wichtig, dass die Aufrufe der *cd2java*-Komponente identisch mit denen des JiBX-Frameworks sind.

Für die Implementierung im Rahmen der Arbeit wurde, trotz der aufgezeigten Nachteile, aus Zeitgründen, Reflexion verwendet. Als Machbarkeitsnachweis ist dies ausreichend und der modulare Aufbau des Frameworks ermöglicht eine einfache Ablösung durch eine Komponente die einen anderen Ansatz verwendet.

Quelltext 5.18 zeigt Java-Code zum Einlesen und Ausgeben von XML-Dokumenten durch JiBX. Im *od2java*-Projekt sind die Interfaces und Klassen *IBindingFactory*, *BindingFactory*, *BindingDirectory*, *IUnmarshallingContext*, *UnmarshallingContext*, *IMarshallingContext* und *MarshallingContext* ebenfalls implementiert um den Zugriff auf die zur Datenbindung verwendeten Klassen zu kapseln.

```
1  IBindingFactory bfact = BindingDirectory.getFactory(Example.class);
2
3  IUnmarshallingContext uctx = bfact.createUnmarshallingContext();
4  FileInputStream in = new FileInputStream("Example.xml");
5  Example exampleObject = (Example)uctx.unmarshalDocument(in, null);
6
7  IMarshallingContext mctx = bfact.createMarshallingContext();
8  FileOutputStream out = new FileOutputStream("Example.xml");
9  mctx.setOutput(out, null);
10 mctx.marshalDocument(exampleObject);
```

Quelltext 5.18: Ein- und Auslesen eines XML-Dokumentes durch das JiBX-Framework

Durch dieses Vorgehen müssen in vorhandenem Code der das JiBX-Framework verwendet nur die Import-Deklarationen ersetzt werden um stattdessen die Klassen aus dem *xsd2dsl*-Framework zu nutzen.

5.5.1 Funktionsweise der einzelnen Klassen

Durch den Nachbau der JiBX-Klassen wird jedoch stark von der Implementierung der Datenbindung abstrahiert. Um die Arbeitsweise der *od2java*-Komponente nachvollziehbar

zu machen, wird beschrieben welche Aktionen die Klassen bei den entsprechenden Aufrufen aus Quelltext 5.18 durchführen.

BindingDirectory

Die *BindingDirectory*-Klasse besitzt die öffentliche Methode *getFactory*, welche als Parameter ein *Class*-Objekt empfängt und als Rückgabewert ein Objekt, das das *IBindingFactory*-Interface implementiert liefert. Dazu instanziiert Sie die Klasse *BindingFactory* und übergibt deren Konstruktur das *Class*-Objekt. Das *Class*-Objekt repräsentiert bei der Verwendung des JiBX-Frameworks die Datenbindungsklasse für das Wurzelelement des XML-Dokumentes. Das *xsd2dsl*-Framework benötigt diese Information eigentlich nicht, da die zuständigen Klassen anhand des Objektdiagramms ermittelt werden können. Um die Kompatibilität zu den JiBX-Klassen zu erhalten wird das *Class*-Objekt dennoch von den verschiedenen Konstruktoren empfangen.

BindingFactory

Der Konstruktor der *BindingFactory*, speichert das *Class*-Objekt in einer Klassenvariable. Da die Klasse das *IBindingFactory*-Interface implementiert, muss sie die Methoden *createUnmarshallingContext* und *createMarshallingContext* implementieren, die ein Objekt vom Typ *IUnmarshallingContext* beziehungsweise *IMarshallingContext* zurückgeben. Dazu wird jeweils dem Konstruktor der Klasse *UnmarshallingContext* beziehungsweise *MarshallingContext* das *Class*-Objekt übergeben und so ein entsprechendes Objekt erzeugt. Die Konstruktoren legen das *Class*-Objekt ebenfalls in einer Klassenvariable ab.

UnmarshallingContext

Die Klasse *UnmarshallingContext* implementiert das *IUnmarshallingContext*-Interface und stellt dazu die Methode *unmarshalDocument* zur Verfügung, welche die Bindung der Daten aus dem Objektdiagramm an die durch die *cd2java*-Komponente erzeugten Klassen durchführt. Dazu erhält sie ein *InputStream*-Objekt, welches das Eingabedokument zur Verfügung stellt.

Für die Befüllung der Datenbindungsklassen mit den Daten aus dem Objektdiagramm wird ein OD-AST benötigt. Da die MontiCore Werkzeuge zur Verarbeitung von Objektdiagrammen für die Verwendung mit Dateien ausgelegt sind und der ursprüngliche Dateipfad anhand des *InputStream*-Objekts nicht ermittelt werden kann, wird der *InputStream* zunächst in eine temporäre Datei geschrieben. Diese wird mit Hilfe des *ODTools* geparst und so in einen OD-AST überführt.

Anschließend wird durch den Aufruf der Methode *bindData* mit dem OD-AST als Parameter die Datenbindung gestartet. Nach erfolgter Datenbindung wird die temporäre OD-Datei gelöscht und als Ergebnis das von der *bindData*-Methode befüllte Objekt zurückgegeben.

Im Zentrum der Datenbindung steht die Methode *createObject*, die aus einem *ASTODObject*-Knoten ein entsprechendes Java-Objekt erzeugt und mit den Daten aus dem Objektdiagramm befüllt. Durch Rekursion bei eventuellen Kompositionsbeziehungen wird so, vom

Wurzelobjekt ausgehend, die gesamte Datenhierarchie vom Objektdiagramm auf Instanzen der Datenbindungsklassen übertragen. Dazu wird in der *bindData*-Methode anhand des *«Root»*-Modifiers das Objekt der obersten Ebene des Datenmodells ermittelt und an die *createObject*-Methode übergeben.

Da der Code der Methode und der von ihr aufgerufenen Hilfsmethoden sehr umfangreich ist, wird an dieser Stelle die Funktionsweise als Auflistung von Arbeitsschritten beschrieben.

1. Mit Hilfe der *Class.forName*-Methode und der Typbezeichnung des *ASTODObject*-Objekts wird ein *Class*-Objekt der für das Objekt benötigten Datenbindungsklasse erzeugt. Per Reflexion wird aus diesem eine Instanz der Datenbindungsklasse erzeugt.

2. Zur Befüllung der Attribute des Objekts wird der Methode *setAttributes* die Instanz der Datenbindungsklasse und die *ASTODAttribute*-Kindknoten des *ASTODObject*-Knotens übergeben. Sie führt für jedes der übergebenen *ASTODAttribute*-Objekte die folgenden Schritte durch:

   (a) Per Reflexion werden die Datenbindungsklasse und eventuelle Oberklassen nach dem Feld das dem Namen des Attributs entspricht durchsucht.

