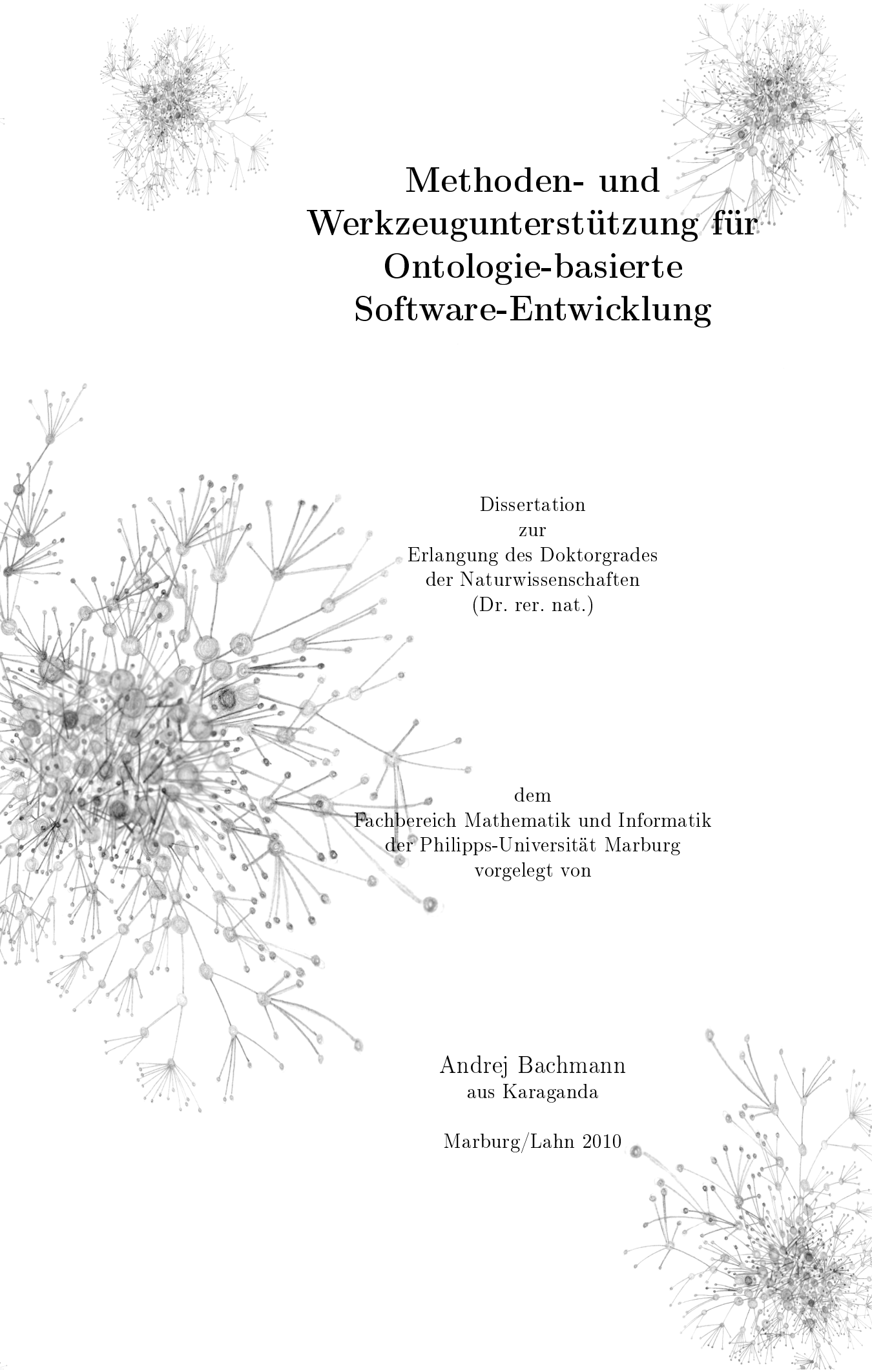


Methoden- und Werkzeugunterstützung für Ontologie-basierte Software-Entwicklung

Dissertation
zur
Erlangung des Doktorgrades
der Naturwissenschaften
(Dr. rer. nat.)

dem
Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg
vorgelegt von

Andrej Bachmann
aus Karaganda

Marburg/Lahn 2010

1. Gutachter: Prof. Dr. Wolfgang Hesse
2. Gutachter: Prof. Dr. Gabriele Taentzer

Tag der Einreichung: 11. August 2010
Tag der mündlichen Prüfung: 16. September 2010

Danksagung

An dieser Stelle möchte ich mich herzlich bei Herrn Prof. Dr. Wolfgang Hesse für die interessante Fragestellung, anregende Diskussionen und vielfältige Unterstützung dieser Arbeit bedanken. Prof. Dr. Gabriele Taentzer danke ich für die vielen wertvollen Ratschläge und die Übernahme des Zweitgutachtens. Für die Unterstützung und Motivation in der kritischen Phase dieser Arbeit möchte ich mich bei Prof. Dr. Manfred Sommer bedanken.
Diese Arbeit wäre ohne der gemeinsamen Idee der Ontologie-basierten Software-Entwicklung, gute Zusammenarbeit, regelmäßigen Diskussionen und Erfahrungsaustausch mit Prof. Dr. Heinrich C. Mayr, Ass. Prof. Mag. Dr. Christian Kop und Dipl.-Ing. Jürgen Vöhringer nicht möglich gewesen.
Darüber hinaus möchte ich mich auch bei allen Studentinnen und Studenten bedanken, die im Rahmen von Praktika und Diplomarbeiten zum Gelingen dieser Arbeit beigetragen haben. Besonders erwähnen möchte ich hierbei Dipl.-Inf. Aaron Ruß, Dipl.-Inf. Nils Andersch und Dipl.-Inf. Benjamin Horst.
Meine besondere Dankbarkeit gilt meiner Frau Viktoria und meiner Tochter Anastasia, die durch ihren Rückhalt, Verständnis und willkommene Ablenkung zur Entstehung dieser Arbeit beigetragen haben. Bei meiner Frau, Katherina Debus und Benedikt Schäfer bedanke ich mich für die kritische Durchsicht meiner Arbeit. Bei Irina Bachmann möchte ich für die Gestalltung des Titelbildes bedanken.

Zusammenfassung

Wiederverwendung wird in der Software-Technik eingesetzt, um sowohl Effizienz als auch die Qualität eines Produkts konstruktiv zu beeinflussen. Während einem Software-Projekt in den späten Phasen ausgereifte Techniken und Werkzeuge zur Verfügung stehen, fehlt weitgehend eine entsprechende Unterstützung bei der Untersuchung des Gegenstandsbereichs und der zugehörigen konzeptuellen Modellierung der Anforderungen. Dabei haben diese Aktivitäten einen entscheidenden Einfluss auf spätere Entwicklung und erfolgreiche Durchführung eines Software-Projekts.
Um die Wiederverwendung auf der konzeptuellen Ebene zu ermöglichen, wird in dieser Arbeit zunächst eine Infrastruktur entwickelt, die mit Hilfe einer Domänen-Ontologie einen Wissens-Austausch zwischen den beteiligten Projekten erlaubt. Anschließend wird ein auf dieser Infrastruktur aufbauender Prozess zur Ontologie-Entwicklung definiert und seine Anbindung an die klassische oder agile Vorgehensmodelle beschrieben. Ergänzend dazu wird ein Werkzeug zur Prozessunterstützung vorgestellt.

Abstract

Re-use is employed in software engineering to influence efficiency constructively and the overall quality of a product. While well-established tools and technologies are available in the advanced phases of software projects, support is missing to a large extent for analysis of the universe of discourse and the corresponding conceptual modelling of the requirements. At the same time these activities have a decisive influence on later development and successful implementation of software projects.
To enable re-use at the conceptual level, this thesis describes first of all, how to develop an infrastructure, which allows an exchange of knowledge between participating projects with the help of domain ontology. Subsequently, a process of an ontology development will be defined, which is based on this infrastructure; its connection to the classical or agile process models will also be described. In addition, a tool to support the process will be introduced.

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

Eine zentrale Aufgabe im Rahmen eines Software-Entwicklungsprozesses ist die Anforderungsanalyse. Es ist eine Wissens-intensive Tätigkeit, bei der Entwickler mit Hilfe von Fachexperten und Kunden Informationen über einen Ausschnitt der Domäne gewinnen, für den später ein Software-Produkt entwickelt werden soll. Mit Hilfe dieser Informationen wird dieser Bereich der Domäne modelliert und es werden somit die Weichen für die Entwicklung des zukünftigen Systems gestellt.

Während in den nachfolgenden Phasen des Software-Entwicklungsprozesses Entwickler durch Wiederverwendung auf der Entwurfs-Ebene (zum Beispiel mit Objekt-orientiertem Paradigma, Entwurfsmustern) und Implementierungs-Ebene (mit Frameworks und Bibliotheken) unterstützt werden, fehlt diese Unterstützung weitgehend auf der konzeptuellen Ebene. Als Konsequenz wird mit jedem neuen Projekt die zugehörige Domäne wiederholt analysiert und modelliert. Zwar lässt sich die Analyse und Modellierung einer Domäne im Rahmen der Anforderungsanalyse nicht einsparen, diese Schritte können aber durch die Wiederverwendung auf der konzeptuellen Ebene effizienter gestaltet werden. Zusätzlich kann die Qualität dieser Modelle dadurch gesteigert werden, indem bereits erprobte Konzeptualisierungen der Domäne übernommen werden.

Jede Form der Wiederverwendung erfordert eine Struktur, mit deren Hilfe für die Wiederverwendung interessante Elemente verkapselt und an anderen Stellen eingebaut werden können. Die Herausforderung bei der Wiederverwendung auf der konzeptuellen Ebene im Vergleich zu den anderen Ebenen liegt darin, Wissensaustausch in einem inhomogenen Umfeld zu ermöglichen. So werden Konzepte und ihre Zusammenhänge nicht im gleichen Projekt wiederverwendet, in dem sie entstanden sind. Dabei bauen

die Projekte in der Regel auf unterschiedlichen Techniken auf.

In einem solchen Umfeld bietet es sich an, Ontologien als den Kern der Wiederverwendungsstruktur zu verwenden. Als Konzeptualisierung einer Domäne, die gemeinsam von allen Beteiligten verwendet wird, ermöglicht eine Ontologie, abstrahiertes Wissen über die Domäne aus Projekten zu sammeln und weiterzugeben. Ontologien wurden bereits in anderen Bereichen der Informatik für die Aufnahme von Wissen über eine Domäne und einen übergreifenden Wissensaustausch erfolgreich eingesetzt und gewinnen auch in der Software-Technik an Bedeutung.

Einführung von Wiederverwendung auf der konzeptuellen Ebene führt zu Veränderungen in dem Prozess der Anforderungsanalyse. Mit einer Domänen-Ontologie steht einem Entwickler eine zusätzliche Quelle zur Verfügung, um Informationen über die Domäne zu gewinnen. Um ihre Nutzung in die Abläufe der Anforderungsanalyse zu integrieren, ist ein neues Prozessmodell erforderlich, das den Entwickler unterstützen und ihn bei der Anwendung der Wiederverwendung auf der konzeptuellen Ebene leiten soll.

1.2 Ziele der Arbeit

Als Ausgangspunkt für diese Arbeit diente die Idee der *Ontologie-basierten Software-Entwicklung (OBSE)* ([BHR+07b], [BHR07a]), die in der Zusammenarbeit mit dem Institut für Angewandte Informatik an der Alpen-Adria-Universität Klagenfurt entstanden ist. Diese Idee baut auf der Integration von Domänen-Ontologien in existierende Software-Entwicklungsprozesse, um Wissen über eine Domäne zwischen verschiedenen Projekten auszutauschen und dadurch Wiederverwendung auf der konzeptuellen Ebene zu ermöglichen. Dabei stellt sich die Frage, wie eine Ontologie effizient, gewinbringend und ohne große Veränderungen von existierenden Software-Prozessen in diese einbinden lässt. Das zentrale Ziel dieser Arbeit ist, diese Integration zu konzipieren und *OBSE* zu einem einsetzbaren Software-Entwicklungsprozess auszubauen. Dafür wurden folgende Teilaufgaben formuliert und umgesetzt:

- Um die neuen Prozesse beschreiben zu können wird zunächst eine Infrastruktur entworfen. Sie soll den statischen Aufbau des Prozesses beschreiben und festlegen, welche Modelle und Transformationen auf verschiedenen Modellierungsebenen eingesetzt werden und wie sie miteinander verknüpft sind.
- Um die Verwendung von Ontologien in und aus Software-Projekten

zu ermöglichen, wird der Ontologie-Entwicklungszyklus mittels Phasen, Abläufen und Rollen definiert und seine Kopplung an Software-Entwicklungszyklen beschrieben.

- Um die praktische Anwendung von *OBSE* zu unterstützen, wird ein Werkzeug entwickelt, mit dem die Infrastrukturelemente des Prozesses verwaltet werden können.
- Um die ersten praktischen Erkenntnisse über den Prozess zu gewinnen, wird anschließend eine Evaluierung anhand von realen Projekten durchgeführt.

