Hochschule Darmstadt

- Fachbereich Informatik -

Transformation des JAXR-Datenmodells in UML/XMI

Abschlussarbeit zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

vorgelegt von

Matthias Küch

Referent:	*Prof. Dr. Gerhard Raffius*
Korreferent:	*Prof. Dr. Wolfgang Weber*
Betreuer:	*Dr. Harald Schöning*

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner Prüfungsbehörde eingereicht worden.

Darmstadt, den 19. Mai 2008

Abstract

Im Bereich der Serviceorientierten Architekturen (SOA) gibt es verschiedene Standards zur Verwaltung von Diensten (Registries). Zur Kapselung dieser Standards wurde die JAVA API for XML Registries (JAXR) entwickelt.
JAXR bietet Anwendungen die Möglichkeit, mit einheitlichen Mitteln auf unterschiedliche Registries zuzugreifen. Zur internen Kompatiblität der zu Grunde liegenden Registry-Datenmodelle, wurde durch JAXR ein eigenes Datenmodell spezifziert.

CentraSite, die SOA-Lösung der Software AG, implementiert JAXR inklusive dieses Datenmodells und erweitert es um benutzerdefinierte Objekttypen.
Inhalt dieser Arbeit ist die Abbildung des JAXR-Datenmodells und seiner Erweiterungen in die Unified Modelling Language (UML). Dazu wurden die grundlegenden Konzepte des Datenmodells untersucht und semantische Ähnlichkeiten zu Konzepten der UML ermittelt. Ziel war es, die Semantik von JAXR-Konzepten zu behalten und die Mittel der UML sinnvoll ihrer Bedeutung entsprechend einzusetzen. Die Abbildung war möglich, da beide Modelle einen objektorientierten Hintergrund haben.
Neben der Wahrung der Semantik wurde ein ausgewogenes Verhältnis von Realisierbarkeit und Verständlichkeit angestrebt.

Zur automatisierten Generierung des Modells wurde der XML-Dialekt XMI evaluiert. Mithilfe von XMI können UML-Modelle als XML-Dokumente dargestellt werden. Dabei wurden Unterschiede in der Darstellung und die Portabilität der XMI-Modelle anhand von gebräuchlichen Modellierungswerkzeugen untersucht. Als Ergebnis dieser Untersuchung konnte die teilweise Inkompatibilät von exportierten XMI-Modellen, speziell bei Erweiterungen des UML-Modells durch Profile, nachgewiesen werden.

Vorwort

Im Vorfeld dieser Arbeit wurde ich häufig von Freunden, Familienmitgliedern und Kommilitonen zum Thema meiner Bachelorarbeit angesprochen. Eine verständliche Antwort fiel mir dabei nicht leicht, da es sich bei der Entwicklung von Modellen um abstrakte Abbildungen von komplexen Themengebieten handelt.

Ein plastisches Beispiel wäre hier die Neueröffnung eines Warenhauses mit verschiedenen Abteilungen und Angeboten. Jede Abteilung hat einen Namen, ein Geschäftsfeld und eine Menge von Dienstleistungen.
Zur Eröffnung würden sich die zukünftigen Besucher stark für das Angebot interessieren. Aufgrund der Fülle des Angebots und der Komplexität des Gebäudes wäre eine Erkundung zu Fuß jedoch sehr anstrengend und Zeit raubend. Obendrein wäre der Besucher von der riesigen Angebotsvielfalt überfordert.

Als Angebot des Warenhauses würde nun ein Weg gesucht werden, den zukünftigen Besuchern die Struktur und Angebotsvielfalt mit einfachen Mitteln darzustellen.
Um die Architektur des gewaltigen Gebäudes zu beschreiben würde man vereinfachte Baupläne, die der Mehrheit der Besucher etwa vom Bau ihres Eigenheims bekannt sind, verwenden.
Zur Veranschaulichung des Warenangebotes würde man sich eine Symbolik für verschiedene Produktgruppen einfallen lassen. Ein Hemd für Bekleidung, ein Pinsel für Malerarbeiten, eine Waage für Rechtsbeistand.

Mithilfe dieser, zugegeben umfangreichen, Abbildung wären die Besucher in der Lage, mit vertrauten Mitteln und Darstellungsformen, die komplexe Welt des Warenhauses zu überblicken und zu verstehen.

Teil meiner Arbeit war es durch das Warenhaus ”JAXR“ zu gehen und die Geschäfte zu erkunden, die Baupläne zu verstehen und Warenlisten zu studieren. Schlussendlich sollte ich den Fülle von Informationen in eine einfache, allgemein verständliche, Darstellungsform (UML) umwandeln.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Die *Software AG* in Darmstadt-Eberstadt bietet *CentraSite* als umfangreiche Lösung zur Implementierung von Serviceorientierten Architekturen (SOA) an. CentraSite liegt das JAXR-Datenmodell der JAXR-Spezifikation von Sun zu Grunde. Durch Erweiterungen kann dieses Modell einen erheblichen Umfang annehmen, zu dessen Übersicht eine Darstellung in einer einheitlichen Modellierungssprache realisiert werden soll.

Durch die Etablierung der *Unified Modelling Language* (UML) als *Lingua franca* der Softwaremodellierung nimmt der Wunsch nach sprachlicher Einheit im Bereich der Informatik zugunsten der UML zu. Entscheidungsträger wollen oder müssen ihre Entscheidungen mithilfe von UML-Diagrammen begründen und evaluieren.
Auch bei der Dokumentation von SOA-Umgebungen böte sich eine Darstellung in UML an. Durch die standardisierte Darstellung fällt es leichter, einen komplexen Themenbereich mit bekannten und erlernten Werkzeugen und Methoden zu verstehen und zu interpretieren. Zwar gibt es verschiedene Wege JAXR-Datenmodelle darzustellen, das Verständnis über die komplexen Beziehungen untereinander (Assoziationen, Vererbungen, Klassifikationen, etc.) werden durch das Einführen von zusätzlichen Notationen jedoch nicht erleichtert.

Die bisherige Lösung über das CentraSite-Frontend *CentraSite Control* stellt Typen und Beziehungen in tabellarischer Form dar. Dadurch können zwar typspezifische Informationen relativ schnell dargestellt werden, umfassende, das ganze Modell einschließende

Informationen lassen sich jedoch nur schwerlich herauslesen.
Ein Zugriff über die XML-Schnittstelle (XQuery) der zu Grunde liegenden Datenbank verspricht noch weniger Übersicht, eröffnet aber eine Möglichkeit zur Gewinnung der abgelegten Informationen und ihrer Weiterverarbeitung in XML.

Gewünscht ist also die Darstellung des Datenmodells als ein hierarchisches Modell mit Rücksicht auf die Typen, Beziehungen, Klassifikationen und Erweiterungen des JAXR-Datenmodells. Die UML mit ihren vielfältigen Darstellungsformen und ihrer Rolle als Industriestandard wurde daher als Mittel zur Realisierung dieser Anforderungen ausgewählt.

1.2 Zielsetzung

Als Teil dieser Arbeit sollte ein Weg gefunden werden, die durch JAXR spezifzierten und durch CentraSite erweiterten Konzepte des JAXR-Datenmodells mit den Mitteln der UML darzustellen. Auf Basis von CentraSite soll es ermöglicht werden, das zu Grunde liegende JAXR-Datenmodell dynamisch in ein UML-Modell umzuwandeln. Die Transformation des Modells soll dynamisch und automatisch ablaufen und eine Möglichkeit zur Importierung in Modellierungswerkzeuge bieten.
Dabei sollen Formen der Abbildung und der technischen Realisierung (XMI) betrachtet werden.

1.3 Serviceorientierte Architekturen

Serviceorientierte Architekturen (SOA) haben in den vergangenen Jahren eine zentrale Rolle in der IT-Landschaft eingenommen. Auch in Zukunft wird der Anteil von SOA in Unternehmen ansteigen und zentraler Bestandteil der IT-Infrastruktur werden. [CIO08] Diese Arbeit befasst sich mit Serviceorientierten Architekturen und bildet deren zugrundeliegende Modelle auf eine allgemeine Darstellungsform ab.

Zur Verdeutlichung der später häufiger verwendeten Begriffe werden diese im Folgenden eingeführt und besprochen.

1.3.1 Registry

In der Literatur wird eine *Registry* als eine Art "Gelbe Seiten" für *Services* bezeichnet. Der deutsche, in dem Zusammenhang aber nicht etablierte, Ausdruck für Registries lautet *Dienstverzeichnis*.

Eine Registry verbindet dynamisch und lose gekoppelte B2B (Business to Business) Parteien und nimmt dabei die Rolle eines Dritten ein. [SUN02]

Vereinfacht dargestellt wird es mithilfe einer Registry, die es dem Kunden ermöglicht Dienste (z. B. Webservices), von denen er nur die Anforderungen kennt, nicht aber Namen oder Anbieter, herauszufinden und in Anspruch zu nehmen. Im Gegenzug wird dem Anbieter der Dienste die Möglichkeit gegeben, für seine Dienste zu werben und die Benutzung abzurechnen. [M$^+$07]

Registries stellen Daten und Metadaten über die angebotenen Dienste bereit. Mit diesen Daten wird es ermöglicht, Dienste in einem öffentlichen Netzwerk oder innerhalb eines Unternehmens (Intranet) bereitzustellen und zugänglich zu machen.
Eine Registry stellt hier einen Mechanismus zum Ausfindigmachen und Gebrauchen des Webservices breit. Dies erfolgt auf standardisierte Art und Weise und kann von unterschiedlichen Applikationen ausgeführt werden. [OAS03]

Klassifikation

Eine in diesem Umfeld häufig anzutreffende Beziehung zwischen Elementen eines Modells ist die Klassifikation. Zum Finden eines Service ist eine ausreichende und erfolgte Klassifizierung des Selbigen notwendig. Die Klassifizierung erfolgt durch die Betrachtung des Services nach verschiedenen Kriterien z. B.: Funktionalität, Benutzung, Konstruktion und Form des Aufrufs. [Ora06]
Eine sinnvolle Einteilung wäre zum Beispiel die Klassifizierung von E-Mail Diensten als Infrastrukturservices oder Services zur Transformation von Daten als Hilfsdienste.

Zur Klassifizierung ist ein tieferes Verständnis der Geschäftsprozesse und der angebotenen Dienste unumgänglich. So verfolgen etwa Klassifizierungen nicht logische Vorgaben, sondern stellen betriebliche oder gesetzliche Rahmenbedingungen dar.

Die *Taxonomie* ist eine Menge von *Konzepten* zur Klassifizierung von Typen. Eine Klassifikation beschreibt also die *erfolgte Klassifizierung* eines Typs mit einem Konzept aus einer Taxonomie. Als Klassifikation wäre für ein Unternehmen der Standort “Europa” denkbar. Wobei “Europa” ein Konzept der Taxonomie ”Geografie“ wäre.

1.3.2 Repository

Der Augenmerk dieser Arbeit liegt auf SOA-Registries, in diesem Teil sollen jedoch zur Abgrenzung auch SOA-Repositories erwähnt werden.

Im Repository, oder Verzeichnisdienst, sind die physischen Elemente eines Webservices abgelegt z. B. WSDL oder Teile des Programmcodes. In einem Repository werden die Instanzen von den Informationen abgelegt, die in die Registry vom Anbieter eingetragen werden.
Generell stellen Repositories in der Informatik einen Platz zum Speichern von Daten dar.

1.3.3 Unterschied Registry und Repository

Um ein Beispiel aus der Realität zu nehmen, könnte man ein Registry als eine Karteikartensammlung in einer Bibliothek betrachten. Auf jeder Karteikarte sind Informationen über den Autor, das Erscheinungsdatum und die maximale Ausleihdauer eines Buches festgehalten.

Ein Repository wäre dann eine riesige Bücherkiste, in der ein guter Bibliothekar Bücher aufgrund seiner Karteikartensammlung wiederfinden kann. [fre08]

Kapitel 2

Java API for XML Registries (JAXR)

Mit der Spezifikation der *Java API for XML Registries* (JAXR) wurde dem Wunsch nach einer transparenten und Registry-unabhängigen API für Registrystandards nachgekommen. In erster Linie wurde JAXR mit Sicht auf die Kompatibilität zu den beiden dominierenden Registries *UDDI* und *ebXML* entwickelt (siehe Abbildung 2.1).
Zuvor hatten Unternehmen, die in ihrem Geschäftsfeld verschiedene Registries einsetzten, keine Möglichkeit Informationen untereinander auszutauschen. [Jav02]

JAXR ist im *Java Specification Request* (JSR) 93 spezifiziert und im *Web Services Developer Pack* (WSDP) als Referenzimplementierung enthalten. [Har08] JAXR ist Teil der J2EE 1.4 Spezifikation. [Jav02]

Um eine grundlegende Kompatibilität zu UDDI und ebXML zu gewähren, ist das JAXR-Datenmodell weitestgehend abwärts kompatibel zu ebXML und UDDI. Zwar bringt JAXR keine ebXML/UDDI-Implementierungen mit, bildet aber typische Elemente nach, sodass eine Portierung erleichtert wird. JAXR basiert größtenteils auf dem Datenmodell von ebXML. [fre08] Dennoch finden wesentliche Elemente wie `Organizations` und `Users` ihren Ursprung in der UDDI.
Um die Besonderheiten beider Standards zu berücksichtigen, wurde ein eigenes Datenmodell für JAXR eingeführt (siehe auch Abbildung 2.3 auf Seite 14).
Diese Arbeit befasst sich mit dem JAXR-Datenmodell (engl. JAXR Information Model).

Das JAXR-Datenmodell besteht aus mehreren Typen, die vom Typ `RegistryObject` erben. Dadurch erhalten die erbenden Typen einen Namen, einen eindeutigen Bezeichner (UDDI-Key) und eine Beschreibung. Durch die Eigenschaften von `RegistryObject` kann eine Subklasse mit anderen Einträgen assoziiert und klassifiziert werden. [Sof08b]

2.1 Registry-Standards

2.1.1 electronic Business XML (ebXML)

ebXML resultierte aus einem Zusammenschluss von OASIS und UN/CEFACT[1] und wird heute von etwa 1000 weiteren Mitgliedern unterstützt. ebXML soll den Austausch von XML-Dokumenten im elektronischen Handel standardisieren und fördern. [Mar08] Weiterhin soll es für kleine und mittelständische Unternehmen leichter gemacht werden, am Welthandel teilzunehmen. ebXML

Mit ebXML werden Informationen zu Unternehmen ausgetauscht. Im einfachsten Fall heißt das: Name, Adresse und Ort. Ziel der Entwicklung von ebXML war es nicht bestehende Formulare und Dokumente zu replizieren, sondern diese komplett in die elektronische Welt zu überführen und Geschäftsprozesse geschlossen in dieser Domäne ablaufen zu lassen. [Web00]

2.1.2 Universal Description, Discovery, and Integration (UDDI)

UDDI wurde durch eine Initiative von IBM, Microsoft und Ariba im September 2000 eingeführt. UDDI soll das aus den USA bekannte Prinzip der Gelben, Weißen und Grünen Seiten abbilden. [M$^+$07]

Ziel von UDDI ist die Entwicklung eines Standards zur Beschleunigung der Integration von Systemen im Marktplatz Internet. UDDI spezifiziert eine Verzeichnisstruktur

[1] UN/CEFACT: United Nations Centre for Trade facilitation and Electronic Business, mit der Aufgabe durch Standardisierung das globale Wachstum zu fördern.

zur Verwaltung von Webservice-Metadaten. Metadaten bestehen in dem Zusammenhang aus allgemeinen Anforderungen, Webservice-Eigenschaften und den benötigten Informationen zur Auffindung des Webservices in der Registry.

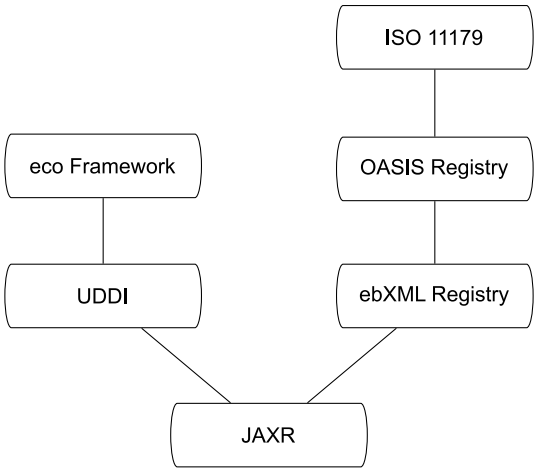

Abbildung 2.1: Entstehung JAXR

2.1.3 JAXR-Erweiterungen

Im Vergleich zu UDDI bringt JAXR folgende Erweiterungen mit: [Har08]

- Mit JAXR werden Assoziationen eingeführt. Eine Assoziation verbindet ein `RegistryObject` mit einem Anderen.
- Mithilfe von Slots können zusätzliche Informationen zu `RegistryObjects` hinzugefügt werden.
- Durch Vererbung besteht die Möglichkeit, das Modell mit eigenen Typen zu erweitern (siehe *User Defined Objects* 2.3.3).