   (b) Der Wert des *ASTODAttribute*-Objekts wird aus dem AST ermittelt und zusammen mit der Instanz der Datenbindungsklasse und dem zuvor bestimmten Feld an die Methode *setField* übergeben.

   (c) Die *setField*-Methode wandelt den als String vorliegenden Wert falls erforderlich in den Java-Datentyp des Feldes um und setzt das Feld des übergebenen Datenbindungsobjekts auf diesen Wert.

3. Zur Erzeugung von Komponenten wird zunächst mit Hilfe der *getLinks* Methode eine Liste mit allen Kompositionsbeziehungen, in denen das *ASTODObject*-Objekt das Kompositum ist, ermittelt. Die Liste enthält Objekte vom Typ *ASTODLink*. Zusammen mit dem *ASTODObject*-Objekt (im Folgenden Kompositum) wird sie an die Methode *setLinks* übergeben. Diese führt für jede der gefundenen Kompositionsbeziehungen die folgenden Schritte durch:

   (a) Anhand der im *ASTODLink*-Objekt verfügbaren Informationen wird aus dem Objektdiagramm das *ASTODObject*-Objekt der Komponente (im Folgenden Komponente) geladen.

   (b) Durch rekursiven Aufruf der *createObject*-Methode wird aus der Komponente ebenfalls ein Java-Objekt vom Typ der entsprechenden Datenbindungsklasse erzeugt. Anschließend wird mit den folgenden Teilschritten die Beziehung zwischen den beiden Java-Objekten hergestellt:

      i. Da aus dem *ASTODLink*-Objekt (und dem Klassendiagramm im Allgemeinen) nicht ermittelt werden kann welche Kardinalität die Kompositionsbeziehung hat, wird zunächst von einer 1-zu-1 Beziehung ausgegangen und geprüft ob die Datenbindungsklasse des Kompositums über ein entsprechendes Feld verfügt. Ist das der Fall wird es mit der Komponente gefüllt.

      ii. Ist die Kardinalität der Kompositionsbeziehung mehrfach, wird das Feld im vorherigen Schritt nicht gefunden. Stattdessen wird das entsprechende Listenattribut in der Datenbindungsklasse des Kompositums ermittelt.

Anschließend werden der Methode *addListItem* Kompositum, Komponente und das Listenfeld übergeben. Sie überprüft ob das Listenattribut bereits eine Listeninstanz enthält und erzeugt eine neue Instanz, sollte das nicht der Fall sein. Anschließend wird per Reflexion die *add*-Methode der Liste aufgerufen um ihr die Komponente hinzuzufügen.

4. Die befüllte Instanz der Datenbindungsklasse wird an die aufrufende Methode zurückgegeben.

Von der *unmarshalDocument*-Methode wird also das, auf die zuvor beschriebene Weise an die Daten des Objektdiagramms gebundene, Java-Objekt zurückgegeben. Durch Typumwandlung lässt es sich in eine Instanz der Datenbindungsklasse konvertieren und kann über die Zugriffsmethoden beliebig ausgelesen und verändert werden.

MarshallingContext

Die Klasse *MarshallingContext* implementiert das *IMarshallingContext*-Interface und muss daher die Methoden *setIndent*, *setOutput* und *marshalDocument* zur Verfügung stellen. Die *marshalDocument*-Methode erzeugt aus Instanzen der Datenbindungsklassen ein Objektdiagramm. Über die *setOutput*-Methode wird der Klasse ein *FileOutputStream*-Objekt übergeben, über welches das Objektdiagramm in eine Datei geschrieben werden soll. Es wird zur späteren Verwendung in einer Klassenvariable gespeichert. Ein Aufruf der Methode *setIndent* hat keinen Effekt auf die Erzeugung des Objektdiagramms, sondern ist nur aus Gründen der JiBX-Kompatibilität möglich.

Die Methode *marshalDocument* erhält als Parameter eine Instanz der Datenbindungsklasse, die der obersten Ebene im Datenmodell entspricht. Zunächst wird von ihr mit Hilfe der *ODNodeFactory* der Wurzelknoten eines OD-ASTs erzeugt. Anschließend wird das Datenbindungsobjekt an die Methode *buildAST* übergeben, welche aus den enthaltenen Daten den AST des Objektdiagramms aufbaut und mit Attributwerten füllt. Zuletzt wird der AST mit Hilfe eines Pretty-Printers in einen String umgewandelt, der über das *FileOutputStream*-Objekt in eine Datei geschrieben wird.

Die Methode *buildAST* startet den Aufbau des OD-ASTs durch den Aufruf der *addObjectToAST* Methode mit dem Datenbindungsobjekt der obersten Ebene des Datenmodells. Sie erzeugt aus einem Java-Objekt einen *ASTODObject*-Knoten und fügt ihn dem Klassendiagramm hinzu. Ähnlich wie die *createObject*-Methode der *UnmarshallingContext*-Klasse werden durch Rekursion Beziehungen zwischen Objekten aufgelöst und in Kompositionsbeziehungen umgewandelt. Der genaue Ablauf wird durch die folgende Auflistung von Arbeitsschritten beschrieben.

1. Per Reflexion wird der Klassenname des übergebenen Java-Objekts bestimmt und durch die *createObject*-Methode gekapselt ein entsprechender *ASTODObject*-Knoten erzeugt und dem OD-AST hinzugefügt.

2. Per Reflexion werden alle Felder der Klasse und eventueller Oberklassen durchlaufen und jeweils die folgenden Schritte durchgeführt:

   (a) Typ, Name und Wert des Feldes werden per Reflexion ausgelesen. Ist ein Wert vorhanden, wird abhängig vom Typ wie folgt fortgefahren:

i. *Primitive Datentypen (String,Integer,..)*: Gekapselt durch die *createAttribute*-Methode wird ein entsprechendes *ASTODAttribute*-Objekt erzeugt und sein Wert mit Hilfe der *setAttributeValue*-Methode auf den des Feldes gesetzt. Anschließend wird es dem *ASTODObject*-Knoten hinzugefügt.

ii. *Datenbindungsklassen*: Durch den rekursiven Aufruf der *addObjectToAST*-Methode mit dem Feldwert als Parameter wird ein neuer *ASTODObject*-Knoten erzeugt und dem Objektdiagramm hinzugefügt. Anschließend wird durch die *createLink*-Methode ein *ASTODLink*-Knoten erzeugt und dem Objektdiagramm hinzugefügt, der die Kompositionsbeziehung zwischen den beiden Objekten ausdrückt.

iii. *Listen mit einer Datenbindungsklasse als Elementtyp*: Per Reflexion werden die Listenelemente ausgelesen. Für jedes der Elemente werden wie in *ii.* beschrieben im Objektdiagramm Repräsentationen für die Komponente und die Kompositionsbeziehung angelegt.