1.3 Aufbau der Arbeit

Diese Arbeit gliedert sich in acht Kapitel. Nach einer kurzen Einleitung werden im *Kapitel zwei* wichtige Grundlagen vorgestellt, die zum besseren Verständnis der verwendeten Techniken beitragen sollen. Im *Kapitel drei* folgt ein Exkurs über Szenarien, wie Ontologien in der Software-Technik eingesetzt werden. Eine Basis dafür liefern aktuelle Forschungsarbeiten in diesem Bereich. Ergänzend dazu wird im *Kapitel drei* die *Ontologie-basierte Software-Entwicklung* vorgestellt, in die Szenarien-Klassifikation eingeordnet und gegenüber anderen Verfahren abgegrenzt.

Kapitel vier und *fünf* dienen der Beschreibung des neuen Prozesses. Zunächst wird im *Kapitel vier* die *OBSE*-Infrastruktur präsentiert. Hier werden die Artefakte wie Domänen-Ontologie, konzeptuelles und Analyse-Modell und Verbindungen zwischen diesen Modellen in Form von Transformationen oder Import- und Export-Brücken aufgezeichnet. *Im Kapitel fünf* werden diese Artefakte in die Prozesse integriert und die Zuständigkeiten der Rollen festgelegt. Zusätzlich werden in diesem Kapitel die Phasen der Ontologie-Entwicklung beschrieben.

Kapitel sechs stellt ein neu entwickeltes Werkzeug für die Prozessunterstützung vor und gibt einen Einblick in seine Umsetzung. Im *Kapitel sieben* wird der Versuchsaufbau für die Evaluierung des *OBSE*-Prozesses vorgestellt und die Ergebnisse werden diskutiert. Abgeschlossen wird diese Arbeit mit einer Zusammenfassung und einem Ausblick mit Vorschlägen, an welchen Stellen die *Ontologie-basierte Software-Entwicklung* noch verändert und weiterentwickelt werden kann.

2 Grundlagen

Dieses Kapitel ist den Methoden der Software-Technik gewidmet, auf denen die *Ontologie-basierte Software-Entwicklung* aufbaut. Software-Technik umfasst eine breit gefächerte Palette von Teilgebieten, die sich mit der Herstellung und dem Betrieb von Software-Systemen beschäftigen. Sie behandeln sowohl Kernprozesse der Software-Entwicklung wie Analyse, Entwurf, Programmierung und Test als auch Unterstützungsprozesse wie Prozess-, Qualitätsmanagement. Drei grundlegende Stichworte aus diesem Themenkomplex spielen für die vorliegende Arbeit eine besondere Rolle: *Modellierung*, *Ontologie* und *Software-Prozess*.

Der Abschnitt 2.1 führt die relevanten Begriffe der Modellierung ein und präsentiert Techniken, die für die Beschreibung von Modellen verwendet werden. Dazu zählen sowohl *UML* – eine standardisierte und etablierte Modellierungssprache als auch *CPM* – eine spezielle auf die Kommunikation mit den Kunden bzw. Fachverantwortlichen ausgerichtete Methode zur Anforderungsanalyse. Abgeschlossen wird diese Einführung mit einem Einblick in die *Modell-zu-Modell-Transformationen*, die Übergänge zwischen den verschiedenen Modellen des *OBSE*-Prozesses ermöglichen.

Wie bereits der Name andeutet, ist ein zentraler Bestandteil der Ontologie-basierten Software-Entwicklung eine *Ontologie*. Als eine relativ neue Basis für die Software-Entwicklung wird sie ausführlich im Abschnitt 2.2 vorgestellt und gegenüber anderen Begriffen wie *Meta-* und *konzeptuelles Modell* abgegrenzt. Eine Ontologie bedarf – analog zu Modellen – eines Ausdrucksmittels, einer Sprache, in der sie verfasst und kommuniziert wird. Ein Überblick über die gängigen Ontologie-Sprachen und ihre Klassifizierung wird im Abschnitt 2.2.4 vorgestellt.

Um Ontologien in Software-Entwicklungsprozesse zu integrieren, erweitert *OBSE* ein bereits bestehendes Software-Prozessmodell *EOS*. Dadurch rücken Prozesse, ihre Elemente und die Möglichkeiten, diese zu beschreiben,

ebenfalls in den Mittelpunkt dieser Arbeit. Im Kapitel 2.3 wird zunächst ein Überblick über Software-Entwicklungsprozesse und dazu aktuell eingesetzte Vorgehensmodelle gegeben. In diesem Zusammenhang wird das evolutionäre Vorgehensmodell *EOS* vorgestellt, an welches *OBSE* anknüpft. Anschließend wird das standardisierte Meta-Modell *SPEM* zur Beschreibung von Software-Entwicklungsprozessen präsentiert, das die formale Grundlage für die Definition von *OBSE*-Prozessen liefert.

2.1 Modellierung

2.1.1 Modelle und Modellierungsräume

Abstraktion gehört zu den grundlegenden kognitiven Fähigkeiten eines Menschen. Regelmäßig greift er auf dieses Mittel zu, um mit der – im Vergleich zu unserem Wahrnehmungsvermögen – komplexen realen Welt zu interagieren. Die Komplexität einer Situation wird dabei dadurch reduziert, dass nur die für eine gegebene Problemstellung relevanten Eigenschaften wahrgenommen und anhand deren Analysen und Lösungen entwickelt werden [HM08]. Als Paradebeispiel dafür kann eine Landkarte dienen, welche die topographischen Informationen eines Geländeausschnitts erfasst und es ermöglicht, Fragestellungen wie Länge des Weges oder Höhendifferenz zwischen zwei Orten anhand der abgebildeten Fakten zu beantworten. Das Ergebnis einer Abstraktion – wie im Falle einer Landkarte – wird als *Modell* bezeichnet.

In der Literatur finden sich unterschiedliche Definitionen des Modell-Begriffs, die sich in der Regel auf Modelle in einem bestimmten Umfeld konzentrieren und deren Eigenschaften hervorheben. Eine kompakte und allgemein anwendbare Definition eines Modells geben Haggett und Chorley [HC67] an: *Ein Modell ist eine vereinfachende Abstraktion der Realität*[1]. Ein Modell steht nach dieser Definition stellvertretend für *etwas* aus der realen Welt. Einen treffenden Begriff für dieses *etwas* fand Stachowiak [Sta73], in dem er die neutrale Bezeichnung *Original* verwendete.

Der Vorteil dieser Begriffswahl liegt darin, dass der zu modellierende Gegenstand[2] nicht anhand seiner Eigenschaften definiert wird, sondern über die *Rolle*, die er in dem Modellierungsprozess spielt. So können sowohl natürliche als auch abstrakte Gegenstände als Original fungieren und somit

[1] Original: A model is a simplified abstraction of reality

[2] Obwohl das Wort Gegenstand an dieser Stelle allgemein verwendet wird, wirkt es im Vergleich zu Original einschränkend und verdeutlicht die gute Wahl des Begriffs.

als Vorlage für ein Modell dienen. Auch bereits erstellte Modelle können die Rolle des Originals einnehmen, wodurch Ketten von Modellen gebildet werden können [HM08].

Im Bereich der Software-Technik und insbesondere im *Model Driven Development (MDD)* werden als Spezialfall dieser Kettenbildung *Modelltransformationen* für die Entwicklung eines Software-Produkts eingesetzt. Ausgehend von fachlichen Anforderungen, die meist in textueller Form Originale aus der realen Welt beschreiben und somit das erste Modell in der Kette darstellen, wird in *MDD* mit Hilfe der Modelltransformationen ein Übergang zum Code – meist das letzte Modell in dieser Kette – hergestellt. Die zugehörige Architektur *Model Driven Architecture (MDA)* gibt drei Modelltypen vor, die für einen schrittweisen Übergang von den Anforderungen zum Code dienen sollen: *Computation Independent Model (CIM)*, *Platform Independent Model (PIM)* und *Platform Specific Model (PSM)*. Der Gedanke, Software-Produkte mit Hilfe von mehreren aufeinander aufbauenden Modellen zu entwickeln, findet sich, wenn auch nicht immer so stark ausgeprägt, auch in den anderen Methoden der Software-Entwicklung.

Was spricht dafür, die Entwicklung eines Software-Produkts aufbauend auf mehreren unterschiedlichen Modellen durchzuführen?

In seinem umfassenden Werk „Allgemeine Modelltheorie“ [Sta73] führte Stachowiak zusätzlich zu dem *Abbildungs-* und *Reduktionsmerkmal*, die sich bereits in der Definition von Chorley und Haggett erkennen lassen, eine weitere wichtige Charakterisierung eines Modells ein: das *pragmatische Merkmal*. Er hebt damit hervor, dass die Modellierung immer zweckgebunden stattfindet. Auf diese Art übernehmen verschiedene Modelle in einem Software-Projekt unterschiedliche Aufgaben.

Im Falle von *MDA* ist es die Aufgabe von *CIM*, einen zu modellierenden Ausschnitt der realen Welt – *den Gegenstandsbereich*[3] – unabhängig von den strukturellen Details des zukünftigen Systems zu erfassen. *PIM* ist ein für eine Umsetzung auf einem beliebigen Computer vorgesehenes Modell, das also von der jeweiligen Rechner-Plattform unabhängig ist. Technisch betrachtet, kann man sich unter *PIM* ein Modell vorstellen, das für eine von der spezifischen Umsetzung unabhängige virtuelle Maschine bestimmt ist. *PSM* stellt dagegen ein Modell dar, das für eine bestimmte Rechner-Plattform entwickelt wurde[4].

[3] Ein weiterer in der Modellierung relevanter Begriff ist der *Untersuchungsbereich*. Dieser beinhaltet den *Gegenstandsbereich* und zusätzlich weitere Elemente aus der realen Welt, die für das Verständnis von Elementen des *Gegenstandsbereichs* wichtig sind.

[4] Für weiterführende Informationen zu *CIM*, *PIM* und *PSM* siehe [GDD06]

In dieser Arbeit wird folgende Definition eines Modells verwendet, welche aus der Modellbeschreibung von Mellor et al. abgeleitet wurde [MCF03] und die wichtigsten Merkmale eines Modells zusammenfasst:

Definition 1 (Modell) *Ein Modell ist eine kohärente Menge formaler Elemente, die zur Beschreibung eines Originals mit einem festgelegten Zweck erstellt wird.*

Implizit deutet diese Definition auf zwei weitere wichtige Bestandteile der Modellierung hin: Eine Menge der Elemente, die in einem Modell verwendet werden können, und Regeln beziehungsweise Einschränkungen, die den konsistenten und zweckgebundenen Gebrauch von diesen Elementen festlegen. Die ersten bilden den Wortschatz einer *Modellierungssprache* und die zweiten legen die Grammatik einer Sprache fest, die in der Modellierung als *Meta-Modell* bezeichnet wird.

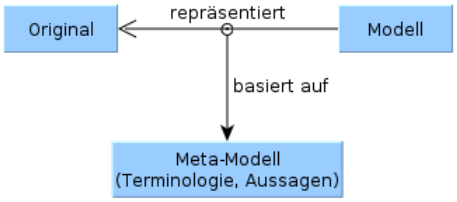


ABBILDUNG 2.1: Zusammenhang der Begriffe: Original, Modell und Meta-Modell (vgl. [GDD06])

Im Unterschied zu anderen Naturwissenschaften und dem Modellieren im weiten Sinne, wo hauptsächlich Modelle selbst und nicht die Mittel zu ihrer Erzeugung im Vordergrund stehen[5], beschäftigt sich die Software-Technik explizit mit Modellierungssprachen wie *Unified Modeling Language (UML)* und den zugehörigen Meta-Modellen wie *Meta Object Facility (MOF)* [Sei03]. Ihre formale Definition spielt sowohl für die einheitliche Interpretation der Modelle, als auch für Entwicklung der passenden Werkzeuge eine wichtige Rolle.

Während Modellierungssprachen in der Software-Technik schon lange ihre Anwendung finden, handelt es sich bei den formalen Sprachbeschreibungen mittels Meta-Modellen um eine vergleichsweise neue Modellierungstechnik, die insbesondere mit *MDA* ihren Einzug in die Software-Technik gehalten hat. Ein Meta-Modell kann nach Seidewitz [Sei03] wie folgt definiert werden:

[5] Zum Beispiel werden in der Physik viele Modelle der realen Welt wie das *Bohrsche Atom-Modell* erstellt. Auch wenn in der Sprache dieses Modells unterschiedliche Kugeln verwendet werden, findet sich in den Physikbüchern nichts über die *Bohrsche Modellierungssprache*.

Definition 2 (Meta-Modell) *Ein Meta-Modell liefert Aussagen über das, was in einem Modell unter Verwendung einer bestimmten Modellierungssprache ausgedrückt werden kann*[6].

Verallgemeinert betrachtet ist ein Meta-Modell ein präskriptives[7] Modell einer Sprache. Die Abbildung 2.1 veranschaulicht schematisch den Zusammenhang zwischen den in diesem Abschnitt eingeführten Begriffen: Original, Modell und Meta-Modell.