2.2 Software AG CentraSite

CentraSite ist eine Registry- und Repositorylösung die in Zusammenarbeit von Fujitsu und der Software AG (SAG) entwickelt wurde.
CentraSite ermöglicht das Verwalten von verschiedenen SOA-Komponenten, darunter ein erweiterbares Datenmodell und Business Process Management (BPM).
Mit Reporting ist es möglich, Services und ihre Verwendungen zu identifizieren und ihre Auswirkungen zu analysieren.[Sof08a]

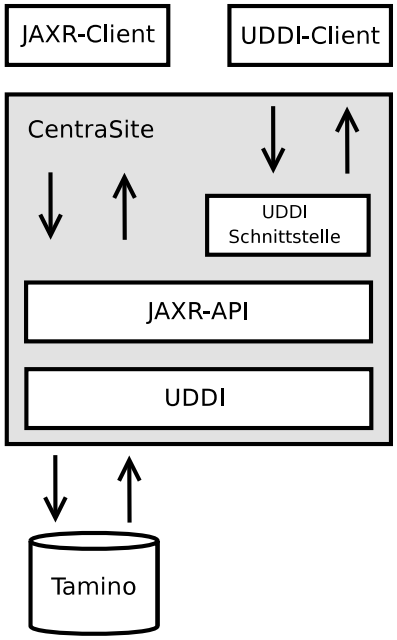

Abbildung 2.2: Aufbau CentraSites als JAXR-Implementierung

Als Datenbank bringt CentraSite *Tamino* mit, die XML-Datenbank der SAG. Die CentraSite-Registry fußt auf UDDI, zu dessen Zugreifbarkeit JAXR implementiert wurde (siehe Abbildung 2.2). Im Grunde besteht die CentraSite-Registry also aus JAXR + UDDI + Tamino. Der Registry-Teil ist im Sinne der JAXR-Spezifikation implementiert. Neben dem Registryzugriff über die JAXR-API wird eine native UDDI-Schnittstelle für die in der Registry gespeicherten UDDI-Elemente angeboten. [Har08]

Zur Administration steht ein Eclipse-Plugin und eine AJAX-basierte GUI für den Browser namens *CentraSite Control* bereit.

2.3 JAXR-Datenmodell

In der Folge wird von Objekttypen die Rede sein, die an verschiedenen Stellen in der Fachliteratur auch als Klassen bezeichnet werden. Zur schärferen Abgrenzung, insbesondere bei der späteren Einführung von UML-Klassen, werden in dieser Arbeit JAXR-Typen als Typen oder Objekttypen bezeichnet. Die Bezeichnung `RegistryObject` wird dann verwendet, wenn die Eigenschaft eines Typs als Subklasse von `RegistryObject` hervorgehoben werden soll.

Wenn auch nicht alle Typen gemäß JAXR-Spezifikation durch CentraSite abgebildet werden, so sollen hier doch alle Typen besprochen und die Realisierung und Verwendung in der vorliegenden JAXR-Implementierung dargestellt werden.
Siehe dazu auch die JAXR-Spezifikation in Version 1.0. [SUN02]

Die Gruppierung von nach `RegistryObject` und `RegistryEntry` abgeleiteten Objekttypen dient hier zur Gruppierung von Objekttypen im Modell. `RegistryObjects` dienen dem Aufbau einer Registry mit Strukturelementen (`Association`, `Classification`, etc.). `RegistryEntries` füllen die Registry (`Service`, `User Defined Objects`, etc.).

2.3.1 RegistryObject

Ein `RegistryObject` ist eine abstrakte Basisklasse für alle anderen Typen im Modell. Typen, die von `RegistryObject` erben, erhalten Metainformationen (Name, Beschreibung, ...) und Beziehungen zu weiteren RegistryObject-Elementen. Durch Assoziationen können weitere Metadaten definiert werden. [SUN02]

`RegistryObjects` beschreiben die Beschaffenheit von Objekttypen auf Instanzebene, z. B. "Software AG" als Instanz von `Organization`.

Ein `RegistryObject` besteht aus mehreren Slots, Assoziationen und Klassifikationen, die im Folgenden beschrieben werden.

Folgende Typen erben unmittelbar von `RegistryObject`:

Organization Die `Organization`-Datenstruktur modelliert Unternehmen und Organisationen. `Organizations` können untereinander assoziiert werden und von anderen `Organizations` erben.

Eine `Organization` kann immer mehrere Services bereitstellen. Ein Unternehmen bietet also mehrere Dienste an.

ServiceBinding Mit `ServiceBindings` können technische Informationen zur Verwendung von `Services` angegeben werden.

SpecificationLink `ServiceBindings` sind mit ein oder mehr `SpecificationLink`-Instanzen assoziiert, welche z. B. auf WSDLs zeigen, also den technischen Spezifikationen eines Services.

Classification Die `Classification` dient zur hierarchischen Klassifikation von Typen. `RegistryObjects` können mit mehreren `Classfications` assoziiert werden. Zum Beispiel kann eine Organisation nach Industrie, Produktkette und Standort klassifiziert werden.
Siehe dazu auch *Klassifikation* 1.3.1 auf Seite 4.

Association `RegistryObject`-Subklassen können mit ein oder mehreren `RegistryObject`-Subklassen assoziiert werden. JAXR definiert den `Association`-Typ als Möglichkeit, `targetObject` und `sourceObject` zu definieren. Somit entsteht durch die Festlegung von Quelle und Ziel eine gerichtete Assoziation. Dabei wird der Typ der `Association` durch die Klassifikation mit der Taxonomie `associationType` bestimmt.
Mögliche `associationType`-Konzepte werden durch `Concept`-Instanzen unterschieden.

Concept Taxonomien bzw. `ClassificationSchemes` repräsentieren eine Menge von `Concepts` bzw. Konzepte und bringen diese in eine strukturierte Beziehung zueinander.

Slot Mit `Slots` wird die Möglichkeit geboten, zusätzliche Attribute zu `RegistryEntries` hinzuzufügen. Die Einführung von `Slots` trägt zur Erweiterung des JAXR-Datenmodells bei.
Mithilfe von Slots können beispielsweise Kardinalitäten und Namen zu Assoziationen spezifiziert, aber auch komplexe Assoziationsklassen realisiert werden.

AuditableEvent `AuditableEvent`-Instanzen sind `RegistryObjects`, die dazu verwendet werden, einen *Audit Trail* zu erstellen. Ein Audit Trail ist eine chronologische Sequenz von Aktivitäten, die in einer bestimmten Zeit auf einem Objekttyp ausgeführt wurden.

`AuditableEvents` werden zwar in CentraSite implementiert, jedoch nicht als Typen auf Modellebene verstanden. Somit findet eine Abbildung auf UML nicht statt.

User `User`-Instanzen sind mit `Organizations` assoziierte `RegistryObjects`, die Informationen über registrierte Benutzer in der Registry bereitstellen. Zusätzlich dienen sie dem Aufbau des Audit Trails. So kann bestimmt werden, welcher Benutzer für einen spezifischen Objekttyp im Modell verantwortlich ist.

ExternalIdentifier `ExternalIdentifier`-Instanzen ermöglichen es, weitere Metadaten zu einem `RegistryObject` hinzuzufügen. Dazu zählen z. B. D-U-N-S[2], eine länderspezifische Sozialversicherungsnummer oder ein Alias-Name.

`ExternalIdentifier` wird als Attribut "externalIdentifier" zu `RegistryObject` hinzugefügt.

PostalAddress `PostalAddress` ermöglicht die Darstellung einer postalischen Adresse für eine `Organization` oder einen `User`.

In der vorliegenden JAXR-Implementierung wird die Adresse als Attribut "postalAddress" von `Organization` mit dem Datentyp `PostalAddress` modelliert.

RegistryEntry Siehe *RegistryEntry* 2.3.2.

[Zai06] [SUN02]

[2] D-U-N-S: Weltweites System zu Identifizierung von Unternehmen über einen neun-stelligen Zahlencode

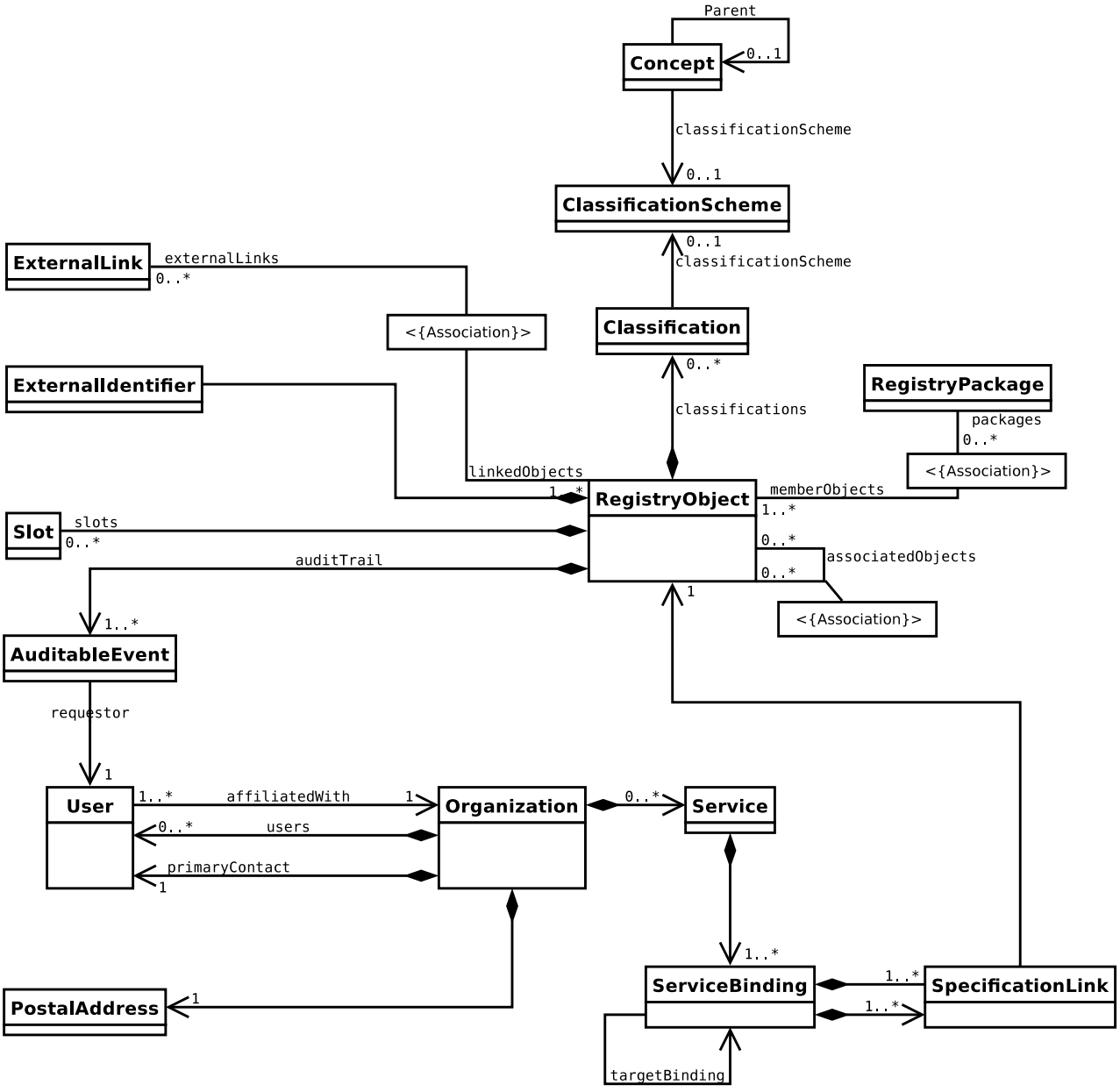

Abbildung 2.3: JAXR-Datenmodell gemäß Spezifikation [SUN02]

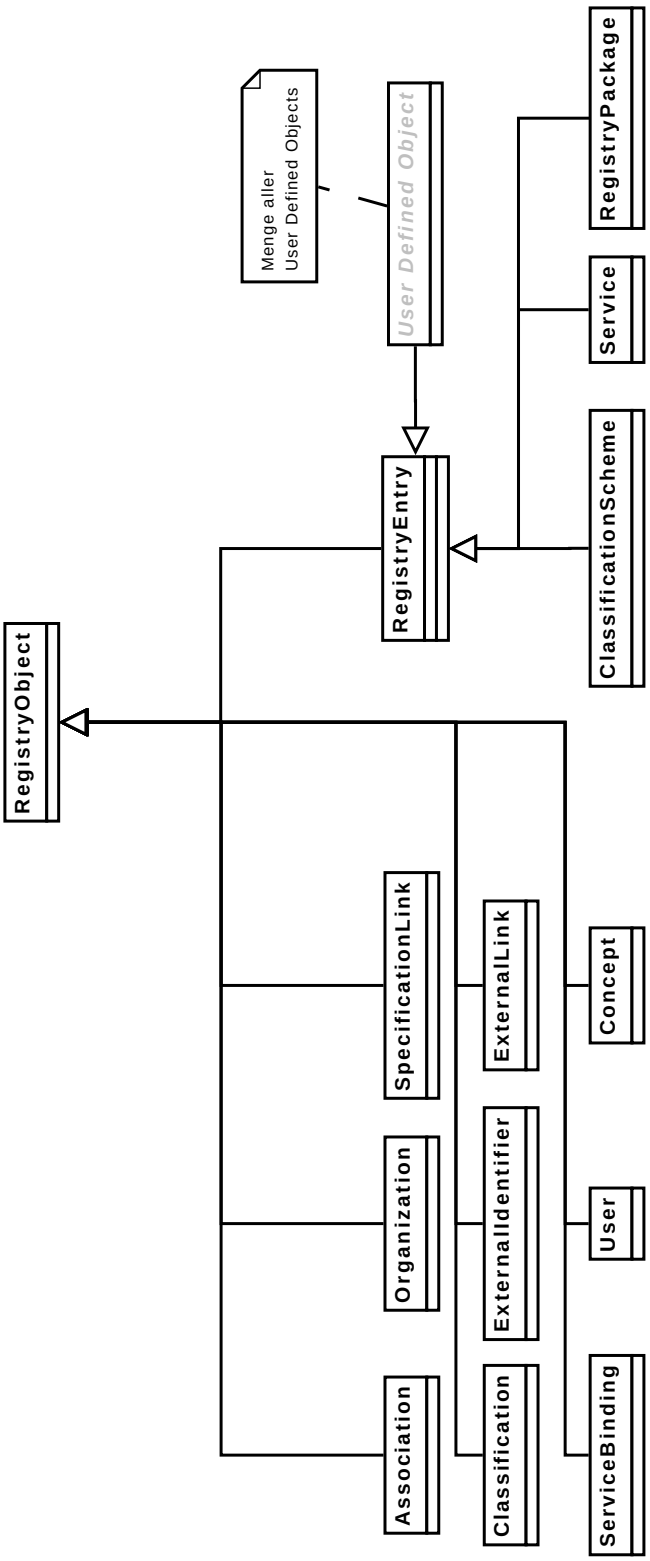

Abbildung 2.4: JAXR-Vererbungshierachie mit CentraSite Erweiterung

2.3.2 RegistryEntry

`RegistryEntries` repräsentieren Typen der Geschäftsprozessebene im Modell. Darunter fällt etwa der Objekttyp `Service`. Im Unterschied zu `RegistryObjects` erhalten diese Typen zusätzliche Metadaten um z. B. eine Versionierung zu ermöglichen. [SUN02]

ClassificationScheme `ClassificationSchemes` repräsentieren Taxonomien mit denen `RegistryObject`-Subklassen klassifiziert werden können.

Service Mit dem `Service`-Typ können im Repository befindliche Webservices dargestellt und mit Metadaten versehen werden.
Weitere Metadaten können mit `ServiceBindings` definiert werden.

RegistryPackage Ein `RegistryPackage` gruppiert eine beliebige Menge von `RegistryObject`-Instanzen zu einer logischen Gruppe von Typen. Dabei kann ein `RegistryObject`-Subtyp beliebig vielen Packages angehören.

ExternalLink Mit einem `ExternalLink` wird ein externer Inhalt durch eine URI[3] referenziert.
Ein typischer `ExternalLink` könnte etwa die Homepage eines Unternehmens sein, die dann mit der entsprechenden `Organization`-Instanz verbunden werden würde.

User Defined Objects Siehe *User Defined Objects* 2.3.3.

[3]URI: Uniform Resource Identifier, Zeichenkette um eine Ressource eindeutig zu identifizieren.

2.3.3 User Defined Objects

CentraSite führt die Möglichkeit der *User Defined Objects* ein. Damit können über *CentraSite Control* zusätzliche Objekttypen angelegt werden.