Tests

Um die korrekte Funktionsweise der Klassen *UnmarshallingContext* und *MarshallingContext* zu überprüfen enthält das *od2java*-Projekt die Testklassen *UnmarshallingContextTests* und *MarshallingContextTests*. Als Testfälle dienen jeweils verschiedene Objektdiagramme. Erstere gibt die durch die *UnmarshallingContext*-Klasse erzeugten Objekte mit Hilfe der *Dumper*-Klasse zur manuellen Überprüfung aus. Die Klasse *MarshallingContextTests* führt für jeden der Testfälle folgende Schritte aus:

1. Durch die in Quelltext 5.18 gezeigten Aufrufe wird das Objektdiagramm eingelesen und anschließenden in eine andere Datei zurückgeschrieben.
2. Mit Hilfe des *ODTools* werden beide Dateien geparst.
3. Die String-Repräsentationen der durch den Parser erzeugten ASTs werden, wie aus den anderen Teilprojekten bereits bekannt, verglichen.
4. Entspricht das zurückgeschriebene Objektdiagramm dem Ursprünglichen ist der Test erfolgreich, da alle Daten korrekt eingelesen und wieder ausgegeben wurden.

5.6 Anwendungsbeispiel

In diesem Abschnitt wird die Verwendung des Migrationsframeworks anhand des bereits bekannten Beispielschemas erläutert. Als Ausgangspunkt dienen das XSD-Schema und verschiedene XML-Dokumente, die Rechnungen für einen Kunden enthalten.

Abbildung 5.9 illustriert den Einsatz der Frameworks im Rahmen des Anwendungsbeispiels und die dabei erzeugten Artefakte. Durch Einsatz des Werkzeugs *Xsd2CdTool* und des Codegenerators wird, wie in den vorherigen Abschnitten beschrieben, der Migrationsschritt zur Ablösung der XML-Schema Definition (*Rechnung.xsd*) durchgeführt. Automatisiert werden kann dies zum Beispiel mit Hilfe des MontiCore-Maven-Plugins, indem die beiden Werkzeuge nacheinander aufgerufen werden.

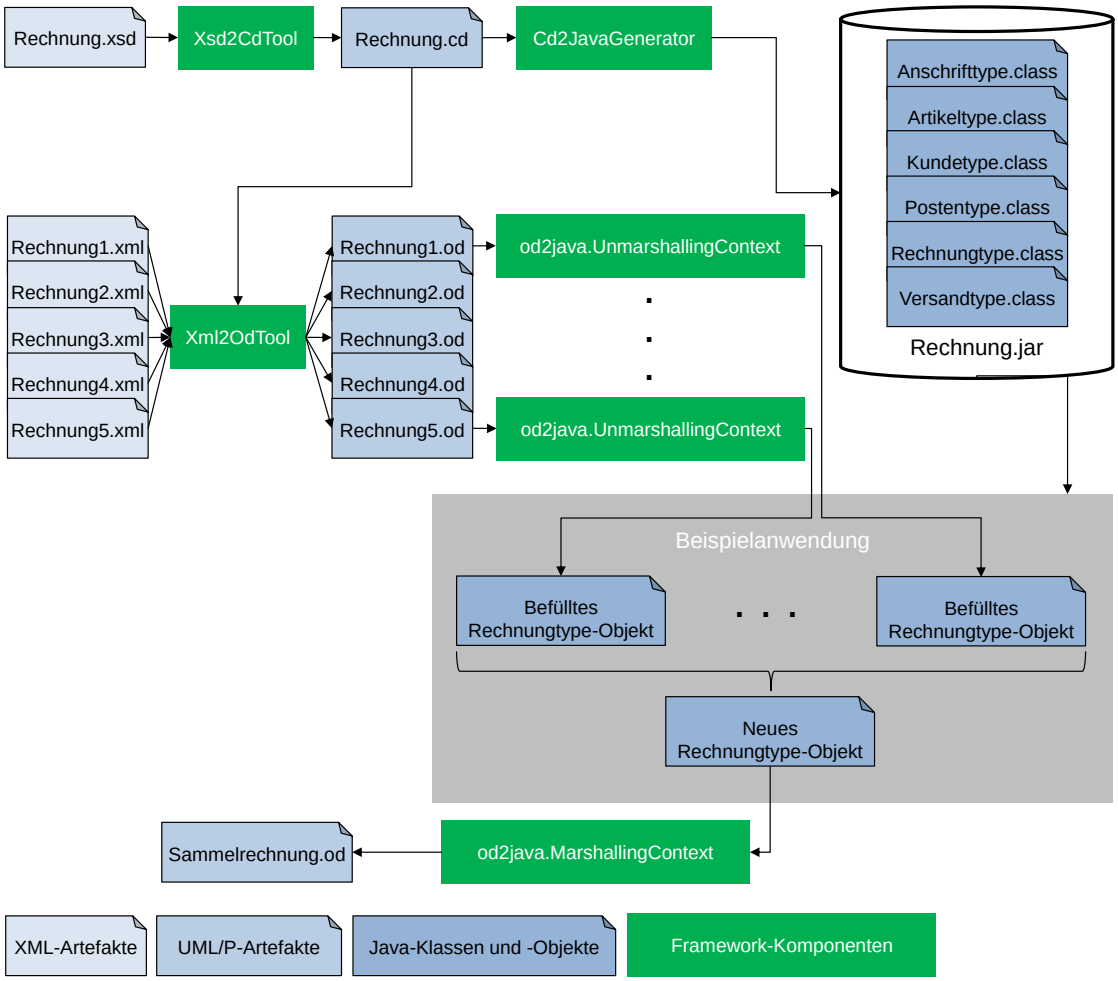


Abbildung 5.9: Artefaktdiagramm zur Darstellung der im Rahmen des Anwendungsbeispiels durchgeführten Transformationen

Dabei werden das Klassendiagramm *Rechnung.cd* erzeugt und aus ihm anschließend die Datenbindungsklassen generiert sowie in Form einer JAR-Datei verfügbar gemacht. Sie können zusammen mit der *od2java*-Komponente in beliebigen Java-Anwendungen eingesetzt werden. Um den Einsatz der *od2java*-Komponente zu erläutern, wurde eine Beispielanwendung implementiert. Die Aufgabe der Beispielanwendung ist, die einzelnen Rechnungsdokumente (*Rechnung[1,2,3,4,5].xml*) zu einer Sammelrechnung zusammenzufassen und den Gesamtbetrag der Rechnungsposten in der Standardausgabe anzuzeigen.