Modellierungsräume

Eine breite Palette an verschiedenen Modellierungssprachen, die in der Software-Entwicklung von universellen Sprachen wie *UML*[8] bis hin zu sehr speziellen wie den *Domänen-spezifischen Sprachen* reicht, ermöglicht es einem Entwickler[9] den Gegenstandsbereich genauer zu beschreiben. Diese Vielfalt birgt – falls sich die Grundlagen dieser Sprachen zu stark unterscheiden – die Gefahr in sich, dass ein Entwickler – insbesondere unter Zeitdruck – sich nur mit ausgewählten Sprachen auseinandersetzt. Durch diese Spezialisierung werden möglicherweise nur ausgewählte Aspekte eines Gegenstandsbereichs abgebildet und die Kommunikation zwischen den Entwicklern wird erschwert, sofern sie unterschiedliche Schwerpunkte setzen [Sei03].

Um zu vermeiden, dass diese Vielfalt zu Zuständen wie beim *Turmbau zu Babel* führt, ist es erstrebenswert, die Beschreibung von Modellierungssprachen auf eine einheitliche Basis zu stellen [GDD06]. Eine gemeinsame Grundlage für die Modelle einer Sprache ist das zugehörige Meta-Modell. Eine Vereinheitlichung der Beschreibung dieser Meta-Modelle kann mit Hilfe eines für sie präskriptiven, eine Ebene höher stehenden Meta-Meta-Modells erreicht werden. Dadurch entsteht eine mehrschichtige Architektur der Modellierungssprachen, die im Zusammenhang mit *MDA* eingeführt wurde. Ihre allgemeine Form (ohne Belegung mit *MDA*-spezifischen Modellen) ist in der Abbildung 2.2 dargestellt.

[6] Original: A meta-model makes statements about what can be expressed in the valid models of a certain modelling language.

[7] Wird ein Meta-Modell nachträglich für eine bereits existierende Sprache entwickelt, trägt es ergänzend zu seinem vorschreibenden auch beschreibenden, deskriptiven Charakter.

[8] Für weitere Informationen zu *UML* siehe das folgende Kapitel 2.1.2

[9] In dieser Arbeit wird *Entwickler* stellvertretend für alle Rollen eines Projekts (Analyst, Architekt, Tester usw.) verwendet.

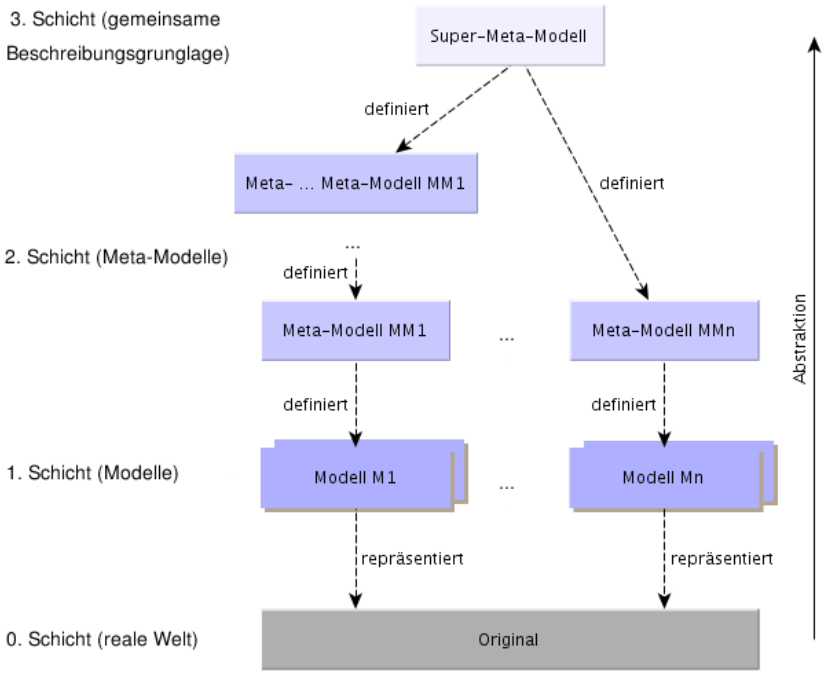


ABBILDUNG 2.2: Modellierungsräume (vgl. [GDD06])

Die unterste Schicht wird durch ein Original (ein Ausschnitt der realen Welt, ein Gegenstandsbereich) gebildet[10], von dem aus die Modelle auf der höher liegenden ersten Schicht erstellt werden. Typische Beispiele für diese Modelle sind *UML-Klassendiagramme*. Die Meta-Modelle in der zweiten Schicht regeln den Aufbau der Modelle. Diese Schicht kann sowohl horizontal als auch vertikal ausgebaut werden. Eine horizontale Erweiterung bedeutet, dass eine zusätzliche Sprache zur Beschreibung eines Originals zur Verfügung gestellt wird.

Vertikaler Ausbau führt zur Bildung von Unterschichten innerhalb der zweiten Schicht. Jede interne Schicht beinhaltet ein Meta-Modell[11], das

[10] Zugehörigkeit der untersten Schicht zu den Modellierungsräumen hat zu kontroversen Diskussionen geführt, die unter anderem von Thomas Kühne angestoßen wurden [Kü02]. Seine Argumentation basiert darauf, dass die Beziehungen zwischen oberen drei Schichten eine andere inhaltliche Qualität haben im Vergleich zu der Beziehung zwischen der untersten und der ersten Schicht (in Abbildung 2.2 sind sie unterschiedlich benannt). Das führt zu einem Bruch im Schichtenaufbau.

[11] Wobei das Präfix *Meta* entsprechend der Anzahl der Unterschichten wiederholt wird. So wird ein Modell zur Definition eines Meta-Modells als *Meta-Meta-Modell* bezeichnet.

darunter liegende Meta-Modelle definiert und sie verallgemeinert. Dadurch nimmt auch der Abstraktionsgrad der Modelle zu.

Genau betrachtet stellt das *Super-Meta-Modell*[12] in der obersten dritten Schicht ein weiteres Glied in dieser Meta-Modellierungskette dar. Allerdings wird von diesem Modell verlangt, dass alle darunter liegenden Meta-Modelle auf dem Super-Meta-Modell basieren. Aus diesem Grund, und um das Ende dieser Kette zu signalisieren, wird dem Super-Meta-Modell eine eigene Schicht zugesprochen. Diese allgemeine Darstellung wird in *MDA* vereinfacht (ohne die Bildung von vertikalen Zwischenebenen in der zweiten Schicht) angewendet. Typisches Beispiel eines Meta-Modells in dieser Schicht ist das *UML-Meta-Modell* und die dritte Schicht wird mit *MOF* besetzt. Durch den hohen Abstraktionsgrad beschreibt ein Super-Meta-Modell wie *MOF* in der Regel allgemeine Grundelemente wie Entitäten und Relationen, aus denen Modelle zusammengesetzt werden.

2.1.2 UML

Unified Modeling Language (UML) ist eine graphische, konzeptuelle Modellierungssprache, die insbesondere in der Modellierung von Software-Systemen eine dominierende Rolle eingenommen hat. Sie ist durch die *Object Management Group (OMG)* standardisiert ([OMG05b] und [OMG07]).

Wie die meisten graphischen Modellierungssprachen im der Software-Technik entstand *UML*, weil (textuelle) Programmiersprachen nicht abstrakt genug sind, um als Grundlage für die Entwurfsdiskussionen zu dienen [Fow04]. Als eine Vereinheitlichung (*Unified*) und eine Weiterentwicklung der bereits existierenden graphischen Sprachen bietet *UML* Ausdrucksmittel sowohl für statische (z.B. Klassendiagramm) als auch für dynamische (z.B. Interaktions- und Aktivitätsdiagramm) Modellierung[13]. Dadurch kann sie auf verschiedene Art und Weise während der Modellierung eines Software-Systems eingesetzt werden.

UML stellt sowohl eine Notation für die graphische Darstellung der Sprachelemente als auch eine Spezifikation in Form von *Meta-Modellen* zur Verfügung. Insgesamt besteht sie aus vier Teilen, die sich gegenseitig ergänzen:

[12] Diese Bezeichnung wird unter anderem von Gasevi'c et all. [GDD06] verwendet.

[13] Zur Zeit bietet *UML* 13 Diagrammtypen an. Allerdings verschwimmen die Grenzen zwischen den Diagrammtypenen in den aktuellen Versionen von *UML* immer mehr, um eine integrative Nutzung der Elemente zu erlauben.

Infrastructure enthält grundlegende Sprachkonstrukte, die gemeinsam von mehreren *OMG*-Spezifikationen – unter anderem *UML* und *MOF* – verwendet werden. Dieses Meta-Modell führt zentrale Modellierungselemente wie *Klasse*, *Assoziation* (im Paket Basics); *Generalisierung*, *Instanz* (im Paket Abstractions), *Attribut*, *Operation* (im Paket Constructs); primitive Datentypen wie *Integer*, *Boolean* (im Paket PrimitiveTypes) ein [OMG07] .

Superstructure instantiiert und importiert Elemente aus *Infrastructure* und reichert diese mit *UML*-spezifischen Eigenschaften an und fügt neue Sprachelemente wie zum Beispiel Aktivität oder Anwendungsfall hinzu [OMG05b].

Object Constraint Language (OCL) ist eine textuelle Sprache zur Beschreibung von Bedingungen in *UML*. Diese Bedingungen können verwendet werden, um zum Beispiel Einschränkungen in Klassendiagrammen oder Vor- und Nachbedingungen von Operationen zu definieren[14].

Diagram Interchange bezieht sich im Unterschied zu den oberen Punkten nicht auf das Festlegen von Sprachelementen, sondern unterstützt die Austauschbarkeit ihrer Diagramme. *Diagram Interchange* definiert Eigenschaften wie Position und Größe von Modellelementen, die mit diesem Standard von einem *UML*-Tool zu einem anderen übertragen werden können. Die semantische Austauschbarkeit wird unabhängig davon in einem weiteren *OMG*-Standard definiert, der übergreifend für *UML*, *MOF* und *XML* gilt. Sein Name ist *XML Metadata Interchange (XMI)*.

Die wichtigsten Einsatzmöglichkeiten von *UML* sind nach Fowler *UML als Skizze*, *Reverse Engineering* und *Forward Engineering* [Fow04]. Im ersten Fall handelt es sich eher um eine Sprache zur Kommunikation zwischen den *UML*-visierten Projektbeteiligten, die anhand von einem kompakten Modell eine Situation untersuchen wollen. Das *Reverse Engineering*, bei dem ein *UML*-Modell aus dem vorhandenen Code erzeugt wird, dient hauptsächlich dem Verständnis des Codes und seiner Optimierung. Der wichtigste Gebrauch von *UML* besteht, meiner Meinung nach, im *Forward Engineering*. In diesem Fall dienen UML-Modelle als Bauplan für das zu entwickelnde Software-System. Eine konsequente Anwendung dieses Einsatzzwecks findet sich in Ansätzen wie *MDA*[15] wieder.

[14] Die Sprache *OCL* wird außerdem eine wichtige Rolle in dieser Arbeit bei der Beschreibung von Transformationen spielen (siehe Kapitel 2.1.4).

[15] Siehe auch 2.1.1

2.1.3 Conceptual Predesign Method

Zu den wichtigsten Teilaufgaben, die während der Entwicklung eines Software-Systems (oder einer seinen Komponenten) durchgeführt werden, gehört die Anforderungsanalyse[16]. Eine umfassende und konsistente Analyse der Anforderungen ist eine wichtige Voraussetzung für eine erfolgreiche Durchführung eines Software-Projekts. Die *CHAOS*-Studie der *Standish Group*, die sich seit 1994 jährlich mit der Untersuchung von problematisch verlaufenden und fehlgeschlagenen Projekten beschäftigt, stellt fest, dass eine der häufigsten Ursachen für das Scheitern von Software-Projekten in den mangelhaft definierten Anforderungen und ihrer Spezifikation liegt [Sta03]. Die Projektverantwortlichen nennen im Zusammenhang mit der Anforderungsanalyse folgende Ursachen[17]:

- 13% – unzureichende Kommunikation mit den Anwendern/Auftraggebern,
- 12% – lückenhafte Anforderungen oder unzureichende, mangelhafte Spezifikation,
- 12% – sich oft ändernde Anforderungen.

Während der letzte Punkt in dieser Auflistung durch das gewählte Vorgehensmodell beeinflusst werden kann und von dessen Flexibilität abhängt, stellen die ersten zwei Punkte ein generelles, von einem Vorgehensmodell weitgehend unabhängiges Problem dar. Sie betreffen die Schnittstelle zwischen einem Projekt und dessen Anwendern (bzw. Fachexperten), für die das Software-System erstellt wird. Die betroffenen Artefakte sind einerseits Anforderungen, die in der Regel in der natürlichen Sprache verfasst sind und aus der Welt der Anwender stammen, und andererseits ihre Spezifikation, die in einer im Vergleich zur natürlichen Sprache formaleren Form verfasst wird. Eine verbreitete, semi-formale Form der Anforderungsspezifikation in den heutigen Software-Projekten sind Modelle in der konzeptuellen Modellierungssprache *UML*.

