Diplomarbeit

Refactoring in der Ontologiegetriebenen Softwareentwicklung

bearbeitet von

Erik Tittel

geboren am 23.06.1986 in Dresden

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Betreuer:
Dipl.-Inf. Jan Reimann, Lehrstuhl Softwaretechnologie
Dipl.-Inf. Tobias Israel, buschmais GbR

Hochschullehrer:
Prof. Dr. rer. nat. habil. Uwe Aßmann

Eingereicht am 03.05.2011

Technische Universität Dresden
Fakultät Informatik

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Name des Studenten: **Erik Tittel**

Immatrikulations-Nr.: **3320620**

Thema: **Refactoring in der Ontologiegetriebenen Softwareentwicklung**

Zielstellung:

Einführung in das Thema

Kein IT-System wird einmal erstellt und erfüllt dann für alle Zeit unverändert und in gleichbleibender Qualität seine ihm zugedachten Aufgaben. Fachliche wie technische Rahmenbedingungen ändern sich stetig und erfordern so eine hohe Flexibilität in vielen Bereichen der Softwareentwicklung. Die Qualität eines Anwendungssystems wird damit nicht allein an dessen Status quo gemessen. Die Fähigkeit sich in kurzer Zeit an neue Gegebenheiten anzupassen ist damit immer auch ein gleichwertiges Qualitätskriterium bei der Bewertung von Softwaresystemen.
Bei der Organisation von Datenmengen und deren Zugang über entsprechende Suchmethodiken, etabliert sich mehr und mehr auch der Einsatz von Ontologien. Diese beschreiben in formaler Notation die (Fein-)Struktur einer Fachdomäne. Dabei sind Ontologien, wie sie im Zusammenhang dieser Arbeit betrachtet werden, selbst wiederum Datenstrukturen. Sowohl das Domänenmodell einer Anwendung als auch die zugehörige Ontologie unterliegen gleichermaßen der Anforderung, flexibel an sich ändernde Gegebenheiten angepasst werden zu können. Die Betrachtung dieser Anforderung soll den Rahmen dieser Arbeit beschreiben.

Fortsetzung Rückseite

Betreuer: Dipl.-Inf. Jan Reimann
Externe Betreuer: Tobias Israel, buschmais GbR

Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Beginn am: 15.11.2010
Einzureichen am: 15.05.2011

Dresden, 19.10.2010

Prof.Dr.rer.nat.habil. U. Aßmann
verantwortl. Hochschullehrer

Fortsetzung Zielstellung:

Aufgabenstellung

Mit der Arbeit soll ein Ansatz für ein (teil-)automatisiertes Refactoring von Domänenmodellen im Zusammenspiel mit assoziierten Ontologien erarbeitet werden. Folgender Prozess wird grundlegend angenommen:
(1) Die Ontologie wird verändert.
(2) Die Datenstruktur des Anwendungssystems wird an die neue Ontologie angepasst.
(3) Die bestehenden Daten werden in die neue Datenstruktur migriert.

Für diesen Anpassungsprozess sind folgende Einzelaspekte zu betrachten:
(1) Wie lassen sich Änderungen der Ontologie klassifizieren?
(2) Welche Konsequenzen haben die Änderungstypen für die Struktur des Datenmodells? Kann die Anpassung des Datenmodells formal beschrieben werden? Wie kann eine Überführung in die neue Struktur (semi-)automatisch durchgeführt werden?
(3) Welche Konsequenzen hat die Änderung des Datenmodells auf den Datenbestand? Wie kann der Datenbestand überführt werden? Wie kann die Konsistenz des Datenbestands nach der Überführung sichergestellt werden?

In einem zweiten Teil der Arbeit sollen durch eine prototypische Entwicklung die Möglichkeiten einer Werkzeugunterstützung des zuvor betrachteten Refactoring-Prozesses dargestellt werden. Basis der umzusetzenden Beispiele ist die Erarbeitung einer Ontologie, in der die Aspekte einer Skillprofilverwaltung für die Mitarbeiter eines Unternehmens abgebildet werden. Das daraus abgeleitete Domänenmodell ist Grundlage einer fiktiven Anwendung. Diese Anwendung ermöglicht die gezielte Suche nach Mitarbeitern, die auf Basis ihrer Ausbildung, erworbener Erfahrungen, etc. für eine Aufgabe/ein Projekt besonders geeignet erscheinen. Die Umsetzung der Anwendung selbst ist nicht Gegenstand der Arbeit. Das beispielhaft umzusetzende Refactoring-werkzeug soll den Entwicklungsprozess dahingehend unterstützen, dass die zuvor betrachteten Auswirkungen einer Ontologieanpassung sich unmittelbar auf das von ihr abgeleitete Datenmodell bzw. vorhandene Datenbestände desselben Modells voll- oder teilautomatisiert übertragen. Für die Notation des Domänenmodells soll EMF/Ecore verwendet werden.

Ziele der Arbeit

Im Rahmen der Diplomarbeit sollen folgende Teilziele erreicht werden:

- Bewertung existierender Forschungsarbeiten hinsichtlich ihrer Relevanz und Einsetzbarkeit im Bezug auf die Diplomarbeit
- Analyse und Kategorisierung möglicher Ontologieänderungen, sowie möglicher Relationen zwischen Ontologien und Datenmodellen
- Definition von Co-Refactorings welche Änderungen an Ontologien semantikerhaltend im Datenmodell umsetzen
- Untersuchung inwieweit Co-Refactorings für die Evolution von Ontologien und den abgeleiteten Datenmodellen einsetzbar sind
- Evaluierung vorhandener Werkzeuge in den Bereichen Modellierung (z. B. EMF, EMFText, GMF), Refactoring (z. B. Refactory) und Ontologien (z. B., OntoMopp, OWLText) hinsichtlich ihrer Eignung für den Bau eines Prototypen
- Konzeption und Implementierung eines Eclipse-basierten Werkzeugs zur Evaluierung der praktischen Nutzbarkeit der theoretischen Ergebnisse
- Evaluierung des erstellten Werkzeugs anhand des o. g. Anwendungsszenarios

Vorwort

Abstract

In this thesis an approach is elaborated for the development and evolution of ontology-driven software systems. Ontology-driven software systems are software systems for which ontologies serve as main design documents. Ontologies are furthermore central parts of the running system. They describe the structure of the system and hold data. Parts of the software system are automatically derived from the structure descriptions of the ontology. This work concentrates on the evolution of those systems, thereby defining a catalogue of ontology refactorings. A tool suite called OntoMore is implemented to show the feasibility of the elaborated approach. OntoMore can transform ontologies in metamodels and models of EMF to integrate them in software systems. It can furthermore execute refactorings synchronously on both structures, which is called Co-Refactoring. Hence the consistent evolution of ontologies and models is ensured. The implementation is evaluated with an example ontology about the freelancer domain.

Abstract (deutsch)

In der vorliegenden Arbeit wird ein Konzept zur Entwicklung und Evolution ontologiegetriebener Softwaresysteme erarbeitet. Ontologiegetriebene Softwaresysteme sind Softwaresysteme, bei denen Ontologien als zentrale Designdokumente zum Einsatz kommen. Ontologien sind gleichzeitig zentrale Bestandteile des ausführbaren Systems und dienen zur Strukturbeschreibung und Datenhaltung. Dabei werden Teile des Softwaresystems automatisch aus den Strukturbeschreibungen der Ontologie abgeleitet. Diese Arbeit konzentriert sich auf die Weiterentwicklung solcher Systeme und stellt dafür einen Katalog von Ontologie-Refactorings auf. Es werden mehrere Werkzeuge, gemeinsam als OntoMore bezeichnet, implementiert, um die Umsetzbarkeit des aufgestellten Konzepts zu zeigen. OntoMore kann Ontologien in Metamodelle und Modelle des EMF umwandeln und somit in Softwaresysteme integrieren. Außerdem ist es in der Lage, Refactorings auf beiden Strukturen synchron auszuführen. Dieser Prozess wird als Co-Refactoring bezeichnet. Damit wird die konsistente Evolution von Ontologien und Modellen sichergestellt. Die Implementierung wird anhand einer Beispiel-Ontologie zum Freelancer-Management evaluiert.

Danksagung

Diese Arbeit wurde in Kooperation mit der buschmais GbR unter der Betreuung von Herrn Dipl.-Inf. Tobias Israel geschrieben.

Meinem Betreuer Tobias Israel sei an dieser Stelle ein besonderer Dank ausgesprochen. Er hat mir in vielen anregenden Diskussionen geholfen, immer auf dem richtigen Weg zu bleiben. Auch den anderen Mitarbeitern von buschmais, insbesondere Torsten Busch, Christiane Köhler und Frank Schwarz, sei für die umfangreiche Unterstützung gedankt.

Ein besonderer Dank gilt ebenso meinem Betreuer vom Lehrstuhl Softwaretechnologie Jan Reimann, der mir mit konstruktiver Kritik und vielen Verbesserungsvorschlägen stets weitergeholfen hat. Ich danke ebenfalls Christian Wende, der geduldig alle meine Fragen zu den am Lehrstuhl entwickelten Werkzeugen beantwortete.

Außerdem sei Matthew Horrdige von der University of Manchester gedankt. Durch seine Hilfe war es mir möglich, auch die kleinsten Details der Manchester OWL Syntax zu verstehen. Abhishek Shivkumar von ClearForest verdanke ich im Übrigen viele Erkenntnisse bezüglich der Abbildung von Ontologien auf Domänenmodelle.

Ich danke meinen Eltern, dass sie Ablenkungen von mir fern hielten und es mir ermöglichten, mich auf meine Arbeit zu konzentrieren. Sie standen mir mit ihren Ratschlägen stets zur Seite.

Nicht zuletzt bedanke ich mich bei allen Helfern, die mit ihren Korrekturen, Hinweisen und Verbesserungsvorschlägen sehr zum Gelingen der Arbeit beigetragen haben. Dazu zählen, abgesehen von den bereits genannten Personen: Anne Berger, Ulrich Kramer und Lisa Rothe.

Erik Tittel, Mai 2011

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Die Welt ändert sich. Auch Softwaresysteme müssen sich ändern, denn sie stellen einen Teil der wirklichen Welt dar, die so genannte Anwendungs- oder Fachdomäne. Als Softwaresystem wollen wir hier ein Computerprogramm bezeichnen, das für den industriellen Einsatz in großen Unternehmen geeignet ist. Ein solches Programm ist universal einsetzbar, wurde getestet, dokumentiert, besteht aus mehreren Modulen und verfügt über ein klar definiertes Interface [Bro95]. Nehmen wir als Beispiel ein Softwaresystem, das für ein Unternehmen Daten über Kunden und verkaufte Produkte speichern und zurückgeben soll. Die Anforderungen des Unternehmens und der Mitarbeiter an dieses Softwaresystem werden sich im Laufe der Zeit ständig ändern. Softwaresysteme müssen demnach beständig weiterentwickelt werden. Andernfalls nimmt der Nutzen des Softwaresystems ab [Leh96]. Es ist also wichtig, während der Entwicklung und des Betriebs eines Softwaresystems Techniken einzusetzen, die Änderungen auf einfache Weise ermöglichen.

Ein Softwaresystem soll Daten speichern und dem Benutzer Informationen zur Verfügung stellen. Daten sind Repräsentationen von Eigenschaften von Objekten der realen Welt [Row07]. In der Unternehmensanwendung sind Daten zum Beispiel Name und Adressen der Kunden sowie Eigenschaften von Produkten. Das *Domänenmodell* ist eine abstrakte Beschreibung aller für das Softwaresystem relevanter Daten und deren Beziehungen und stellt damit die Struktur eines Softwaresystems dar.

Daten werden oft in Datenbanken gespeichert. Die Speicherung erfolgt dort in tabellarischer Form. Beziehungen zu Daten anderer Tabellen werden durch Schlüsselwerte ausgedrückt. Die Entwicklung des Semantic Web [BLHL01] hat zu einer Reihe neuer Techniken zur Datenspeicherung geführt, weil im Web die Daten nicht zentral gespeichert und verwaltet werden, sondern in unterschiedlicher Struktur über das ganze Web verteilt sind. Die homogene, tabellenartige Struktur der Daten ist somit nicht gegeben. Die Beziehungen zwischen den Daten treten an dieser Stelle stärker in den Vordergrund. Zu diesen neuen Techniken gehören die *Ontologien*. Ontologien sind Begriffsnetze, die eine netzartige Struktur von Daten beschreiben. Sie ermöglichen gegenüber Datenbanken einige zusätzliche Funktionen, die sie als Datenspeicher interessant machen. Dazu gehören beispielsweise die semantische Suche, Ableitung impliziten Wissens und Konsistenzprüfungen [HKRS08].

Im Rahmen dieser Arbeit wird untersucht, welche Möglichkeiten bestehen, Softwaresystemen mit Ontologien zu modellieren. Die in der Ontologie definierten Konzepte müssen folglich auf ein Domänenmodell abgebildet werden. Somit können die in der Ontologie gespeicherten Daten und Strukturen direkt in das Softwaresystem integriert werden. Dies erfordert jedoch, dass das Domänenmodell in einer Struktur vorliegt, die eine definierte Beziehung zwischen Domänenmodell und Ontologie erlaubt. Dies ist bei der *modellgetriebenen Softwareentwicklung* (MDSD) der Fall. In der MDSD werden Software-Systeme auf einer höheren Abstraktionsebene mit Modellen beschrieben [SVEH07]. Das Domänenmodell liegt hier explizit in Form von Metamodellen vor, die mit den Ontologien in Beziehung gesetzt werden können.

Die Herausforderung besteht nun darin, Änderungen der Struktur der Daten oder der Daten selbst möglichst einfach auf die Ontologie und das Domänenmodell zu übertragen. Für die Durchführung von Änderungen mit klar definierten Anfangs- und Endzuständen hat sich das *Refactoring* in der Softwareentwicklung etabliert. Das Prinzip des Refactorings wurde ursprünglich dazu entworfen, das Design von Programmcode durch systematische Umstrukturierungen zu verbessern. Diese Verbesserungen äußern sich beispielsweise in einer besseren Lesbarkeit oder Wartbarkeit des Programmcodes. Martin Fowler hat für objektorientierte Computerprogramme einen allgemein anerkannten Katalog von Refactorings aufgestellt [FBB+99]. Dabei beschreibt ein Refactoring eine Folge von Änderungen, die in ihrer Gesamtheit zu einer wohldefinierten Umstrukturierung führen. Außerdem existiert mit *Refactory* bereits ein Framework, welches Refactorings auf beliebigen Modellen durchführen kann [RSA10]. Nun soll das Prinzip des Refactorings auch auf Ontologien übertragen werden, um Änderungen systematisch durchzuführen. An dieser Stelle stehen wir vor dem Problem, dass Refactorings auf zwei zusammengehörigen Strukturen – Ontologie und Domänenmodell – synchron ausgeführt werden müssen. Um dieses Problem zu lösen, wird das Prinzip des *Co-Refactorings* eingeführt. Dies ermöglicht die Synchronisation von Refactorings auf zwei unterschiedlichen Strukturen. Eine Änderung der Ontologie oder des Domänenmodells führt somit zu einer Änderung der jeweils anderen Struktur, sodass die Beziehung zwischen beiden Strukturen konsistent bleibt.

1.2 Zielsetzung

Ziel der Arbeit ist es, zu untersuchen, inwieweit Ontologien zum Design von Softwaresystemen eingesetzt werden können. Dazu werden vier Teilbereiche bearbeitet. Zuerst ist die Beziehung zwischen Ontologien und Domänenmodellen zu untersuchen. Es soll festgestellt werden, in welchem Umfang Ontologien auf Domänenmodelle abgebildet werden können und wie eine entsprechende Transformation formalisiert werden kann. Im Rahmen des MOST-Projekts[1] wurde bereits eine Abbildung von Domänenmodell auf Ontologie erarbeitet. Diese kann als Ausgangspunkt für die Erarbeitung einer Abbildung in die andere Richtung dienen.

Des Weiteren sind Refactorings für Ontologien und Domänenmodelle zu entwerfen. Diese Refactorings stellen formale Änderungsoperationen dar, die es ermöglichen, Ontologien und Domänenmodelle von einem definierten Zustand in einen anderen definierten Zustand zu versetzen. Die Definition der Ontologie-Refactorings kann sich dabei an vorhandenen Arbeiten zur Ontologie-Evolution orientieren [NK04, GPS10]. Darauf aufbauend soll ein Konzept für Co-Refactorings aufgestellt werden, mit dem Refactorings auf mehreren Modellen synchron ausgeführt werden können. Refactory als Werkzeug zum generischen Modell-Refactoring soll dabei zur Umsetzung des Co-Refactorings eingesetzt werden [RSA10].

Die Implementierung eines Prototyps soll die Umsetzbarkeit der erarbeiteten Konzepte demonstrieren. Der Prototyp umfasst zwei Programmteile. Der erste Teil ermöglicht eine Transformation von Ontologien in Domänenmodelle, während der zweite Teil Co-Refactorings auf beiden Strukturen durchführt.

Die erarbeiteten Konzepte werden schließlich anhand einer Beispiel-Ontologie evaluiert, welche die Grundlage für ein Softwaresystem zum Freelancer-Management darstellt. Die Arbeit beschränkt sich dabei auf die Erstellung der Ontologie. Die Erstellung des Softwaresystems und die Anbindung der Ontologie daran liegen außerhalb des Rahmens der Arbeit.

[1] `http://www.most-project.eu`

1.3 Gliederung

Die vorliegend Arbeit gliedert sich in neun Kapitel. Im Kapitel 2 werden mit der MDSD und dem Refactoring die Grundlagen dieser Arbeit behandelt. Das Kapitel 3 stellt die Grundlagen zu Ontologien und semantischen Techniken vor. Es behandelt darüber hinaus die Ontologie-Evolution und die Unterschiede zwischen Ontologie-Evolution und Refactoring. Im Anschluss an diese theoretischen Betrachtungen werden im Kapitel 4 mit OntoMoPP und Refactory die in dieser Arbeit verwendeten Werkzeuge vorgestellt. Das Kapitel 5 stellt ausführlich den aktuellen Stand der Forschung bezüglich der Ontologie-Domänenmodell-Abbildung, der Ontologie-Evolution und der Co-Evolution dar. Der Schwerpunkt dieser Arbeit, das Konzept zur Entwicklung und Evolution ontologiegetriebener Softwaresysteme, wird in Kapitel 6 präsentiert. Das Kapitel 7 zeigt die praktische Umsetzung dieses Konzepts. Anschließend erfolgt im Kapitel 8 die Evaluation der implementierten Refactorings und Werkzeuge anhand einer Beispiel-Ontologie. Zum Schluss fasst das Kapitel 9 die Ergebnisse der Arbeit zusammen und gibt einen Ausblick auf mögliche zukünftige Entwicklungen.

2 Grundlagen

2.1 Modellgetriebene Softwareentwicklung

Die modellgetriebene Softwareentwicklung (MDSD) „ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen“ [SVEH07, S. 11]. In diesem Zusammenhang werden Softwaresysteme auf abstrakter Ebene durch formale Modelle beschrieben.

Zur Unterscheidung zwischen dem Design eines Systems und dessen tatsächlicher Umsetzung werden die Begriffe *Problemraum* und *Lösungsraum* verwendet. Der Problemraum ist die Zusammenfassung aller Ideen, die beschreiben, welches Problem vom Softwaresystem zu lösen ist. Dazu werden die Begriffe der Anwendungsdomäne verwendet. Mit Bezug auf das Beispiel der Unternehmensanwendung aus der Einleitung werden im Problemraum die Abläufe im Unternehmen beschrieben. Es wird beispielsweise festgehalten, welche Daten zu Kunden des Unternehmens gespeichert werden sollen, welche Daten zu Produkten gespeichert werden sollen und welche Beziehungen zwischen Kunden und Produkten bestehen. Der Lösungsraum umfasst hingegen die Art und Weise, wie Techniken zur Lösung des Problems eingesetzt werden. Wird ein Softwaresystem zur Lösung des Problems verwendet, beschreibt der Lösungsraum, wie Programmiersprachen, Frameworks oder andere Computerprogramme zur Anwendung kommen. In der genannten Unternehmensanwendung kann beispielsweise Java mit dem Enterprise-Application-Framework zur Abbildung der Geschäftsprozesse eingesetzt werden. Die Daten können in einer relationalen Datenbank gespeichert und Informationen über eine Weboberfläche, die ein Webframework verwendet, angezeigt werden. In der MDSD ist der Lösungsraum austauschbar, der Softwareentwickler kann sich so auf das eigentliche Problem – das Design des Systems – konzentrieren.

In der MDSD nimmt das Metamodell die Rolle des Domänenmodells ein und beschreibt damit die Struktur des Softwaresystems [SVEH07]. Auf Basis dieser Beschreibung können nun formale Modelle erstellt werden. Formal bedeutet hier, dass durch das Metamodell genau festgelegt ist, was Bestandteil eines Modells ist und was nicht. Abbildung 2.1 zeigt einen Ausschnitt der Begriffswelt der Modellierung nach Stahl et al. Danach hat ein Metamodell eine abstrakte Syntax, die definiert, welche Konzepte in Modellen vorkommen dürfen, welche Eigenschaften diese Konzepte haben und wie diese in Beziehung stehen. Die Modelle eines Metamodells können mit einer oder mehreren domänenspezifischen Sprachen (Domain Specific Language, DSL) dargestellt werden. Die DSL ist somit eine spezielle Schreibweise für Modelle des Metamodells und enthält eine konkrete Syntax sowie eine Semantik. Die konkrete Syntax beschreibt die Notation der Modelle. Das kann zum Beispiel eine textuelle oder auch grafische Repräsentation sein. Die Semantik definiert hingegen die Bedeutung der Sprachelemente und somit auch die Bedeutung der Modelle. Die DSL stellt somit eine Sprache dar, mit der Modelle einer bestimmten Domäne beschrieben werden. DSLs setzen dazu Sprachelemente ein, die spezifisch für die Domäne sind und beschränken sich auf die Elemente, mit denen die Konzepte der Domäne ausreichenden beschrieben werden können. Darin unterscheiden sie sich von General Purpose Languages (GPL) wie Java oder C++.

Eine Besonderheit der MDSD ist die automatische Codegenerierung. Sobald das Metamodell erstellt ist, kann daraus automatisch Programmcode erzeugt werden, der die Struktur

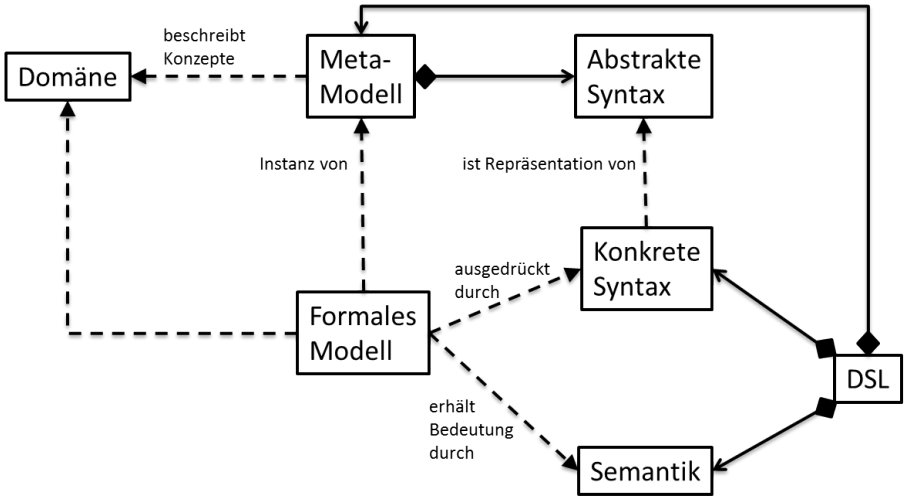

Abbildung 2.1: Begriffswelt MDSD nach [SVEH07, S. 28]

des Systems beschreibt. Dies verringert den Implementierungsaufwand und erhöht somit die Entwicklungsgeschwindigkeit. Außerdem führt dieses Vorgehen zu besserer Codequalität, da einmal geschriebener (und getesteter) Code immer wieder verwendet werden kann. Darüber hinaus kann die automatische Codegenerierung genutzt werden, um automatisch Editoren zu erzeugen, mit denen die Modelle erstellt und bearbeitet werden können.

Die Object Managmement Group (OMG)[1] hat mit der Meta Object Facility (MOF) einen Standard aufgestellt, der die Beziehungen zwischen Modellen und Metamodellen beschreibt [OMG06]. Es wird von einer Architektur mit vier Metaebenen ausgegangen, wie sie schon von Karagiannis und Kühn beschrieben wurde [KK02]. Die vier Metaebenen sind in Abbildung 2.2 dargestellt.

Die bisher besprochenen Modelle befinden sich auf Ebene M1. Sie repräsentieren einen Teil der Realität, in diesem Fall Softwaresysteme. Diese Modelle werden mit einer Modellsprache erstellt, die wiederum eine konkrete Repräsentation (d. h. eine Syntax) eines Modells auf der nächsthöheren Ebene ist. Wenn eine solche Beziehung zwischen zwei Modellen besteht, ist das Modell der oberen Metaebene das Metamodell des Modells auf der darunter liegenden Metaebene. Somit wird das Modell auf Ebene M2 zum Metamodell. Die Beziehung zwischen Metamodell und Modell ist demnach eine andere als zwischen dem Modell und den Softwareobjekten. Zwischen dem Metamodell auf der Ebene M2 und dem Meta-Metamodell auf der Ebene M3 besteht ebenfalls eine Metamodell-Beziehung. Die Abstraktionsschritte vom Modell zum Metamodell könnten in dieser Art beliebig fortgesetzt werden. In der Praxis hat sich jedoch eine Abstraktion bis zur Ebene M3 als zweckmäßig herausgestellt [KK02, S. 182]. Das Meta-Metamodell ist ausdrucksstark genug, um sich selbst beschreiben zu können. Das heißt, dass das Meta-Metamodell mit einer Repräsentation seiner selbst erstellt werden kann, woraus wiederum folgt, dass die Metamodellsprache und die Meta-Metamodellsprache gleich sind. Die MOF ist ein Beispiel für ein solches Meta-Metamodell. Die Einteilung in Modell und Metamodell ist immer in Abhängigkeit vom betrachteten Objekt zu treffen, da, je nach Betrachtungsweise, ein Modell mehreren Metaebenen zugeordnet werden kann.

Das Eclipse Modeling Framework (EMF)[2] ist ein Framework zur Anwendung des MDSD-Ansatzes im Eclipse-Umfeld [SBPM09]. Das EMF baut auf Essential MOF (EMOF) auf, welches die wichtigsten Konzepte der MOF vereint und bietet mit Ecore eine Implementierung

[1] `http://www.omg.org/`
[2] `http://www.eclipse.org/modeling/emf/`

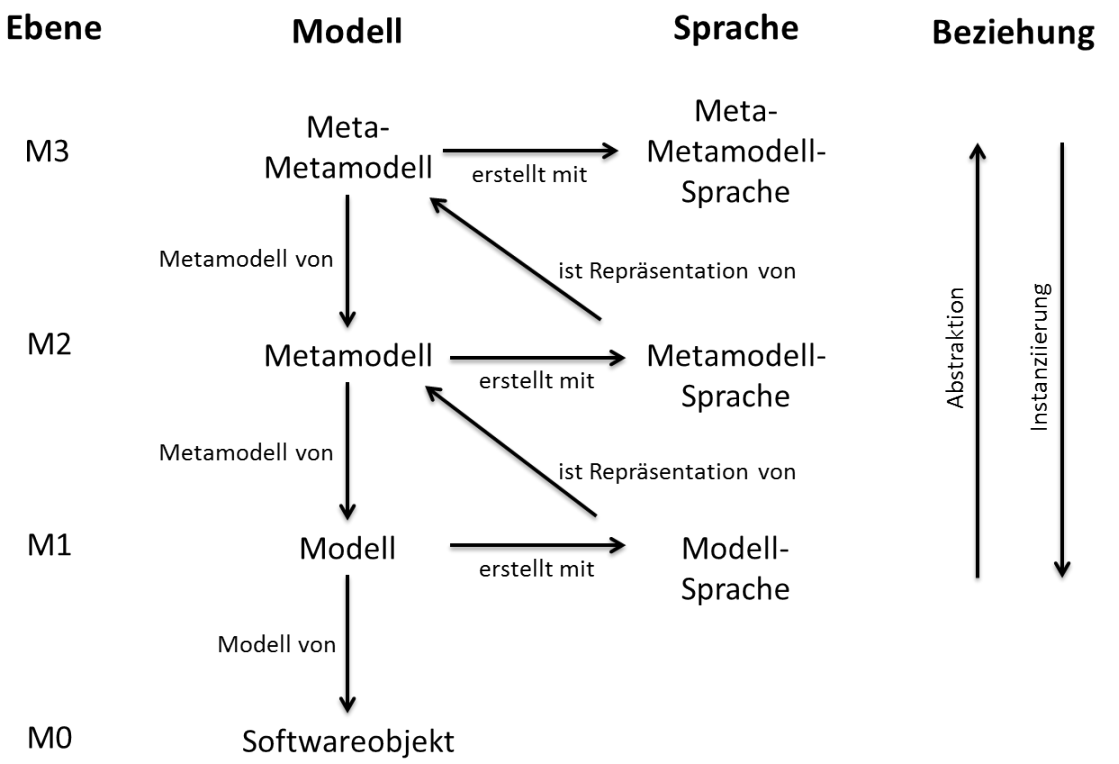

Abbildung 2.2: Die vier Metaebenen nach [KK02, S. 182]

dieses Meta-Metamodells. Mit EMF können Metamodelle auf unterschiedliche Weise entworfen werden. Es ist möglich, Metamodelle aus Java-Interfaces, von XML-Schemas oder von UML-Diagrammen abzuleiten. Alternativ können Metamodelle auch direkt mit Ecore modelliert werden. Anschließend kann auf Basis der Metamodelle ausführbarer Programmcode generiert werden.

In dieser Arbeit kommt EMF zum Einsatz, da hier das Domänenmodell explizit in Form der Metamodelle vorliegt. Somit kann die Domänenbeschreibung, die in Form einer Ontologie vorliegt, auf diese explizite Darstellung abgebildet werden. Aus der Domänenbeschreibung in Form der Ontologie wird so, über den Zwischenschritt der EMF-Metamodelle, ein Softwaresystem erzeugt.

2.2 Refactoring

Schon in den 1980er Jahren entdeckten Ward Cunningham und Kent Beck die Bedeutung der Umstrukturierung von Programmcode während sie mit Smalltalk arbeiteten [FBB+99]. Die Idee, die Struktur des Programmcodes und somit das Design des Softwaresystems kontinuierlich zu verbessern, ist allerdings schon wesentlich älter. Harlan D. Mills schlägt schon 1971 vor, Softwaresysteme inkrementell zu entwickeln [Mil71]. Das heißt, zu Beginn der Entwicklung wird ein Design des Gesamtsystems aufgestellt. Dieses wird aber während der Entwicklung kontinuierlich weiterentwickelt. Programmcode und Systemdesign werden also beständig umstrukturiert. Dieser Ansatz resultiert daraus, dass der erste Designentwurf kaum korrekt und vollständig sein kann, insbesondere für große Systeme [Bro95]. Der Begriff *Refactoring* wurde erstmals 1990 von Opdyke und Johnson eingeführt [OJ90]. Opdyke beschäftigte sich anschließend in seiner Doktorarbeit mit der genaueren Betrachtung von Refactorings als Umstruktu-

rierungen von objektorientierten Frameworks zur besseren Wiederverwendung [Opd92]. Weitreichende Verbreitung fand Refactoring mit dem 1999 veröffentlichten Buch „Refactoring“ von Martin Fowler et al. [FBB+99]. In diesem Buch hat Fowler einen Katalog von 72 Refactorings zusammengetragen[3]. Jedes Refactoring hat einen Namen, eine kurze Beschreibung mit einem Klassendiagramm oder Codeausschnitt, eine Motivation, eine Schritt-für-Schritt-Anleitung und ein Beispiel.

Martin Fowler definiert Refactoring wie folgt.

> *„Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.“* [FBB+99, S. xvi]

Es ist zu beachten, dass Refactoring aus drei Blickwinkeln betrachtet werden kann. Refactoring ist (1) der Prozess der Umstrukturierung eines Softwaresystems zur Verbesserung der „internen Struktur“, (2) die Änderung am Programmcode und (3) die Tätigkeit der Umstrukturierung. Wie bereits geschehen, wird auch weiterhin für die ersten beiden Einsatzzwecke der Begriff Refactoring verwendet. Aus dem Kontext sollte immer ersichtlich sein, ob der Prozess oder die Änderung gemeint ist. Als Verb zur Bezeichnung der Tätigkeit wird in dieser Arbeit *refaktorieren* verwendet.

Fowler sagt, ein Refactoring dürfe das sichtbare Verhalten („external behavior“ oder „observable behavior“) der Software nicht ändern. Er definiert aber nicht exakt, was unter Verhalten zu verstehen ist. Fowler bezieht sich jedoch sehr häufig auf Tests, um das Verhalten eines Programms zu prüfen. Dies macht deutlich, dass die Tests eines Programms das Verhalten darstellen. Das heißt, dass die Tests die gleichen Ergebnisse vor und nach dem Refactoring liefern müssen. Opdyke hingegen sagt, dass das Programm vor und nach dem Refactoring für alle Eingaben die gleichen Ausgaben liefern muss [Opd92, S. 2]. Dafür definiert er formale Vorbedingungen, die vor Ausführung des Refactorings erfüllt sein müssen. Dies funktioniert jedoch nur für sehr einfache Refactorings, wie die Umbenennung einer Variable. Komplexe Refactorings, die sich aus mehreren Einzelschritten zusammensetzen, erfordern Test, um zu prüfen, ob die Ausführung des Refactorings das Verhalten des Programms nicht verändert hat [FBB+99]. In seinem Buch „Refactoring“ nimmt Fowler bereits einen starken Bezug zu Tests, mit deren Hilfe ein Softwaresystem inkrementell entwickelt wird. Dieser Ansatz wurde später wesentlich durch Kent Beck zur Testgetriebenen Entwicklung (Test-Driven Development, TDD) weiterentwickelt [Bec03]. Der Abschnitt 7.2 geht näher auf diesen Ansatz der Softwareentwicklung ein.

Betrachten wir ein Beispiel, um zu verstehen, wie ein Refactoring funktioniert. *Replace Data Value with Object* ist eines der Refactorings aus dem Katalog von Fowler. Wir gehen davon aus, dass unser Softwaresystem eine Klasse hat, die eine Bestellung repräsentiert (*Order*). Jede Bestellung hat einen Kunden, von dem zu Beginn lediglich der Name als String gespeichert wird (Listing 2.1). Die Absicht des Refactorings besteht nun darin, diesen String durch ein vollwertiges Objekt zu ersetzten, um zusätzliche Informationen zum Kunden abspeichern zu können. Dazu wird eine neue Klasse angelegt, die den Kunden repräsentiert (*Customer*). In der Order-Klasse wird anschließend der String durch das Customer-Objekt ersetzt und die Get- und Set-Methode sowie der Konstruktor werden entsprechend angepasst (Listing 2.2). Die Tests in Listing 2.3 zeigen, dass das Refactoring erfolgreich ausgeführt wurde. Das Customer-Objekt ist in diesem Zustand ein reines *Daten-Objekt*, das heißt, jede Bestellung hat ihren

[3] Aktualisierungen und Erweiterungen dieses Katalogs können unter `http://www.refactoring.com/` abgerufen werden.

Listing 2.1: *Replace Data Value with Object* vor dem Refactoring

```
public class Order {

  private String customer;

  public Order(String customer) {
    this.customer = customer;
  }

  public String getCustomer() {
    return customer;
  }

  public void setCustomer(String customer) {
    this.customer = customer;
  }
}
```

Listing 2.2: *Replace Data Value with Object* nach dem Refactoring

```
public class Customer {

  private String name;

  public Customer(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
public class Order {

  private Customer customer;

  public Order(String customer) {
    this.customer = new Customer(customer);
  }

  public String getCustomer() {
    return customer.getName();
  }

  public void setCustomer(String customer) {
    this.customer = new Customer(customer);
  }
}
```

Listing 2.3: Tests für das *Replace Data Value with Object*-Refactoring

```
public class CustomerTest {

  @Test
  public void testOrder() {
    Order myOrder = new Order("John");
    assertEquals("John", myOrder.getCustomer());
  }

  @Test
  public void testSetCustomer() {
    Order myOrder = new Order("");
    myOrder.setCustomer("John");
    assertEquals("John", myOrder.getCustomer());
  }
}
```

eigenen Kunden. Um es in ein *Referenz-Objekt* zu verwandeln, kann im Anschluss ein weiteres Refactoring angewandt werden - *Change Value to Reference*. Der gezeigte Programmcode ist an das Beispiel von Fowler zu diesem Refactoring angelehnt [FBB+99, S. 176].

Nun stellt sich nur noch die Frage, in welcher Situation welches Refactoring ausgeführt werden sollte. Um diese Frage zu beantworten, haben Fowler und Beck das Prinzip der „Bad Smells“ eingeführt [FBB+99]. Ein „Bad Smell“ beschreibt Strukturen, die Indikatoren dafür sein können, dass bestimmte Refactorings durchzuführen sind. Darüber hinaus wird eine Empfehlung gegeben, welche Refactorings in Frage kommen. Einer der wichtigsten „Bad Smells“ ist *Duplicated Code*. Er tritt immer dann auf, wenn derselbe oder sehr ähnlicher Programmcode an mehreren Stellen im System vorkommt. Eine derartige Redundanz kann beispielsweise entfernt werden, indem der sich wiederholende Programmcode in eine Methode ausgelagert wird, die an allen benötigten Stellen aufgerufen wird. Robert C. Martin hat in seinem Buch „Clean Code“ das Prinzip der „Smells“ aufgegriffen und weiterentwickelt [Mar09].

Refactoring kann also zur inkrementellen Entwicklung von Software eingesetzt werden. Daraus ergibt sich eine Reihe von Vorteilen. Der Wichtigste ist ein *sauberer Programmcode* („Clean Code“). Einen sauberen Programmcode zu schreiben, war die ursprüngliche Motivation für Refactoring und ist immer noch das wichtigste Ergebnis. Ein Programmcode ist dann sauber, wenn er einfach verständlich, schlicht, direkt und seine Absicht sofort zu erkennen ist [Mar09]. Obwohl Refactoring während der Entwicklung zusätzliche Zeit in Anspruch nimmt, führt es im Endeffekt zu einer kürzeren Entwicklungszeit und einer einfacheren Wartung der erstellten Software [MAP+08, S. 263]. Dies resultiert vor allem aus einer ständigen Verbesserung des Designs, was zur Folge hat, dass der Programmcode einfacher zu verstehen ist und Fehler in der Software reduziert werden [FBB+99].

Die ersten Ansätze des Refactorings wurden für Smalltalk entwickelt und ein Großteil der heutigen Dokumentation zu Refactoring basiert auf Java oder C++. Obwohl ursprünglich für objektorientierte Programmiersprachen entwickelt, ist Refactoring jedoch nicht auf diese beschränkt. Im Rahmen der modellgetriebenen Softwareentwicklung kommt der Bedarf auf, nicht nur Programmcode, sondern auch Modelle zu refaktorieren. Dies führte zur Entwicklung des *Modell-Refactorings* [RSA10, MMBJ09], in dem Modelle umstrukturiert werden, um deren Struktur zu verbessern.

3 Ontologien und semantische Techniken

3.1 Grundlagen semantischer Techniken

2001 erkannten Tim Berners-Lee et al., dass nahezu alle Informationen, die im World Wide Web gespeichert sind, nur für Menschen zugänglich und nicht für die automatische Verarbeitung geeignet sind [BLHL01]. Berners-Lee et al. entwickelten die Idee, die Bedeutung von Daten im Web explizit anzugeben und führten dafür den Begriff *Semantic Web* ein. Das Ziel hinter dieser Idee ist, dass Computerprogramme (so genannte *Software-Agenten*) Informationen im Web „verstehen“ und so miteinander in Beziehung stehende Daten als zusammengehörig erkennen können. Als Beispiel kann man sich einen Software-Agenten zur Reiseplanung vorstellen. Der Benutzer gibt lediglich den Zeitraum und das gewünschte Reiseziel ein und der Software-Agent sucht automatisch alle nötigen Informationen aus dem Web zusammen, wie zum Beispiel Reiseverbindungen, Unterkünfte, Sehenswürdigkeiten. Alle diese Informationen können mit einer herkömmlichen Textsuche nicht gefunden oder in Beziehung gesetzt werden. Ein „Gästehaus“ wird als Unterkunft nicht gefunden, wenn man nach einem „Hotel“ sucht, einfach deshalb, weil es sich um zwei unterschiedliche Suchbegriffe handelt. Im Semantic Web hingegen werden allen Daten zusätzliche maschinenlesbare Informationen hinzugefügt, die diese Daten und deren Beziehungen zu anderen Daten näher beschreiben. So könnte zu einer Stadt gespeichert werden, welche Unterkünfte es in dieser Stadt gibt und zu einem Hotel oder einem Gästehaus könnte gespeichert sein, dass beides eine Unterkunft ist. Wenn diese Informationen maschinenlesbar gespeichert sind, kann ein Software-Agent logische Schlüsse aus Daten ziehen, für die bislang menschliches Verständnis nötig war. Die Fähigkeit von Computerprogrammen, die Bedeutung von Daten zu verstehen, hat zu dem Namen Semantic Web geführt. Die Vision von Berners-Lee et al. ist nun, nahezu alle Dinge der realen Welt und alle Daten im Web mit derartigen zusätzlichen Informationen zu versehen. Dieser Ansatz wird auch als das *Netz der Dinge* bezeichnet wird [Con10].

Betrachten wir nun die technischen Grundlagen des Semantic Web. Die Abbildung 3.1 zeigt die vom World Wide Web Consortium (W3C)[1] entworfene Schichtenarchitektur des Semantic Web [KM01]. Im Folgenden werden die unteren drei Schichten – URIs, XML und RDF – besprochen. Im Abschnitt 3.2 werden die darauf aufbauenden Konzepte RDF Schema, Ontologien und Logik behandelt. Alle weiteren Konzepte der Architektur sind im Rahmen dieser Arbeit nicht von Belang.

In der untersten Schicht befindet sich der *Uniform Resource Identifier* (URI) als Basis der Semantic Web-Architektur. Der URI ist eine Verallgemeinerung des durch das Web bekanntgewordenen *Uniform Resource Locator* (URL). Beides sind Zeichenketten, doch während der URL eine bestimmte Internetseite bezeichnet, kann mit einem URI jedes beliebige Objekt der wirklichen oder virtuellen Welt referenziert werden. Ein wesentliches Konzept des Semantic Web ist, dass jedes einzelne dieser Objekte eindeutig identifiziert werden kann. Der URI bietet genau diese Möglichkeit [BLHL01]. In Abbildung 3.2 ist das Namensschema eines URI zu sehen, wie es vom W3C im RFC 3986 definiert wird [BLFM05]. Der URI enthält fünf Teile. Das *Schema* definiert das Protokoll, über das die angefragte Ressource geladen wird und ist der Teil vor dem ersten Doppelpunkt. Beispiele für Protokolle sind „http“, „ftp“ und „doi“.

[1] `http://www.w3.org`

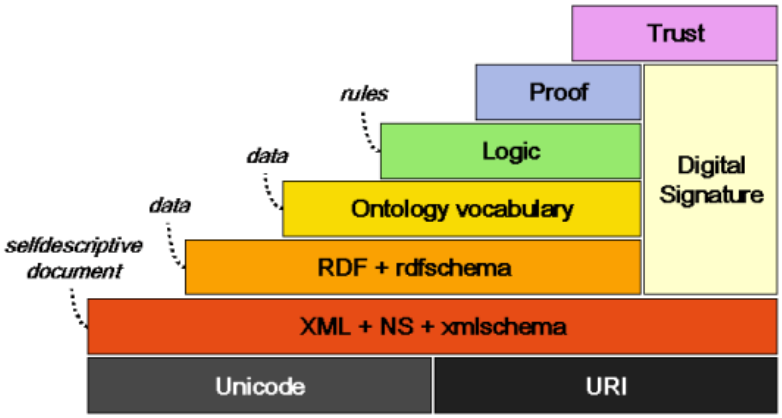

Abbildung 3.1: Schichtenarchitektur des Semantik Web aus [KM01]

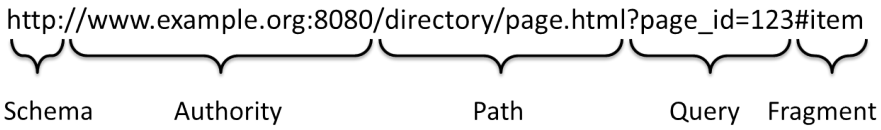

Abbildung 3.2: URI-Schema nach dem RFC 3986

Als nächstes folgt, falls vorhanden, die *Authority* beginnend mit zwei Schrägstrichen. Dies ist die Identifikation für den Server auf dem die Daten liegen, optional gefolgt von einem durch Doppelpunkt getrennten Port. Der *Path* beginnt mit einem Schrägstrich und bezeichnet den Verzeichnispfad und den Dateinamen. *Query* und *Fragment* sind wiederum optional und werden verwendet um Ressourcen mit bestimmten Eigenschaften abzufragen bzw. bestimmte Teile von Ressourcen zu identifizieren.

Wird der URI, wie im RFC 3987 beschrieben, in Unicode codiert, spricht man von einem *Internationalized Resource Identifier* (IRI) [DS05]. Dieser kann Zeichen aus beliebigen Zeichensätzen verwenden. Bezeichner, wie URIs und IRIs, können also eingesetzt werden, um beliebige Objekte zu identifizieren und bilden somit die Basis des Semantic Web.

In der zweiten Schicht der Semantic Web-Architektur befindet sich die *Extensible Markup Language* (XML) [BPSM$^+$08]. Diese Sprache wird zur Beschreibung von Dokumenten verwendet. Sie kann dazu eingesetzt werden, einzelne Teile eines Dokuments mit beliebigen Informationen zu versehen. Eine Besonderheit der XML sind die Namensräume, mit denen der Gültigkeitsbereich von Bezeichnern angegeben wird. Somit ist bei Reduzierung der URIs auf das Fragment eine eindeutige Bezeichnung von Ressourcen auch dann noch möglich, wenn es mehrere Ressourcen mit dem gleichen Fragment aus unterschiedlichen Quellen gibt.

Die erste tatsächliche Erweiterung der bisher besprochenen Technologien zum Semantic Web stellt das *Resource Description Framework* (RDF) dar, das sich in der dritten Schicht der Semantic Web-Architektur befindet [MM04]. Das RDF nutzt eine Tripel-Syntax, um beliebige Dinge näher zu beschreiben. Hierbei kann es sich um Personen oder Objekte der realen Welt aber auch um Ideen oder elektronische Dokumente handeln. Jedes Tripel besteht aus Subjekt, Prädikat und Objekt [Hor08]. Das *Subjekt* ist das Element, welches durch das Tripel näher beschrieben wird. Das *Prädikat* stellt dabei eine Beziehung des Subjekts zu einem anderen Element oder zu einem Wert dar. Beide werden durch einen IRI eindeutig identifiziert. Das *Objekt* schließlich ist das Ziel der Beziehung. Dies kann ein anderes, durch einen IRI identifiziertes Element oder ein Literal sein. Ein Literal ist ein atomarer Wert, wie eine Zahl oder eine

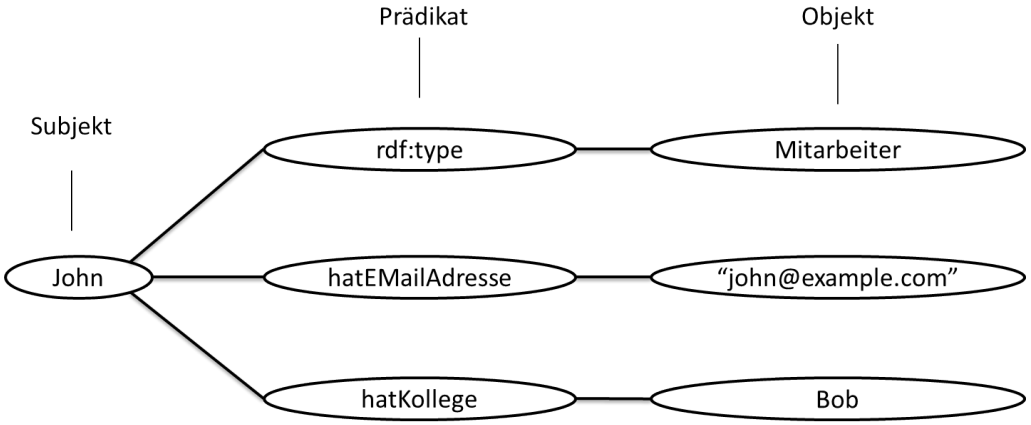

Abbildung 3.3: Beispiel eines RDF-Graphen

Zeichenkette. Mehrere RDF-Tripel können so Eigenschaften und Beziehungen vieler Elemente beschreiben und bilden dadurch einen *RDF-Graphen*. Die Abbildung 3.3 zeigt ein Beispiel eines solchen RDF-Graphen, in dem das Subjekt *John* durch drei Tripel näher beschrieben wird.

Das Prädikat *rdf:type* sagt aus, dass John „vom Typ“ Mitarbeiter ist. Die Zeichenkette `rdf:` ist hierbei eine Abkürzung für den Namesraum `http://www.w3.org/1999/02/22-rdf-syntax-ns#`. Das Objekt des Prädikats *hatEMailAdresse* ist ein einfaches Literal, wohingegen das Prädikat *hatKollege* auf Bob, also ein anderes Element, zeigt. Sämtliche Namen von Elementen im Diagramm sind lediglich Abkürzungen für vollständige IRIs. Das W3C hat mit *RDF/XML* eine XML-Syntax zur Speicherung und zum Austausch von RDF-Tripeln definiert [Bec04]. Das Listing 3.1 zeigt das Beispiel aus Abbildung 3.3 in dieser RDF/XML-Syntax. Zur Vereinfachung der Schreibweise hat das W3C außerdem die *Turtle*-Syntax eingeführt [BBL08]. Diese Syntax ging aus der von Berners-Lee entwickelten *Notation 3* (N3) hervor [HKRS08, S. 40] und basiert auf einer Tripel-Darstellung, wie das Listing 3.2 zeigt.