Quelltext 5.19 zeigt die Implementierung der Beispielanwendung. Zunächst wird in Zeile 1 bis 4 das *Xml2OdTool* aufgerufen, um die XML-Rechnungen im Ordner *input* in Objektdiagramme zu transformieren und diese im Ordner *output* zu speichern. In Zeile 6 bis 14 wird eine neue Instanz der Klasse *Rechnungtype* erzeugt, welche die Daten der Sammelrechnung aufnimmt. Zur Summierung des Gesamtbetrags aller Rechnungsposten wird in Zeile 16 die Variable *gesamtkosten* initialisiert. Ab Zeile 21 werden alle OD-Dateien im *output*- Verzeichnis durchlaufen.

In Zeile 22 wird mit Hilfe der *getRechnung*-Methode die OD-Datei eingelesen. Dabei kapselt

```
1  Xml2OdTool.main(new String[] { "src/main/resources/input/",
2      "-o", "src/main/resources/output/",
3      "-synthesis", "xml", "xml2od",
4      "-cdpath", "src/main/resources/model/Rechnung.cd" });
5
6  Rechnungtype sammelrechnung = new Rechnungtype();
7  sammelrechnung.setRechnungsnummer("2014-S1");
8  Date today = new Date();
9  sammelrechnung.setDatum(DateFormat.getDateInstance(DateFormat.
       ↪ DEFAULT, new Locale("de","DE")).format(today));
10 sammelrechnung.setPosten(new Postentype());
11 Versandtype versand = new Versandtype();
12 versand.setVersandart("Kumuliert");
13 versand.setKosten(BigDecimal.ZERO);
14 sammelrechnung.getPosten().setVersand(versand);
15
16 BigDecimal gesamtkosten = BigDecimal.ZERO;
17
18 File folder = new File("src/main/resources/output/");
19 File[] listOfFiles = folder.listFiles();
20
21 for (File file : listOfFiles) {
22   Rechnungtype rechnung = getRechnung("src/main/resources/output/" +
         ↪  file.getName());
23   if (sammelrechnung.getKunde() == null){
24     sammelrechnung.setKunde(rechnung.getKunde());
25   }
26   for (Artikeltype artikel : rechnung.getPosten().getArtikelList()){
27     sammelrechnung.getPosten().getArtikelList().add(artikel);
28     gesamtkosten = gesamtkosten.add(artikel.getGesamtpreis());
29   }
30   if (rechnung.getPosten().getVersand().getKosten().compareTo(
         ↪ BigDecimal.ZERO) != 0) {
31     sammelrechnung.getPosten().getVersand().setKosten(sammelrechnung
           ↪ .getPosten().getVersand().getKosten().add(rechnung.
           ↪ getPosten().getVersand().getKosten()));
32     gesamtkosten = gesamtkosten.add(rechnung.getPosten().getVersand
           ↪ ().getKosten());
33   }
34 }
35
36 writeRechnung(sammelrechnung,"src/main/resources/result/
       ↪ Sammelrechnung.od");
37 System.out.println("Gesamtbetrag aller Rechnungsposten: " +
       ↪ gesamtkosten.toString() + ".");
```

Quelltext 5.19: Beispielanwendung zur Datenverarbeitung mit Hilfe des *xsd2dsl*-Frameworks

die Methode lediglich die aus Quelltext 5.18 bekannten Aufrufe der *od2java*-Komponente. Falls die Sammelrechnung noch keine Kundendaten hat, werden diese in Zeile 24 aus der aktuellen Rechnungsdatei übernommen. Die Schleife in Zeile 26 bis 29 durchläuft alle Artikel, die als Posten in der aktuellen Rechnung aufgeführt sind, und kopiert sie zu den Posten der Sammelrechnung. In Zeile 28 werden die Kosten des Artikels zu den Gesamtkosten addiert. Sollten in der aktuellen Rechnung Versandkosten angefallen sein, werden

diese zu den Versandkosten der Sammelrechnung (Zeile 31) und den Gesamtkosten (Zeile 32) addiert.

Nachdem alle Rechnungsdokumente behandelt wurden, wird das *Rechnungtype*-Objekt, das die Sammelrechnung beinhaltet, mit Hilfe der *writeRechnung*-Methode, die ebenfalls nur die aus Quelltext 5.18 bekannten Aufrufe der *od2java*-Komponente kapselt, in die Datei *Sammelrechnung.od* im Verzeichnis *result* geschrieben. In Zeile 37 wird der Gesamtbetrag aller Rechnungsposten ausgegeben.

Sämtlicher Code in Quelltext 5.19 ist JiBX-kompatibel und könnte auch aus bestehender Software, die über XML-Datenbindung durch JiBX mit dem ursprünglichen Schema arbeitet, stammen. Auch wenn es sich nur um ein fiktives Beispiel handelt, macht dies deutlich, dass das *xsd2dsl*-Framework die effiziente Ablösung des *JiBX*-Frameworks durch die Wiederverwendung der darauf basierenden Anwendungen ermöglicht.

Kapitel 6

Evaluierung an einem Fallbeispiel

Die angedachte Evaluierung des *xsd2dsl*-Frameworks durch einen Industriepartner konnte aus terminlichen Gründen nicht vor dem Abschlussdatum der Arbeit erfolgen. Es wurde jedoch ein umfangreiches XML-Schema aus dem industriellen Kontext als Fallbeispiel für die Evaluierung zur Verfügung gestellt.

6.1 Anwendungsszenario

Das Anwendungsszenario sieht die vollständige Ablösung des Schemas und der vorhandenen Datensätze vor. Zur Datenbindung wird in einigen Anwendungen die mit den Daten des Schemas arbeiten JiBX verwendet. Die Umstellung dieser Anwendungen auf die neuen Datenbindungsklassen soll ebenfalls evaluiert werden. Anschließend soll das Datenmodell an einigen Stellen modifiziert werden, um die verbesserte Erweiterbarkeit zu belegen.

6.2 Einsatz des Frameworks

Im Rahmen der Evaluierung wurde das Framework wie folgt eingesetzt. Das Fallbeispiel besteht aus den Schemadateien *cds.xsd* und *cds_types.xsd*, wobei Letztere in die Erstgenannte eingebunden ist. Daher wurde zunächst das *Xsd2CdTool* verwendet um das Klassendiagramm *Cds.cd* aus den XSD-Dateien zu erzeugen. Es enthält insgesamt 61 Klassendefinitionen und 96 Kompositionsbeziehungen, welche die Baumstruktur des Datenmodells abbilden. Anhand der im Klassendiagramm definierten Hierarchie wurden aus den vorhandenen XML-Dokumenten mit Hilfe des *Xml2OdTool* zum Datenmodell konforme Objektdiagramme erzeugt.