Eine Spezifikation baut auf Anforderungen auf und dient als Ausgangsbasis für weitere Schritte und Artefakte, die in einem Software-Projekt durchgeführt werden bzw. entstehen. Die Übergänge zwischen diesen Artefakten können als eine Kette von Transformationen angesehen werden. Moderne Technologien wie *Model Driven Architecture* und eine ausgereifte Unterstützung durch aktuelle Entwicklungswerkzeuge machen eine weitge-

[16] Verbreitet ist auch die englische Bezeichnung *Requirements Engineering*.
[17] Laut *CHAOS*-Studie aus dem Jahr 2003

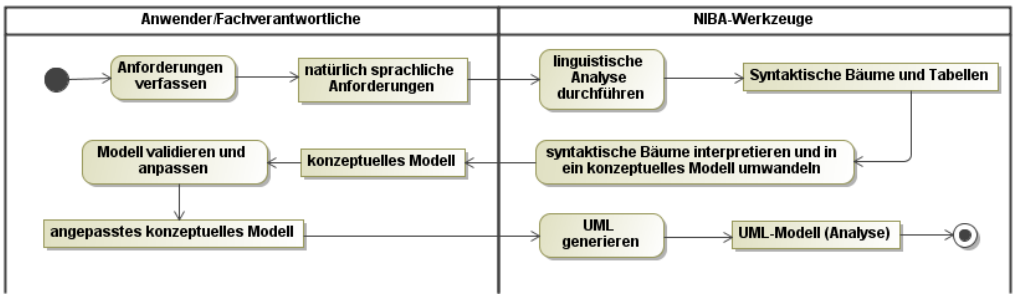


ABBILDUNG 2.3: Schematische Darstellung des *NIBA*-Prozesses.

hend automatische Durchführung dieser Transformationen – beginnend bei der Spezifikation bis hin zum Code – möglich und die dabei entstehenden Anwendungen immer weniger fehleranfällig. Das Übergang von den Anforderungen zu deren Spezifikation wird dagegen noch nicht ausreichend automatisiert bzw. unterstützt [MK02].

Eine Automatisierung der Schritte, die bei der Analyse der Anforderungen bis hin zur Erstellung der entsprechenden Spezifikation anstehen, strebt das *NIBA*-Projekt[18] an [FKM+00]. Dieses Projekt definiert einen Prozess, bei dem ausgehend von den einfachen[19], natürlich-sprachlichen Anforderungsdokumenten (siehe Abbildung 2.3) zunächst eine linguistische Auswertung durchgeführt wird und die entsprechenden Analyse-Bäume und -Tabellen aufgebaut werden [FKM+02]. Sie dienen als Ausgangsbasis für ein konzeptuelles Modell, das durch eine automatisierte Interpretation der linguistischen Bestandteile abgeleitet wird. Das Diagramm in Abbildung 2.3 zeigt auch, dass zukünftige Anwender des Software-Systems dieses konzeptuelle Modell anhand von Anforderungen validieren und anpassen können, bevor es in einem weiteren Schritt semi-automatisch in ein *UML*-Modell der System-Analyse transformiert wird.

Das konzeptuelle Modell und die zugehörigen Abläufe bilden neben den linguistischen Komponenten einen wichtigen Teilbereich des *NIBA*-Prozesses. Er wird mit *Conceptual Predesign Method (CPM)* bezeichnet

[18] *NIBA* steht für *Natural Language Requirements Analysis*. Das Projekt wird von der Klaus Tschira Stiftung in Heidelberg gefördert.

[19] In der aktuellen Version der *NIBA*-Werkzeuge ist es nicht mehr erforderlich, dass die Begriffe in dem Anforderungstext mit Markierungen versehen werden, damit sie richtig zugeordnet und sprachlich klassifiziert werden können. Die Voraussetzung dafür ist, dass im Text einfache Satzformen verwendet werden und Querverweise (Wörte wie dieser) nur in unmittelbarer Nähe zu ihrem Bezugspunkt (Nebensätze, direkt folgende Sätze) erlaubt sind.

[MK02] und dient als ein Bindeglied zwischen der Welt der Benutzer und der Welt der Entwickler. Wie unter anderem die *CHAOS*-Studie zeigt, spielt eine effiziente Zusammenarbeit zwischen diesen Gruppen eine wichtige Rolle für eine erfolgreiche Durchführung eines Software-Projekts. Während die Fachverantwortlichen Wissen über den entsprechenden Untersuchungsbereich besitzen und in der Regel keine Vorkenntnisse aus dem Bereich der Software-Technik mitbringen, besitzen die Entwickler umgekehrt softwaretechnisches Fachwissen, kennen sich aber mit dem Untersuchungsbereich wenig aus.

Die für den Wissensaustausch in einem typischen Software-Projekt zur Verfügung stehende Artefakte – Anforderungsbeschreibungen und *UML*-Modelle in der Form von Anwendungsfall-, Aktivitäts- und Klassen-Diagrammen – unterscheiden sich sehr stark in ihrer Einsetzbarkeit bezüglich der beteiligten Gruppen. Die ersten können sowohl von den Anwendern als auch von den Entwicklern verwendet werden, um über den Untersuchungsbereich zu diskutieren. Allerdings lassen sie auf Grund ihrer natürlich-sprachlichen Form einen aus Sicht eines Software-Entwicklers zu großen Interpretationsspielraum zu [FKM+02].

UML-Modelle richten sich dagegen hauptsächlich nach den Bedürfnissen eines Software-Entwicklers. In erster Linie als eine Ausgangsbasis gedacht, um den Code des zu entwickelnden Systems zu erstellen, sind sie zu komplex und zu abstrakt, um hinreichend von den Anwendern oder Fachverantwortlichen des zukünftigen Systems verstanden zu werden. Außerdem werden in den *UML*-Modellen oft Detail-Entscheidungen verlangt, die schon in den frühen Projekt-Phasen in den Entwurf des zukünftigen Software-Systems eingreifen, deren Auswirkungen und Konsequenzen während der Anforderungsanalyse von den Entwicklern (und erst recht von den Anwendern) noch nicht ausreichend abgeschätzt werden können [MK02].

Die beschriebene Situation war die Ausgangsbasis für die Entwicklung der *Conceptual Predesign Methode*. Diese soll eine konzeptuelle Modellierung auf einer Ebene ermöglichen, die einerseits von den Anwendern bzw. deren Fachverantwortlichen ohne Vorkenntnisse verstanden wird und gleichzeitig den Entwicklern eine – im Vergleich zur natürlichen Sprache – formalere Grundlage anbietet, um in den Anforderungen definierte Sachverhalte abbilden zu können. Wie der Name der Methode andeutet, liegt ihr Schwerpunkt *vor* der Entwurfsphase des zukünftigen Software-Systems.

Das zentrale Element von *CPM* ist – wie in Abbildung 2.3 gezeigt – ein konzeptuelles Modell, das als ein neues Artefakt zwischen den Anforderungen und den *UML*-Modellen platziert wird. Da *UML* eine konzeptuelle Mo-

dellierungssprache ist, handelt es sich bei den *UML*-Modellen ebenfalls um konzeptuelle Modelle. Der Unterschied zu den *CPM*-Modellen liegt nicht nur in der sprachlichen Klassifikation, sondern im Detaillierungsgrad und der möglichen Komplexität der Modelle.

Konkret werden *CPM*-Modelle in der konzeptuellen Modellierungssprache *Conceptual Predesign Language (CPL)*[20] ausgedrückt. Die Sprache bietet folgende Haupt-Modellierungselemente an [MK06]:

ThingType (*Dingtyp*) Ein *ThingType* steht für Gegenstände bzw. vergegenständlichte Sachverhalte eines Untersuchungsbereichs. Dieses Modellierungselement beinhaltet eine Bezeichnung, Beschreibung und den anwendbaren Wertebereich des zu modellierenden Konzepts. *ThingType* ist ein zentrales Konzept von *CPL* und ist mit *Klassen* in *UML* vergleichbar. Allerdings besitzt ein *ThingType* keine Attribute. Sie werden in *CPL* ebenfalls als *ThingTypes* modelliert, was zu einer vergleichbar flachen Struktur führt.

ConnectionType (*Zusammenhangstyp*) Mit Hilfe von diesem Modellierungelement werden *ThingTypes* miteinander in Beziehung gesetzt. *ConnectionTypes* entsprechen weitgehend den einfachen Assoziationen in *UML* und können Multiplizitäten, Bezeichnung, Beschreibung und Rollen enthalten. Neben allgemeinen *ConnectionTypes* bietet *CPL* in seiner aktuellen Version zwei vordefinierte Beziehungstypen: *Synonym* und *Generalisierung/Spezialisierung*.

OperationType (*Operationstyp*) Dieses Element beschreibt – wie der Name bereits andeutet – Operationen, die in einem Untersuchungsbereich angeboten werden. Sie sind mit *ThingType*-Elementen verknüpft, die als Parameter, Dienstnehmer oder Dienstanbieter dienen können. Im Vergleich zu *UML*-Operationen sind sie nicht einem *ThingType* untergeordnet. Sie sind eigenständig und können verschiedenen *ThingTypes* in unterschiedlichen Rollen (Dienstnehmer oder -anbieter) zugeordnet werden.

CooperationType (*Kooperationstyp*) Dieses Modellierungselement ermöglicht es, Prozesse eines Untersuchungsbereichs zu modellieren. Ein *CooperationType* verkapselt ein oder mehrere *OperationTypes*[21] und hat Vor- und Nachbedingungen. Eine Verknüpfung der Nachbedingungen

[20] für weiterführende Informationen siehe auch Kapitel 4.1.3

[21] Laufen *OperationTypes* zeitlich versetzt ab, so werden sie in verschiedene *CooperationTypes* verpackt und über Vor- und Nachbedingungen verknüpft. Spielt die Reihenfolge der Ausführung dagegen keine Rolle, können sie von einem *CooperationType* verkapselt werden, um diese Unabhängigkeit zu signalisieren.

von (mehreren) *CooperationTypes* mit Vorbedingung(en) eines weiteren *CooperationTypes* dient zur Herstellung einer Ablaufsfolge. Ein Beispiel eines *CooperationTypes* Bestellung bearbeiten ist in der Abbildung 2.4 gezeigt. Das Modellierungselement *CooperationType* in *CPL* hat eine gewisse Ähnlichkeit zu den Anwendungsfällen in *UML*.

Die *CPL*-Modellierungselemente *ThingType* und *ConnectionType* werden für die statische Modellierung des Untersuchungsbereichs benutzt. *OperationType* und *CooperationType* sind für die Abbildung seiner dynamischen Aspekte bestimmt. Die Verknüpfung der statischen mit den dynamischen Hauptmodellierungselemente erfolgt über die Möglichkeit, einen *OperationType* mit *ThingTypes* zu verknüpfen.

Die *Conceptual Predesign Language* bietet zwei Darstellungsformen an: graphisch und textuell/tabellarisch. Ein kleiner Einblick in die graphische Darstellung ist bereits in der Abbildung 2.4 gegeben. Diese Darstellungsform wurde zeitlich etwas später als Ergänzung zur tabellarischen Darstellung entwickelt. Primär wird ein mit *CPL* modellierter Untersuchungsbereich glossarartig, in tabellarischer Form abgebildet. Dabei entsteht für jedes Hauptmodellierungselement aus der obigen Auflistung eine separate Tabelle, die in den Zeilen einzelne Elemente dieses Typs aus dem Untersuchungsbereich und in den Spalten die Eigenschaften dieser Elemente enthält.

Ein einführendes Beispiel, das einen kleinen Ausschnitt aus den Tabellen für statische Elemente des Fachbereichs Mathematik und Informatik an der Philipps-Universität Marburg darstellt, ist in den Tabellen 2.1 und 2.2 gezeigt. Dieses Modell entstand bei der Analyse des Software-Projekts Scheinverwaltung am Fachbereich.

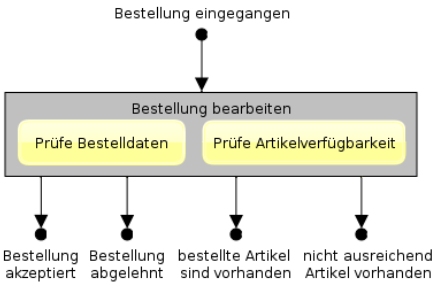


ABBILDUNG 2.4: Beispiel eines *CooperationTypes*

Diese glossarartige, tabellarische Darstellung hat sich im Vergleich zu der graphischen Darstellung als vorteilhaft bei der Kommunikation mit den Fachverantwortlichen erwiesen [MK06]. Einer der Gründe dafür ist die Verbreitung von Tabellenkalkulationsanwendungen in fast allen Bereichen der Wirtschaft. Damit gehört der Umgang mit vergleichbaren Tabellen zum Alltag der meisten Benutzer eines zukünftigen Software-Systems. Bei Bedarf ist es möglich, dass die Benutzer direkt Tabellenkalkulations-Anwendungen zum Erstellen und Editieren von *CPL*-Modellen benutzen und somit in einer

Name	Beispiel	Wertebereich	Synonyme Id	Beschreibung	Quelle
Veranstaltung	Praktische Informatik			Veranstaltung steht stellvertrettend für mehrere Lehrformen (Vorlesung, Seminar, ...) an einer Universität. Sie wird in einem festgelegten Zeitraum von Dozenten angeboten und von Studenten besucht.	Anforderungsdokument, Kapitel 2.2.1
Dozent	Prof. Dr. M. Mustermann			Berechtigte Personen, die eine Veranstalltung an einer Universität durführen dürfen.	Anforderungsdokument, Kapitel 2.2.1
Student				Eine an der Universität eingeschriebene Person.	Anforderungsdokument, Kapitel 2.1.1
Matrikelnummer	1300075	5-stellie natürliche Zahl		Eine eindeutige identifikatiosnummer eines Studenten.	Anforderungsdokument, Kapitel 2.1.1
Schein			Modulbescheinigung	Ein Leistungsnachweis, der einem Studenten eine erfolgreiche Teilnahmen an einer Veranstaltung bescheinigt.	Anforderungsdokument, Kapitel 2.3.4
Modulbescheinigung			Schein	Siehe den Dingtyp Schein	Anforderungsdokument, Kapitel 2.3.5

TABELLE 2.1: Beispiel eines *CPL*-Modells: Ausschnitt der *ThingType*-Tabelle

ihnen vertrauten Umgebung arbeiten können. Im Unterschied zu der graphischen Darstellung oder anderen konzeptuellen Modellierungssprachen wie *UML* kann durch Ein- und Ausblenden von bestimmten Spalten (wie zum Beispiel Vergegenständlichung in der *ConntectionType*-Tabelle) die Detailliertheit und Komplexität des Modells unkompliziert und fein granular an verschiedene Betrachter-Gruppen angepasst werden.