Listing 3.1: RDF/XML-Syntax für den RDF-Graphen aus Abbildung 3.3

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://www.example.com/rdfDoc#"
     xml:base="http://www.example.com/rdfDoc"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">

    <rdf:Description rdf:about="http://www.example.com/rdfDoc#Mitarbeiter"/>
    <rdf:Description rdf:about="http://www.example.com/rdfDoc#hatKollege"/>
    <rdf:Description rdf:about="http://www.example.com/rdfDoc#hatEMailAdresse"/>

    <rdf:Description rdf:about="http://www.example.com/rdfDoc#Bob"/>

    <rdf:Description rdf:about="http://www.example.com/rdfDoc#John">
        <rdf:type rdf:resource="http://www.example.com/rdfDoc#Mitarbeiter"/>
        <hatEMailAdresse>john@example.com</hatEMailAdresse>
        <hatKollege rdf:resource="http://www.example.com/rdfDoc#Bob"/>
    </rdf:Description>
</rdf:RDF>
```

Listing 3.2: Turtle-Syntax für den RDF-Graphen aus Abbildung 3.3

```
@prefix ex: <http://www.example.com/rdfDoc#>
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

  <ex:John> <rdf:type>           <ex:Mitarbeiter> .
  <ex:John> <ex:hatKollege>      <ex:Bob> .
  <ex:John> <ex:hatEMailAdresse> "john@example.com" .
```

3.2 Ontologien

3.2.1 Definition und Begriffe

Der Begriff *Ontologie* kommt ursprünglich aus der Philosophie. Dort bezeichnet er die Lehre vom Sein der Dinge [Yil06]. Die erste wissenschaftliche Definition von Ontologien im Bereich der Informatik wird 1993 von Gruber angegeben.

> *„An ontology is an explicit specification of a conceptualization."* [Gru93, S. 2]

Durch den Artikel „The Semantic Web" von Berners-Lee et al. hat dieser Begriff auch in der Informatik eine weite Verbreitung gefunden [BLHL01]. Nach Berners-Lee et al. bezeichnet eine Ontologie ein Begriffsnetz, das alle Konzepte einer Anwendungsdomäne und ihre Beziehungen untereinander klar definiert. Die Ontologie stellt damit ein gemeinsames Vokabular von Begriffen und deren Bedeutung dar. Als Ontologie wird weiterhin das Software-Artefakt bezeichnet, in der dieses Vokabular gespeichert wird.

Die wichtigsten Elemente zur Modellierung von Wissen in einer Ontologie sind Klassen, Propertys und Individuen [Hor08]. Abbildung 3.4 zeigt diese Elemente anhand einer kleinen Beispiel-Ontologie. Eine *Klasse* ist eine Menge von Individuen mit bestimmten Eigenschaften. Das Konzept einer Klasse ist also vergleichbar mit dem mathematischen Konzept der Menge. Ein *Individuum* ist demnach ein konkretes Element, das zu einer Klasse gehören kann aber nicht gehören muss. Die Unterscheidung in Klasse und Individuum ist dabei keineswegs eindeutig. So können beispielsweise Länder als Klassen dargestellt werden, die Ortschaften als Individuen enthalten oder selbst Individuen der Klasse *Geografische Region* sein. Als Faustregel gilt, dass Individuen diejenigen Elemente einer Ontologie sind, welche selbst keine weiteren Elemente enthalten [NM01]. Sowohl Beziehungen zwischen Klassen als auch zwischen Individuen werden durch *Propertys* angegeben. Propertys entsprechen somit den Prädikaten des RDF. Es wird dabei zwischen *ObjectPropertys*, die Beziehungen zu anderen Elementen anzeigen und *DataPropertys*, die Beziehungen zu Literalen anzeigen, unterschieden. Die Menge aller Klassen und deren Beziehungen untereinander beschreibt die Struktur der Ontologie und wird auch als *Terminology Box* (TBox) bezeichnet. Die Menge der Individuen und deren Beziehungen stellt den Datenbestand der Ontologie dar und wird als *Assertion Box* (ABox) bezeichnet. Gemeinsam ergeben TBox und ABox die so genannte *Knowledge Base* [Hor08].

Um Ontologien beschreiben zu können, wurde das RDF zum *RDF Schema* (RDFS) weiterentwickelt [BG04]. Das RDFS definiert eine Reihe von Prädikaten mit besonderer Bedeutung, wodurch es möglich wird, mit RDF-Tripeln Ontologien zu beschreiben. Beispiele für solche Prädikate sind `rdfs:Class`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain` und `rdfs:range`. Durch derartige Prädikate können Klassen- und Property-Hierarchien sowie Definitions- und Wertebereich von Propertys definiert werden. Im vorherigen Abschnitt wurde zum Beispiel durch das RDF-Prädikat `rdf:type` angegeben, dass *John* „vom Typ" *Mitarbeiter* ist. *Mitarbeiter* ist jedoch ein beliebiges Element im RDF-Graphen. Erst durch die RDFS-Definition `rdfs:Class` wird *Mitarbeiter* zu einer Klasse.

3.2.2 Die Web Ontology Language (OWL)

Die heute geläufigste Beschreibungssprache für Ontologien ist die *Web Ontology Language* (OWL[2]) [MvH04]. Sie setzt auf dem RDFS auf und erweitert es um zusätzliche Konzepte [Hor08]. So ist es beispielsweise möglich, Klassen auf Basis bestehender Klassen und logischer Mengenoperationen zu definieren. Die Klasse der Java-Experten kann zum Beispiel als

[2] Die Abkürzung ist tatsächlich OWL und nicht WOL, da die Aussprache von OWL klar ist, und zwar wie beim englischen Wort für Eule: /aʊl/. [HKRS08, S. 125]

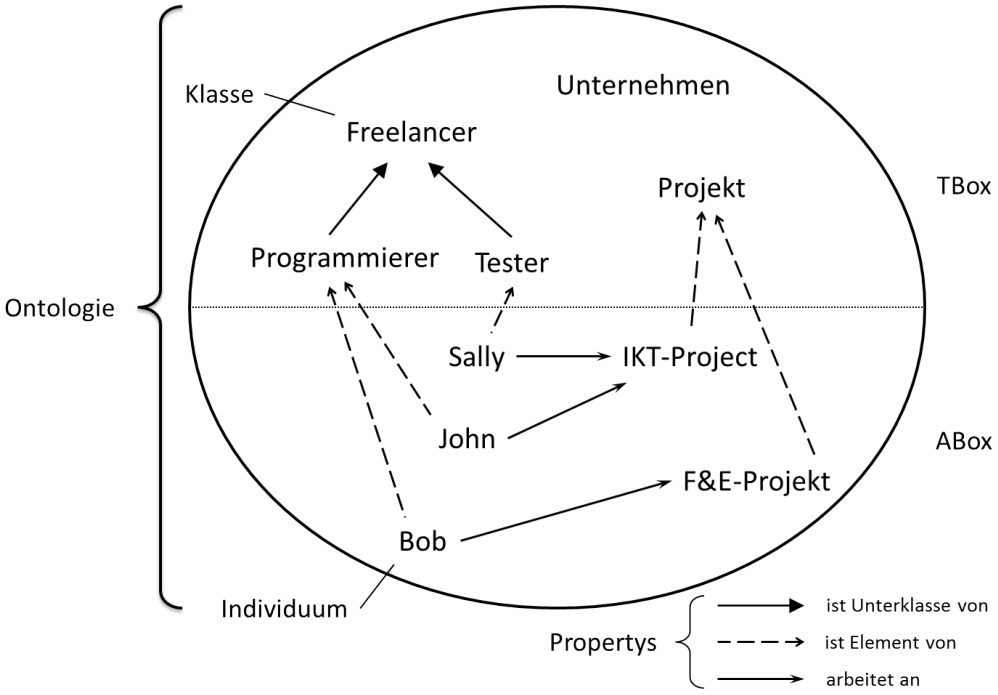

Abbildung 3.4: Beispiel einer Ontologie

Schnittmenge der Menge aller Personen, die mit Java arbeiten und der Menge aller Personen, die mindestens fünf Jahre Berufserfahrung haben, beschrieben werden.

Eine Klasse, die mit einer IRI eindeutig beschrieben ist, wird als *benannte Klasse* bezeichnet. In OWL 2 ist es jedoch auch möglich, *anonyme Klassen* mit *Klassen-Ausdrücken* zu definieren [MPSP09]. Die Klassen werden hierbei nicht durch einen Namen, sondern durch eine Beschreibung der Eigenschaften aller zur Klasse gehörenden Individuen definiert. Dies erfolgt zum Beispiel durch eine Aufzählung aller zulässigen Individuen, durch Einschränkungen von Eigenschaften, oder durch logische Mengenoperationen, wie Vereinigung, Schnittmenge oder Komplement.

Zusätzliche Eigenschaften von Klassen und Propertys werden über *Axiome* angegeben. Mit dem SubClassOf-Axiom wird beispielsweise die Unterklassen-Beziehungen zwischen zwei Klassen definiert. Domain- und Range-Axiome geben die Klasse von Quelle bzw. Ziel einer Property an. Individuen werden mit speziellen Axiomen, so genannten *Aussagen* (engl. assertions), näher beschrieben. Diese Aussagen geben an, zu welcher Klasse ein Individuum gehört und welche Beziehungen es zu anderen Individuen hat. Die OWL 2-Spezifikation listet alle zur Verfügung stehenden Axiome und Aussagen mit Erklärungen und Beispielen auf [MPSP09].

Des Weiteren können mit OWL *Einschränkungen* (engl. restrictions) von Klassen und Propertys vorgenommen werden. Die Klasse aller Arbeitnehmer kann beispielsweise definiert werden, indem die Klasse aller Personen auf diejenigen Personen eingeschränkt wird, welche einen Arbeitgeber haben. Mit Einschränkungen kann auch die Anzahl der Propertys für alle Individuen einer Klasse angegeben werden. So kann beispielsweise definiert werden, dass jeder Kunde exakt zwei Propertys *hatAdresse* besitzt, die ihn mit seiner Rechnungs- und seiner Lieferadresse verbinden. Die angegebene Anzahl wird auch als Kardinalität bezeichnet, da sie angibt, wie groß die Menge der Individuen ist, zu denen eine Beziehung besteht. Die Einschränkung ist dabei selbst eine anonyme Klasse. Je nach dem in welcher Beziehung die Einschränkungen mit den benannten Klassen stehen, werden die benannten Klassen in primitive und definierte

Klassen eingeteilt [Hor11, S. 53]. *Primitive Klassen* sind Unterklassen der Einschränkungen. Die Einschränkungen stellen somit notwendige Bedingungen dar. Das heißt, ein Individuum muss diese Einschränkung erfüllen, damit es zur Klasse gehört, es kann aber auch noch andere Individuen mit diesen Einschränkungen geben, die nicht zur Klasse gehören. *Definierte Klassen* sind dagegen äquivalent zu den Einschränkungen. Die Einschränkungen sind also notwendige und hinreichende Bedingungen. Somit enthält die Klasse alle Individuen mit diesen Einschränkungen.

Die OWL Working Group des W3C[3] hat im Oktober 2009 die neue Version *OWL 2* (früher unter dem Namen OWL 1.1 bekannt) veröffentlicht [W3C09]. OWL 2 erweitert den bisherigen OWL-Standard um zusätzliche Ausdrucksmittel [Hor08]. Dazu gehört die Möglichkeit, numerische Klassen-Einschränkungen mit „qualifizierten" Zielen anzugeben (engl. qualified number restrictions). Damit ist es beispielsweise möglich, eine Hand als etwas zu beschreiben, das vier Finger und einen Daumen hat. Die Propertys, welche die Beziehungen in Ontologien darstellen, können in OWL 2 mit weiteren Attributen, wie reflexiv, transitiv oder symmetrisch näher beschrieben werden. Eine transitive Property, die Beziehungen zwischen geografische Regionen herstellt, ist zum Beispiel *befindetSichIn*. Darüber hinaus führt OWL 2 Datentypen und Annotationen ein.

An dieser Stelle sei noch auf einige Besonderheiten von Ontologien hingewiesen. RDF-Graphen und somit auch Ontologien gehen von der „*Annahme nicht-eindeutiger Namen*" (engl. non-unique name assumption) aus [HKRS08, S. 65]. Das bedeutet, dass zwei unterschiedliche Bezeichner (IRIs) denselben Gegenstand bezeichnen können, solange nicht explizit angegeben ist, dass beide Bezeichner unterschiedliche Dinge bezeichnen. Ein wesentlicher Unterschied zwischen Ontologien und anderen Speichertechnologien, wie Datenbanken, ist die „*Offene-Welt-Annahme*" (engl. open-world assumption, OWA) [HKRS08, S. 145]. In Ontologien geht man immer davon aus, dass die vorliegenden Daten unvollständig sind und behandelt daher alle nicht vorliegenden Informationen als ungewiss, anstatt als falsch. Stellen wir uns vor, in einer Ontologie ist ein Mitarbeiter John gespeichert, zu dem keine weiteren Informationen vorliegen. Mit einer Klasse aller Mitarbeiter, die an keinem Projekt arbeiten, soll nun festgestellt werden, welche Mitarbeiter neuen Projekten zugewiesen werden können. Allerdings erscheint John nicht als Individuum dieser Klasse. Es könnte sein, dass John zwar an einem Projekt arbeitet, jedoch die Information darüber nicht vorliegt. Damit abgeleitet werden kann, dass John ein Individuum der Klasse ist, muss also explizit angegeben werden, das John an keinem Projekt arbeitet. Die OWA bedingt weiterhin, dass sich Klassen überlappen können, solange nicht angegeben ist, dass sie disjunkt sind. Im Gegensatz zur OWA funktionieren Datenbanken nach der „*Geschlossenen-Welt-Annahme*" (engl. closed-world assumption, CWA). Das heißt, dass alle nicht vorliegenden Informationen als falsch behandelt werden. Steht der Mitarbeiter mit der Nummer 1234 nicht in der Datenbank, gibt es ihn nicht.

3.2.3 Ontologie-Syntaxen

Im vorherigen Abschnitt wurden bereits die RDF/XML-Syntax und die Turtle-Syntax zur Speicherung von RDF-Tripeln vorgestellt. Diese Syntaxen eignen sich ebenfalls zur Speicherung von Ontologien. Die RDF/XML-Syntax wurde vom W3C zur Standard-Syntax für Ontologien ernannt [W3C09]. Darüber hinaus gibt es eine Reihe weiterer Syntaxen, in denen Ontologien gespeichert werden können. Die *OWL/XML*-Syntax folgt wie die RDF/XML-Syntax dem XML-Standard, verwendet aber spezielle Ausdrucksmittel für Konzepte des OWL-Standards [MPPS09]. Die *Functional Syntax* ist eine sehr kompakte Syntax, die sich an der Schreibweise der Prädikatenlogik orientiert [MPSP09]. Die OWL/XML- und Functional Syntax sind

[3] `http://www.w3.org/2007/OWL/`

Axiom-basiert. Das bedeutet, dass alle Axiome der Ontologie als eigenständige Konzepte geschrieben werden. Die Aussage, dass John zur Klasse der Mitarbeiter gehört, ist unabhängig von der Definition des Individuums John. Die *Manchester OWL Syntax* hingegen ist Frame-basiert und wurde mit dem Ziel erschaffen, die Ontologie für den Menschen möglichst einfach lesbar darzustellen [HPS09]. Die Manchester OWL Syntax schreibt dazu alle Axiome, die ein bestimmtes Element betreffen in einen Frame. Die Listings 3.3, 3.4, 3.5 und 3.6 stellen die unterschiedlichen Syntaxen für eine Ontologie, die dem RDF-Graphen aus Abbildung 3.3 basiert, gegenüber.

Listing 3.3: RDF/XML-Syntax für eine Ontologie basierend auf dem RDF-Graphen aus Abbildung 3.3

```
<?xml version="1.0"?>
<rdf:RDF xmlns="http://ontoref.org/ontoTest#"
     xml:base="http://ontoref.org/ontoTest"
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:owl="http://www.w3.org/2002/07/owl#">

    <owl:Ontology rdf:about="http://ontoref.org/ontoTest"/>

    <owl:Class rdf:about="http://ontoref.org/ontoTest#Mitarbeiter"/>
    <owl:ObjectProperty rdf:about="http://ontoref.org/ontoTest#hatKollege"/>
    <owl:DataProperty rdf:about="http://ontoref.org/ontoTest#hatEMailAdresse"/>

    <owl:NamedIndividual rdf:about="http://ontoref.org/ontoTest#Bob"/>

    <owl:NamedIndividual rdf:about="http://ontoref.org/ontoTest#John">
        <rdf:type rdf:resource="http://ontoref.org/ontoTest#Mitarbeiter"/>
        <hatEMailAdresse>john@example.com</hatEMailAdresse>
        <hatKollege rdf:resource="http://ontoref.org/ontoTest#Bob"/>
    </owl:NamedIndividual>
</rdf:RDF>
```

Listing 3.4: OWL/XML-Syntax für eine Ontologie basierend auf dem RDF-Graphen aus Abbildung 3.3

```
<?xml version="1.0"?>
<!DOCTYPE Ontology [
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>
<Ontology xmlns="http://www.w3.org/2002/07/owl#"
     xml:base="http://ontoref.org/ontoTest"
     ontologyIRI="http://ontoref.org/ontoTest">
    <Prefix name="" IRI="http://ontoref.org/ontoTest#"/>

    <Declaration>
        <Class IRI="#Mitarbeiter"/>
    </Declaration>
    <Declaration>
        <ObjectProperty IRI="#hatKollege"/>
    </Declaration>
    <Declaration>
        <DataProperty IRI="#hatEMailAdresse"/>
    </Declaration>
    <Declaration>
        <NamedIndividual IRI="#Bob"/>
    </Declaration>
    <Declaration>
        <NamedIndividual IRI="#John"/>
    </Declaration>
    <ClassAssertion>
        <Class IRI="#Mitarbeiter"/>
        <NamedIndividual IRI="#John"/>
    </ClassAssertion>
    <ObjectPropertyAssertion>
        <ObjectProperty IRI="#hatKollege"/>
```

```
31          <NamedIndividual IRI="#John"/>
32          <NamedIndividual IRI="#Bob"/>
33      </ObjectPropertyAssertion>
34      <DataPropertyAssertion>
35          <DataProperty IRI="#hatEMailAdresse"/>
36          <NamedIndividual IRI="#John"/>
37          <Literal datatypeIRI="&rdf;PlainLiteral">john@example.com</Literal>
38      </DataPropertyAssertion>
39  </Ontology>
```

Listing 3.5: Functional Syntax für eine Ontologie basierend auf dem RDF-Graphen aus Abbildung 3.3

```
1  Prefix(:=<http://ontoref.org/ontoTest#>)
2  Ontology(<http://ontoref.org/ontoTest>
3
4  Declaration(Class(:Mitarbeiter))
5  Declaration(ObjectProperty(:hatKollege))
6  Declaration(DataProperty(:hatEMailAdresse))
7  Declaration(NamedIndividual(:Bob))
8  Declaration(NamedIndividual(:John))
9  ClassAssertion(:Mitarbeiter :John)
10 ObjectPropertyAssertion(:hatKollege :John :Bob)
11 DataPropertyAssertion(:hatEMailAdresse :John "john@example.com")
12 )
```

Listing 3.6: Manchester OWL Syntax für eine Ontologie basierend auf dem RDF-Graphen aus Abbildung 3.3

```
1  Prefix: : <http://ontoref.org/ontoTest#>
2  Ontology: <http://ontoref.org/ontoTest>
3
4  Class: Mitarbeiter
5  ObjectProperty: hatKollege
6  DataProperty: hatEMailAdresse
7
8  Individual: Bob
9
10 Individual: John
11     Types: Mitarbeiter
12     Facts:
13         hatKollege  Bob,
14         hatEMailAdresse  "john@example.com"
```

Wie wir in diesem Abschnitt gesehen haben, sind die meisten Syntaxen für Ontologien nicht besonders zugänglich für die direkte Bearbeitung. Es sind also Werkzeuge nötig, die eine grafische Benutzeroberfläche zur komfortablen Erstellung und Bearbeitung von Ontologien zur Verfügung stellen. Das Kapitel 5 geht näher auf solche Werkzeuge ein.

3.2.4 Beschreibungslogiken, Teilsprachen und Profile

Bisher wurden die Bestandteile von Ontologien und deren Darstellung behandelt. Nun soll es um die *formale Semantik* (also die formale Bedeutung) gehen. Die formale Semantik von Ontologien wird mit *Beschreibungslogiken* ausgedrückt [HKRS08, S. 163]. Eine Beschreibungslogik definiert, welche Konzepte in einer Ontologie zur Wissensmodellierung eingesetzt werden können und bestimmt damit die *Ausdruckstärke* der Ontologie[4]. Auf der Basis dieser Beschreibungslogiken kann ein Reasoner[5] logische Schlussfolgerungen über den Daten einer Ontologie ziehen. Damit ist es unter anderem möglich, die logische Klassenhierarchie aufzustellen (man

[4] Verglichen mit der MDSD legt eine Beschreibungslogik die Elemente der abstrakten Syntax der Ontologie-Beschreibungssprache fest.

[5] Für weitere Informationen über Reasoner sei auf den Artikel [DCtTdK11] von Dentler et al. verwiesen.

sagt: die Ontologie klassifizieren) und Konsistenzprüfungen durchzuführen. Die Konsistenzprüfungen prüfen dabei die semantische Konsistenz der Ontologie. Diese kann in zwei Fällen verletzt sein. Erstens, es gibt eine Klasse, die keine Individuen enthalten kann. Dies geschieht immer dann, wenn eine Klasse als die Schnittmenge zweier disjunkter Klassen definiert wird. Zweitens, ein Individuum gehört zu zwei disjunkten Klassen. Dieser Fall tritt zum Beispiel ein, wenn ein Individuum durch eine Property referenziert wird, die Klasse des Individuums aber disjunkt zum Wertebereich der Property ist. Die Klassifizierung und Konsistenzprüfung können bei großen Ontologien einige Zeit in Anspruch nehmen, je nachdem, welche Konzepte zur Beschreibung der Ontologie eingesetzt werden. Beschreibungslogiken stellen also einen Kompromiss zwischen Ausdrucksstärke und der Zeit, die für logische Schlussfolgerungen benötigt wird, dar [HKRS08, S. 163].

Die einfachste Beschreibungslogik, die in Ontologien sinnvoll zum Einsatz kommen kann, ist $\mathcal{ALC}$ (Attribute Language with Complement)[6]. Mit ihr können die Negation, Mengen und deren Komplemente, Konjunktionen, Disjunktionen und allgemeine Klassen-Einschränkungen definiert werden[7]. $\mathcal{ALC}$ ist die Basis für alle weiteren Beschreibungslogiken. OWL 2 umfasst insgesamt drei Teil-Sprachen, denen unterschiedlich mächtige Beschreibungslogiken zugrunde liegen [HKRS08]. *OWL Lite* ist die Teilsprache mit der geringsten Ausdrucksstärke, ihr liegt die Beschreibungslogik $\mathcal{SHIF}$ zugrunde. Mit dieser Teilsprache können Klassen- und Property-Hierarchien dargestellt und einfache Eigenschaften von Propertys (transitiv, funktional, invers) angegeben werden. OWL Lite dient dazu, Taxonomien in Ontologien zu überführen.

Die Teilsprache *OWL DL* enthält OWL Lite und führt eine Reihe weiterer Konzepte ein, sodass die damit beschriebenen Ontologien für die meisten Anwendungsfälle ausdrucksstark genug sind. OWL DL basiert auf der schon wesentlich mächtigeren Beschreibungslogik $\mathcal{SHOIN}(\mathcal{D})$ und wird von den meisten Ontologie-Werkzeugen unterstützt.

Der komplette Sprachumfang von OWL 2 wird auch als *OWL Full* bezeichnet. OWL Full basiert auf der Beschreibungslogik $\mathcal{SROIQ}$, bietet die größte Ausdrucksstärke, ist aber, im Gegensatz zu OWL Lite und OWL DL, nicht mehr entscheidbar. Das heißt, es gibt Situationen, in denen eine Klassifizierung oder Konsistenzprüfung nicht mehr in endlicher Zeit durchgeführt werden kann.

Abgesehen von den Teilsprachen bietet OWL seit der Version 2 auch so genannte *Profile* [MGH+09, Hor08]. Profile sind syntaktische Teilsprachen von OWL DL, mit denen logische Schlussfolgerungen in höchstens polynomieller Laufzeit, bezogen auf die Anzahl der in der Ontologie definierten Konzepte, durchgeführt werden können. Es gibt insgesamt drei Profile. *OWL 2 EL* ist für sehr große aber einfach strukturierte Ontologien gedacht. *OWL 2 QL* ist speziell für Abfragen über Ontologien mit vielen Individuen ausgelegt. *OWL 2 RL* basiert auf einer Regelsprache und stellt die größte Ausdrucksstärke der drei Profile zur Verfügung.

3.2.5 Abfragen mit SPARQL

SPARQL (SPARQL Protocol and RDF Query Language) ist eine Abfragesprache für RDF-Graphen [PS08]. SPARQL orientiert sich an der SQL-Abfragesprache für relationale Datenbanken und nutzt die Turtle-Syntax zur Definition von Abfragen. Dazu werden in einer WHERE-Klausel beliebig viele RDF-Tripel in Turtle-Syntax definiert, welche Variablen enthalten können, die wiederum in einer SELECT-Klausel zu einer Ergebnismenge zusammengefügt werden. Eine Beispiel-Anfrage an den RDF-Graphen aus Abbildung 3.3 ist in Listing 3.7 zu sehen. Das Ergebnis der Abfrage besteht aus nur einem Datensatz mit einem Wert: Bob.

[6] Für detaillierte Informationen zu Beschreibungslogiken sei auf die Website `http://www.cs.man.ac.uk/~ezolin/dl/` und auf die Arbeiten von Franz Baader et al. verwiesen [BHS05, BCM+10].

[7] In Beschreibungslogiken spricht man von *Concepts* anstelle der Klassen und von *Roles* anstelle der Propertys.

Listing 3.7: SPARQL-Abfrage an den RDF-Graphen aus Abbildung 3.3