Anschließend wurde der *Cd2JavaGenerator* verwendet, um aus der Datei *Cds.cd* die Datenbindungsklassen zu erstellen und eine entsprechende JAR-Datei zu erzeugen. Sie wurde in die bestehenden Anwendungen eingebunden, um die verwendeten JiBX-Klassen abzulösen. Danach wurden die neuen Versionen der Anwendungen auf Kompilierbarkeit getestet. Die Einbindung des JiBX-Frameworks wurde durch die Einbindung des *od2java*-Projekts abgelöst und die Pfadangaben zu den XML-Dokumenten durch Pfade zu den erzeugten Objektdiagrammen ersetzt. Zum Abschluss wurde die korrekte Ausführung der migrierten Anwendungen erfolgreich überprüft.

Um die Erweiterbarkeit des Datenmodells nach der Transformation in ein Klassendiagramm zu evaluieren, wurden die folgenden, grundlegenden Änderungsszenarien erprobt:

- Entfernen eines Attributs
- Hinzufügen eines Attributs
- Entfernen einer Komposition
- Hinzufügen einer Komposition

Die Umsetzung dieser Änderungen in einem Klassendiagramm ist simpel und konnte wie erwartet mit geringem Aufwand durchgeführt werden. Danach wurden jeweils die neuen Versionen der Datenhaltungsklassen generiert. Durch die robuste Implementierung der *od2java*-Komponente ließen sich in allen Szenarien die zum ursprünglichen Klassendiagramm konformen Objektdiagramme an die neuen Datenhaltungsklassen binden und durch die Klasse *MarshallingContext* in Objektdiagramme der neuen Struktur überführen.

Bei der Kombination dieser Szenarien, beispielsweise um ein Attribut zwischen zwei Klassen zu verschieben, wurden jedoch die Attributwerte nicht erhalten. Das Framework hat auch in diesen Fällen eine komfortable, automatisierte Migration der Objektdiagramme ermöglicht. Durch die Verschiebung der alten Datenhaltungsklassen in ein anderes Java-Package ließen sich beide Versionen des Datenmodells parallel verwenden. So konnte effizient eine Migrationsanwendung erstellt werden, die die ursprünglichen Objektdiagramme vollständig Einliest und die Attributwerte in Instanzen der neuen Datenhaltungsklassen kopiert. Diese konnten abschließend in OD-Dateien ausgegeben werden, wodurch die Anpassung des Datenmodells und die Migration der Daten abgeschlossen werden konnte.

6.3 Auswertung anhand der Anforderungen

Die folgende Auflistung bewertet die Erfüllung der in Kapitel 3 definierten Anforderungen 1.-10.:

1. Das Migrationsframework hat die Ablösung des XSD-Schemas ermöglicht, indem es dieses in ein Klassendiagramm transformiert hat.
2. Mit Ausnahme der Einschränkungen von Wertebereichen einfacher Inhaltstypen (siehe Abschnitt 7.1.2) wurden dabei alle Informationen der XSD-Grammatik erhalten.
3. Durch die Speicherung in Form eines Klassendiagramms wurde eine bessere Lesbarkeit des Datenmodells erreicht. Die gute Erweiterbarkeit wurde erfolgreich getestet.
4. Durch die Transformation der XML-Dokumente in Objektdiagramme wurden die Daten in Dokumente einer zur Datenhaltung geeigneten DSL überführt.
5. Die Validierung der Daten wurde durch das Migrationsframework zum Zeitpunkt der Evaluierung nicht unterstützt (siehe Abschnitt 7.1.3).
6. Die in den Objektdiagrammen enthaltenen Daten wurde durch das Framework als Java-Objekte mit adäquaten Zugriffsmethoden bereitgestellt.

7. Die Java-Objekte konnten modifiziert und anschließend wieder in Objektdiagramme zurückgeschrieben werden.

8. Alle Aufgaben der Sprachverarbeitung wurden mit Hilfe von neu erstellen oder bestehenden MontiCore-Sprachen bewältigt.

9. Zur Verwendung des Migrationsframeworks waren neben MontiCore keine weiteren externen Softwarebibliotheken erforderlich.

10. Das JiBX-Framework konnte in den untersuchten Anwendungen durch die Einbindung der *od2java*-Komponente abgelöst werden.

Zusammenfassend ist das Ergebnis der Evaluation positiv zu bewerten. Die meisten der Anforderungen wurden während der Tests zufriedenstellend erfüllt. Die nicht gelösten Anforderungen sind der begrenzten Implementierungszeit geschuldet. Für sie werden in Abschnitt 7.1 mögliche Lösungen aufgezeigt und deren Umsetzung im Rahmen weiterführender Arbeiten motiviert.

Der im Rahmen der Arbeit untersuchte Ansatz ist zur Ablösung von XML-Datenhaltungsformaten geeignet.

Kapitel 7

Zusammenfassung und Ausblick

Um die Arbeit abzurunden, werden die Ergebnisse der vorhergehenden Kapitel im Folgenden zusammengefasst. Abschnitt 7.1 zeigt mögliche Anschlusspunkte an die Arbeit.

Im Rahmen der Arbeit wurde das *xsd2dsl*-Framework zur vollständigen Ablösung von XML-Formaten und entsprechenden Daten durch DSLs entwickelt. Dazu wurden einleitend der Nutzen eines derartigen Frameworks motiviert und verwandte Arbeiten analysiert. Dabei wurde festgestellt, dass bisher keine ganzheitlichen Konzepte und Implementierungen im Sinne der Aufgabenstellung vorhanden sind. Dennoch konnten Kenntnisse für die Konzipierung des Frameworks gewonnen werden, da einige der verwandten Arbeiten gute Lösungsansätze für Teilfunktionen des Migrationsframeworks boten.

Nach der Einführung der für die Arbeit relevanten Grundlagen aus den Bereichen *XML*, *XSD*, *DSLs*, *UML/P* und *MontiCore* in Kapitel 2 sind zunächst die durch die Aufgabenstellung gestellten Anforderungen an das Migrationsframework analysiert worden. Anhand der Anforderungen wurde in Kapitel 4 ein detailliertes Konzept für den modularen Aufbau des Frameworks entwickelt. Außerdem wurde festgelegt, dass die XSD-Grammatiken in UML/P-Klassendiagramme und die XML-Daten in UML/P-Objektdiagramme transformiert werden. Die Hauptargumente dafür sind die bereits vorhandene Werkzeuginfrastruktur und die Möglichkeit der Verwendung im Kontext modellgetriebener Softwareentwicklung.

Auf Basis des Konzepts wurde das Migrationsframework anschließend mit Hilfe von MontiCore in Java implementiert. Kapitel 5 beschreibt die Funktionsweise der einzelnen Komponenten, deren Zusammenspiel bei Migration und Datenbindung sowie die erzeugten Artefakte an einem fiktiven Anwendungsbeispiel. Durch die Implementierung wurde die Umsetzbarkeit der erarbeiteten Konzepte belegt und eine solide Grundlage für weiterführende Arbeiten geschaffen.