User Defined Objects werden nicht von JAXR spezifiziert, jedoch durch Vererbung in dieser oder ähnlicher Form ermöglicht (siehe *JAXR-Erweiterungen* 2.1.3). JAXR sieht standardmäßig keine benutzerdefinierte Typen vor. Durch das CentraSite-Konzept der User Defined Objects wird diese Einschränkung aufgehoben.
CentraSite greift dabei nicht über die JAXR-API, sondern direkt auf die zu Grunde liegende Datenbank zu. Dort werden dann die neuen Typen als Subtypen von `RegistryEntry` angelegt. [Har08]

Durch zusätzliche Typdefinitionen kann der Modellierer das JAXR-Modell so an seine eigenen Bedürfnisse anpassen. Typischerweise übersteigt die Anzahl der User Defined Objects dabei leicht die der statischen Typen und bildet einen Großteil des Modells.

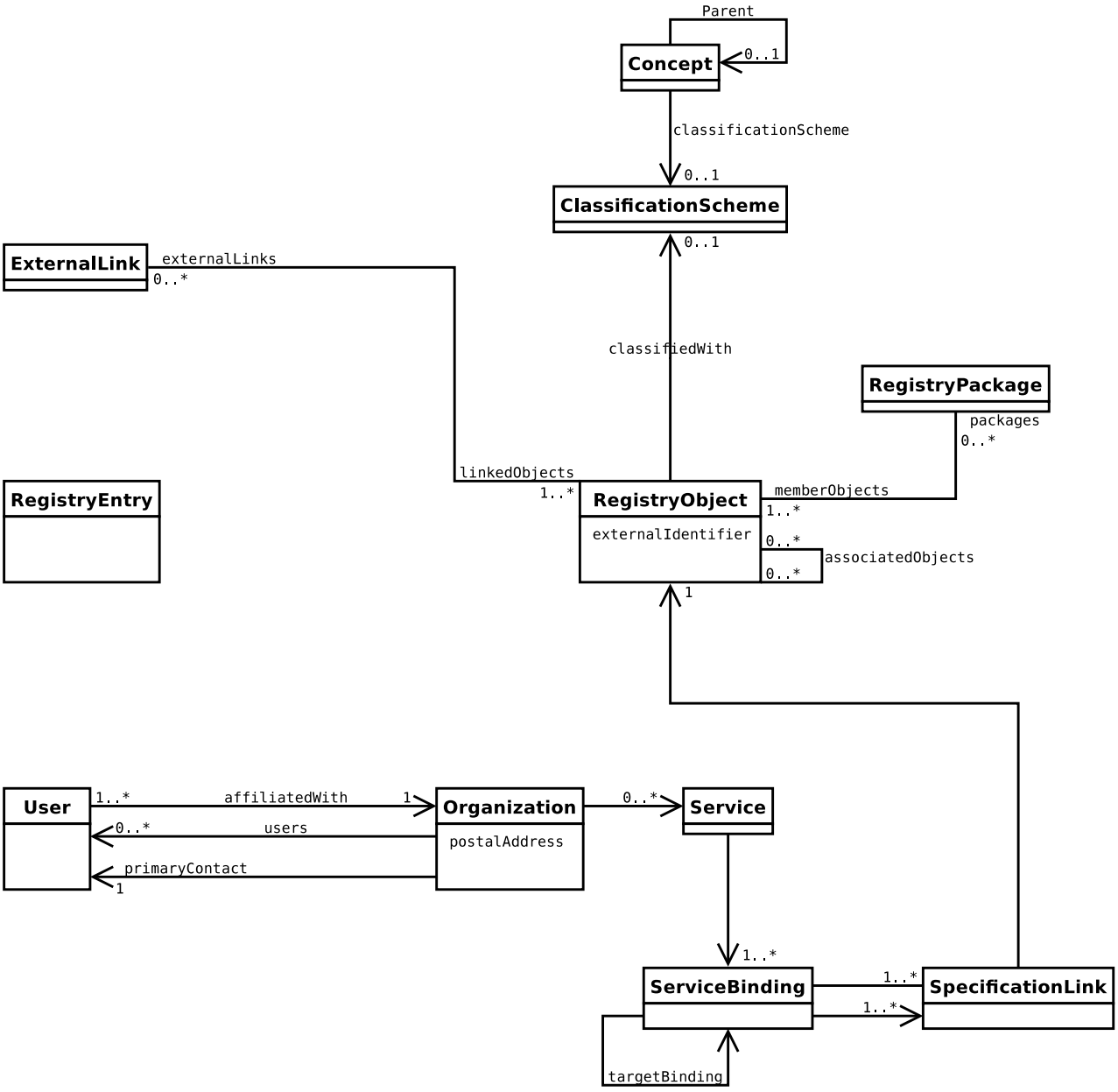

Abbildung 2.5: JAXR-Datenmodell nach CentraSite JAXR-Implementierung

Kapitel 3

Abbildung von JAXR auf UML

3.1 Konzept

Dieses Kapitel befasst sich mit der Abbildung des JAXR-Datenmodells auf ein UML-Modell. Dazu gehört die sinnvolle Darstellung von JAXR-Elementen auf Konzepte der UML. Bereits in der JAXR-Spezifikation wird versucht, die Elemente von JAXR in einer UML-ähnlichen Notation darzustellen. Dabei werden jedoch verschiedene logische Ebenen vermischt und unschlüssige Darstellungsformen eingeführt (siehe Abbildung 2.3 auf Seite 14).

Aufgabe dieser Abbildung ist die Betrachtung aller von `RegistryObject` abgeleiteten Typen, d. h. all jene Typen, die in der Registry angelegt wurden bzw. bereits als Teile der JAXR-Spezifikation existieren. Weiterhin soll ein schlüssiges Gesamtkonzept zum Verständnis und logischen Nachvollziehbarkeit des Modells entstehen.
Die Abbildung von JAXR auf UML wird im Folgenden in einen statischen und in einen dynamischen Teil untergliedert.

Der *statische* Teil umfasst die bereits beschriebenen Elemente der JAXR-Spezifikation (siehe *JAXR-Datenmodell* 2.3) und ihrer nicht generierbaren, weil nicht im Modell repräsentierten, Beziehungen untereinander. Diese Elemente mögen je nach JAXR-Implementierung zwar spezifikationstreu nachgebildet sein, ihre Darstellung auf einer einheitlichen Ebene ist jedoch nicht sichergestellt.

Hier gibt die JAXR-Spezifikation auch keine einheitliche Trennung vor, sodass eine Interpretation der unterschiedlichen Ebenen dem Design der JAXR-Implementierung zufällt. Eine Gewinnung der benötigten Informationen aus dem Modell ist daher über einheitliche Mittel nicht sichergestellt und muss implementierungsabhängig betrachtet werden. Eine grundsätzliche Vorgehensweise ist durch diese Arbeit jedoch gegeben.

Der *dynamische* Teil besteht aus den im Modell angelegten oder erweiterten `RegistryEntries` (vgl. User Defined Objects), `Slots`, `Classifications` und `Associations` und ihrer Abbildung in UML.

3.1.1 Informationsreduktion

Da eine teilweise Abbildung des Modells einen nicht geringen logischen Aufwand nach sich ziehen würde, soll das Modell immer als Ganzes exportiert werden, daraus folgt bei komplexen Architekturen ein umso komplexeres UML-Modell.
Das entstehende Modell verzichtet zusätzlich, bis auf wenige Ausnahmen, auf die Verwendung von Packages zur Organisation der Klassen. Da es im JAXR-Datenmodell keine adäquate Form von Sichten gibt, werden alle abgebildeten Typen als Teil des *Root Packages* abgebildet. Zwar besteht die Möglichkeit, `RegistryObject`-Instanzen bestimmten `RegistryPackages` zuzuordnen, die Tatsache das eine Instanz jedoch beliebig vielen `RegistryPackages` zugeordnet werden kann, widerspricht dem UML-Verständnis von Packages, in dem jede Klasse nur einem Package angehören kann. [UML07] Darüber hinaus sollen nicht Instanzen sondern Typen geordnet werden.

Es erfolgt demnach keine Beschränkung auf eine Menge von Typen, die im neu entstehenden Modell abgebildet werden sollen. Dem Benutzer obliegt es also selbst, welche Klassen des UML-Root-Packages er in Form eines UML-Diagramms darstellen möchte. Bei komplexen Umgebungen führt diese Problematik schnell zum Verlust der Übersicht und zu einer nicht mehr zu überschauenden Menge von Klassen und Beziehungen im Modell.

Die Möglichkeiten der UML verführen an dieser Stelle dazu, diverse exotische Notationen einzuführen, um verschiedene Ebenen voneinander abzugrenzen. Diese Variante der Abbildung würde auf der einen Seite die Logik innerhalb des Modells stärken, auf der

anderen aber dazu führen, dass für den Betrachter ebenso tief gehende UML-Kenntnisse nötig wären.

Im Abbildungsprozess musste also ein Mittel aus Übersichtlichkeit, Verständlichkeit und Vollständigkeit gefunden werden.

3.2 Abbildung

Die Menge der zu betrachtenden Konzepte besteht aus den in der JAXR-Spezifikation *spezifizierten Elementen*, den *User Defined Objects* und den *angelegten Beziehungen und Slots* der Typen im Modell.

Die vom Benutzer durchgeführten Erweiterungen im Modell (User Defined Objects, Beziehungen und Typen) werden in jedem Fall dynamisch generiert, in manchen Fällen (siehe *Typen aus dem JAXR-Datenmodell* 3.2.1) können auch JAXR-Typen aus dem Modell generiert werden. Diese Ausnahme ergibt sich aus den unterschiedlichen Modellierungsebenen der JAXR-Spezifikation. So findet sich das `RegistryEntry`-Objekt auch in der vorliegenden JAXR-Implementierung wieder. Andere Typen wiederum lassen sich nicht aus dem Modell herauslesen, da sie auf einer anderen Ebene verstanden und nicht als Typ in der Registry abgelegt worden sind.

Der dynamische Teil besteht hier aus dem Extrahieren des `RegistryEntry`-Objekts und dem statischen Hinzufügen seiner spezifizierten aber möglicherweise nicht modellierten Beziehungen zu anderen Typen.

Die statische Modellierung von JAXR-Typen wurde gewählt, um den Betrachtern des Modells, die mit dem JAXR-Datenmodell vertraut sind, einen einfacheren Einstieg zu ermöglichen und Zusammenhänge zu verdeutlichen, auch wenn diese tatsächlich in der JAXR-Implementierung nicht auf diese Weise repräsentiert werden.
Damit verbunden ist die Übernahme der in der JAXR-Spezifikation modellierten Assoziationen zwischen Typen (z. B. `RegistryObject`, `ClassificationScheme`, etc.) auch wenn diese logisch nicht auf einer Ebene mit benutzerdefinierten Assoziationen sind.

3.2.1 Typen aus dem JAXR-Datenmodell

Statische Typen können zum Teil nicht aus der vorliegenden Registry heraus gelesen werden, da sie zwar gemäß Spezifikation implementiert sind, jedoch nicht auf Typebene abgebildet werden.
Diesem Umstand geschuldet muss eine implementierungsabhängige Interpretation der vorliegenden Typen stattfinden. Möglicherweise wird dadurch das manuelle (statische) Hinzufügen der Typen zum Modell notwendig, also die Modellierung durch statischen Code.
Des Weiteren ist es möglich, dass Typen in der Registry zwar erscheinen, jedoch nicht über die in der Spezifikation festgelegten Assoziationen und Vererbungen verfügen. Diese Elemente müssen ebenfalls statisch hinzugefügt werden.

Statisch wird in dieser Arbeit als Synonym für gleich bleibende, weil definierte, Elemente in einem dynamisch generierten Modell verwendet.

Statische Typen sind:

`RegistryEntry*`	Elternelement aller User Defined Objects u.a.
`RegistryObject*`	Elternelement aller Typen
`User`	
`AuditableEvent`	*entfällt*
`Organization`	
`Service`	
`ServiceBinding*`	
`SpecificationLink*`	
`Concept`	
`PostalAdress`	*entfällt, modelliert als Attribut von* `Organization`
`Classification`	*als Klassifikation dargestellt*
`Association`	*als Assoziation dargestellt*
`ExternalLink`	
`ExternalIdentifier`	*entfällt, modelliert als Attribut von* `RegistryObject`
`RegistryPackage`	Zuordnung über Assoziation

Tabelle 3.1: Durch das JAXR Datenmodell definierte Typen

Mit * gekennzeichnete Typen müssen statisch angegeben werden, da sie nicht oder nicht reproduzierbar implementiert sind.

Im Folgenden werden nun Konzepte zur Abbildung von JAXR auf UML entwickelt. Neben Fragen der Realisierbarkeit und Übersichtlichkeit, soll auch die semantische Gleichheit zwischen JAXR-Elementen und UML-Elementen sichergestellt und überprüft werden.

3.2.2 Objekttyp - Klasse

Definition UML-Klasse Laut [UML07] ist es die Aufgabe einer Klasse, eine Menge von Objekten und deren gemeinsame, charakteristische Eigenschaften zu klassifizieren.

Die UML-Klasse als zentrales Modell der UML findet bei der Abbildung des JAXR-Datenmodells mehrfach Verwendung:

- als Repräsentation von `RegistryObject` und seinen Subklassen.
- als Datentyp (siehe 3.2.3).
- als Taxonomie (siehe 3.2.6).

Die unterschiedliche Verwendung der UML-Klasse ist durch die vollzogenen Gliederungen in Packages zu unterscheiden. Datentypen werden im Package `BasicDatatypes` und Taxonomien im Paket `Taxonomy` abgelegt. Dazu später mehr.

Da es sich bei einem `RegistryObject` um ein Element mit Eigenschaften, Beziehungen und Attributen (Slots) handelt, bietet sich eine Abbildung auf UML-Klassen an. JAXR-Typen werden also als Klassen modelliert.

Zentraler Bestandteil dieses Abbildungsschritts ist das Auffinden von Typen im Modell. Weiterhin die Identifikation von Assoziationen, Slots und Klassifikationen und die Ermittlung von statisch zu modellierenden Typen (siehe *Typen aus dem JAXR-Datenmodell* 3.2.1). Die ermittelten Elemente eines Typs führen zur Anwendung der nachfolgenden Abbildungen.

3.2.3 Slots - Attribute

Um den die `Slots` eines `RegistryObjects` abzubilden, werden die gängigsten Basisdatentypen (Strings, Integer, etc.) und spezifische Datentypen (Adresstypen usw.) als Klassen im Paket `BasicDatatypes` modelliert.

Die Abbildung der Slots auf Attribute erfolgt nun vollständig durch die Übernahme des Namens, der Datentypen und ggf. der Transformation der Kardinalitätsnotationen.

3.2.4 Association - Assoziation

Definition UML-Assoziation Eine Assoziation besteht in der UML aus einer Menge von Tupeln, durch deren Werte Elemente referenziert werden. [UML07]

Assoziationen haben im JAXR-Modell ein `sourceObject` und ein `targetObject`. Das `sourceObject` kann dabei dem `targetObject` entsprechen. Im Gegensatz zu einer UML-Assoziation ist es bei einer JAXR-Assoziation nicht möglich, mehr als zwei Assoziationsenden (n-äre Assoziation) zu modellieren. [SUN02]

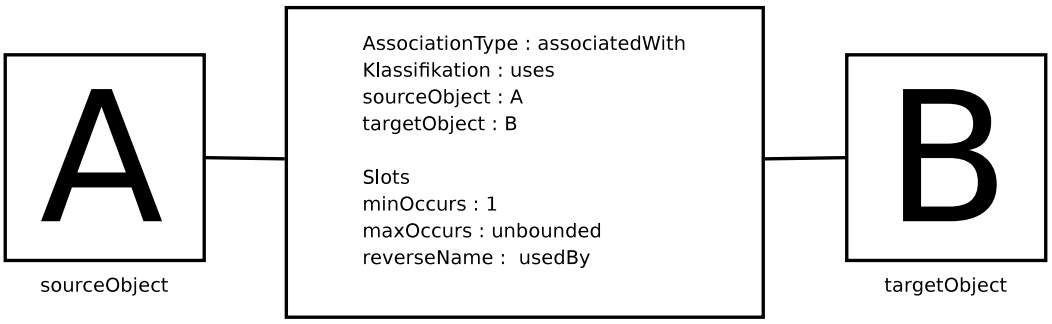

Abbildung 3.1: Die Assoziation im JAXR-Modell

Aus dem CentraSite-Modell sollen nur solche Assoziationen in das UML-Modell übernommen werden, die den AssociationType `associatedWith` haben.

Die Unterscheidung des AssociationType wird hier nötig, da ansonsten auch Klassifikationsbeziehungen berücksichtigt werden müssten, die in der Registry mit einer ähnlichen Syntax definiert sind.
Durch die Slots `minOccurs` und `maxOccurs` der Assoziation werden die Kardinalitäten spezifiziert, falls dies nicht der Fall sein sollte, findet keine Übernahme der Kardinalitäten statt (siehe Abbildung 3.1). Um der von JAXR propagierten Form von Source und Target treu zu bleiben, werden Assoziationen gerichtet dargestellt. Sämtliche Assoziationen des JAXR-Datenmodells werden also als gerichtete Assoziationen abgebildet und sind demnach nicht beidseitig navigierbar.
Eine Ausnahme bilden hier die in der JAXR-Spezifikation definierten Assoziationen, bei denen zum Teil keine Navigationsrichtung definiert wurde.