```
1 SELECT ?kollege
2 WHERE { ex:John ex:hatKollege ?kollege }
```

Um Abfragen auch über Ontologien durchführen zu können, wird aktuell der Nachfolger *SPARQL 1.1* entwickelt. Diese neue Version ermöglicht Abfragen über der OWL DL-Teilsprache. Zum Zeitpunkt der Erstellung dieser Arbeit liegt ein erster Entwurf des W3C für SPARQL 1.1 vor [GO10]. Die Turtle-Syntax, auf der SPARQL basiert, wurde jedoch für RDF-Tripel und nicht für Ontologien entwickelt. Deshalb werden Abfragen an Ontologien schnell umfangreich, kompliziert zu schreiben und schwierig zu verstehen. Aus diesem Grund wird mit *Terp* eine neue Abfrage-Syntax entwickelt [SBS10]. Terp kombiniert die Turtle-Syntax mit der Manchester OWL Syntax, um kurze, aussagekräftige Anfragen an Ontologien stellen zu können. Einen ähnlichen Ansatz verfolgt *SPARQLAS*. Es erweitert Abfragen um OWL-Elemente (wie Klassen, Propertys oder Individuen) und wandelt diese dann in herkömmliche SPARQL-Abfragen um [Sch10b].

3.2.6 Beispiele für Ontologien

Ontologien sollen ein gemeinsames Vokabular von Begriffen definieren, das weltweit verfügbar ist. Sie haben bereits vor allem in der Medizin, Biologie und Geografie Anwendung gefunden [Hor08]. Als Beispiele können hier die SNOMED CT[8] (Systematized Nomenclature of Medicine - Clinical Terms), die GO[9] (Gene Ontology), die BioPAX[10] (Biological Pathway Exchange) und die SWEET[11] (Semantic Web for Earth and Environmental Terminology Project) genannt werden. Dies sind Beispiele für teils sehr große Verzeichnisse, die ein gemeinsames Vokabular für viele verteilte Anwender zur Verfügung stellen. Die SNOMED CT allein enthält knapp 300.000 Klassen und wird vom National Health Service (NHS) in Großbritannien landesweit in der Gesundheitsversorgung eingesetzt [Hor08].

Die genannten Beispiele sind *Domain-Ontologien* [NV04]. Das heißt, sie definieren Begriffe für eine bestimmte Anwendungsdomäne. Erstellen nun viele Anwender unabhängig voneinander Ontologien zu einer Anwendungsdomäne, kommt es schnell dazu, dass für dieselben Konzepte unterschiedliche Begriffe verwendet werden. Weiterhin gibt es auch Konzepte, die in vielen unterschiedlichen Anwendungsdomänen immer wieder vorkommen, beispielsweise das Konzept einer Person mit deren Namen und Beziehungen zu anderen Personen. Um derartige Konzepte domänenübergreifend zu definieren, wurden *Top-Level-Ontologien* (auch Foundational Ontologies oder Upper Ontologies) entwickelt [NV04]. Top-Level-Ontologien enthalten Konzepte, die in Domain-Ontologien unterschiedlicher Domänen wiederverwendet werden können. Die bekanntesten Beispiele dafür sind die Dublin Core-Ontologie[12] (DC) und die Friend-of-a-Friend-Ontologie[13] (FOAF). Die größte aktuelle Ontologie ist DBpedia[14], eine Ontologie, die ihren Datenbestand aus der Wikipedia bezieht. Weitere Informationen über öffentliche Ontologien können auf der Linked Data-Website `http://linkeddata.org/` eingesehen werden.

[8] `http://www.ihtsdo.org/snomed-ct/`
[9] `http://www.geneontology.org/`
[10] `http://www.biopax.org/`
[11] `http://sweet.jpl.nasa.gov/`
[12] `http://dublincore.org/`
[13] `http://www.foaf-project.org/`
[14] `http://dbpedia.org/`

3.3 Ontologie-Evolution und Versionierung

Ontologien sind, wie Softwaresysteme, Änderungen unterworfen. Deshalb soll an dieser Stelle die Ontologie-Evolution näher betrachtet werden. Sie stellt den Teilbereich der Ontologie-Forschung dar, der sich mit der Veränderung und Weiterentwicklung von Ontologien beschäftigt. Haase und Stojanovic definieren den Begriff der Ontologie-Evolution wie folgt.

> *„Ontology Evolution can be defined as the timely adaptation of an ontology to the arisen changes and the consistent management of these changes.“* [HS05, S. 1]

Ontologie-Evolution ist demnach eine Anpassung der Ontologie an sich ändernde Rahmenbedingungen. Dabei müssen Änderungen an der Ontologie immer so ausgeführt werden, dass die Ontologie konsistent bleibt. Das heißt, sie darf keine logischen Widersprüche aufweisen. Der Teil der Definition „consistent management of these changes“ bezieht sich weiterhin darauf, dass die Änderungen unter Umständen auch auf andere verteilte Ontologien angewendet werden. Im Bereich der Ontologie-Evolution wird von Änderungsoperationen statt Refactorings gesprochen. Diese lassen sich in elementare und komplexe Änderungsoperationen einteilen [Sto04, S. 60]. Elementare Änderungen sind dabei das Hinzufügen, Verändern oder Löschen einzelner Elemente. Während sich komplexe Änderungen aus beliebigen Kombinationen der elementaren Änderungen ergeben.

Die Umsetzung entsprechender Änderungen in der Ontologie erfordert einen Prozess mit wohldefinierten Tätigkeiten [SMMS02]. Stojanovic et al. definieren dafür einen Evolutionsprozess, der aus sechs Phasen besteht, die in Abbildung 3.5 zu sehen sind. (1) Zu Beginn werden die Anforderungen an die Ontologie analysiert und nach neuen Begriffen gesucht, die in die Ontologie aufgenommen werden sollen. Daraus werden die gewünschten Änderungen der Ontologie abgeleitet. (2) Dies können elementare oder komplexe Änderungen sein. Aus der genauen Definition der durchzuführenden Änderungen ergibt sich die Repräsentation der Änderung. (3) Als nächstes müssen die Auswirkungen einer Änderung überprüft werden. Wird beispielsweise ein Element gelöscht, das an anderer Stelle referenziert wird, ist die Ontologie (syntaktisch) inkonsistent. In diesem Fall müssen weitere Änderungen (das Löschen der Referenz) ermittelt werden, um die Konsistenz der Ontologie zu erhalten. (4) Ist die Erhaltung der Konsistenz sichergestellt, können die Änderungen tatsächlich ausgeführt werden. (5) Die Änderung der Ontologie erfordert eine entsprechende Anpassung aller abhängigen Artefakte, wie andere Ontologien oder Softwaresysteme, die auf die Ontologie zugreifen. (6) Der letzte Schritt im Evolutionsprozess besteht in der Überprüfung der Korrektheit der Änderungen.

Die im Punkt (3) erwähnte **Konsistenz einer Ontologie** hat drei Formen. Die *semantische Konsistenz* wurde bereits im Rahmen der Beschreibungslogiken erwähnt (im Abschnitt 3.2.4). Sie beschreibt die Korrektheit der Ontologie bezüglich logischer Schlussfolgerungen. Das Beispiel aus diesem Abschnitt beschreibt die *syntaktische Konsistenz*. Sie stellt sicher, dass die Ontologie konform zur abstrakten und konkreten Syntax der verwendeten Beschreibungssprache ist. Die dritte Form der Konsistenz ist die *natürliche Bedeutung* der Konzepte der Ontologie [SMMS02, S. 289]. Diese sollen, unter Berücksichtigung des gesunden Menschenverstandes, sinnvoll sein. So wäre *Computer* als Unterklasse von *Mitarbeiter* bezüglich syntaktischer und semantischer Konsistenz korrekt, jedoch nicht sehr sinnvoll. Die natürliche Bedeutung kann durch eine ausführliche Beschreibung der Konzepte explizit in der Ontologie dargestellt werden. Würde man die zwei Konzepte des Beispiels den disjunkten Superklassen *Technisches Gerät* bzw. *Person* zuweisen, ist die semantische Inkonsistenz offensichtlich. Die natürliche Bedeutung lässt sich also idealerweise auf die semantische Inkonsistenz reduzieren und wird deshalb in dieser Arbeit nicht weiter betrachtet.

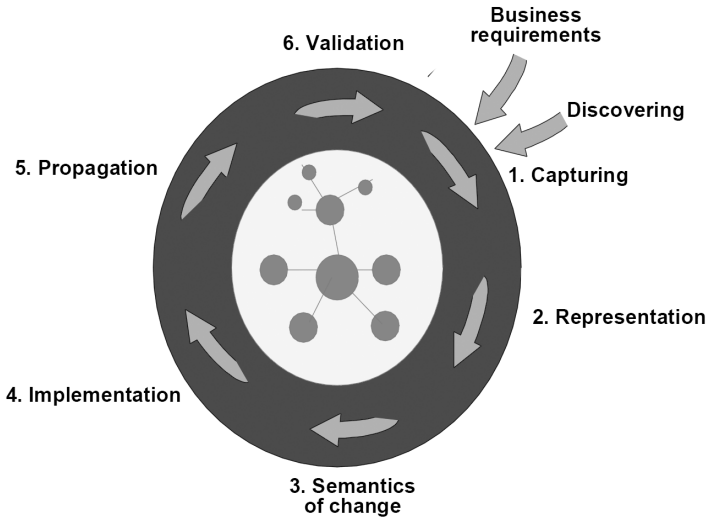

Abbildung 3.5: Ontologie-Evolutionsprozess aus [SMMS02]

Noy et al. haben festgestellt, dass die Ontologie-Evolution vergleichbar mit der Datenbank-Schema-Evolution ist [NK04]. Jedoch gibt es bedeutende Unterschiede zwischen beiden Ansätzen, die näher erläutert werden sollen. Ein Datenbankschema stellt nur die Struktur der Datenbank dar. Eine Ontologien hingegen enthält Struktur und Daten. Änderungen der Ontologie umfassen also gegebenenfalls auch die in der Ontologie gespeicherten Daten. Weiterhin definieren die Konzepte und Beziehungen in der Ontologie eine explizite Semantik, die bei Änderungen berücksichtigt werden muss. Darüber hinaus greifen viele unabhängige Institutionen, wie private Benutzer und Unternehmen, direkt auf die Ontologien zu. Andere Ontologien verwenden Teile der Ontologie wieder und Anwendungen greifen auf den Datenbestand zu. Somit müssen alle Änderungen an alle abhängigen Ontologien und Programme weitervermittelt werden. Auf Datenbanken wird hingegen meist über eine Zugriffsschicht zugegriffen, die sich unter der Kontrolle des Eigentümers der Datenbank befindet. Aufgrund dieser Indirektion können Änderungen an der Datenbank vorgenommen werden ohne dass notwendigerweise die Programme, die auf die Datenbank zugreifen, ebenfalls geändert werden müssen [NK04, S. 432]. Ontologien und Datenbankschemata unterscheiden sich weiterhin in der Wissensrepräsentation. Ontologien haben eine größere Ausdrucksstärke als Datenbankschemata. So ist es beispielsweise möglich, mit Ontologien die Kardinalität von Beziehungen zu definieren. Dies ist über die reine Schema-Definition einer Datenbank nicht möglich. Außerdem gibt es in Ontologien keine klare Trennung zwischen der Struktur und den Daten. Das heißt, dass Individuen von Klassen wiederum Klassen sein können. Daraus folgt, dass Ontologien und damit die darauf ausgeführten Änderungsoperationen wesentlich komplexer sind als Datenbankschemata. Die Betrachtung dieser Unterschiede macht deutlich, dass zwar Erkenntnisse der Schema-Evolution in begrenztem Rahmen auf Ontologien übertragen werden können, jedoch die Besonderheiten von Ontologien berücksichtigt werden müssen.

Eine besondere Erwähnung gilt an dieser Stelle der Beziehung zwischen Evolution und Versionierung. Wie in Abbildung 3.6 gezeigt, werden bezüglich der Weiterentwicklung von Datenbankschemata Evolution und Versionierung als zwei getrennte Aktivitäten verstanden. In der Evolution wird das Datenbankschema weiterentwickelt und es soll sichergestellt werden, dass auf Datenbestände des alten Schemas auch über das neue Schema zugegriffen werden kann. Die Versionierung beschäftigt sich hingegen mit der Frage, wie Datenbestände unterschiedlicher Versionen über Schemata anderer Versionen verfügbar sind. Im Unterschied dazu sind in

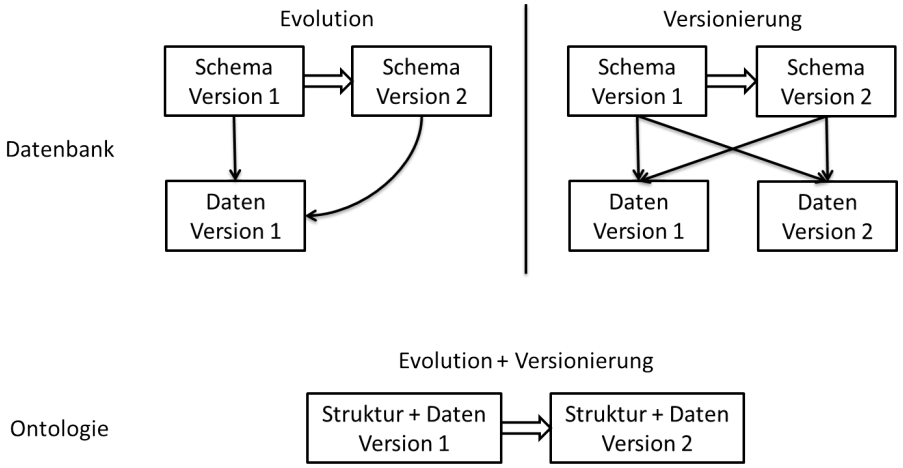

Abbildung 3.6: Ontologie-Evolution vs. Ontologie-Versionierung nach [NK04]

Ontologien die Struktur und die Daten nicht getrennt, weshalb Evolution und Versionierung als eine gemeinsame Aktivität betrachtet werden müssen [NK04, S. 433]. Ontologien unterstützen diesbezüglich ein Versionierungskonzept. Das bedeutet, dass einer Ontologie eine eindeutige Versionsnummer zugewiesen werden kann. So können mehrere Versionen derselben Ontologie gleichzeitig existieren.

3.4 Unterschiede zwischen Ontologie-Evolution und Refactoring

Die Änderungsoperationen der Ontologie-Evolution stellen Umstrukturierungen der Ontologie dar. Deshalb soll hier die Beziehung zum Refactoring betrachtet werden. Ontologie-Refactorings unterscheiden sich grundsätzlich von Refactorings, wie sie aus der objektorientierten Programmierung bekannt sind. Wie in Kapitel 2.2 behandelt, werden bei einem Refactoring Änderungen am Programmcode vorgenommen, wobei die Semantik des Programms erhalten bleiben soll. Die Semantik ergibt sich in diesem Zusammenhang aus dem Verhalten des Programms. Eine Ontologie repräsentiert stattdessen Daten und deren Struktur. Die Semantik einer Ontologie hat damit eine statische Beschaffenheit und ist mit der Ausführungssemantik eines Programms nicht vergleichbar. Bei der Umstrukturierung einer Ontologie liegt das Augenmerk also auf der Erhaltung der syntaktischen und semantischen Konsistenz, wie sie im Abschnitt 3.3 definiert werden. Noy et al. beschreiben die Konsistenz einer Ontologie mit so genannten Verträglichkeitsdimensionen noch detaillierter [NK04, S. 434]. Sie führen dabei das Konzept der *Erhaltung der Individuen* ein, das die Auswirkungen einer Änderung auf den Datenbestand der Ontologie beschreibt. Für jede Änderung muss also nicht nur die Erhaltung der syntaktischen und semantischen Konsistenz geprüft werden, sondern auch, ob die bestehenden Individuen und deren Beziehungen noch konsistent zur Struktur der Ontologie sind. Noy et al. gehen davon aus, dass eine automatische Einteilung der Änderungsoperationen entsprechend ihrer Konsistenzerhaltung nicht möglich ist [NK04, S. 435].

Der Unterschied zwischen Refactorings von objektorientierten Programmen und Refactorings von Ontologien wird besonders deutlich, wenn das *Prinzip der zwei Hüte* von Kent Beck betrachtet wird [FBB+99]. Beck sagt, dass das Programmieren aus zwei klar voneinander getrennten Aktivitäten besteht. Die eine Aktivität besteht darin, Tests und Produktionscode zu schreiben, um dem Programm neue Funktionalitäten hinzuzufügen. Die andere Aktivität besteht in der Umstrukturierung des vorhandenen Programms ohne die Semantik (die Funk-

tionalität) zu ändern. Der Programmierer setzt sich also im übertragenen Sinne entweder den einen oder den anderen Hut auf und arbeitet immer an nur genau einer dieser beiden Aufgaben. Bei Ontologien ist diese Unterscheidung jedoch nicht immer sinnvoll. Es gibt nur wenige Änderungen einer Ontologie, welche die Semantik einer Ontologie erhalten. Dazu gehören die Umbenennung von Konzepten, das Austauschen semantisch äquivalenter Konzepte oder die Verwendung von unterschiedlichen Schreibweisen für dasselbe Konzept. Im Unterschied zum Refactoring aus der objektorientierten Programmierung zielen also Umstrukturierungen aus der Ontologie-Evolution darauf ab, die Semantik bewusst zu verändern und dabei die Konsistenz zu erhalten.

Ein Vergleich von Ontologien und objektorientierten Programmen lässt weiterhin erkennen, dass Ontologien keine Interfaces haben, die interne Strukturen verbergen. Das Prinzip der Datenkapselung nach David Parnas ist eines der wichtigsten Prinzipien der objektorientierten Programmierung [Par76]. Es erlaubt, die interne Implementierung eines Programms zu ändern, während die Schnittstelle und das Verhalten nach außen gleich bleibt. Ontologien verfolgen hingegen das Prinzip, dass alle Strukturen nach außen sichtbar sein sollen. Ändert man Teile der Ontologie, hat das also zwangsläufig auch Auswirkungen auf deren äußere Erscheinung.

Diese Betrachtungen decken nur einen kleinen Teil aller Unterschiede zwischen Ontologie-Evolution und Refactorings ab. Lediglich erwähnt werden sollen an dieser Stelle die strukturellen Unterschiede zwischen den Konzepten einer Ontologie und den Konzepten der objektorientierten Programmierung. Konzepte wie die Annahme nicht-eindeutiger Namen, die Offene-Welt-Annahme, anonyme Klassen und globale Propertys sind Eigenheiten von Ontologien, die in objektorientierten Ansätzen nicht zu finden sind.

3.5 Zukünftige Entwicklungen – Linked Data

Bizer, Heath und Berners-Lee haben 2009 einen Artikel veröffentlicht, in dem sie den aktuellen Stand semantischer Techniken begutachten und einen Ausblick auf mögliche zukünftige Entwicklungen geben [BHBL09]. Bizer et al. heben insbesondere die Relevanz der Vernetzung von unstrukturierten, global verteilten Daten hervor. Diese soll einen freien Zugriff auf alle im Web verfügbaren Informationen ermöglichen. Dieser Ansatz wird mit dem Begriff *Linked Data* beschrieben. Linked Data folgt vier einfachen Regeln.

- „*Use URIs as names for things*
- *Use HTTP URIs so that people can look up those names.*
- *When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL)*
- *Include links to other URIs, so that they can discover more things.*“ [BL06].

Die Vision von Linked Data ist, dass alle Daten der Welt über frei verfügbare Ontologien klassifiziert werden können. Dies hätte eine wesentliche Vereinfachung der Kommunikation über das Web zur Folge, da vorhandene Klassifikationen wiederverwendet werden können und somit eine einheitliche Begriffswelt für alle Beteiligten existiert. Bevor diese Vision wahr werden kann, müssen zuerst entsprechende Ontologien erstellt werden, die für eine einheitliche Klassifikation geeignet sind. Besonders erwähnenswert ist in diesem Zusammenhang das Projekt *Linking Open Data*[15] (LOD), dessen Ziel es ist, im Web frei verfügbare Daten zu finden, in RDF-Tripel zu übersetzen und nach den Prinzipen von Linked Data der Öffentlichkeit zur Verfügung zu

[15] `http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData`

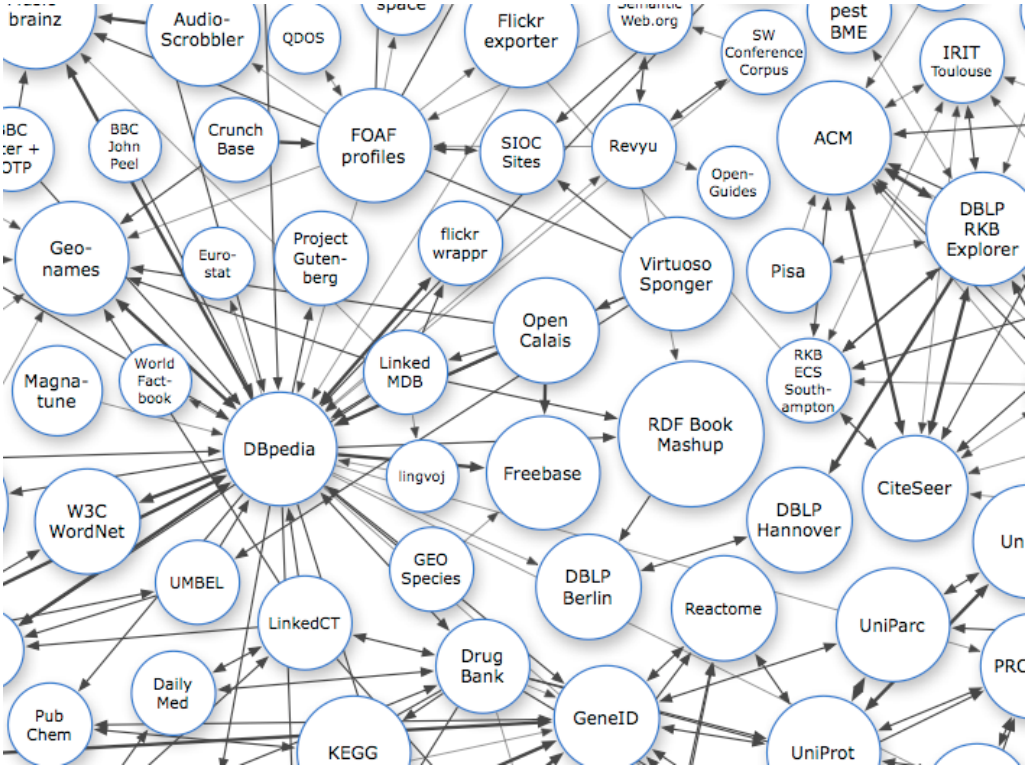

Abbildung 3.7: Ausschnitt aus dem Linking Open Data Cloud Diagram [CJ10]

stellen. Der aktuelle Stand dieses Projekts wird im *Linking Open Data Cloud Diagram* grafisch dargestellt. Abbildung 3.7 zeigt einen Ausschnitt dieses Diagramms. Jeder Kreis repräsentiert eine Ontologie, wobei die Größe der Kreise in etwa zur Anzahl der Konzepte korreliert, die in der Ontologie definiert sind. Die Pfeile verdeutlichen Referenzen zwischen den Ontologien.

Das *Semantic Web Journal*[16], dessen erste Ausgabe im Dezember 2010 erschienen ist, enthält darüber hinaus die neusten wissenschaftlichen Erkenntnisse aus dem Bereich Semantic Web.

3.6 Zusammenfassung

Dieses Kapitel gab einen Überblick über die Konzepte, welche dieser Arbeit zugrunde liegen. In der MDSD dienen Metamodelle dazu, die Struktur von Softwaresystemen zu beschreiben und die Möglichkeiten der Codegenerierung zu nutzen. Refactorings strukturieren Programmcode so um, dass dessen Struktur verbessert wird, jedoch das Verhalten des Programms unverändert bleibt. In Ontologien können Daten und deren Struktur gespeichert werden. Sie bilden die Grundlage des Semantic Web. Ontologien lassen sich darüber hinaus zur Darstellung von Softwaresystemen ähnlich den Metamodellen der MDSD einsetzten. In der Ontologie-Evolution werden Ontologien mit Änderungsoperationen, die den Refactorings ähnlich sind, weiterentwickelt. Dies ebnet den Weg, um Ontologien zum Design von Softwaresystemen einzusetzen, wobei diese jederzeit an sich ändernde Bedingungen angepasst werden können.

[16] `http://semantic-web-journal.net/`

4 Eingesetzte Werkzeuge

4.1 Verbindung von Ontologien und Modellen mit OntoMoPP

Zur Darstellung und Bearbeitung von Ontologien wird in dieser Arbeit *OntoMoPP* (Ontology Model Parser and Printer) verwendet. Dies ist ein Teilprojekt von *EMFText*[1], das am Lehrstuhl Softwaretechnologie der Fakultät Informatik der TU Dresden entwickelt wurde [HJK+09]. EMFText ist ein Werkzeug zur Erstellung von textuellen DSLs. Es baut auf dem EMF auf und nutzt Ecore zur Definition der abstrakten Syntax einer Sprache. Auf Basis dieser abstrakten Syntax kann der Sprachdesigner eine oder mehrere textuelle Notationen, so genannte konkrete Syntaxen, entwerfen. Die Grammatik der konkreten Syntax wird dabei mit Regeln der erweiterten Backus-Naur-Form (EBNF) definiert. EMFText kann aus der abstrakten und konkreten Syntax automatisch Parser, Printer und Editoren generieren. Der Parser kann Text lesen, der konform zur angegebenen Syntax ist und diesen in ein Modell umwandeln. Der Printer ist das Gegenstück zum Parser. Er wandelt das Modell in Text um, damit es in einer Datei gespeichert oder in einem Editor angezeigt werden kann. Der Editor ermöglicht schließlich die Bearbeitung der Modelle unter Nutzung der textuellen Syntax. Er unterstützt dabei Syntax-Hervorhebung, Code-Vervollständigung und -Navigation.

OntoMoPP wurde entwickelt, um mit EMFText auch Ontologien bearbeiten zu können. Basis von OntoMoPP ist das *OWL-Metamodell*[2], mit dem Ontologien in Manchester OWL Syntax beschrieben werden können [WH09, S. 4]. Auf diesem OWL-Metamodell bauen zwei weitere Bestandteile von OntoMoPP auf: OWLText und OWLizer. *OWLText* ist ein Text-Editor zur Bearbeitung von Ontologien. Die verwendete konkrete Syntax entspricht dabei der Manchester OWL Syntax. Der Ansatz, die komplette Ontologie auf Basis ihrer textuellen Repräsentation zu bearbeiten, unterscheidet OWLText von anderen Ontologie-Editoren, wie Protégé und NeOn-Toolkit, die eine Oberfläche mit Textboxen und Eingabefeldern für einzelne Elemente der Ontologie bereitstellen. Die Abbildung 4.1 zeigt den Ontologie-Editor von OWLText mit einer geöffneten Ontologie und der dazugehörigen Outline des Ontologie-Modells. *OWLizer* ist ein Werkzeug zur Umwandlung von Metamodellen und Modellen in Ontologien. Es wird im Abschnitt 5.1.1.2 ausführlich behandelt.

OntoMoPP kommt in dieser Arbeit zum Einsatz, da es eine Infrastruktur zur Bearbeitung von Ontologien bereitstellt, die sich nahtlos in EMF integriert. Auf der Basis von EMF können damit Ontologien als Modelle gehandhabt und Domänenmodelle in Form von Ecore-Modellen dargestellt werden. Somit bietet OntoMoPP und EMFText eine Architektur, die zur Darstellung und Bearbeitung von Ontologien sowie von Domänenmodellen verwendet werden kann.

[1] `http://www.emftext.org/`
[2] `http://www.emftext.org/index.php/EMFText_Concrete_Syntax_Zoo_OWL2_Manchester`

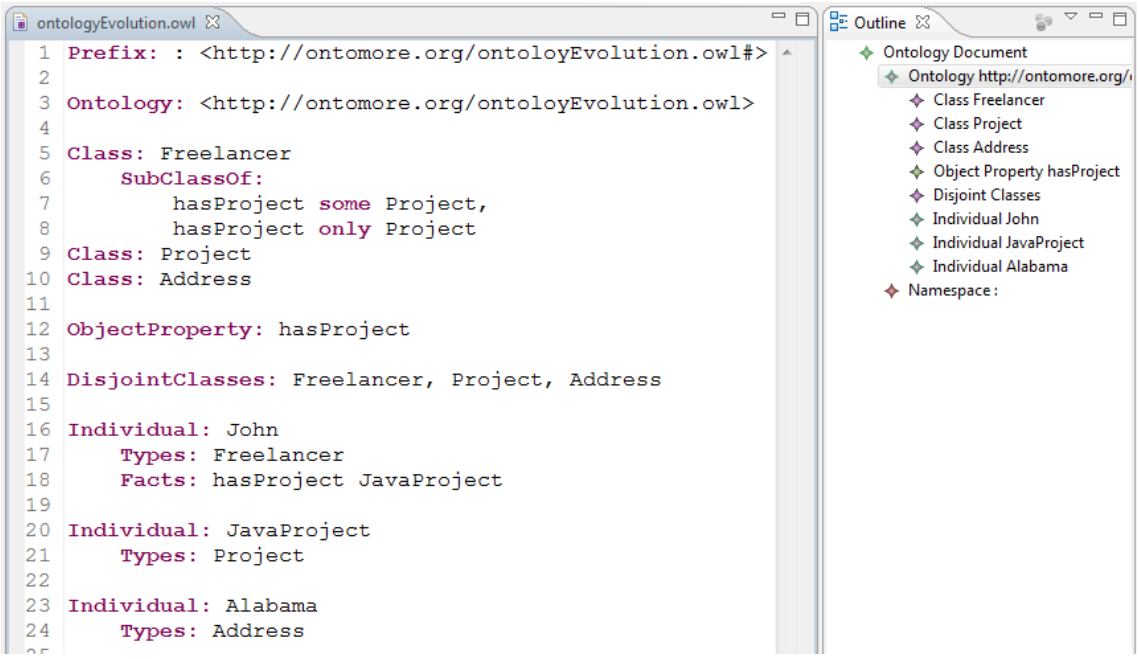

Abbildung 4.1: Screenshot des Ontologie-Editors OWLText

4.2 Generisches Modell-Refactoring mit Refactory

In seiner Diplomarbeit hat Jan Reimann einen Ansatz für generisches Modell-Refactoring entwickelt und in dem Werkzeug *Refactory* umgesetzt [Rei10]. Der Ansatz basiert auf der Definition von generischen Refactorings. Diese können für konkrete Umstrukturierungen in unterschiedlichen Modellen immer wiederverwendet werden. Abbildung 4.2 zeigt einen Überblick des Ansatzes. Generische Refactorings werden durch *Role Models* definiert. Diese nutzen das Prinzip der Rollen nach Riehle und Gross [RG98], um Elemente und deren Beziehungen zu definieren, die an dem Refactoring beteiligt sind. Zu jedem Role Model gehört eine *Refactoring Specification*. Diese beschreibt die auszuführenden Änderungen für ein Refactoring. Dazu werden in einer eigens dafür entworfenen DSL Änderungsoperationen definiert, mit denen die im Role Model definierten Elemente neu erstellt, verändert oder gelöscht werden können. Um ein Refactoring auf einem konkreten Modell tatsächlich ausführen zu können, muss eine Abbildung der allgemeinen Konzepte der Role Models auf konkrete Elemente eines Metamodells erfolgen. Dies wird durch ein *Role Mapping* umgesetzt. Das Role Mapping bildet also jedes Element des Role Models auf ein Element eines Metamodells ab. Auf Basis dieser Role Mappings können auf allen Modellen des Metamodells Refactorings ausgeführt werden. Da die generischen Refactorings allgemeine Umstrukturierungen beschreiben, können sie über mehrere Role Mappings unterschiedlichen Metamodellen zugeordnet werden. Dies stellt eine Wiederverwendung der Role Models und Refactoring Specifications sicher. Daraus resultiert ebenfalls, dass Refactorings für Modelle beliebiger Metamodelle definiert und auf ihnen ausgeführt werden können.

Das Werkzeug Refactory nutzt EMFText, um den vorgestellten Ansatz technisch zu verwirklichen. Demnach sind Role Models, Refactoring Specifications und Role Mappings als Modelle des EMF umgesetzt. Die Oberfläche von Refactory integriert sich ebenfalls in EMF. Das heißt, dass Refactorings auf beliebigen Modellen des EMF durchgeführt werden können, unabhängig von der Repräsentation der Modelle.

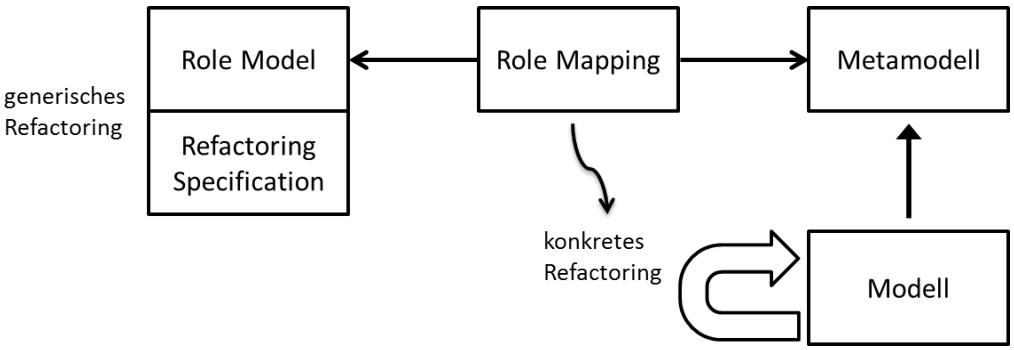

Abbildung 4.2: Architektur des generischen Modell-Refactorings nach [Rei10]

Mit Refactory können Modell-Refactorings ebenfalls auf Ontologien angewandt werden. Voraussetzung dafür ist, dass die Ontologien als Modelle vorliegen. OntoMoPP stellt genau das sicher. Der gemeinsame Einsatz von OntoMoPP und Refactory ermöglicht somit die Zusammenführung von Ontologien und Domänenmodellen, wobei Refactorings in beiden Strukturen durchgeführt werden können.

5 Stand der Forschung

5.1 Abbildung von Ontologien auf Domänenmodelle

Um Ontologien als Modellierungswerkzeuge einzusetzen, muss zunächst geklärt werden, welche Beziehung zwischen Ontologien und Domänenmodellen besteht. Wie in Kapitel 3.2 gezeigt, stellen Ontologien Konzepte der realen Welt dar. Um das in den Ontologien gespeicherte Domänenwissen nun auch in Softwaresystemen nutzen zu können, muss eine Beziehung zwischen Ontologie und Softwaresystem hergestellt werden. Einerseits können Ontologien in der Modellierung eingesetzt werden. Hier existieren bereits Modelle, welche die Anwendungsdomäne beschreiben. Diese Modelle können um Ontologien erweitert werden, um Domänenwissen detaillierter darzustellen. Andererseits besteht die Möglichkeit, Ontologien auf Datenstrukturen auf der Ebene von Datenbanken abzubilden. Diese Abbildung kann lediglich die Konzepte beider Techniken umfassen, die einander entsprechen, erlaubt jedoch, bestehende Daten einer Datenbank in eine Ontologie zu übernehmen. Beide Vorgehensweisen werden im Folgenden näher vorgestellt.

5.1.1 Einsatz von Ontologien in der Modellierung

Aufgrund ihrer formalen Beschreibung kann mit einer Ontologie Domänenwissen detailliert dargestellt werden. Dies ermöglicht es, logische Schlussfolgerungen zu ziehen, die ohne Ontologie nicht möglich wären [SWGP10]. Es gibt zwei Ansätze, mit denen das Ziel verfolgt wird, die Vorteile von Ontologien und von Modellierungstechniken in der Softwareentwicklung zusammenzuführen [Bre08]. Der erste Ansatz besteht darin, Modellierungstechniken, wie UML, einzusetzen um Ontologien zu modellieren. Der zweite Ansatz verfolgt das Ziel, bestehende Modelle um Ontologien zu ergänzen oder Ontologien aus diesen Modellen zu generieren, um in den Ontologien Informationen zu halten, die in den Modellen aufgrund ihrer begrenzten Ausdrucksstärke nicht dargestellt werden können.

5.1.1.1 UML als Modellierungssprache für Ontologien

Zum ersten Ansatz sei gesagt, dass die Ontologien selbst die Domänenmodelle darstellen und nicht auf eine andere Repräsentation eines Domänenmodells abgebildet werden. Die Herausforderung besteht nun darin, Ontologien in einer Weise zu modellieren, wie man es von herkömmlichen Domänenmodellen gewohnt ist. Schon 1999 haben Cranefield und Purvis festgestellt, dass Ontologien im Bereich der Softwareentwicklung in größerem Maße eingesetzt werden können, wenn es möglich ist, sie mit den bekannten objektorientierten Methoden zu modellieren [CP99]. Die Autoren schlagen dazu vor, UML als Modellierungssprache für Ontologien einzusetzen. Es sei allerdings darauf hingewiesen, dass dieser Vorschlag aus einer Zeit stammt, in der Ontologien vorwiegend im Bereich der Künstlichen Intelligenz eingesetzt wurden und noch nicht standardisiert waren. Der erste Ontologie-Standard erschien 2004 mit OWL; der heute genutzte Standard, OWL 2, wurde 2009 vom W3C verabschiedet. Cranefield und Purvis schlagen vor, Ontologien mit einer Kombination aus UML-Klassendiagrammen, OCL-Contraints und UML-Objektdiagrammen zu modellieren. Sie stützen sich dabei auf die

Tabelle 5.1: Auszug des UML-OWL-Mappings des ODM [OMG09, S. 213]

UML-Element	OWL-Element	Kommentar
class	class	
instance	individual	OWL individual unabhängig von Klasse
attribute	property	OWL property kann global sein
enumeration	oneOf	
package	ontology	

Ontologie-Beschreibungssprachen, die zum damaligen Zeitpunkt den Stand der Technik darstellten: KIF und KL-ONE [CP99]. Diese stellen jedoch nur einen Bruchteil der Ausdrucksstärke heutiger Ontologie-Beschreibungssprachen dar. Cranefield und Purvis erkennen allerdings schon zu diesem frühen Entwicklungsstadium der Ontologien, dass es nicht immer möglich oder sinnvoll ist, UML zur Modellierung von Ontologien einzusetzen. Entweder müsste der UML-Standard erweitert werden, um die Semantik von Ontologien besser abbilden zu können oder die Ontologie wird lediglich als Teil eines Softwaresystems mit eingeschränktem Einsatzbereich verwendet [CP99].

Die OMG hat diesen Gedanken aufgegriffen und 2009 einen Standard zur Beschreibung von Ontologien mit UML veröffentlicht, das Ontology Definition Metamodel (ODM) [OMG09]. Dieser Standard definiert Metamodelle konform zu MOF für RDF, OWL und weiteren verwandten Techniken, wie Common Logic und Topic Maps. Anschließend werden UML-Profile für RDF, OWL und Topic Maps definiert, die diese Metamodelle nutzen, um UML zu erweitern. Die Erweiterung von UML um ein neues Metamodell für OWL ist notwendig, da sich viele Konzepte in OWL und UML unterscheiden [GDD10, S. 235]. Beide Standards definieren das Konzept einer Klasse. Jedoch stellen OWL-Klassen Mengen dar, die sich in ihrer Definition von den UML-Klassen unterscheiden. Die Definition von OWL-Klassen lehnt sich stärker an dem mathematischen Mengenkonzept an. Darüber hinaus bestehen mehrere Möglichkeiten, anonyme Klassen zu definieren (vgl. Abschnitt 3.2.2). In Ontologien gibt es keine Einschränkung der Beschaffenheit von Individuen. OWL-Klassen gruppieren lediglich eine gewisse Teilmenge aller bekannter Individuen. UML-Klassen hingegen werden durch einen Namen eindeutig identifiziert und legen fest, welche Art von Objekten überhaupt existieren kann. Weiterhin besteht eine Ähnlichkeit zwischen der Property von OWL und dem Attribut bzw. der Assoziation von UML, jedoch gibt es auch hier im Detail Unterschiede [GDD10, S. 238]. Der wichtigste Unterschied besteht darin, dass Propertys in OWL eigenständige Konzepte (engl. first class concepts) sind, wohingegen Attribute und Assoziationen in UML immer zu einer Klasse gehören müssen und keine eigenständigen Konzepte darstellen. Deshalb wird im ODM ein separates Klassen-ähnliches Konzept *OWLObjectProperty* definiert, das die Besonderheiten einer Property darstellt.

Darüber hinaus stellt der ODM-Standard die Konzepte von OWL und UML gegenüber und gibt einen informellen Vergleich von ähnlichen Konzepten [OMG09, S. 213]. Die Tabelle 5.1 zeigt einen Ausschnitt dieses Vergleiches. Die OMG hat außerdem Konzepte zusammengetragen, die keine Entsprechung im jeweils anderen Standard haben. UML-Konzepte, die nicht in OWL vorkommen, sind zum Beispiel navigierbare Assoziationen, abgeleitete Attribute bzw. Referenzen und abstrakte Konzepte. Und OWL-Konzepte, die nicht in UML vorkommen, sind zum Beispiel Thing (die Superklasse aller Klassen), globale Propertys, eigenständige Individuen (ohne zugeordnete Klasse), Mengeneinschränkungen (allValuesFrom und someValuesFrom), Eigenschaften von Propertys (symmetrisch, transitiv), disjunkte und komplementäre Klassen. Es sei allerdings darauf hingewiesen, dass das ODM auf der ersten Version von OWL von 2004 basiert und noch nicht die Konzepte des neuen OWL 2-Standards berücksichtigt.

Das ODM definiert eine klare Beziehung zwischen OWL und UML. Die Abbildung zwischen beiden Techniken berücksichtigt dabei alle Konzepte und teilt diese, in Konzepte, die eine Entsprechung in der jeweils anderen Technik haben und solche, deren Abbildung nicht möglich ist, ein. Das OWL-UML-Profil definiert eine Erweiterung von UML. Es erlaubt, Ontologien mit UML-Klassendiagrammen zu erstellen und zu bearbeiten. Jedoch geht das ODM nicht darauf ein, wie die zusätzlichen Möglichkeiten von Ontologien zur Beschreibung von Softwaresystemen genutzt werden können. Das ODM betrachtet auch nicht die Beziehung zwischen Ontologie und Softwaresystem. Somit bleibt offen, wie in der Ontologie modellierte Konzepte tatsächlich in einem Softwaresystem Anwendung finden. Das ODM kann aber zumindest als Grundlage für eine Abbildung von Ontologien auf Domänenmodelle dienen.

5.1.1.2 Modelle durch Ontologien erweitern

Der zweite Ansatz besteht in der Erweiterung von Modellen durch Ontologien. Ontologien werden an dieser Stelle eingesetzt, um Konzepte zu modellieren, die in Modellen nicht dargestellt werden können. Dazu gehören zum Beispiel Eigenschaften von Beziehungen, wie Transitivität oder Reflexivität. Ontologien werden außerdem eingesetzt, um logische Schlussfolgerungen über Daten zu ziehen, was mit Modellen nicht möglich ist. Silva Parreiras hat diesbezüglich festgestellt, dass die Ansätze von Ontologien und Modellierung in vielen Belangen ähnlich sind, es jedoch im Detail starke Unterschiede gibt [PSW07a]. Ontologien und Modelle der MDSD können laut Silva Parreiras nicht gleichgesetzt werden. Stattdessen müssen einzelne Teile beider Technologien gefunden werden, die gemeinsam verwendet werden können. Er schlägt vor, Modelle der MDSD um Konzepte der Ontologien zu erweitern, damit die Vorteile beider Technologien genutzt werden können. Dieser Ansatz wird vom MOST-Projekt umgesetzt.

Mit dem MOST-Projekt (Marrying Ontology and Software Technology) verfolgt man das Ziel, Ontologien und semantische Techniken in die modellgetriebene Softwareentwicklung zu integrieren und sie somit zu einer ontologiegetriebenen Softwareentwicklung (ODSD) weiterzuentwickeln. Damit soll die Qualität des zu erstellenden Softwaresystems verbessert werden [MA08]. Dies wird ermöglicht, indem Informationen aus sonst getrennten Datenbeständen über den gesamten Entwicklungsprozess zusammengeführt werden. Mit dem MOST-Ansatz sollen *ontologiebewusste Softwaresysteme* erstellt werden. Das heißt, Ontologien werden lediglich während des Entwicklungsprozesses eingesetzt. Guarino unterscheidet davon die *ontologiegetriebenen Softwaresysteme*, bei denen eine Ontologie ein fester Bestandteil des erstellten Softwaresystems ist [Gua98].

Für die Betrachtungen der vorliegenden Arbeit ist insbesondere das Work Package 1 (WP1) des MOST-Projekts von Bedeutung, in dem Ontologien mit der Model-driven Architecture (MDA) integriert werden. Zu diesem Zweck wird ein gemeinsames Metamodell erarbeitet, das die Konzepte von UML und OWL vereint und es somit ermöglicht, Modelle mit Konzepten aus beiden Welten zu erstellen. Darüber hinaus wurde eine Integrationstechnik entwickelt, die Artefakte des gesamten Entwicklungsprozesses in einer gemeinsamen Ontologie zusammenführt. Sofern diese Artefakte als Modelle vorliegen (z. B. Modelle von UML-Diagrammen oder Modelle von Java-Code), können sie in Ontologien transformiert werden. Einmal erstellt können diese Ontologien zu einem gemeinsamen Datenbestand verknüpft werden, sodass Abfragen relevante Ergebnisse aus allen Artefakten zurückgeben [Two10]. Damit wird dem Softwareentwickler ein Überblick gegeben, wie beispielsweise ein bestimmter Teil des Quellcodes mit der Anforderungsdokumentation zusammenhängt. Zur Transformation der Modelle in Ontologien wurde das Werkzeug OWLizer entwickelt, dessen Architektur in Abbildung 5.1 zu sehen ist [SWGP10]. OWLizer transformiert beliebige Metamodelle in die TBox und die dazugehören Modelle in die ABox einer Ontologie. Die dabei verwendete Abbildung von Konzepten des Ecore-Meta-Metamodells auf Konzepte des OWL-Metamodells ist in Tabelle 5.2 zu sehen.

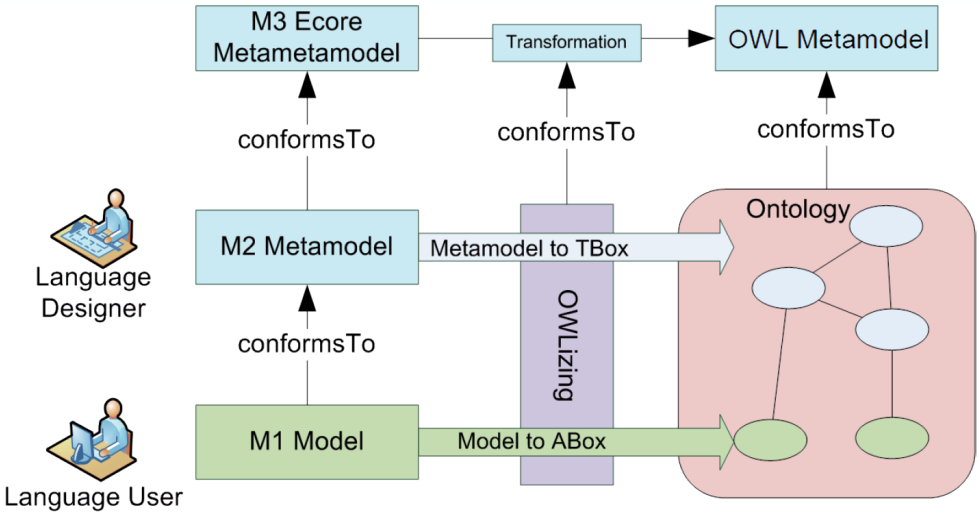

Abbildung 5.1: OWLizer-Architektur aus [SWGP10, S. 16]

Tabelle 5.2: Ecore-OWL-Mapping des OWLizers [SWGP10, S. 16]

Ecore	**OWL**
package	ontology
class	class
instance and literals	individual and literals
reference, attribute	object property, data property
data types	data types
enummeration	enummeration
multiplicity	cardinality

Auf Basis dieser Transformation können Modelle mit Annotationen erweitert werden, um eine größere Ausdrucksstärke zu erhalten [Sch10a]. Die Annotationen stellen dabei semantische Konzepte dar, die den Ontologien vorbehalten sind. So wird beispielsweise das Verhalten einer Methode als SPARQL-Abfrage definiert, die nach der Transformation über dem tatsächlichen Datenbestand ausgeführt werden kann. Mit der Transformation derartiger Annotationen ist es außerdem möglich, Eigenschaften von Propertys, wie die Transitivität oder Reflexivität, zu modellieren.

Die Ansätze des MOST-Projekts wurden im Framework TwoUse (Transforming and Weaving Ontologies and UML in Software Engineering) implementiert, womit eine ontologiegetriebene Softwareentwicklung ermöglicht wird [PSW07b].

Im MOST-Projekt wird keine vollständige Betrachtung möglicher Abbildungen zwischen Ontologien und Modellen durchgeführt. Stattdessen werden nur Beziehungen zwischen den Elementen in Ontologien bzw. Modellen betrachtet, die für eine Erweiterung der Modelle durch Ontologien sinnvoll sind. Die Ontologien stellen also Erweiterungen bestehender Modelle dar. Das MOST-Projekt verfolgt das Ziel, Modelle und die in diese Modelle integrierten Ontologien automatisch in ausführbaren Programmcode zu transformieren. Somit wird eine definierte Beziehung von den Modellen und Ontologien auf Softwaresysteme hergestellt. Es ist jedoch nicht angedacht, das komplette Domänenmodell mit einer Ontologie zu beschreiben.

5.1.2 Abbildungen zwischen Ontologien und Datenbanken

Die zweite Möglichkeit, Ontologien und Softwaresysteme zu verbinden, ist, Ontologien mit Datenbanken in Beziehung zu setzen. In Datenbanken wird ein Großteil der digital verfügbaren Daten gespeichert, während Ontologien die Basis des Semantic Web darstellen [SSM10]. Aus diesem Grund gibt es schon seit Beginn der Entwicklung von Ontologien die Bestrebung, eine Abbildung zwischen Datenbanken und Ontologien (Database-to-Ontology-Mapping) herzustellen. Spanos et al. haben zwei unterschiedliche Herangehensweisen identifiziert, mit denen diese Bestrebungen adressiert werden. Einerseits wird eine neue Ontologie auf Basis einer bestehenden Datenbank erstellt. Andererseits wird ein Mapping zwischen einer bestehenden Datenbank und einer bestehenden Ontologie erstellt [SSM10]. Ein Großteil der aktuellen Forschung beschäftigt sich mit dem Ansatz, Daten aus relationalen Datenbanken in RDF-Tripel zu transformieren. Der naive Ansatz besteht darin, Tabellen auf Klassen und Attribute auf Prädikate abzubilden. Jedes Tupel einer Tabelle wird somit durch einen anonymen Knoten im RDF-Graphen repräsentiert, der über Prädikate mit allen Attributwerten verbunden ist. Abbildung 5.2 zeigt beispielhaft eine Tabelle und den daraus generierten RDF-Graphen. Es wurden viele Techniken entwickelt, die auf diesem Ansatz aufbauen. Spanos et al. geben einen Überblick über den aktuellen Stand der Forschungen in diesem Bereich [SSM10].

Dem Autor sind zum jetzigen Zeitpunkt jedoch keine Technologien bekannt, die Datenbank-Schemata auf einer höheren Abstraktionsstufe (auf Ebene der Tabellen, Assoziationen und Fremdschlüssel) auf Ontologien nach dem OWL-Standard abbilden können. Dies ist jedoch eine Voraussetzung dafür, eine Beziehung zwischen Domänenmodell und Ontologie ableiten zu können. Die momentan existierenden Ansätze sind auf die Transformation von Tabellen relationaler Datenbanken auf RDF-Tripel beschränkt. Deshalb kommt eine Integration von Ontologien in Softwaresysteme über eine Anbindung an Datenbanken zum aktuellen Stand der Forschung nicht in Frage. Aktuelle Ansätze in diesem Bereich ermöglichen es, Daten automatisch von Datenbanken in Ontologien zu übernehmen. Jedoch existieren noch keine Ansätze, wie Konzepte einer höherer Abstraktionsstufe, wie sie beispielsweise in Entity-Relationship-Modellen vorkommen, auf die Konzepte einer Ontologie abgebildet werden können.

Person

ID	Name	Wohnort
1	John	Alabama
2	Sally	Sydney

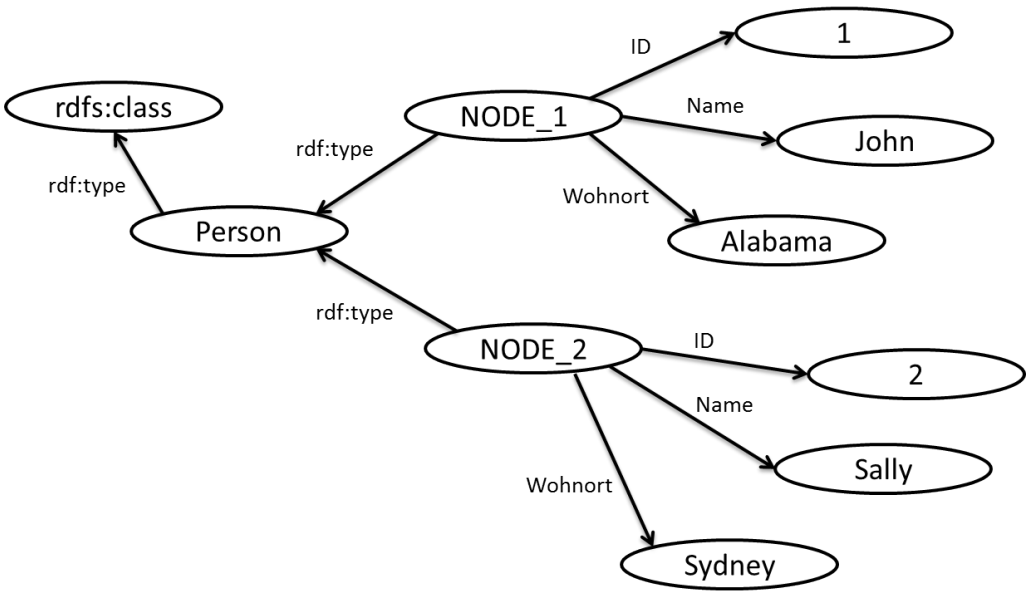

Abbildung 5.2: Tabelle mit daraus abgeleitetem RDF-Graph

5.2 Ontologie-Evolution

In diesem Abschnitt werden vorhandene Ansätze zur Ontologie-Evolution untersucht. Dazu werden zuerst die Anforderungen definiert, die an eine Ontologie-Evolution gestellt werden. Unter dem Gesichtspunkt dieser Anforderungen werden anschließend Forschungsarbeiten und Ontologie-Frameworks vorgestellt, woraus letztendlich eine Bewertung der vorhandenen Arbeiten abgeleitet wird. Ziel der Untersuchungen ist es, relevante Refactorings für Ontologien zu finden.

5.2.1 Anforderungen an Ontologie-Evolution

Wie schon im Abschnitt 1.1 argumentiert, müssen Ontologien ständig weiterentwickelt werden. Es ist möglich, dass Fehler im Design erst nach der Fertigstellung der Ontologie festgestellt werden. Die Anwendungsdomäne der Ontologie aber auch die Anforderungen der Benutzer an die Ontologie und deren Wissensrepräsentation können sich im Laufe der Zeit ändern. In all diesen Fällen ist eine Ontologie-Evolution notwendig. Stojanovic und Motik haben Anforderungen an Ontologie-Editoren für eine erfolgreiche Ontologie-Evolution zusammengestellt [SM02]. Diese Anforderungen sollen hier – angepasst an diese Arbeit und den aktuellen Stand der Forschung – zur Bewertung vorhandener Ontologie-Frameworks dienen und bilden die Grundlage für die Umsetzung der eigenen Konzeption im Kapitel 7.

1) Abdeckung Entsprechend der zugrunde liegenden Ontologie-Beschreibungssprache muss ein Ansatz zur Ontologie-Evolution elementare Änderungen für alle Sprachkonstrukte zur Verfügung stellen. Als Beschreibungssprache wird in dieser Arbeit OWL 2 zugrunde gelegt, da sie einen einheitlichen Standard darstellt [W3C09]. Aufbauend auf diesen elementaren Änderungen sollen komplexe Änderungen möglich sein. Alle Änderungen müssen sicherstellen, dass die Konsistenz der Ontologie erhalten bleibt und müssen eine Aussage zur Erhaltung der Individuen treffen.

2) Steuerung durch den Benutzer Änderungen erfordern unter Umständen zusätzliche Eingaben, um in der jeweiligen Situation korrekt ausgeführt werden zu können. Dem Benutzer sollte in einer solchen Situation Auskunft darüber gegeben werden, welche Eingaben nötig sind und wie diese die Ausführung der Änderung beeinflussen. Somit kann der Benutzer zwischen mehreren möglichen Alternativen zur Ausführung der Änderung wählen.

3) Transparenz Der Benutzer soll über alle Auswirkungen einer Änderung informiert werden bevor diese ausgeführt wird. Dies kann durch einen Vorher-Nachher-Vergleich erfolgen oder mit der Auflistung aller durchzuführenden elementaren Operationen. Die Konsistenz der Ontologie sollte dabei vom Refactoring sichergestellt werden.

4) Umkehrbarkeit Durchgeführte Änderungen sollen wieder rückgängig gemacht werden können. Dazu ist ein Undo/Redo-Mechanismus notwendig, der nach Möglichkeit mehrere Änderungen rückgängig machen oder wiederherstellen kann. Eine Historie zum Vergleich unterschiedlicher Versionen der Ontologie wäre an dieser Stelle ebenfalls denkbar.

5) Bedienbarkeit Der verwendete Editor soll dem Benutzer eine einfache Durchführung von Änderungen ermöglichen. Er zeigt dem Benutzer Informationen über mögliche Änderungen an und führt ihn durch den Änderungsprozess. Weiterhin identifiziert der Editor Inkonsistenzen sowie syntaktische Fehler und zeigt, an welcher Stelle der Ontologie sie auftreten.

5.2.2 Elementare und komplexe Änderungsoperationen

Es gibt bereits einige Ansätze, die elementare Änderungsoperationen definieren und durch deren Kombination komplexe Änderungsoperationen abgeleitet werden. Diese Ansätze sollen im Folgenden näher vorgestellt werden.

Javed et al. klassifizieren Änderungsoperationen einer Ontologie nach ihrer Komplexität mit vier Granularitätsstufen [JAP09]. Die unterste Stufe enthält dabei Operationen zum Erstellen oder Löschen einzelner Elemente der Ontologie, wie Klassen, Propertys, Individuen, Unterklassenbeziehungen und Domain- bzw. Range-Axiome. Diese elementaren Operationen werden auf der nächsten Stufe zu so genannten Kontext-Änderungsoperationen kombiniert. Je nach Kontext, in dem eine solche Operation durchgeführt wird, können unterschiedliche elementare Änderungsoperationen zum Einsatz kommen. Die Operation Move löscht beispielsweise ein Element der Ontologie und fügt ein neues ein. Hierbei kann es sich je nach Anwendungsfall um eine Klasse handeln, deren Superklasse sich ändert oder um ein Individuum, das in eine andere Klasse verschoben wird. Weitere Kontext-Änderungsoperationen sind Merge, Split und Copy. Auf den oberen beiden Granularitätsstufen werden diese Änderungsoperationen weiter abstrahiert, sodass Änderungen auf der Ebene der Anwendungsdomäne durchgeführt werden können. Es wird also nicht mehr ein Individuum einer Klasse hinzugefügt. Stattdessen wird beispielsweise ein Mitarbeiter eines Unternehmens einer Abteilung zugewiesen. Solche Änderungen der Domänen-Ebene werden dann über die Kontext-Änderungsoperationen auf elementare Änderungen der Ontologie abgebildet. Javed et al. haben die Änderungsoperationen auf der Basis von zwei Fallstudien erstellt. Sie verfolgten dabei das Ziel, Änderungsoperationen zu finden, die während der Erstellung und dem Einsatz der Ontologien tatsächlich Verwendung finden. Es wurden also keine formalen Kriterien aufgestellt, um die Korrektheit der Änderungsoperationen zu prüfen. Die Implementierung des Ansatzes von Javed et al. steht allerdings noch aus.

Noy und Klein gehen mit den schon im Abschnitt 3.4 erwähnten Verträglichkeitsdimensionen näher auf die Auswirkungen einer Änderungsoperation ein [NK04]. Sie haben ebenfalls einen Katalog von möglichen Änderungsoperationen aufgestellt und klassifizieren jede Änderung entsprechend der Erhaltung der Individuen. Es bleibt jedoch unklar, nach welchen Kriterien die Änderungsoperationen ausgewählt wurden. Die von Noy und Klein aufgestellten Änderungsoperationen konzentrieren sich dabei auf Veränderungen einzelner Axiome der Ontologie. Beispiele dafür sind das Hinzufügen oder Entfernen einer Klasse sowie das Verschieben einer Klasse in der Vererbungshierarchie (wobei lediglich das SubClassOf-Axiom geändert wird).

Noy und Klein erkennen jedoch ebenfalls die Notwendigkeit, elementare Änderungsoperationen zu komplexen Änderungsoperationen zusammenzusetzen [NK04, S. 435]. Das Ändern von Eigenschaften einer Property kann beispielsweise in zwei getrennten Operationen umgesetzt werden: Das Löschen der ursprünglichen Property und das Hinzufügen einer neuen Property mit den neuen Eigenschaften. Führt man beide Änderungen getrennt aus, befindet sich die Ontologie nach der Löschoperation unter Umständen in einem inkonsistenten Zustand, wenn Property-Assertions von Individuen auf eine nicht existierende Property verweisen. Erst die Kombination aus beiden elementaren Änderungen führt wieder zu einem konsistenten Zustand ohne Datenverlust. Eine Implementierung liegt aber auch hier nicht vor.

5.2.3 KAON – Das Karlsruhe Ontology and Semantic Web Framework

Das Karlsruhe Ontology and Semantic Web Framework (KAON[1]) wurde vom Forschungszentrum Informatik (FZI) und der Universität Karlsruhe von 2001 bis 2005 entwickelt. Es handelt sich um ein Framework, mit dem das Ziel verfolgt wird, semantische Technologien Forschungs-

[1] `http://kaon.semanticweb.org/`

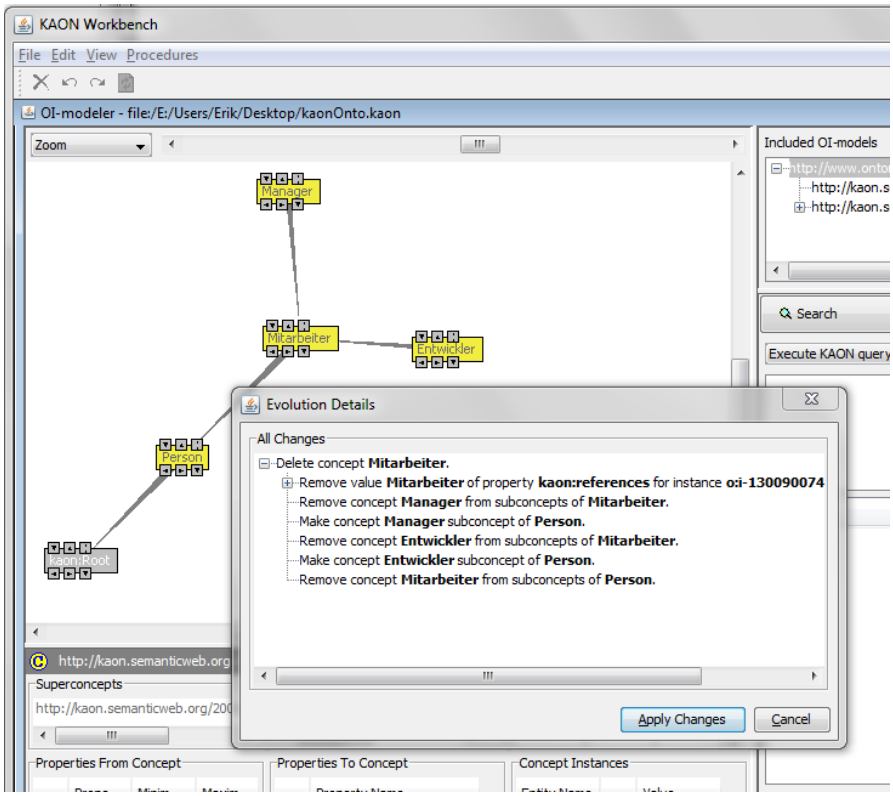

Abbildung 5.3: Ontologie-Evolution in KAON

und Industrieprojekten zugänglich zu machen. Es ist insbesondere darauf ausgelegt, semantische Technologien in E-Commerce-Szenarien zum Wissensmanagement und zur automatischen Generierung von Web-Portalen anzuwenden.

Die Erkenntnisse aus der Doktorarbeit von Ljiljana Stojanovic zur Ontologie-Evolution sind in die Implementierung von KAON eingeflossen, wodurch dieses Framework einige Besonderheiten in diesem Bereich aufweist [Sto04]. Eine Besonderheit besteht zum Beispiel darin, dass KAON alle Änderungen des Benutzers an der Ontologie überwacht und feststellt, ob durch eine Änderung die Ontologie inkonsistent werden würde. Ist das der Fall, werden automatisch weitere Änderungen berechnet, die ausgeführt werden müssen, um die Konsistenz der Ontologie zu erhalten. Diese Änderungen werden dem Benutzer dann zur Bestätigung angezeigt, bevor sie ausgeführt werden. Wenn eine Klasse gelöscht wird, würden zum Beispiel die Unterklassen inkonsistent werden, da ihre Superklasse-Beziehungen auf ein nicht mehr existierendes Element zeigen. KAON erkennt diesen Umstand und bietet entsprechende Aktionen an, um die Konsistenz zu erhalten. Abbildung 5.3 zeigt eine Situation, in der die Klasse Mitarbeiter gelöscht werden soll. Die Unterklassen Entwickler und Manager werden daraufhin automatisch der Elternklasse Person zugeordnet.

So genannte Evolutionsstrategien kommen dann zum Einsatz, wenn es mehrere Möglichkeiten gibt, inkonsistente Zustände aufzulösen [Sto04, S. 91]. Im obigen Beispiel gibt es drei unterschiedliche Strategien, wie mit den verwaisten Unterklassen umgegangen werden kann. Sie können, wie gezeigt, der Superklasse der gelöschten Klasse zugewiesen, sie können der Wurzel-Klasse zugewiesen, oder sie können ebenfalls gelöscht werden. Solche Strategien können vom Benutzer global in einem Eigenschaften-Dialog festgelegt werden, wie in Abbildung 5.4 gezeigt. KAON wird anschließend, entsprechend der vorgegebenen Strategien, Änderungsoperationen zur Auflösung von Inkonsistenzen vorschlagen.

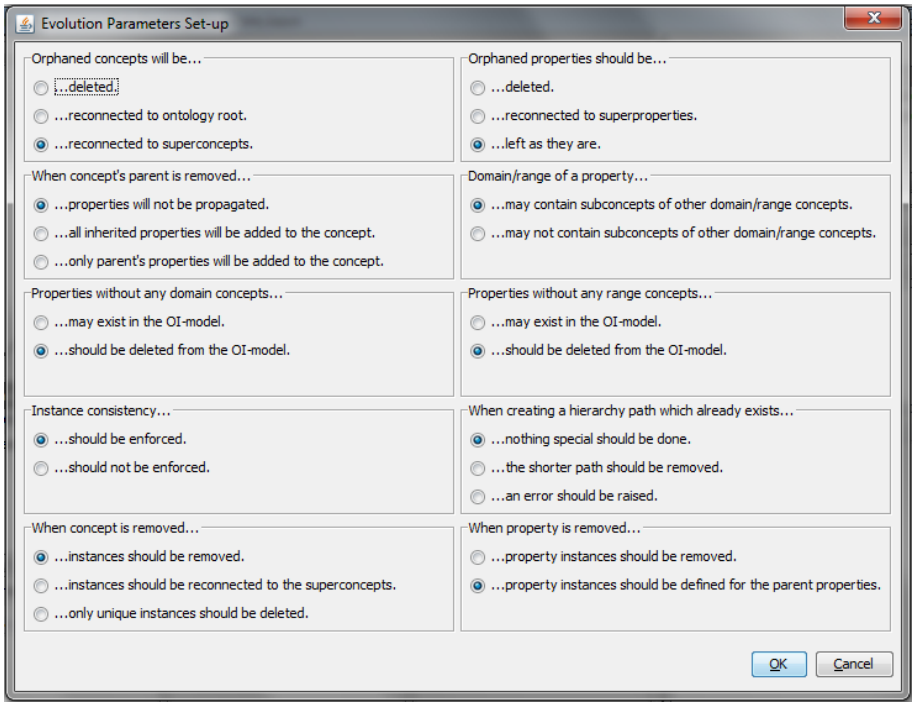

Abbildung 5.4: Evolutionsstrategien in KAON

Eine Undo/Redo-Funktion ermöglicht es, durchgeführte Änderungen jederzeit rückgängig zu machen oder wiederherzustellen. Eine Historie (Evolution-Log) zeichnet bei Bedarf sämtliche Änderungen auf. Somit können unterschiedliche Versionen der Ontologie verglichen und durchgeführte Änderungen auf andere Ontologien übertragen werden.

KAON setzt auf RDF und RDFS auf, erweitert jedoch beide Standards zu einer proprietären Ontologie-Beschreibungssprache, der KAON Ontology Language [Sto04]. Diese Sprache führt einige Konzepte ein, die Ähnlichkeiten zu Konzepten aufweisen, die später im Rahmen von OWL bzw. OWL 2 definiert werden. Dazu zählen Eigenschaften von Propertys (symmetrisch, transitiv, invers) und Kardinalitäten. Jedoch bestehen konzeptionelle Unterschiede zu OWL. Domain und Range sowie Kardinalitäten von Propertys sind in der KAON Ontology Language Restriktionen, in OWL werden diese Konzepte lediglich als Inferenzregel angesehen. In KAON kann die Anzahl von Individuen, die über eine bestimmte Property mit einem anderen Individuum verbunden sind mit Minimum- und Maximum-Werten angegeben werden. Werden über diese Property mehr oder weniger Individuen miteinander verbunden als angegeben, ist die Ontologie inkonsistent. In OWL hingegen dient die Kardinalität lediglich dazu, ableiten zu können, dass ein Individuum zu einer Klasse gehört, wenn es eine bestimmte Anzahl an Verbindungen über eine bestimmte Property hat. Stimmt die Anzahl der Verbindungen nicht mit der angegebenen Kardinalität überein, gehört das betreffende Individuum nicht zur Klasse. Die Ontology bleibt aber konsistent.

Stojanovic teilt Änderungsoperationen ebenfalls in unterschiedliche Granularitätsstufen ein. Sie unterscheidet zwischen elementaren, kompositen und komplexen Änderungen [Sto04, S. 60]. Elementare Änderungen sind, wie schon zuvor erwähnt, einfache Änderungen an einzelnen Elementen der Ontologie. Als Änderungsoperationen werden hier jedoch lediglich Hinzufügen und Entfernen betrachtet. Die Veränderung eines Elements wird durch Löschen und anschließendes Hinzufügen abgebildet. So ist sichergestellt, dass die Menge der elementaren Änderungen minimal ist [Sto04, S. 39]. Die von Stojanovic aufgestellten elementaren Änderungsoperatio-

nen umfassen unter anderem das Hinzufügen und Entfernen von Klassen, Propertys, Domain und Range sowie Elementen, die spezifisch für die KAON Ontology Language sind. Komposite Änderungen betreffen Elemente, die direkt miteinander in Beziehung stehen. Diese Änderungen ergeben sich aus den möglichen Beziehungen, der in der KAON Ontology Language definierten Konzepte. Dazu gehören zum Beispiel die Verschiebung einer Klasse in der Vererbungshierarchie, das Aufteilen oder Verschmelzen von Klassen oder das Kopieren einer Klasse [Sto04, S. 57]. Komplexe Änderungen können beliebige Elemente der Ontologie betreffen und werden aus einer Kombination von elementaren und kompositen Änderungen gebildet. Ein Beispiel ist die Verschiebung einer Menge von Geschwister-Klassen an eine andere Position innerhalb der Vererbungshierarchie [Sto04, S. 59].

Alle Änderungsoperationen für KAON sind so definiert, dass sie die Konsistenz der Ontologie erhalten. Die Erhaltung der Individuen nach Noy und Klein wird in KAON jedoch nicht betrachtet. Stojanovic führt eine formale Definition für elementare Änderungsoperationen ein. Diese Definition besteht aus einem Namen, der formalen Syntax, einer natürlichsprachigen Beschreibung der Änderungen sowie formalen Vor- und Nachbedingungen. Die Vorbedingungen müssen erfüllt sein, bevor die Änderung ausgeführt werden kann. Für das Hinzufügen einer Unterklassenbeziehung wird beispielsweise definiert, dass Super- und Unterklasse nicht gleich sein dürfen und bisher noch keine Unterklassenbeziehung besteht. Die Nachbedingungen prüfen die erfolgreiche Ausführung der Änderung. Im Beispiel wird lediglich das Vorhandensein der neuen Unterklassenbeziehung geprüft.

5.2.4 Das NeOn-Toolkit

Das NeOn-Toolkit ist ein auf Eclipse basierendes Open-Source-Framework zur Bearbeitung von verteilten Ontologien. Es wurde von 2006 bis 2010 im Rahmen des NeOn-Projekts[2] entwickelt, an dem die Open University, die Universidad Politecnica Madrid, das Karlsruhe Institute of Technology (KIT) und das Forschungszentrum Informatik (FZI) beteiligt waren. Die Verantwortlichen des Projekts gehen davon aus, dass sich Ontologien zu einer Schlüsseltechnologie entwickelt haben, um Daten und Prozesse in semantische Anwendungen zu integrieren. Dabei stellen Ontologien keine eigenständigen und unabhängigen Artefakte dar, sondern sind stark vernetzt, modular und hoch dynamisch. Dieser Sachverhalt wird mit dem Begriff *Networked Ontologies* beschrieben.

Ziel des NeOn-Projekts ist es, den gesamten Entwicklungs- und Lebenszyklus dieser Networked Ontologies zu untersuchen sowie Methoden und Technologien zu entwickeln, mit denen die Erstellung, Weiterentwicklung und Nutzung von verteilten Ontologien zur Anwendungsreife gebracht wird. Das heißt: Ontologien sollen gemeinschaftlich von vielen Anwendern erstellt werden, sie sollen in verteilten semantischen Anwendungen zum Einsatz kommen und sie sollen dynamisch weiterentwickelt werden können. Um diese Ziele zu erreichen, wurden eine Vielzahl von Methoden und Programmen entwickelt. Eines dieser Programme ist das NeOn-Toolkit.

In das NeOn-Toolkit wurden viele Funktionen von KAON übernommen, zum Beispiel die Konsistenzerhaltung bei Löschoperationen oder das Logging von Änderungen. Das NeOn-Toolkit setzt jedoch auf den Ontologie-Standards des W3C auf, nutzt die OWL-API[3] und unterstützt große Teile von OWL 2. Es existieren viele Plugins für das NeOn-Toolkit, die Funktionen zu unterschiedlichen Bereichen im Ontologie-Lebenszyklus zur Verfügung stellen. Es gibt Plugins zur grafischen Darstellung von Ontologien (KC-Viz, OntoModel), zur Einbindung vorhandener Konzepte anderer Ontologien (Watson), zum Mapping von Ontologien (R2O, FOAM) und zum Zugriff auf Ontologien über ein zentrales Verzeichnis (Oyster). Im

[2] `http://www.neon-project.org/`
[3] `http://owlapi.sourceforge.net/`

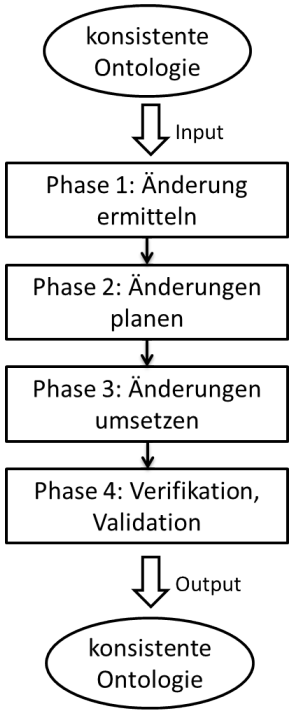

Abbildung 5.5: Evolutions-Prozess des NeOn-Projekts [PH10]

Rahmen dieser Arbeit sind diejenigen Plugins besonders interessant, welche für die Evolution von Ontologien eingesetzt werden: Evolva[4] und RaDON[5]. Diese beiden Plugins sollen im Folgenden näher vorgestellt werden.

Im NeOn-Projekt wurde der Evolutionsprozess, der ursprünglich von Stojanovic et al. aufgestellt wurde, weiterentwickelt und vereinfacht (vgl. Abbildung 3.5). Die Abbildung 5.5 zeigt den Evolutionsprozess, der dem NeOn-Toolkit zugrunde liegt [PH10]. Dieser Prozess beginnt und endet mit einer Ontologie in einem konsistenten Zustand, weil die Ontologie-Evolution die Handhabung von Änderungen einer Ontologie und die konsistente Einbringung dieser Änderungen ist. Die erste Phase dient dazu, mögliche oder notwendige Änderungen der Ontologie zu ermitteln und zu organisieren. In der zweiten Phase werden die Auswirkungen der Änderungen auf die Ontologie untersucht, bevor die Änderungen in der dritten Phase tatsächlich umgesetzt werden. Schließlich wird in der letzten Phase überprüft, ob die Änderungen den Wünschen und Vorstellungen entsprechen.

Palma et al. gehen von zwei verschiedenen Herangehensweisen aus, wie dieser Evolutionsprozess umgesetzt werden kann [PH10]. Beim *Bottom-up*-Verfahren werden Änderungen implizit, mit Methoden des maschinellen Lernens, ermittelt und automatisch in die Ontologie eingebracht [May09]. Dieses Verfahren wird im Evolva-Plugin umgesetzt. Beim alternativen *Top-down*-Verfahren werden Änderungen explizit vom Ontologie-Entwickler angegeben und durchgeführt. Anschließend wird untersucht, welche Auswirkungen diese Änderungen auf andere Teile der Ontologie hatten [Qi08]. Dieser Ansatz wird vom RaDON-Plugin unterstützt.

Betrachten wir zuerst das Bottom-up-Verfahren. Dieses soll Änderungen durch den Benutzer ermöglichen und gleichzeitig auch automatische Erweiterungen der Ontologie zulassen. Beide Anforderungen werden von Evolva erfüllt, wobei Evolva den kompletten Evolutionsprozess abdeckt [ZSDM09]. Die Benutzeroberfläche von Evolva ist in Abbildung 5.6 zu sehen. Um eine

[4] `http://evolva.kmi.open.ac.uk`
[5] `http://neon-toolkit.org/wiki/RaDON`

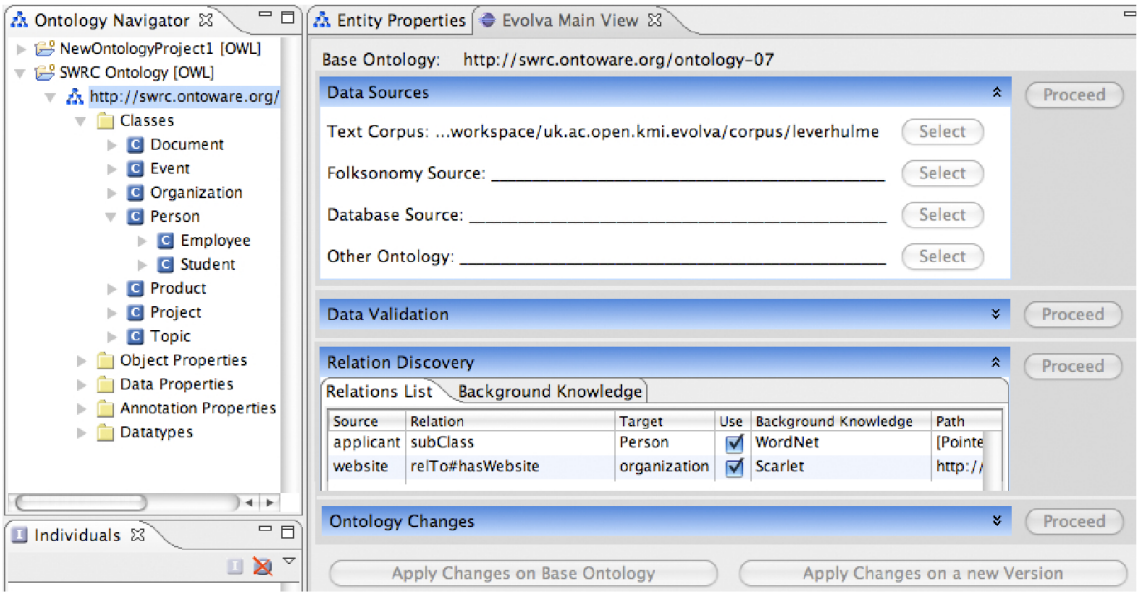

Abbildung 5.6: Oberfläche von Evolva [ZSDM09, S. 911]

Ontologie mit Evolva weiterzuentwickeln, wählt der Benutzer zuerst die gewünschte Ontologie oder nur einige Konzepte daraus aus. Anschließend durchsucht Evolva unterschiedliche Quellen nach passenden Erweiterungen. Diese Quellen können Texte in Form von Textdateien, PDF-Dokumenten oder HTML-Seiten, Folksonomies, Datenbanken oder andere Ontologien sein. Um Konzepte aus Text-Dokumenten zu extrahieren wird das Plugin Text2Onto verwendet, das Natural Language Processing (NLP) einsetzt, um relevante Begriffe aus natürlichsprachigen Texten herauszufiltern. Die gefundenen Konzepte werden dann mit den bestehenden Konzepten der Ontologie verglichen, wobei Konzepte aussortiert werden, die schon vorhanden sind oder aufgrund verschiedener Kriterien, wie z. B. zu kurze oder zu lange Wortlänge, als irrelevant eingestuft werden. An diesem Punkt kann der Benutzer in den Prozess eingreifen und auf die Auswahl der neuen Konzepte Einfluss nehmen. Wurde die Auswahl neu zu integrierender Konzepte getroffen, werden diese durch Evolva automatisch in die Ontologie integriert. Dazu wird Hintergrundwissen genutzt, um Konzepte in die richtige Stelle der Ontologie einzuordnen. Evolva stützt sich hierbei auf WordNet, das Beziehungen zwischen Wörtern der englischen Sprache enthält, und Scarlet, das Online-Ontologien durchsucht, um zu ermitteln in welchen Beziehungen bestimmte Begriffe stehen. Der Benutzer hat auch an dieser Stelle die Möglichkeit, auf die Ergebnisse Einfluss zu nehmen und die Integration seinen Wünschen anzupassen. Letztendlich hat der Benutzer die Wahl, die neuen Konzepte in die bestehende Ontologie zu integrieren oder eine neue Version der Ontologie mit den neuen Konzepten anzulegen.

Kommen wir nun zum Top-down-Verfahren, das sich mit der Frage beschäftigt, wie mit vom Benutzer durchgeführten Änderungen und daraus resultierenden Fehlern umgegangen werden soll [JHQ+09]. Es gibt zwei Möglichkeiten, auf Änderungen zu reagieren. Einerseits kann man Fehler, die durch Änderungen entstehen, einfach hinnehmen. Das erspart die Prüfung der Ontologie nach jeder Änderung. Das kann vor allem bei großen Ontologien mit vielen Änderungen eine große Erleichterung sein, insofern man mit eventuellen Fehlern in der Ontologie leben kann. Allerdings benötigt man dafür spezielle Reasoning-Techniken, die auch mit einer fehlerhaften Ontologie umgehen können. Andererseits kann man die Ontologie nach jeder Änderung auf Fehlerfreiheit prüfen.

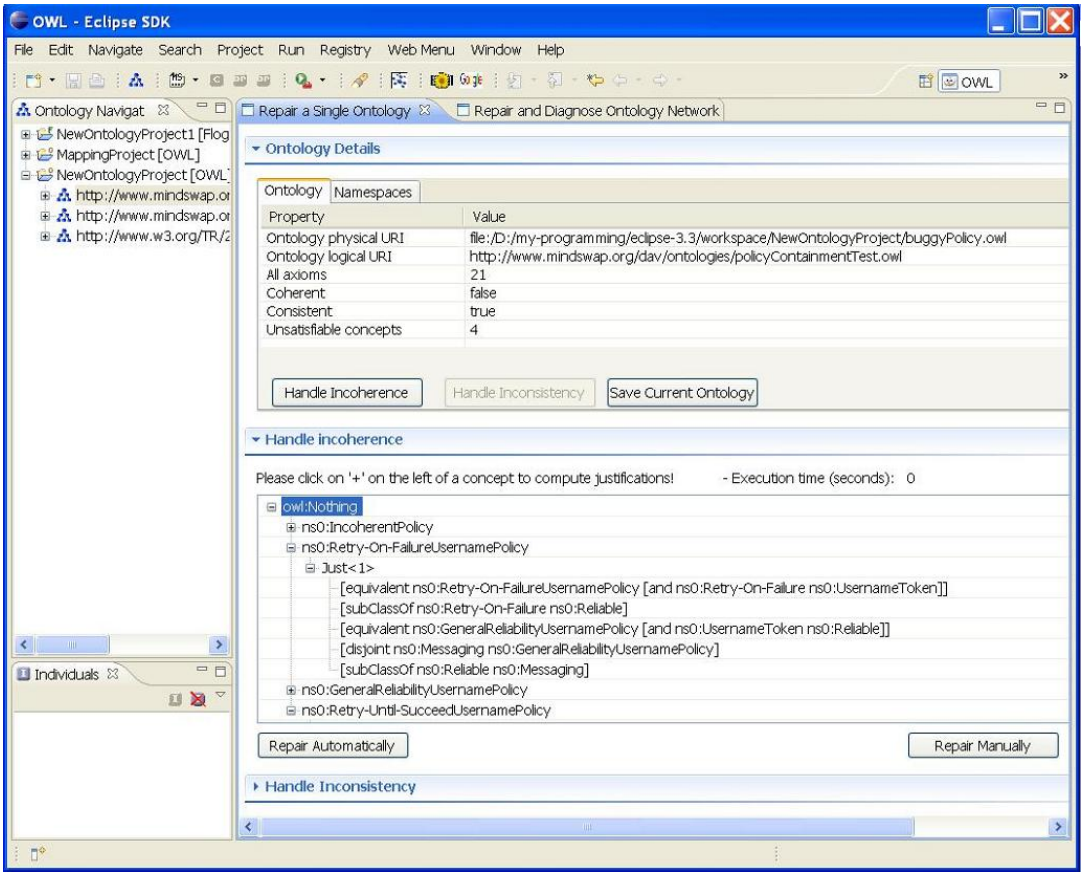

Abbildung 5.7: Oberfläche von RaDON

Der letztere Ansatz wird von RaDON (Repair and Diagnosis of Networked Ontologies) verfolgt. RaDON unterscheidet zwischen Inkonsistenz (ein Individuum gehört zu zwei disjunkten Klassen) und Inkohärenz (eine Klasse ist unerfüllbar, d. h. sie kann keine Individuen haben). Kommt es durch eine Änderung der Ontologie zu einer Inkonsistenz oder Inkohärenz, ermittelt RaDON die Ursache und zeigt sie dem Benutzer an. Dazu wird die vollständige Ableitung berechnet, die letztendlich zur Inkonsistenz oder Inkohärenz führt, wie die Abbildung 5.7[6] zeigt. RaDON ist in der Lage die gefundenen Fehler automatisch zu beheben. Dafür werden so genannte MIS (Minimal Inconsistent Subset) und MUPS (Minimal Unsatisfiability Preserving Subset) berechnet, die den kleinsten Teil der Ontologie darstellen, der fehlerhaft ist. Treten mehrere dieser MIS oder MUPS auf, entscheidet ein Rankingalgorithmus, welcher Bereich zuerst bereinigt werden soll. Die Bereinigung erfolgt, indem die Axiome der Ontologie, die den Fehler verursachen, gelöscht werden. Alternativ kann sich der Benutzer auch dafür entscheiden, die Bereinigung manuell durchzuführen. Dann führt RaDON durch den Reparaturprozess und schlägt Axiome, die Fehler verursachen, zur Löschung vor. Die wichtigste Errungenschaft von Ji, dem Entwickler von RaDON, ist die Entwicklung von parakonsistenz-basierten Algorithmen. Erst mit diesen Algorithmen ist es möglich, Reasoning über inkonsistenten Ontologien durchzuführen, und damit detaillierte Informationen über den Grund der Inkonsistenz zu erlangen. Parakonsistenz-basierte Algorithmen übersetzen dazu eine inkonsistente Ontologie automatisch in eine Ontologie, welche die maximale konsistente Teilmenge der inkonsistenten Ontologie darstellt.

[6] `http://neon-toolkit.org/w/images/02-computeMUPS.JPG`

Das NeOn-Toolkit unterstützt bislang keine komplexen Umstrukturierungen zur Weiterentwicklung der Ontologie. Alle Änderungen, die über die Erweiterung mit Evolva hinausgehen, müssen vom Benutzer auf elementarer Ebene manuell durchgeführt werden. Das NeOn-Toolkit hat diesbezüglich die Funktion von KAON übernommen, die Auswirkungen einer Änderung auf andere Teile der Ontologie zu bestimmen und dem Benutzer zur Bestätigung anzuzeigen. Wird beispielsweise eine Klasse gelöscht, müssen alle SubClassOf-Axiome bestehender Unterklassen ebenfalls gelöscht werden, damit die Ontologie konsistent bleibt. Nach einer Änderung zeigt das NeOn-Toolkit alle Axiome an, die hinzugefügt oder entfernt werden müssen, um Semantik und Konsistenz zu erhalten und lässt den Benutzer entscheiden, welche Änderungen durchgeführt werden sollen.

Das NeOn-Toolkit unterstützt also nicht nur die Erhaltung der Konsistenz sondern auch die Erhaltung der Individuen bezüglich der Verträglichkeitsdimensionen aus Abschnitt 3.4. Dies geschieht dadurch, dass dem Benutzer angezeigt wird, welche Auswirkungen eine Löschoperation auf den aktuellen Datenbestand hat.

Für das NeOn-Toolkit ist weiterhin eine vollständige Undo/Redo-Funktionalität konzipiert, die aber noch nicht Bestandteil der Implementierung ist. Durchgeführte Änderungen an einer Ontologie werden im NeOn-Toolkit vom Change Capturing-Plugin aufgezeichnet und stehen als Historie oder zur Übertragung auf andere Ontologien zur Verfügung.

5.2.5 Protégé

Ein weit verbreitetes Framework zur Erstellung, Bearbeitung und Nutzung von Ontologien ist Protégé[7] [NTNM10, NND+09]. Es wird vom Stanford Center for Biomedical Informatics Research[8] von der Stanford University School of Medicine in Zusammenarbeit mit der University of Manchester entwickelt. Die Entwicklung von Protégé wurde maßgeblich durch dessen Einsatz in unterschiedlichen Anwendungsszenarien beeinflusst[9]. Theoretisch-wissenschaftliche Erkenntnisse haben keinen so großen Einfluss auf die Entwicklung, wie es bei KAON oder beim NeOn-Toolkit der Fall ist.

Protégé basiert auf einer modularen Java-Plattform, die Erweiterungen über Plugins ermöglicht. Der Zugriff auf Ontologien erfolgte bis zur Version 3.x über die Protégé-OWL-API. Seit der Version 4.x wird stattdessen die OWL-API eingesetzt und der komplette OWL 2-Standard unterstützt. Mit dieser API können Ontologien in unterschiedlichen Syntaxen geladen, bearbeitet und gespeichert werden. Die Oberfläche von Protégé ist in einer Vielzahl von Ansichten organisiert, die bestimmte Elemente (Klassen, Propertys, Individuen) und die dazugehörigen Axiome anzeigen. Die Eingabe erfolgt bei Protégé, ähnlich zum NeOn-Toolkit, über Eingabefelder. Zur Eingabe einzelne Elemente, wie zum Beispiel Klassen-Ausdrücken, kommt dabei die Manchester OWL Syntax zum Einsatz.

Grundsätzlich kann der Benutzer die Ontologie frei bearbeiten und beliebig Elemente hinzufügen, bearbeiten oder entfernen. Protégé stellt, wie das NeOn-Toolkit, bei Löschoperationen sicher, dass die Konsistenz der Ontologie erhalten bleibt, indem weitere Elemente automatisch entfernt werden. Wird zum Beispiel eine Klasse gelöscht, werden alle Referenzen auf diese Klasse ebenfalls entfernt. Der Benutzer wird hierüber aber nicht informiert. Beim Hinzufügen neuer Axiome stellt Protégé mit Hilfe von Eingabedialogen sicher, dass die Eingaben wenigstens syntaktisch korrekt sind. In der Version 4.1.0 bietet Protégé bereits einige Änderungsoperationen an, die von der direkten Modifikation einzelner Ontologie-Elemente abstrahieren. Diese Änderungen werden in Edits und Refactorings unterteilt. Wie in Tabelle 5.3 zu sehen, fügen Edits der Ontologie oder ausgewählten Elementen neue Axiome hinzu oder entfernen vorhandene.

[7] http://protege.stanford.edu/
[8] http://bmir.stanford.edu/
[9] http://protege.stanford.edu/community/projects.html

Tabelle 5.3: Protégé Edits

Edits	**Kommentar**
Duplicate selected class...	Klasse mit allen Axiomen kopieren
Convert to primitive class Convert to defined class	Umwandlung der Klassendefinition
Add covering axiom	Zusätzliches Axiom hinzufügen
Make all individuals distinct... Make primitive siblings disjoint Remove disjoints for subclasses... Remove all disjoint axioms...	Distinct- und Disjoint-Klauseln hinzufügen oder entfernen

Tabelle 5.4: Protégé Refactorings

Refactorings	**Kommentar**
Change entity URI... Change multiple entity URIs... Change ontology URI... Convert entity URIs to labels...	Umbenennung
Convert property assertion on class/individual puns to annotations Coerce data property values into property range	Hinzufügen von Annotationen; Werte auf Wertebereich einschränken
Split subclass axioms Amalgamate subclass axioms Split disjoint classes into pairwise disjoints Amalgamate disjoint classes into larger disjoint sets	Schreibweise von Axiomen ändern
Convert qualified min cardinality 1 to someValuesFrom	Semantisch äquivalente Elemente austauschen
Copy/move/delete axioms ...	Axiome in andere Ontologien kopieren, verschieben oder Löschen
Merge ontologies...	Konzepte mehrerer Ontologien in einer Ontologie vereinen

Teilweise werden dazu Informationen aus anderen Teilen der Ontologie verwendet. Das Edit *Add covering axiom* erstellt beispielsweise ein SubClassOf-Axiom, das die Vereinigung aller direkten Unterklassen enthält.

Die Tabelle 5.4 zeigt die Refactorings von Protégé. Sie verändern nicht die Struktur der Ontologie, sondern lediglich die Schreibweise von Elementen. Ausnahmen davon bilden die beiden Refactorings *Copy/move/delete axioms* und *Merge ontologies*. Ersteres erlaubt das kontrollierte Kopieren, Verschieben oder Löschen von Elementen, wobei Protégé alle betroffenen Axiome ermittelt und dem Benutzer anzeigt. Letzteres vereint Konzepte mehrerer Ontologien in einer gemeinsamen Ontologie. Mit allen anderen Refactorings können die Bezeichner von Elementen geändert, Annotationen hinzugefügt und semantisch äquivalente Schreibweisen gegeneinander ausgetauscht werden. Die Superklassen-Beziehungen einer Klasse können beispielsweise als separate Axiome oder als ein einziges Axiom geschrieben sein. Das Listing 5.1 zeigt diese beiden äquivalenten Schreibweisen für eine Klasse in Manchester OWL Syntax. Die Einschränkungen *min 1* und *someValuesFrom* sind ebenfalls semantisch äquivalent, da `some` bezüglich der OWL-Semantik „mindestens 1“ bedeutet.

Listing 5.1: Unterschiedliche Schreibweise von Axiomen in Manchester OWL Syntax