Kapitel 6 beinhaltet die Evaluierung des Migrationsframeworks anhand eines Fallbeispiels. Dazu wurde der Einsatz des *xsd2dsl*-Frameworks bei der Migration eines im industriellen Kontext eingesetzten XML-Schemas dokumentiert und anhand der in Kapitel 3 definierten Anforderungen ausgewertet.

7.1 Ausblick

In den folgenden Abschnitten werden Aufgaben und Themenbereiche aufgezeigt, die im Rahmen der Arbeit nicht abschließend gelöst beziehungsweise bearbeitet wurden. Sie stellen mögliche Anschlusspunkte für weiterführende Arbeiten im Kontext des Migrationsframeworks dar und geben so einen Ausblick auf dessen weitere Entwicklung und Einsatzmöglichkeiten.

7.1.1 Weitere Evaluierung

Zunächst sollte eine weitere Evaluierung des Migrationsframeworks durch unabhängige Anwender erfolgen. Die Ergebnisse aus Kapitel 6 sind zwar zufriedenstellend, haben aber aufgrund der eigenen Durchführung eine verminderte Aussagekraft. Im Rahmen einer umfassenden externen Evaluierung mit industriell genutzten XML-Formaten können Einsatz- und Entwicklungspotentiale des Frameworks optimal ermittelt werden.

7.1.2 Einschränkung einfacher Inhaltstypen durch OCL/P

Aufgrund der geforderten Kompatibilität zu JiBX wurden bei der Implementierung des Frameworks, mit Ausnahme von Enumerationen, alle Einschränkungen der Wertebereiche einfacher Inhaltstypen durch Verwendung der jeweiligen Basistypen ignoriert. Dadurch gehen zentrale Informationen des ursprünglichen XML-Datenmodells verloren. Ursprünglich invalide XML-Dokumente können durch die aufgehobene Einschränkung von Wertebereichen nach der Migration gültige Datensätze darstellen.

Durch die Integration der *Object Constraint Language* (OCL) [www14q] in die UML/P besteht die Möglichkeit Invarianten innerhalb von UML/P-Modellen auszudrücken. Die Erhaltung der Einschränkungen bei der Transformation der Grammatiken ist daher prinzipiell möglich. Die Umsetzung konnte aber aufgrund des Umfangs im Rahmen dieser Arbeit nicht genauer evaluiert und durchgeführt werden.

7.1.3 Validierung von Objektdiagrammen durch Klassendiagramme

Im Rahmen der Konzeption (vgl. Abschnitt 4.1.1) wurde aufgrund der schlechten Lesbarkeit gegen eine direkte Transformation der XML-Grammatiken in DSL-Grammatiken entschieden. Als Nachteil wurde dabei in Kauf genommen, dass die Validierung der migrierten Daten separat implementiert werden muss. Durch die Wahl von Klassen- und Objektdiagrammen bedeutet dies, dass die im Objektdiagramm enthaltenen Daten anhand der im Klassendiagramm modellierten Datenstrukturen validiert werden müssen.

Auf die Umsetzung der Validierung wurde im Rahmen der Arbeit verzichtet. Es existieren jedoch Arbeiten, die die Machbarkeit von Konsistenzprüfungen zwischen UML/P-Klassen- und -Objektdiagrammen belegen [Hul13, Kus10] und als Grundlage für die Implementierung der Validierung genutzt werden können.

7.1.4 Ablösung von Reflexion bei der Datenbindung

Wie in Abschnitt 5.5 beschrieben, ist die Datenbindung mit Hilfe von Reflexion implementiert. Auch wenn dadurch zur Laufzeit keine Datenstrukturen manipuliert, sondern nur Objekte erzeugt und mit Daten befüllt werden, sollte die Datenbindung auf anderem Wege implementiert werden. Eine Möglichkeit dazu wird in Abschnitt 4.6 beschrieben.

7.1.5 Implementierung von Kontextbedingungen in den verschiedenen Teilprojekten

An vielen Stellen des Frameworks werden mit Hilfe von MontiCore erstellte Grammatiken verwendet. Nicht alle im Kontext des Migrationsframeworks benötigten Eigenschaften der Dokumente der verschiedenen Sprachen können durch Grammatikregeln erzwungen werden. MontiCore bietet jedoch die Möglichkeit durch die Programmierung von Kontextbedingungen sprach- und anwendungsspezifisch Eigenschaften von Eingabedokumenten zu definieren. Diese werden vor der Verarbeitung überprüft, um bei Verstößen detaillierte Warnungen oder Fehlermeldungen auszugeben.

So könnte das *xsd-lang*-Projekt um Kontextbedingungen erweitert werden, die überprüfen, ob ein geparstes XSD-Schema wohlgeformt im Sinne der XML ist. Nicht alle Regeln des Standards, beispielsweise die Eindeutigkeit der Attribute innerhalb eines Elements, lassen sich durch die Grammatikbeschreibung abbilden.

Ein weiteres Beispiel für sinnvolle Kontextbedingung ist die Einschränkung von Klassen- und Objektdiagrammen auf die vom Framework verwendeten Sprachkomponenten. Durch sie kann gewährleistet werden, dass alle im jeweiligen Modell vorhandenen Informationen auch durch das Framework berücksichtigt werden.

Insgesamt wird durch die Implementierung von Kontextbedingungen die Benutzerfreundlichkeit und Robustheit des Migrationsframeworks erhöht.

Anhang A

Mapping von Standardtypen

XSD-Standardtyp	*Java-Typ*
anySimpleType	java.lang.String
anyURI	java.lang.String
base64Binary	byte[]
boolean	boolean
byte	java.lang.Byte
date	java.sql.Date
dateTime	java.util.Date
decimal	java.math.BigDecimal
double	java.lang.Double
duration	java.lang.String
ENTITY	java.lang.String
ENTITIES	java.lang.String
float	java.lang.Float
gDay	java.lang.String
gMonth	java.lang.String
gMonthDay	java.lang.String
gYear	java.lang.String
gYearMonth	java.lang.String
hexBinary	byte[]
ID	java.lang.String
IDREF	java.lang.String
IDREFS	java.lang.String
int	java.lang.Integer
integer	java.math.BigInteger
language	java.lang.String
long	java.lang.Long
Name	java.lang.String
negativeInteger	java.lang.String
nonNegativeInteger	java.lang.String
nonPositiveInteger	java.lang.String
normalizedString	java.lang.String
NCName	java.lang.String
NMTOKEN	java.lang.String
NMTOKENS	java.lang.String
NOTATION	java.lang.String
positiveInteger	java.lang.String
short	java.lang.Short
string	java.lang.String
time	java.sql.Time
token	java.lang.String
unsignedByte	java.lang.String
unsignedInt	java.lang.String
unsignedLong	java.lang.String
unsignedShort	java.lang.String