Laut [R+07] muss bei einer einseitigen Navigationsrichtung das nicht-navigierbare Ende der Assoziation durch ein "×" angezeigt werden oder für das gesamte Modell generell für nicht navigierbar erklärt werden. Das sei hiermit getan. Der Rollenname der

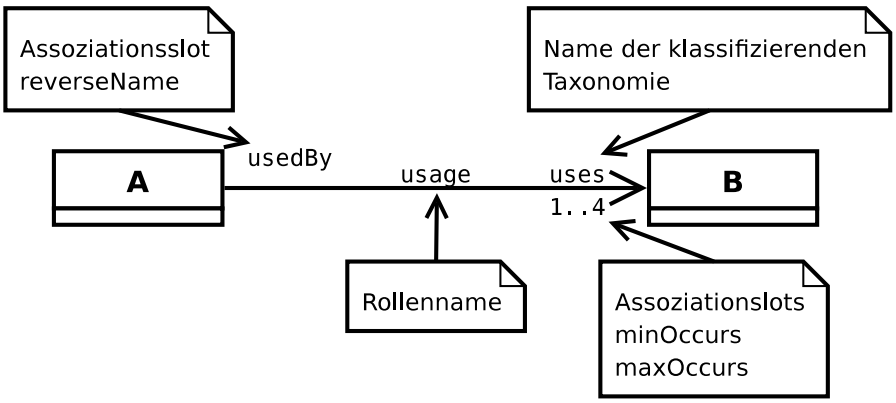

Abbildung 3.2: Abbildung JAXR-Assoziation auf UML

Assoziation wird durch die Klassifikation der Assoziation, dem Konzept aus der Taxonomie `AssociationType`, definiert, z. B. "`uses`". In CentraSite ist es möglich, durch den Klassifikationsslot `reverseName` den Namen des anderen Endes der Assoziation festzulegen, z. B. "`usedBy`". [Sch07]

Um diesen komplexen Vorgang zu verdeutlichen, wird dieser Abbildungsschritt in der Abbildung 3.2 durchgeführt:

Aufgrund der einseitigen Navigierbarkeit bekommt nur die Klasse "A" Kardinalitäten zugewiesen. `uses`, übernommen von der Klassifikation der Assoziation, und `usedBy`, definiert durch `reverseName`, bezeichnen die jeweiligen Rollen in der Assoziation.

3.2.5 Assoziationsslots - Assoziationsklasse

Definition UML-Assoziationsklasse Die Assoziationsklasse ist in der UML ein Element, welches sowohl Eigenschaften einer Klasse als auch einer Assoziation aufweist. Eine Assoziationsklasse kann als eine Assoziation mit den Eigenschaften einer Klasse, sowie als Klasse mit den Eigenschaften einer Assoziation gesehen werden.
Die Assoziationsklasse als Beziehung zwischen mehreren Klassen definiert eine Menge von Eigenschaften, die nicht Teil der miteinander in Beziehung stehenden Klassen sind. [UML07]

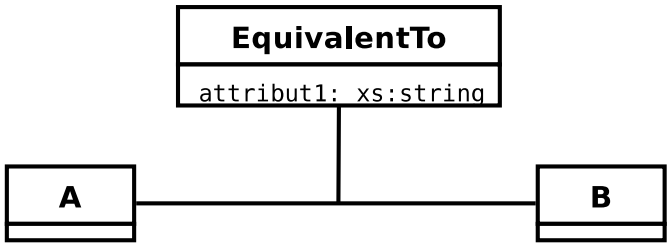

Abbildung 3.3: Assoziationsklasse in UML

Im JAXR-Datenmodell ist es möglich nahezu allen Elementen Slots hinzuzufügen, dadurch können auch Assoziationen um Slots erweitert werden. Dabei entsteht ein der UML-Asssoziationsklasse ähnliches Konstrukt, das dazu benutzt werden kann, bestimmte Eigenschaften der Assoziation auszudrücken.
Die JAXR-Assoziationsklasse wird eher als Assoziation mit den Eigenschaften (Slots) eines Typs modelliert. [UML07] Demgegenüber wird die UML-Assoziationsklasse als Klasse mit den Eigenschaften einer Assoziation modelliert. Über das Klassifikationskonzept `AssociationType` wird, wie bei der normalen Assoziation, der Typ der Assoziation festgelegt (hier "equivalentTo"). Die Übernahme der Rollennamen findet ebenso statt. Der Unterschied zur bereits behandelten normalen Assoziation besteht dabei nur im

Hinzufügen von Slots, dieser Vorgang ist aus der Modellierung der `RegistryObjects` bekannt und syntaktisch gleich.

3.2.6 Klassifikationen

Einzelne Typen können mit ein oder mehreren Taxonomien klassifiziert werden. Klassifikationen werden durch die JAXR-Assoziation mit dem Klassifikationsschema `classifiedBy` realisiert. Dabei ist es nicht unüblich, dass ein Typ drei oder mehr Klassifikationen aufweist.

Die in *Informationsreduktion* 3.1.1 erwähnte Rücksichtsnahme auf die Übersichtlichkeit schlägt sich am ehesten bei der Darstellung der Klassifikationen nieder. Oft werden schon lose Verbindungen in einem geschäftlichen Gebiet als Klassifikation dargestellt, was zu einer großen Anzahl von Klassifikationen im Modell führt.

Da es in der UML kein gleichwertiges Element zur JAXR-Klassifikation gibt, werden im Folgenden drei verschiedene Ansätze zur Darstellung und logischen Abgrenzung gegenüber einer normalen Assoziation vorgestellt.

Als Stereotyp

Definition UML-Stereotyp Ein Stereotyp definiert, wie eine existierende Klasse erweitert werden kann und erweitert sie um plattform- bzw. domänenspezifische Bedeutungen oder Notationen. [UML07]

Mit *Stereotypen* lassen sich UML-Elemente und ihre Bedeutung im System einfach veranschaulichen. Da Stereotypen bezeichnen und einordnen, bietet sich die Modellierung einer Klassifikation als Stereotyp an. [Pil03]

Abbildung 3.4: Klassifikationen als Stereotypen

Entgegen älterer UML-Versionen ist es in UML 2.1 möglich, ein Element mit mehreren Stereotypen auszuzeichnen [R+07].Diese multiple Auszeichnung eines Elements mit Stereotypen kommt der Mehrfachklassifizierung von JAXR-Typen sehr nahe. Allerdings sind diese Mehrfachstereotypen sehr exotisch und ihre Unterstützung selbst in führenden Modellierungswerkzeugen nicht gewährleistet.

Zur Realisierung der unterschiedlichen Stereotypen würde man hier in einem UML-Profil[1] mehrere Stereotypen für die Menge der verwendbaren Klassifikationen anlegen.

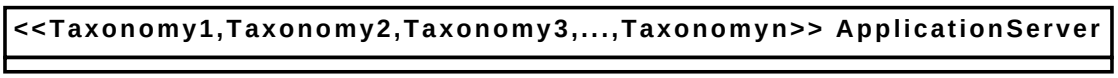

Abbildung 3.5: Klassifikationen als Stereotypen im Klassennamen

Da Profile zwar in XMI (siehe Kapitel *4.1 XMI*) abgebildet werden können, aber nicht von allen Modellierungswerkzeugen gleich interpretiert werden, empfiehlt es sich, Stereotypen als Teil des Klassennamens auszuschreiben. Durch den fehlenden Umbruch im Namen entstehen dabei jedoch zwangsläufig in die Breite gezogene Klassenelemente.

Trotz alledem handelt es sich bei der Abbildung von Klassifikationen durch Stereotypen um den logischsten und vielleicht saubersten Weg, Klassifikationen in der UML abzubilden.
Die fragwürdige Unterstützung auf der einen und die in die Breite gezogenen Klassennamen auf der anderen Seite relativieren jedoch diese Vorteile.

Als Assoziation

Der nächste Ansatz orientiert sich an der Idee zur Darstellung von Klassifikationen als Assoziationen mit besonderer Rollen-Notation. Da aber im vorliegenden Modell Assoziationen mit beliebigen Rollennamen definiert werden können, ist eine visuelle Trennung von Assoziation und Klassifikation schwierig.

[1] UML-Profil: In der UML ist ein Profil eine Gruppierung für Stereotypen ähnlich einem Paket

Zur Abbildung wird im Modell die Klasse `Taxonomy` eingeführt. Sämtliche Klassifikationen haben als Ursprung die zu klassifizierende Klasse und als Ziel die Klasse `Taxonomy`. Die Art der Klassifizierung wird über den Rollennamen der Assoziation angegeben.

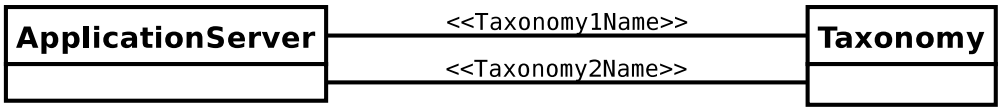

Abbildung 3.6: Klassifikation als Assoziation

Es wird also für jede Klassifizierung eine eigene Assoziation angelegt. Ein Typ mit 10 Klassifikationen hätte also 10 Assoziationen zu der Klasse `Taxonomy`. Durch die Anzahl der Assoziationen und ihrer zum Teil wilden Wegfindung durch das Modell entsteht ein hohes Maß an Unübersichtlichkeit.

Da nun ein großer Teil der UML-Assoziationen im Modell von einer unbestimmten Menge von Klassen zu genau einer Klasse gezogen werden, ist eine Unterscheidung aufgrund des Rollennamens nahezu unmöglich. Bereits bei eher simplen Modellen sticht die mangelnde Übersicht ins Auge. In der Folge wird die Abbildung von Klassifikationen mittels Assoziationen verworfen.

Als Datentyp mit Stereotypnotation

Bei dieser Lösung wird die Idee des Stereotypen erneut aufgegriffen und wie zuvor werden die Taxonomien als Klasse modelliert.

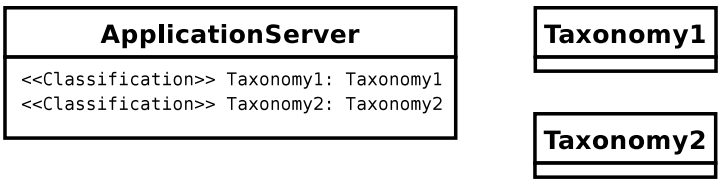

Abbildung 3.7: Klassifikation als Datentyp

Bei dieser Variante ist die eindeutige Namensvergabe sehr wichtig. Zwar ist es möglich der Klassifikation keinen zusätzlichen Namen zuzuweisen (`name1`), was aber dazu führen kann, dass Attribute mit dem Namen "«`Classification`» " mehrfach auftreten und so nicht voneinander unterschieden werden können.

Inkonsequenterweise lassen einige Modellierungswerkzeuge die Doppelverwendung von Attributbezeichnern zu. Dadurch auftretende Inkompatibilität, wird mit dem oben genannten Ansatz behoben. Leider kommt es durch diesen Lösungsansatz bei der UML-Darstellung "Name : Datentyp" zur unglücklichen doppelten Erwähnung des Datentypnamens. Diese Redundanz soll zur Gewährleistung der Eindeutigkeit jedoch zugelassen werden.

Für jede Taxonomie im Modell wird im Paket *Taxonomy* eine eigene Klasse angelegt. Auf der Seite des zu klassifizierenden Typs wird ein Attribut mit dem Stereotyp "«Classification» name1" und einem beliebigen Namen angelegt. Der Datentyp des in der Ausgangsklasse angelegten Attributs bestimmt die Klassifikation. `Name` steht für einen (beliebigen) Namen der Klassifikation. Folglich würde eine Klassifikation durch die Taxonomie *Product* "«Classification» Product : Product" lauten.

Zusammenfassung Abbildung der Klassifikation

- Als Stereotyp
  - *Keine einheitliche Interpretation durch Modellierungswerkzeuge.*
  - *In die Länge gezogene Klassennamen.*
  - Semantisch und logisch sauberste Abbildung.
- Als Assoziation
  - *Sehr unübersichtlich.*
  - Rollennamen drücken Klassifikation aus.
- Als Datentyp mit Stereotypnotation
  - Durch Verwendung der Stereotypnotation ähnliche Vorteile wie beim Stereotyp-Ansatz.
  - Name und Datentyp sind redundant.

Aufgrund der jeweiligen starken Nachteile der Ansätze ”Stereotyp“ und ”Assoziation“ (kursiv) wurde die Abbildung der Klassifikation als Datentyp mit Stereotypnotation gewählt.

Kapitel 4

Serialisierung des UML-Modells

4.1 XML Metamodel Interchange (XMI)

XMI ist ein XML Dialekt, mit dem es möglich ist, verschiedene *Metamodelle*, die zum *Meta-Object Facility* (MOF) Standard kompatibel sind, auszutauschen. Der auf der MOF aufbauende UML-Standard kann so serialisiert und in Textform verarbeitet werden. Damit ist es mittels XMI möglich, UML-Modelle standardisiert zwischen verschiedenen Modellierungswerkzeugen oder anderen Programmen auszutauschen. [Kov02] Zur Definition der XML-Datenstrukturen stehen dafür die *Document Type Definition* (DTD) und das mächtigere *XML-Schema* zur Verfügung. [Wol05]

XMI wurde von der OMG als Antwort auf den Wunsch nach einem textuellen Standard für die Darstellung von objektorientierten Modellen auf allen MOF-Metaebenen (siehe 4.3) entwickelt. Seit der ersten Definition des Standards im Jahr 1999 gab es 4 weitere Versionen: XMI 1.1, die *XML Namespaces* unterstützt, XMI 2.0 mit einer besseren Unterstützung von *XML Schema* und *Document Type Definitions* (DTD) und XMI 2.1 mit der Möglichkeit zur Definition von UML-Diagrammen.

Bei der Darstellung von UML verwendet XMI spezielle Tags, um übliche Elemente der UML, wie Klassen, Attribute, Methoden, Datentypen, etc., in XML abzubilden. Dazu kommen die Darstellungen von Beziehungen, wie Assoziationen, Vererbungen oder Stereotypen etc., durch XML-Elemente.

Da XMI das von der OMG eigens entwickelte Werkzeug zur Serialisierung von MOF-Modellen (und damit UML) ist, wird das JAXR-Datenmodell gemäß der erarbeiteten Abbildung in XMI exportiert.

Als Standard legt XMI dabei die Regeln fest, wie aus einem MOF-Modell eine DTD oder XML-Schema erstellt werden kann. Durch dieses XMI-Schema bzw. XMI-DTD ergibt sich die Möglichkeit, ein bestehendes XMI-Modell gegen das Schema bzw. die DTD zu validieren. [Wol05]

4.2 Meta-Object Facility (MOF)

MOF ist, wie die UML, ein Standard der *Object Management Group* (OMG). Bei der Entwicklung der UML wurde ein mächtiges Werkzeug zur Beschreibung von Metamodellen notwendig. Aus diesem Grund entwickelte die OMG zusätzlich die Meta-Object Facility (dt. etwa Meta-Objekt Einrichtung).

Die Meta-Object Facility definiert verschiedene Konzepte: Packages, Klassen, Methoden, Attribute, Operationen, Referenzen, Constraints, Assoziationen und Datentypen. Diese Konzepte sind mithilfe von UML-Diagrammen beschrieben.
Aktuell liegt die MOF in der Version 2.0 vor und untergliedert sich in zwei Varianten: *Essential MOF* (EMOF) und *Complete MOF* (CMOF).

EMOF ermöglicht in vereinfachter Form die Abbildung von MOF Modellen nach XMI, also die *Essentials* (dt. Grundlagen), wohingegen CMOF die komplette Neudefinition von Metamodellen ermöglicht, demnach *complete* ist.

Zusammenfassend lässt sich sagen: MOF beschreibt ein Metamodell und ermöglicht dessen Abbildung in andere zur MOF-kompatible Modelle. [Gop06]

4.2.1 Metamodelle

Bei *Metamodellen* handelt es sich um selbst beschreibende Modelle, d. h. ein Modell, das mit seinen eigenen Elementen beschrieben werden kann.