```
1 # drei getrennte SubClassOf-Axiome
2 Class: Entwickler
3   SubClassOf:
4     Mitarbeiter,
5     kenntTechnik some Technik,
6     hatKollege some Mitarbeiter
```

Tabelle 5.5: Bewertung der Systeme zum Ontologie-Refactoring

Anforderung	KAON	NeOn-Toolkit	Protégé
1) Abdeckung	–	–	+
2) Steuerung durch den Benutzer	+	+	–
3) Transparenz	+	+	o
4) Umkehrbarkeit	+	–	+
5) Bedienbarkeit	–	+	o

```
7
8  # ein einzelnes SubClassOf-Axiom
9  Class: Entwickler
10   SubClassOf:
11     Mitarbeiter and (kenntTechnik some Technik) and (hatKollege some Mitarbeiter)
```

Wie gesagt, sind die vorgestellten Edits und Refactorings in Protégé implementiert. Allerdings existiert keine formale Definition der Semantik oder der Vor- und Nachbedingungen dieser Änderungen. Auch eine Einteilung in elementare und komplexe Änderungen findet nicht statt. Einige Refactorings betreffen einzelne Elemente (*Change entity URI*) andere betreffen möglicherweise mehrere Axiome an unterschiedlichen Stellen der ganzen Ontologie (*Convert qualified min cardinality 1 to someValuesFrom*). Protégé besitzt weiterhin eine vollständige Undo/Redo-Funktionalität. Das Aufzeichnen von Änderungen ist jedoch nicht vorgesehen.

5.2.6 Bewertung vorhandener Systeme zur Ontologie-Evolution

In diesem Abschnitt wurden Ansätze zur Ontologie-Evolution betrachtet. Die Arbeiten von Javed, Stojanovic sowie Noy und Klein stellen die theoretischen Grundlagen für das Änderungs-Management von Ontologien dar. Für die Bewertung sind jedoch nur die bereits implementierten Systeme interessant: KAON, NeOn-Toolkit und Protégé. Diese drei Programme stellen nur einen kleinen Ausschnitt aus einer Vielzahl von Ontologie-Werkzeugen dar [Den04]. Aktuelle Forschungsprojekte weisen den ausgewählten Programmen jedoch eine besondere Bedeutung zu, da diese besonders weit entwickelt und stark verbreitet sind [Cas10]. Damit ist die Relevanz der Auswahl gegeben. Die Tabelle 5.5 zeigt die Bewertung der Werkzeuge anhand der im Abschnitt 5.2.1 aufgestellten Anforderungen. Dabei bedeutet „+“, dass die Anforderung erfüllt wurde, „–“, dass die Anforderung nicht erfüllt wurde und „o“, dass keine Bewertung möglich ist. Die Bewertung bezieht sich auf die jeweils aktuelle Version der Werkzeuge zum März 2011: KAON 1.2.7, NeOn-Toolkit 2.4.2 und Protégé 4.1.0 beta.

KAON baut auf der proprietären KAON Ontologie Language auf, die nicht zu OWL 2 konform ist. Die Abdeckung des OWL 2-Standards ist damit nicht gegeben. Das NeOn-Toolkit unterstützt zumindest Teile des OWL 2-Standards, ermöglicht jedoch nur elementare Änderungen durch den Benutzer. Das Evolva-Plugin bietet recht umfangreiche Möglichkeiten, eine Ontologie um neue Konzepte zu erweitern, komplexe Umstrukturierungen sind jedoch nicht vorgesehen. Protégé unterstützt den kompletten OWL 2-Standard, ermöglicht die Bearbeitung aller Konzepte auf elementarem Level und stellt Edits bzw. Refactorings für komplexere Umstrukturierungen zur Verfügung.

Das Prinzip der Evolutionsstrategien ermöglicht in KAON eine detaillierte Steuerung von Ontologie-Änderungen durch den Benutzer. Das NeOn-Toolkit übernimmt diese Funktionalitäten von KAON. Im Unterschied zu KAON wird die Vorgehensweise jedoch bei jeder Änderung erneut abgefragt und nicht global festgelegt. In Protégé sind Maßnahmen zur Konsistenzerhaltung bei Änderungen fest implementiert und lassen sich durch den Benutzer nicht beeinflussen.

KAON zeigt einen umfangreichen Bericht aller Änderungen, die indirekt aus Änderungen des Benutzers resultieren, bevor diese tatsächlich durchgeführt werden. Die Transparenz ist damit

gegeben. Für das NeOn-Toolkit gilt das gleiche. Protégé zeigt Auswirkungen von Änderungen lediglich für das Refactoring *Copy/move/delete axioms* an.

Undo/Redo-Mechanismen sind in KAON und Protégé vorhanden. Für das NeOn-Toolkit ist eine entsprechende Funktion angedacht, jedoch noch nicht implementiert. Die Aufzeichnung aller durchgeführten Änderungen ist nur mit dem Evolution Log in KAON und dem Change Capturing-Plugin im NeOn-Toolkit möglich.

Mögliche komplexe Änderungen werden von KAON nicht angezeigt, lediglich elementare Änderungen an einzelnen Objekten. Eine Fehleranalyse im Falle einer Inkonsistenz wird nicht unterstützt. Mit den Plugins Evolva und RaDON bietet das NeOn-Toolkit eine umfangreiche Unterstützung in der Auswahl möglicher Erweiterungen einer Ontologie und der Analyse syntaktischer und semantischer Fehler. Mit Protégé sind elementare Änderungen und auf höherer Abstraktionsebene Edits und Refactorings möglich. Die Auswertung von inkonsistenten Zuständen ist jedoch abhängig vom eingesetzten Reasoner.

Um die Betrachtungen zur Ontologie-Evolution abzurunden, soll an dieser Stelle darauf hingewiesen werden, dass Änderungsoperationen von Ontologien stark von den eingesetzten Prinzipien des Ontologie-Engineering abhängen. So können je nach Ansatz der Wissensmodellierung unterschiedliche Umstrukturierungen sinnvoll sein. Stojanovic et al. gehen beispielsweise davon aus, dass sich die Ontologie nur dann in einem konsistenten Zustand befindet, wenn zu jeder Property eine Domain und eine Range explizit angegeben sind [SMMS02, S. 3]. Demgegenüber wird in modernen Quellen zum Ontologie-Engineering von der Verwendung von Domain und Range abgeraten [Hor11, S. 37], [NP10, S. 36]. Welche Änderungen für eine Ontologie angebracht sind, hängt also auch davon ab, welche Konzepte in der Ontologie eingesetzt werden.

5.3 Co-Evolution von Metamodellen und Modellen

An dieser Stelle sollen Techniken untersucht werden, die für ein Co-Refactoring, bei dem Refactorings auf unterschiedlichen Strukturen synchron durchgeführt werden, zum Einsatz kommen können. Dem Autor sind jedoch keine Techniken bekannt, die genau dies umsetzen können. Allerdings existiert mit der Co-Evolution ein Ansatz, welcher der Idee des Co-Refactorings nahe kommt. Weitere verwandte Techniken sind die Datenbank-Schema-Evolution und die Modell-Transformation. Die Datenbank-Schema-Evolution, zum Beispiel mit LiquiBase[10], ist dem Co-Refactoring ähnlich, beschränkt sich jedoch auf Änderungen des Datenbank-Schemas und dessen Auswirkungen auf den Datenbestand [Mah09]. Mit einer Modell-Transformation, wie sie beispielsweise durch den QVT-Standard (Query/View/Transformation, [OMG08]) beschrieben wird, könnte ein Co-Refactoring umgesetzt werden. Ein Co-Refactoring besteht aus dieser Perspektive betrachtet aus mehreren parallelen Modell-Transformationen, die synchronisiert werden müssen. Die Ansätze zur Datenbank-Schema-Evolution und Modell-Transformation sind jedoch für diese Arbeit nur von geringer Bedeutung. Einerseits beschäftigt sich diese Arbeit mit der Änderung von Ontologien, die sich in ihrer Struktur stark von Datenbanken unterscheiden und andererseits soll in dieser Arbeit mit Refactory ein Werkzeug eingesetzt werden, das Refatorings auf Modellen durchführen kann, weshalb eine Modell-Transformation nicht nötig ist. Im Folgenden wird also nur der Ansatz der Co-Evolution ausführlich betrachtet, um daraus mögliche Ansätze für ein Co-Refactoring abzuleiten.

Die Co-Evolution ist die automatische Migration von Modellen bei Veränderung der dazugehörigen Metamodelle [Wac07]. Wird in der MDSD ein Metamodell weiterentwickelt, ergibt sich das Problem, dass die zum Metamodell gehörigen Modelle unter Umständen nicht mehr

[10] `http://www.liquibase.org/`

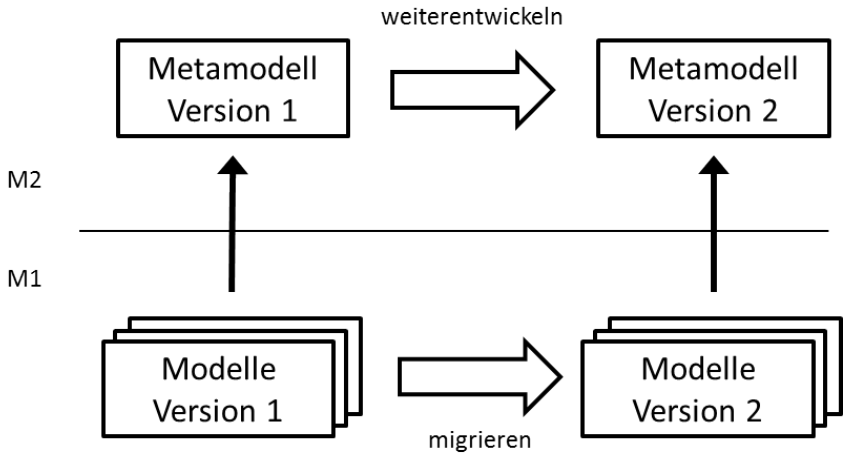

Abbildung 5.8: COPE-Vorgehensweise

konsistent sind. Um dieses Problem zu lösen hat Wachsmuth eine Klassifikation von möglichen Metamodell-Änderungen vorgeschlagen, die sich an objektorientierten Refactorings orientieren. Die Auswirkungen jeder Änderung dieser Klassifikation sind klar definiert, sodass bekannt ist, ob und in welcher Weise die Modelle angepasst werden müssen. Zur Migration der Modelle schlägt Wachsmuth eine Transformation mit QVT vor. Markus Herrmannsdörfer hat den Ansatz von Wachsmuth aufgegriffen, weiterentwickelt und in einem EMF-basierten Framework umgesetzt: COPE (Coupled Evolution of Metamodels and Models)[11] [Her10].

COPE basiert auf EMF und bietet die Möglichkeit einer Migration von EMF-Modellen bei einer Veränderung der entsprechenden Ecore-Metamodelle. Die Abbildung 5.8 zeigt die prinzipielle Vorgehensweise von COPE. Ein Metamodell wird durch den Benutzer erstellt und über die Zeit weiterentwickelt. Dazu kann der Benutzer das Metamodell direkt modifizieren oder vorgefertigte *Change Operations* verwenden. Jede Änderung am Metamodell wird dabei von COPE aufgezeichnet und in einer Historie gespeichert. In dieser Historie können mehrere Versionen zu einem Metamodell verwaltet werden, wobei sich einzelne Versionen aufgrund der ausgeführten Änderungsoperationen unterscheiden. Für jede Änderungsoperation wird außerdem eine Migrationsstrategie für die Modelle des Metamodells gespeichert. Wenn beispielsweise einer Klasse ein neues Attribut hinzugefügt wird, muss jedes Element in allen Modellen um dieses neue Attribut und einen entsprechenden Wert ergänzt werden. Für jede der Change Operations ist bereits eine Migrationsstrategie vordefiniert. Ändert der Benutzer hingegen das Metamodell manuell, muss er selbst für diese Änderungen eine Migrationsstrategie in Form eines Groovy-Scripts angeben. Jede Veränderung besteht also aus einer Modifikation des Metamodells und einer dazugehörigen Migrationsstrategie für die Modelle und wird daher auch als *Coupled Operation* bezeichnet.

Wurden alle nötigen Änderungen für eine neue Version des Metamodells durchgeführt, schließt der Benutzer die Änderungen in der Historie mit dem Anlegen einer neuen Version ab. Daraufhin besteht die Möglichkeit aus den Migrationsstrategien aller Änderungen dieser Version einen Migrator zu generieren, der Modelle älterer Versionen automatisch aktualisiert, sodass sie konform zur neuen Version des Metamodells sind.

COPE verfolgt den Ansatz sämtliche Änderungen des Metamodells von der Erstellung an aufzuzeichnen. Dieses ist der Operations-basierte Ansatz [Eys08]. Die Alternative dazu ist ein Differenz-basierter Ansatz, der zum Einsatz kommt, wenn bereits mehrere Versionen ei-

[11] `http://cope.in.tum.de`

nes Metamodells existieren und erst im Nachhinein eine Migrationsstrategie für die Modelle abgeleitet werden soll. Moritz Eysholdt hat in seiner Master-Arbeit den Differenz-basierten Ansatz für EMF umgesetzt [Eys09]. Dazu hat er zwei Komponenten entwickelt. *Epatch* ist eine Erweiterung von EMF Compare[12] und dient dazu, nachträglich zwei Versionen eines Metamodells zu vergleichen und ein Differenzmodell aufzustellen. Im Anschluss leitet *Metapatch* von diesem Differenzmodell eine Modell-Transformation ab, welche die Migrationsstrategie für die Modelle darstellt. Beide Ansätze, COPE und Epatch/Metapatch, werden im Rahmen des Edapt-Projekts[13] in das Eclipse Modeling Project aufgenommen.

Co-Evolution ist für diese Arbeit von zwei Gesichtspunkten aus interessant. Es stellt eine Weiterentwicklung von zwei unterschiedlichen Strukturen (Metamodelle und Modelle) dar und ist somit vergleichbar mit Co-Refactoring. Der Unterschied besteht jedoch darin, dass beide Strukturen nicht synchron geändert werden. Nur das Metamodell wird direkt verändert, die Veränderung der Modelle erfolgt indirekt über eine Migration. Ein weiterer Unterschied besteht darin, dass bei Co-Evolution eine natürliche Asymmetrie zwischen beiden Strukturen vorliegt. Zu einem Metamodell gibt es in der Regel mehrere Modelle. Wobei jedes Modell auf sein Metamodell verweist, aber das Metamodelle seine Modelle nicht kennt. Beim Co-Refactoring wird hingegen von einer Gleichberechtigung beider Strukturen ausgegangen. Das heißt, dass Änderungen in der einen Struktur entsprechend in der anderen Struktur repliziert werden, unabhängig von der Richtung.

Der zweite Gesichtspunkt, von dem aus Co-Evolution interessant ist, ist die Nutzung dieser Technik zur Datenmigration. In dieser Arbeit sollen Ontologien und Domänenmodelle weiterentwickelt werden. Wobei die Ontologien die Struktur und die Daten einer Domäne darstellen und somit auf der Seite der Domänenmodelle zu einem Metamodell und einem dazugehörigen Modell korrespondieren. Ein Refactoring auf der Seite der Domänenmodelle erfordert also die Migration eines Modells zu einem sich ändernden Metamodell. Diese Anforderung kann vom Ansatz der Co-Evolution umgesetzt werden.

5.4 Zusammenfassung

Dieses Kapitel gab einen Überblick über die bereits vorhandenen Ansätze und Techniken, welche für die Konzeption ontologiegetriebener Softwaresysteme interessant sind. Bezüglich der Abbildung von Ontologien auf Domänenmodelle wurden die beiden Ansätze betrachtet, Ontologien in der Modellierung (als Modellierungsartefakte oder als Erweiterung bestehender Modelle) einzusetzen und Ontologien mit Datenbanken zu verbinden. Im Anschluss wurden bestehende Ansätze zum Ontologie-Refactoring betrachtet. Dazu wurden die Ontologie-Frameworks KAON, NeOn-Toolkit und Protégé untersucht und bewertet. Der letzte Abschnitt behandelte die Co-Evolution. Unter den Ansätzen, die einem Co-Refactoring ähnlich sind, ist das der Wichtigste für diese Arbeit.

[12] `http://www.eclipse.org/modeling/emft/?project=compare`
[13] `http://www.eclipse.org/edapt/`

6 Konzept zur Entwicklung und Evolution ontologiegetriebener Softwaresysteme

6.1 Gesamtkonzept

In dieser Arbeit wird ein Ansatz zur Entwicklung und Evolution ontologiegetriebener Softwaresysteme erarbeitet. Ontologiegetrieben bedeutet, dass eine Ontologie zur Modellierung des Systems eingesetzt wird und gleichzeitig zentraler Bestandteil des erstellten Softwaresystems ist. Es muss demnach untersucht werden, wie das Design eines Softwaresystems mit Ontologien umgesetzt werden kann und wie Ontologien und Softwaresysteme in Beziehung stehen. Es wird nachfolgend zwischen dem Softwaresystem und der Anwendung unterschieden. Das Softwaresystem stellt die Gesamtheit aller Programme und Datenstrukturen dar, während die Anwendung lediglich den ausführbaren Teil des Softwaresystems beschreibt, der auf die Ontologie zugreift. Die Ontologie und die Anwendung sind demnach getrennte Teile eines Softwaresystems.

Wie in Abschnitt 5.1 gezeigt, gibt es bereits mehrere Ansätze, bei denen Ontologien oder semantische Techniken zum Entwurf oder als Bestandteil des Systems eingesetzt werden. Jedoch zielen diese Ansätze nicht darauf ab, Ontologien für das Design eines gesamten Systems zu verwenden. Softwaresysteme können mit Ontologien detaillierter beschrieben werden als mit anderen Modellierungstechniken wie der UML. Außerdem sind Design-Entscheidungen nicht mehr über viele unterschiedliche Techniken und Artefakte verteilt, wenn eine Ontologie als einziges Design-Artefakt eingesetzt wird.

Damit eine schnelle und einfache Softwareentwicklung erfolgen kann, sollte die Anwendung nach dem Vorbild der MDSD zu möglichst großen Teilen aus der Ontologie abgeleitet werden. Im Unterschied zur MDSD soll es der Anwendung jedoch möglich sein, auf die Ontologie zuzugreifen, um Daten und Informationen zur Struktur des Systems abzufragen. Es muss also geklärt werden, welche Möglichkeiten bestehen, Ontologien in Softwaresysteme einzubinden, sodass sie als Grundlage zur automatischen Generierung der Anwendung dienen können.

Ontologiegetriebene Softwaresysteme müssen wie andere Softwaresysteme beständig weiterentwickelt werden, damit sie neuen Anforderungen gerecht werden. Hierbei besteht jedoch die Herausforderung, die Ontologien und die daraus abgeleiteten Anwendungen synchron weiterzuentwickeln. Es muss also ein Konzept zur Evolution des Gesamtsystems aufgestellt werden, das die Beziehung zwischen Ontologien und Anwendung berücksichtigt.

Die vorangegangenen Überlegungen dieses Abschnitts lassen sich zu folgenden Anforderungen an ontologiegetriebene Softwaresysteme zusammenfassen.

Design (Domäne durch Ontologie darstellen) Um das Design des Gesamtsystems auf Basis der Ontologie vornehmen zu können, muss die Ontologie die komplette Anwendungsdomäne des Softwaresystems umfassen. Das heißt, die Ontologie dient nicht nur als Datenspeicher sondern bildet darüber hinaus anwendungsspezifische Details ab, sodass eine automatische Ableitung möglichst großer Teile der Anwendung aus ihr möglich wird.

Codegenerierung (Automatische Ableitung der Anwendung) Das mit der Ontologie erstellte Design muss letztendlich in eine ausführbare Anwendung umgewandelt werden. Diese Transformation sollte automatisch erfolgen, um den Entwicklungsaufwand gering zu halten.

Evolution (Design-Änderungen umsetzen) Da das Design und somit die Ontologie Änderungen unterworfen ist, muss sichergestellt sein, dass diese Änderungen konsistenzerhaltend in die Anwendung übertragen werden. Dies sollte eine möglichst geringe Einflussnahme des Entwicklers erfordern.

Betrachten wir nun die bereits existierenden Ansätze, die Ontologien in Softwaresysteme zu integrieren. Der einfachste Weg eine Ontologie in einem Softwaresystem zu nutzen, ist der Direktzugriff über eine API. Die OWL-API (Version 3) stellt die Referenzimplementierung für den OWL 2-Standard dar und wird von vielen aktuellen Ontologie-Frameworks genutzt (z. B. Protégé 4.x, NeOn-Toolkit). Ältere Frameworks (z. B. Protégé 3.x) nutzen die Protégé-OWL-API, die allerdings nicht konform zu OWL 2 ist. Die genannten Frameworks nutzen die APIs dazu, um auf Ontologien zuzugreifen und diese zu bearbeiten.

Ontologie-APIs können aber auch dazu genutzt werden, anwendungsunabhängig auf Ontologien zuzugreifen und diese somit als Datenspeicher zu verwenden. Nyulas et al. nutzen beispielsweise die Protégé-OWL-API, um eine Datenbank als Speichertechnologie gegen eine Ontologie auszutauschen [NND+09]. Die oberen Anwendungsschichten wie Service- und Präsentationsschicht bleiben dabei unverändert. Mit der Protégé-OWL-API ist es dabei auch möglich, die Ontologie in einer Datenbank zu speichern, um effizient zugreifen zu können [NND+09, S. 5]. Das Schema der Datenbank ist dabei unabhängig von der Ontologie, es besteht lediglich aus einer einzigen Tabelle, welche die Ontologie und ihre Instanz-Daten enthält. Sollten sehr große Datenbestände einen noch effizienteren Zugriff erfordern, können *Tripelstores* eingesetzt werden [Lee04]. Tripelstores sind spezielle Datenbanken, die Ontologien in Form von RDF-Tripeln speichern. Der Zugriff erfolgt dann über Schnittstellen, wie Sesame[1] oder Jena[2].

Der Nachteil dieser Anbindung der Ontologie an die Anwendung ist, dass sich beides unabhängig weiterentwickeln kann. Die Konsistenz zwischen Ontologie und Anwendung muss also über manuelle Anpassungen sichergestellt werden. Um diese manuellen Anpassungen zu reduzieren, schlägt Liang eine zusätzliche Schicht im Softwaresystem vor, die zwischen Anwendung und Ontologie vermittelt [Lia05, S. 11]. Diese Zwischenschicht verwaltet unterschiedliche Versionen einer Ontologie und lenkt Anfragen der Anwendung an die richtige Version um.

Die Ansätze, eine Ontologie über eine API oder eine Zwischenschicht an eine Anwendung anzubinden, gehen jedoch nicht darauf ein, wie die Ontologie zur Gestaltung des Gesamtsystems eingesetzt werden kann. Der einzige Ansatz zur Entwicklung eines ontologiegetriebenen Softwaresystems, der dem Autor bislang bekannt ist, kommt von Knublauch [Knu04]. Knublauch setzt eine Ontologie als Basis für eine Semantic-Web-Anwendung ein, wobei Teile der Anwendung (z. B. die Benutzeroberfläche) mit Protégé automatisch aus der Ontologie generiert werden. Dieser Ansatz ähnelt dem der MDSD. Jedoch werden bei Knublauch die Domänenmodelle (in Form der Ontologien) nicht nur als Grundlage zur Codegenerierung, sondern außerdem als Bestandteil des Systems zur Laufzeit genutzt. Der Ansatz von Knublauch ist vielversprechend, da mit geringem technologischem Aufwand Ontologien direkt in Softwaresysteme integriert werden können. Das im Folgenden erarbeitete Konzept baut in wesentlichen Teilen auf dem Ansatz von Knublauch auf, erweitert ihn jedoch um die Betrachtung der Evolution von Ontologie und Softwaresystem.

[1] `http://www.openrdf.org/`
[2] `http://jena.sourceforge.net/`

Auf Basis der untersuchten Ansätze wird nun ein Konzept vorgestellt, das die Entwicklung ontologiegetriebener Softwaresysteme ermöglicht und gleichzeitig die Evolution dieses Systems berücksichtigt. Dieses Konzept basiert auf der MDSD und ist daher nicht auf eine bestimmte Art von Softwaresystemen eingeschränkt. Es wurden drei Anforderungen vorgestellt: Design, Codegenerierung und Evolution. Die Anforderung des Designs wird durch die Modellierung der Anwendungsdomäne mit einer Ontologie erreicht. Vorhandene Frameworks der MDSD, wie EMF, setzen die Anforderung der Codegenerierung um. Daraus folgt, dass eine Transformation der Ontologie in Artefakte der MDSD notwendig ist. Die Evolution letztendlich wird mit einem Refactoring beider Strukturen umgesetzt. Die Weiterentwicklung der Ontologie sollte dabei möglichst automatisch auf alle abhängigen Strukturen übertragen werden. Dafür wird das Konzept des *Co-Refactorings* eingeführt, das im Abschnitt 6.4 detailliert behandelt wird.

Der skizzierte Ansatz wird mit **OntoMore** (Ontologie-Modell-Refactoring) bezeichnet, da es sich um eine Abbildung von Ontologien auf Modelle und ein Refactoring über diesen beiden Strukturen handelt. Das Gesamtkonzept von OntoMore wird in Abbildung 6.1 dargestellt. Es besteht im Wesentlichen aus zwei Teilen: Der Transformation von Ontologien in Metamodelle bzw. Modelle und dem Co-Refactoring beider Strukturen. Die Ontologie stellt dabei eine vollständige Beschreibung der Anwendungsdomäne dar. Sie wird in ein Metamodell und ein Modell der MDSD transformiert, auf Basis derer die Codegenerierung erfolgen kann. Es ist zu beachten, dass die Ontologie die Struktur der Domäne und auch die Instanz-Daten enthält. Diese beiden Dinge sind in der MDSD getrennt. Deshalb wird die TBox der Ontologie in ein Metamodell und die ABox der Ontologie in ein Modell transformiert. Die Transformation erfolgt demnach über mehrere Metaebenen hinweg und muss außerdem sicherstellen, dass die erstellten Modelle konform zum Metamodell sind. Aus dem Metamodell kann im Anschluss die Struktur der Anwendung generiert werden. Diese greift auf die Modelle zu, wobei unterschiedliche Persistenz-Techniken, wie Dateien oder Datenbanken, zum Einsatz kommen können. Auf Basis des Metamodells können DSLs beispielsweise in textueller oder grafischer Form definiert werden, mit denen die Bearbeitung der Modelle vereinfacht wird. Alternativ kann die Anwendung über die Persistenz-Schicht auch direkt auf die ABox der Ontologie zugreifen, womit die Transformation der ABox in Modelle nicht mehr nötig ist. Um eine Weiterentwicklung des Gesamtsystems zu ermöglichen, müssen Änderungen in der Ontologie und dem Metamodell bzw. Modell synchron umgesetzt werden. Dazu werden Co-Refactorings zwischen diesen beiden Strukturen definiert.

Die nachfolgenden drei Abschnitte behandeln einzelne Teile des Gesamtkonzepts im Detail. Zuerst wird die Transformation von Ontologien in Metamodelle und Modelle ausführlich behandelt. Im Anschluss werden Refactorings für Ontologien und Metamodelle definiert, welche die Grundlage für Co-Refactorings bilden. Der letzte Abschnitt geht dann auf das Co-Refactoring im Detail ein.

6.2 OWL-Ecore-Transformation

Der erste Schritt in der Umsetzung des OntoMore-Ansatzes ist die Transformation einer Ontologie in ein MOF-konformes Metamodell und Modell. Die Ontologie wird zum Design des Softwaresystems eingesetzt und stellt aufgrund seiner formalen Notation sehr detaillierte Möglichkeiten zur Beschreibung der Anwendungsdomäne bereit. Das Metamodell hingegen bietet flexible Möglichkeiten der Codegenerierung. Mit einer Transformation der Ontologie in ein Metamodell können somit die Vorteile beider Techniken vereint werden.

Wie im Abschnitt 4.1 vorgestellt, wurde bereits mit OWLizer ein Werkzeug entwickelt, das eine Transformation von Metamodellen und Modellen in Ontologien durchführt. Dieser Ansatz soll in dieser Arbeit aufgegriffen und weiterentwickelt werden. Ziel ist es also, Transformations-

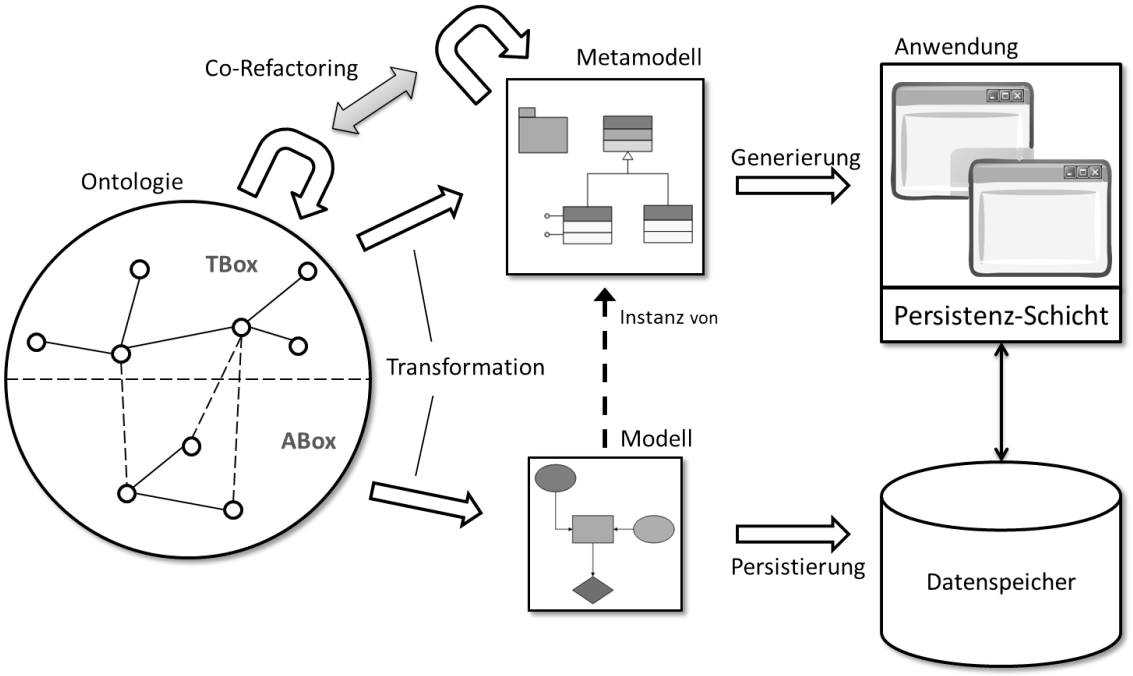

Abbildung 6.1: Gesamtkonzept von OntoMore

vorschriften zu definieren, die invers zu denen des OWLizer sind und eine korrespondierende Transformation von Ontologien zu Metamodellen bzw. Modellen zu erstellen. Durch die Erweiterung des OWLizer wird somit eine bidirektionale Abbildung geschaffen, die es ermöglicht, Ontologien und Metamodelle bzw. Modelle in beliebiger Richtung ineinander umzuwandeln. Somit können Änderungen in jedem der beiden Technologieräume vorgenommen und anschließend in die jeweils andere Technik übertragen werden.

Die inverse Transformation für den OWLzier beschreibt also eine Abbildung der TBox einer Ontologie auf ein Metamodell und eine Abbildung der ABox einer Ontologie auf ein Modell, das konform zum erstellten Metamodell sein muss. Die nachfolgend konzipierte Transformation stützt sich auf die vom OWLizer eingesetzten Techniken. Die Ontologie, die als Modell des OWL-Metamodells von OntoMoPP vorliegt, wird hierbei auf ein Ecore-Metamodell und ein dazugehöriges Modell abgebildet. Deshalb wird nachfolgend von einer **OWL-Ecore-Transformation** gesprochen.

Die Abbildung 6.2 zeigt das Prinzip einer Modelltransformation, wie es vom QVT-Standard beschrieben wird, im Vergleich zur OWL-Ecore-Transformation mit Bezug auf die Metaebenen. Die Transformation der Ontologie in das Metamodell und das Modell umfasst hierbei mehrere Metaebenen. Mit dem QVT-Ansatz wäre dies auch möglich, jedoch nur unter erheblichem Aufwand, da der Ansatz darauf nicht ausgelegt ist. So müsste eine zweistufige Transformation definiert werden, bei der die zweite Transformationsvorschrift auf Basis des erzeugten Metamodells dynamisch generiert wird. Der QVT-Ansatz transformiert also in der Regel ein Modell in ein anderes Modell derselben Metaebene. Sind die Metamodelle beider Modelle gleich, spricht man von einer endogenen Transformation. Sind sie unterschiedlich, spricht man von einer exogenen Transformation [MCG05, S. 2]. Die Zuordnung der Artefakte zu den Metaebenen ist in diesem Fall klar. Die Transformation wird auf Basis der Metamodelle auf Ebene Mn+1 definiert und auf den Modellen auf Ebene Mn durchgeführt. Das heißt, die Metamodelle lassen sich eindeutig der Ebene M2 und die Modelle der Ebene M1 zuordnen.

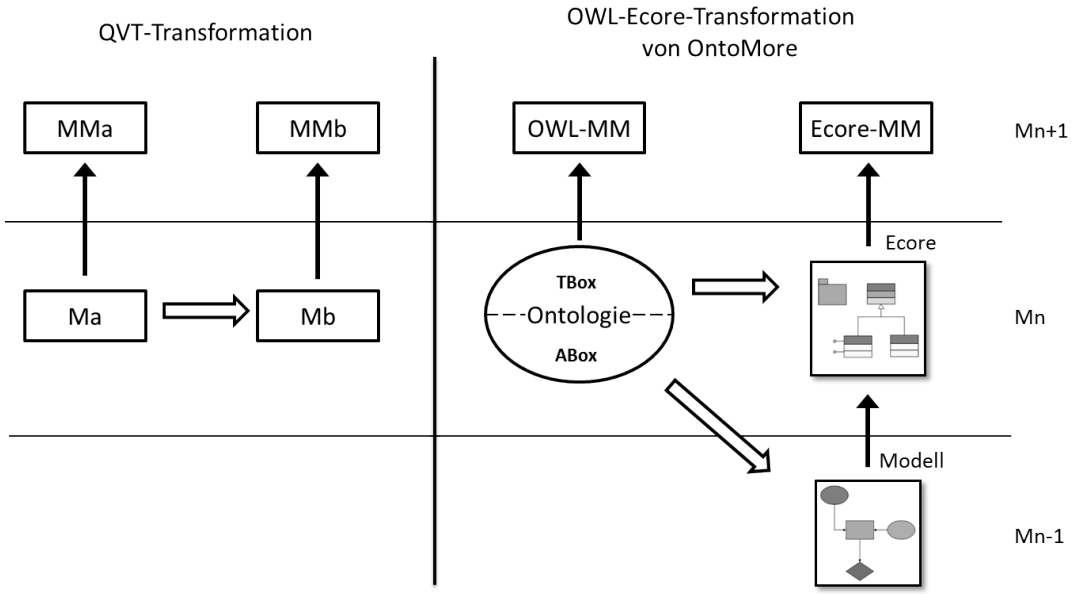

Abbildung 6.2: QVT und OWL-Ecore-Transformation im Vergleich

Die OWL-Ecore-Transformation ist eine exogene Transformation, da die Metamodelle, auf Basis derer die Transformation definiert wird, unterschiedlich sind. Dabei spielt das Ecore-Modell auf der Ebene Mn einerseits die Rolle eines Modells, in der Transformation aus der TBox, und andererseits die Rolle eines Metamodells für das aus der ABox generierte Modell. Das Ecore-Modell befindet sich also, je nach Betrachtungsweise, auf der Ebene M1 oder der Ebene M2.

Zur Definition der OWL-Ecore-Transformation wurde eine eindeutige Zuordnung der Elemente aus dem OWL-Metamodell von OntoMoPP auf Elemente des Ecore-Metamodells erarbeitet. Die Abbildungen 6.3 und 6.4 zeigen stark vereinfachte Ausschnitte aus den Metamodellen von OWL und Ecore. Sie beschränken sich auf die Elemente, die für die Transformation von Bedeutung sind.

Die Tabelle 6.1 zeigt die definierte Abbildung, indem sie jeweils die Klassen der Metamodelle, die aufeinander abgebildet werden, gegenüber stellt. Die in Klammern aufgeführten Attribute und Referenzen werden in der angegebenen Reihenfolge aufeinander abgebildet. Dabei kommt es vor, dass ein Pfad aus mehreren Referenzen und Klassen in OWL einer einzelnen Referenz in Ecore zugeordnet wird. Ein solcher Pfad wird durch eine Liste dargestellt, in der die Referenzen getrennt durch $\rightarrow$ aufgelistet werden. Hinter den Referenzen steht dabei mit Doppelpunkt getrennt die konkrete Klasse, auf welche die Referenz verweist. Beispielsweise wird die reflexive *eSuperTypes*-Referenz in Ecore in OWL durch einen Pfad über die Klassen *Disjunction*, *Conjunction* und *ClassAtomic* beschrieben.

Einige der Abbildungen umfassen mehrere zu transformierende Elemente. Beispielsweise kann es mehrere Superklassen geben. In der Tabelle sind auch in solchen Fällen die Datentypen der konkreten Elemente anstelle der Listentypen angegeben. Bei der ersten Abbildung von *OntologyDocument* und *Ontology* auf *EPackge* wird die Ontologie-URI als *nsURI* übernommen. Damit sie auf *name* und *nsPrefix* abgebildet werden kann, muss sie jedoch gekürzt werden, um den Namenskonventionen zu genügen. Es ist zu beachten, dass die OWL-Ecore-Transformation lediglich den Fall berücksichtigt, dass eine einzelne Ontologie in ein einzelnes Ecore-Modell transformiert wird. Die Verallgemeinerung auf Networked Ontologies (Ontologien mit Import-Beziehungen) und mehrere in Beziehung stehende Ecore-Modelle wäre denkbar. Eine Onto-

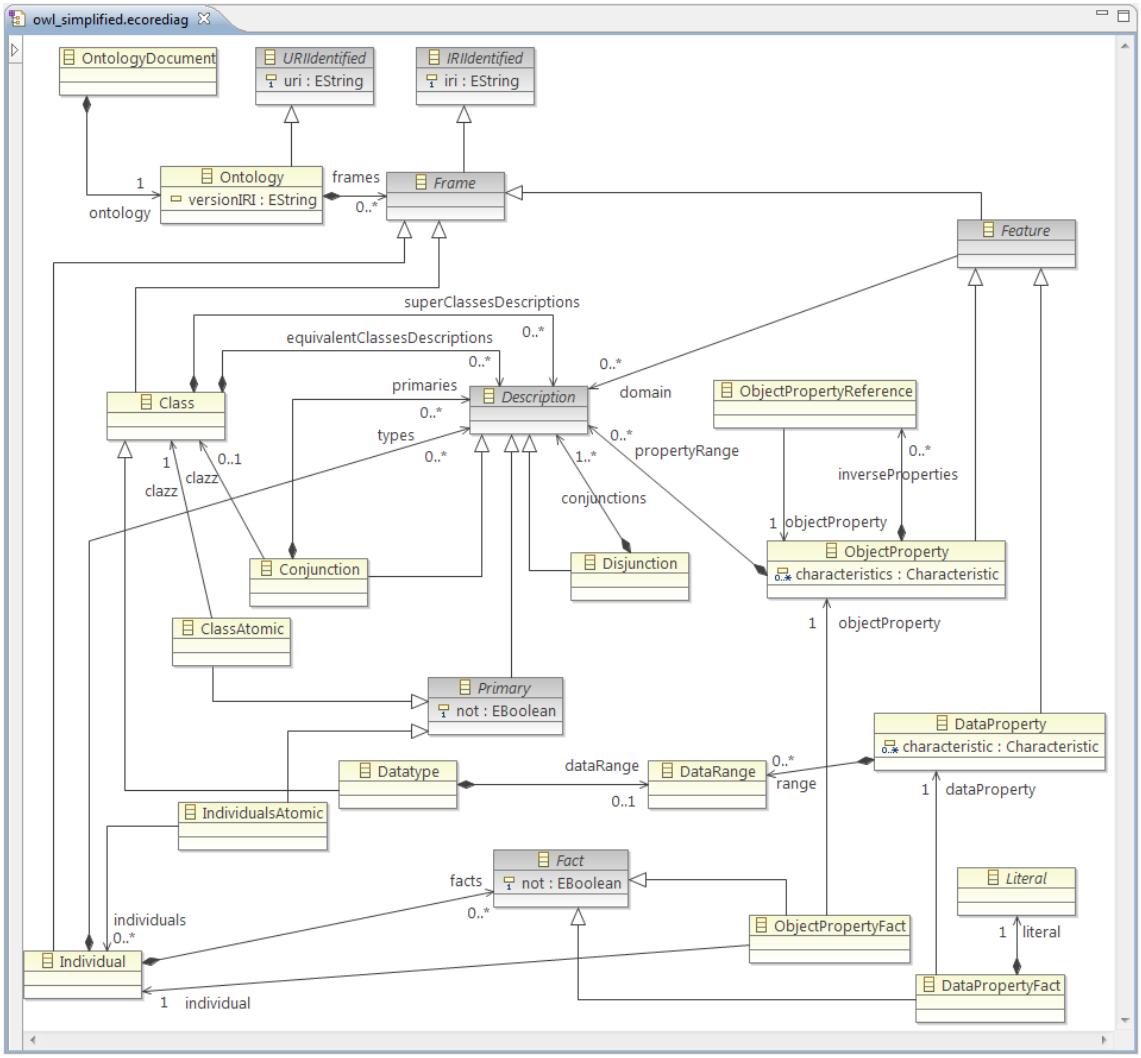

Abbildung 6.3: Ausschnitt aus dem OWL-Metamodell

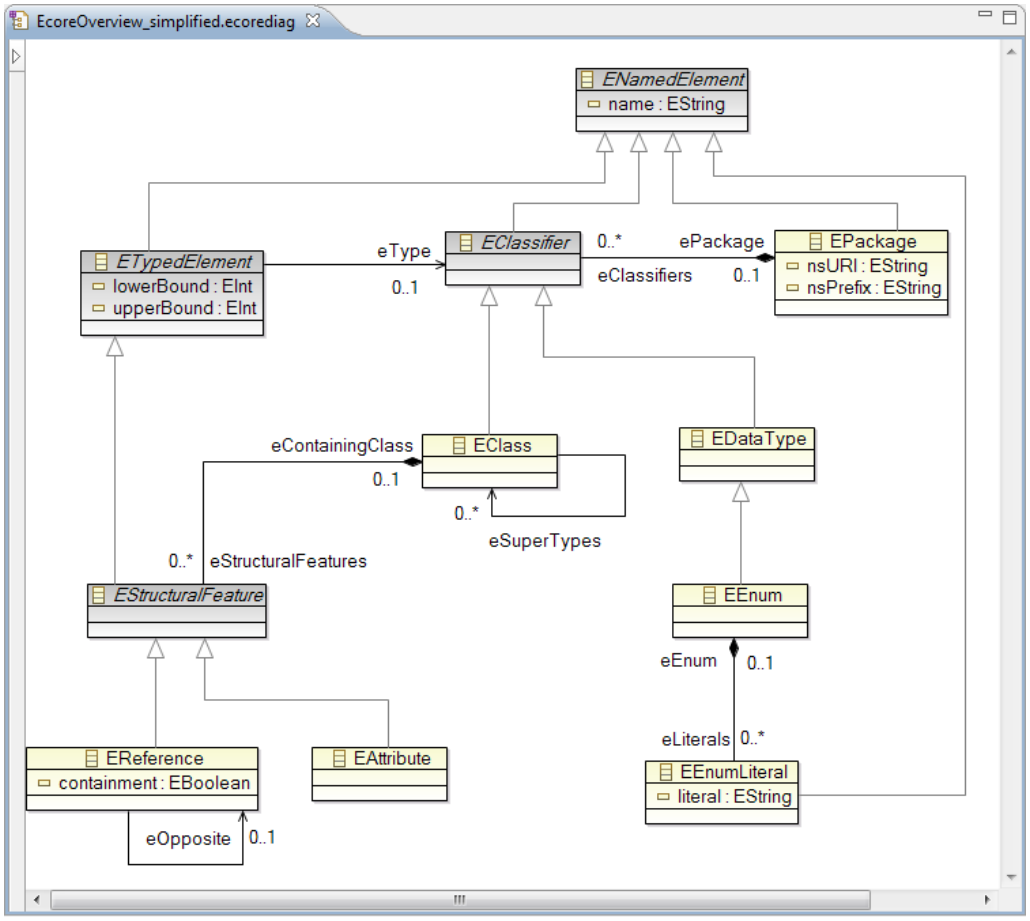

Abbildung 6.4: Ausschnitt aus dem Ecore-Metamodell

Tabelle 6.1: OWL-Ecore-Abbildung bezüglich der TBox

OWL	**Ecore**
OntologyDocument + Ontology(uri: string)	EPackage(name: EString, nsPrefix: EString, nsURI: EString)
Class(iri: string, superClassesDescriptions: Disjunction → conjunctions: Conjunction → primaries: ClassAtomic → clazz: Class)	EClass(name: EString, eSuperTypes: EClass)
ObjectProperty(iri: string, domain: Class, range: Class, Annotation: lowerbound, Annotation: upperbound, Annotation: containment, inverseOf: ObjectProperty)	EReference(name: EString, containingEClass: EClass, eType: EClass, lowerBound: EInt, upperBound: EInt, containment: EBoolean, eOpposite: EReference)
DataProperty(iri: string, domain: Class, range: DataRange)	EAttribute(name: EString, containingEClass: EClass, eType: EDataType)
Datatype(iri: string)	EDataType(name: EString)
Class(iri: string, equivalentClassesDescriptions: Disjunction → conjunctions: Conjunction → primaries: IndividualsAtomic → individuals: Individual(iri: string)	EEnum(name: EString) + EEnumLiteral(literal: EString)

logie würde immer genau einem Ecore-Package entsprechen, wobei beide über dieselbe URI identifiziert und somit einander zugeordnet werden können. Die ausführliche Betrachtung der Networked Ontologies liegt aber außerhalb des Rahmens dieser Arbeit.

Darüber hinaus wurde auch die Beziehung zwischen der ABox und dem Modell spezifiziert, wie in Tabelle 6.2 zu sehen. Die Abbildung besteht hierbei zwischen den Elementen des OWL-Metamodells und den Elementen eines generischen Ecore-Modells, da das eigentliche Ecore-Modell erst durch die TBox-Transformation erstellt wird und vorher noch nicht bekannt ist. Das bedeutet, dass die *ObjectPropertyAssertion* und *DataPropertyAssertion* nicht auf konkrete Elemente von Ecore abgebildet werden können, sondern zur Laufzeit der Transformation dynamisch den generierten Objekten zugewiesen werden müssen. In der Tabelle 6.2 sind dafür die reflexiven Methoden *eSet* und *eGet* angegeben, mit denen der Wert eines *StructuralFeature* gesetzt werden kann, welches erst zur Laufzeit bekannt ist. Weiterhin ist zu beachten, dass die Namen (IRIs) von Individuen keine Entsprechung auf Ecore-Seite haben. EObjects werden ausschließlich über ihre Klasse sowie über ihre Attribute und Referenzen charakterisiert.

Die definierte Abbildung ist vollständig bezüglich der Ecore-Konzepte. Es existiert eine Abbildung auf jedes konkrete Element des Ecore-Metamodells, das eine sinnvolle Entsprechung im OWL-Metamodell hat. Ecore-Konzepte, auf die nicht abgebildet wird, sind EFactory, EOperation und EParameter. Das Konzept einer Factory sowie Operationen sind in OWL nicht vorgesehen. Annotationen werden nicht als Teil der Abbildung betrachtet, da sie zur Erweiterung von OWL dienen, um eine sinnvolle Transformation zu ermöglichen.

Es ist durchaus möglich, die Abbildung auf Basis anderer Konzepte zu definieren. So können beispielsweise PropertyRestrictions, anstelle von ObjectPropertys und DataPropertys auf Attribute bzw. Referenzen abgebildet werden. Damit ist es möglich, die Kardinalitäten direkt in der Ontologie ohne Annotationen zu modellieren. Allerdings ist in diesem Fall eine Unterscheidung der PropertyRestrictions von atomaren Superklassen notwendig, da PropertyRestrictions ihrer-

Tabelle 6.2: OWL-Ecore-Abbildung bezüglich der ABox

OWL	**Ecore**
Individual(types: Disjunction → conjunctions: Conjunction → primaries: ClassAtomic → clazz: Class)	EObject(eClass: EClass)
ObjectPropertyAssertion(objectProperty: ObjectProperty, individual: Individual)	eSet/eGet(eStructuralFeature: EReference, value: EObject)
DataPropertyAssertion(dataProperty: DataProperty, literal: Literal)	eSet/eGet(eStructuralFeature: EAttribute, value: Object)

seits Beschreibungen anonymer Klassen sind. PropertyRestrictions kommen hier deshalb nicht zum Einsatz, da dies nicht konform zur Abbildung des OWLizer ist und somit eine bidirektionale Transformation nicht gegeben wäre.

Die erarbeitete Abbildung unterliegt gewissen Einschränkungen. Wie schon in Abschnitt 5.1 behandelt, ist es nicht möglich, alle Konzepte einer Ontologie auf UML-Konzepte abzubilden. Dies gilt in gleicher Weise für Ecore, da es dem UML-Standard sehr ähnlich ist. Die Transformation kann also nur die Konzepte der Ontologie umfassen, die auch mit Ecore darstellbar sind. An dieser Stelle stellt sich die Frage, wie sich der Teil der Ontologie, der tatsächlich transformiert werden kann, klassifizieren lässt. Dazu wurde eine Einschränkung der Ontologien auf Basis der Beschreibungslogiken untersucht. Es hat sich jedoch herausgestellt, dass Beschreibungslogiken für eine derartige Klassifizierung nicht geeignet sind. Schon einfache Beschreibungslogiken, wie $\mathcal{ALC}$, enthalten Konzepte, die in Ecore nicht darstellbar sind. Dazu zählen beispielsweise globale Rollen, eigenständige Individuen und logische Mengenoperationen über Klassen, wie Vereinigung, Schnittmenge oder Komplement. Damit ist die Beschreibung des transformierbaren Teils einer Ontologie mit einer Teilmenge der Ontologie-Beschreibungssprache nicht möglich.

Ein anderer Ansatz den transformierbaren Teil einer Ontologie zu klassifizieren ist, die Auflistung aller Einschränkungen, die in ihrer Gesamtheit eben diesen Teil definieren. Dazu wurde das Konzept der **transformierbaren Ontologien** entworfen. Transformierbare Ontologien sind Ontologien, die sich vollständig in ein Ecore-Modell transformieren lassen. Sie werden durch Einschränkungen über OWL-DL definiert. Außerdem werden transformierbare Ontologien durch Annotationen erweitert, die Konzepte von Ecore beschreiben, die in OWL nicht dargestellt werden können. Somit ist eine direkte Abbildung von OWL nach Ecore möglich. Nachfolgend werden die Einschränkungen an transformierbare Ontologien aufgelistet, wobei Beispiele in Manchester OWL Syntax angegeben werden.

Einschränkungen der TBox

1. Es werden keine anderen Ontologien referenziert. Ausnahmen davon bilden die Imports der Standard-Präfixe: rdf, rdfs, xsd und owl.

2. Die Ontologie hat nur die folgenden Top-Level-Konzepte:
   - Klasse
   - ObjectProperty
   - DataProperty
   - Datentyp

3. Die Namen (IRIs) aller Klassen, ObjectPropertys, DataPropertys und Datentypen sind global eindeutig.
4. Annotationen können an beliebigen Elementen vorkommen. Sie werden nur im Bezug auf die lowerbound-, upperbound- und containment-Eigenschaften der ObjectProperty ausgewertet.
5. Superklassen des SubClassOf-Axioms sind immer atomar.
6. Jede ObjectProperty hat eine Domain und eine Range. Domain und Range sind immer atomare Klassen. Cardinality- und containment-Eigenschaften sind als Annotationen angegeben. Sind keine Annotationen angegeben, werden folgende Standardwerte angenommen: lowerBound=0, upperBound=1, containment=false. Ein Beispiel einer korrekten ObjectProperty ist in Listing 6.1 zu sehen.
7. Die folgenden Annotationen müssen in der Ontologie definiert werden, wenn sie in ObjectPropertys genutzt werden.
   - AnnotationProperty: lowerbound
   - AnnotationProperty: upperbound
   - AnnotationProperty: containment
8. Die containment-Beziehungen zwischen den Klassen müssen in einer Baumstruktur angelegt sein, die alle Klassen ausgehend von einer Wurzel-Klasse einschließt.
9. Jede DataProperty hat eine Domain und eine Range. Die Domain ist immer eine atomare Klasse. Die Range ist ein Datentyp der Ontologie oder einer der folgenden: xsd:string, xsd:int, xsd:float oder xsd:boolean. Ein Beispiel einer korrekten DataProperty ist in Listing 6.2 zu sehen.
10. Alle Klassen, die nicht in Unterklassenbeziehungen stehen müssen disjunkt sein. Nur damit ist eine semantische Übereinstimmung mit EClasses aus Ecore gegeben.

Listing 6.1: Beispiel einer ObjectProperty