Tabelle A.1: JiBX-kompatible Zuordnung von XSD-Standardtypen zu Java-Typen

Anhang B

Transformationsergebnisse des Beispielschemas

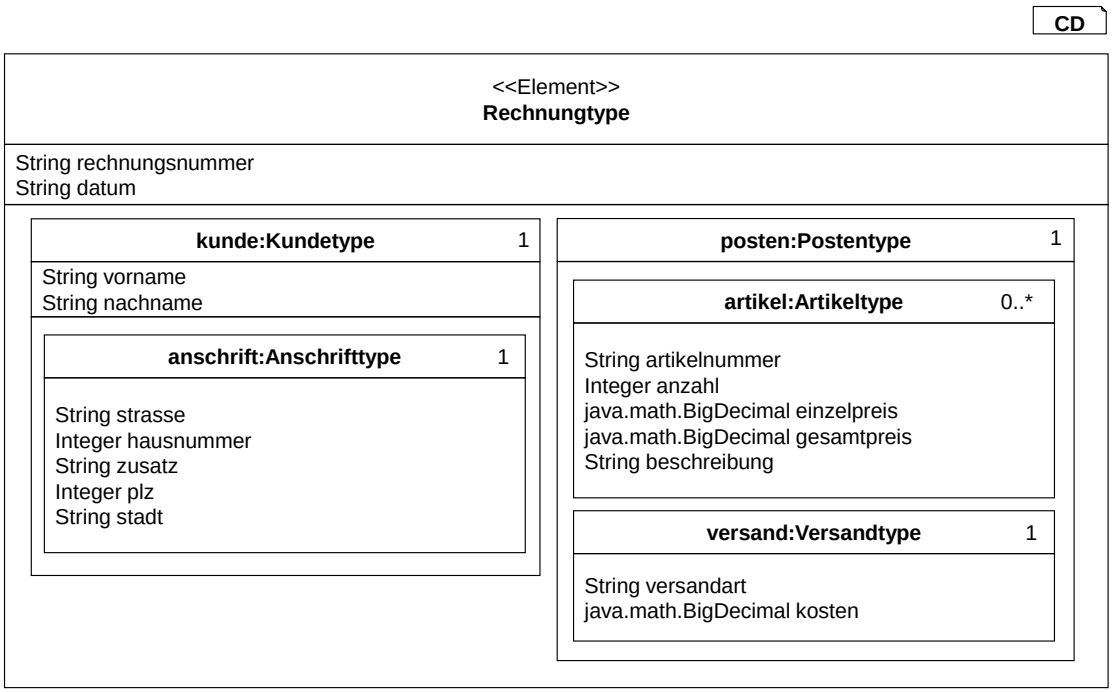


Abbildung B.1: Aus dem Beispielschema erzeugtes Klassendiagramm

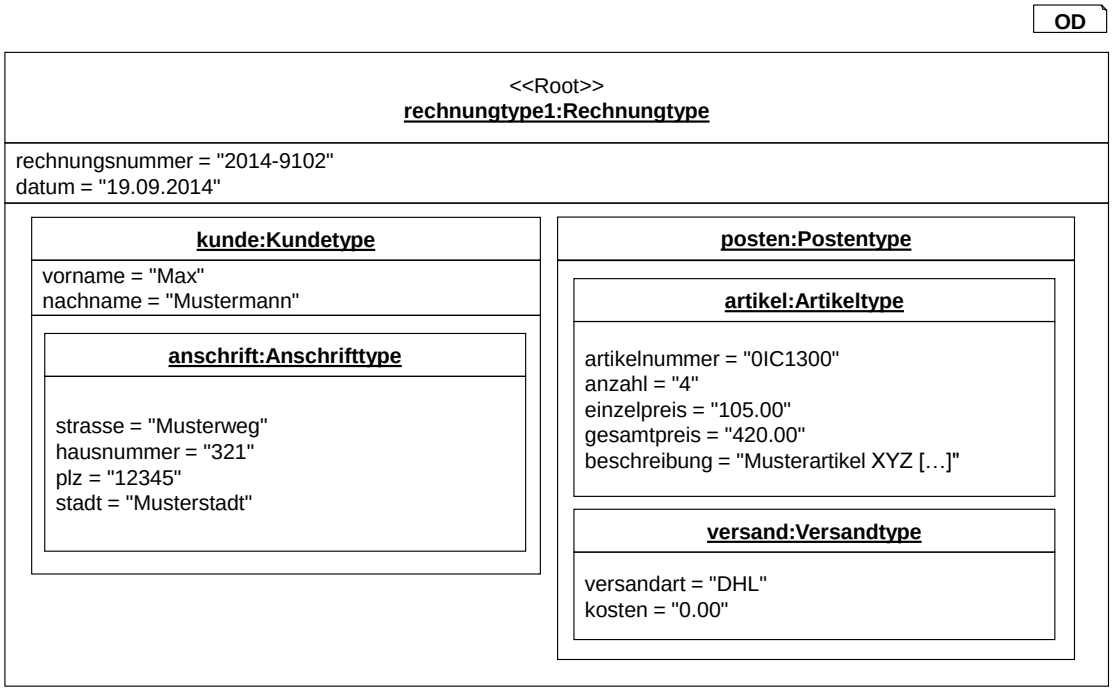


Abbildung B.2: Aus dem Beispieldokument erzeugtes Objektdiagramm

Anhang C

CD

Auf der beiliegenden CD befinden sich folgende Verzeichnisse und Dateien.

- *xsd2dsl*: Implementierung des Migrationsframeworks
  - Unterverzeichnisse für jedes der in Abschnitt 5.1 aufgeführten Projekte
    - *src/main*: Java-Quellcode
    - *src/test*: JUnit-Tests und Testdaten
    - *target*: Java Archive mit Klassendateien
    - *doc*: Javadoc Dokumentation
- *examples*: Anwendungsbeispiele aus Abschnitt 5.6 und Kapitel 6
- *Ausarbeitung.pdf* - Dieses Dokument als PDF-Datei

Literaturverzeichnis

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[BFRS02] P. Bohannon, J. Freire, P. Roy, and J. Simeon. From XML schema to relations: a cost-based approach to XML storage. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 64–75, 2002.

[BKK04] Martin Bernauer, Gerti Kappel, and Gerhard Kramler. Representing XML Schema in UML – A Comparison of Approaches. In Nora Koch, Piero Fraternali, and Martin Wirsing, editors, *Web Engineering*, volume 3140 of *Lecture Notes in Computer Science*, pages 440–444. Springer Berlin Heidelberg, 2004.

[Bos98] J. Bosak. Media-independent publishing: four myths about XML. *Computer*, 31(10):120–122, October 1998.

[BS98] D. Barry and T. Stanienda. Solving the Java object storage problem. *Computer*, 31(11):33–40, November 1998.