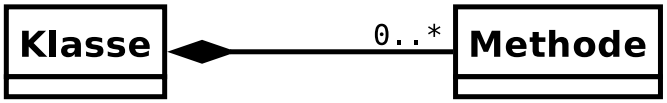

Abbildung 4.1: Beziehung Klasse - Methode auf Ebene M3 der MOF-Architektur

Im Falle der UML würde das Hinzufügen von Methoden zu einer Klasse, wie in Abbildung 4.1 dargestellt, mit Elementen der UML ausgedrückt werden. Die UML beschreibt sich also selbst, indem sie das Hinzufügen von Methoden durch eine n-fache Komposition zwischen der Klasse "Klasse" und der Klasse "Methode" modelliert.
Metamodelle sind also eine Menge von Konzepten, die ein Vokabular bereitstellen, um über eine bestimmte Wissensdomäne (inkl. ihrer selbst) zu diskutieren. [Bra03]

4.3 MOF-Architektur

Die MOF-Architektur besteht aus 4 Schichten. Die unterste, im Sinne des Abstraktionsgrades, Schicht (M0) stellt die reale Welt dar, die oberste Schicht (M3) ein Meta-Metamodell. Letzteres weist folglich den höchsten Abstraktionsgrad auf. Auf Schicht M2 befindet sich das konzeptionelle Modell, also das zu beschreibende Metamodell an sich (hier UML).
Selbst modellierte Modelle werden auf Schicht M1 repräsentiert. Ein auf Schicht M1 modelliertes Modell entspräche der Abbildung von JAXR auf UML. Siehe Abbildung 4.2. [Uck07]

Abbildung 4.2 auf der folgenden Seite dient der Verdeutlichung der verschiedenen Ebenen der MOF-Architektur. Auf der linken Seite sind die MOF-Ebenen und ihre Repräsentation in UML zu sehen, auf der rechten Seite das XML/XMI-Gegenstück. Die jeweiligen XML-Fragmente sind sinnvoll abgekürzt.

Die Beziehungen zwischen den Abstraktionsebenen sind folgendermaßen zu verstehen: Abstraktionsgrad$_{n-1}$ ist eine Instanz von Abstraktionsgrad$_n$.

Als Teil dieser Arbeit sollen Veränderungen auf der Schicht *M1 - UML-Klasse* abgebildet werden, darunter fällt z. B. die Verarbeitung von neu erstellten Typen im JAXR-Datenmodell zu Klassen.
Nicht abgebildet wird die Schicht M2, in welcher MOF-Metamodelle als XMI-Schema beschrieben werden. Schicht M0, also die konkreten Ausprägungen der Typen im JAXR-Datenmodell, wird ebenfalls nicht modelliert. [Jec99]

Abbildung 4.2: Schichten der MOF-Architektur

4.4 Erstellung eines XMI-Modells

Mit der XML Metadata Interchange (XMI) Spezifikation werden MOF-konforme Modelle in eine serialisierte Form (XML) gebracht. XMI bietet dabei Methoden an, um MOF zu serialisieren. Ergebnis dieser Transformation ist ein XMI-Dokument. Der Aufbau eines

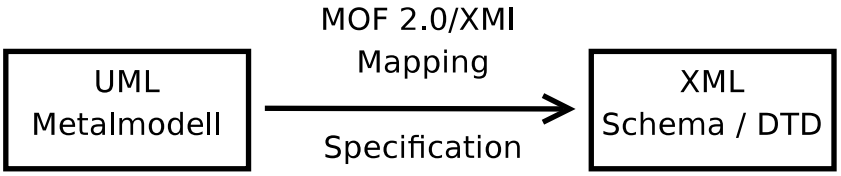

Abbildung 4.3: Mapping von UML auf ein XML-Schema / DTD

XMI-Modells ist in der *MOF 2.0/XMI Mapping Specification* beschrieben. [OMG05]
Da der UML ein MOF-basiertes Metamodell zu Grunde liegt und die UML-Spezifikation sich der Hilfsmittel der MOF bedient, werden Bezeichner und Elementenamen aus der UML-Spezifikation verwendet, um sie im XMI-Kontext wiedererkennbar einzusetzen. Darunter fallen beispielsweise solche UML-Konstrukte wie `packagedElement` als Element. Die Eigenschaften `isVisible` und `isPublic` als Attribute.

4.5 Elemente der UML-Spezifikation in XMI

packagedElement definiert alle Elemente, die direkt einem Paket zugeordnet werden können, z.B. Klassen.

ownedAttribute repräsentiert ein Attribut oder ein Assoziationsende bei gerichteten Assoziationen.

memberEnd repräsentiert die Teilnahme des übergeordneten Elements an einer Assoziation (bei ungerichteten Assoziationen, eigenständiges Assoziationselement).

generalization spezifiziert die Basisklasse dieser Klasse.

Hinzu kommen die Attribute zur Definition der Eigenschaften eines Elements, z.B. Sichtbarkeit.

4.5.1 Anwendung von XMI

Laut Spezifikation ist es möglich, Elemente des MOF Modells im XML-Schema mit dem Klassennamen zu benennen, sodass im XMI-Dokuments Elemente in der Art von `<Klassenname1>` auftauchen. Eine weitere Möglichkeit ist die Spezifizierung des Namens über XML-Attribute wie `name="Klassenname1"`.
Hier wird dem Modellierer ein erheblicher Grad an Freiheit beigemessen, was wiederum beim Studium verschiedener XMI-Darstellungen zu Verwirrungen führen kann. Gängige Modellierungswerkzeuge wählen die letztgenannte Variante in Kombination mit dem Element `<packagedElement>` mit der Definition von `xmi:type` als `uml:class`. [OMG05]

Eine grundsätzliche Struktur eines XMI-Dokuments könnte folgendermaßen aussehen:

XMI	Angabe der Versionsnummer, enthält *Documentation* und *Extension*.
Documentation	Enthält Informationen über den XMI-Exporter, die XMI-Exporterversion und weitere Daten über den Ersteller des Dokuments.
Modell	Das Modell macht den mit Abstand größten Teil des Dokuments aus.
Extension	Bietet Exportern die Möglichkeit, eigene Zusätze dem XMI-Dokument hinzuzufügen.

Einschub JAXR und MOF Zum grundsätzlichen Verständnis dieses Kapitels, trägt an dieser Stelle die Beantwortung der Frage bei, warum bei der Abbildung von JAXR nach UML auf die Möglichkeiten der Meta Object Facility verzichtet wurde.

Zur Abbildung des JAXR nach UML wäre die Verwendung von MOF ungeeignet gewesen, da dem JAXR-Datenmodell kein MOF-basiertes Modell zu Grunde liegt.
Zwar werden grundsätzliche Zusammenhänge in der Spezifikation mittels UML dargestellt, eine gründliche Spezifizierung von MOF-Komponenten findet jedoch nicht statt. Es kann davon ausgegangen werden, dass bei der Spezifikation von JAXR keine Rücksicht auf Meta Object Facilites genommen wurde und eine erzwungene Abbildung daher unpraktikabel wäre.

4.6 Basiselemente der UML in XMI 2.1

Im Folgenden werden die zur Darstellung von Klassen, Attributen, Assoziationen und Vererbungen notwendigen Elemente der UML und ihrer Abbildung in XMI vorgestellt.

Mit der Version 2.1 bietet XMI die Möglichkeit, Diagramme einzubetten. Da die Visualisierung der Klassen in Klassendiagrammen jedoch individuell geschehen soll, wird von einer Erstellung von Diagrammen zur Laufzeit abgesehen.

In den folgenden Beispielen wurden die Identifikationsnummern (ID) auf ein lesbares Format reduziert. In der Praxis werden längere und eindeutigere IDs verwendet.

Zur Erstellung der XML-Modelle wurde die Ausgabe des *Enterprise Architect* XMI-Exports als Referenz genommen, um eine vollständige Kompatibilität mit diesem UML-Werkzeug zu erreichen.

An denen mit `[...]` gekennzeichneten Stellen wurden für die Beispiele unwesentliche Informationen weggelassen, um die Übersichtlichkeit zu wahren.

4.6.1 UML-Modell

Ein UML-Modell wird durch das Element `<uml:Model>` definiert. Der Vollständigkeit halber soll hier auch der XMI-Kopf aufgeführt werden. Das wichtigste Attribut des Elements `<xmi:XMI>` ist die Versionsnummer (hier 2.1), da XMI keine Abwärtskompatibilität aufweist.

Beispiel 4.1: UML-Modell in XMI

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.1"
      xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
3   <xmi:Documentation exporter="Matthias Kuech CentraSite XMI 2.1 Exporter"
        exporterVersion="0.1"/>
4   <uml:Model name="CS_Model" visibility="public" xmi:id="CS:UML:Model" xmi:
        type="uml:Model">
5   ...
6   </uml:Model>
7 </xmi:XMI>
```

4.6.2 Package

UML-Modelle sind durch *Packages* zur besseren Organisation und Verwaltung in Pakete gegliedert. Jedenfalls wird ein Basispaket (Rootpackage) benötigt, hier `CS:Package`. CS steht in diesem und weiteren Zusammenhängen für CentraSite. Die Definition der Sichtbarkeit (visibility) erfolgt für alle Elemente des XMI-Dokuments mit ”public“.

Beispiel 4.2: Package in XMI

```
1 <packagedElement xmi:type="uml:Package" xmi:id="CS:Package" name="PackageName
      " visibility="public">
2   ...
3 </packagedElement>
```

4.6.3 Klasse

Klassen werden mit dem Type `uml:Class` angegeben. Klassen können Elemente zur Definition von Vererbungen (siehe 4.6.5) und Assoziationen (siehe 4.6.6) enthalten.

Abbildung 4.4: Notation UML-Klasse

Beispiel 4.3: UML-Klasse in XMI

```
1 <packagedElement xmi:type="uml:Class" xmi:id="idKlasse1" name="Klasse1"
      visibility="public">
2   <ownedAttribute xmi:type="uml:Property" xmi:id="idVar1" name="var1"
        visibility="public" isDerived="false">
3     <lowerValue xmi:type="uml:LiteralInteger" xmi:id="idVar1Lower" value="1"/
          >
4     <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="idVar1Upper"
          value="-1"/>
5   </ownedAttribute>
6 </packagedElement>
```

Das Attribut `var1` hat keinen Datentyp und wird so als datentyplos interpretiert.
Die Elemente `<lowerValue>` und `<upperValue>` spezifizieren die Kardinalität des Attributs. Die Value "-1" steht für unendlich, zulässige Werte sind 0 bis n. Wobei die `lowerValue` nicht größer als die `upperValue` sein darf.
`xmi:type` spezifiziert den Datentyp des Wertes. Neben dem Wert `uml:LiteralInteger` für natürliche Zahlen, wird `uml:LiteralUnlimitedNatural` als Darstellung der Unbegrenztheit gewählt.

4.6.4 Attribute

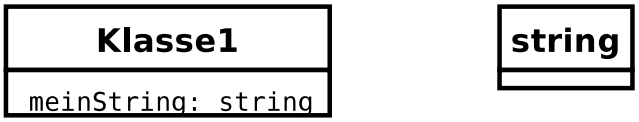

Abbildung 4.5: Notation UML-Attribute

Zur Darstellung von Datentypen wird eine Referenzierung des Datentyps benötigt. Da Modellierungswerkzeuge über keine standardisierte Datentypbibliothek verfügen, ist es für das Programm nicht ersichtlich, ob es sich bei dem Datentyp "string" um den Java-Datentyp "String" oder um den XML-Typ "xs:string" handelt. Sinnvollerweise sollen daher alle Datentypen als eigenständige Klasse modelliert werden.

Beispiel 4.4: UML- Attribute in XMI

```
1 <packagedElement xmi:type="uml:Class" xmi:id="idString" name="string"
      visibility="public"/>
2 <packagedElement xmi:type="uml:Class" xmi:id="idKlasse1" name="Klasse1"
      visibility="public">
3     <ownedAttribute xmi:type="uml:Property" xmi:id="idMeinString" name="
          meinString" visibility="private" isDerived="false">
4       <type xmi:idref="idString"/>
5       <lowerValue xmi:type="uml:LiteralInteger" xmi:id="idMeinStringLower"
            value="1"/>
6       <upperValue xmi:type="uml:LiteralInteger" xmi:id="
            idMeinStringLowerUpper" value="1"/>
7     </ownedAttribute>
8 </packagedElement>
```

Die Klasse `Klasse1` hat ein Attribut mit dem Namen `meinString` vom Typ `string`, welches über die ID `idString` mit der Klasse `string` referenziert wird (xmi:idref).

4.6.5 Vererbung

Mit dem Element `<generalization>` kann eine Vererbungsbeziehung zu einer Superklasse definiert werden. In diesem Beispiel wird die Vererbung zwischen Klassen betrachtet.

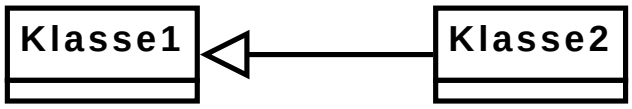

Abbildung 4.6: Notation UML-Vererbung

Beispiel 4.5: UML-Vererbung in XMI

```
1 <packagedElement xmi:type="uml:Class" xmi:id="idKlasse1" name="Klasse1"
      visibility="public"/>
2
3 <packagedElement xmi:type="uml:Class" xmi:id="idKlasse2" name="Klasse2"
      visibility="public">
4     <generalization xmi:type="uml:Generalization" xmi:id="
          idKlasse2Generalisierung" general="idKlasse1"/>
5 </packagedElement>
```

Das Element `<generalization>` verweist mit dem Attribut `general` auf die Elternklasse.

4.6.6 Assoziation

Je nach Art der Assoziation kommt es zu unterschiedlichen Definitionen an unterschiedlichen Stellen des Dokuments.

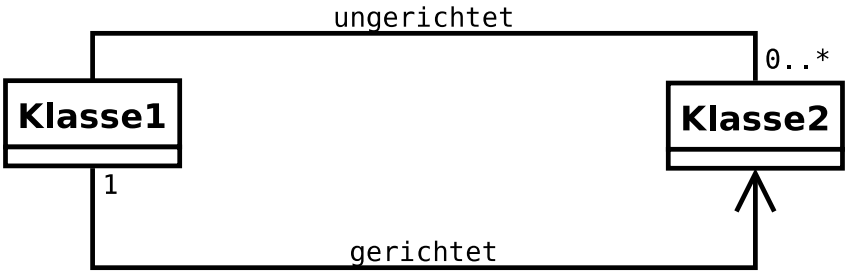

Abbildung 4.7: Notation gerichtete und ungerichtete Assoziation in UML

4.6.7 Gerichtete Assoziation

Beispiel 4.6: Gerichtete Assoziation der UML in XMI

```
1  <packagedElement xmi:type="uml:Class" xmi:id="idKlasse1" name="Klasse1"
       visibility="public">
2    <ownedAttribute xmi:type="uml:Property" xmi:id="idKlasse1Attribut"
         visibility="public" association="idGerichteteAssoziation" [...]>
3      <type xmi:idref="idKlasse2"/>
4    </ownedAttribute>
5  </packagedElement>
6
7  <packagedElement xmi:type="uml:Class" xmi:id="idKlasse2" name="Klasse2"
       visibility="public"/>
8
9  <packagedElement xmi:type="uml:Association" xmi:id="idGerichteteAssoziation"
       name="gerichtet" visibility="public">er
10   <memberEnd xmi:idref="idKlasse1Attribut"/>
11   <memberEnd xmi:idref="idPropertyGerichteteAssoziation"/>
12   <ownedEnd xmi:type="uml:Property" xmi:id="idPropertyGerichteteAssoziation"
         visibility="public" association="idGerichteteAssoziation" [...]>
13     <type xmi:idref="idKlasse1"/>
14     <lowerValue xmi:type="uml:LiteralInteger" xmi:id="idKlasse1Lower" value="
           1"/>
15     <upperValue xmi:type="uml:LiteralInteger" xmi:id="idKlasse1Upper" value="
           1"/>
16   </ownedEnd>
17 </packagedElement>
```

Bei der gerichteten, also einseitig navigierbaren, Assoziation zwischen `Klasse1` und `Klasse2` wird `Klasse1` ein `ownedAttribut` mit dem Typ `Klasse2` und dem Attributwert "idGerichteteAssoziation" zugewiesen. Das `packagedElement` vom Typ `uml:Association` definiert für diese Assoziation nur noch die Kardinalität der navigierbaren Seite.

Da die Assoziation auf der Quellseite als Attribut modelliert ist, obliegt es dem XMI importierenden Programm, die Assoziation als Attribut, als Assoziation oder als Attribut und Assoziation darzustellen. Die UML-Spezifikation lässt hier alle Varianten zu. [Fow05]

4.6.8 Ungerichtete Assoziation

Das untere Beispiel bezieht sich mit seinen Referenzen auf die in 4.6.7 definierten Klassen.

Beispiel 4.7: Ungerichtete Assoziation der UML in XMI