```
Prefix: : <http://ontomore.org/ontoTest#>
Ontology: <http://ontomore.org/ontoTest>
...
ObjectProperty: freelancers
    Annotations:
        lowerbound "0",
        upperbound "-1",
        containment "true"
    Domain: Management
    Range: Freelancer
...
```

Listing 6.2: Beispiel einer DataProperty

```
Prefix: : <http://ontomore.org/ontoTest#>
Ontology: <http://ontomore.org/ontoTest>
...
DataProperty: employeeName
    Domain: Employee
    Range: xsd:string
...
```

Einschränkungen der ABox

1. Jedes Individuum gehört zu genau einer atomaren Klasse, die mit einer Class Assertion angegeben ist.

2. ObjectPropertys und DataPropertys können in Property-Assertions genutzt werden, um Individuen genauer zu beschreiben. Das Ziel jeder Property muss dabei ein Literal oder eine Referenz auf ein anderes Individuum sein.

3. Das erste Individuum entspricht dem Wurzel-Element in Ecore. Soll ein anderes Individuum das Wurzel-Element sein, kann es mit der Annotation rootelement entsprechend gekennzeichnet werden. Sofern verwendet, muss diese Annotation ebenfalls definiert werden. Ein Beispiel für ein entsprechendes Individuum ist in Listing 6.3 zu sehen.

4. Individuen können über beliebige ObjectPropertys verbunden sein. Für jedes Individuum muss es aber genau einen Pfad von containment-ObjectPropertys vom Wurzel-Element zu diesem Individuum geben.

Listing 6.3: Beispiel eines Individuums

```
Prefix: : <http://ontomore.org/ontoTest#>
Ontology: <http://ontomore.org/ontoTest>
...
AnnotationProperty: rootelement

Individual: MySystem
    Annotations:
       rootelement "true"
    Types: FreelancerSystem
    Facts:
       hasName "FreelancerSystem",
       hasPerson John,
       hasPerson Max

Individual: John
    Types: Person
    Facts:
        hasName "John",
        hasAge 35,
        shoeSize 8.5f
...
```

Für eine formale, maschinenlesbare Definition der Einschränkungen kann eine Regelsprache wie SWRL (Semantic Web Rule Language) zum Einsatz kommen [HPSB+04]. SWRL vereint Konzepte von OWL und der RuleML, ist jedoch unentscheidbar. Um einen entscheidbaren Formalismus zu definieren, ist demnach eine Meta-Ontologie angebracht, die alle Konzepte einer transformierbaren Ontologie enthält. Diese Ontologie kann Konzepte vorhandener Meta-Ontologien, zum Beispiel die der OMV-Ontologie (Ontology Metadata Vocabulary)[3], wiederverwenden. Eine Ontologie kann dann mit Konzepten dieser Meta-Ontologie definiert werden, was eine eindeutige Einteilung der Ontologie in einen transformierbaren und einen nicht-transformierbaren Teil erlaubt. Die Definition der Meta-Ontologie ist jedoch zu umfangreich, als das sie im Rahmen dieser Arbeit behandelt werden könnte.

Werden die definierten Einschränkungen eingehalten, haben die daraus resultierenden transformierbaren Ontologien die Ausdrucksstärke von Ecore. Konzepte von Ontologien, die nicht mir Ecore darstellbar sind, können dennoch in der Ontologie spezifiziert werden. Die transformierbaren Ontologien sind so konzipiert, dass die OWL-Ecore-Transformation nur den Teil der

[3] http://omv2.sourceforge.net/

Ontologie transformiert, der sich auch mit Ecore darstellen lässt, ohne eventuell vorhandene weitere Teile der Ontologie zu berücksichtigen. Somit wird sichergestellt, dass die größere Ausdrucksstärke von Ontologien tatsächlich zum Einsatz kommen kann. Auf nicht transformierte Teile der Ontologie kann die generierte Anwendung beispielsweise direkt über eine Ontologie-API zugreifen.

6.3 Refactorings für OWL und Ecore

6.3.1 Definition und Katalog von Ontologie-Refactorings

In diesem Abschnitt werden Refactorings definiert, die für OWL und Ecore zum Einsatz kommen können, um die Erstellung und Bearbeitung von Ontologien bzw. Ecore-Modellen zu vereinfachen. Die Refactorings dienen hier also speziell dazu, eine Ontologie bzw. ein Ecore-Modell weiterzuentwickeln. Dieser Ansatz unterscheidet sich von objektorientierten Refactorings darin, dass die Semantik bewusst verändert wird. Eine Beschränkung auf rein semantikerhaltende Änderungen, wie sie beispielsweise von den Protégé-Refactorings aus Tabelle 5.4 definiert werden, würde kaum einen Mehrwert für die Weiterentwicklung einer Ontologie bedeuten, da lediglich äquivalente Schreibweisen einzelner Elemente gegeneinander ausgetauscht werden und nicht die Struktur der Ontologie verändert wird.

Die in diesem Abschnitt vorgestellten Refactorings basieren auf den Untersuchungen aus Abschnitt 5.2 und berücksichtigen die Abbildung zwischen OWL und Ecore bezüglich der OWL-Ecore-Transformation. Das heißt, dass speziell die Refactorings ausgesucht wurden, die auf beiden Strukturen durchführbar sind. Dabei wird in Anlehnung an die Definition der Ontologie-Evolution von Haase und Stojanovic aus Abschnitt 3.3 folgende Definition eines Ontologie-Refactorings angenommen.

> Ein **Ontologie-Refactoring** ist eine konsistenzerhaltende Veränderung einer Ontologie, deren sämtliche Auswirkungen im Vorfeld angegeben werden können.

Die Definition besteht aus zwei Teilen. Im ersten Teil wird gesagt, dass die Veränderung konsistenzerhaltend sein muss. Es wird also davon ausgegangen, dass sich die Ontologie vor dem Refactoring in einem konsistenten Zustand befindet und es wird gefordert, dass die Ontologie syntaktisch und semantisch konsistent bleibt. Der zweite Teil bezieht sich auf die Erhaltung der Individuen. Es ist möglich, dass bestehende Daten der Ontologie gelöscht oder verändert werden müssen, um die Konsistenz zu erhalten. In jedem Fall muss aber im Vorfeld angegeben werden, welche Änderungen sich aus dem Refactoring ergeben. Nur so hat der Benutzer die Möglichkeit die Änderungen zu bewerten und deren Ausführung entweder zu bestätigen oder zu widerrufen.

Betrachten wir als Beispiel das Löschen einer Klasse. Diese atomare Änderung ist kein Ontologie-Refactoring, da die Ontologie unter Umständen inkonsistent wird, wenn andere Axiome der Ontologie auf diese Klasse verweisen. Würden ebenfalls alle Axiome gelöscht, die sich direkt oder indirekt auf diese Klasse beziehen, kann man immerhin von einer konsistenzerhaltenden Veränderung sprechen. Ein Ontologie-Refactoring liegt aber erst dann vor, wenn all diese Änderungen vor ihrer Ausführung explizit angegeben sind.

In dieser Arbeit wird nicht weiter auf elementare Änderungen, wie das Hinzufügen oder Löschen einzelner Klassen, eingegangen. Die in Abschnitt 5.2 vorgestellten Arbeiten von Javed, Stojanovic, Noy und Klein betrachten derartige Änderungen bereits sehr ausführlich [JAP09, Sto04, NK04]. Weiterhin bieten aktuelle Ontologie-Frameworks eine umfangreiche Unterstützung im Management elementarer Änderungen. So ist es möglich, die Auswirkungen von Einfüge- oder Löschoperationen einzelner Elemente direkt nachzuvollziehen. Werkzeuge,

die den Benutzer dabei unterstützen, sind zum Beispiel das in Abschnitt 5.2.4 vorgestellte Plugin RaDON oder der Reasoner Pellet[4], der eine Funktion zum inkrementellen Reasoning bietet [GHWK07].

Komplexe Umstrukturierungen, die aus mehreren elementaren Änderungen bestehen oder eine Analyse mehrerer Bereiche der Ontologie erfordern, werden von vorhanden Arbeiten jedoch noch nicht oder nur sehr geringfügig abgedeckt. Aus diesem Grund konzentriert sich diese Arbeit auf komplexe Ontologie-Refactorings, die sich aus mehreren Einzeloperationen zusammensetzen. Die Tabelle 6.3 zeigt den erarbeiteten Katalog von Refactorings. Dabei wird ein Ontologie-Refactoring jeweils dem zugehörigen Ecore-Refactoring gegenübergestellt und jedes Refactoring wird durch eine kurze Beschreibung näher erläutert. Im Anschluss daran wird auf die genaue Definition der Refactorings eingegangen.

Die aufgestellte Liste von Refactorings ist bei weitem nicht vollständig. Sie gibt lediglich einen Überblick der Ontologie-Refactorings, die speziell für Ontologien in Frage kommen, die auf Ecore-Modelle abgebildet werden können. Je nach Anwendungsfall und tatsächlicher Nutzung der Ontologie können weitere Refactorings hinzukommen. Darüber hinaus gibt es viele weitere Ontologie-Refactorings, die hier nicht aufgeführt sind, weil sich deren Auswirkungen nicht auf Ecore-Modelle übertragen lassen. Der Anhang A enthält eine kleine Auswahl dieser Refactorings.

Ein Katalog ähnlicher Refactorings für Ontologien, unabhängig von einer Abbildung auf Domänenmodelle, wurde bereits von Gröner et al. aufgestellt [GPS10]. Dies zeigt, dass die definierten Ontologie-Refactorings allgemeingültig sind und nicht von der Abbildung auf Ecore abhängen. Gröner et al. konzentrierten sich jedoch darauf, die Refactorings zu erkennen, nachdem sie manuell durchgeführt wurden. Im Unterschied dazu geht es hier darum, die Refactorings automatisch durchzuführen. Eine nachträgliche Bestimmung der Refactorings aus den durchgeführten Änderungen ist dann nicht mehr nötig, da offensichtlich ist, welches Refactoring ausgeführt wurde.

Einige der aufgestellten Refactorings sind an Refactorings aus dem Katalog von Fowler angelehnt [FBB+99]. Dies ist in der strukturellen Ähnlichkeit von Ecore und der objektorientierten Programmierung begründet. So befindet sich im Fowler-Katalog ebenfalls ein *Replace Data Value With Object*. *Pull Up Property* bzw. *Pull Up Feature* kommt in Form von *Pull Up Field* vor. Und *Introduce Inverse Property* bzw. *Introduce Inverse Reference* kommt als *Change Unidirectional Association to Bidirectional* vor. Diese Refactorings ähneln sich in ihrer Funktionsweise und unterscheiden sich vor allem aufgrund der unterschiedlichen Elemente, über denen sie ausgeführt werden, was letztendlich zu der unterschiedlichen Benennung führt. Dennoch muss darauf hingewiesen werden, dass die hier definierten Refactorings eine höhere Abstraktionsebene beschreiben, als die von Fowler beschriebenen Refactorings. Beispielsweise definiert das Refactoring *Replace Data Value With Object* lediglich die Änderung eines Attributs zu einer Referenz. Wie der Zugriff darauf erfolgt, wird nicht betrachtet. Fowler hingegen gibt für dieses Refactoring detaillierte Schritte an, wie Konstruktor sowie Get- und Set-Methoden der Klasse anzupassen sind (vgl. Abschnitt 2.2).

6.3.2 Schema eines Ontologie-Refactorings

Um die Refactorings über die Kommentare der Tabelle 6.3 hinausgehend zu beschreiben, ist eine exakte Definition nötig. In Anlehnung an Stojanovic, Fowler und Gröner et al. wird dazu folgendes Schema verwendet [Sto04, FBB+99, GPS10].

[4] `http://clarkparsia.com/pellet/`

Tabelle 6.3: Refactoring-Katalog für OWL und Ecore

Nr.	OWL-Refactoring	Ecore-Refactoring
1	*Duplicate Class* Eine neue Klasse wird erstellt und alle Axiome einer vorhandenen Klasse werden übernommen.	*Duplicate Class* Eine neue Klasse wird erstellt und die Superklassen-Beziehungen sowie alle Attribute und Refenzen werden von einer vorhandenen Klasse übernommen.
2	*Convert DataProperty to ObjectProperty* In OWL wird eine DataProperty in eine ObjectProperty umgewandelt. Die Domain bleibt erhalten. Als Range wird eine vorhandene oder neu erstellte Klasse angegeben.	*Replace Data Value With Object* Ein Attribut einer Klasse wird durch eine Referenz ersetzt. Der Typ der Referenz ist eine vorhandene oder neu erstellte Klasse.
3	*Move Property* Ersetzt die Domain einer Property durch eine andere Klasse.	*Move Feature* Verschiebt ein Attribut oder eine Referenz in eine andere Klasse.
4	*Pull Up Property* Die Domain einer Property wird durch eine ihrer Superklassen ersetzt. Spezialfall von Move Property.	*Pull Up Feature* Ein Attribut oder eine Referenz wird in eine Superklasse verschoben. Spezialfall von Move Feature.
5	*Push Down Property* Die Domain einer DataProperty oder ObjectProperty wird durch eine ihrer Unterklassen ersetzt. Invers zu Pull Up Property.	*Push Down Feature* Ein Attribut oder eine Referenz wird in eine Unterklasse verschoben. Invers zu Pull up Feature.
6	*Extract Class* Axiome einer Klasse werden in eine neu erstellte Klasse verschoben.	*Extract Class* Attribute oder Referenzen einer Klasse werden in eine neu erstellte Klasse verschoben.
7	*Extract Superclass* Spezialfall von Extract Class. Die neu erstellte Klasse ist eine Superklasse der Klasse, von der die Axiome extrahiert wurden.	*Extract Superclass* Spezialfall von Extract Class. Die neu erstellte Klasse ist eine Superklasse der Klasse, von der Attribute oder Referenzen extrahiert wurden.
8	*Extract Subclass* Spezialfall von Extract Class. Die neu erstellte Klasse ist eine Unterklasse der Klasse, von der die Axiome extrahiert wurden.	*Extract Subclass* Spezialfall von Extract Class. Die neu erstellte Klasse ist eine Unterklasse der Klasse, von der Attribute oder Referenzen extrahiert wurden.
9	*Inline Class* Alle Axiome einer Klasse werden in eine andere Klasse verschoben. Anschließend wird die erstere Klasse gelöscht. Inverse zu Extract Class.	*Inline Class* Alle Attribute und Referenzen einer Klasse werden in eine andere Klasse verschoben. Anschließend wird die erstere Klasse gelöscht. Inverse zu Extract Class.
10	*Collapse Hierarchy* Die Axiome aller Unterklassen einer ausgewählten Klasse werden in diese verschoben. Die Unterklassen werden gelöscht. Entspricht der Ausführung von Inline Class für alle Unterklassen.	*Collapse Hierarchy* Die Attribute und Referenzen aller Unterklassen einer ausgewählten Klasse werden in diese verschoben. Die Unterklassen werden gelöscht. Entspricht der Ausführung von Inline Class für alle Unterklassen.
11	*Introduce Inverse Property* Es wird eine neue ObjectProperty angelegt, deren Domain dem Range und deren Range der Domain einer ausgewählten ObjectProperty entspricht. Beiden Properties wird ein InverseOf-Axiom zur jeweils anderen Property hinzugefügt.	*Introduce Inverse Reference* Zu einer Referenz wird eine neue Referenz in die entgegengesetzte Richtung angelegt. Beide Referenzen stehen über die eOpposite-Eigenschaft miteinander in Beziehung.
12	*Remove Inverse Property* Die Property, die über ein InverseOf-Axiom mit einer ausgewählten Property in Beziehung steht, wird gelöscht. Invers zu Introduce Inverse Property.	*Remove Inverse Reference* Die Referenz, die über die eOpposite-Eigenschaft mit einer ausgewählten Referenz in Beziehung steht, wird gelöscht. Invers zu Introduce Inverse Reference.

Tabelle 6.4: Refactoring-Katalog für OWL und Ecore (Forts.)

Nr.	OWL-Refactoring	Ecore-Refactoring
13	*Replace SubClassOf with Property* Entfernt das SubClassOf-Axiom zu einer Superklasse und erstellt eine neue PropertyRestriction einer vorhandenen oder neu erstellten ObjectProperty zu dieser Superklasse.	*Replace Inheritance with Reference* Entfernt die Superklassen-Beziehung zwischen zwei Klassen und fügt der Unterklasse eine neue Referenz auf die Superklasse hinzu.
14	*Replace Property with SubClassOf* Entfernt eine PropertyRestriction einer Klasse und fügt dafür ein SubClassOf-Axiom zum Range der ObjectProperty ein. Invers zu Replace SubClassOf with Property.	*Replace Reference with Inheritance* Entfernt eine Referenz einer Klasse und fügt stattdessen eine Superklassen-Beziehung zum Ziel der Referenz hinzu. Invers zu Replace Inheritance with Reference.

Name Eine eindeutige Bezeichnung für das Refactoring.

Motivation Beschreibung der Ausgangssituation inklusive des Problems, das mit dem Refactoring gelöst werden soll.

Vorbedingungen Auflistung notwendiger Eingaben und Beschreibung der Bedingungen, die erfüllt sein müssen, damit das Refactoring ausgeführt werden kann.

Lösung Die vom Refactoring durchzuführenden Änderungen, basierend auf der ausgewählten Beschreibungssprache.

Nachbedingungen Beschreibung der Zusicherungen eines erfolgreich ausgeführten Refactorings.

Erhaltung der Individuen Betrachtung möglicher Auswirkungen des Refactorings auf den Datenbestand der Ontologie.

Beispiel Ein einfaches Beispiel, das zeigt, wie das Refactoring funktioniert.

Die Änderungen eines Refactorings, wie sie im Punkt Lösung der Definition angegeben werden, sollen sicherstellen, dass die syntaktische und semantische Konsistenz der Ontologie erhalten bleiben. Dabei wird laut der Definition von Ontologie-Refactorings davon ausgegangen, dass sich die Ontologie vor Ausführung des Refactorings in einem konsistenten Zustand befindet.

Eine besondere Rolle nimmt hierbei die Erhaltung der Individuen ein. Sie bestimmt, in welcher Weise die Refactorings Auswirkungen auf den Datenbestand (die Axiome der ABox) haben. Bezüglich dieser Auswirkungen lassen sich Refactorings in drei Arten aufteilen [Eys08, S. 51]: (1) Das Refactoring hat keine Auswirkungen auf den Datenbestand. Dies ist der Fall, wenn neue Elemente der Ontologie hinzugefügt werden, wie im Refactoring *Duplicate Class*. (2) Durch das Refactoring müssen Daten gelöscht werden, um die Konsistenz der Ontologie sicherzustellen. Dieser Fall tritt ein, wenn bestehende Elemente der Ontologie entfernt werden, zum Beispiel im Refactoring *Remove Inverse Property*. (3) Der Datenbestand kann automatisch in eine Form transformiert werden, die konsistent zur neuen Struktur der Ontologie ist. Dies ist bei Refactorings der Fall, bei denen bestehende Elemente der Ontologie verändert oder durch andere ersetzt aber nicht ersatzlos gelöscht werden, wie beim Refactoring *Convert DataProperty to ObjectProperty*. Für jedes Refactoring ist die Einteilung in eine der drei Arten einzeln zu untersuchen, da eine automatische Klassifizierung, wie bereits von Noy und Klein festgestellt wurde, nicht möglich ist [NK04]. Das Schema der Refactorings enthält einen dementsprechenden Punkt, der Auskunft darüber gibt, in welchem Umfang eine Datenmigration möglich ist.

Im Anhang B werden die Auswirkungen der hier vorgestellten Ontologie-Refactorings auf den Datenbestand im Detail behandelt.

6.3.3 Beispiel eines Ontologie-Refactorings

An dieser Stelle wird beispielhaft die vollständige Definition für das Ontologie-Refactoring *Convert DataProperty to ObjectProperty* aufgeführt, welches die Komplexität der aufgestellten Refactorings gut darstellt. Die Beschreibungen erfolgen informal in natürlicher Sprache, wobei sich die Ausführungen im Punkt Lösung auf den OWL 2-Standard beziehen [MPSP09]. Die einzelnen Anweisungen sind dabei durchnummeriert, um einen linearen Ablauf zu suggerieren. Die Ausführung der Anweisungen in einer anderen Reihenfolge ist jedoch gegebenenfalls auch möglich, falls keine Reihenfolge-Abhängigkeiten zwischen den einzelnen Schritten bestehen. Die Betrachtungen beschränken sich hier auf die Ontologie-Refactorings, da Umstrukturierungen von Ecore-Modellen und deren Auswirkungen auf den Datenbestand von Herrmannsdörfer im Rahmen von COPE schon ausführlich behandelt wurden [Her10].

Name

Convert DataProperty to ObjectProperty

Motivation

Eine DataProperty verbindet eine Klasse mit einem Datentyp. Das Ziel der Property soll aber strukturierte Informationen enthalten, die mit einem Literal nicht dargestellt werden können.

Vorbedingungen

- Der Name einer neuen oder vorhandenen Klasse muss eingegeben werden.
- Eine existierende DataProperty ist auszuwählen.

Lösung

1. Eine neue ObjectProperty wird erstellt. Der Name wird von der DataProperty übernommen.
2. Wenn noch keine Klasse mit dem angegebenen Namen existiert, wird eine neue Range-Klasse mit diesem Namen erstellt.
3. Die ObjectProperty erhält ein Range-Axiom mit der angegebenen bzw. neu erstellten Range-Klasse.
4. Das Domain-Axiom wird, sofern vorhanden, von der DataProperty übernommen.
5. Annotations- und Characteristics-Axiome werden ebenfalls von der DataProperty übernommen.
6. EquivalentProperties- und DisjointProperties-Axiome der DataPropertys werden, wenn nötig, angepasst.
7. Die ausgewählte DataProperty und all ihre Axiome werden gelöscht.

Nachbedingungen

- Die Ausgewählte DataProperty wurde gelöscht.
- Es existiert eine ObjectProperty mit dem Namen der ausgewählten DataProperty.
- Die ObjectProperty hat als Range eine existierende Klasse.

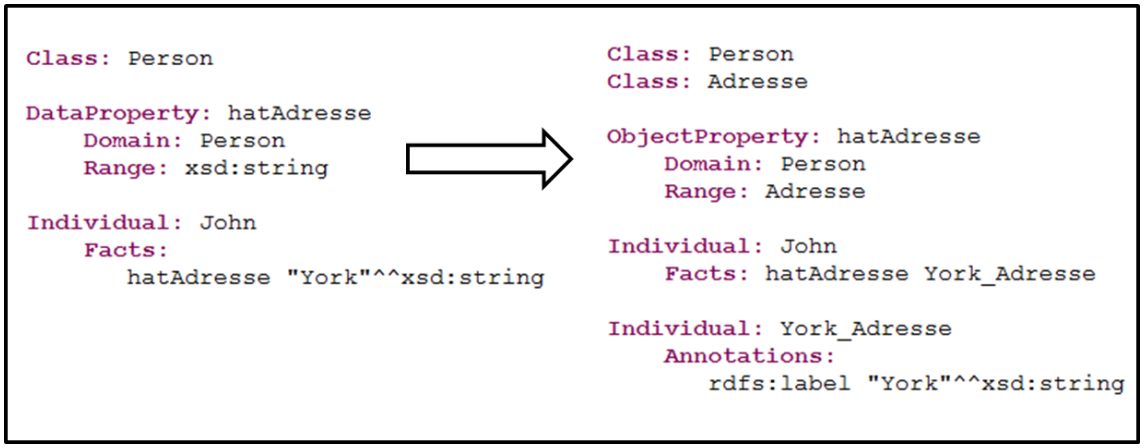

Abbildung 6.5: Beispiel für Convert DataProperty to ObjectProperty

Erhaltung der Individuen
Der Datenbestand wird inkonsistent, wenn die ausgewählte DataProperty bereits Individuen mit Literalen verbindet. In diesem Fall muss jede DataProperty-Assertion dieser DataProperty durch eine ObjectProperty-Assertion auf ein neues Individuum der Ziel-Klasse ersetzt werden. Die Inkonsistenz des Datenbestands lässt sich also automatisch auflösen.

Beispiel
Die Abbildung 6.5 zeigt einen Ausschnitt einer Ontologie in Manchester OWL Syntax mit einem einfachen Beispiel für dieses Refactoring. Die DataProperty hatAdresse wird durch eine entsprechende ObjectProperty ersetzt. Die Property-Assertion des Individuums wird automatisch angepasst und auf ein neu erstelltes Individuum umgelenkt. Das neue Individuum wird automatisch generiert, wobei das Literal der ersetzten DataProperty-Assertion als Label erhalten bleibt. Für die automatische Datenmigration sind auch andere Vorgehensweisen denkbar. So könnte das neue Individuum auch über eine DataProperty mit dem Literal verbunden sein.

6.4 Co-Refactoring

6.4.1 Definition und Ansätze

In diesem Abschnitt wird ein Co-Refactoring-Ansatz erarbeitet. Wie im Gesamtkonzept in Abbildung 6.1 gezeigt, ist ein synchrones Refactoring über unterschiedlichen Modellen nötig, um das beschriebene Konzept zur Entwicklung und Evolution ontologiegetriebener Softwaresysteme umzusetzen. Dieses synchrone Refactoring wird durch ein Co-Refactoring verwirklicht. Der hier erarbeitete Ansatz für Co-Refactoring ist jedoch völlig unabhängig von der Art der Modelle. Er ist also nicht auf Ontologien und Ecore-Modelle beschränkt.

Die Idee eines Co-Refactorings besteht also darin, jeweils ein Refactoring auf zwei (bzw. im allgemeinen Fall mehreren) Modellen gleichzeitig auszuführen. Dabei können sich die beiden Refactorings durchaus unterscheiden. Da keine Annahme über die Art der Modelle getroffen wird, können diese auch zu unterschiedlichen Metamodellen gehören. Dies erfordert unter Umständen völlig unterschiedliche Refactorings in den beiden Modellen. Generische Refactorings bieten nicht genügend Flexibilität, um alle diese Unterschiede abzudecken. Es müssen also konkrete Refactorings zum Einsatz kommen. Wobei diese konkreten Refactorings von den gleichen aber durchaus auch von unterschiedlichen generischen Refactorings abgeleitet sein können.

Im Unterschied dazu, jeweils ein Refactoring auf zwei Modellen manuell auszuführen, stehen die beiden Modelle beim Co-Refactoring in einer bestimmten Beziehung. Es wird angenommen, dass eine Relation zwischen den Metamodellen der Modelle existiert, auf Grundlage derer jedes Element des einen Modells mit entsprechenden Elementen des anderen Modells in Beziehung gesetzt werden kann. Diese Relation muss sicherstellen, dass die Änderungen eines Elements des einen Modells auf korrespondierende Änderungen von Elementen im anderen Modell übertragen werden können. Weitere Einschränkungen der Relation werden nicht vorgenommen. So ist es beispielsweise möglich, dass Änderungen in einem Element Änderungen an mehreren Elementen im anderen Modell erfordern, weshalb es sich im Allgemeinen nicht um eine eindeutige Abbildung handelt. Die Bijektivität der Beziehung ist ebenfalls nicht gegeben, da weder von Injektivität noch von Surjektivität ausgegangen werden kann. So kann es vorkommen, dass unterschiedliche Konzepte des einen Metamodells demselben Konzept des anderen Metamodells entsprechen. Außerdem genügt es, wenn sich die Relation auf die Konzepte der Metamodelle beschränkt, die durch die Refactorings beeinflusst werden.

Die Beziehung zwischen den Modellen muss auch nach dem Co-Refactoring noch erhalten sein, damit weitere Co-Refactorings ausgeführt werden können. Es sind also für jedes Modell Refactorings zu finden, die einander in der Art entsprechen, dass durch deren Ausführung die Relation nicht beeinträchtig wird. Über die Art und Weise der Ausführung des Co-Refactorings wird dabei keine Annahme getroffen. Es ist nicht von Belang, ob die einzelnen Refactorings tatsächlich gleichzeitig oder in einer bestimmten Reihenfolge ausgeführt werden, solange das Co-Refactoring als Ganzes die Beziehung zwischen den Modellen erhält. In gleicher Weise trifft ein Co-Refactoring keine Annahme über die Richtung. Es spielt keine Rolle auf welchem Modell der Benutzer das Refactoring zuerst ausführt und auf welche Modelle damit die entstehenden Änderungen übertragen werden müssen. Zusammenfassend lässt sich Co-Refactoring wie folgt definieren.

> **Co-Refactoring** ist die Ausführung zusammengehöriger Refactorings auf mehreren voneinander abhängigen Strukturen, wobei die Beziehung zwischen diesen Strukturen erhalten bleibt.

Es ergibt sich also die Frage, wie Änderungen eines Refactorings auf dem einen Modell auf die anderen Modelle übertragen werden können. Um dies zu untersuchen, wurden drei mögliche Vorgehensweisen erarbeitet, die in Abbildung 6.6 zu sehen sind und nun näher vorgestellt werden. Dabei ist zu beachten, dass im Folgenden immer nur von zwei Modellen ausgegangen wird, auf denen ein Co-Refactoring ausgeführt wird. Dies stellt jedoch keine Einschränkung der Annahme dar, dass Co-Refactorings auf beliebig vielen abhängigen Strukturen ausgeführt werden können.

Die erste Möglichkeit besteht in einer *vollständigen Transformation*. Das heißt, dass ein Refactoring auf nur einem Modell ausgeführt wird und dieses Modell entsprechend der Relation vollständig in das andere Modell transformiert wird. Dieser Ansatz kann ohne großen Aufwand umgesetzt werden, wenn auf Basis der Beziehung beider Modelle bereits eine Transformation definiert wurde. Dabei wird im Grunde nur ein Refactoring ausgeführt. Es muss also kein korrespondierendes Refactoring für das andere Modell gefunden werden. Weiterhin existieren neben der programmatischen Umsetzung viele Techniken, mit denen eine derartige Modelltransformation umgesetzt werden kann [Hub08]. Dieser Ansatz ist vor allem dann sinnvoll, wenn es sich um eine vollständige Beziehung der beiden Modelle handelt und es somit keine Elemente gibt, die nicht in das jeweils andere Modell transformiert werden. Der offensichtliche Nachteil besteht jedoch darin, dass das Ziel-Modell der Transformation komplett neu erstellt wird. Änderungen in diesem Modell, die nicht durch Refactorings abgedeckt werden, werden

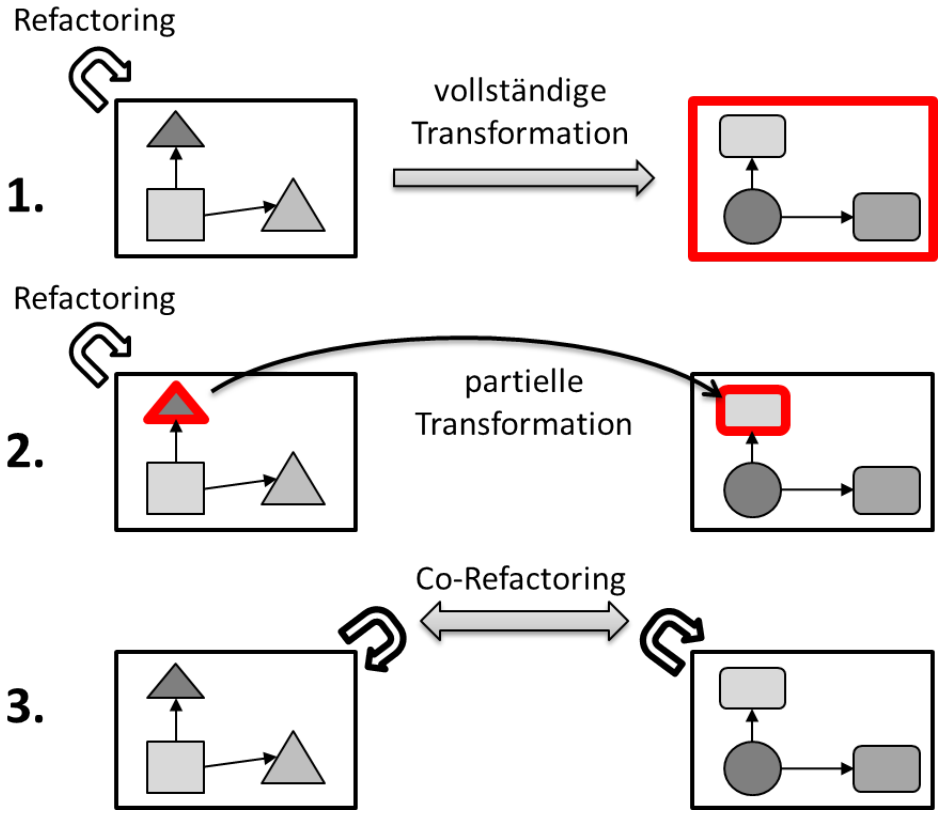

Abbildung 6.6: Ansätze des Co-Refactoring

somit überschrieben. Ein Verschmelzen des bestehenden Modells mit dem neu Transformierten unter Beibehaltung eventueller Änderungen ist zwar denkbar, jedoch sehr aufwendig.

Der zweite Ansatz geht mit einer *partiellen Transformation* auf das Problem der Erhaltung von Änderungen im abhängigen Modell ein. Hier wird nach einem Refactoring auf dem ersten Modell nicht mehr das komplette Modell transformiert. Stattdessen werden die Änderungen ermittelt, die sich durch das Refactoring ergeben. Dies kann mit Techniken wie dem EMF Change Recorder des EMF Compare-Projekts erfolgen [HKK09]. Es wird also eine Folge von Änderungen – ein so genanntes Change Set – ermittelt, die in ein korrespondierendes Change Set für das abhängige Modell umgewandelt und auf diesem ausgeführt wird. Die dadurch entstehenden Änderungen betreffen folglich nur die Bereiche beider Modelle, die vom Refactoring betroffen sind. Zusätzliche Informationen der Modelle, die nicht von der Relation abgedeckt werden, bleiben erhalten. Die Herausforderung dieses Ansatzes besteht in der Konvertierung der Change Sets. Mit den Tripel Graph Grammars (TGG) existiert ein Ansatz, Änderungen in einem Modell mit definierten Regeln auf ein anderes Modell zu übertragen [KW07]. Die TGG beschränken sich jedoch auf die synchrone Änderung von nur zwei Modellen mit festgelegten Metamodellen. Eine Verallgemeinerung auf beliebig viele Modelle und beliebige Metamodelle ist damit nicht gegeben. Des Weiteren müssen beim Einsatz der TGGs alle Änderungen im Voraus bekannt sein, da bei TGG ein fester Regelsatz von Transformationsregeln zum Einsatz kommt. Refactorings können jedoch durchaus Änderungen erfordern, die erst zur Ausführung des Refactorings angegeben werden. Dies würde eine sehr aufwendige dynamische Generierung der TGG-Regeln zur Laufzeit erfordern.

Mit dem dritten Ansatz wird versucht, die Vorteile der einfachen Ausführung, autonomer Modelle und allgemeingültiger Anwendbarkeit zu vereinen. Da er der Idee des Co-Refactorings

am nächsten kommt, wird er selbst als *Co-Refactoring* bezeichnet. Bei diesem Ansatz wird zu jedem Refactoring, das auf dem einen Modell ausgeführt wird, ein korrespondierendes Refactoring auf dem anderen Modell ausgeführt. Ersteres Modell wird dabei als *originales Modell* bezeichnet, die anderen als *abhängige Modelle*. Dabei muss durch die Definition und Zuordnung der Refactorings sichergestellt sein, dass die Beziehung beider Modelle erhalten bleibt. Abgesehen von der Zuordnung der Modelle und Refactorings müssen dazu weitere Eingabewerte für die korrespondierenden Refactorings konvertiert werden. Auf der Grundlage eines Refactoring-Werkzeugs, wie Refactory, ist es somit möglich, Refactorings für Modelle beliebiger Metamodelle zu definieren. Diese Refactorings und alle weiteren Eingabewerte müssen aufeinander abgebildet werden, um den Co-Refactoring-Ansatz umzusetzen. Damit wird erreicht, dass der Ansatz für beliebige Modelle anwendbar ist und nur die Teile der Modelle geändert werden, die durch das Refactoring betroffen sind. Im Folgenden wird der Co-Refactoring-Ansatz ausführlich behandelt.

6.4.2 Herleitung und Architekturvergleich

Wie im Kapitel 5.3 vorgestellt, gibt es mit COPE einen zum Co-Refactoring vergleichbaren Ansatz. Der Unterschied besteht darin, dass bei COPE ein Metamodell weiterentwickelt wird und anschließend die dazugehörigen Modelle migriert werden. Während beim Co-Refactoring ein Modell weiterentwickelt wird und dann alle abhängigen Modelle eine entsprechende Weiterentwicklung erfahren sollen. Der Umstand wird in Abbildung 6.7 grafisch verdeutlicht. COPE definiert Coupled Operations, die aus einer Änderung des Metamodells plus einer Migrationsstrategie für diese Änderung bestehen. Aus der Summe mehrerer Änderungen ergibt sich schließlich eine neue Version des Metamodells und die Migrationsstrategien werden zu einer Gesamt-Migrationsstrategie zusammengefasst, die Grundlage für einen generierten Modell-Migrator ist.

Das Vorgehen beim Co-Refactoring ist ähnlich, nur stellen hier die Änderungen auf beiden Seiten Refactorings dar. Weiterhin ist das Co-Refactoring eine symmetrische Transformation beider Strukturen. Das heißt, die abhängigen Modelle werden innerhalb derselben Co-Refactoring-Operation transformiert. Bei COPE hingegen, erfolgt die Migration der Modelle unabhängig von der Weiterentwicklung der Metamodelle. Als letzter Punkt ist zu nennen, dass sich ein Co-Refactoring auf beliebig viele abhängige Modelle ausweiten lässt und nicht auf zwei in Beziehung stehende Modelle beschränkt ist.

Die Herausforderung besteht nun darin, das Prinzip der Coupled Operation auf den Ansatz des Co-Refactorings zu übertragen. Ziel ist es also, eine Struktur zu finden, die korrespondierende Refactorings für in Beziehung stehende Modelle zusammenfasst. Im Folgenden beschäftigen wir uns damit, wie diese Struktur aussehen muss und wie sie hergestellt werden kann.

6.4.3 Der CoRefactorer

Bevor das Co-Refactoring ausführlich behandelt werden kann, muss geklärt sein, wie es sich in die Refactoring-Architektur einordnet. Bei Betrachtung der existierenden Evolutionsstrategien von Stojanovic sowie Palma und Haase wird deutlich, dass ein Refactoring-Prozess aus drei Schritten bestehen muss: Planung, Durchführung und Prüfung (vgl. dazu die Abbildungen 3.5 und 5.5). Entsprechend dieser drei Schnitte wird eine Refactoring-Architektur angenommen, wie sie in Abbildung 6.8 zu sehen ist. Der Pre-Refactorer ermittelt in Interaktion mit dem Benutzer alle notwendigen Eingaben für ein Refactoring und prüft dabei alle Vorbedingungen. Dies erfordert, dass die Auswirkungen des Refactorings in einer Art Probe-Durchlauf analysiert werden. Diese Eingaben werden zusammengefasst als ein *Input-Datensatz* an den Refactorer übergeben, der das Refactoring tatsächlich ausführt. An dieser Stelle kommt der CoRefactorer

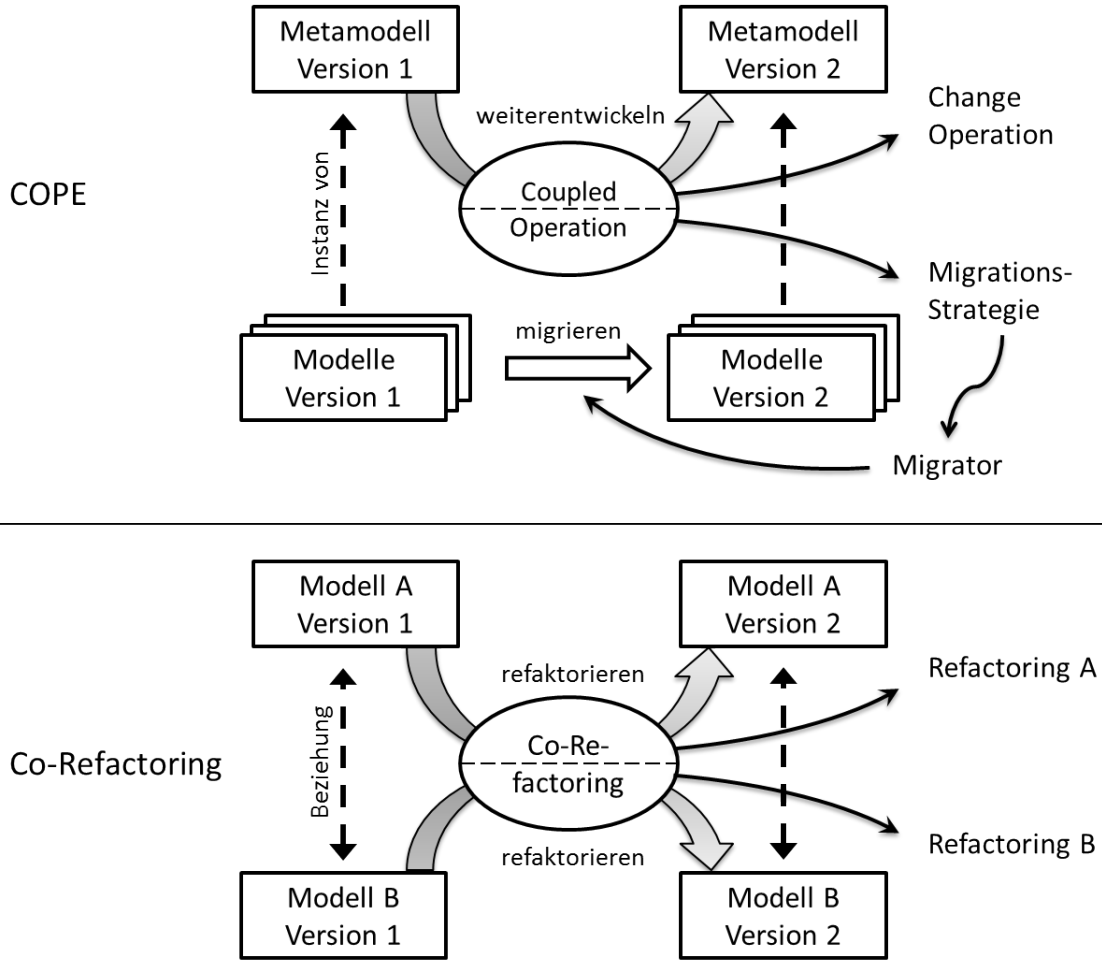

Abbildung 6.7: Vergleich zwischen COPE und Co-Refactoring

ins Spiel, der die Eingaben abfängt und daraus weitere Input-Datensätze für Refactorings auf weiteren Modellen ableitet. Auch diese Input-Datensätze werden dann an den Refactorer weitergereicht. Der Refactorer übergibt das refaktorierte Modell schließlich dem Validator, der die Nachbedingungen prüft.

Die wesentliche Aufgabe des CoRefactorers besteht also darin, für ein gegebenes Refactoring ein dazugehöriges Refactoring für alle abhängigen Modelle zu finden. Hierbei sind allerdings, neben dem Refactoring selbst, weitere Eingabewerte zu beachten, die zur Ausführung des Refactorings notwendig sind. Diese Eingaben werden in dem Input-Datensatz zusammengefasst. Die daraus resultierende Idee ist, dass der CoRefactorer aus einem gegebenen Input-Datensatz einen korrespondierenden Input-Datensatz für alle abhängigen Modelle generieren kann. Anschließend kann jeder Input-Datensatz an ein Refactoring-Werkzeug weitergereicht werden, dass in der Lage ist, die Refactorings auszuführen, z. B. Refactory. Die Abbildung 6.9 zeigt die prinzipielle Vorgehensweise im Detail. Dem CoRefactorer wird der Input-Datensatz für das erste Refactoring übergeben. Dieser entspricht in der Regel dem Input-Datensatz des ersten Refactorings. Die Input-Datensätze aller korrespondierenden Refactorings werden aus dem ersten abgeleitet.

Zur Ausführung eines Refactorings muss jeder Input-Datensatz vier Einträge enthalten: Das Modell, das Refactoring, Input-Elemente und Eingabewerte. Das Modell kann ein beliebiges Modell eines beliebigen Metamodells sein, auf dem das Refactoring ausgeführt werden soll.

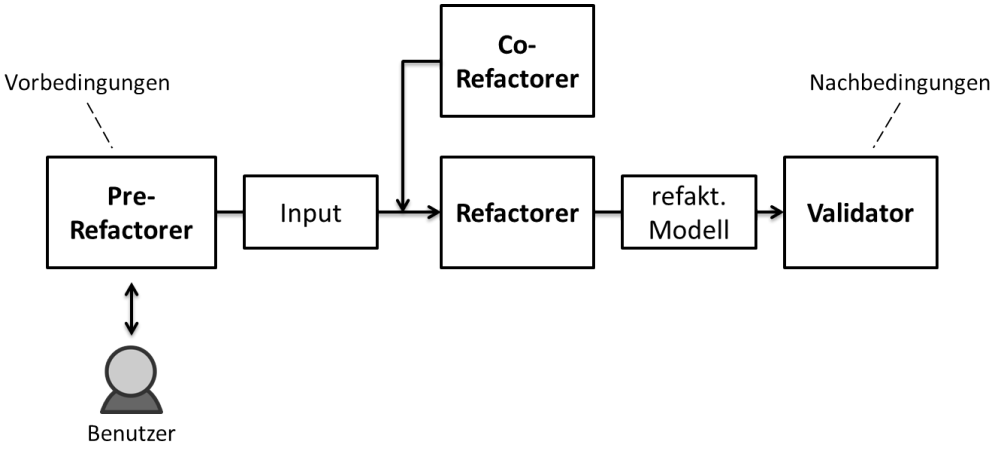

Abbildung 6.8: Einordnung des CoRefactorer in die Refactoring-Architektur

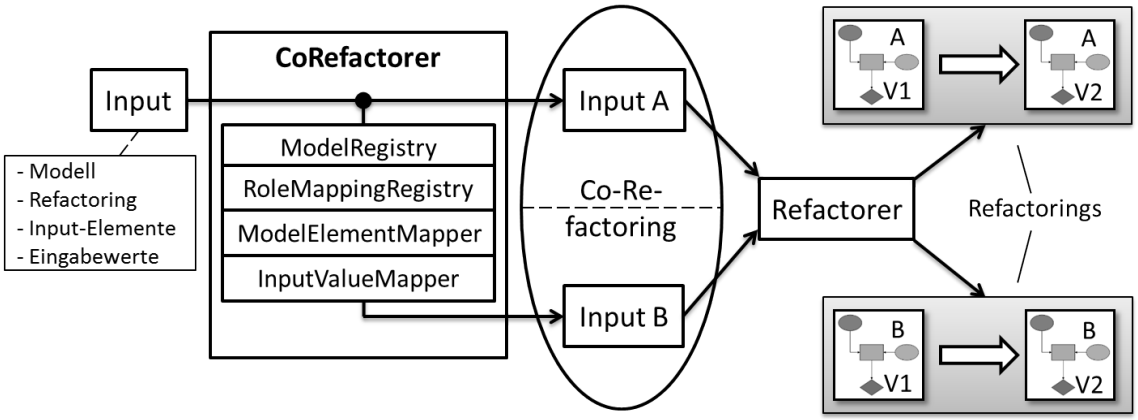

Abbildung 6.9: Co-Refactoring-Architektur

Das Refactoring ist die tatsächliche Umstrukturierung und wird in Refactory durch ein Role Mapping dargestellt. Die Input-Elemente sind die ausgewählten Elemente des Modells, die als Ausgangspunkt für das Refactoring dienen. Für viele Refactorings ist dies lediglich ein einziges Element. Im Allgemeinen können es aber auch mehrere sein. Die Eingabewerte sind zusätzlich benötigte Eingaben des Benutzers, wie der Name eines neu zu erstellenden Elements. Ein Co-Refactoring erfordert, dass für jeden dieser vier Einträge korrespondierende Daten für alle abhängigen Modelle bestimmt werden. Eine eindeutige Abbildung der Änderungen in den einzelnen Modellen ist erst gegeben, wenn die Beziehung der Metamodelle und alle vier Einträge des Input-Datensatzes berücksichtigt werden. Da ein Co-Refactoring jedoch nicht alle möglichen Änderungen eines Modells abdeckt, ist diese Abbildung nicht surjektiv und damit auch nicht bijektiv. Wie ebenfalls in Abbildung 6.9 zu sehen ist, wird eine Architektur aus vier Komponenten vorgeschlagen, um die Einträge eines Input-Datensatzes auf korrespondierende Werte abzubilden. Diese Komponenten werden im Folgenden näher erläutert.

Die **ModelRegistry** stellt die Verbindung zwischen allen abhängigen Modellen her. Basierend auf einem beliebigen Modell kann die ModelRegistry Auskunft darüber geben, welche Modelle zum Gegebenen in Beziehung stehen und somit co-refaktoriert werden sollen. An der ModelRegistry müssen vorher die in Beziehung stehenden Modelle registriert werden. Dies kann entweder durch den Benutzer oder automatisch bei der Transformation des einen in das andere Modell erfolgen.

Zusammengehörige konkrete Refactorings werden mit der **RoleMappingRegistry** aufeinander abgebildet. Dabei ist zu beachten, dass unterschiedliche Metamodelle durchaus strukturell unterschiedliche Refactorings erfordern, um die Beziehung zwischen zwei Modellen zu erhalten. Daraus folgt, dass die konkreten Refactorings, dargestellt durch die Role Mappings, verwendet werden müssen. Es würde nicht ausreichen, die generischen Role Models von Refactory einzusetzen. Zur Bestimmung eines korrespondierenden Role Mappings sind drei Eingaben nötig: (1) das Role Mapping, zu dem ein passendes gefunden werden soll (2) abhängig von der Relation zwischen den Metamodellen, die Input-Elemente, auf denen das originale Refactoring ausgeführt werden soll (3) und die Richtung des Co-Refactorings. Die Richtung enthält dabei das Metamodell des originalen Modells und das Metamodell des abhängigen Modells und gibt somit die Richtung des Co-Refactorings auf Metamodell-Ebene an. Im Rahmen dieser Arbeit wird beispielsweise ein Co-Refactoring mit der Richtung vom OWL-Metamodell zum Ecore-Metamodell umgesetzt. Für jede Richtung, in die ein Co-Refactoring erfolgen soll, ist ein Mapper zu registrieren, der auf Basis des Role Mappings und der Input-Elemente für das Quell-Metamodell entscheidet, welches Role Mapping für das Ziel-Metamodell in Frage kommt.

Jedes Refactoring hat als Ausgangspunkt ein oder mehrere Input-Elemente. Wird ein Co-Refactoring ausgeführt, müssen die Input-Elemente für alle abhängigen Modelle ebenfalls bekannt sein. Der **ModelElementMapper** übernimmt die Aufgabe, diese zu bestimmen. Es bestehen mehrere Möglichkeiten, die Input-Elemente aufeinander abzubilden, je nachdem wie das originale Modell mit den abhängigen Modellen in Beziehung steht. Werden die abhängigen Modelle über eine Modelltransformation aus dem originalen Modell erstellt, kann während der Transformation, mit Techniken wie QVT Traces [OMG08], ein Mapping-Modell erstellt werden, welches eine direkte Beziehung von den Elementen des originalen Modells zu den Elementen der abhängigen Modelle enthält. Der ModelElementMapper müsste in diesem Fall lediglich der Beziehung aus dem Mapping-Modell folgen, um die korrespondierenden Input-Elemente zu finden. Bei diesem Vorgehen besteht jedoch das Problem, dass das Mapping-Modell bei jeder Veränderung der Modelle, und somit auch bei Refactorings und Co-Refactorings, aktualisiert werden muss. Die Alternative besteht in einem Suchalgorithmus, der auf Basis der Relation zwischen den Metamodellen und Eigenschaften des Input-Elements aus dem originalen Modell die richtigen Input-Elemente in den abhängigen Modellen bestimmt. Der ModelElementMapper benötigt dazu, neben den Input-Elementen des originalen Modells, das Ziel-Modell und die Richtung des Co-Refactorings. Wie bei der RoleMappingRegistry muss zuvor für jede benötigte Richtung ein Mapper registriert werden, der hier allerdings die Input-Elemente in Abhängigkeit vom Quell- und Ziel-Metamodell bestimmt.

Das vierte Modul des CoRefactorer ist der **InputValueMapper**. Dieser dient dazu, die Eingabewerte der Refactorings für die abhängigen Modelle zu transformieren. Besteht der Eingabewert lediglich aus dem Namen eines neu zu erzeugenden Elements, bleibt dieser in der Regel unverändert. Jedoch besteht die Möglichkeit, dass solche Eingabewerte aufgrund von Namenskonventionen für andere Modelle, zum Beispiel mit Skriptsprachen, umgeformt werden müssen. Eine Sonderstellung nimmt hier die Auswahl von vorhandenen Elementen des Modells ein. Bestehen die Eingabewerte, abgesehen vom Input-Element, aus weiteren ausgewählten Elementen des Modells, müssen diese auf entsprechende Elemente des abhängigen Modells abgebildet werden. Dazu kann der ModelElementMapper zum Einsatz kommen.

6.5 Zusammenfassung

In diesem Kapitel wurde ein Konzept zur Entwicklung und Evolution ontologiegetriebener Softwaresysteme erarbeitet. Der Ansatz wird mit OntoMore bezeichnet und basiert auf einer Ontologie zur Modellierung der Anwendungsdomäne. Diese Ontologie wird in Modelle der MDSD

transformiert, aus denen schließlich durch Codegenerierung eine ausführbare Anwendung abgeleitet werden kann. Der Ansatz des Co-Refactorings stellt die synchrone Weiterentwicklung von Ontologie und Modellen sicher, sodass insgesamt die drei Anforderungen an den Ansatz – Design, Codegenerierung und Evolution – erfüllt sind.

Die OWL-Ecore-Transformation definiert eine Metaebenen-übergreifende Transformation von Ontologien des OWLizer-Metamodells in Ecore-Metamodelle und dazugehörige Modelle. Die Transformation ist nur für bestimmte Teile der Ontologie möglich. Diese Teile werden mit Einschränkungen beschrieben, die in ihrer Gesamtheit transformierbare Ontologien definieren.

Zur Umsetzung der Evolution wurde der Begriff Ontologie-Refactoring definiert. Darauf aufbauend wurde ein Katalog von Refactorings für Ontologien und Ecore-Modelle zusammengestellt. Jedes Ontologie-Refactoring besteht aus einem Namen, der Motivation, Vorbedingungen, einer Lösung, Nachbedingungen, der Betrachtung der Erhaltung der Individuen und einem Beispiel. Ein Beispiel demonstriert die ausführliche Beschreibung von Ontologie-Refactorings nach diesem Schema.

Schließlich wird der Ansatz des Co-Refactorings vorgestellt, der im Allgemeinen die Ausführung von Refactorings auf voneinander abhängigen Strukturen beschreibt. Wobei die Beziehung zwischen diesen Strukturen erhalten bleibt. Das Co-Refactoring wurde aus dem COPE-Ansatz abgeleitet und stützt sich auf die Idee, sämtliche Eingabewerte eines Refactoring in einem Input-Datensatz zu kapseln, der für alle abhängigen Modelle entsprechend transformiert wird.

7 Praktische Umsetzung

7.1 Architektur und eingesetzte Techniken

Die Umsetzung des OntoMore-Ansatzes enthält drei Teile: Den OWL-Ecore-Transformator, Refactorings für OWL und Ecore sowie den CoRefactorer. Wie in Abschnitt 6.1 gezeigt, kann die Generierung einer Anwendung aus einer Ontologie auf unterschiedliche Art und Weise erfolgen. In dieser Arbeit kommen die Techniken des EMF zum Einsatz, da diese flexible Möglichkeiten zur Codegenerierung bereithalten. Dementsprechend setzt der OWL-Ecore-Transformator, wie in Abschnitt 6.2 definiert, eine Abbildung von einer Ontologie auf ein Metamodell und ein Modell von Ecore in einem Eclipse-Plugin um. Darüber hinaus existiert mit Refactory bereits ein Ansatz für generisches Modell-Refactoring auf EMF-Basis, das als Grundlage für das Co-Refactoring dienen kann. Die Refactorings für OWL und Ecore wurden mit den DSLs von Refactory umgesetzt und können unabhängig voneinander auf den Modellen von OWL bzw. Ecore ausgeführt werden. Der CoRefactorer erweitert Refactory dahin gehend, dass er die Ausführung von Co-Refactoring ermöglicht. Er wurde ebenfalls in Form von Eclipse-Plugins realisiert.

Die Schichten-Architektur der eingesetzten Techniken ist in Abbildung 7.1 zu sehen. Dabei sind die Techniken einer Schicht von der jeweils darunter liegenden Technik abhängig. Die Grundlage aller Techniken ist das EMF. EMFText, auf der nächsten Stufe, stellt die Möglichkeit zur Verfügung, textuelle DSLs zu entwerfen. Diese Funktion wird von OntoMoPP genutzt, um basierend auf einem OWL-Metamodell eine textuelle Syntax für Ontologien zu definieren, die der Manchester OWL Syntax entspricht. OWLText ist der Editor, der die Bearbeitung von Ontologien in dieser Syntax erlaubt. Das OWL-Metamodell wird außerdem vom OWLizer für die Transformation beliebiger Metamodelle und Modelle in Ontologien genutzt.

Refactory nutzt DSLs zur Definition von Modell-Refactorings. Diese DSLs wurden mit EMFText spezifiziert. Die Refactorings lassen sich aber auf beliebigen EMF-Modellen ausführen, unabhängig von der Notation der Modelle. So können beispielsweise auch aus OWLText Refactorings aufgerufen werden. Mit diesen kann die dargestellte Ontologie umstrukturiert werden, da sie von einem Modell konform zum OWL-Metamodell repräsentiert wird.

Der OWL-Ecore-Transformator, definiert auf Basis des OWL-Metamodells eine Transformation von Ontologien dieses Metamodells in Ecore-Modelle. Die Abbildung orientiert sich dabei an der des OWLizer, um eine bidirektionale Transformation zwischen OWL und Ecore zu ermöglichen. Die Abhängigkeit des OWL-Ecore-Transformators vom OWLizer ist rein konzeptionell, in dem Sinne, dass er die inverse Transformation des OWLizer umsetzt. Was die Implementierung betrifft, greift der OWL-Ecore-Transformator aber nur auf das OWL-Metamodell zu.

Der CoRefactorer vervielfältigt Input-Datensätze für Refactorings und reicht diese an Refactory weiter, um die Refactorings tatsächlich auszuführen. Der CoRefactorer hat noch keine grafische Benutzeroberfläche. Die Anbindung an eine solche ist jedoch problemlos möglich.

Die angedachte Arbeitsweise von OntoMore sieht vor, dass Ontologien mit OWLText erstellt und bearbeitet werden. Diese Ontologien können aus Eclipse heraus mit dem OWL-Ecore-Transformator über ein Kontextmenü in Metamodell und Modelle konvertiert werden. Aus OWLText lässt sich außerdem der Pre-Refactorer aufrufen, der alle nötigen Eingabewerte für ein Refactoring ermittelt und an den CoRefactorer weiterreicht. Dieser leitet daraus das

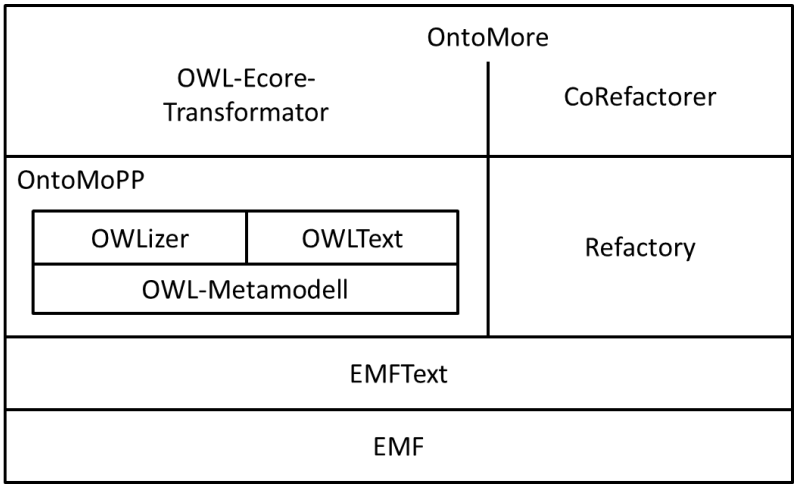

Abbildung 7.1: Schichten-Architektur der von OntoMore eingesetzten Techniken

CoRefactoring, bestehend aus mehreren Input-Datensätzen, ab und reicht sie zur Ausführung an Refactory weiter. Refactory nimmt die eigentlichen Änderungen an der Ontologie und dem zugehörigen Metamodell vor. Schließlich wird die geänderte Ontologie in OWLText sichtbar. Die Bearbeitung der Ontologien durch den Benutzer erfolgt also nur über OWLText, welches sich in die bestehende Infrastruktur von EMF und EMFText integriert.

7.2 Testgetriebene Entwicklung

Die Implementierung wurde nach dem Ansatz der testgetriebenen Entwicklung (TDD) durchgeführt, weil damit eine flexible Softwareentwicklung möglich ist [Bec03]. Wie eben gezeigt, basieren die Plugins von OntoMore auf Refactory und OntoMoPP, die sich ihrerseits noch in der Entwicklung befinden. Deshalb ist es erforderlich, dass OntoMore ohne großen Aufwand an Änderungen in diesen beiden Anwendungen angepasst werden kann. TDD setzt die Prinzipien der agilen Softwareentwicklung um und erfüllt somit diese Anforderung. Der Umstand, dass eine relativ kleine Anwendung ohne großen organisatorischen Aufwand zu entwickeln ist, unterstützt den Einsatz von TDD.

In der TDD werden Tests zum Design des Softwaresystems eingesetzt. Diese Design-Tests unterscheiden sich wesentlich von Tests, die sonst dazu eingesetzt werden, nach der Fertigstellung des Systems dessen Fehlerfreiheit zu prüfen. Stattdessen werden in der TDD Tests vor der eigentlichen Entwicklung geschrieben, um Design-Entscheidungen zu dokumentieren und gleichzeitig automatisch prüfen zu können. Erst nachdem eine Design-Entscheidung durch einen Test eindeutig definiert wurde, wird der Produktionscode geschrieben, der diesen Test und somit die Design-Entscheidung erfüllt. Dieses Vorgehen findet in sehr kleinen Schritten statt, damit auch die kleinsten Details im Design des Gesamtsystems durch Tests abgedeckt werden. Die Tests sind also sehr fein-granular und werden mit einfachen Unit-Tests umgesetzt, die sich in ihrer Gesamtheit zu kompletten Regressions-Testsuiten kombinieren lassen. TDD ersetzt jedoch nicht etablierte Test-Techniken, wie Integrations-, Performance-, Akzeptanz- oder Systemtests. Das Vorgehen der TDD zielt darauf ab, zu jedem Zeitpunkt ein sauberes und getestetes Design des Systems vorliegen zu haben, welches sich somit schnell und einfach an geänderte Anforderungen anpassen lässt. TDD verfolgt jedoch nicht das Ziel, eine möglichst große Testabdeckung zu erreichen. Es werden nur so viele Testfälle geschrieben, wie nötig sind, um das Design des Systems daraus abzuleiten.

Die fünf sich ständig wiederholenden Arbeitsschritte der TDD sollen an einem sehr einfachen Beispiel aus einem Test-Modul des CoRefactorers demonstriert werden [Bec03, S. 24]. Während der Tests für den CoRefactorer ist es notwendig Dateien mit einem bestimmten Namensschema einzulesen. Eine Eingabe-Datei hat einen Dateinamen mit der Endung „_IN“ während eine Datei mit den Daten, die als Ergebnisse erwartet werden, einen Dateinamen mit der Endung „_EXP“ hat. Die Klasse `TestFileName` soll einen gegebenen Dateinamen um die jeweils richtige Endung erweitern. (1) Dazu wird zuerst ein Test geschrieben, der den ersten Fall prüft (Listing 7.1).

(2) Bevor dieser Test kompiliert werden kann, muss die Klasse `TestFileName` und die Methode `getInputFileName` implementiert werden (Listing 7.2).

Listing 7.1: Test für Input-Dateinamen