Auf der anderen Seite ist die glossarartige Darstellung ein Schritt in die Richtung der Formalisierung von Anforderungen und bietet Vorteile im Vergleich zu den natürlichsprachlich erfassten Anforderungsdokumenten. Die Elemente eines Untersuchungsbereichs werden bei ihrer Aufnahme in die *CPL*-Tabelle bereits klassifiziert und durch Ausfüllen von Spalten charakterisiert. So stehen wichtige Informationen zu einem Element in einer Zeile und sind nicht über ein Anforderungsdokument verstreut. *CPL*-Tabellen sind übersichtlicher und ermöglichen schnellen Zugriff auf die Elemente eines Untersuchungsbereichs. Auch Synonyme fallen in diesem Format schneller auf, da fast identische Charakterisierungen schneller identifiziert werden können.

Den Software-Entwicklern kommt es entgegen, dass die Vorstrukturierung mit den bekannten Elementen der späteren Modellierung weitgehend

Name	Vergegen-ständlichung	beteiligter Dingtyp	Perspektive	Kardinalität	Rolle	Quelle
hat		Student	wird identifiziert durch	1		Anforderungs-dokument, Kapitel 2.2.1-1
		Matrikelnummer	zugeordnet zu	1		
betreut		Dozent	betreut	1..3	Betreuer	Anforderungs-dokument, Kapitel 2.2.1-1
		Veranstaltung	wird betreut von	1..N		
ist		Teilnehmer	ist ein	1		Anforderungs-dokument, Kapitel 2.2.1-1
		Student		1		
besucht	Teilnahme	Teilnehmer	besucht	0..N	Besucher	Anforderungs-dokument, Kapitel 2.2.1-1
		Veranstaltung	wird besucht von	3..N		

TABELLE 2.2: Beispiel eines *CPL*-Modells: Ausschnitt der *ConnectionType*-Tabelle

kompatibel ist[22] und die entsprechende *UML*-Modelle semi-automatisch mit den Werkzeugen von *CPM* abgeleitet werden können [Ker01].

Die Zugänglichkeit von *CPL* einerseits und die von der Sprache angebotenen Strukturierungsmöglichkeiten andererseits machen sie sowohl für die Fachverantwortlichen als auch für die Software-Entwickler attraktiv. Zusätzlich dazu eröffnen die Einbettung von *CPL* in den *NIBA*-Prozess und seine linguistische Werkzeugunterstützung neue Möglichkeiten bei der Durchführung der Anforderungsanalyse.

2.1.4 Modell-zu-Modell-Transformationen

Zu den wichtigsten Mitteln, die während der Modellierung eines Software-Systems angewendet werden, zählen verschiedene Transformationen. Im einfachsten und häufigsten Fall handelt es sich dabei um *Modellmodifikationen*, die jedes Mal angewendet werden, wenn ein Modell verändert wird, zum Beispiel um ein neues Element aufzunehmen. Modellmodifikationen sind eine Untergruppe der Modell-zu-Modell-Transformationen mit der Besonderheit, dass das Quell-Modell gleichzeitig als Ziel-Modell dient.

Im Allgemeinen handelt es sich bei *Modell-zu-Modell*-Transformationen um Transformationen, bei denen sowohl die Quelle als auch ihr Ergebnis ein Modell ist. Im Vergleich zu den *Modell-zu-Text*-Transformationen[23],

[22] So können *ThingTypes* später als Klassen oder Attribute, *ConnectionsTypes* als Assoziationen, *OperationTypes* und *CooperationTypes* als Operationen in *UML* modelliert werden.

[23] Genauer betrachtet, sind Modell-zu-Text-Transformationen eine Unterklasse der *Modell-zu-Modell-Transformationen*, deren Ergebnis eine textuelle Darstellungsform

die bereits seit Jahren erforscht werden und in der Regel Bestandteil von Modellierungswerkzeugen sind, ist die Forschung im Bereich der Modell-zu-Modell-Transformationen relativ jung.

In der Software-Technik lieferte insbesondere *MDA* als eine Technik, die explizit auf Modell-zu-Modell-Transformationen beruht, einen Anstoß für die Untersuchung der Ansätze, mit denen solche Transformationen festgelegt und durchgeführt werden können. Zu dem Zeitpunkt, als die *OMG* den *MDA*-Ansatz vorstellte, herrschte ein Mangel an geeigneten Transformationsansätzen. Dieser Umstand hat zu einer Ausschreibung geführt, zu der mehrere Entwürfe eines Transformationsmechanismus eingereicht wurden und nach der in einem Review- und Auswahl-Prozess [GGKH03] der entsprechende *OMG*-Standard unter dem Namen *MOF QVT* ausgewählt und veröffentlicht wurde. [OMG05a].

Bereits der Name des Standards (*MOF QVT*) betont, dass sich die Vorgabe des Transformationsmechanismus ausschließlich auf *MOF*-konforme Modelle[24] bezieht, was zur Folge hat, dass die entsprechenden Meta-Modelle Ausprägungen von *MOF* sind. Das Akronym *QVT* steht für *Query/View/Transformation*. So wird in dem Standard neben einem Verfahren für die Durchführung von Modell-zu-Modell-Transformationen zusätzlich die Art geregelt, wie *Anfragen* (*Query*) an die Modelle formuliert und wie die *Sichten* (*View*) auf diese definiert werden können. Sowohl Anfragen als auch Sichten werden bei der Beschreibung von Transformationen hauptsächlich als Hilfsmittel betrachtet und werden im Standard diesen untergeordnet.

Query (*Anfrage*) Eine Anfrage ermöglicht es, entsprechend einem vorgegebenen Auswahlkriterium eine Menge von Elementen eines Eingabe-Modells zu erhalten. Als eine geeignete Sprache, um diese Auswahlkritterien an *MOF*-konforme Modelle zu formulieren, hat sich *OCL* erwiesen[25] [GGKH03]. Die *OCL*-Anweisung model.objectsOfType(Class) ist ein einfaches Beispiel für eine Anfrage, die einzig die Klassen des Modells model liefert. Alternativ kann auch ein Boolean-Wert als Ergebnis einer Anfrage zurückgegeben werden.

(z.B. mit Programmiersprachen wie *Java*) besitzt. Allerdings werden auf Grund des unterschiedlichen Entwicklungsstands und der verschiedenen Techniken, die für die Transformationen angewendet werden, diese zwei Arten von Transformationen in der Forschung oft unabhängig voneinander betrachtet.

[24] Siehe auch Kapitel 2.1.2

[25] Ursprünglich für die Verarbeitung von Modellen, die auf *einem* Meta-Modell basieren, entwickelt, musste *OCL* für ihre Verwendung im Rahmen des *MOF QVT*-Standards geringfügig erweitert werden, um *gleichzeitig* mit auf verschiedenen Meta-Modellen aufbauenden Modellen agieren zu können.

View (*Sicht*) Eine Sicht legt fest, wie das Ergebnis in dem Ausgabe-Modell aussehen soll. Dadurch verknüpft eine Sicht das Ein- und Ausgabe-Modell miteinander. Als Eingabe-Modell kann auch ein durch die Anfrage definierter Auschnitt dienen.

Transformation *MOF QVT*-Transformationen beschreiben, wie die Überführung eines Eingabe-Modells in die vordefinierte Sicht – ein Ausgabe-Modell – durchgeführt werden soll.

Für Transformationen sieht *MOF QVT* einen *hybriden* Ansatz vor. Zusammenfassend betrachtet wird bei den Modell-zu-Modell Transformationen zwischen den folgenden Transformationsarten unterschieden [CH03]:

- direkte Manipulation
- relationaler Ansatz
- Graphentransformation
- Struktur-getriebener Ansatz
- beliebige Kombinationen dieser Ansätze im Rahmen eines hybriden Ansatzes.

Verfahren, die auf der *direkten Manipulation* aufbauen, erlauben einen direkten Zugriff auf die Modell-Elemente. Die Transformationsregeln[26] werden in diesem Fall typischerweise *imperativ* definiert. Dieser Transformationsansatz kann als die niedrigste Ebene bei der Definition von Transformationen betrachtet werden. Sie sind *unidirektional*, da jede Richtung einzeln implementiert werden muss. Wegen ihrer imperativen Grundlage sind sie nicht frei von *Seiteneffekten*.

Relationale Ansätze ermöglichen es dagegen, Transformationen *deklarativ* zu beschreiben. Die *Relationale Algebra* bildet die Grundlage für diese Ansätze. Verallgemeinert betrachtet werden Transformationen in Form von Relationen zwischen den Elementen des Eingabe- und Ausgabe-Modells definiert, die durch Einschränkungen auf gewünschte Situationen angepasst werden können. So definierte Transformationen tragen einen deklarativen Charakter und sind nicht direkt ausführbar. Um Modellelemente erzeugen zu können, werden oft logische Programmiersprachen eingesetzt. Im Allgemeinen sind sie *bidirektional*, können aber auf eine Richtung eingeschränkt werden. Sie sind aufgrund ihrer deklarativen Definition frei von *Seiteneffekten*.

Graphentransformationen basieren auf der mathematischen Grundla-

[26] In diesem Fall handelt es sich trotz der Bezeichnung weniger um Regeln als um in Code gegossene Abläufe.

ge der *Graphentheorie*. Eine Transformation wird mit Hilfe von mindestens zwei Graph-Mustern definiert. In dem Eingabe-Modell wird nach dem *Quellmuster (LHS)* gesucht, das in das *Zielmuster (RHS)* des Ausgabe-Modells überführt wird[27]. Oft wird zusätzlich ein *Negativ-Muster (NAC)*[28] angegeben, das die Anwendung einer Transformation unterbindet, falls dieses Muster in dem Eingabe-Modell gefunden wurde. Bei Graphentransformationen handelt es sich um mächtige, deklarative Ansätze, die allerdings auch die höchste Komplexität aufweisen. Insbesondere die Bestimmung, ob eine Modell-Transformation sicher terminiert wird und wie sich die Reihenfolge der Regeln auf das Ergebnis auswirken kann, gehören zu den Problemstellen dieses Ansatzes [CH03]. Sie sind analog zu den relationalen Ansätzen frei von *Seiteneffekten*.

Zu den *Struktur-getriebenen Ansätzen* gehören pragmatische Ansätze, die für spezielle, in der Praxis oft benötigte Situationen entwickelt wurden. Beispiele dafür sind Abbildungen von *Java Enterprise Beans (EJB)* oder *UML*-Modelle auf ein *Relationales Datenbank-Schema*. Diese Transformationen gehen von einem *hierarchischen Aufbau* der Modelle aus und laufen in der Regel in *zwei Phasen* ab. Zunächst wird die hierarchische Struktur abgebildet, dann werden die Werte der Modellelemente wie zum Beispiel ihre Bezeichner übertragen. Sie sind *unidirektional* und die speziellen Lösungen werden gezielt frei von *Seiteneffekten* definiert. Inwieweit sich dieser *Struktur-getriebener Ansatz* auf allgemeine Modell-zu-Modell-Transformationen erweitern lässt, wurde noch wenig untersucht [CH03].

Zu den *hybriden Ansätzen* zählen beliebige Kombinationen der oben vorgestellten Verfahren. So baut die hybride Lösung von *QVT* auf einem zweistufigen deklarativen Teil (*QVT Relations* und *QVT Core*) auf. Zusätzlich werden sie um zwei imperative Teile (*Operational Mappings Language* und *Black Box Implementations*) ergänzt, die eine direkte Manipulation von Modellelementen erlauben. Primär wird in dem Standard auf den Relationalen Ansatz gesetzt, der mittels der Sprache zur Beschreibung von Transformationen *QVT Relations* umgesetzt ist. Sie kann als eine Schnittstelle nach außen zum Verfasser der Transformationen angesehen werden. Intern werden die relational definierten Transformationen auf das Kernmodul *QVT Core* abgebildet. Dadurch wird die Repräsentationsschicht von der technischen Darstellung der Transformationen abgetrennt. *QVT Core* dient als zentrale interne Schnittstelle für die Werkzeuge, die zum Beispiel Transformationen ausführbar machen können.