```
1  <packagedElement xmi:type="uml:Association" xmi:id="idUngerichteteAssoziation
       " name="ungerichtet" visibility="public">
2    <memberEnd xmi:idref="idMemberEnd1"/>
3    <ownedEnd xmi:type="uml:Property" xmi:id="idMemberEnd1" visibility="public"
         association="idUngerichteteAssoziation" [...]>
4      <type xmi:idref="idKlasse2"/>
5      <lowerValue xmi:type="uml:LiteralInteger" xmi:id="idKlasse2Lower" value="
           0"/>
6      <upperValue xmi:type="uml:LiteralUnlimitedNatural" xmi:id="idKlasse2Upper
           " value="-1"/>
7    </ownedEnd>
8    <memberEnd xmi:idref="idMemberEnd2"/>
9    <ownedEnd xmi:type="uml:Property" xmi:id="idMemberEnd2" visibility="public"
         association="idUngerichteteAssoziation" [...]>
10     <type xmi:idref="idKlasse1"/>
11   </ownedEnd>
12 </packagedElement>
```

Die ungerichtete, also beidseitig navigierbare, Assoziation spezifiziert jeweils zwei `memberEnds` und `ownedEnds`. Auch die nur einseitige Definition der Kardinalitäten

wird durch das XMI-Dokument repräsentiert. Die Definition der ungerichteten Assoziation erfolgt in diesem Fall nur in einem `packagedElement` des Typs `uml:Association` und nicht in den beiden Klassen, die die beiden Enden der Beziehung darstellen.

Insbesondere im Hinblick auf eine dynamische Generierung der Assoziation ist die ungerichtete Assoziation einfacher abzubilden, da sie an jeder Stelle im Dokument eingefügt werden kann, und nicht parallel zur Generierung eines Klassenelements erkannt und erstellt werden muss.

4.7 Unterstützung durch Modellierungwerkzeuge

Das vorliegende XMI-Modell soll nun zur Überprüfung der Portabilität in verschiedene Modellierungswerkzeuge importiert werden. Da das XMI-Modell zwar alle Elemente des UML-Modells definiert, sie aber nicht in grafischer Form darstellt, muss ein Weg zur Visualisierung des Modells gefunden werden. Mit der Version 2.1 bietet XMI die Möglichkeit, Diagramme im XMI-Modell abzulegen und diese dann in eine SVG-Grafik[1] zu transformieren.
In diesem Teil soll jedoch der Möglichkeit nachgegangen werden, ein MOF-kompatibles Modell, hier UML, in Modellierungswerkzeuge zu importieren und dem Benutzer die Wahl der Darstellung zu überlassen. Dieser Schritt kann der Dokumentation oder der Evaluierung eines bestehenden Modells dienen.

UML-Modellierungswerkzeuge verhalten sich verschiedentlich beim Behandeln von XMI-Modellen. Nicht Wenige lagern erhebliche Teile des Modells in die Erweiterungs-Elemente des XMI-Modells aus, sodass die Kompatibilität der Programme untereinander stark sinkt. Ebenso drängt sich die, nicht belegbare, Vermutung auf, die Modellierungswerkzeuge könnten anhand der Informationen über den Ersteller des Modells unterschiedliche Interpretationsalgorithmen bemühen (vgl. *Anwendung von XMI 4.5.1* auf Seite 38).

Zur Belegung und Differenzierung dieser Kompatiblitätsprobleme soll das folgende Kapitel die Portabilität von Modellen unterschiedlicher Herkunft untersuchen.

4.7.1 CentraSite Export

Als Teil dieser Arbeit wurde *CentraSite Export* entwickelt, dass die Ergebnisse der behandelten Abbildung des JAXR-Datenmodells auf UML implementiert. Mit denen aus dem vorherigen Kapitel gewonnen Methoden zur technischen Realisierung wurde "CentraSite Export" als ein auf die XML-Datenbank CentraSites angewandtes XQuery realisiert. Das JAXR-Datenmodell wird also aus der Datenbank nach XMI extrahiert, siehe *Implementierung im CentraSite-Umfeld* 5.

[1] SVG: Scalable Vector Graphics, XML-Dialekt zur Darstellung von Vektorgrafiken.

Das aus dem Export resultierende XMI-Modell orientiert sich, zur besseren Kompatibilität, im Aufbau stark an die Modelle des *Enterprise Architects* von Sparx Systems (siehe 4.7.3).

4.7.2 UModel

Bei UModel 2007 handelt es sich um ein relativ neues UML-Modellierungswerkzeug der österreichischen Firma Altova, vor allem bekannt durch XMLSpy. UModel ist sowohl unter Windows als auch unter Linux (mit WINE[2]) lauffähig.

Altova UModel zeichnet sich beim Import von XMI-Dokumenten besonders durch seine aussagekräftigen Fehlermeldungen aus. Unauflösbare Assoziationen oder fehlende eindeutige Bezeichner werden bemängelt und nicht in das Modell übernommen. Im Gegensatz zur Konkurrenz, bei der der Benutzer von solchen Fehlermeldungen "verschont" bleibt, erweist sich UModel gerade bei der Modellierung eines XMI-Dokuments als sehr hilfreich.

UModels XMI-Import und -Export beschränkt sich ausschließlich auf die Version 2.1. Die Kompatibilität zu früheren Versionen ist nicht implementiert.

4.7.3 Enterprise Architect

Von der australischen Firma Sparx Systems entwickelt, erfreut sich der *Enterprise Architect* (EA) verbreiteter Verwendung in der Industrie. Wie bei UModel lässt sich auch hier neben Windows eine Linux Lauffähigkeit mit Wine erreichen. Enterprise Architect unterstützt sämtliche XMI-Versionen seit Version 1.0 und diente bei der Erstellung dieser Arbeit als Referenz.

Im Gegensatz zu UModel verhält sich der EA wesentlich fehlertoleranter und importiert auch Klassen ohne eindeutige Bezeichner. Fehlerhafte Assoziationen werden nicht übernommen.
Leider werden diese unabsichtlichen Änderungen am Modell im Stillen durchgeführt und der Benutzer erhält keine Rückmeldung über Gelingen oder Misslingen der Aktion.

[2]WINE: WINE is not an Emulator. API um Windows-Software unter Linux laufen zu lassen

So existiert zwar nach dem Import ein Modell; über die Vollständigkeit des Modells wird der Benutzer jedoch im Unklaren gelassen.

Enterprise Architect bietet verschiedene Optionen zum Exportieren an. Darunter fallen auch Optionen, die festlegen, ob ein vollständiges Roundtripping möglich sein soll. Dadurch wird das XMI-Modell um "Tagged Values" ergänzt, die EA-spezifische Ergänzungen enthalten. Ohne diese Einstellungen ist selbst der Export eines Modells mit Stereotypen und der anschließende Rück-Import nicht möglich. Die Stereotypen gehen hierbei verloren.

4.7.4 BOUML

BOUML ist ein freies, ständig weiter entwickeltes, Modellierungswerkzeug unter der GPL 2.0. Es steht für alle gängigen Plattformen zur Verfügung. Der XMI Support wird über sogenannte *Plug-Outs* realisiert, einem Plugin-System mit dem auch weitere Import- und Exportformate realisiert werden können. Den Changelogs der vergangenen Versionen zufolge, wird bei der Entwicklung von BOUML hoher Wert auf die Entwicklung der Import- und Export-Funktionalitäten gelegt. [BOU08]

Entgegen seiner kommerziellen Konkurrenz hat BOUML eine sehr schlichte grafische Oberfläche, die sich auf das Hinzufügen von Klassen, Attributen und den übrigen Elementen der UML beschränkt.

Der XMI-Exporter liegt in der Version 1.5.4 vor. Wie schon bei anderen Programmen festgestellt, werden beim Export modellierte Stereotypen nicht im XMI-Modell berücksichtigt. Beim Import eines eigenen Modells fällt weiterhin auf, dass BOUML den Import von Elementen mit bestimmten Zeichen verweigert, z. B. $<, >, \_$. Laut [OMG06] spezifiziert die UML jedoch keine Namens-Syntax, weshalb diese Einschränkung von BOUML selbst stammt. In der UML sind alle Namen erlaubt, die sich durch einem MOF-Namen und damit als ein XML-Attribut ausdrücken lassen. Diese Einschränkung macht BOUML im vorliegenden Zusammenhang unbrauchbar.

4.7.5 eUML2 und Omondo UML

Neben den erwähnten reinen Modellierungswerkzeugen gibt es Erweiterungen (Plugins) für das Eclipse-Framework. Die bedeutensten Plugins sind hier *Omondo UML* und das darauf aufbauenede *eUML2*. Beide Plugins liegen in einer freien und in einer kommerziellen Variante vor. Da aber beide denselben Ursprung haben, bringen sie auch dieselben Probleme mit. Ein Import von XMI-Modellen ist zwar theoretisch möglich, beim tatsächlichen Import bricht der Vorgang jedoch kommentarlos ab. In einem eigens angelegten Eclipse-Projekt, sind im Vergleich zum Zustand vor dem Import keine Veränderungen festzustellen. Die darauf folgende Internet-Recherche ergab ebenfalls nur unzufrieden stellende Ergebnisse.
Ein möglicher Grund dafür könnte sein, dass keine unmittelbare Notwendigkeit besteht, fertig modellierte UML-Diagramme fremder Herkunft in ein bestehendes Java-Projekt zu importieren. Aus der Sicht von Omondo UML und eUML2 werden Modelle im Projekt erzeugt und mit dem Code im selben Projekt synchronisiert. Die UML-Darstellung soll daher nicht der Modellierung sondern der Implementierung und der Dokumentation dienen.

Ein weiterer, für die Zukunft erfolgversprechender, Ansatz im Bezug auf Eclipse ist das *Eclipse Modelling Framework* (EMF). Durch dieses Framework wird weiteren Plugins eine Möglichkeit gegeben, auf MOF-Modelle zuzugreifen und sie weiterzuverarbeiten. EMF stellt eine Implementierung der EMOF bereit und ermöglicht es, unter anderem XMI-Dateien grafisch zu bearbeiten und zu speichern. [Ecl08]

4.7.6 Rational Systems Developer

Rational Systems Developer von IBM ist ein Nachfolger des klassischen Rational Rose. Der Systems Developer ist für alle gängigen Plattformen erhältlich und funktioniert wahlweise als Eclipse-Plugin oder als autarke Eclipse-Installation. Dabei werden die Vorteile der Eclipse-Plattform durch Verwendung der EMF genutzt.
Mit EMF als Basis ist eine deutlich stärkere Kopplung an die Richtlinien der MOF feststellbar als bei anderen Werkzeugen. So besteht die Möglichkeit, xmi:ids beizubehalten und im Projekt für ein späteres Roundtripping zu bewahren. Der Rational System

Developer bietet insgesamt den ausgereiftesten Eindruck aller Eclipse-basierten Modellierungswerkzeuge. Beim Import traten jedoch nicht reproduzierbare Fehler bei der Referenzierung von Assoziationsenden auf. Klassen und Assoziationen wurden zwar vollständig importiert, die an einer Assoziation beteiligten Klassen zum Teil aber nicht aufgelöst.

Mit dem Rational Systems Developer kann ausschließlich aus bzw. in XMI 2.1 importiert/exportiert werden.

4.8 Portabilität

Bei der Generierung eines XMI-Dokuments eröffnen sich vielfältige Fehlerquellen, die über die Validierbarkeit des Dokuments im Sinne einer XML-Datei hinausgehen. In XMI werden Elemente untereinander mit xmi:ids verknüpft. Neben der Validierbarkeit muss also auch die interne Integretät des Modells sichergestellt werden.

XMI als vollständige Abbildung der UML bietet ebenso viele Elemente, wie sie die UML zu bieten hat und ermöglicht entsprechend viele Kombinationsmöglichkeiten.
Speziell die Darstellung von Stereotypen ist ein leidiges, weil unklares Thema bei der Darstellung in XMI. So ist es beinahe ausgeschlossen, dass Programm A die Stereotypen des XMI-Modells von Programm B interpretiert. Dabei handelt es sich auf der einen Seite um ein Problem der UML, die Stereotypen mittels *Profiles* nur sehr schwammig definiert, und auf der anderen Seite ein Problem der Modellierungswerkzeuge. Diese achten in erster Linie auf ein vollständiges Roundtripping zwischen Import und Export ihrer eigenen Schnittstelle; gerne auch durch proprietäre Erweiterungen am XMI-Modell.

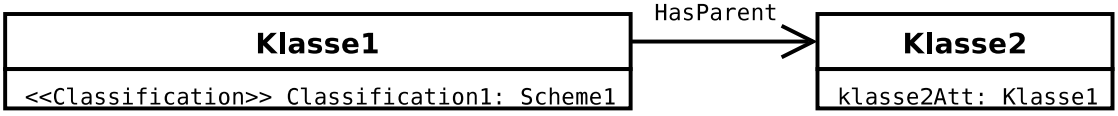

Abbildung 4.8: Test-Modell zur Überprüfung der Portabilität

Um die Eigenschaften des Exports zu testen, soll das Modell aus Abbildung 4.8 im Programm A modelliert werden. Anschließend findet ein Import des Modells durch

Programm B statt. Besonderer Fokus liegt dabei auf dem Umgang mit dem Stereotyp "«Classification»", da hier erfahrungsgemäß die meisten Probleme auftreten. Stereotypen werden mit den zur Verfügung stehenden Funktionen des Programms modelliert.

"×" steht für eine erfolgreiche Importierung des Modells von A nach B. Beschränkungen und Auffälligkeiten werden durch Indizes dokumentiert.

	CS	**EA**	**UM**	**RS**	**BO**	**EU**
CS		$\times^{1}$	$\times$	$\times^{2}$	$!^{3}$	-
EA			$\times^{*}$	$\times^{*}$	$\times^{*}$	-
UM		$\times^{*}$		$\times^{*}$	$\times^{*}$	-
RS		$\times^{*}$	$\times^{*}$		$\times^{*}$	-
BO		$\times^{4*}$	$\times^{*}$	$\times^{5}$		-
EU		-	-	-	-	

Tabelle 4.1: Kreuztabelle XMI-Portabilität

1 Mögliche Unregelmäßigkeiten werden nicht bemängelt und das Modell trotzdem unvollständig übernommen. Fehlerresitzenz bei doppelten Namen und doppelten eindeutigen Bezeichnern (`xmi:id`).

2 Modell wird bis auf Assoziationsenden vollständig übernommen.

3 Namen mit Doppelpunkten, Bindestrichen, etc. führen zum Abbruch des Imports.

4 Datentypen der Attribute gehen verloren.

5 Stereotypen werden samt Profil übernommen, jedoch nicht den Klassen zugewiesen.

* Stereotyp geht verloren.

CS	CentraSite Exporter 0.2
EA	Enterprise Architect 7.1
UM	UModel 2008
RS	Rational Systems Developer 7.0.5
BO	BOUML 4.2.1 (XMI 2.1 Import Release 1.2.6)
EU	eUML2 3.1.0

4.8.1 Unterstützung von Stereotypen

Stereotypen werden als Teil von UML-Profilen definiert. Profile erlauben die Erweiterung und Anpassung des Modells (Tailoring) an eine Anwendungsdomäne. Das kann dann nötig werden, wenn die Mittel der UML zur Beschreibung eines Konzepts im Modell nicht mehr ausreichen.
In dieser Arbeit werden Stereotypen zur Abbildung von Klassifikationen genutzt. (siehe *Abbildung Klassifikation* 3.2.6 auf Seite 27) [R+07]

Beim Betrachten der Versuchsergebnisse fällt jedoch auf, dass bei allen getesteten Modellierungswerkzeugen Probleme bei der Interpretation fremder Stereotypen auftreten. Mit der spezifikationstreuen Abbildung der UML durch XMI, werden auch Profile als optionale und bewusst unklar definierte Elemente übernommen. Aus diesem Grund gibt es verschiedene Möglichkeiten Stereotypen in einem XMI-Modell darzustellen (dazu auch [nUM08]).