```
1 @Test
2 public void testGetInputFileName() {
3     TestFileName testFileName = new TestFileName("dir/file.txt");
4     assertEquals("dir/file_IN.txt", testFileName.getInputFileName());
5 }
```

Listing 7.2: Erste Implementierung der Klasse TestFileName

```
1  public class TestFileName {
2
3      private URI fileName;
4
5      public TestFileName(String fileName) {
6          this.fileName = URI.createURI(fileName);;
7      }
8
9      public String getInputFileName() {
10         return null;
11     }
12 }
```

(3) Der Test kann nun ausgeführt werden, schlägt aber fehl, da die Methode im Moment nur null zurückgibt. Diesen fehlerhaften Test zu sehen ist wichtig, da damit gezeigt wird, dass der Test eine Funktion abdeckt, die bisher noch nicht Bestandteil des Systems ist. (4) Die einfachste Möglichkeit den Test erfolgreich auszuführen, ist, einfach den String `"dir/file_IN.txt"` zurückzugeben. Wir gehen an dieser Stelle schon einen Schritt weiter und implementieren die Methode für beliebige Dateinamen (Listing 7.3).

Wird der Test aus Listing 7.1 erneut ausgeführt, ist er erfolgreich. Bei der Erweiterung des Programms um einen neuen Test und die entsprechende Implementierung für die Endung „_EXP“, fällt auf, dass die Implementierungen von `getInputFileName` und `getExpectedFileName` zu großen Teilen redundant sind (Listings 7.4 und 7.5).

Listing 7.3: Implementierung der Methode getInputFileName

```
1 private String getInputFileName() {
2     String fileNameWithoutExtension = fileName.trimFileExtension().toString();
3     String extension = fileName.fileExtension();
4     URI uriWithAppendix = URI.createURI(fileNameWithoutExtension + "_IN");
5     uriWithAppendix = uriWithAppendix.appendFileExtension(extension);
6     return uriWithAppendix.toString();
7 }
```

Listing 7.4: Zweiter Test: Expected-Dateinamen

```
1 @Test
2 public void testGetExpectedFileName() {
3     TestFileName testFileName = new TestFileName("dir/file.txt");
4     assertEquals("dir/file_EXP.txt", testFileName.getExpectedFileName());
5 }
```

Listing 7.5: Implementierung der Methode getExpectedFileName

```
private String getExpectedFileName() {
    String fileNameWithoutExtension = fileName.trimFileExtension().toString();
    String extension = fileName.fileExtension();
    URI uriWithAppendix = URI.createURI(fileNameWithoutExtension + "_EXP");
    uriWithAppendix = uriWithAppendix.appendFileExtension(extension);
    return uriWithAppendix.toString();
}
```

Listing 7.6: Finale Implementierung der Klasse TestFileName

```
public class TestFileName {

    private URI fileName;

    public TestFileName(String fileName) {
        this.fileName = URI.createURI(fileName);;
    }

    public String getInputFileName() {
        return appendToFileName("_IN");
    }

    public String getExpectedFileName() {
        return appendToFileName("_EXP");
    }

    private String appendToFileName(String appendix) {
        String fileNameWithoutExtension = fileName.trimFileExtension().toString();
        String extension = fileName.fileExtension();
        URI uriWithAppendix = URI.createURI(fileNameWithoutExtension + appendix);
        uriWithAppendix = uriWithAppendix.appendFileExtension(extension);
        return uriWithAppendix.toString();
    }
}
```

(5) Diese Redundanz kann mit einem Refactoring, wie *Extract Method*, entfernt werden, sodass schließlich das finale Design aus Listing 7.6 zustande kommt. Indem die Tests erneut ausgeführt werden, kann schnell und einfach geprüft werden, ob das Refactoring korrekt durchgeführt wurde.

Ein wesentliches Grundprinzip der TDD ist, dass die Tests mit so wenig Code und Aufwand wie möglich zur erfolgreichen Ausführung gebracht werden. Es wird sogar zwischenzeitlich eine unsaubere Implementierung toleriert, wenn sie die Tests erfüllt. Sobald die Tests erfolgreich sind, werden solche unsauberen Implementierungen und Redundanzen im Code durch Refactorings beseitigt. Refactorings treiben also wesentlich das Design des Gesamtsystems.

Ein wichtiger Punkt der gezeigten Vorgehensweise ist, dass neue Anforderungen nur dann im System berücksichtigt werden, wenn sie tatsächlich auftreten. Somit ist sichergestellt, dass das Design des Systems optimal für die aktuell abgedeckten Anforderungen ist. Dies ist wiederum eine Voraussetzung dafür, dass das System möglichst einfach um neue Funktionen erweitert werden kann.

Das vorgestellte Beispiel ist sehr einfach und vermag sicher nicht die TDD in vollem Umfang darzustellen. Die einzelnen Schritte des Beispiels mögen zu simpel und offensichtlich für einen erfahrenen Programmierer wirken. Jedoch ist gerade ein derart detailliertes Vorgehen in solchen kleinen Schritten entscheidend, damit für jedes Detail des Designs eine bewusste Entscheidung getroffen wird.

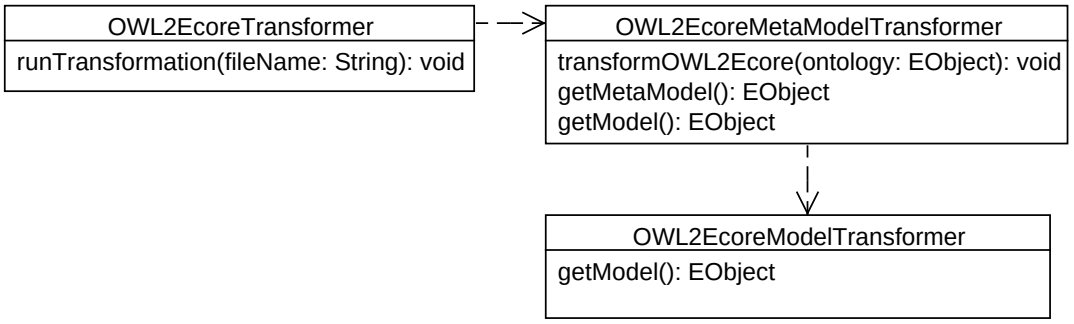

Abbildung 7.2: Basis-Konzepte des OWL-Ecore-Transformators

7.3 OWL-Ecore-Transformator

Zur Umsetzung der OWL-Ecore-Transformation aus Kapitel 6.2 wurde der OWL-Ecore-Transformator als Eclipse-Plugin entwickelt. Dieses Plugin soll es ermöglichen, über einen Kontextmenü-Eintrag eine ausgewählte Ontologie über die Eclipse-Oberfläche in ein Metamodell und ein dazugehöriges Modell zu konvertieren.

Wie im Abschnitt 6.2 diskutiert, können vorhandene Ansätze der Modelltransformation, wie beispielsweise QVT oder ATL[1], dafür nicht zur Anwendung kommen, da sie nicht für eine Transformation über mehrere Metaebene ausgelegt sind. Die OWL-Ecore-Transformation erfordert jedoch genau diese Metaebenen-übergreifende Transformation. Deshalb wird der OWL-Ecore-Transformator nach dem Vorbild des OWLizer programmatisch mit Java umgesetzt. Das Modell wird dabei auf Basis des erzeugten Ecore-Metamodells, mit Hilfe der Reflective-API von EMF, dynamisch generiert.

Der OWL-Ecore-Transformer wird im Plugin `org.ontomore.transformation` implementiert. Abbildung 7.2 zeigt die wichtigsten Klassen und Methoden. Die Klasse `OWL2EcoreTransformer` wird von der grafischen Benutzeroberfläche mit dem Dateinamen der Ontologie aufgerufen und kümmert sich um das Dateimanagement. Dazu gehört das Laden der Ontologie von der Datei, das Speichern des Ergebnisses der Transformation in einer neuen Datei und das Aktualisieren des Workspaces. Die eigentliche Transformation wird in der Klasse `OWL2EcoreMetaModelTransformer` durchgeführt. Wobei diese lediglich die Transformation der TBox in das Metamodell entsprechend der Tabelle 6.2 übernimmt. Die Transformation der ABox in das Modell wird in der Klasse `OWL2EcoreModelTransformer` durchgeführt. Sie setzt demnach die Spezifikation der Tabelle 6.1 um.

Die aktuelle Implementierung greift direkt auf die Konzepte des OWL- und des Ecore-Metamodells zu. Um die Implementierung zu verallgemeinern und robuster gegenüber Änderungen dieser Metamodelle zu machen, wären Wrapper-Layer denkbar, die einen indirekten Zugriff auf die Metamodelle erlauben. Die prinzipielle Architektur dieses Ansatzes ist in Abbildung 7.3 zu sehen. Dieser Ansatz würde es außerdem erlauben, die Metamodelle auszutauschen. Statt dem OWL-Metamodell des OWLizer könnten somit auch andere Ontologie-Metamodell zum Einsatz kommen und die Transformation wäre nicht mehr auf Ecore beschränkt, sondern könnte auch andere Formate zur Darstellung des Domänenmodells berücksichtigen. Das Prinzip der Wrapper-Layer wurde für die Elemente *Ontology* und *EPackage* beispielhaft umgesetzt (in den Klassen `OntologyWrapper` bzw. `PackageWrapper`).

[1] `http://www.eclipse.org/atl/`

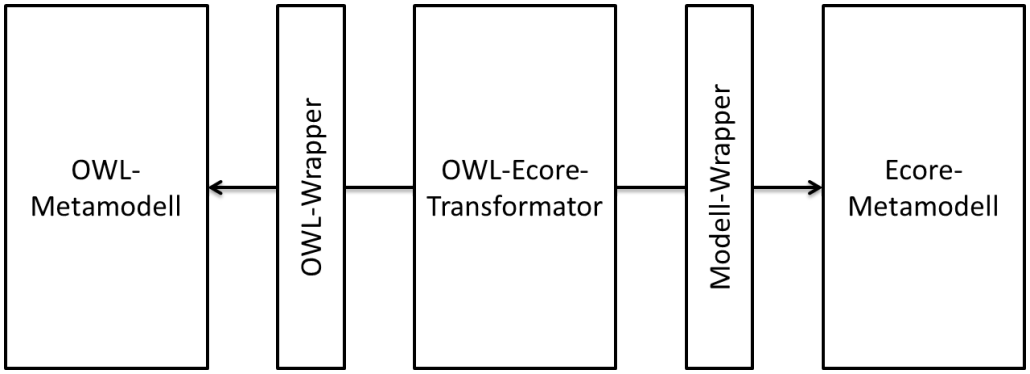

Abbildung 7.3: Wrapper-Layer für den OWL-Ecore-Transformator

Eine umfangreiche Testsuite testet die Implementierung der OWL-Ecore-Transformation mit zahlreichen Testfällen. Dazu wurde mit dem `ModelComparator` ein Modul geschrieben, das unter Zugriff auf EMF Compare automatisch die transformierten mit den erwarteten Modellen vergleicht.

Der OWL-Ecore-Transformator ermöglicht also eine zum OWLizer inverse Transformation, bezogen auf transformierbare Ontologien, wie sie im Abschnitt 6.1 definiert wurden.

7.4 Refactoring mit Refactory

Die Umsetzung der in Abschnitt 6.3 definierten Refactorings erfolgte mit Refactory [Rei10]. Wie bereits in Abbildung 4.2 gezeigt, gibt es bei Refactory drei DSLs, mit denen ein Refactoring definiert wird. Das generische Refactoring besteht aus dem *Role Model*, das die Struktur der zu verändernden Elemente mit Hilfe von Rollen beschreibt und der *Refactoring Specification*, welche die Umstrukturierungen basierend auf dem Role Model definiert. Dieses generische Refactoring wird mit dem *Role Mapping* auf ein konkretes Metamodell abgebildet, womit das Refactoring tatsächlich ausführbar wird.

Der genaue Aufbau dieser drei Artefakte soll an dem ersten Beispiel aus Abschnitt 6.3 demonstriert werden. Das Refactoring **Convert DataProperty to ObjectProperty** erfordert im Allgemeinen, dass ein Element in einem Container gegen ein anderes ausgetauscht wird, wobei weitere abhängige Elemente zu beachten sind. Das dafür erstellte Role Model heißt *Replace Feature* und ist in Abbildung 7.4 zu sehen. Der Container enthält ein `OrigFeature`, das ersetzt werden soll und ein `NewFeature`, das neu hinzugefügt wird. Beide Features besitzen ein Attribut, über das der Name vom `OrigFeature` an das `NewFeature` weitergegeben werden kann. Optional enthält das `OrigFeature` eine Referenz auf eine `Domain`, die an das `NewFeature` weitergegeben werden muss. Die `Range` kommt hingegen neu hinzu und muss dementsprechend mit einem neuen Namen versehen werden. Als neue Klasse wird die `Range` dem `Container` hinzugefügt und wird vom `NewFeature` referenziert.

Die Refactoring Specification zu diesem Role Model beschreibt den Punkt Lösung im Ontologie-Refactoring-Schema aus Abschnitt 6.3, sie ist in Abbildung 7.5 zu sehen. Zuerst werden ausgehend vom Input-Element alle Elemente bestimmt, die zur Ausführung den Rollen entsprechen: Der `Container`, das `OrigFeature` und die `Domain`. Anschließend wird das `NewFeature` nach dem `OrigFeature` im `Container` eingefügt und mit dem Namen des `OrigFeature` versehen. Danach wird die `Range` neu erstellt und erhält einen neuen Namen. Als nächstes werden die Referenzen von `NewFeature` zu `Domain` und `Range` gesetzt. Zum Schluss wird das `OrigFeature` entfernt.

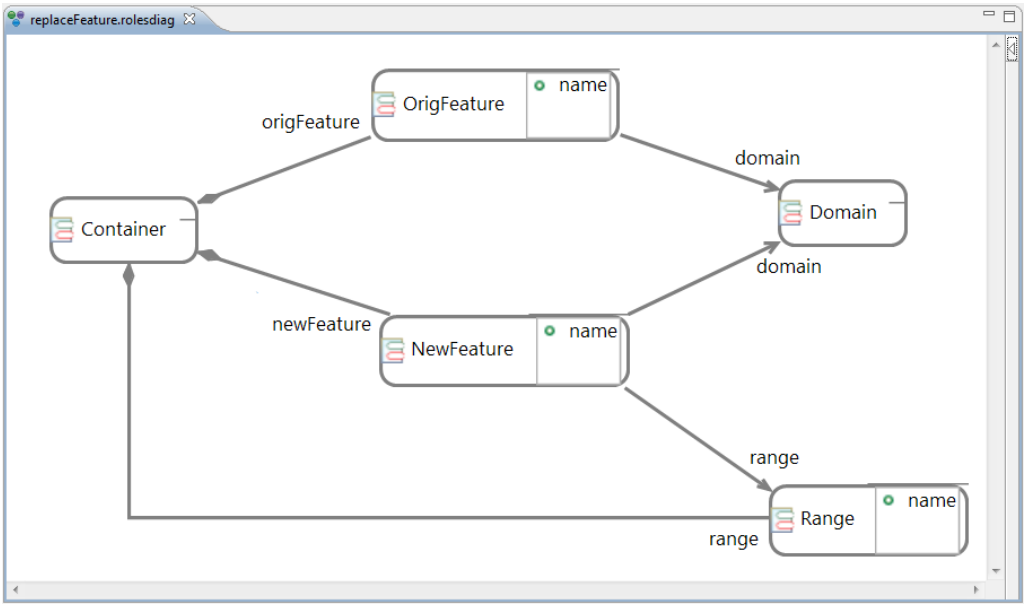

Abbildung 7.4: Role Model von Replace Feature

replaceFeature.refspec

```
1  REFACTORING FOR <ReplaceFeature>
2
3  STEPS {
4      object container := Container as trace(INPUT);
5      object origFeature := OrigFeature from filter(INPUT);
6      object domain := Domain from path(INPUT);
7
8      index afterOrigFeature := after(origFeature);
9      create new newFeature:NewFeature in container at afterOrigFeature;
10     assign origFeature.name for newFeature.name;
11
12     index afterNewFeature := after(newFeature);
13     create new range:Range in container at afterNewFeature;
14     assign range.name;
15
16     set use of domain in newFeature;
17     set use of range in newFeature;
18
19     remove origFeature;
20 }
```

Abbildung 7.5: Refactoring Specification von Replace Feature

convertDataProperty.rolemapping