[27] Die Abkürzungen *LHS* und *RHS* stehen für *left hand side* bzw. *rigth hand side*.
[28] *NAC* steht für *negative application condition*.

Imperative Teile des Strandards haben Schnittstellen sowohl zu der Sprache *QVT Relations* als auch direkt zu dem Kernmodul *QVT Core* und stehen somit dem Verfasser der Transformationen als auch für technische Weiterverarbeitung zur Verfügung. *Operational Mappings Language* ist eine zu dem Inhalt des Standards gehörende imperative Sprache. *Black Box Implementations* ist dagegen eine Schnittstelle nach außen. Sie kann für beliebige Sprach-Erweiterungen benutzt werden. Sie bringt einerseits enorme Flexibilität mit sich, andererseits auch die Gefahr, dass die Transparenz des im Mittelpunkt stehenden relationalen Ansatzes darunter leidet, was sich durch Seiteneffekte oder in Nicht-Determiniertheit eines Transformationsprozesses äußern kann. Aus diesem Grund soll *Black Box Implementations* – falls überhaupt erforderlich – mit Bedacht eingesetzt werden.

MOF QVT ist zur Zeit der einzige durch eine große anerkannte Institution (*OMG*) standardisierte Ansatz im Bereich der Modell-zu-Modell-Transformationen. Da es sich dabei um ein Rahmenwerk bzw. einen Entwurf handelt, exisitieren für diesen Standard verschiedene Implementierungen, die nicht nur eine Deklaration – dies ist auch mit den Mitteln von *QVT* direkt möglich – sondern auch eine Ausführung der Transformation erlauben.

Die Trennung zwischen der standardisierten Deklaration der Transformationsregeln und Implementierungen für die Ausführung der Regeln bringt mehrere Vorteile mit sich. Zum Beispiel werden bereits seit dem Erscheinen von *MOF QVT* im Rahmen der *Ontology Definition Metamodel*-Initiative [OMG06b] rein relationale Regeln für Transformationen zwischen verschiedenen Ontologie-Sprachen wie *OWL*, *Description Logic* und *UML* definiert. Obwohl zur Zeit noch keine Implementierung existiert, die den Standard vollständig umsetzt, können relationale Deklarationen benutzt werden, um ihre Machbarkeit zu zeigen und formale Untersuchungen ihrer Eigenschaften unabhängig von einer Implementierung durchzuführen [BVEL04].

Vorhaben, bei denen eine Ausführung der Regeln ebenfalls eine wichtige Rolle spielt und zu denen diese Arbeit gehört, können auf den bereits existierenden Implementierungen des Standards aufbauen. Auch wenn diese noch nicht den endgültigen Stand erreicht haben und somit leichte Abweichungen aufweisen, ermöglicht die Ausrichtung an dem *QVT*-Standard einen Wechsel der verwendeten Implementierung. Aufgrund der aktuellen intensiven Entwicklung im Bereich der Werkzeuge für den *QVT*-konformen Transformationsansatz könnte ein späterer Wechsel sinnvoll sein, wenn dadurch bessere Konformität oder Qualität der Transformationsausführung erreicht wird. Da die Werkzeuge sich weitgehend in dem Rahmen des *QVT*-Standards bewegen, sollte sich der Aufwand dieser Umstellung in einem

vertretbaren Rahmen halten[29].

Transformationen mit *ATL*

Ursprünglich wurde die Zielsetzung für den *ATL*-Transformationsmechanismus unabhängig von dem *MOF QVT*-Standard definiert[30]. Diese – trotz vieler Gemeinsamkeiten parallele, von dem Standard vorerst abgekoppelte Entwicklung – wurde dadurch gerechtfertigt, dass *ATL* bereits vor der endgültigen Fertigstellung von *MOF QVT* eine Standard-ähnliche Implementierung anbieten konnte, die als eine der ersten eine Möglichkeit für die Ausführung von Transformationsregeln umgesetzt hat [BJRV03]. Die mit diesem Vorsprung erkauften Abweichungen von dem inzwischen veröffentlichten *MOF QVT*-Standard werden inzwischen untersucht, und erste Ansätze für die Anpassung an den Standard wurden erarbeitet [JK06].

Relationale, deklarative *ATL*-Regeln bestehen aus einer Regel-Deklaration und zwei Abschnitten from und to (siehe Listing 2.1). Der from-Abschnitt legt fest, welche Elemente von der Transformation betroffen sind. Diese Vorauswahl stellt eine Anfrage im Sinne von *MOF QVT* dar. Der Abschnitt to beschreibt, worauf die Elemente aus der from-Anfrage im Ausgabe-Modell abgebildet werden. In dem vorgestellten und in der Literatur oft benutzten Beispiel wird ein atomares[31] *UML*-Attribut auf eine Spalte einer relationalen Datenbanktabelle abgebildet.

Als Ausgangsbasis für die Regeln dieser Art dienen – wie bereits erwähnt – Meta-Modelle der beiden Modelle, die in *Ecore* verfasst sind, wodurch die Sprache *OCL* und auch sonst *OCL*-ähnliche Syntax bei der Definition der Regeln verwendet wird. Die Meta-Modelle dürfen sich unterscheiden, was exogene Transformationen wie im Listing 2.1 erlaubt[32]. Die Aufteilung in from und to implizirt, dass die Regel in *ATL generell gerichtet*

[29] Hauptsächlich müssten bei dieser Umstellung nur die Abweichungen vom Standard, falls vorhanden, überarbeitet werden.

[30] In einigen wenigen Punkten weichen die Zielsetzungen von *ATL* und *QVT* stark voneinander ab. So wurde zum Beispiel bei *QVT* gefordert, dass *nur ein* Eingabe-Modell als Input benutzt werden darf. *ATL*-Entwickler versuchten zunächst *mehrere* Eingabe-Modelle zu erlauben. Auch die Frage, ob ein Eingabe-Modell direkt durch Transformationen manipuliert werden darf, wird von den beiden Ansätzen unterschiedlich gehandhabt. [BJRV03]

[31] Atomar bedeutet in diesem Zusammenhang, dass nur ein einziges Element in dem Attribut gespeichert werden darf und keine Listen von Elementen.

[32] In der Transformation werden die verwendeten Meta-Modelle als Präfix (uml oder relational) vor einem Element des entsprechenden Meta-Modells gesetzt.

ist[33]. Analog zu *MOF QVT* erlaubt *ATL* eine Integration von imperativen Anweisungen, die durch das Schlüsselwort do nach dem to-Block eingeleitet werden.

```
1  rule AtomicAttribute2Column {
2      from
3          attr : uml!Attribute (
4              attr.type.oclIsKindOf(uml!DataType) and
5              not attr.multiValued
6          )
7      to
8          out : relational!Column (
9              name <- attr.name,
10             type <- attr.type
11         )
12 }
```

LISTING 2.1: Beispiel einer deklarativen *ATL*-Transformationsregel (*UML*-Attribut auf eine Spalte in einer relationalen Datenbank-Tabelle)

Für die Darstellung der Transformationsregeln schlägt der *MOF QVT*-Standard eine graphische Darstellung (siehe Abbildung 2.5) vor. Auf der linken Seite zeigt sie Meta-Modell-Elemente des Eingabe-Modells, die transformiert werden sollen. Auf der rechten Seite steht dann die entsprechende Struktur aus dem Ausgabe-Meta-Modell, die durch die Transformation entstehen soll.

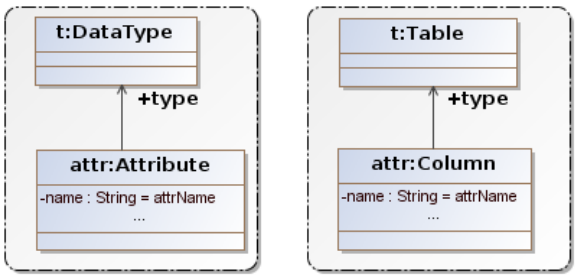


ABBILDUNG 2.5: Visualisierung der AtomicAttribute2Column-Transformationsregel

In diesem Beispiel wird ein Datentyp auf eine Tabelle abgebildet und

[33]In diesem Punkt (und in dem Syntax) unterscheidet sich *ATL* von *MOF QVT*, da der Letztere bidirektionale Regeln erlaubt.

seine Attribute auf Spalten dieser Tabelle, wobei die Namen dieser Spalten den Namen der zugehörigen Attributen entsprechen.

Diese Darstellung ist vor allem dann nützlich, wenn komplexe Strukturen eines Meta-Modells auf das andere Meta-Modell abgebildet werden. Für die Darstellung von imperativen Regeln[34] oder komplizierten Anfragen in dem *from*-Teil ist sie dagegen wenig geeignet. Aus diesem Grund werden in dieser Arbeit für die Darstellung der Transformationsregeln beide Formen verwendet.

[34]Nach dem *MOF QVT*-Standard soll der do-Abschnitt dem Diagramm direkt als Quellcode angehängt werden.

> Es gibt eine Wissenschaft, welche das Seiende als Seiendes untersucht und das demselben an sich Zukommende.
>
> *(Aristoteles* Metaphysik, Buch IV*)*

2.2 Ontologie

Der Begriff Ontologie, der sich aus den griechischen Worten *on* (Partizip Präsens vom dem Wort *einai*, das für *sein* steht) und *logos* zusammensetzt, wurde zum ersten Mal im 17. Jahrhundert verwendet. In seiner Arbeit *Ogdoas Scholastica* stellte Jacob Lorhard von der Universität St. Gallen eine Ontologie auf, die inhaltlich eine protestantische Sicht auf die Welt des frühen 17. Jahrhundert darstellt. Die Besonderheit dieser Arbeit bestand einerseits in der diagrammartigen Darstellung der Ontologie (siehe Abbildung 2.6), andererseits im Aufbau des ontologischen Systems, das Parallelen zu den modernen Ontologien aufweist. [OSU08]

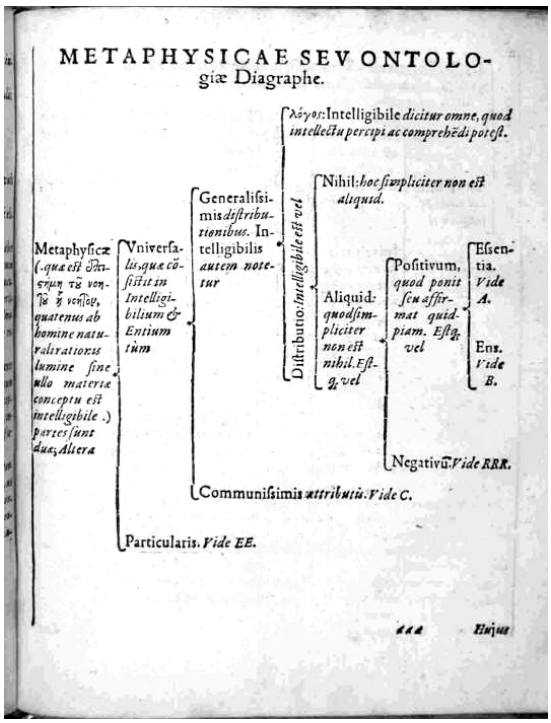


Abbildung 2.6: Ausschnitt aus der Lorhards Ontologie, 1606

ABBILDUNG 2.7: Rudolf Goclenius (Göckel)

Annähernd zeitgleich zu der Arbeit von Jacob Lorhard hat der Philosoph Rudolf Goclenius (Göckel) von der Philipps-Universität Marburg den Begriff Ontologie in sein *Lexicon philosophicum* aufgenommen. Sein Vorschlag, mit diesem Begriff ein Teilgebiet der Metaphysik *metaphysica generalis* zu bezeichnen, hat sich in der Philosophie durchgesetzt. Der Begriff Metaphysik wird mit den Werken von Aristoteles aus dem 4. Jahrhundert verbunden, die von seinen Schülern unter dem Namen *tà metà tà physiká*[35] zusammengefasst wurden. Die allgemeine Metaphysik wird als die Lehre vom Sein oder Seienden bezeichnet. Rudolf Goclenius hat durch seine Zuordnung des Begriffs Ontologie eine Abgrenzung zu dem anderen Teil der Metaphysik *methaphysica specialis* geschaffen, das sich mit Gott und der Seele beschäftigt.

Der Begriff Ontologie verbreitete sich erst später und wurde im 18. Jahrhundert populär. Ein Beitrag dazu leistete der Werk *Philosophia prima sive Ontologia* von Christian Wollf.

2.2.1 Ontologie in der Philosophie

Ontologie ist eine Disziplin der Philosophie und hat zur Aufgabe, die Grundlagen des Seiens zu erforschen. Sie basiert auf Theorien über Beständigkeit und Veränderlichkeit, Identität, Klassifikation und Kausalität und beschäftigt sich mit Fragen wie: Welche Arten von Objekten existieren? Wie unterscheiden sich diese voneinander? Was sind die Eigenschaften eines Gegenstands und in welcher Beziehung stehen sie zu ihm? Welche Eigenschaften können in welchem Ausmaße geändert werden, so dass noch vom gleichen Gegenstand gesprochen werden kann?