[Cer02] Ethan Cerami. *Web services essentials: distributed applications with XML-RPC, SOAP, UDDI & WSDL*. O'Reilly Media, Inc., 2002.

[CHC+13] Christoph Ficek, Hans Grönniger, Christoph Herrmann, Holger Krahn, Claas Pinkernell, Holger Rendel, Bernhard Rumpe, Martin Schindler, and Steven Völkel. *MontiCore 2.2.0 - Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen*. RWTH Aachen, Software Engineering, 2013.

[CZR03] A. Chaudhri, Roberto Zicari, and Awais Rashid. *XML Data Management: Native XML and XML Enabled DataBase Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[FFI+04] Ira R. Forman, Nate Forman, Dr. John Vlissides Ibm, Ira R. Forman, and Nate Forman. Java Reflection in Action, 2004.

[Fla96] David Flanagan. *Java in a Nutshell: A Desktop Quick Reference for Java Programmers*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[GHVJ09] Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Professionelle Softwareentwicklung. Addison-Wesley, 2009.

[HRS+05] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G. Burke, Rajesh Bordawekar, Igor Pechtchanski, and Vivek Sarkar. XJ: Facilitating XML processing in Java. In *14th International Conference on World Wide Web (WWW2005)*, pages 278–287. ACM Press, May 2005.

[Hul13] Max Hultzsch. CDUnit: Unittests für Klassendiagramme durch multimodale Objektdiagrammspezifikationen. Diplomarbeit, Rheinisch-Westfälische Technische Hochschule Aachen, Germany, Juli 2013.

[Jon03] Joel Jones. Abstract syntax tree implementation idioms. In *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003)*, 2003.

[Kra12] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Shaker Verlag, 2012.

[Kus10] Eugen Kuss. Einbettung von UML/P-Klassen- und Objektdiagrammen in Alloy Analyzer zur konstruktiven Konsistenzprüfung. Diplomarbeit, Rheinisch-Westfälische Technische Hochschule Aachen, Germany, Oktober 2010.

[Law04] Ramon Lawrence. The space efficiency of XML. *Information and Software Technology*, 46(11):753–759, 2004.

[LDL08] Tak Cheung Lam, Jianxun Jason Ding, and Jyh-Charn Liu. XML Document Parsing: Operational and Performance Characteristics. *IEEE Computer*, 41(9):30–37, 2008.

[Lee11] David Lee. JXON: an architecture for schema and annotation driven JSON/XML bidirectional transformations. In *Proceedings of Balisage: The Markup Conference*, 2011.

[Mad10] Lech Madeyski. *Test-Driven Development*. Springer, 2010.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: Reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 706–706, New York, NY, USA, 2006. ACM.

[MLMK05] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML Schema Languages Using Formal Language Theory. *ACM Trans. Internet Technol.*, 5(4):660–704, November 2005.

[Nog13] Falco Nogatz. From XML Schema to JSON Schema - Comparison and Translation with Constraint Handling Rules. Bachelorarbeit, Universität Ulm, Germany, December 2013.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[PLMC03] Dongwon Lee Penn, Dongwon Lee, Murali Mani, and Wesley W. Chu. Schema Conversion Methods between XML and Relational Models. In *Yan / Information and Software Technology 48 (2006) 245–252 Knowledge Transformation for the Semantic Web, IOS*, pages 1–17. IOS Press, 2003.

[PQ95] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(K) Parser Generator. *Softw. Pract. Exper.*, 25(7):789–810, July 1995.

[RBG02] Nicholas Routledge, Linda Bird, and Andrew Goodchild. UML and XML Schema. In *Proceedings of the 13th Australasian Database Conference - Volume 5*, ADC '02, pages 157–166, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*. 2. Auflage. Springer, September 2011.

[Rum12] Bernhard Rumpe. *Agile Modellierung mit UML : Codegenerierung, Testfälle, Refactoring*. 2. Auflage. Springer, Juni 2012.

[Sch12] Martin Schindler. *Eine Werkzeuginfrastruktur zur agilen Entwicklung mit der UML/P*. Shaker Verlag, 2012.

[Ter96] P. D. Terry. Compilers and Compiler Generators: An Introduction with C++, 1996.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific Languages: An Annotated Bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[Vö11] Steven Völkel. *Kompositionale Entwicklung domänenspezifischer Sprachen*. Shaker Verlag, 2011.

[www14a] ANTLR. `http://www.antlr.org/`, August 2014.

[www14b] Apache Maven Project. `http://maven.apache.org/`, August 2014.

[www14c] Comparing XML Schema Languages. `http://www.xml.com/pub/a/2001/12/12/schemacompare.html`, September 2014.

[www14d] Document Object Model (DOM). `http://www.w3.org/DOM/`, September 2014.

[www14e] Extensible Markup Language (XML). `http://www.w3.org/XML/`, August 2014.

[www14f] Extensible Markup Language (XML) 1.0. `http://www.w3.org/TR/1998/REC-xml-19980210`, August 2014.

[www14g] Extensible Markup Language (XML) 1.0 (Fifth Edition). `http://www.w3.org/TR/REC-xml/`, August 2014.

[www14h] FreeMarker Java Template Engine. `http://freemarker.org/`, September 2014.

[www14i] Java programming dynamics, Part 2: Introducing reflection. `http://www.ibm.com/developerworks/java/library/j-dyn0603/`, September 2014.

[www14j] Javadoc Tool Home Page |. `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`, August 2014.

[www14k] JiBX: Binding XML to Java Code. `http://www.jibx.org/`, August 2014.

[www14l] JSON. `http://www.json.org/`, September 2014.

[www14m] JSON Schema. http://json-schema.org/, September 2014.

[www14n] JSR-000222 JavaTM Architecture for XML Binding 2.2 - Maintenance Release 2. http://jcp.org/aboutJava/communityprocess/mrel/jsr222/index2.html, August 2014.

[www14o] JUnit. http://junit.org/, August 2014.

[www14p] MontiCore: Data Explorer. http://www.monticore.de/dex/, August 2014.

[www14q] OCL. http://www.omg.org/spec/OCL/, September 2014.

[www14r] Unified Modeling Language (UML). http://www.uml.org/, August 2014.

[www14s] World Wide Web Consortium (W3C). http://www.w3.org/, September 2014.

[www14t] XML Applications and Initiatives. http://xml.coverpages.org/xmlApplications.html, September 2014.

[www14u] XML Schema. http://www.w3.org/XML/Schema, August 2014.

[www14v] XMLBeans. http://xmlbeans.apache.org/, September 2014.

[www14w] XMLmodeling.com | Creating Innovation Through Technology. http://xmlmodeling.com/, August 2014.

[www14x] XSL Transformations (XSLT) Version 2.0. http://www.w3.org/TR/xslt20/, September 2014.