```
1  ROLEMODELMAPPING FOR <http://org.emftext/owl.ecore>
2
3  "Convert Data Property To Object Property" maps <ReplaceFeature> {
4
5       Container := Ontology {
6          origFeature := frames:DataProperty;
7          newFeature := frames:ObjectProperty;
8          range := frames:Class;
9       };
10
11      OrigFeature := DataProperty (name -> iri) {
12         domain := domain:Disjunction -> conjunctions:Conjunction
13         -> primaries:ClassAtomic -> clazz;
14      };
15
16      NewFeature := ObjectProperty (name -> iri) {
17         domain := domain:Disjunction -> conjunctions:Conjunction
18         -> primaries:ClassAtomic -> clazz;
19         range := propertyRange:Disjunction -> conjunctions:Conjunction
20         -> primaries:ClassAtomic -> clazz;
21      };
22
23      Domain := Class;
24
25      Range := Class (name -> iri);
26 }
27
```

Abbildung 7.6: Role Mapping von Replace Feature auf das OWL-Metamodell

Diese Refactoring Specification bildet nur die grundlegenden Operationen der Konzeption dieses Refactorings von Seite 66 ab. Zur Vereinfachung wird hier angenommen, dass die `Range`-Klasse immer neu erstellt wird, anstatt eine vorhandene auszuwählen. Außerdem können die EquivalentProperties- und DisjointProperties-Axiome nicht geprüft und angepasst werden, da diese Axiome vom OWL-Metamodell nicht unterstützt werden. Eine Auswertung der EquivalentTo-Elemente mit Refactory ist ebenfalls nicht möglich, da dazu Schleifen-Konstrukte nötig wären, die in der Refactoring Specification nicht verfügbar sind.

Die Einschränkung, dass es in der Refactoring Specification keine Konstrukte für Wiederholung, Verzweigung und Wertzuweisung gibt, führt auch dazu, dass die Datenmigration aus dem Punkt Erhaltung der Individuen aus dem Refactoring-Schema oftmals nicht umgesetzt werden kann. Im gezeigten Beispiel müsste zum Beispiel durch alle DataProperty-Assertions iteriert werden. Die Assertions, welche die ersetzte DataProperty verwenden, müssen durch entsprechende ObjectProperty-Assertions ersetzt werden. Dabei muss für jede dieser neuen ObjectProperty-Assertions ein neues Individuum erstellt werden, dem abhängig von der gewählten Migrationsstrategie neue Daten hinzugefügt werden. Für derart komplexe Änderungen ist nur eine GPL sinnvoll zu gebrauchen.

Das erstellte generische Refactoring wird nun mit dem Role Mapping aus Abbildung 7.6 auf das OWL-Metamodell abgebildet. Damit wird das konkrete Refactoring *Convert DataProperty to ObjectProperty* definiert. Da die Propertys First Class Concepts sind, spielt die `Ontology` die Rolle des `Containers`. `OrigFeature` und `NewFeature` werden auf `DataProperty` bzw. `ObjectProperty` abgebildet und `Domain` sowie `Range` spielen jeweils die Rolle einer `Class`. Die Referenzen aus dem Role Model werden auf entsprechende Referenzen des OWL-Metamodells abgebildet. Dabei kommen oft Pfade über mehrere Elemente des Metamodells zum Einsatz. Die Abbildung 7.7 zeigt den für dieses Refactoring relevanten Ausschnitt des OWL-Metamodells.

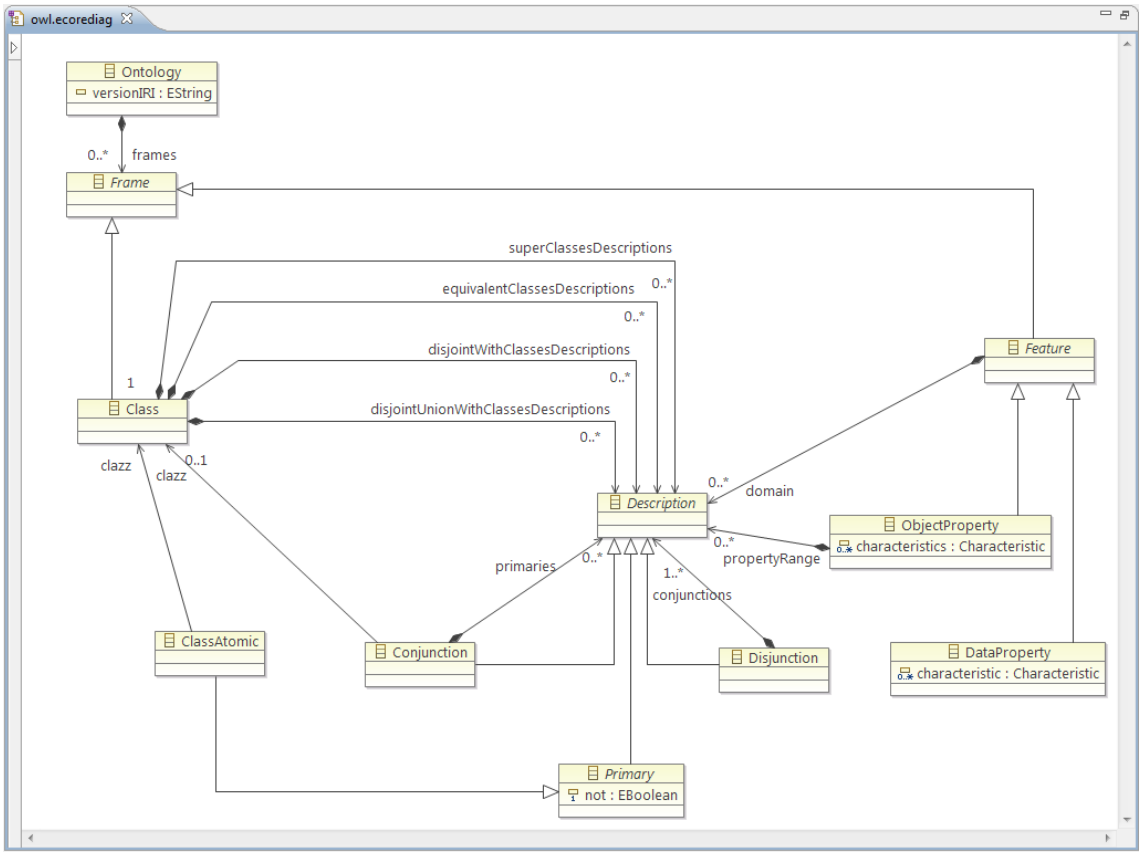

Abbildung 7.7: Ausschnitt des OWL-Metamodells für Convert DataProperty to ObjectProperty

Tabelle 7.1: Implementierte Refactorings für OWL und Ecore

Nr.	OWL-Refactoring	Ecore-Refactoring
1	Rename Element / Rename Ontology (Rename X)	Rename EElement (Rename X)
2	Duplicate Class (Duplicate With Reference)	Duplicate Class (Duplicate With Reference)
3	Convert DataProperty to ObjectProperty (Replace Feature)	Replace Data Value With Object (Replace Feature in Container)
4	Pull Up Property (Rereference X)	Pull Up Feature (MoveX)
5	Extract Superclass (Extract Loosely X)	Extract Superclass (Extract X)
6	Introduce Inverse Property (Introduce Inverse Reference in Container)	Introduce Inverse Reference (Introduce Inverse Reference)

Nach der gezeigten Vorgehensweise wurden, neben einfachen Rename-Refactorings, fünf weitere OWL-Refactorings und die dazugehörigen Ecore-Refactorings aus dem Refactoring-Katalog (Tabelle 6.3) umgesetzt. Die implementierten Refactorings sind in Tabelle 7.1 zusammengefasst. Die generischen Refactorings sind jeweils in Klammern angegeben.

Die Erhaltung der Individuen wurde in diesen Refactorings nicht berücksichtigt. Dies könnte mit einem so genannten Post-Prozessor umgesetzt werden. Der Post-Prozessor ist eine Java-Klasse, die nach der Ausführung eines Refactorings aufgerufen wird, um zusätzliche Umstrukturierungen vorzunehmen, die mit der Refactoring Specification nicht abgedeckt werden können [Rei10, S. 96]. Demnach beschränken sich die implementierten Refactorings auf die TBox der Ontologie. Damit ist eine direkte Abbildung der Änderungen auf die Ecore-Modelle möglich.

Wenn die generierte Anwendung nicht auf Instanzen dieser Modelle, sondern direkt auf die Ontologie zugreift, würde sich eine Abbildung von Refactorings ergeben, die sich nicht vollständig entsprechen. Die OWL-Refactorings führen dann Umstrukturierungen in der TBox und der ABox durch, während die Ecore-Refactorings lediglich die Änderungen an der TBox auf die Ecore-Modelle übertragen. Würden hingegen Modell-Instanzen eingesetzt, müssten diese bei jedem Refactoring entsprechend angepasst werden. Dafür gibt es allerdings mit den hier eingesetzten Techniken noch keine zufriedenstellende Lösung. Da Refactory auf die Umstrukturierung von Modellen nur einer Metaebene beschränkt ist, müssen die Modell-Instanzen manuell angepasst werden. Eine Migration nach dem COPE-Ansatz wäre denkbar, müsste aber zu den Refactorings auf der OWL-Seite synchronisiert werden.

In Refactory werden generische Refactorings erstellt, damit sie zur Definition unterschiedlicher konkreter Refactorings für ein oder mehrere Metamodelle wiederverwendet werden können. Aufgrund der starken strukturellen Unterschiede zwischen OWL und Ecore kommt dieser Mechanismus hier jedoch kaum zum Einsatz. Das entsprechende Ecore-Refactoring für das eben vorgestellte *Convert DataProperty to ObjectProperty* ist *Replace Data Value With Object*. Dieses Refactoring erfordert jedoch ein anderes Role Model, welches in Abbildung 7.8 zu sehen ist. Somit kann das zuvor definierte generische Refactoring nicht für das Ecore-Refactoring wiederverwendet werden. Abgesehen vom Rename-Refactoring lässt mit *Duplicate Class* nur ein Refactoring des Katalogs eine Wiederverwendung des generischen Refactorings zu.

Die Wiederverwendung der generischen Refactorings wird durch das komplexe Metamodell von OWL weiterhin beeinträchtigt. Selbst bei einer starken Abstraktion des Metamodells, wie im Role Model aus Abbildung 7.4 zu sehen, bleiben die Role Models zu komplex, sodass sie nicht in anderen Refactorings Anwendung finden können. So wurde für jedes der implementierten OWL-Refactoring ein eigenes generisches Refactoring angelegt.

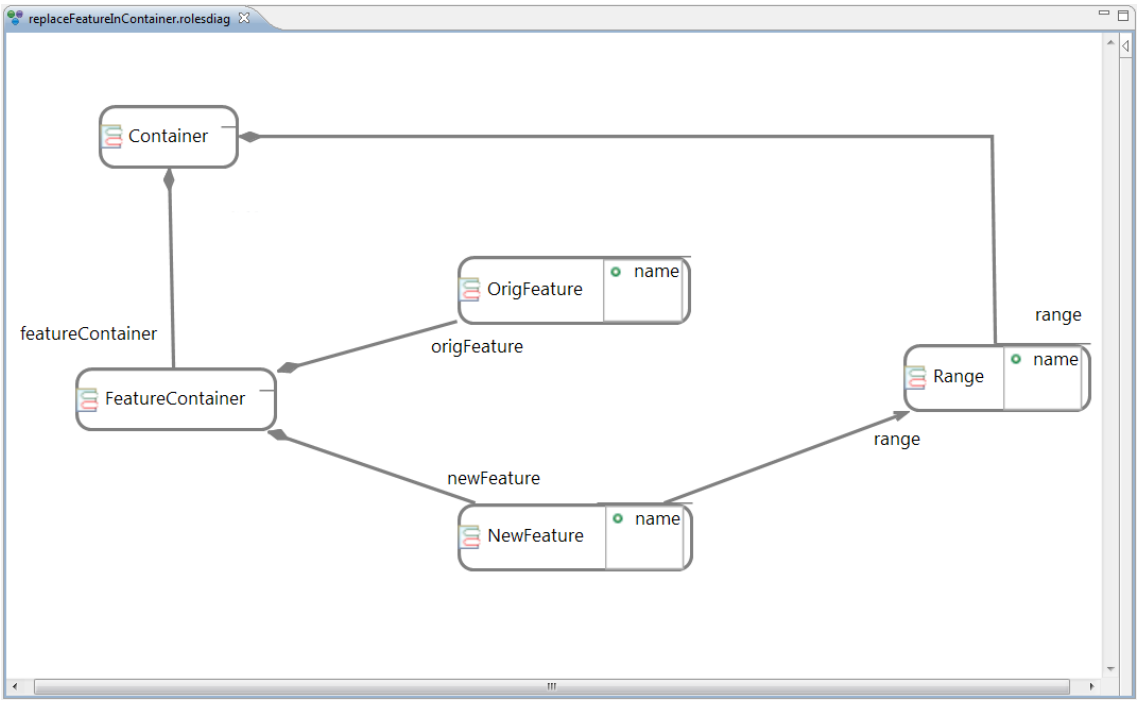

Abbildung 7.8: Role Model von Replace Feature in Container

7.5 CoRefactorer

Der CoRefactorer ist ein Software-Modul, welches das Konzept des Co-Refactoring aus Abschnitt 6.4 für den Anwendungsfall von OWL- und Ecore-Refactorings umsetzt. Es besteht aus vier Eclipse-Plugins, die in Abbildung 7.9 zu sehen sind. Das Core-Plugin stellt die eigentliche Funktion des Co-Refactorings bereit. Dazu greift es auf drei weitere Plugins zu, die jeweils eine der Komponenten aus Abbildung 6.9 darstellen. Der InputValueMapper wurde hier nicht implementiert, da für den OWL-Ecore-Anwendungsfall keine Umformung der Eingabewerte notwendig ist. Die Eingaben für OWL-Refactorings sind immer gleich zu den entsprechenden Eingaben für Ecore-Refactorings, und werden deshalb direkt übernommen. Alle vier Plugins greifen auf ein Utility-Plugin zu, das allgemeine Funktionen für die OntoMore-Implementierung bereithält, wie beispielsweise das Laden und Speichern von Modellen.

Das Core-Plugin enthält lediglich eine Klasse, die in Abbildung 7.10 zu sehen ist. Die einzige öffentliche Methode der Klasse nimmt den Input-Datensatz für das originale Modell entgegen, leitet daraus die Input-Datensätze für alle korrespondierenden Modelle ab und ruft schließlich Refactory zur Ausführung der Refactorings auf. Der Rückgabewert ist eine Liste aller refaktorierten Modelle.

Alle untergeordneten Plugins sind als Registrierungen ausgelegt, denen zur Laufzeit neue Einträge hinzugefügt werden können. Die `ModelRegistry`, zu sehen in Abbildung 7.11, besitzt ein Interface, über das Paare zusammengehöriger Modelle registriert und alle korrespondierenden Modelle eines Modells abgefragt werden können. Außerdem können einzelne oder alle Modell-Paare entfernt werden. Mit diesem Modul können also die Beziehungen zwischen originalem und den abhängigen Modellen dargestellt werden.

Die `RoleMappingRegistry` speichert die Zuordnung von konkreten Refactorings in Form von `RoleMappings`. Wie in Abbildung 7.12 zu sehen, enthält das Interface eine Methode zur Ermittlung des korrespondierenden Role Mappings abhängig vom Input-Element und der Richtung. Die Richtung enthält die Metamodelle von originalem und abhängigem Modell und gibt

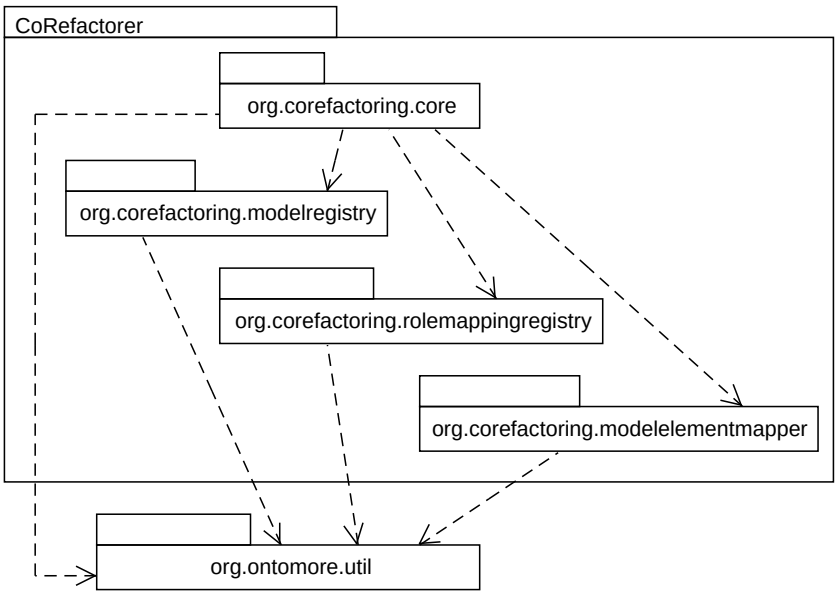

Abbildung 7.9: CoRefactorer Plugin-Architektur

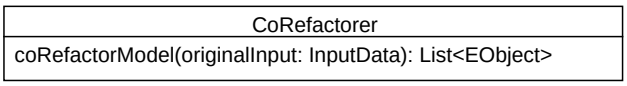

Abbildung 7.10: Die CoRefactorer-Klasse

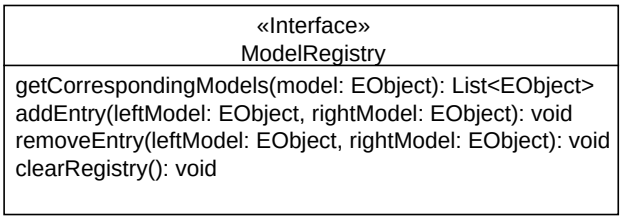

Abbildung 7.11: Die ModelRegistry

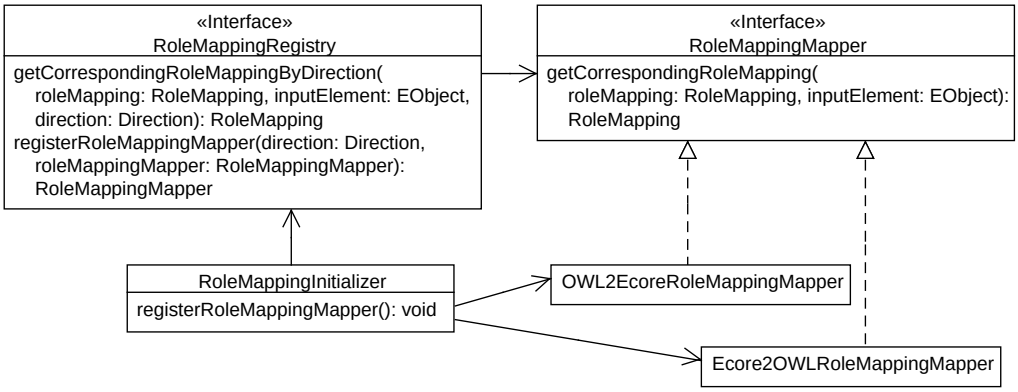

Abbildung 7.12: Die RoleMappingRegistry

somit an, von welchem Metamodell in welches andere das Refactoring zu übersetzen ist. Damit eine Zuordnung von Role Mappings erfolgen kann, müssen zunächst für jede verwendete Richtung `RoleMappingMapper` registriert werden. Diese ermitteln speziell für eine Richtung das zugehörige `RoleMapping` in Abhängigkeit vom Input-Element. Die Methode `registerRoleMappingMapper` verhält sich dabei wie die `put`-Methode einer Map. Jede Richtung muss hier separat implementiert werden, da nicht davon ausgegangen werden kann, dass es sich um eine bijektive Abbildung der Refactorings zweier Metamodelle handelt. Zum Beispiel muss das Rename-Refactoring von Ecore abhängig vom Input-Element auf eines von zwei unterschiedlichen Rename-Refactorings für das OWL-Metamodell (*Rename Element* oder *Rename Ontology*) abgebildet werden. Der `RoleMappingInitializer` ist dafür zuständig alle bekannten RoleMappingMapper zu registrieren. Diese Klasse kann durch einen beliebigen anderen Registrierungsmechanismus, z. B. eine Registrierung unter Nutzung von Eclipse-Extension-Points, ersetzt werden.

Der `ModelElementMapper` hat die Aufgabe, Elemente des originalen Modells auf Elemente der abhängigen Modelle abzubilden. Seine Struktur ist in Abbildung 7.13 zu sehen. Sie gleicht der Struktur der `RoleMappingRegistry`. Für jedes Element des originalen Modells wird also ein korrespondierendes Element im Ziel-Modell zurückgegeben. Das Ziel-Modell ist dabei eines der abhängigen Modelle der `ModelRegistry`. Die Richtung wird hier direkt aus Input-Element und Ziel-Modell bestimmt und muss nicht extra angegeben werden. Wie bei der `RoleMappingRegistry` muss für jede verwendete Richtung ein Mapper, hier ein `ModelElementMapper`, registriert werden. Dieser setzt dann die Abbildung der Input-Elemente speziell für eine bestimmte Richtung um. Wie bereits in der Konzeption (Abschnitt 6.4.3) angesprochen, gibt es mehrere Möglichkeiten, die Abbildung der Elemente zu verwirklichen. In der vorliegenden Implementierung wurde ein Suchalgorithmus umgesetzt, der die zugehörigen Elemente auf Basis der Klasse und den Eigenschaften des Input-Elements, wie dem Namen, im Ziel-Modell findet. Dieser Suchalgorithmus basiert also lediglich auf der Abbildung der Metamodelle (in diesem Fall auf der Abbildung der OWL-Ecore-Transformation) und ist unabhängig von den konkreten Modellen. Die Abbildung der Elemente über Typ und Name setzt allerdings voraus, dass die abhängigen Modelle konform zum originalen Modell bezüglich der Abbildung der OWL-Ecore-Transformation sind.

Ein wichtiger Bestandteil der Implementierung von OntoMore ist das Utility-Package. Es stellt insbesondere für den CoRefactorer wichtige Funktionen zur Verfügung. Die Abbildung 7.14 zeigt einen Überblick über die wichtigsten Klassen und Methoden dieses Pakets. Die Klasse `RefactoryUtil` stellt die Anbindung an Refactory dar. Sie enthält Methoden, um Role Model,

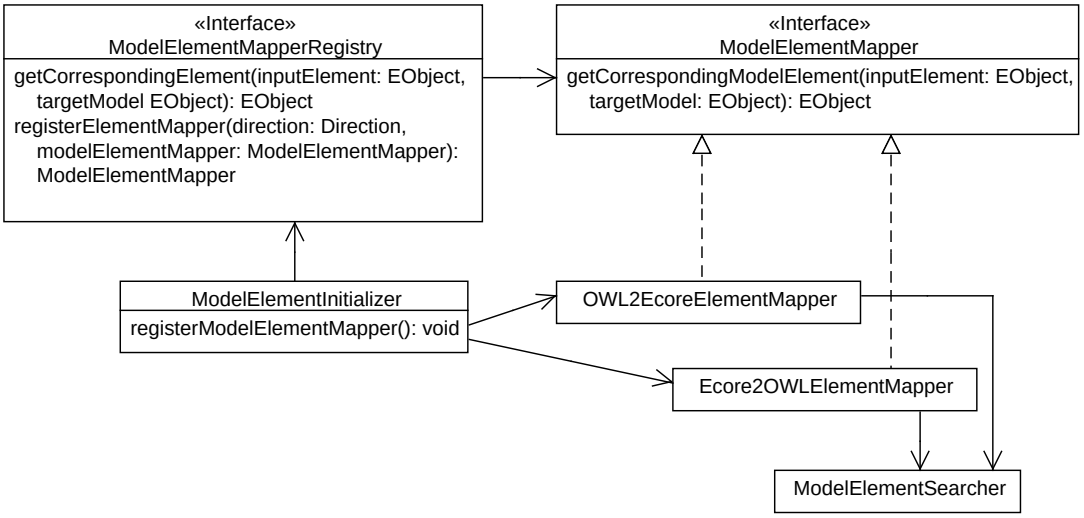

Abbildung 7.13: Der ModelElementMapper

Refactoring Specification und Role Mapping Model zu registrieren und somit die Refactorings dem Programm bekannt zu machen. Im Anschluss können mit der Methode `refactorModel` Refactorings basierend auf einem Input-Datensatz ausgeführt werden. Die Klasse `InputData` stellt genau einen solchen Input-Datensatz dar und enthält, wie von der Konzeption vorgesehen, Modell, Role Mapping, Input-Elemente und Eingabewerte. Die schon besprochene Richtung wird von der Klasse `Direction` implementiert. Quell- und Ziel-Metamodell werden in ihr als Strings der Metamodell-URIs repräsentiert. Die Klasse `ModelStorageUtil` ist für das Laden und Speichern von Modellen zuständig. Der `ModelComparator` gibt unter Nutzung von EMF Compare die Auskunft, ob zwei Modelle gleich sind. Er wird, wie schon erwähnt, zum Testen des OWL-Ecore-Transformators und des CoRefactorers verwendet.

Das Utility-Paket enthält weiterhin ein Unterpaket, das `ModelProvider` enthält, die in Abbildung 7.15 im Detail zu sehen sind. Diese Enums dienen während der Tests dazu, die Modelle von Refactory bereitzustellen, die zur Darstellung der Refactorings verwendet werden: Role Model, Refactoring Specification, Role Mapping Model und Role Mapping. Jeder `ModelProvider` enthält ein Enum-Literal für jedes bekannte Modell inklusive dessen Dateipfad und eine Methode, die das Modell lädt und zurückgibt. Die `RoleMappingProvider` stellen den direkten Zugriff auf die Role Mappings zur Verfügung. Role Mappings haben kein eigenes Modell, sondern sind Bestandteil des Role Mapping Models. Das Role Mapping Model *Rename* für OWL enthält beispielsweise die zwei Role Mappings *Rename Element* und *Rename Ontology*. Jeder `RoleMappingProvider` speichert die Zuordnung von Role Mappings zu Role Mapping Models, damit das richtige Role Mapping Model geladen und das gewünschte Role Mapping daraus abgefragt werden kann.

Für den Test der Implementierung enthält jedes Paket des CoRefactorer ein Unterpaket mit Unit-Tests, welche die öffentlichen Methoden der Klassen des Pakets testen. Diese Tests wurden entsprechend der TDD-Vorgehensweise angelegt und tragen somit auch zur Dokumentation des System-Designs bei. Die einzige Ausnahme von dieser Positionierung der Tests sind die Tests des Core-Plugins. Diese wurden in das separate Plugin `org.corefactoring.test` ausgelagert, da sie umfangreiche Testdaten benötigen. In diesem Plugin werden alle implementierten Refactorings für OWL und Ecore sowie alle Co-Refactorings getestet. Dazu gibt es für jedes Refactoring einen Unterordner mit drei Dateien. Die Datei mit der Endung „`_IN`“

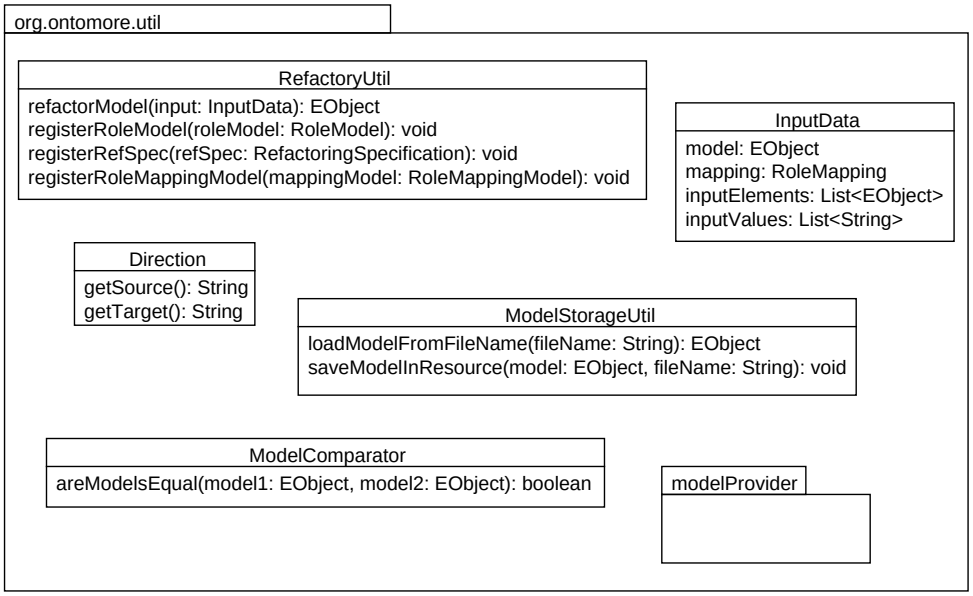

Abbildung 7.14: Das OntoMore-Utility-Paket

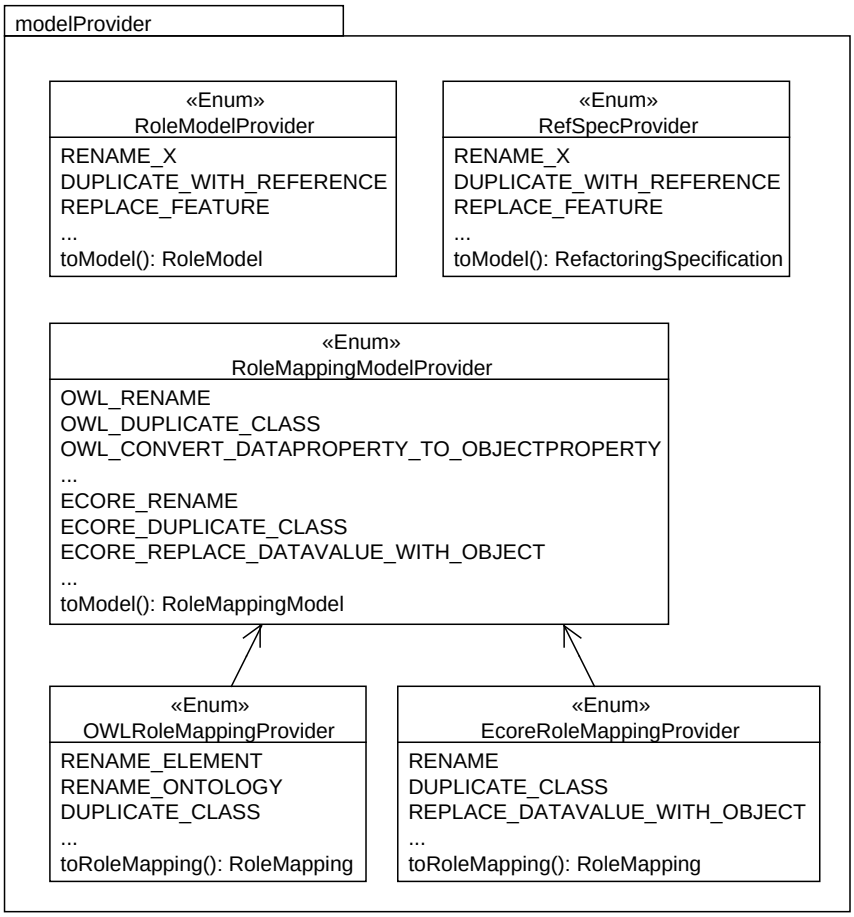

Abbildung 7.15: Die ModelProvider

enthält das zu refaktorierende Modell, die Datei mit der Endung „_EXP“ enthält das erwartete Ergebniss und die Datei mit der Endung „_REF“ enthält das tatsächliche Ergebnis des Refactorings zum manuellen Vergleich. Für die Co-Refactorings liegen für das originale und für das abhängige Modell jeweils diese drei Dateien vor. Wobei Co-Refactoring-Tests für beide Richtungen, von OWL nach Ecore und von Ecore nach OWL, vorliegen. Womit gezeigt ist, dass das Co-Refactoring unabhängig von der Richtung funktioniert. Darüber hinaus gruppiert die Klasse CoRefactoringTestSuite alle Unit-Test zu einer großen Regressions-Testsuite für das komplette Programm.

7.6 Zusammenfassung

Dieses Kapitel zeigte die technische Umsetzung von OntoMore unter Nutzung der vorhandenen Werkzeuge OntoMoPP und Refactory. Die Implementierung wurde nach dem Prinzip der TDD umgesetzt und besteht aus dem OWL-Ecore-Transformator, den Refactorings und dem CoRefactorer.

Der OWL-Ecore-Transformator transformiert Ontologien in Ecore-Metamodelle und dazugehörige Modelle. Diese Transformation wurde programmatisch in Java umgesetzt, um einer Metaebenen-übergreifenden Transformation gerecht zu werden. Im Zusammenspiel mit dem OWLizer ermöglicht dies eine bidirektionale Transformation zwischen transformierbaren Ontologien und Ecore-Modellen.

Die Refactorings für OWL und Ecore wurden mit den DSLs von Refactory umgesetzt. Die Komplexität des OWL-Metamodells erforderte, dass für jedes Refactoring ein eigenes Role Model, Refactoring Specification und Role Mapping definiert wurde. Es war aufgrund technischer Einschränkungen von Refactory jedoch nicht möglich, innerhalb dieser Refactorings die Datenmigration zu verwirklichen.

Der CoRefactorer erweitert das Werkzeug Refactory um die Möglichkeit, korrespondierende Refactorings auf mehreren in Beziehung stehenden Modellen auszuführen. Dazu wurden drei Module erstellt, welche die jeweils korrespondierenden Werte für Modelle, Refactorings (in Form von Role Mappings) und Input-Elemente ermitteln. Das Programm wurde mit einer umfangreichen Testsuite getestet, die außerdem die Funktionsfähigkeit des Programms demonstriert.

8 Evaluation

8.1 Die Beispiel-Ontologie: FrOnto

Zur Evaluation der erarbeiteten Konzepte wurde die Beispiel-Ontologie **FrOnto** (Freelancer-Ontologie) erstellt. Diese Ontologie enthält Begriffe, deren Definition und deren Beziehungen für den Anwendungsbereich von Freelancern, Projekten und Unternehmen. Mit FrOnto können Freelancer mit ihren Qualifikationen und beruflichen Schwerpunkten genauso wie Projekte mit ihren Anforderungen an die Projektteilnehmer detailliert beschrieben werden. Ziel der Ontologie ist es, ein gemeinsames Vokabular zur Beschreibung von Freelancern und Projekten zu etablieren. Somit können Freelancer entsprechend ihrer Qualifikationen den passenden Projektstellen zugewiesen werden.

Abbildung 8.1 zeigt das Einsatz-Szenario von FrOnto. Dieses Szenario wurde auf Basis der Erkenntnisse aus der Diplomarbeit von Christiane Köhler erstellt [Köh09]. Auf der einen Seite stellen Freelancer Informationen über ihre Qualifikationen auf offene Plattformen im Web zur Verfügung, während auf der anderen Seite IT-Unternehmen Informationen über Projekte auf diesen Plattformen veröffentlichen. Somit erfolgt ein gegenseitiger Informationsaustausch, der es ermöglicht, dass Freelancer passende Projekte und Projektanbieter passende Freelancer für ihre Projekte finden können. Als eine dritte Partei können auch Projektvermittler auf die Daten zugreifen und gezielt passende Freelancer und Projekte zusammenbringen. Projektvermittler werden im Rahmen dieser Arbeit jedoch als reine Benutzer behandelt, die das System zwar benutzen aber keine eigenen Daten hinein stellen. Die FrOnto spielt in diesem Zusammenhang die Rolle eines gemeinsamen Vokabulars und stellt sicher, dass alle beteiligten Freelancer und Unternehmen Daten in denselben Formaten austauschen und sich somit verstehen können. Die veröffentlichten Daten sind einfache RDF-Tripel. Diese werden entweder direkt in XHTML-Dokumente eingebunden, zum Beispiel mit RDFa [AB08], oder in Ontologien zusammengefasst. Die Tripel stellen dabei die Beziehung der Daten zu den Konzepten der FrOnto her. Dies geschieht über die global eindeutigen IRIs und den Import-Mechanismus der Ontologien.

Der Vorteil dieses Ansatzes ist ein stark verringerter Aufwand der Datenhaltung und -erfassung und ein vereinfachter Datenaustausch. Dies resultiert daraus, dass nicht mehr jedes Unternehmen seine eigene proprietäre Freelancer-Anwendung verwalten muss, sondern auf offene Plattformen und die FrOnto als einen gemeinsames Vokabular zugreifen kann. Mit einer offenen Plattform ist dabei ein System gemeint, an dem sich jeder interessierte Freelancer und jedes interessierte Unternehmen zum Informationsaustausch registrieren kann. Die auf der Plattform zur Verfügung gestellten Informationen sollten allerdings Zugangsbeschränkungen unterliegen, die der Urheber der Daten festlegt.

Auch für die Freelancer verbessert sich die Situation, da sie ihre Daten auf nur noch einer Plattform in Form von RDF-Tripeln konform zur FrOnto ablegen. Von dort können die Daten dann an andere Plattformen oder direkt an Unternehmen verteilt werden. Somit entfällt die Notwendigkeit, mehrere Profile bei unterschiedlichen Unternehmen verwalten zu müssen.

Die Standardisierung der Daten durch die Ontologie verbessert den Datenaustausch dahingehend, dass eine Konvertierung zwischen unterschiedlichen Formaten nicht mehr notwendig ist. Jedes Unternehmen könnte seine eigene Anwendung einsetzen, um auf die standardisierten Daten zuzugreifen und diese in das gewünschte Format umzuformen.

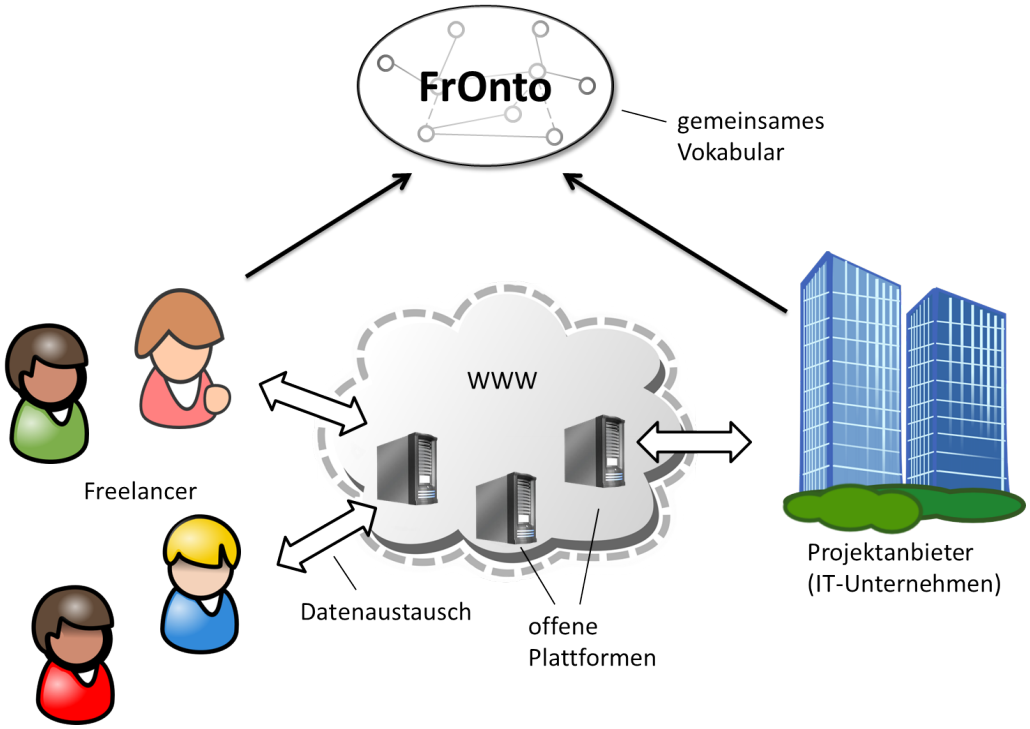

Abbildung 8.1: Einsatz-Szenario von FrOnto

Die Beispiel-Ontologie wurde mit dem Ziel erstellt, die in dieser Arbeit entwickelten Konzepte zur Evolution von ontologiegetriebenen Softwaresystemen zu bewerten. FrOnto stellt somit nicht die komplette Anwendungsdomäne von Freelancern und Projekten in vollem Detailumfang dar, sondern ist eher als erster Anhaltspunkt zur Modellierung der wichtigsten Begriffe dieser Domäne zu verstehen. Die Abbildung 8.2 gibt einen Überblick über die wichtigsten Begriffe der Ontologie.

FrOnto wurde basierend auf den Tutorials zum Ontologie Engineering von Horridge sowie Noy und McGuiness erstellt [Hor11, NM01]. Wie schon in Kapitel 6.2 angemerkt, beschränkt sich der erarbeitete Ansatz auf Ontologien ohne Import-Beziehungen. Aus diesem Grund hat auch die FrOnto keine Import-Beziehungen, abgesehen von den Standard-Imports. Im Allgemeinen ließen sich jedoch Konzepte aus vielen bestehenden Top-Level-Ontologien, wie DC, FOAF oder DOLCE Upper Ontology[1], wiederverwenden.

Die Ontologie nutzt Funktionen, die mit der Version 2 neu in den OWL-Standard aufgenommen wurden, um Bereiche der Anwendungsdomäne einfach und präzise zu modellieren. Als Beispiel sei an dieser Stelle die definierte Klasse *FreelancerMitJavaKenntnissen* erklärt. Ihr einziges EquivalentTo-Axiom hat in Manchester OWL Syntax die Form „`Freelancer that hatFertigkeit some JavaTechnik`“. *JavaTechnik* ist dabei eine Enumerated-Klasse, die mit einem OneOf-Axiom alle Individuen der Klasse Technik gruppiert, die mit Java in Zusammenhang stehen. Somit kann ein Reasoner ableiten, dass alle Freelancer, die eine dieser Java-Techniken kennen, zur Klasse *FreelancerMitJavaKenntnissen* gehören. Nach diesem Prinzip können Freelancer, Unternehmen, Projekte und Stellenausschreibungen nach unterschiedlichen Kriterien klassifiziert werden.

[1] `http://www.loa-cnr.it/DOLCE.html`

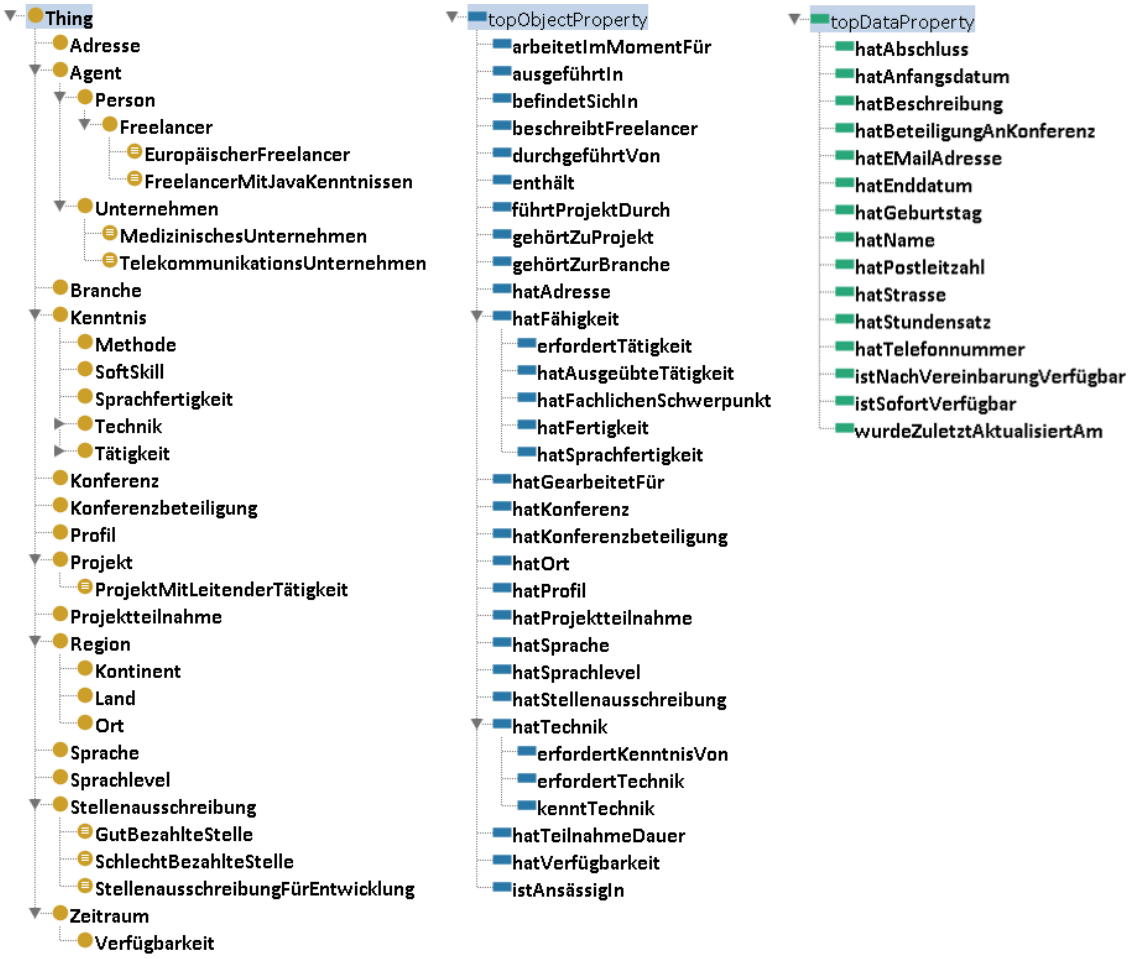

Abbildung 8.2: Klassen, ObjectPropertys und DataPropertys von FrOnto

8.2 Bewertung der Umsetzung

In diesem Abschnitt wird eine Bewertung der praktischen Umsetzung aus Kapitel 7 vorgenommen. Diese Umsetzung konzentriert sich auf die Evolution als eine der drei Anforderungen an ontologiegetriebene Softwaresysteme aus dem Abschnitt 6.1. Die beiden anderen Anforderungen, Design und Codegenerierung, liegen nicht im Fokus dieser Arbeit. Deshalb beschränken sich die folgenden Ausführungen auf eine Bewertung der Einsatzfähigkeit der implementierten Ontologie-Refactorings.

Zur einfacheren Darstellung der Refactorings, wurde die Beispiel-Ontologie auf die *MiniFrOnto* reduziert. Diese enthält nur noch die Konzepte, auf die sich die Ontologie-Refactorings beziehen. Somit kann auf verständliche Weise gezeigt werden, an welchen Stellen die implementierten Refactorings zum Einsatz kommen. Wie bereits in Kapitel 7.4 erwähnt, beschränken sich die implementierten Ontologie-Refactorings auf die TBox. Deshalb enthält die MiniFrOnto keine Individuen. Auf die Datenmigration wird aber noch am Ende des Abschnitts in einem Vergleich mit der OWL-API eingegangen.

Während der Entwicklung von FrOnto hat sich herausgestellt, dass alle implementierten Refactorings zum Einsatz kommen können. Die jeweiligen Einsatzbereiche der Refactorings sollen im Folgenden kurz vorgestellt werden. Die zwei Zustände der MiniFrOnto, vor und nach den Refactorings, sind im Anhang C dargestellt.

Das **Rename**-Refactoring kann beispielsweise bei der Klasse *Freiberufler* eingesetzt werden, um sie in *Freelancer* umzubenennen. Freiberufler der IT-Branche werden in der vorliegenden Anwendungsdomäne durch diesen neuen Begriff besser beschrieben. Sämtliche Verweise auf die umbenannte Klasse werden automatisch angepasst, sodass die syntaktische Konsistenz der Ontologie erhalten bleibt. Auf die semantische Konsistenz und die Erhaltung der Individuen hat dieses Refactoring keine Auswirkungen.

Das Refactoring **Duplicate Class** kann beim Anlegen der unterschiedlichen Techniken zum Einsatz kommen. Ein Freelancer kann seine Qualifikation beschreiben, indem er angibt, welche Techniken er kennt. Diese Techniken sind in Unterklassen unterteilt, die alle sehr ähnlich sind. Es ist deshalb sinnvoll, zum Erstellen einer neuen Unterklasse eine bestehende Klasse zu kopieren. Somit wird beispielsweise die Klasse *Datenbank* erstellt, indem die Klasse *Programmiersprache* kopiert und mit einem neuen Namen versehen wird. Das SubClassOf-Axiom zur Klasse *Technik* wird dabei für die neue Klasse übernommen. Im Allgemeinen werden auch alle weiteren SubClassOf-Axiome auf die neue Klasse übertragen.

Die *Adresse* eines Freelancers wurde zuerst als DataProperty *hatAdresse* modelliert, die lediglich einen String enthält. Mit dem Refactoring **Convert DataProperty to ObjectProperty** kann die Adresse zu einer eigenständigen Klasse umgeformt werden. Dazu wird die DataProperty durch eine ObjectProperty ersetzt und die neue Klasse *Adresse* eingefügt. Diese neue Klasse könnte im Anschluss durch Data- oder ObjectPropertys erweitert werden, mit denen Details der Adresse dargestellt werden, zum Beispiel *hatOrt*, *hatStraße* und *hatPostleitzahl*.

Das **Pull Up Property**-Refactoring dient dazu, die Domain-Einschränkung einer Property zu lockern. Die MiniFrOnto enthält die Klasse *Verfügbarkeit*, welche die Verfügbarkeit eines Freelancers mit einem Anfangs- und einem Enddatum, dargestellt durch zwei DataPropertys, angibt. Später kam die Klasse *Zeitraum* hinzu, mit der zum Beispiel angegeben wird, wie lange ein Projekt dauert. Die Klasse *Zeitraum* ist eine Verallgemeinerung und damit eine Superklasse von *Verfügbarkeit*. Damit die zwei DataPropertys *Anfangs-* und *Enddatum* auch von Individuen der Klasse *Zeitraum* genutzt werden können, müssen deren Domain-Einschränkungen durch diese Superklasse ersetzt werden. Dies wird durch das *Pull Up Property*-Refactoring umgesetzt.

Wie schon zuvor erwähnt, gibt es in der MiniFrOnto die Klasse *Freelancer*, welche die Domain der Property *hatAdresse* darstellt. Jedoch ist es etwas zu speziell, dass nur Freelancer Adressen

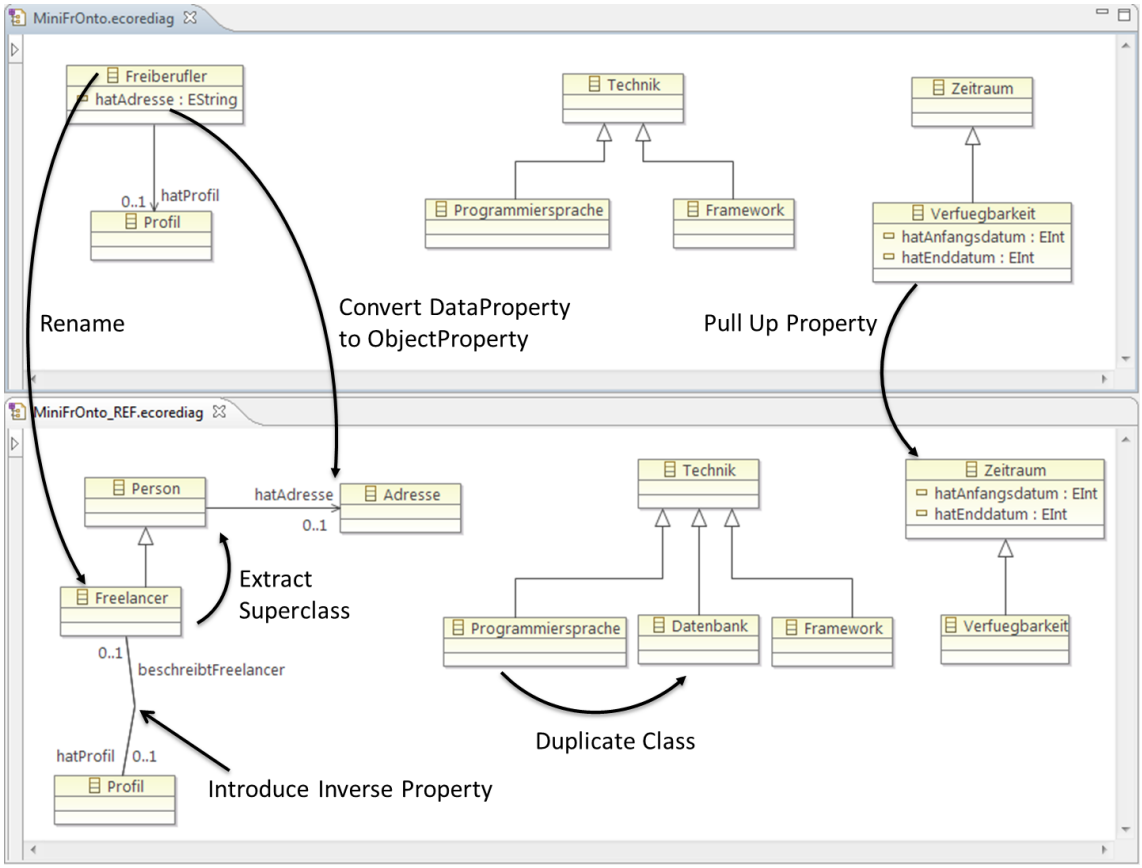

Abbildung 8.3: Refactorings auf dem assoziiertem Ecore-Modell

haben, da dies im Allgemeinen für alle Personen gilt. Für diesen Fall kann das Refactoring **Extract Superclass** auf der Property *hatAdresse* angewandt werden, um die neue Superklasse *Person* zu erstellen und gleichzeitig als Domain der Property *hatAdresse* zu verwenden.

Jeder Freelancer hat ein *Profil*, in dem sein beruflicher Werdegang festgehalten ist. Die DataProperty *hatProfil* verbindet dazu den Freelancer mit seinem Profil. Beim Aufruf eines Profils wäre es jedoch wünschenswert, auch wieder zurück zum Freelancer navigieren zu können. Dies kann mit einer inversen Property *beschreibtFreelancer* erreicht werden. Das Refactoring **Introduce Inverse Property** erstellt diese inverse Property automatisch.

Die vorgestellten Refactorings können auch am Ecore-Metamodell nachvollzogen werden, das mit der MiniFrOnto in Beziehung steht. Dazu wurde die MiniFrOnto mit dem OWL-Ecore-Transformator in ein Ecore-Metamodell transformiert. Die MiniFrOnto ist dabei vollständig transformierbar, wohingegen die FrOnto auch Konzepte enthält, die nicht transformiert werden können, zum Beispiel die bereits erwähnte definierte Klasse *FreelancerMitJavaKenntnissen*. Die Abbildung 8.3 zeigt das Metamodell der MiniFrOnto in der oberen Hälfte vor und in der unteren Hälfte nach den Refactorings.

Die besprochenen Refactoring-Situationen der FrOnto zeigen, dass die implementierten Refactorings unter realen Bedingungen tatsächlich zum Einsatz kommen können. Allerdings sei noch einmal darauf hingewiesen, dass die implementierten Refactorings nicht das komplette Refactoring-Schema aus Abschnitt 6.3 abdecken. Es werden keine Vor- und Nachbedingungen geprüft und die Refactorings beschränken sich auf die TBox der Ontologie. Das Prüfen der Vor- und Nachbedingungen liegt außerhalb des Rahmens der Arbeit. Die Beschränkung auf die TBox resultiert aus den technischen Einschränkungen von Refactory.

Den Einsatz von Refactory als Werkzeug zum Ontologie-Refactoring gilt es weiterhin zu überdenken. Refactory wurde als Werkzeug zum Modell-Refactoring entwickelt. Im Ansatz dieser Arbeit werden Ontologien auf Ecore-Modelle abgebildet und liegen ihrerseits als Modelle vor. Deshalb wurde Refactory als Werkzeug gewählt. Jedoch hat sich im Laufe der Arbeit herausgestellt, dass die Vorteile der Generizität von Refactory nicht auf Ontologien übertragen werden können und dass das Werkzeug die bei Ontologien erforderliche Komplexität der Umstrukturierungen nicht ausreichend abdeckt. Insbesondere die Erhaltung der Individuen erfordert oft eine komplexe Ausführungssemantik mit Prüfungen und Datenzugriffen über verteilte Bereiche der gesamten Ontologie. Das bedeutet, dass die DSL der Refactoring Specification bis zu einer Ausdrucksstärke, die der einer GPL nahe kommt, erweitert werden müsste. In diesem Fall wäre jedoch der Einsatz einer DSL nicht mehr gerechtfertigt, da die Umstrukturierungen dann mit geringerem Aufwand auch direkt in Java über die OWL-API implementiert werden können.

Wird Java zur Implementierung der Refactorings eingesetzt, besteht allerdings keine Möglichkeit des Co-Refactorings von Modellen. Diese Einschränkung könnte umgangen werden, indem der Post-Prozessor von Refactory verwendet wird. Der Post-Prozessor dient dazu, nach Ausführung eines Refactorings Umstrukturierungen und Prüfungen vorzunehmen, die über die Möglichkeiten der Refactoring Specification hinausgehen. Er kann in diesem Zusammenhang genutzt werden um in Java implementierte Refactorings aus Refactory heraus aufzurufen und über den CoRefactorer mit anderen Modell-Refactorings in Beziehung zu setzen. Über den Post-Prozessor können die Ontologie-Refactorings somit ohne weiteren Aufwand in die Eclipse-Umgebung integriert werden.

Zum Vergleich der Refactory-Umsetzung mit einer Java-Implementierung, wurde das Refactoring *Convert DataProperty to ObjectProperty* mit Java unter Nutzung der OWL-API implementiert. Die OWL-API ermöglicht es, in aussagekräftiger Notation beliebige Operationen auf der Ontologie auszuführen. Dabei werden für Klassen und Methoden in der OWL-API die gleichen Bezeichnungen verwendet, wie im OWL 2-Standard. Der Vorteil besteht hier darin, dass Refactorings in lediglich einer Technologie (Java) umgesetzt werden und nicht in drei verschiedenen DSLs. Das Listing 8.1 zeigt den Teil der Java-Implementierung, der der Refactory-Umsetzung aus Kapitel 7.4 entspricht. Allerdings sorgt die OWL-API beim Löschen der DataProperty automatisch dafür, dass alle abhängigen Axiome gelöscht werden, sodass die Ontologie konsistent bleibt. Weiterhin ist es mit Java möglich, das komplette Refactoring-Schema aus Abschnitt 6.3, und somit auch die Datenmigration, umzusetzen. Der vollständige Code des Refactorings befindet sich zum Vergleich mit der Refactory-Umsetzung im Anhang D.

Zur abschließenden Bewertung der Refactoring-Architektur aus Kapitel 7, bestehend aus OntoMore und den zugrundeliegenden Werkzeugen Refactory und OntoMoPP, wird im Folgenden behandelt, in welchem Umfang die Anforderungen an Ontologie-Evolution aus dem Abschnitt 5.2.1 erfüllt wurden. Somit kann die erarbeitete Umsetzung in bestehende Techniken eingeordnet werden.

1) Abdeckung Mit OWLText können alle Konstrukte der Ontologie bearbeitet werden, wobei syntaktische und semantische Fehler schon während der Eingabe angezeigt werden. OWLText setzt dabei im Gegensatz zu anderen Werkzeugen auf die Manchester OWL Syntax zur Speicherung der Ontologie und basiert auf einem OWL-Metamodell, das nicht konform zum OWL 2-Standard ist. Die erarbeiteten Refactorings erlauben darüber hinaus komplexe Umstrukturierungen der TBox, welche die Konsistenz der Ontologie erhalten.

2) Steuerung durch den Benutzer Zur Steuerung der Refactorings sieht Refactory Eingabedialoge vor, die Eingaben abhängig vom Refactoring vom Benutzer entgegennehmen, zum Beispiel den Namen eines neu zu erstellenden Elements. Somit kann der Benutzer Einfluss

Listing 8.1: Ausschnitt der Implementierung von *Convert DataProperty to ObjectProperty* mit Java und OWL-API

```
void executeRefactoringSteps() {
    // create ObjectProperty and set range
    objectProperty = factory.getOWLObjectProperty(inputElement.getIRI());
    rangeClass = factory.getOWLClass(":" + inputValue, pm);
    OWLAxiom rangeAxiom = factory.getOWLObjectPropertyRangeAxiom(objectProperty,
        rangeClass);
    axiomsToAdd.add(rangeAxiom);

    // transfer domains
    Set<OWLClassExpression> domains = inputElement.getDomains(ontology);
    for (OWLClassExpression domain : domains) {
        OWLAxiom domainAxiom = factory.getOWLObjectPropertyDomainAxiom(
            objectProperty, domain);
        axiomsToAdd.add(domainAxiom);
    }