Nach Mario Bunge ist Ontologie eine formale Wissenschaft. Sie benutzt mathematische Grundlagen und Logik, um ontologische Theorien abzuleiten. Wie andere Naturwissenschaften beschäftigt sich die Ontologie mit

[35] Der Name stammt von der Einordnung dieser Werke in einem Regal. Die Schüler fassten zuerst alle seine Werke bezüglich Physik zusammen. Arbeiten von Aristoteles, die thematisch über die Physik hinaus gingen, wurden *hinter* den Physik-Werken eingeordnet.

konkreten Gegenständen und der Realität. Beide untersuchen ihre Natur. Während sich zum Beispiel Physik detailliert mit Eigenschaften der Materie auseinander setzt und ihre Theorien empirisch untersucht werden können, abstrahiert Ontologie von diesen Details und konzentriert sich auf bereichsübergreifende Relationen und Eigenschaften, die von der Phase oder dem Zustand ihrer aktueller Existenz unabhängig sind. Die ontologischen Hypothesen werden durch Erkenntnisse aus den Naturwissenschaften bestätigt oder widerlegt. Folgende Hypothesen der Ontologie gelten durch naturwissenschaftliche Erkenntnisse als bestätigt (siehe [Bun77], Seite 16):

M1 Es existiert eine Welt außerhalb des Beobachters.

M2 Die Welt besteht aus Gegenständen.

M3 Form ist eine Eigenschaft der Gegenstände.

M4 Systeme werden aus Gruppen von Gegenständen gebildet.

M5 Jedes System, mit der Ausnahme des Universums, interagiert über bestimte Beziehungen mit anderen Systemen und ist isoliert von diesen in anderen Bereichen.

M6 Jeder Gegenstand ist Veränderungen unterworfen.

M7 Nichts kommt aus dem Nichts und nichts verschwindet im Nichts.

M8 Es existieren Gesetze und alles hält sich an diese Gesetze.

M9 Es existieren mehrere Arten von Gesetzen.

M10 Es existieren mehrere Ebenen der Strukturierung.

Naturwissenschaften nutzen diese Hypothesen bei der Konstruktion der wissenschaftlichen Theorien, beim Entwurf von Werkzeugen, aber auch um ihre Untersuchungsergebnisse zu überprüfen. Als ein Beispiel dafür nennt Mario Bunge die metaphysische Frage nach der Existenz von ultimativer Materie, die zu *Heisenbergs Theorie der Elementarteilchen* geführt hat. Auch weitere Beispiele aus Biologie und Sozialwissenschaften finden sich bei Mario Bunge ([Bun77], Seite 19). Er betont, dass keine scharfe Abgrenzung zwischen Ontologie und den Naturwissenschaften besteht. Viele Begriffe, die einen generellen Charakter besitzen – wie Aggregation, Eigenschaft, Beziehung, Kausalität usw. – wurden teilweise von den jeweiligen Naturwissenschaften adaptiert und bilden ein formales Grundgerüst dieser Wissenschaften. Dies betrifft auch die Informatik. Insbesondere in der Software-Technik spielen die Begriffe wie Aggregation, Klassifikation, Eigenschaft und Beziehung eine zentrale Rolle.

2.2.2 Begriffsübernahme in die Informatik

Verschiedene Schulen in der Philosophie haben sich mit der Ontologie beschäftigt. Für die Übertragung des Begriffs in die Informatik haben nach Berry Smith [Smi04] die Aufsätze mit dem Thema *On What There Is* von Willard Van Orman Quine eine wichtige Rolle gespielt. Sie können als Vorstufe der Ontologie in den Informationssystemen angesehen werden. 1967 zitierte S. H. Mealy die Aufsätze von Quine bei seiner Arbeit über Grundlagen der Datenmodellierung und verwendete den Begriff zum ersten Mal in der Informatik. Mealy beschrieb drei verschiedene Bereiche, die bei der Datenverarbeitung essentiell sind:

- Die reale Welt,
- Vorstellungen darüber in den Gedanken des Menschen,
- Symbole auf Papier oder einem anderen Speichermedium.

In seiner Diskussion dazu schreibt er „This is an issue of ontology, or the question what exists“ (zitiert nach [Smi04], Seite 23). Bei seiner Aufzählung lässt sich eine Variante des bekannten Semiotischen Dreiecks von Ogden und Richards erkennen (siehe Abbildung 2.8). Das Dreieck wird als ein, den Ontologien übergeordnetes, allgemeines Konzept angesehen.

Das Dreieck setzt voraus, dass die reale Welt unabhängig von dem Beobachter existiert und dass die Gegenstände möglicherweise mehrere oder keine Repräsentationen besitzen. Wesentlich dabei ist, dass die Ontologie in diesem Fall von der Vorstellung und den Gedanken des Menschen abhängig ist und einen epistemischen Charakter[36] trägt. Somit wird eine Ontologie in der Informatik in der Regel basierend auf Erfahrungen hergeleitet und wird nicht als a priori[37] gegeben angesehen.

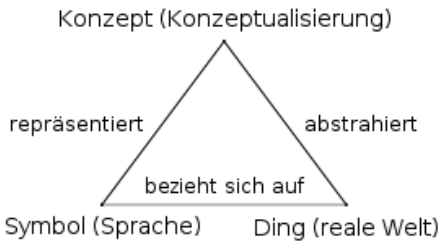


Abbildung 2.8: Semiotisches Dreieck

Durch die Arbeit von Mealy wurde der Begriff Ontologie zuerst im Bereich Datenbanksysteme eingeführt. Es entstanden erste Sprachen, die

[36] *Episteme* ist griechisch für Wissen; *epistemisch* = erkenntnistheoretisch

[37] Was in diesem Zusammenhang bedeuten würde, dass eine Ontologie nicht aus Beobachtungen abgeleitet werden kann, sondern durch Axiome beschrieben wird, die möglicherweise gar nicht bewiesen beziehungsweise widerlegt werden können.

von der physikalischen Darstellung der Daten abstrahieren und den heutigen konzeptuellen Modellierungssprachen zugeordnet werden können. Dazu zählt vor allem das *Entity/Relationship-Modell (E/R-Modell)* [Che76]. Anfangs spielte der Begriff Ontologie in diesem Bereich bei den damals aktuellen Themen (*Drei-Schema-Architektur* und *E/R-Modell*) nur eine untergeordnete Rolle. Laut Smith und Welty [SW01] verlief die Entwicklung zu jener Zeit noch ad hoc. Trotzdem fällt zum Beispiel auf, dass die Grundelemente des ER-Modells: *Entity* und *Relationship* auf die ontologische Annahme zurückgeführt werden können, dass die Struktur der Welt durch die Kategorien Entität und Beziehung beschrieben werden kann.

Der Begriff Ontologie hat sich in der Informatik erst in der Forschung über die *Künstliche Intelligenz* in den neunziger Jahren verbreitet. Der Schwerpunkt der Entwicklung lag auf Systemen, die in der Lage sein sollten, Wissensverarbeitung zu simulieren, indem sie mittels automatischer Schlussfolgerungen neue Erkenntnisse gewinnen und somit als intelligent wirken. Dabei wich der ursprünglich verfolgte Ansatz, Intelligenz allein durch einprogrammierte Verhaltensweisen zu simulieren, einer Kombination aus der Verhaltens- und Umweltnachbildung. Nicht das Verhalten an sich selbst, sondern Verhalten im Kontext einer konkreten Umwelt bestimmen die Intelligenz. Einer der führender Befürworter dieses Perspektivenwechsels – William Clancey – argumentierte: „Das wichtigste Anliegen der Wissensverarbeitung ist die Modellierung von Systemen in der Welt und nicht die Nachbildung, wie die Menschen denken.“[38]

Damit rückte die Modellierung einer Umgebung, in der ein künstlicher Agent agiert, in den Vordergrund und übernahm die Aufgabe, eine Brücke zwischen zwei wichtigen und gleichzeitig sehr unterschiedlichen Komponenten der maschinellen Wissensverarbeitung zu schlagen. Auf der einen Seite steht das Verhalten des Agenten, der zur Problemlösung eingesetzt wird, und auf der anderen der zugrunde liegende Weltausschnitt. Da bei seiner Modellierung von der spezifischen Problemstellung selbst abstrahiert werden sollte, gewannen Ontologieen an dieser Stelle an Bedeutung ([Gua95]). Es entstand eine große Anzahl von Domänen-Ontologien wie zum Beispiel *LKIF Core Ontology*[39] in der Rechtswissenschaft oder *GALEN*[40] in der Medizin.

[38] Original: The primary concern of knowledge engineering is modelling systems in the world, not replicating how people think.
[39] http://www.estrellaproject.org/
[40] http://www.opengalen.org/

Ontologien im *Semantischen Web* – ein Anwendungsbeispiel

Einen weiteren Entwicklungsschub erfuhr Ontologie in der Informatik durch die Vision des *Semantischen Webs* [BLHL01]. Eine entscheidene Erweiterung, die durch diese Vision propagiert wurde, besteht darin, bis dahin zwar maschinenlesbare, aber nur durch Menschen interpretierbare Inhalte im Internet ebenfalls maschinen-interpretierbar zu machen. Praktisch gesehen sollen die Webseiten (und insbesondere Verknüpfungen zwischen diesen) mit semantischen Informationen angereichert werden, die in einer formaler Sprache definiert werden und so den Maschinen – bei dem *Semantischen Web* wird in der Regel von *Agenten* oder intelligenten *Web-Services* gesprochen – zugänglich zu machen.

Der Unterschied zwischen dem Semantischen Web und dem Internet, wie es zur Zeit noch vorwiegend vorgefunden wird, kann anhand der Funktionsweise einer Suchmaschine für Webseiten verdeutlicht werden. Eine Suchanfrage zu dem Stichwort „Jaguar“, läuft zum Beispiel bei Google rein syntaktisch (durch Zeichenkettenvergleich) ab und die Ergebnisse werden entsprechend einer Gewichtung dargestellt, die keine inhaltliche Abhängigkeit von dem Suchbegriff besitzt, sondern auf der Anzahl der zu der Webseite führenden Verweise basiert. Die gefundenen Treffer stehen in diesem Fall für eine Raubkatze, ein Flugzeug-Modell, ein Betriebssystem oder einen Auto-Hersteller usw.

Insgesamt lassen sich aktuell 14 verschiedene Bedeutungen des Wortes identifizieren. In der Regel sind für einen Benutzer die meisten der 72.500.000[41] gefundenen Treffer uninteressant, da er sich zum Beispiel nur über das Raubtier „Jaguar“ informieren wollte. Zusätzlich begünstigt die Gewichtung einer Suchmaschine populäre Bedeutungen eines Suchbegriffs, und sie werden an den Anfang der Trefferliste gesetzt. Recherchen in weniger populären Bereichen werden dadurch zusätzlich erschwert. Eine konventionelle Suchmaschine ist somit ohne zusätzliche Angaben nicht in der Lage, eine Unterscheidung zu treffen, für welche Bedeutung eines Suchbegriffs der Benutzer sich gerade interessiert, und dementsprechend relevante Ergebnisse höher in der Trefferliste zu platzieren.

Um unter anderem eine semantische Suche zu ermöglichen, sieht das *Semantische Web* in seiner Architektur fünf Schichten vor (siehe Abbildung 2.9[42]). Zusätzlich zu den ersten zwei Schichten: *Unicode* und *eXtensible*

[41] Stand Juli 2009

[42] Ausführliche Darstellung der Schichtenarchitektur ist unter http://www.w3.org/2007/03/layerCake.png bei dem W3C-Konsortium abrufbar.

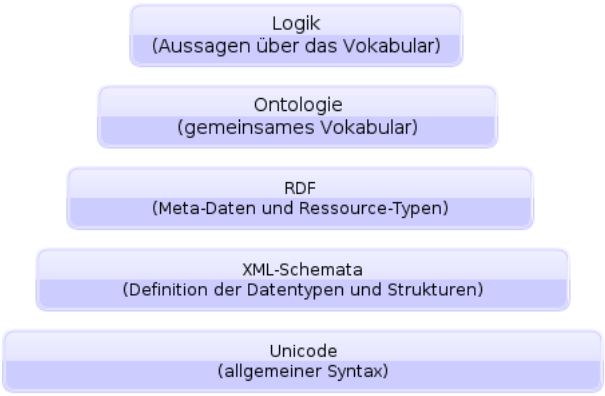


ABBILDUNG 2.9: Ein vereinfachtes Schichtenmodell des *Semantischen Webs*

Markup Language (XML), die die Grundlage jeder Webseite bilden, kommen bei dem *Semantischen Netz* drei weitere Schichten hinzu: *Resource Description Framework (RDF)*, *Ontologie* und *Logik*. *RDF* ermöglicht es, Quellen (Ressources) im Web mit Metadaten in standartisierter Form zu versehen. Zu den Quellen zählt alles, was mit dem *Uniform Resource Identifier (URI)*[43] referenziert werden kann (zum Beispiel Webseiten, Geräte, Dateien). *RDF*-Ausdrücke zur Beschreibung der Metadaten sind ein Tripel der Form *Subjekt-Prädikat-Objekt*. Dabei wird ein durch ein *URI* erreichbares Subjekt durch ein Prädikat (Attribut) charakterisiert und mit dem Wert des Objekts belegt. *RDF*-Ausdrücke können als gerichtete Graphen visualisiert werden. Ein möglicher Graph für das „Jaguar"-Beispiel – basierend auf den Webseiten von *Wikipedia* – ist in der Abbildung 2.10 dargestellt. Mit *URI* referenzierte Elemente werden mit Ovalen, Text mit Rechtecken und die Prädikate als gerichtete Kanten mit Beschriftung dargestellt.