Auf der einen Seite stärken UML-Profile das UML-Metamodell durch Erweiterbarkeit, auf der anderen Seite wird dadurch die Portabilität einer standardisierten UML gefährdet. [Hah04]

Zur Umgehung dieses allgemeinen Problems, werden in dieser Arbeit Stereotypen als Teil des Namens der Attribute einer Klasse modelliert.

Kapitel 5

Implementierung im CentraSite-Umfeld

Nach der Abbildung des JAXR-Datenmodells auf das UML-Modell und der Serialisierung dieses Modells durch die Mittel der MOF nach XMI erfolgt nun die technische Umsetzung. Dieser Schritt stellt die Implementierung der vorher entwickelten Abbildungen von JAXR auf UML und von UML auf XMI dar.

Zum Aufbau des XMI-Modells wurden die Informationen über eine XQuery aus der CentraSite zu Grunde liegenden XML-Datenbank Tamino gewonnen (siehe Abbildung 5.1).

Der Aufbau der XQuery gliedert sich in folgende Schritte:

1. Generierung von Klassen, inklusive deren Attribute, Generalisierungen und Klassifikationen.
2. Statische Definition von Basis-Datentypen.
3. Generierung der Assoziationen bzw. Assoziationsklassen.
4. Generierung der Taxonomieklassen und Klassifikationen.
5. Statische Definition von nicht repräsentierten JAXR-Elementen.

Ein auftretendes Problem bei der Serialisierung des Modells ist die Generierung eindeutiger Bezeichner, um Referenzierungen im XMI-Modell zu ermöglichen (`xmi:id`).
Die folgenden Abschnitte behandeln auch dieses Problem. Unter Berücksichtigung bereits bestehender Bezeichner soll die Eindeutigkeit durch Verwendung von kombinierten einmaligen Typeigenschaften erreicht werden.

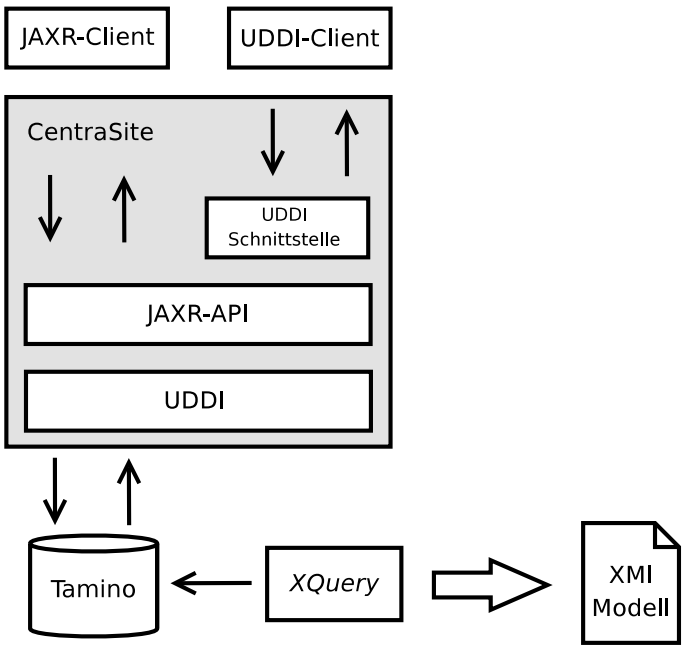

Abbildung 5.1: Aufbau CentraSites als JAXR-Implementierung mit XQuery-Abfrage

5.1 Generierung von Klassen, deren Attribute, Generalisierungen und Klassifikationen

Zu definieren: Als Klasse mit `xmi:id`, `name`

Ziel dieses Schrittes ist die Generierung von Klassen für `User Defined Objects` und Typen der JAXR-Spezifzikation, welche in der Registry aufzufinden sind.

In diesem Schritt wird grundsätzlich die Menge aller Typen als User Defined Objects verstanden. Ausnahmen sind die durch JAXR definierten Typen, welche identifiziert und individuell behandelt werden müssen.

Um die Ableitung aller User Defined Objects von `RegistryEntry` zu realisieren, wird jeder Klasse ein `<generalization>`-Element mit dem Ziel `RegistryEntry` hinzugefügt.

Aufgrund der semantischen Unterschiede zwischen User Defined Objects und spezifizierten JAXR-Typen ist hier eine Verwendung des für Benutzertypen zuständigen Algorithmus unangebracht, da die JAXR-Typen wesentlich komplexere und spezifischere Beziehungen untereinander haben. Beispielsweise würde `RegistryEntry` von sich selbst erben, würde es wie ein User Defined Object behandelt werden.

Die `xmi:id` eines jeden Typs wird mit der Übernahme des bestehenden *UDDI-Keys* aus dem JAXR-Modell realisiert. Dieser stellt eine weltweit eindeutige ID dar und ermöglicht die Referenzierung fast aller Typen im JAXR-Datenmodell.

5.1.1 Attribute

Zu definieren: Als Attribut mit `xmi:id`, `name`, `type`, `minOccurs`, `maxOccurs`

Falls ein Typ Slots aufweist, werden die zugehörigen Attribute über eine weitere Abfrage generiert. Um die Kardinalitäten der Attribute zu bestimmen, werden die Slots `minOccurs` und `maxOccurs` ausgewertet. Die Kardinalitäten können abgesehen von der Unbeschränkheits-Notation übernommen werden. Das in der Implementierung verwendete `unbounded` wird durch die XMI-Notation für *unbeschränkt* mit "-1" definiert. Die Datentypen werden auf die Klassen des Pakets "Basic Datatypes" abgebildet (siehe 5.2).

Die Generierung der `xmi:ids` gestaltet sich hier schwieriger, da Slots nicht über UDDI-Keys identifiziert werden. Um die Eindeutigkeit zu gewährleisten, wird die ID aus dem String "`UML:Attribute:`", dem Namen des enthaltenden Typs und dem Slotnamen gewonnen. Ein exemplarisches Resultat wäre:
`UML:Attribute:InterstageBusinessProcessKey`.

Wobei das Teilwort `InterstageBusinessProcess` für den Namen des Typs, und `Key` für den Namen des Attributs stünde.
Die Eindeutigkeit ist hier gewährleistet, da die Menge aller Slots bezüglich eines Typs eindeutig ist. Soll heißen, ein Slotname darf innerhalb eines Typs nicht mehrfach verwendet werden.

5.1.2 Klassifikation

Zu definieren: Als Attribut mit `xmi:id`, `name`, `type` Zur Angabe der Klassifikationen und ihrer Eigenschaften werden syntaktisch die gleichen Elemente der XMI genutzt wie bei der Attributnotation. Siehe auch 3.2.6 *Klassifikationen* auf Seite 27.

Entgegen der Vorgehensweise zur Generierung von eindeutigen Bezeichnern bei Attributen wird die Eindeutigkeit bei Klassifikationen durch die Verwendung des UDDI-Keys der Taxonomie realisiert. Jeder Typ kann, gemäß JAXR, nur einfach mit einer Taxonomie klassifiziert werden, was wiederum bedeutet, dass die Übernahme des UDDI-Keys der klassifizierenden Taxonomie die Eindeutigkeit des Bezeichners gewährleistet.

Wie in *Portabilität von XMI-Modellen* 4.8 besprochen, muss an dieser Stelle der Name des Attributs eindeutig werden.

Der Ansatz zur Lösung dieses Problems ist die Benennung der JAXR-Klassifizierung. Eine Benennung von Klassifikationen ist aber nicht gebräuchlich, da eine Klassifikation durch die referenzierte Taxonomie seine Semantik ausdrückt.
Falls keine Bezeichnung für eine Klassifikation im Modell vorliegt, soll die Ergänzung des Attributnamens, um den Namen der klassifzierenden Taxonomie, die Eindeutigkeit des Attributs sicherstellen.

5.2 Statische Definition von Basis-Datentypen

Zu definieren: Als Klasse mit `xmi:id`, `name`

Die Basisdatentypen umfassen im Wesentlichen nur eine Menge von Klassen mit einem Namen, der dem verwendeten Datentyp entspricht, z. B. `xs:integer`.
Sämtliche Klassen zur Modellierung von Basis-Datentypen sind im UML-Paket “Basic Datatypes” zusammengefasst (siehe Tabelle 4.2). Die eindeutigen Bezeichner werden aus dem Namen und dem Zusatz `DATATYPES` gewonnen.

xs:string	xs:date
xs:decimal	xs:dateTime
xs:integer	xs:time
xs:float	xs:duration
xs:nonNegativeInteger	xs:anyURI
xs:int	xs:double
xs:long	CS:enumaration
xs:boolean	CS:ipv4_type
xs:anySimpleType	CS:internationalStringType
amount	currencyID

Tabelle 5.1: Statisch angelegte Datentypen

5.3 Generierung der Assoziationen bzw. Assoziationsklassen

Bei der Generierung von Assoziationen wird die klare Trennung von Quelle und Ziel der Assoziation im JAXR-Datenmodell genutzt. So lassen sich beispielsweise alle von einer Klasse ausgehenden Assoziationen (Source) ermitteln.
Um die gewünschten, zu modellierenden Assoziationen abzufragen, werden nur jene Assoziationen angezeigt, die vom Konzept `associatedWith` der Taxonomie `associationType` sind. Im nächsten Schritt wird eine Weiche zwischen Assoziation und Assoziationsklasse gesetzt. Dazu wird ermittelt, ob die Assoziation in ihren Slots Namen vergeben hat, was darauf schließen lässt, ob sie eigene Attribute besitzt, was wiederum für eine Assoziationsklasse zutreffend wäre.

5.3.1 Assoziation

Zu definieren: Als Assoziation mit `xmi:type`, `xmi:id`, `sourceObject`, `targetObject`, `type`, Rollennamen (`name` bzw. Namen der Assoziationsenden), Kardinalitäten (`minOccurs`, `maxOccurs`)

Zur Unterscheidung gegenüber einer Assoziationsklasse wird "`uml:Association`" als `xmi:type` gesetzt.

Jede Assoziation hat zwei `memberEnd`-Elemente, diese referenzieren zusätzlich das Element `ownedEnd` und `navigableOwnedEnd`. `navigableOwnedEnd` ist das navigierbare Ende und wird mit dem Datentyp des `targetObjects` belegt.

Die Ermittlung des Rollennamens erfolgt über die Klassifikation der Assoziation. Analog wird das nicht-navigierbare Ende mit den Eigenschaften des `sourceObjects` belegt. Um hier an den Namen zu gelangen, wird, falls existent, der Slot "`reverseName`" der Assoziation verwendet. Dieser Umstand beruht auf einer Designentscheidung von CentraSite.

Der eindeutige Bezeichner wird aus dem String "`Association:`", dem UDDI-Key der Assoziation und den UDDI-Key von `source-` und `targetObject` gewonnen. Unter der Vorraussetzung, dass in jeder Klasse die Kombination Source/Target + Assoziationstyp eindeutig ist, ist eine Verwendung dieser Beziehung als Teil des eindeutigen Bezeichners angebracht.

5.3.2 Assoziationsklasse

Zu definieren: Als Assoziation mit `xmi:type`, `xmi:id`, `sourceObject`, `targetObject`, `type`, Rollennamen (`name` bzw. Namen der Assoziationsenden), Kardinalitäten (`minOccurs`, `maxOccurs`)

Wie bei der Assoziation wird der `xmi:type` zur Unterscheidung gegenüber der Assoziation mit "`uml:AssociationClass`" gesetzt.

Eine weitere Abfrage generiert `ownedAttribute`-Elemente für die Slots der Assoziation. Der Datentyp wird auf die statischen Datentypen im Paket "Basic Datatypes" abgebildet und die Attributnamen übernommen.

Die restlichen Elemente der Assoziationsklasse (`ownedEnd`, `navigableOwnedEnd` etc.) werden wie bei der Assoziation gesetzt.

5.4 Generierung der Taxonomieklassen

Zu definieren: Als Klasse mit `xmi:id`, `name` Für jede Taxonomie im JAXR-Modell wird eine eigene Klasse im Package "TaxonomyPackage" angelegt. Das Element `packagedElement` wird statisch mit seinem Typ (`uml:Class`) definiert. Dynamisch kommen `name` und `xmi:id` hinzu.

Da die Referenzen zu diesen Klassen schon in 5.1.2 gesetzt wurden, muss die `xmi:id` nach der gleichen Vorgehensweise ermittelt werden. In diesem Fall die Verwendung des UDDI-Keys des `classficiationScheme` im Modell.

5.5 Statische Angabe von nicht repräsentierten JAXR-Elementen

Zur Angabe der noch fehlenden JAXR-Elemente wird ein fertiges XMI-Modell als Klartext in die XQuery geschrieben. XQuery erlaubt die Wiedergabe von Strings, die als solche gekennzeichnet an die Ergebnismenge angehängt werden sollen. Dem neu generierten XMI-Modell wird also ein statisch generiertes XMI-Teilmodell angehängt (siehe auch Abbildung 2.5 auf Seite 18).

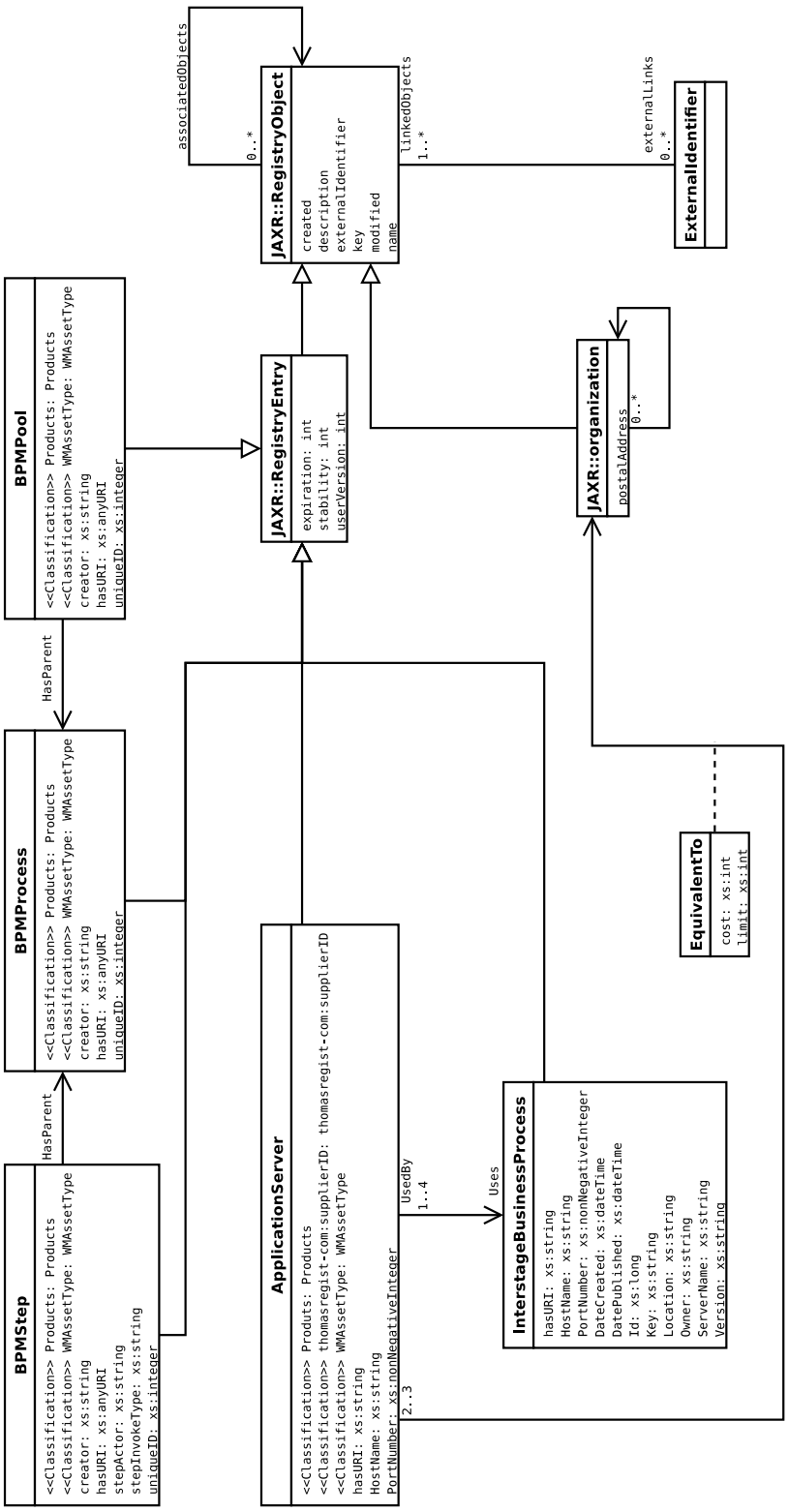

Abbildung 5.2: Ausschnitt eines nach UML transformierten JAXR-Datenmodells

Kapitel 6

Konzept zur Transformation von UML ins JAXR-Datenmodell

Die Abbildung des JAXR-Modells in UML/XMI ermöglicht es dem Modellierer, sich nun mit bekannten Mitteln einen Überblick über das System zu verschaffen. Der nächste logische Schritt wäre die Modellierung in UML und der Export nach JAXR.

Zwar bieten die JAXR-API und CentraSite ausreichende Möglichkeiten, das JAXR-Modell zu erweitern und zu verändern, an den Komfort grafischer Modellierungswerkzeuge reichen sie jedoch nicht heran.
Bisher wurden Veränderungen am Modell in textueller-objektorientierter Form durch Verwendung der JAXR-API und in tabellarisch-objektorientierter Form durch Verwendung von CentraSite Control durchgeführt.

Nach dem *Export* aus dem JAXR-Modell soll nun ein Konzept zur Realisierung des *Imports* von UML nach JAXR entwickelt werden.

6.1 Problem der Uneindeutigkeit

Da in diesem Schritt auf die von verschiedenen Modellierungswerkzeuge generierten XMI-Modelle zurückgegriffen werden muss, wird man auf eindeutige Bezeichner von Typen und Assoziationen verzichten müssen. Nahezu alle untersuchten Programme verwerfen die vom importierten XMI-Modell angegebenen eindeutigen Bezeichner und

ersetzen diese durch eigens generierte. Durch den Verlust dieser Referenzierungen wird die eindeutige Identifikation von bereits bestehenden Elementen im JAXR-Modell unmöglich.

Es müssen also Konzepte zur Identifikation von Elementen entwickelt werden. Ein Ansatz wäre die Identifikation über den Namen der Elemente und ihrer Rolle in Beziehungen. Die Klasse `Service` würde also über Namensvergleiche mit dem Typ `service` im Modell identifiziert werden können. Erkennbare Unterschiede müssten zugunsten des UML-Modells übertragen werden müssen.

Da dieser Ansatz sehr fehleranfällig und aufwendig ist und es bei geringsten Abweichungen zu Duplizitäten und Redundanzen kommen kann, soll von einer leeren Registry ausgegangen werden. Die Registry soll ohne User Defined Objects und weitere nicht deterministische Eigenschaften des Modells vorliegen.

Ein vollständiges Roundtripping wäre aufgrund der genannten Probleme in jedem Fall ausgeschlossen.

Zusätzlich ist es notwendig Designrichtlinien anzugeben, die dem Modellierer gewisse Vorgaben und Einschränkungen machen. So darf der Modellierer bspw. keine zusätzlichen Datentypen anlegen, da diese Funktionalität durch die Registry nicht vorgesehen ist. Darüber hinaus werden Klassifikationen nur als solche erkannt, wenn sie sich an der Notation für Klassifikationen orientieren mit einer benannten Assoziation zu einer Klasse im Paket `Taxonomy`.

6.2 Durch JAXR spezifizierte Elemente

Zur Abbildung der spezifizierten JAXR-Elemente würde, wie beim Export, ein großer Teil statisch angegeben werden müssen. Zum Vorteil gelingt dabei die Tatsache, dass CentraSite über alle Installationen hinweg einheitliche UDDI-Keys für JAXR-Typen definiert. Auf Basis dieses Wissens ist es möglich, Assoziationen bzw. Klassifizierungen von Typen wie `RegistryEntry` zu erkennen, in das JAXR-Modell zu integrieren und mit dem bekannten UDDI-Key als Quelle oder Ziel zu versehen.

Eine detaillierte Typdefinition der statischen JAXR-Typen, wie beim Export, findet also nicht statt, da davon ausgegangen werden kann, dass diese im Modell bereits definiert sind. So muss zwar von einer leeren von User Defined Objects bereinigten Registry ausgegangen werden, eine Registry im Auslieferungszustand mit den definierten JAXR-Typen kann jedoch als Basis genutzt werden.