    // remove DataProperty
    OWLEntityRemover remover = new OWLEntityRemover(manager, Collections.singleton(
        ontology));
    inputElement.accept(remover);
    changeAxioms.addAll(remover.getChanges());
}
```

darauf nehmen, auf welche Elemente sich ein Refactoring auswirkt und welche Eigenschaften neu erstellte Elemente haben. Die Evolutionsstrategien von Stojanovic aus Abschnitt 5.2.3 können jedoch mit Refactory nicht umgesetzt werden, da die Refactoring Specification nicht zur Laufzeit verändert werden kann.

3) Transparenz Bevor eines der implementierten Refactorings tatsächlich ausgeführt wird, zeigt Refactory einen Dialog an, in dem der Zustand der Ontologie vor und nach dem Refactoring gegenübergestellt wird. Somit kann der Benutzer schon vor der Ausführung erkennen, welche Änderungen aus dem Refactoring resultieren.

4) Umkehrbarkeit OWLText besitzt eine Undo/Redo-Funktion, die auch für ausgeführte Refactorings zur Verfügung steht. Dabei wird ein Refactoring als eine atomare Änderung behandelt und kann als solche auch wieder komplett rückgängig gemacht werden.

5) Bedienbarkeit Wie bereits erwähnt, setzt OWLText auf die Manchester OWL Syntax zur Bearbeitung der Ontologie. Mit den Funktionen der Syntax-Hervorhebung und Code-Vervollständigung können damit zumindest kleine Ontologien schnell und einfach bearbeitet werden. Mögliche Refactorings werden dabei über das Kontextmenü eingeblendet, abhängig davon, welches Element bzw. welche Elemente der Benutzer ausgewählt hat.

8.3 Grenzen der Umsetzung

An dieser Stelle soll auf die Einschränkungen und mögliche Probleme der Umsetzung eingegangen werden. Die technische Umsetzung von OntoMore basiert mit OWLText auf der Manchester OWL Syntax zur Darstellung und Bearbeitung der kompletten Ontologie. Das dahinterliegende OWL-Metamodell ist demnach auf die Präsentation dieser Syntax ausgelegt und damit nicht konform zum OWL 2-Standard. Insbesondere können nicht alle Ontologien in der Manchester OWL Syntax dargestellt werden, da diese Syntax einige Funktionen von

OWL 2 nicht unterstützt. Dies betrifft vor allem Konzepte, die mit Frames nicht darstellbar sind. Dazu gehören unter anderem *General Concept Inclusions* (GCI). GCIs treten immer dann auf, wenn eine anonyme Klasse auf der linken Seite eines SubClassOf-Axioms auftritt. Dieses Axiom hat laut der OWL 2-Spezifikation den Aufbau `SubClassOf(subClassExpression superClassExpression)`. Dabei können die Klassen-Ausdrücke für Unter- und Superklasse beliebig komplex sein. Die Manchester OWL Syntax als Frame-basierte Syntax beschränkt den Klassen-Ausdruck für die Unterklasse jedoch auf atomare Klassen. Beispielsweise kann so eine Menge von Individuen, die mit dem OneOf-Operator gruppiert werden, nicht mehr als Unterklasse einer bestehenden Klasse definiert werden.

Vor dem Einsatz der Manchester OWL Syntax muss also festgestellt werden, ob derartige Konstrukte zur Beschreibung der Anwendungsdomäne eingesetzt werden sollen. Es kann keine allgemeingültige Aussage darüber getroffen werden, wann GCIs benötigt werden, da dies immer von der Art und Weise abhängt, wie die Domäne modelliert wird. Allerdings enthält bereits die FrOnto, als eine relativ einfache Ontologie, mit der Klasse *JavaTechnik* eine indirekte GCI.

Eine weitere Einschränkung betrifft die Arbeit mit großen Ontologien. Während der Erstellung der FrOnto hat sich gezeigt, dass die Bearbeitung von Ontologien in einem Text-Editor schnell an seine Grenzen stößt. Trotz der recht kompakten Syntax, In-Code-Navigation und Code-Vervollständigung wird die Ontologie schnell unübersichtlich und schwer zu bearbeiten. Schon die recht kleine FrOnto umfasst in Manchester OWL Syntax über 1800 Zeilen. Für eine einfache Bearbeitung sind Ansichten notwendig, die Elemente einzelner Aspekte der Ontologie mit deren Beziehungen zu anderen Elementen zeigen. Protégé bietet dazu beispielsweise eine Ansicht, die alle Klassen und die zur ausgewählten Klasse gehörenden Individuen anzeigt.

Beim Einsatz großer Ontologien muss auch die Performance der Anwendung berücksichtigt werden. Im Rahmen des SEALS-Projekts[2] wurden erste Performance-Messungen für existierende Ontologie-Werkzeuge und -Editoren durchgeführt [Cas10]. Diese Messungen umfassen lediglich die Zeit für den Import und Export einer Ontologie und zeigen, dass diese für reale Ontologien durchaus mehrere Sekunden betragen kann. Performance-Tests für Protégé zeigen, dass eine dateibasierte Ontologie bis zu einer Größe von etwa 100.000 Elementen eingesetzt werden kann [PW08]. Größere Ontologien mit mehreren Millionen von Elementen können in Datenbanken gespeichert werden, auf welche die OWL-API ebenfalls zugreifen kann. Für OWLText existieren noch keine Performance-Tests. Allerdings ist OWLText auf dateibasierte Ontologien beschränkt und stellt Ontologien intern als EMF-Modelle dar, was die Performance negativ beeinflussen könnte. Vor dem Einsatz von OWLText sollte also die Größe der zu erstellenden Ontologie abgeschätzt werden, um mögliche Performance-Probleme zu vermeiden.

8.4 Grenzen der Konzeption

In Kapitel 6 wurde angeführt, dass das Konzept ontologiegetriebener Softwaresysteme allgemein für alle Softwaresysteme angewandt werden kann. Diese Überlegung resultiert aus der Tatsache, dass Ontologien ausdrucksstark genug sind, nicht nur große Teile der realen Welt sondern auch beliebige Softwaresysteme zu beschreiben. Es muss aber angemerkt werden, dass das Prinzip, Anwendungen automatisch aus Ontologien zu generieren, nur für eine begrenzte Auswahl von Anwendungen praktikabel ist. Dies betrifft vor allem wissensbasierte Systeme, die eine große Menge an vernetzten Daten halten, um deren Benutzern relevante Informationen zur Verfügung zu stellen. Ontologien eignen sich besonders gut zur Modellierung solcher Anwendungen, weshalb die größten Anwendungsfelder von Ontologien in der Medizin, Biologie, Geografie und öffentlichen Datensammlungen zu finden sind [CJ10]. Solche wissensbasierten

[2] `http://www.seals-project.eu/`

Systeme stellen weiterhin die Basis des Semantic Web dar, für das Ontologien ursprünglich entwickelt wurden.

Wenn Anwendungen automatisch aus einer Ontologie abgeleitet werden sollen, ist es jedoch nötig, Teile der Anwendung zu modellieren, die nicht zum eigentlichen Datenbestand gehören [Knu04]. Dazu zählen zum Beispiel interne Datenstrukturen oder Programmelemente, welche die Interaktion mit dem Benutzer steuern. In wissensbasierten Systemen sind nur wenige solcher Programmelemente zu finden, da sie im Wesentlichen für die Eingabe und Anzeige von Daten ausgelegt sind. Softwaresysteme anderer Bereiche, wie beispielsweise Finanzsoftware oder Unternehmensmanagment-Systeme, erfordern hingegen komplexe Geschäftslogiken, mit denen Berechnungen und programminterne Abläufe beschrieben werden. Natürlich können auch Prozesse und Abläufe mit Ontologien modelliert werden, dies würde jedoch schnell zu komplex und schwer verständlich werden, da Ontologien nicht darauf ausgelegt sind. Programmiersprachen sind an dieser Stelle besser geeignet, um solche Abläufe zu beschreiben.

Daraus folgt, dass vor dem Einsatz ontologiegetriebener Softwaresysteme genau abzuwägen ist, ob die Anwendungsdomäne für die Modellierung mit einer Ontologie geeignet ist. Weiterhin gilt es, ähnlich zur MDSD, zu entscheiden, welche Teile der Ontologie tatsächlich in ausführbaren Programmcode transformiert werden sollen. Beispielsweise ist es vorstellbar, dass die Ausführungssemantik nur im Programmcode definiert wird, während die Struktur des Systems aus der Ontologie abgeleitet wird.

8.5 Zusammenfassung

In diesem Kapitel wurde mit der FrOnto eine Beispiel-Ontologie vorgestellt, die beispielhaft zeigt, wie eine Anwendungsdomäne mit einer Ontologie modelliert werden kann. Alle im Rahmen dieser Arbeit implementierten Ontologie-Refactorings fanden bei der Erstellung dieser Ontologie Anwendung, womit deren Einsetzbarkeit gezeigt ist. Zum Vergleich mit der Umsetzung der Refactorings mit Refactory, wurde ein einzelnes Refactoring mit Java und der OWL-API implementiert. Mit dieser Umsetzung ist es schwieriger, eine Beziehung zu anderen Refactorings innerhalb des Co-Refactorings herzustellen, aber dafür kann das Refactoring selbst wesentlich einfacher beschrieben werden.

Außerdem wurden die Grenzen der praktischen Umsetzung dieser Arbeit betrachtet. Diese bestehen vor allem darin, dass die Manchester OWL Syntax nicht den kompletten OWL 2-Standard abdeckt und ein reiner Text-Editor für große Ontologien ungeeignet sein kann.

Bezüglich der Einschränkungen ontologiegetriebener Softwaresysteme wurde die Überlegung angestellt, dass solche Systeme vor allem für wissensbasierte Anwendungen und weniger für Systeme mit komplexer Geschäftslogik geeignet sind.

9 Zusammenfassung

9.1 Ergebnisse

Ziel der Arbeit war es, zu untersuchen, inwieweit Ontologien zum Design von Softwaresystemen eingesetzt werden können. Die Arbeit wurde dazu in vier Teilbereiche eingeteilt: die Betrachtung der Beziehung zwischen Ontologien und Domänenmodellen; die Konzeption von Refactorings und Co-Refactorings für Ontologien und Domänenmodelle; die Implementierung eines Werkzeugs zur Umsetzung der Transformation und des Co-Refactorings; und die Evaluation anhand einer Beispiel-Ontologie. Im Folgenden werden zunächst die Erkenntnisse aus dem Gesamtkonzept und anschließend die Ergebnisse der vier Teilbereiche zusammengefasst.

9.1.1 Das Gesamtkonzept

Das Gesamtkonzept der Arbeit aus Kapitel 6 sieht vor, Softwaresysteme mit Ontologien zu modellieren. Mit diesem Vorgehen können Softwaresysteme detaillierter beschrieben werden, als es mit anderen Modellierungstechniken möglich ist. Darüber hinaus ist es mit Ontologien möglich, neue Konzepte in der Softwareentwicklung einzusetzen, wie semantische Suche und die Ableitung impliziten Wissens. Damit die Vorteile von Ontologien tatsächlich in Softwaresystemen zum Einsatz kommen können, muss eine definierte Beziehung zwischen beiden bestehen. Der Ansatz von Knublauch kann an dieser Stelle vielversprechend eingesetzt werden, um Ontologien in Softwaresysteme zu integrieren. Er sieht vor, dass Anwendungen oder zumindest Teile davon automatisch aus Ontologien über eine Codegenerierung abgeleitet werden. Der Ansatz ist somit ähnlich zur MDSD, verwendet aber Ontologien anstatt der Metamodelle und Modelle. In dieser Arbeit wird ein Konzept entwickelt, das auf dem Ansatz von Knublauch aufbaut, aber in einem Zwischenschritt auf Metamodelle und Modelle der MDSD zurückgreift, um die bereits vorhandenen Möglichkeiten der Codegenerierung und weitere, bereits existierende Werkzeuge dieses Bereichs einsetzen zu können.

Damit ontologiegetriebene Softwaresysteme beständig an sich ändernde Anforderungen angepasst werden können, wurde eine Evolutionsstrategie auf der Basis von Ontologie-Refactorings entwickelt. Diese Ontologie-Refactorings werden über Co-Refactorings auf die Metamodelle der MDSD übertragen. Ein wichtiger Aspekt der Refactorings ist, dass sie nicht nur die Struktur der Ontologie verändern, sondern auch die Datenmigration mit der Erhaltung der Individuen berücksichtigen. Die Kombination des Ansatzes zur Code-Generierung aus Ontologien von Knublauch und des Ansatzes zur Erhaltung der Individuen von Noy und Klein ist demnach ein aussichtsreiches Konzept, Softwaresysteme auf der Basis von Ontologien zu modellieren und außerdem flexibel weiterentwickeln zu können.

9.1.2 Beziehung zwischen Ontologien und Domänenmodellen

Die Konzeption und Umsetzung der OWL-Ecore-Transformation aus dem Abschnitt 6.2 hat gezeigt, dass es möglich ist, eine Abbildung zwischen Ontologien und den Metamodellen bzw. Modellen der MDSD herzustellen. Allerdings hat sich herausgestellt, dass eine Abbildung der Ontologien auf Modelle der MDSD problematisch ist, da es starke Unterschiede zwischen den

Prinzipien beider Technologieräume gibt, sodass viele Konzepte nicht oder nur indirekt abgebildet werden können. Mit den transformierbaren Ontologien wurde ein Konzept entwickelt, das exakt den Teil einer Ontologie beschreibt, der auf Modelle der MDSD abgebildet werden kann. Dieser Teil der Ontologie hat jedoch die gleiche Ausdrucksstärke wie die Modelle. Die aus dem Metamodell generierte Anwendung kann nur indirekt auf den nicht-transformierbaren Teil der Ontologie zugreifen, sodass die eigentlichen Vorteile von Ontologien gegenüber den Modellen nicht zur Modellierung des Systems zum Einsatz kommen können.

Der Nutzen vorhandener Techniken der MDSD für die Modellierung ontologiegetriebener Softwaresystemen sollte also nicht überbewertet werden. Zwar existieren bereits viele Werkzeuge, die Codegenerierung, Refactoring und Modellierung im MDSD-Umfeld umsetzen können, aber diese Funktionen lassen sich nur in sehr eingeschränktem Maße auf Ontologien übertragen. Weiterhin erhöht die Abbildung der Ontologien auf Modelle die Komplexität des Gesamtsystems, da der Entwickler nun drei Artefakte, die Ontologie, die Modelle und den Programmcode, berücksichtigen muss. Die Generierung des Programmcodes direkt aus der Ontologie könnte diese Komplexität reduzieren.

9.1.3 Konzeption von Refactorings und Co-Refactorings

Im Abschnitt 6.3 wurden Ontologie-Refactorings zur Weiterentwicklung von Ontologien definiert. Darüber hinaus wurde ein Katalog von Refactorings aufgestellt. Die Refactorings dieses Katalogs beschreiben Umstrukturierungen, welche sich auch auf Ecore-Modelle übertragen lassen. Dabei wurde deutlich, dass es stark vom aktuellen Verwendungszweck der Ontologie und der angewandten Prinzipien des Ontologie-Engineering abhängt, welche Refactorings tatsächlich zum Einsatz kommen können. Jedes Ontologie-Refactoring lässt sich dabei mit einem Schema darstellen, das zwar keine formale Definition aller Änderungen repräsentiert aber gut geeignet ist, unterschiedliche Refactorings sowie deren Auswirkungen zu beschreiben und zu vergleichen. Eine formale Definition der Änderungen würde bei vielen Refactorings, aufgrund der Komplexität der Umstrukturierungen, zu kompliziert werden.

Damit ontologiegetriebene Softwaresysteme stetig weiterentwickelt werden können, ist es nötig, die Auswirkungen eines Refactorings auf die gesamte Ontologie, also auch auf den Datenbestand der ABox, zu berücksichtigen. Dazu wurde das Prinzip der Erhaltung der Individuen in die Definition der Ontologie-Refactorings aufgenommen.

Mit den aufgestellten Refactorings ist eine schnelle und einfache Weiterentwicklung von Ontologien möglich. Jedoch unterscheidet sich der Refactoring-Ansatz dieser Arbeit von dem der objektorientierten Programmierung. In letzterem Fall wird die Struktur eines Programms verbessert, wobei die Funktionalität unverändert bleibt. Neue Funktionen werden in einer vom Refactoring vollständig getrennten Aktivität hinzugefügt. Bei den Ontologie-Refactorings wird hingegen der Ansatz der Evolution umgesetzt und mit jedem Refactoring auch die Semantik der Ontologie verändert. Die einzige Eigenschaft einer Ontologie, von der geprüft werden kann, ob sie unverändert geblieben ist, ist die Konsistenz. Konsistenzprüfungen der Ontologie stellen demnach das Gegenstück zu den Tests der objektorientierten Programmierung dar. Abgesehen von der Konsistenz muss bezüglich der Erhaltung der Individuen jedes Ontologie-Refactoring auf Basis seiner Definition sicherstellen, dass der Datenbestand nur in einer im Voraus definierten Weise geändert wird.

Co-Refactorings wurden in dieser Arbeit eingesetzt, um Änderungen an Ontologien auf Modelle der MDSD zu übertragen. Dazu wurde im Abschnitt 6.4 ein allgemeiner Co-Refactoring-Ansatz entwickelt, der nicht auf Ontologien und Ecore-Modelle beschränkt, sondern dazu geeignet ist, Änderungen synchron auf beliebigen Modellen durchzuführen. Dieser Ansatz bildet Änderungen eines Modells eindeutig auf Änderungen eines anderen Modells ab. Dies wird mit

einer Beziehung zwischen den Metamodellen und einer Abbildung alle Eingaben des Refactorings über einem Modell auf entsprechende Eingaben für das andere Modell erreicht.

9.1.4 Implementierung von OntoMore

Die Implementierung von OntoMore im Kapitel 7 mit den zwei Programmteilen zur OWL-Ecore-Transformation und dem Co-Refactoring sowie den mit Refactory umgesetzten Ontologie- und Ecore-Refactorings zeigt die Umsetzbarkeit des erarbeiteten Konzepts. Dabei hat sich herausgestellt, dass die Abbildung von Ontologien auf Metamodelle und Modelle nur programmatisch und nicht mit vorhandenen Modell-Transformations-Ansätzen sinnvoll umgesetzt werden kann, da sich die Transformation über mehrere Metaebenen erstreckt.

Die Ontologie-Refactorings wurden mit Refactory umgesetzt. Dies ist ein Werkzeug zur Definition und Ausführung von generischen Refactorings auf beliebigen Modellen. Allerdings konnten die Vorteile generischer Refactorings nicht genutzt werden. Weil Ontologie-Refactorings sehr komplex sind, können die generischen Refactorings in nur seltenen Fällen wiederverwendet werden, sodass der Aufwand ein Refactoring mit den DSLs von Refactory zu spezifizieren oftmals größer ist, als eine Implementierung mit Java.

Der CoRefactorer setzt den Co-Refactoring-Ansatz auf der Basis von Refactory um und erweitert dieses Werkzeug somit um die Funktion Refactorings unabhängig von der Art der Modelle auf mehreren Modellen synchron auszuführen. Die Co-Refactorings von Ontologien und Modellen blieben jedoch wegen technischer Einschränkungen auf die TBox beschränkt.

9.1.5 Evaluation anhand einer Beispiel-Ontologie

Im Kapitel 8 wurde mit der Beispiel-Ontologie FrOnto gezeigt, wie ein Domänenmodell mit einer Ontologie modelliert werden kann. Der Fokus der Arbeit lag allerdings auf der technischen Umsetzung des Ansatzes ontologiegetriebener Softwaresysteme und nicht auf dem Design einer Anwendungsdomäne mit Ontologien. Die Einsatzfähigkeit der implementierten Ontologie-Refactorings wurde dadurch bestätigt, dass diese tatsächlich während der Entwicklung der FrOnto zum Einsatz kommen konnten.

In der Evaluation wurden weiterhin die Grenzen des erarbeiteten Ansatzes aufgezeigt. Bezüglich der Umsetzung betrifft dies vor allem den Umstand, dass mit OWLText ein Werkzeug eingesetzt wird, dass Ontologien in der Manchester OWL Syntax darstellt. Somit ist es nicht konform zu anderen Ontologie-Werkzeugen und bietet wenig Übersichtlichkeit beim Bearbeiten großer Ontologien.

Bezüglich der Konzeption sind ontologiegetriebene Softwaresysteme darauf beschränkt, nur für wissensbasierte Systeme sinnvoll eingesetzt werden zu können. Zusammenfassend ist der Einsatz von Ontologien zur Modellierung von Softwaresystemen dann sinnvoll, wenn folgende drei Kriterien erfüllt sind.

- Daten liegen global verteilt vor und werden von vielen unabhängigen Teilnehmern erstellt und bearbeitet.
- Es soll ein gemeinsames Vokabular für alle Teilnehmer festgelegt werden, um den Datenaustausch zu standardisieren.
- Daten sollen von vielen unterschiedlichen und unabhängigen Quellen in einer gemeinsamen Plattform integriert werden.

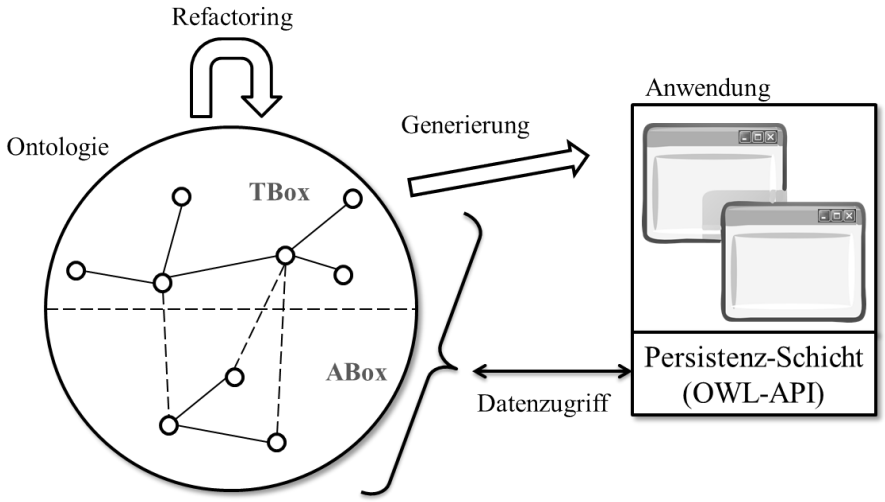

Abbildung 9.1: Vereinfachtes Konzept ontologiegetriebener Softwaresysteme

9.2 Ausblick

Zum Abschluss der Arbeit werden Empfehlungen dazu gegeben, wie die Ergebnisse dieser Arbeit weiterentwickelt werden können. Auch hier wird zuerst auf das Gesamtkonzept ontologiegetriebener Softwaresysteme eingegangen und anschließend auf einzelne Bereiche dieses Konzepts.

Vereinfachung des Konzepts ontologiegetriebener Softwaresysteme Das Konzept zur Entwicklung und Evolution ontologiegetriebener Softwaresysteme könnte stark vereinfacht werden, indem die Codegenerierung, wie bei Knublauchs Ansatz, direkt aus den Ontologien erfolgt. Wie in Abbildung 9.1 zu sehen, wird die Anwendung aus der TBox der Ontologie abgeleitet, greift aber auch wieder, über eine Persistenz-Schicht, auf die komplette Ontologie zu. Die Ontologie dient demnach nicht nur als Design-Artefakt, sondern auch als zentraler Bestandteil des Systems zur Datenspeicherung und Beschreibung der Systemstruktur.

In diesem vereinfachten Ansatz stellen Ontologien die einzigen Domänenmodelle dar. Somit sind eine Abbildung von Ontologie auf andere Domänenmodelle und auch das Co-Refactoring nicht mehr nötig. Refactorings werden nur auf der Ontologie durchgeführt. Die OWL-API bietet in diesem Zusammenhang uneingeschränkten Zugriff auf die Ontologie unabhängig von der eingesetzten Syntax. Somit kann die Datenmigration ohne Probleme umgesetzt werden. Die Verwirklichung von Evolutionsstrategien nach Stojanovic für OWL 2-Ontologien wäre ebenfalls denkbar. Ein weiterer Vorteil des Ansatzes besteht darin, dass auf die volle Ausdrucksstärke der Ontologien zurückgegriffen werden kann. Die Möglichkeiten der Ontologien im Design von Softwaresystemen werden also nur noch durch den Codegenerator begrenzt und nicht mehr durch die Modelle.

Codegenerierung Die eingesetzten Techniken werden im vereinfachten Ansatz auf Ontologien, Codegeneratoren und eine Ziel-Programmiersprache, wie Java, reduziert. Dies vereinfacht nicht nur die Erstellung solcher Systeme, sondern vermeidet auch eine redundante Präsentation der Anwendungsdomäne in Ontologie und Modellen. Mit dem Wegfall der MDSD aus dem Konzept können vorhandene Programme aus diesem Bereich nicht mehr eingesetzt werden. Jedoch wurden im Umfeld des Semantic Web bereits erste Anwendungen entwickelt, die

aus Ontologien Java-Code für die Anwendungsentwicklung generieren und somit eine mögliche Alternative darstellen. Die Bestrebungen in diese Richtung werden vom W3C im Projekt Tripresso[1] gebündelt. Beispiele für Techniken, die in diesem Projekt entwickelt werden sind Elmo für die Sesame-API und RDFReactor. Auch für Protégé sind bereits Codegeneratoren verfügbar [Knu04]. Darüber hinaus zeigen Kalyanpur et al. wie eine automatische Abbildung von Ontologien auf Java-Code umgesetzt werden kann [KPBP04].

Networked Ontologies Ein Punkt, der bislang im Konzept noch nicht berücksichtigt wurde, sind Networked Ontologies. Diesbezüglich muss noch untersucht werden, welche Auswirkungen Import-Beziehungen auf die Modellierung mit Ontologien haben. Die Verwendung von Import-Beziehungen ist von großer Bedeutung, da diese das eigentliche Wiederverwendungsprinzip von Ontologien darstellen. Somit ließen sich nicht nur vorhandene Ontologien, sondern auch die daraus abgeleiteten Anwendungsteile wiederverwenden. Nur über Import-Beziehungen können Ontologien ihre Stärke als standardisiertes Vokabular ausspielen.

Design mit Ontologien Der Fokus der vorliegenden Arbeit lag auf der technischen Umsetzung des Konzepts für ontologiegetriebene Softwaresysteme. Nicht näher behandelt wurde, wie eine Anwendungsdomäne mit einer Ontologie modelliert werden kann. Der Forschungsbereich des Ontologie-Engineering beschäftigt sich eingehend mit der Frage, wie Wissen in einer Ontologie am besten dargestellt werden kann. Ontology Design Patterns[2] und andere Techniken der Ontologie-Erstellung müssen noch dahin gehend untersucht werden, inwieweit sie für die Modellierung von Softwaresystemen geeignet sind und welche Auswirkungen sie auf die Codegenerierung haben.

Wie in Kapitel 8.4 erläutert, eignen sich ontologiegetriebene Softwaresysteme vorwiegend für wissensbasierte Systeme. Diesbezüglich gilt es zu untersuchen, inwieweit andere Systeme mit komplexer Geschäftslogik mit Ontologien modelliert werden können und für welche Konzepte eines Softwaresystems die Modellierung mit einer Ontologie sinnvoll ist. Daraus könnten konkrete Empfehlungen abgeleitet werden, welche Anwendungen sich für die Modellierung mit Ontologien eignen.

Weiterentwicklung der Ontologie-Refactorings Die in dieser Arbeit aufgestellten Ontologie-Refactorings können lediglich als erste Anhaltspunkte für komplexe Umstrukturierungen in Ontologien angesehen werden. Von diesem Punkt aus sind mehrere Möglichkeiten denkbar, Ontologie-Refactorings weiterzuentwickeln. Einerseits ist die Datenmigration selbst für elementare Refactorings in den bestehenden Ontologie-Werkzeugen meist nicht zufriedenstellend umgesetzt, sodass noch Bedarf daran besteht, die existierenden Erkenntnisse der Ontologie-Evolution auch praktisch umzusetzen. Andererseits könnten Refactorings auch im Bereich der Erstellung und Veränderung komplexer Ontologie-Konstrukte weiterentwickelt werden. Das Refactoring *Create Value Partition* ist ein Beispiel dafür. In diesen Bereich fallen ebenso die Ontology Design Patterns. Hier gilt es zu untersuchen, ob die Erstellung und Bearbeitung solcher Patterns durch Ontologie-Refactorings vereinfacht werden kann.

Neben der bisher vorwiegend eingesetzten Top-Down-Entwicklungsmethode, bei der eine Ontologie vom Gesamtkonzept hin zu den kleineren Details entwickelt wird, zeichnet sich ein Trend hin zur Bottom-Up-Evolution ab. Dabei werden Techniken des Data-Mining und des NLP eingesetzt, um Ontologien automatisch aus unstrukturierten Datenbeständen, wie Folksonomies, zu generieren. Als Beispiel für ein solches Vorgehen wurde im Abschnitt 5.2.4 das Plugin Evolva vorgestellt. An dieser Stelle gilt es zu untersuchen, wie Ontologie-Refactorings

[1] `http://semanticweb.org/wiki/Tripresso`
[2] `http://ontologydesignpatterns.org/`

eingesetzt werden können, um die automatische oder semi-automatische Klassifizierung der unstrukturierten Datenbestände zu verbessern.

Die Ontologie-Refactorings dieser Arbeit wurden basierend auf der Struktur der Ontologie aufgestellt. Vorgehensweisen der Ontologie-Erstellung wurden jedoch noch nicht berücksichtigt. Indem solche Vorgehensweisen und bewährte Methoden analysiert werden, könnten weitere Refactorings gefunden werden, die beispielsweise dem Benutzer oft ausgeführte Tätigkeiten abnehmen.

Co-Refactoring Für das vereinfachte Konzept ontologiegetriebener Softwaresysteme ist ein Co-Refactoring nicht mehr nötig. Jedoch gilt es zu untersuchen, in welchen anderen Bereichen das Co-Refactoring zum Einsatz kommen kann. In dieser Arbeit wurde ein allgemeiner Ansatz für Co-Refactoring entwickelt. Die Implementierung im Rahmen dieser Arbeit beschränkt sich jedoch auf OWL- und Ecore-Modelle. Diesbezüglich muss noch geklärt werden, wie eine Abbildung zwischen Input-Datensätzen für andere Metamodelle als OWL und Ecore in den Modulen des CoRefactorer umgesetzt werden kann.

Eine Frage für weitere Forschung ist außerdem, ob sich die Abbildungen zwischen den Input-Datensätzen für bestimmte Metamodelle verallgemeinern lassen, womit die Abbildungen für zwei Metamodelle auf zwei andere Metamodelle übertragen werden können. Die aktuelle Implementierung geht weiterhin davon aus, dass eine Abbildung für jede Richtung einzeln implementiert werden muss. An dieser Stelle ist zu untersuchen, für welche Anwendungsfälle es möglich ist, umkehrbare Abbildungen zu definieren, sodass die Abbildungen für eine Richtung aus den Abbildungen für die entgegengesetzte Richtung automatisch abgeleitet werden können.

Im Rahmen der Arbeit gab es für Co-Refactorings immer nur zwei miteinander in Beziehung stehende Modelle. Im Allgemeinen sieht der Co-Refactoring-Ansatz jedoch beliebig viele Modelle vor. Hier ist zu prüfen, welche Auswirkungen sich für das Co-Refactoring ergeben, wenn dieser Ansatz auf mehrere Modelle angewandt wird, die verkettet, baum- oder netzwerkartig verknüpft sind.

Darüber hinaus kann die Implementierung des CoRefactorer um zahlreiche Details erweitert werden. Dies betrifft zum Beispiel die Abbildung für Eingabewerte. Dafür muss eine generische Struktur für Refactoring-Eingabewerte gefunden werden. Eingabewerte für Refactorings können beliebig viele atomare Werte wie Zeichenketten, Zahlen oder Wahrheitswerte sein. Sie können außerdem eine Auswahl von Elementen des zu refaktorierenden Modells enthalten. All diese Werte müssen auf korrespondierende Werte des abhängigen Modells abgebildet werden. Außerdem kann der `ModelElementMapper` um die Funktion erweitert werden, Abbildungen für mehrere Input-Elemente zurückzugeben.

Schlussbemerkung Die Erkenntnisse dieser Arbeit zeigen, wie sich die Softwareentwicklung im Bereich des Semantic Web in den nächsten Jahren weiterentwickeln könnte. Ontologiegetriebene Softwaresysteme eröffnen neue Möglichkeiten, Anwendungen für das Semantic Web zu erstellen. Sie tragen somit dazu bei, die Kommunikation und den Datenaustausch zwischen Menschen zu verbessern.

Anhang

A Weitere Ontologie-Refactorings

Die Tabelle 10.1 stellt eine Erweiterung der Tabelle 6.3 dar. In ihr sind weitere Ontologie-Refactorings aufgelistet, die zur Weiterentwicklung von Ontologien eingesetzt werden können. Die hier aufgelisteten Refactorings betreffen jedoch auch Teile der Ontologie, die nicht auf Ecore-Modelle abgebildet werden können und somit nicht Bestandteil der transformierbaren Ontologien sind. Sofern die Ontologie jedoch unabhängig von einem Ecore-Modell entwickelt wird, können diese Refactorings uneingeschränkt eingesetzt werden.

Tabelle 10.1: Weitere Ontologie-Refactorings für OWL

Nr.	OWL-Refactoring	Kommentar
1	Add Covering Axiom	Fügt ein neues SubClassOf-Axiom, welches die Vereinigung aller Unterklassen enthält, ein.
2	Add Closure Axiom	Fügt eine neue only-PropertyRestriction, welche die Vereinigung der Ziele aller bestehenden PropertyRestrictions enthält, ein.
3	Create Value Partition	Erstellt eine Value Partition inklusive der Unterklassen-Beziehungen, einem Covering-Axiom und einer funktionalen ObjectProperty.
4	Add/Delete Member of Value Partition	Entfernt eine Unterklasse einer Value Partition oder fügt eine neue hinzu.
5	Assign Individuals to Class	Fügt einer Menge von Individuen ein Types-Axiom zu einer ausgewählten Klasse hinzu.
6	Convert Class Expressions to Explicit Class	Wandelt eine class expression in eine defined class um, die auch an anderer Stelle für dieselbe class expression verwendet werden kann.
7	Normalize Class Expression	Konvertiert die class expression in Klausel-Normalform.

B Auswirkungen von Ontologie-Refactorings bezüglich der Datenmigration

Die Erhaltung der Individuen ist ein wichtiges Kriterium zur Bewertung von Ontologie-Refactorings. Es sagt aus, ob für ein Refactoring eine Datenmigration notwendig und ob sie möglich ist. Die Tabelle 10.2 enthält in der zweiten Spalte noch einmal die Ontologie-Refactorings aus Tabelle 6.3, in der dritten Spalte wird hier jedoch die Auswirkung auf den Datenbestand angegeben und die vierte Spalte enthält einen Kommentar zur Erklärung. Dazu wird die Klassifizierung aus Kapitel 6.3 verwendet. Somit bedeutet „+“, dass das Refactoring keine Auswirkungen auf die Individuen hat, „−“, dass es zwangsläufig zum Verlust von Instanz-Daten kommt, wenn die Ontologie konsistent bleiben soll und „o“, dass eine zusätzliche Umstrukturierung die Konsistenz und die Instanz-Daten erhalten kann.

Tabelle 10.2: Auswirkungen der Ontologie-Refactoring auf den Datenbestand

Nr.	OWL-Refactoring	Eff	Kommentar
1	*Duplicate Class* Eine neue Klasse wird erstellt und alle Axiome einer vorhandenen Klasse werden übernommen.	+	Es gibt keine Auswirkungen auf vorhandene Individuen.
2	*Convert DataProperty to ObjectProperty* In OWL wird eine DataProperty in eine ObjectProperty umgewandelt. Die Domain bleibt erhalten. Als Range wird eine vorhandene oder neu erstellte Klasse angegeben.	o	Jede DataProperty-Assertion muss gegen eine ObjectProperty-Assertion auf ein neues Individuum ausgetauscht werden.
3	*Move Property* Ersetzt die Domain einer Property durch eine andere Klasse.	–	Sofern die neue Domain disjunkt zur alten ist, müssen Property-Assertions dieser Property gelöscht werden.
4	*Pull Up Property* Die Domain einer Property wird durch eine ihrer Superklassen ersetzt. Spezialfall von Move Property.	+	Alle Individuen der Domain der Property sind aufgrund der Unterklassen-Beziehung nach dem Refactoring immer noch in der Domain.
5	*Push Down Property* Die Domain einer DataProperty oder ObjectProperty wird durch eine ihrer Unterklassen ersetzt. Invers zu Pull Up Property.	–	Der Definitionsbereich der Property wird eingeschränkt. Alle Individuen, die nicht mehr zur Domain gehören, müssen die Property-Assertion aufgeben.
6	*Extract Class* Axiome einer Klasse werden in eine neu erstellte Klasse verschoben.	o	Die Assertions der Individuen können ebenfalls zu Individuen der neuen Klasse verschoben werden.
7	*Extract Superclass* Spezialfall von Extract Class. Die neu erstellte Klasse ist eine Superklasse der Klasse, von der die Axiome extrahiert wurden.	o	Wie bei Extract class.
8	*Extract Subclass* Spezialfall von Extract Class. Die neu erstellte Klasse ist eine Unterklasse der Klasse, von der die Axiome extrahiert wurden.	o	Wie bei Extract class.
9	*Inline Class* Alle Axiome einer Klasse werden in eine andere Klasse verschoben. Anschließend wird die erstere Klasse gelöscht. Inverse zu Extract Class.	o	Die Assertions der verschobenen Axiome müssen zu Individuen der Ziel-Klasse verschoben werden.
10	*Collapse Hierarchy* Die Axiome aller Unterklassen einer ausgewählten Klasse werden in diese verschoben. Die Unterklassen werden gelöscht. Entspricht der Ausführung von Inline Class für alle Unterklassen.	o	Die Anweisungen für Inline Class müssen für jede Unterklasse durchgeführt werden.
11	*Introduce Inverse Property* Es wird eine neue ObjectProperty angelegt, deren Domain dem Range und deren Range der Domain einer ausgewählten ObjectProperty entspricht. Beiden Properties wird ein InverseOf-Axiom zur jeweils anderen Property hinzugefügt.	+	Es gibt keine Auswirkungen auf vorhandene Individuen.
12	*Remove Inverse Property* Die Property, die über ein InverseOf-Axiom mit einer ausgewählten Property in Beziehung steht, wird gelöscht. Invers zu Introduce Inverse Property.	–	Alle Verwendungen der inversen Property müssen gelöscht werden.

Tabelle 10.3: Auswirkungen der Ontologie-Refactoring auf den Datenbestand (Forts.)

Nr.	OWL-Refactoring	Eff	Kommentar
13	*Replace SubClassOf with Property* Entfernt das SubClassOf-Axiom zu einer Superklasse und erstellt eine neue PropertyRestriction einer vorhandenen oder neu erstellten ObjectProperty zu dieser Superklasse.	o	Für alle Axiome der Superklasse müssen jeweils neue Individuen erzeugt werden, die von den bestehenden Individuen der Unterklasse referenziert werden.
14	*Replace Property with SubClassOf* Entfernt eine PropertyRestriction einer Klasse und fügt dafür ein SubClassOf-Axiom zum Range der ObjectProperty ein. Invers zu Replace SubClassOf with Property.	o	Die Assertions der über die Property referenzierten Individuen müssen in die Individuen der Unterklasse integriert werden.

C Die MiniFrOnto vor und nach dem Refactoring

Das folgende Listing zeigt die MiniFrOnto in der Manchester OWL Syntax vor dem Refactoring. Die MiniFrOnto stellt einen Ausschnitt der Beispiel-Ontologie FrOnto dar, der auf die Konzepte reduziert wurde, die von den implementierten Refactorings verändert werden. Auf der nächsten Seite ist dieselbe Ontologie nach dem Refactoring dargestellt.

Listing 10.1: MiniFrOnto vor dem Refactoring

```
Prefix: : <http://www.buschmais.com/mfronto.owl#>

Ontology: <http://www.buschmais.com/mfronto.owl>

Class: Freiberufler

Class: Unternehmen

Class: Technik

Class: Programmiersprache
    SubClassOf: Technik

Class: Framework
    SubClassOf: Technik

Class: Zeitraum

Class: Verfuegbarkeit
    SubClassOf: Zeitraum

Class: Profil

DisjointClasses: Freiberufler, Unternehmen, Technik, Zeitraum

ObjectProperty: hatProfil
  Domain: Freiberufler
    Range: Profil

DataProperty: hatAdresse
  Domain: Freiberufler
  Range: xsd:string

DataProperty: hatAnfangsdatum
    Domain: Verfuegbarkeit
    Range: xsd:dateTime

DataProperty: hatEnddatum
    Domain: Verfuegbarkeit
    Range: xsd:dateTime
```

Listing 10.2: MiniFrOnto nach dem Refactoring

```
Prefix: : <http://www.buschmais.com/mfronto.owl#>

Ontology: <http://www.buschmais.com/mfronto.owl>

Class: Person

Class: Freelancer
  SubClassOf: Person

Class: Unternehmen

Class: Technik

Class: Programmiersprache
    SubClassOf: Technik

Class: Datenbank
  SubClassOf: Technik

Class: Framework
    SubClassOf: Technik

Class: Zeitraum

Class: Verfuegbarkeit
    SubClassOf: Zeitraum

Class: Profil

Class: Adresse

DisjointClasses: Freelancer, Unternehmen, Technik, Zeitraum

ObjectProperty: hatProfil
  Domain: Freelancer
    Range: Profil
  InverseOf: beschreibtFreelancer

ObjectProperty: beschreibtFreelancer
  Domain: Profil
  Range: Freelancer
  InverseOf: hatProfil

ObjectProperty: hatAdresse
  Domain: Person
  Range: Adresse

DataProperty: hatAnfangsdatum
    Domain: Zeitraum
    Range: xsd:dateTime

DataProperty: hatEnddatum
    Domain: Zeitraum
    Range: xsd:dateTime
```

D Refactoring mit der OWL-API

Das Listing 10.3 zeigt die vollständige Implementierung von *Convert DataProperty to ObjectProperty* mit Java und der OWL-API. Dabei wird das Template-Method-Pattern [GHJV94] verwendet um Funktionalitäten, die alle Refactorings betreffen in die gemeinsame Superklasse *Refactoring* auszulagern. Dazu zählen das Entgegennehmen der Eingabewerte und das Ausführen der Änderungen. Änderungen an der Ontologie werden von der OWL-API über das Command-Pattern verwirklicht. Dazu wird für jedes hinzuzufügende Axiom ein ChangeAxiom erstellt, das die eigentliche Änderung repräsentiert. Alle Änderungen werden erst am Ende des Refactorings gemeinsam ausgeführt. Die OWL-API bestimmt bei einer Löschoperation automatisch alle abhängigen Axiome, die ebenfalls gelöscht werden müssen, um die Konsistenz zu erhalten. Dies geschieht wie in Zeile 41 zu sehen über das Visitor-Pattern. Der *OWLEntityRemover* spielt dabei die Rolle des Visitors der alle zu löschenden Axiome bestimmt.

Listing 10.3: Vollständige Implementierung von *Convert DataProperty to ObjectProperty* mit Java und OWL-API

```
1  public class ConvertDataPropertyToObjectProperty extends Refactoring {
2
3      private OWLObjectProperty objectProperty;
4      private OWLClass rangeClass;
5
6      @Override
7      protected void executeRefactoringSteps() {
8          // create ObjectProperty and set range
9          objectProperty = factory.getOWLObjectProperty(inputElement.getIRI());
10         rangeClass = factory.getOWLClass(":" + inputValue, pm);
11         OWLAxiom rangeAxiom = factory.getOWLObjectPropertyRangeAxiom(objectProperty
               , rangeClass);
12         axiomsToAdd.add(rangeAxiom);
13
14         // transfer domains
15         Set<OWLClassExpression> domains = inputElement.getDomains(ontology);
16         for (OWLClassExpression domain : domains) {
17             OWLAxiom domainAxiom = factory.getOWLObjectPropertyDomainAxiom(
                   objectProperty, domain);
18             axiomsToAdd.add(domainAxiom);
19         }
20
21         // transfer annotations
22         Set<OWLAnnotation> annotations = inputElement.getAnnotations(ontology);
23         for (OWLAnnotation annoation : annotations) {
24             OWLAxiom annotationAxiom = factory.getOWLAnnotationAssertionAxiom(
25                     objectProperty.getIRI(), annoation);
26             axiomsToAdd.add(annotationAxiom);
27         }
28
29         // transfer characteristics
30         Set<OWLAxiom> referencingAxioms = inputElement.getReferencingAxioms(
               ontology);
31         for (OWLAxiom owlAxiom : referencingAxioms) {
32             if (owlAxiom instanceof OWLFunctionalDataPropertyAxiom) {
33                 OWLFunctionalObjectPropertyAxiom functionalAxiom = factory
34                         .getOWLFunctionalObjectPropertyAxiom(objectProperty);
35                 axiomsToAdd.add(functionalAxiom);
36             }
37         }
38
39         // remove DataProperty
40         OWLEntityRemover remover = new OWLEntityRemover(manager, Collections.
               singleton(ontology));
41         inputElement.accept(remover);
42         changeAxioms.addAll(remover.getChanges());
43     }
44
```

```
45
46      @Override
47      protected void dataMigration() {
48          Set<OWLAxiom> referencingAxioms = inputElement.getReferencingAxioms(
                ontology);
49          int indivCount = 0;
50          for (OWLAxiom owlAxiom : referencingAxioms) {
51              if (owlAxiom instanceof OWLDataPropertyAssertionAxiom) {
52                  OWLDataPropertyAssertionAxiom dataPropertyAssertion = (
                        OWLDataPropertyAssertionAxiom) owlAxiom;
53                  OWLLiteral literal = dataPropertyAssertion.getObject();
54                  OWLIndividual subject = dataPropertyAssertion.getSubject();
55
56                  // create new individual and set ObjectPropertyAssertion
57                  OWLNamedIndividual object = factory.getOWLNamedIndividual(":
                        GenIndiv"
58                          + indivCount, pm);
59                  OWLObjectPropertyAssertionAxiom objectPropertyAssertion = factory
60                          .getOWLObjectPropertyAssertionAxiom(objectProperty, subject
                                , object);
61                  axiomsToAdd.add(objectPropertyAssertion);
62
63                  // assign new individual to range class
64                  OWLClassAssertionAxiom classAssertion = factory.
                        getOWLClassAssertionAxiom(
65                          rangeClass, object);
66                  axiomsToAdd.add(classAssertion);
67
68                  // put former DataProperty value in annotation
69                  OWLAnnotationProperty annotationProperty = factory.
                        getOWLAnnotationProperty(IRI
70                          .create("http://www.w3.org/2000/01/rdf-schema#label"));
71                  OWLAxiom annotationAxiom = factory.getOWLAnnotationAssertionAxiom(
72                          annotationProperty, object.getIRI(), literal);
73                  axiomsToAdd.add(annotationAxiom);
74                  indivCount++;
75              }
76          }
77      }
78  }
```

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

ABox	Assertion Box
API	Application Programming Interface
ATL	ATL Transformation Language
COPE	Coupled Evolution of Metamodels and Models
CWA	Closed-World Assumption
DC	Dublin Core Ontology
DOLCE	Descriptive Ontology for Linguistic and Cognitive Engineering
DSL	Domain Specific Language
EBNF	Erweiterte Backus-Naur-Form
EMF	Eclipse Modelling Framework
EMOF	Essential Meta Object Facility
FOAF	Friend of a Friend Ontology
FrOnto	Freelancer-Ontology
FZI	Forschungszentrum Informatik
GCI	General Concept Inclusions
GPL	General Purpose Language
IRI	Internationalized Resource Identifier
KAON	Karlsruhe Ontology and Semantic Web Framework
KIT	Karlsruhe Institute of Technology
LOD	Linking Open Data
MDA	Model-driven Architecture
MDSD	Model-driven Software Development
MIS	Minimal Inconsistent Subset
MOF	Meta Object Facility
MOST	Marrying Ontology and Software Technology
MUPS	Minimal Unsatisfiability Preserving Subset

NLP	Natural Language Processing
ODM	Ontology Definition Metamodel
ODSD	Ontology-driven Software Development
OMG	Object Management Group
OntoMoPP	Ontology Model Parser and Printer
OntoMore	Ontologie-Modell-Refactoring
OWA	Open-World Assumption
OWL	Web Ontology Language
QVT	Query/View/Transformation
RaDON	Repair and Diagnosis of Networked Ontologies
RDF	Resource Description Framework
RDFS	RDF Schema
SEALS	Semantic Evaluation At Large Scale
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
SWRL	Semantic Web Rule Language
TBox	Terminology Box
TDD	Test-Driven Development
TGG	Tripel Graph Grammar
TwoUse	Transforming and Weaving Ontologies and UML in Software Engineering
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Literaturverzeichnis

[AB08] **Adida, Ben** und **Mark Birbeck**: *RDFa Primer – Bridging the Human and Data Webs*. `http://www.w3.org/TR/xhtml-rdfa-primer/`, Oktober 2008. W3C Working Group Note.

[BBL08] **Beckett, David** und **Tim Berners-Lee**: *Turtle – Terse RDF Triple Language*. `http://www.w3.org/TeamSubmission/turtle/`, Januar 2008. W3C Team Submission.

[BCM+10] **Baader, Franz**, **Diego Calvanese**, **Deborah L. McGuinness**, **Daniele Nardi** und **Peter F. Patel-Schneider** (Herausgeber): *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2 Auflage, Mai 2010.

[Bec03] **Beck, Kent**: *Test-Driven Development by Example*. Addison Wesley, 2003.

[Bec04] **Beckett, Dave**: *RDF/XML Syntax Specification (Revised)*. `http://www.w3.org/TR/rdf-syntax-grammar/`, Februar 2004. W3C Recommendation.

[BG04] **Brickley, Dan** und **R.V. Guha**: *RDF Vocabulary Description Language 1.0: RDF Schema*. `http://www.w3.org/TR/rdf-schema/`, Februar 2004. W3C Recommendation.

[BHBL09] **Bizer, Christian**, **Tom Heath** und **Tim Berners-Lee**: *Linked Data – The Story So Far*. Int. J. Semantic Web Inf. Syst, 5(3):1–22, 2009.

[BHS05] **Baader, Franz**, **Ian Horrocks** und **Ulrike Sattler**: *Description Logics as Ontology Languages for the Semantic Web*. In: **Hutter, Dieter** und **Werner Stephan** (Herausgeber): *Mechanizing Mathematical Reasoning: Essays in Honor of Jörg Siekmann on the Occasion of His 60th Birthday*, Nummer 2605 in *Lecture Notes in Artificial Intelligence*, Seiten 228–248. Springer, 2005.

[BL06] **Berners-Lee, Tim**: *Linked Data – Design Issues*. `http://www.w3.org/DesignIssues/LinkedData.html`, Juli 2006. Besucht am 13. März 2011.

[BLFM05] **Berners-Lee, Tim**, **R. Fielding** und **L. Masinter**: *RFC 3986 – Uniform Resource Identifier (URI): Generic Syntax*. `http://tools.ietf.org/html/rfc3986`, Januar 2005. Besucht am 11. März 2011.

[BLHL01] **Berners-Lee, Tim**, **James Hendler** und **Ora Lassila**: *The Semantic Web*. Scientific American, 284(5):34–43, 2001.

[BPSM+08] **Bray, Tim**, **Jean Paoli**, **C. M. Sperberg-McQueen**, **Eve Maler** und **François Yergeau**: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. `http://www.w3.org/TR/2008/REC-xml-20081126/`, November 2008. W3C Recommendation.

[Bre08] **Breed, Aditi**: *Purpose of using Ontologies in Software Engineering*. `http://people.cis.ksu.edu/~abreed/CIS890/Sft_Engineering.pdf`, Februar 2008. Kansas State University.

[Bro95] **Brooks, Jr., Frederick P.**: *The Mythical Man-Month*. Addison Wesley, 1995.

[Cas10] **Castro, Raul Garcia**: *Results of the first evaluation of ontology engineering tools*, November 2010. SEALS-Project Deliverable D10.3.

[CJ10] **Cyganiak, Richard** und **Anja Jentzsch**: *Linking Open Data Cloud Diagram*. `http://lod-cloud.net/`, September 2010. Besucht am 19. April 2011.

[Con10] **Conner, Margery**: *Sensors empower the Internet of Things*. EDN, 10:32–38, Mai 2010.

[CP99] **Cranefield, Stephen** und **Martin K. Purvis**: *UML as an Ontology Modelling Language*. In: *Intelligent Information Integration*, Band 23 der Reihe *CEUR Workshop Proceedings*, 1999.

[DCtTdK11] **Dentler, Kathrin**, **Ronald Cornet**, **Annette ten Teije** und **Nicolette de Keizer**: *Comparison of Reasoners for large Ontologies in the OWL 2 EL Profile*. Semantic Web Journal, 1, 2011. Survey Article in Review.

[Den04] **Denny, Michael**: *Ontology Tools Survey, Revisited*. `http://www.xml.com/pub/a/2004/07/14/onto.html`, Juli 2004. Besucht am 30. März 2011.

[DS05] **Duerst, M.** und **M. Suignard**: *RFC 3987 – Internationalized Resource Identifiers (IRIs)*. `http://tools.ietf.org/html/rfc3987`, Januar 2005. Besucht am 11. März 2011.

[Eys08] **Eysholdt, Moritz**: *Towards EMF Ecore based Meta Model Evolution and Model Co-Evolution*. In: *Proceedings of the Second Workshop on MDSD Today*, Seiten 51–59. Shaker Verlag, Oktober 2008.

[Eys09] **Eysholdt, Moritz**: *EMF Ecore Based Meta Model Evolution and Model Co-Evolution*. Master's Thesis, University of Oldenburg, Germany, April 2009.

[FBB+99] **Fowler, Martin**, **Kent Beck**, **John Brant**, **William F. Opdyke** und **Don Roberts**: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[GDD10] **Gaševic, Dragan**, **Dragan Djuric** und **Vladan Devedžic**: *Model Driven Engineering and Ontology Development*. Springer, November 2010.

[GHJV94] **Gamma, Erich**, **Richard Helm**, **Ralph E. Johnson** und **John Vlissides**: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 1994.

[GHWK07] **Grau, Bernardo Cuenca**, **Christian Halaschek-Wiener** und **Yevgeny Kazakov**: *History matters: incremental ontology reasoning using modules*. In: *Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference*, ISWC'07/ASWC'07, Seiten 183–196, Berlin, Heidelberg, 2007. Springer-Verlag.

[GO10] **Glimm, Birte** und **Chimezie Ogbuji**: *SPARQL 1.1 Entailment Regimes*. `http://www.w3.org/TR/sparql11-entailment/`, Oktober 2010. W3C Working Draft.

[GPS10] **Gröner, Gerd**, **Fernando Silva Parreiras** und **Steffen Staab**: *Semantic Recognition of Ontology Refactoring*. In: *International Semantic Web Conference (1)*, Seiten 273–288, 2010.

[Gru93] **Gruber, Thomas R.**: *A Translation Approach to Portable Ontology Specifications*. Knowl. Acquis., 5(2):199–220, 1993.

[Gua98] **Guarino, Nicola**: *Formal Ontology and Information Systems*. In: **Guarino, Nicola** (Herausgeber): *FOIS'98*, Seiten 3–15, Amsterdam, Juni 1998. IOS Press.

[Her10] **Herrmannsdörfer, Markus**: *COPE - A Workbench for the Coupled Evolution of Metamodels and Models*. In: *SLE*, Seiten 286–295, 2010.

[HJK+09] **Heidenreich, Florian**, **Jendrik Johannes**, **Sven Karol**, **Mirko Seifert** und **Christian Wende**: *Derivation and Refinement of Textual Syntax for Models*. In: *Fifth European Conference on Model-Driven Architecture Foundations and Applications, ECMDA-FA 2009, 23 - 26 June 2009, Enschede, The Netherlands*, 2009.

[HKK09] **Hillner, Stanley**, **Heiko Kern** und **Stefan Kühne**: *Berechnung von Modelldifferenzen als Basis für die Evolution von Prozessmodellen*. In: **Münch, Jürgen** und **Peter Liggesmeyer** (Herausgeber): *Workshop Modellgetriebene Softwarearchitektur – Evolution, Integration und Migration (MSEIM), Software Engineering 2009*, Nummer P-150 in *LNI*, Seiten 375–382, Bonn, 2009. Bonner Köllen Verlag.

[HKRS08] **Hitzler, Pascal**, **Markus Krötzsch**, **Sebastian Rudolph** und **York Sure**: *Semantic Web*. Springer, 2008.

[Hor08] **Horrocks, Ian**: *Ontologies and the Semantic Web*. Communications of the ACM, 51(12):58–67, Dezember 2008.

[Hor11] **Horridge, Matthew**: *A Practical Guide To Building OWL Ontologies Using The Protégé-OWL Plugin and CO-ODE Tools*. The University Of Manchester, 1.3:0–108, 2011.

[HPS09] **Horridge, Matthew** und **Peter F. Patel-Schneider**: *OWL 2 Web Ontology Language Manchester Syntax*. `http://www.w3.org/TR/2009/NOTE-owl2-manchester-syntax-20091027/`, Oktober 2009. W3C Working Group Note.

[HPSB+04] **Horrocks, Ian**, **Peter F. Patel-Schneider**, **Harold Boley**, **Said Tabet**, **Benjamin Grosof** und **Mike Dean**: *SWRL: A Semantic Web Rule Language Combining OWL and RuleML*. `http://www.w3.org/Submission/SWRL/`, Mai 2004. W3C Member Submission.

[HS05] **Haase, Peter** und **Ljiljana Stojanovic**: *Consistent Evolution of OWL Ontologies*. In: **Gómez-Pérez, Asunción** und **Jérôme Euzenat** (Herausgeber): *The Semantic Web: Research and Applications*, Band 3532 der Reihe *Lecture Notes in Computer Science*, Seiten 91–133. Springer Berlin / Heidelberg, 2005.

[Hub08] **Huber, Philipp**: *The Model Transformation Language Jungle – An Evaluation and Extension of Existing Approaches*. Diplomarbeit, Technische Universität Wien, Mai 2008.

[JAP09] **Javed, Muhammad**, **Yalemisew M. Abgaz** und **Claus Pahl**: *A Pattern-Based Framework of Change Operators for Ontology Evolution*. In: *Proceedings of the Confederated International Workshops and Posters on On the Move to Meaningful Internet Systems: ADI, CAMS, EI2N, ISDE, IWSSA, MONET, OnToContent, ODIS, ORM, OTM Academy, SWWS, SEMELS, Beyond SAWSDL, and COMBEK 2009*, OTM '09, Seiten 544–553, Berlin, Heidelberg, 2009. Springer-Verlag.

[JHQ+09] **Ji, Qiu**, **Peter Haase**, **Guilin Qi**, **Pascal Hitzler** und **Steffen Stadtmüller**: *RaDON – Repair and Diagnosis in Ontology Networks*. In: *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC 2009 Heraklion, Seiten 863–867, Berlin, Heidelberg, 2009. Springer-Verlag.

[Köh09] **Köhler, Christiane**: *Erarbeitung einer Konzeption zur Verbesserung des Beziehungsmanagements zwischen Freelancern und IT-Unternehmen*. Diplomarbeit, Berufsakademie Sachsen Staatliche Studienakademie Dresden, 2009.

[KK02] **Karagiannis, Dimitris** und **Harald Kühn**: *Metamodelling Platforms*. In: *EC-WEB '02: Proceedings of the Third International Conference on E-Commerce and Web Technologies*. Springer-Verlag, September 2002.

[KM01] **Koivunen, Marja-Riitta** und **Eric Miller**: *W3C Semantic Web Activity*. Technischer Bericht, W3C, http://www.w3.org/2001/12/semweb-fin/w3csw, 2001.

[Knu04] **Knublauch, Holger**: *Ontology-Driven Software Development in the Context of the Semantic Web: An Example Scenario with Protégé/OWL*. In: **Frankel, David S.**, **Elisa F. Kendall** und **Deborah L. McGuinness** (Herausgeber): *1st International Workshop on the Model-Driven Semantic Web (MDSW2004)*, 2004.

[KPBP04] **Kalyanpur, Aditya**, **Daniel Jiménez Pastor**, **Steve Battle** und **Julian A. Padget**: *Automatic Mapping of OWL Ontologies into Java*. In: *SEKE*, Seiten 98–103, 2004.

[KW07] **Kindler, Ekkart** und **Robert Wagner**: *Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios*. Technischer Bericht, Software Engineering Group, Department of Computer Science, University of Paderborn, 2007.

[Lee04] **Lee, Ryan**: *Scalability Report on Triple Store Applications*. Technischer Bericht, SIMILE Project, 2004.

[Leh96] **Lehman, Meir M.**: *Laws of Software Evolution Revisited*. In: *Lecture Notes in Computer Science*, Seiten 108–124. Springer, 1996.

[Lia05] **Liang, Yaozhong**: *Ontology Versioning and Evolution for Semantic Web-Based Applications*. Technischer Bericht, University of Southampton, Juli 2005.

[MA08] **Miksa, Krzysztof** und **Uwe Assman**: *MOST Project Factsheet*. `https://www.most-project.eu/admin/xinha/plugins/ExtendedFileManager/images/MOST_factsheet.pdf`, Februar 2008. Besucht am 13. März 2011.

[Mah09] **Mahler, Dirk**: *Fließender Übergang – Refactoring von Datenbankschemas mit Liquibase*. DatabasePro, 2:92–96, 2009.

[MAP+08] **Moser, Raimund**, **Pekka Abrahamsson**, **Witold Pedrycz**, **Alberto Sillitti** und **Giancarlo Succi**: *A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team*. In: **Meyer, Bertrand**, **Jerzy R. Nawrocki** und **Bartosz Walter** (Herausgeber): *LNCS 5082*, Kapitel Balancing Agility and Formalism in Software Engineering, Seiten 252–266. Springer-Verlag, Berlin, Heidelberg, 2008.

[Mar09] **Martin, Robert C.**: *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2009.

[May09] **Maynard, Diana**: *Bottom-up evolution of networked ontologies from metadata*. `http://www.neon-project.org/nw/Deliverables`, Dezember 2009. NeOn-Project Deliverable D1.5.4.

[MCG05] **Mens, Tom**, **Krzysztof Czarnecki** und **Pieter Van Gorp**: *A Taxonomy of Model Transformations*. In: *Proc. Dagstuhl Seminar 04101 on Language Engineering for Model-Drieven Software Development*, 2005.

[MGH+09] **Motik, Boris**, **Bernardo Cuenca Grau**, **Ian Horrocks**, **Zhe Wu**, **Achille Fokoue** und **Carsten Lutz**: *OWL 2 Web Ontology Language Profiles*. `http://www.w3.org/TR/owl2-profiles/`, Oktober 2009. W3C Recommendation.

[Mil71] **Mills, Harlan D.**: *Top-down Programming in Large Systems*. In: **Rustin, Randall** (Herausgeber): *Debugging Techniques in Large Systems*, Seiten 41–55. Prentice-Hall, Englewood Cliffs, N. J., 1971.

[MM04] **Manola, Frank** und **Eric Miller**: *RDF Primer*. `http://www.w3.org/TR/rdf-primer/`, Februar 2004. W3C Recommendation.

[MMBJ09] **Moha, Naouel**, **Vincent Mahé**, **Olivier Barais** und **Jean-Marc Jézéquel**: *Generic Model Refactorings*. In: *Model Driven Engineering Languages and Systems*, Band 5795/2009 der Reihe *Lecture Notes in Computer Science*, Seiten 628–643. Springer, 2009.

[MPPS09] **Motik, Boris**, **Bijan Parsia** und **Peter F. Patel-Schneider**: *OWL 2 Web Ontology Language XML Serialization*. `http://www.w3.org/TR/2009/REC-owl2-xml-serialization-20091027/`, Oktober 2009. W3C Recommendation.

[MPSP09] **Motik, Boris**, **Peter F. Patel-Schneider** und **Bijan Parsia**: *OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax*. `http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/`, Oktober 2009. W3C Recommendation.

[MvH04] **McGuinness, Deborah L.** und **Frank van Harmelen**: *OWL Web Ontology Language Overview*. `http://www.w3.org/TR/owl-features/`, Februar 2004. W3C Recommendation.

[NK04] **Noy, Natalya Fridman** und **Michel Klein**: *Ontology Evolution: Not the Same as Schema Evolution*. Knowledge and Information Systems, 6(4):428–440, Juli 2004.

[NM01] **Noy, Natalya Fridman** und **Deborah L. McGuinness**: *Ontology Development 101: A Guide to Creating Your First Ontology*. Technischer Bericht, Stanford Knowledge Systems Laboratory and Stanford Medical Informatics, März 2001.

[NND+09] **Nyulas, Csongor**, **Natalya Fridman Noy**, **Michael Dorf**, **Nicholas Griffith** und **Mark A. Musen**: *Ontology-Driven Software: What We Learned From Using Ontologies As Infrastructure For Software*. In: *5th International Workshop on Semantic Web Enabled Software Engineering (SWESE) at ISWC 2009*, Chantilly, VA, 2009.

[NP10] **NeOn-Project**: *NeOn Toolkit Doku*. `http://neon-toolkit.org/w/images/Doku_2.3.pdf`, Januar 2010. Documentation of NTK 2.3.

[NTNM10] **Noy, Natalya Fridman**, **Tania Tudorache**, **Csongor Nyulas** und **Mark Musen**: *The ontology life cycle: Integrated tools for editing, publishing, peer review, and evolution of ontologies*. Technischer Bericht, Stanford Center for Biomedical Informatics Research, Stanford University, Stanford, CA, November 2010. `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3041389/`.

[NV04] **Navigli, Roberto** und **Paola Velardi**: *Learning Domain Ontologies from Document Warehouses and Dedicated Web Sites*. Comput. Linguist., 30:151–179, Juni 2004.

[OJ90] **Opdyke, William F.** und **Ralph E. Johnson**: *Refactoring: An aid in designing application frameworks and evolving object-oriented systems*. In: *SOOPPA '90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*, September 1990.

[OMG06] **OMG**: *Meta Object Facility (MOF) Core Specification*. Object Management Group, Januar 2006.

[OMG08] **OMG**: *Meta Object Facility (MOF) 2.0 Query/View/Transformation, V1.0*. `http://www.omg.org/spec/QVT/1.0/`, April 2008. OMG Specifications.

[OMG09] **OMG**: *Ontology Definition Metamodel (ODM)*. `http://www.omg.org/spec/ODM/1.0/`, Mai 2009. OMG Specifications.

[Opd92] **Opdyke, William F.**: *Refactoring object-oriented frameworks*. Doktorarbeit, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

[Par76] **Parnas, David L.**: *On the Design and Development of Program Families*. IEEE Transactions on Software Engineering, 2:1–9, 1976.

[PH10] **Palma, Paul** und **Peter Haase**: *Ontology Evolution*. `http://www.neon-project.org/nw/NeOn_Book`, Januar 2010. NeOn Book.

[PS08] **Prud'hommeaux, Eric** und **Andy Seaborne**: *SPARQL Query Language for RDF*. `http://www.w3.org/TR/rdf-sparql-query/`, Januar 2008. W3C Recommendation.

[PSW07a] **Parreiras, Fernando Silva**, **Steffen Staab** und **Andreas Winter**: *On marrying ontological and metamodeling technical spaces*. In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM*

SIGSOFT symposium on The foundations of software engineering, ESEC-FSE '07, Seiten 439–448, New York, NY, USA, 2007. ACM.

[PSW07b] **Parreiras, Fernando Silva**, **Steffen Staab** und **Andreas Winter**: *TwoUse: Integrating UML models and OWL ontologies*. Technical Report 16/2007, Universität Koblenz-Landau, Fachbereich Informatik, April 2007.

[PW08] **Protege-Wiki**: *Scalability and Tuning in Protege*. `http://protegewiki.stanford.edu/wiki/Scalability_and_Tuning`, Mai 2008. Besucht am 19. April 2011.

[Qi08] **Qi, Guilin**: *Diagnosing and repairing inconsistent networked ontologies*. `http://www.neon-project.org/nw/Deliverables`, September 2008. NeOn-Project Deliverable D1.2.3.

[Rei10] **Reimann, Jan**: *Generisches Modellrefactoring für EMFText*. Diplomarbeit, Technische Universität Dresden, Dresden, Juli 2010.

[RG98] **Riehle, Dirk** und **Thomas Gross**: *Role model based framework design and integration*. In: *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, Seiten 117–133, New York, NY, USA, 1998. ACM.

[Row07] **Rowley, Jennifer**: *The wisdom hierarchy: representations of the DIKW hierarchy*. In: *Journal of Information Science*, Band 33, Seiten 163–180, April 2007.

[RSA10] **Reimann, Jan**, **Mirko Seifert** und **Uwe Aßmann**: *Role-Based Generic Model Refactoring*. In: **Petriu, Dorina**, **Nicolas Rouquette** und **Øystein Haugen** (Herausgeber): *Model Driven Engineering Languages and Systems*, Band 6395 der Reihe *Lecture Notes in Computer Science*, Seiten 78–92. Springer Berlin / Heidelberg, 2010.

[SBPM09] **Steinberg, Dave**, **Frank Budinsky**, **Marcelo Paternostro** und **Ed Merks**: *EMF Eclipse Modeling Framework*. Addison Wesley, 2. Auflage, 2009.

[SBS10] **Sirin, Evren**, **Blazej Bulka** und **Michael Smith**: *Terp: Syntax for OWL-friendly SPARQL queries*. In: *OWLED 2010, Seventh International Workshop, San Francisco, California, USA*, Juni 2010.

[Sch10a] **Schneider, Carsten**: *Towards an Eclipse Ontology Framework: Integrating OWL and the Eclipse Modeling Framework*. In: *WeST - Institute for Web Science and Technologies*. University of Koblenz-Landau, Juni 2010.

[Sch10b] **Schneider, Mark**: *SPARQLAS: Writing SPARQL Queries in OWL Syntax*. Bachelor Thesis, University of Koblenz-Landau, Deutschland, 2010.

[SM02] **Stojanovic, Ljiljana** und **Boris Motik**: *Ontology Evolution within Ontology Editors*. In: *EKAW'02/EON Workshop*, Seiten 53–62, 2002.

[SMMS02] **Stojanovic, Ljiljana**, **Alexander Maedche**, **Boris Motik** und **Nenad Stojanovic**: *User-Driven Ontology Evolution Management*. In: *EKAW*, Seiten 285–300, 2002.

[SSM10] **Spanos, Dimitrios-Emmanuel**, **Periklis Stavrou** und **Nikolas Mitrou**: *Bringing Relational Databases into the Semantic Web: A Survey*. Semantic Web Journal, 1, November 2010.

[Sto04] **Stojanovic, Ljiljana**: *Methods and Tools for Ontology Evolution*. Doktorarbeit, Universitaet Fridericiana zu Karlsruhe, Karlsruhe, August 2004.

[SVEH07] **Stahl, Thomas**, **Markus Völter**, **Sven Efftinge** und **Arno Haase**: *Modellgetriebene Softwareentwicklung*. dpunkt-Verlag, 2. Auflage, Mai 2007.

[SWGP10] **Staab, Steffen**, **Tobias Walter**, **Gerd Gröner** und **Fernando Silva Parreiras**: *Model driven engineering with ontology technologies*. In: *Proceedings of the 6th international conference on Semantic technologies for software engineering*, ReasoningWeb'10, Seiten 62–98, Berlin, Heidelberg, 2010. Springer-Verlag.

[Two10] **TwoUse**: *OWLizer: Software Languages -> OWL ontologies Part I*. http://twouse.blogspot.com/2010/03/owlizer-transforming-ecore-into-owl.html, März 2010. Besucht am 26. März 2011.

[W3C09] **W3C, OWL Working Group**: *OWL 2 Web Ontology Language Document Overview*. `http://www.w3.org/TR/owl2-overview/`, Oktober 2009. W3C Recommendation.

[Wac07] **Wachsmuth, Guido**: *Metamodel Adaptation and Model Co-adaptation*. In: **Ernst, Erik** (Herausgeber): *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, Band 4609 der Reihe *Lecture Notes in Computer Science*, Seiten 600–624. Springer-Verlag, Juli 2007.

[WH09] **Wende, Christian** und **Florian Heidenreich**: *A Model-based Product-Line for Scalable Ontology Languages*. In: *1st International Workshop on Model-Driven Product-Line Engineering (MDPLE 2009) at ECMDA-FA*, 2009.

[Yil06] **Yildiz, Burcu**: *Ontology Evolution and Versioning The state of the art*. `http://publik.tuwien.ac.at/files/pub-inf_4603.pdf`, Oktober 2006. Technische Universität Wien.

[ZSDM09] **Zablith, Fouad**, **Marta Sabou**, **Mathieu D'Aquin** und **Enrico Motta**: *Ontology Evolution with Evolva*. In: *Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications*, ESWC 2009 Heraklion, Seiten 908–912, Berlin, Heidelberg, 2009. Springer-Verlag.

Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, den 03.05.2011