6.3 UML-Klasse

Zur Abbildung von UML-Klassen im Modell wird für jede Klasse ein User Defined Object angelegt. Attribute werden, inklusive ihrer Kardinalität und Datentyps, als Slots modelliert. Da die Datentypen der Attribute im XMI-Modell an den JAXR-Datentypen (inkl. Erweiterungen) orientiert sind, können die Datentypen problemlos über ihren Namen identifiziert werden.

6.4 Assoziationen und Klassifikation

Zugehörige Assoziationen werden über die Eindeutigkeit des Tupels `sourceObject` und `targetObject` identifiziert und entsprechend angelegt. Die zugehörige Taxonomie `AssociationType` wird, wie gehabt, mit dem Konzept `associatedWith` definiert.

Die Abbildung der Klassifikationen gestaltet sich schwieriger. Klassifikationen müssen aufgrund des Musters "«Classification» Name" erkannt werden. Nach erfolgreicher Identifikation einer Klassifikation muss der Typ der Taxonomie im JAXR-Modell über den Datentyp des UML-Attributs ermittelt und assoziiert werden. Das Klassifikationsschema `AssociationType` wird als `classifiedBy` definiert.

6.5 Nicht berücksichtigte Elemente

Da einige JAXR-Elemente, bei der Abbildung in das UML-Modell, keine Berücksichtigung gefunden haben, müssen diese Elemente durch die JAXR-API oder anderweitig generiert werden. Dazu zählen z. B. der zum Element assoziierte User und die Beschreibung des Elements. An dieser Stelle gehen Informationen über die Zuständigkeiten und Veränderungen am JAXR-Modell verloren. Dieser Informationsverlust ist aber insofern zu verschmerzen, als das erhebliche Veränderungen am Modell durchgeführt werden und vormalige Verantwortlichkeiten ihre Bedeutung verloren haben.

6.6 Implementierung

Durch die Anzahl der notwendigen Veränderungen an der Registry steigt die Dimension der möglichen Fehler und Inkonsistenzen. Um die Integrität der Registry zu wahren, ist dabei die Verwendung der JAXR-API zu überprüfen.

Mittels direkter Manipulation an der zu Grunde liegenden Datenbank könnten zwar allumfassend Änderungen durchgeführt werden, dadurch entstünde aber eine nicht mehr zu kalkulierende Fehleranfälligkeit. Die JAXR-API würde diese Fehlermöglichkeiten eingrenzen und Inkonsistenzen im Modell minimieren. Dennoch ist zu überprüfen, ob der gewünschte Funktionsumfang erfüllt wird. Die Einschränkung, über die API keine User Defined Objects definieren zu können, sei hier als Beispiel genannt.

6.7 Randbedingungen für das Konzept

Es sind also für die Implementierung folgende Punkte festzustellen:

- Anwendung erfolgt auf eine Registry im Auslieferungszustand.
- Beschränkung auf die Klassen des Root Packages (User Defined Objects, ohne "JAXR").
- Beschränkung auf die in "Basic Datatypes" definierten Datentypen.
- Auflösung von Assoziationen und Klassifikationen und die Zuweisung des entsprechenden `AssociationTypes`.
- Notwendigkeit des Direktzugriffs auf die Datenbank zur Erstellung von User Defined Objects.
- Weitestgehende Verwendung der JAXR-API zur Vermeidung von Inkonsistenzen.

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Diese Arbeit entwickelt ein Konzept zur dynamischen Abbildung des JAXR-Datenmodells und seiner benutzerdefinierten Erweiterungen (User Defined Objects) auf UML und liefert eine Implementierung für den Einsatz im CentraSite-Umfeld.

Da es sich bei dem JAXR-Datenmodell um einen objektorientierten Ansatz handelt, konnten für die Konzepte des JAXR-Datenmodells Analogien aus dem UML-Modell gefunden werden. Dabei wurden die JAXR-Elemente der Typen, Slots, Assoziationen und Klassifikationen auf ihre Eigenschaften überprüft und eine sinnvolle Abbildung auf Elemente der UML ermittelt.
Die Menge aller verfügbaren Lösungen wurde nach Semantik, Realisierbarkeit und Übersichtlichkeit beurteilt. Zur Abbildung erfolgte eine Gewichtung dieser Aspekte und ein Abgleich der JAXR- und UML-Spezifikation hinsichtlich jener Semantik und der Notation.
JAXR-Typen werden auf UML-Klassen abgebildet, da JAXR-Typen, ebenso wie UML-Klassen, eine Menge von Objekten nach ihren charakteristischen Eigenschaften gruppieren. Slots zur Darstellung von Eigenschaften von Typen werden als Attribute einer UML-Klasse abgebildet. Die Assoziationen aus dem JAXR-Datenmodell werden als gerichtete UML-Assoziationen dargestellt. Zusätzlich wurden Assoziationen, zu deren genauerer Spezifizierung zusätzliche Slots definiert wurden, auf UML-Assoziationsklassen abgebildet werden.

Besonders komplex gestaltete sich die Wahl der Darstellungsform für Klassifikationen. Da ein Konzept zur Klassifikation in der UML nicht existiert, musste die Darstellung über Stereotypen realisiert werden. Stereotypen können Klassen einordnen und bezeichnen, was dem Sinn einer Klassifikation am ehesten entspricht.
In der Arbeit wurden verschiedene Varianten zur Darstellung von Klassifikationen in der UML entwickelt. Besonders wurde hier Wert auf die Ausgewogenheit der Aspekte Semantik, Realisierbarkeit und Übersichtlichkeit gelegt. Die Semantik wurde durch die Abbildung mittels Stereotypen weitestgehend erfüllt, die Übersichtlichkeit konnte durch Vermeidung von zusätzlichen Beziehungen im Modell sichergestellt werden, zur Überprüfung der Realisierbarkeit wurde die Darstellung von Stereotypen überprüft.

Zur automatisierten Generierung des UML-Modells, aus dem von CentraSite implementierten JAXR-Datenmodell, wurde eine serialisierte Form der UML notwendig. Dazu wurde die Meta Object Facility (MOF) der OMG betrachtet. MOF bietet einen einheitlichen Standard zur Darstellung von Metamodellen und spezifiziert deren Serialisierung in den XML-Dialekt XMI.
Mit XMI ist es möglich UML-Modelle zwischen unterschiedlichen Modellierungswerkzeugen auszutauschen. Als Teil dieser Arbeit wurde das dynamisch generierte XMI-Modell in verschiedene Programme importiert und auf seine Vollständigkeit überprüft. Ergebnis dieser Untersuchung war die Feststellung der mangelhaften Unterstützung von Stereotypen. Zur Umgehung dieses hausgemachten Problems der UML, und damit XMI, wurden Stereotypen als Teil des Namens modelliert und von der nur schwach spezifizierten Variante über Profile abgesehen.

Auf Basis des entwickelten Abbildungskonzepts konnte eine XQuery zur dynamischen Generierung des UML-Modells entwickelt werden.
Mittels dieser Implementierung kann das Datenmodell jederzeit, vollständig und mitsamt seiner benutzerdefinierten Erweiterungen, in ein UML-Modell transformiert werden.
Dadurch zeigt sich für den Modellierer ein völlig neues Bild seines Modells. War es ihm zuvor nur möglich, das erstellte Modell vor seinem geistigen Auge als Ganzes zu betrachten, bietet sich für ihn nun die Möglichkeit die Komposition aller erstellten Typen, Slots, Assoziationen und Klassifikationen in standardisierter Form darzustellen. Das generierte Modell kann dem Modellierer bei der Evaluation und Dokumentation

dienen, zusätzlich ist es ihm überlassen, welche Aspekte des Modells er herausstreicht und als interessant beurteilt.

Unabhängig von der vorliegenden JAXR-Implementierung ist es darüber hinaus möglich, die entwickelten Konzepte auch auf andere JAXR-Implementierungen zu übertragen.

7.2 Ausblick

Mithilfe der erarbeiteten Konzepte und den entwickelten Mechanismen zur automatischen Generierung des Modells können zukünftig weitere Erweiterungen durchgeführt werden.
Eine sinnvolle und populäre Möglichkeit wäre die grafische Darstellung von UML-Diagrammen über CentraSite-Control. Mit dem XML-Dialekt der Scalable Vector Graphics (SVG) stünde dazu ein passendes Grafikformat bereit. Unter Verwendung von SVG kann durch eine XML-Transformation ein in XMI integriertes UML-Diagramm als Vektorgrafik dargestellt werden.

Raum für weitere Verbesserungen bietet die Portabilität des Modells. Möglicherweise könnte auf die Eigenarten der verschiedenen Modellierungswerkzeuge eingegangen werden, um so eine vollständige Unterstützung zu erreichen. Vorstellbar wäre dazu eine programmabhängige XMI-Generierung. Eine derartige Spezialisierung einzelner Teile des XMI-Modells würde jedoch einen erheblichen Implementierungsaufwand mit sich ziehen und von der Idee eines standardisierten XMI-Modells abkommen. Der sinnvollste Weg wäre hier die Beobachtung der Entwicklungen, insbesondere im Hinblick auf das Eclipse Modelling Framework (EMF) oder die Fokussierung auf ein bestimmtes Werkzeug und seine Eigenheiten.

Logische Folge dieser Arbeit und Weiterführung des Konzepts aus Kapitel 6 ist die Transformation von UML-Modellen ins JAXR-Datenmodell. Dieser umgekehrte Weg böte dem Modellierer die Möglichkeit UML-Modelle mit einem Modellierungswerkzeug seiner Wahl zu entwickeln, und es auf Basis der erarbeiteten Konzepte in die entsprechende SOA-Umgebung zu importieren.

Anhang

Auf der beiliegenden CD-ROM befinden sich:

- Dieses Dokument im PDF-Format
- Zitierte Internetquellen im PDF-Format
- Die entwickelte XQuery-Implementierung
- Ein beispielhaftes XMI-Modell

Glossar

AJAX

Asynchronous JavaScript and XML. Techniken zur Entwicklung von asynchronen und interaktiven Web-Applikationen

API

Application Programming Interfaces (API) stellen mithilfe von Programmiersprachenabhängigen Aufrufen und Strukturen eine Schnittstelle zu einem Programm bereit

Roundtripping

Ist die Konvertierung eines Formats in ein Anderes und zurück. Entscheidend ist dabei die anschließende Übereinstimmung des Formats mit dem Original

Serialisierung

Beschreibt das Speichern eines oder mehrerer Objekte auf einem Speichermedium

Taxonomie

Ist die Durchführung und Wissenschaft der Klassifikation

Webservice

Ein Software System, dass die Interoperabilität von Software über ein Netzwerk ermöglicht

WSDL

XML-Dialekt zur Beschreibung von Webservices

XQuery

Ist eine Abfragesprache für XML-Dokumente mit den Möglichkeiten einer Programmiersprache

Stichwortverzeichnis

Literaturverzeichnis

[BOU08] BOUML: *BOUML a free UML 2 tool box*. http://bouml.free.fr, 2008. – [Online; 15.05.2008]

[Bra03] BRAUN, Edna: Introduction to MOF. In: *Object Oriented Information Systems, International Conference* (2003)

[CIO08] CIO: *Service-orientierte Architekturen 2008 - SOA Markt klärt sich auf*. http://www.cio.de/knowledgecenter/soa/850601/index.html, 2008. – [Online; 15.05.2008]

[Ecl08] ECLIPSE.ORG: *Eclipse Modelling Framework (EMF) FAQ*. http://wiki.eclipse.org/EMF-FAQ, 2008. – [Online; 15.05.2008]

[Fow05] FOWLER, Martin: *UML Distilled Third Edition*. Amsterdam : Addison Wesley, 2005

[fre08] FREEBXML: *ebXML Overview*. http://ebxmlrr.sourceforge.net/wiki/index.php/Overview, 2008. – [Online; 15.05.2008]

[Gop06] GOPINATH, Shankar: *Real-Time UML to XMI Conversion*. 2006

[Hah04] HAHN, Jürgen: *Profiles - das Ende einer standardisierten UML?* http://www.jeckle.de/uml-glasklar/profiles.pdf, 2004. – [Online; 15.05.2008]

[Har08] HARBARTH, Juliane: *JAXR*. http://documentation.softwareag.com/crossvision/inm/print/jaxr_tutorial.pdf, 2008. – [Intern, Ausschnitt, Online; 15.05.2008]

[Jav02] JAVA WORLD: *Discover and publish Web services with JAXR*. http://www.javaworld.com/javaworld/jw-06-2002/jw-0614-jaxr.html, 2002. – [Online; 15.05.2008]

[Jec99] JECKLE, Mario: XMI - Case-Tool unabhängiges Modellieren. (1999)

[Kov02] KOVSE, Jernej: Generic XMI-Based UML Model Transformations. In: *Object Oriented Information Systems, International Conference* (2002)

[M+07] MELZER, Ingo u. a.: *Service-orientierte Architekturen mit Web Services*. München : Spektrum, 2007

[Mar08] MARCHAL, Benoit: *ebXML: An Electronic Business Scenario*. `http://www.developer.com/xml/article.php/2234201`, 2008. – [Online; 15.05.2008]

[nUM08] NUML: *Profiles and Stereotypes*. `http://numl.sourceforge.net/index.php/Profiles_and_Stereotypes#XMI_Representation`, 2008. – [Online; 15.05.2008]

[OAS03] OASIS: Introduction to UDDI: Important Features and Functional Concepts. (2003)

[OMG05] OMG: MOF 2.0/XMI Mapping Specification, v2.1. (2005)

[OMG06] OMG: *Issue 6389: Syntax of names*. `http://www.omg.org/issues/uml2-rtf.html#Issue6389`, 2006. – [Online; 15.05.2008]

[Ora06] ORACLE: *Mastering SOA Building a Portfolio of Services*. `http://www.oracle.com/technology/tech/soa/mastering-soa-series/part1.html`, 2006. – [Online; 15.05.2008]

[Pil03] PILONE, Daniel: *UML kurz und gut*. Köln : O'Reilly, 2003

[R+07] RUPP, Chris u. a.: *UML glasklar*. München : Hanser, 2007

[Sch07] SCHÖNING, Harald: Enhanced Modelling Capabilities - Design Specification of INM2301. (2007)

[Sof08a] SOFTWARE AG: *Introducing CentraSite*. `http://documentation.softwareag.com/crossvision/inm/concepts/conceptsover.htm`, 2008. – [Online; 15.05.2008]

[Sof08b] SOFTWARE AG: *Open Standards*. `http://www1.softwareag.com/gr/products/cv/overview/standards/default.asp`, 2008. – [Online; 15.05.2008]

[SUN02] *JAVA API for XML Registries (JAXR)*. Bd. JAXR Version 1.0. Sun Microsystems, 2002

[Uck07] UCKAT, Benedikt: *Metamodellierung mit MOF und Ecore und deren Anwendung im Rahmen des MDA-Ansatzes*. 2007

[UML07] *OMG Unified Modeling Language (OMG UML), Superstructure (UML-Spezifikation)*. Bd. V2.1.2. OMG, 2007

[Web00] WEBBER, David R R.: Understanding ebXML, UDDI and XML/EDI. (2000)

[Wol05] WOLFF, Julia: *Visualisierung rekonstruierter Softwarearchitektur mit UML*. 2005

[Zai06] ZAIKIN, Mikalai: *JAXR*. `http://java.boot.by/wsd-guide/ch06.html`, 2006. – [Online; 15.05.2008]