Eine Seite von *Wikipedia* (Subjekt), die eine Beschreibung des Begriffs „Jaguar" enthält, wird durch mehrere Prädikate charakterisiert. Die Objekte können wiederum wie im Fall von http://de.wikipedia.org/Katzen als Subjekte agieren. So lässt sich von diesem Diagramm ablesen, dass es sich bei dem Begriff „Jaguar" in diesem Fall um eine Katze aus der Ordnung der Raubtiere handelt. Die Verknüpfung zwischen den Begriffen „Jaguar" und „Raubtier" ist mit Hilfe dieser Metadaten hergestellt und könnte für die Suche verwendet werden. Allerdings sind die Bezeichner der Prädikate ist eine und ist ein im Allgemeinen frei wählbare, natürlichsprachliche Ausdrücke, die einem Menschen bei der Klassifikation durchaus hilfreich sein können,

[43] Für weitere Informationen siehe http://tools.ietf.org/html/rfc3986.

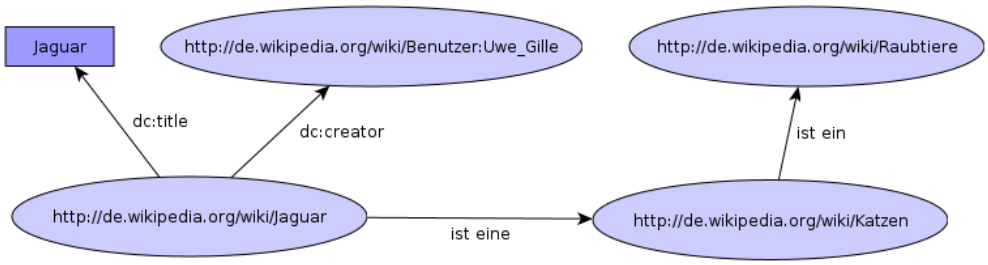


ABBILDUNG 2.10: Ein graphischer *RDF*-Beispiel

eine Suchmaschine aber vor eine in der Regel unlösbare Aufgabe stellen.

An dieser Stelle fehlt ein gemeinsames Vokabular, das bei der Verwendung und Interpretation der Prädikate verwendet wird. Also eine Ontologie (die vierte Schicht in der Abbildung 2.9), in der die zu verwendenden Prädikate und ihre Interpretation festgelegt sind. So sind in Abbildung 2.10 zwei Prädikate mit dem Präfix dc: markiert und signalisieren damit, dass diese Prädikate aus einer Ontologie namens *Doublin Core*[44] stammen und dort definiert sind. *Doublin Core* ist eine vergleichsweise kompakte Ontologie, die sich auf die Klassifizierung von Webseiten konzentriert. Eine Suchmaschine, die unter Berücksichtigung der in *Doublin Core* definierten Prädikate arbeiten würde, wäre zum Beispiel in der Lage, durch maschinelle Auswertung von dc:creator *Wikipedia*-Artikel zu finden, bei denen Uwe Gille als (Co-)Autor agiert.

Inzwischen werden Ontologien in verschiedenen Bereichen der Informatik wie Wissensverarbeitung und -management, Verarbeitung der natürlichen Sprache, Datenbankdesign und -integration, Bio-Informatik, usw. benutzt und haben sich zu einem etablierten Werkzeug entwickelt.

2.2.3 Begriffsklärung und Charakterisierung

So vielfältig die Einsatzmöglichkeiten von Ontologien in der Informatik sind, so unterschiedlich fallen die Auffassungen aus, was unter dem Begriff Ontologie verstanden wird. Folgende Beschreibung bringt die meisten Definitionen auf den kleinsten gemeinsamen Nenner ([UJ99]):

> *Eine Ontologie kann verschiedene Formen annehmen, aber notwendigerweise enthält sie ein Vokabular der Begriffe und eine*

[44] http://www.cs.umd.edu/projects/plus/SHOE/onts/dublin.html

> *Spezifikation von deren Bedeutung. Dies schließt Definitionen und Angaben darüber ein, wie die Konzepte zusammenhängen, die zusammen genommen der Domäne eine Struktur aufprägen und mögliche Interpretationen dieser Begriffe einschränken.*[45]

Interessant ist an dieser Stelle zu bemerken, dass Uschold et al. von *einer* Ontologie sprechen und nicht von *der* Ontologie, wie das in der Philosophie der Fall ist. Während seiner Übertragung in die Informatik hat gleichzeitig zum inhaltlichen ein grammatikalischer Wandel in der Verwendung des Begriffs stattgefunden. Eine Ontologie in der Informatik ist keine Disziplin mehr, die abhängig von der philosophischen Schule ihre Beschreibung der Welt aufstellt. Sie wird in der Informatik als ein Artefakt betrachtet, das in der Regel an eine Domäne gebunden ist und sich mit deren Struktur beschäftigt.

Eine Untersuchung von Guarino und Giaretta ([GG95]) liefert einen Überblick, in welchen Ausprägungen der Begriff Ontologie im Bereich der Wissensverarbeitung (*Knowledge Engineering*) verwendet wird. Die Autoren identifizierten neben der philosophischen insgesamt sechs weitere unterschiedliche Interpretationen, die nach einer gründlichen Analyse durch die Autoren zu den folgenden drei Ausprägungen zusammengefasst wurden:

1. Ontologie als eine Repräsentation eines konzeptuellen Systems, dass durch seine logischen Eigenschaften charakterisiert wird.
2. Ontologie als eine Spezifikation einer ontologischen Verpflichtung[46].
3. Ontologie als ein Synonym für Konzeptualisierung.

Der letzte Punkt dieser Auflistung erinnert visuell stark an die bekannte und sehr oft zitierte Ontologie-Definition von Gruber [Gru93a]: „*Eine Ontologie ist eine explizite Spezifikation einer Konzeptualisierung*“[47]. Die genaue Bedeutung dieser Definition hängt stark von der Interpretation der Begriffe Spezifikation und Konzeptualisierung ab.

Eine Konzeptualisierung kann sowohl extensional (durch eine Auflistung von Entitäten, die einem Konzept oder einer Relation zugeordnet werden) als auch intensional (durch Auflistung von charakterisierenden Eigen-

[45] Original: An Ontology may take a variety of forms, but necessarily it will include a vocabulary of terms, and some specifications of their meaning. This includes definitions and an indication of how concepts are interrelated which collectively impose a structure on the domain and constrain the possible interpretations of terms.

[46] Gebräuchlicher ist der englische Begriff *ontological commitment*. Gemeint ist eine Verständigung auf ein gemeinsames Vokabular und die zugehörige Interpretation.

[47] Original: An ontology is an explicit specification of a conceptualization.

schaften) beschrieben werden. Der Unterschied zwischen den beiden Beschreibungsformen kann anhand des folgenden Beispiels von Genesereth und Nilsson ([GN87]) veranschaulicht werden:

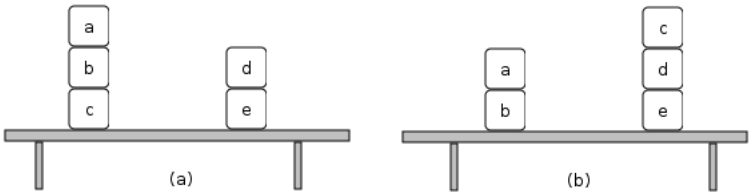


ABBILDUNG 2.11: Zwei verschiedene Konzeptualisierungen?

Die in Abbildung 2.11 dargestellte Konzeptualisierung enthält Blöcke a bis e, die auf einem Tisch platziert und übereinander gestapelt werden können. Das kann durch Relationen wie on (für Blöcke, die aufeinander liegen) und table (für Blöcke, die auf dem Tisch liegen) ausgedrückt werden. Dabei handelt es sich bei on um eine binäre und bei table um eine unäre Relation. Extensional bedeutet, dass in diesem Fall eine Relation wie zum Beispiel table explizit durch die Menge der Blöcke definiert wird. Im Fall a) ist das die Menge $\{c, e\}$ und im Fall b) – $\{b, e\}$. Bei der extensionalen Betrachtung handelt es sich also bei den in Abbildung 2.11 dargestellten Beispielen um zwei unterschiedliche Konzeptualisierungen.

Diese Sicht auf Konzeptualisierung ist allerdings für eine Ontologie-Definition problematisch, da sie durch die extensionale Betrachtungsweise eine Momentaufnahme des Untersuchungsbereichs darstellt.[48] Das impliziert außerdem, dass eine Veränderung im System in der Regel eine neue Definition der Relationen nach sich zieht. Wie in dem obigen Beispiel hängt die Bedeutung der Relation table von dem jeweiligen Zustand des Untersuchungsbereichs ab. Dieses Verhalten würde, wenn eine Ontologie auf einer extensional beschriebenen Konzeptualisierung basieren würde, zu einer sehr ungewönlichen Definition des Ontolgie-Begriffs führen, die der tatsächlichen Verwendung des Begriffs fremd wäre.

Im Falle einer intensional beschriebenen Konzeptualisierung werden Konzepte und Relationen durch ihre charakteristischen Eigenschaften und nicht durch die Angabe der Mengen von Elementen festgelegt. Zum Beispiel kann die Relation table durch die Eigenschaft charakterisiert werden, dass sich ein Block und der Tisch berühren, ohne dass die Menge der betroffenen Blöcke festgelegt wird. Dadurch nimmt sie eine semantische Bedeutung an,

[48] Für eine ausführliche Diskussion siehe auch [Gua97].

die auch intuitiv dieser Relation zugeordnet wird. Die Beispiele a) und b) würden in diesem Fall der gleichen Konzeptualisierung angehören, da sich die Definition von table durch das Vertauschen der Blöcke auf dem Tisch nicht mehr ändert. Sie könnten als zwei unterschiedliche Zustände einer und derselben Konzeptualisierung betrachtet werden. Formal kann eine intensional beschriebene Konzeptualisierung zum Beispiel mit Hilfe von der *Montague*-Semantik[49] abgebildet werden. Folgende Definition (siehe auch [Gui05]) legt den Begriff Konzeptualisierung für diese Arbeit fest:

Definition 3 (Konzeptualisierung) *Eine Konzeptualisierung ist eine intensionale, semantische Struktur, bestehend aus einer Menge von Objekten (Konzepten) einer Domäne und einer Menge von in dieser Domäne erlaubten n-stelligen Relationen.*

Konzepte und Relationen tragen in diesem Fall selbst semantische Information über den Untersuchungsbereich. Eine Ontologie wird in der Regel von mehreren, unterschiedlichen Gruppen angehörigen Aktoren (Agenten, Personen) benutzt. Eine vereinheitlichte und konsistente Benutzung der Konzepte und Relationen ist eine wichtige Voraussetzung, um eine Ontologie zweckmäßig verwenden zu können und von ihr zu profitieren. Das macht die Einführung des Begriffs *ontologische Verpflichtung* notwendig. Entsprechend Gruber [Gru93a] wird ontologische Verpflichtung wie folgt definiert:

Definition 4 (Ontologische Verpflichtung) [50] *Eine ontologische Verpflichtung ist eine Vereinbarung*[51]*, ein gemeinsames Vokabular in einer kohärenten und konsistenten Weise zu benutzen.*

Ein weiterer wichtiger Begriff in der Ontologie-Definition von Gruber (siehe Seite 35) ist *explizite Spezifikation*. Wird dieser Begriff als eine Art formale Beschreibung (analog zu einer Spezifikation eines Algorithmus) der

[49] Die *Montague*-Semantik zeigt auf, wie Syntax und Semantik mithilfe der Methoden der mathematischen Logik (*Intensionale Logik*) systematisch verbunden werden können.

[50] Englisch: ontological commitment.

[51] Diese Vereinbahrung kann zwischen unterschiedlichen Arten der Beteiligten getroffen werden. In dem Bereich der Künstlichen Intelligenz sind das in der Regel Agenten. Beim *Semantischen Web* ist das eine Kooperation zwischen Menschen, die ihre Inhalte mit *Tags* versehen und (Such-)Maschinen, die diese auswerten. In der Software-Technik würde sich der Fokus mehr in die Richtung der Kommunikation zwischen Menschen wie zum Beispiel Kunden und Entwicklern verschieben.

Konzepte und Beziehungen verstanden, die in einer Konzeptualisierung vorkommen können, dann ist eine Ontologie kein Synonym einer Konzeptualisierung (siehe Seite 35). Sie unterscheidet sich nach dieser Definition sehr von der Ausprägung in dem 3. Punkt in dieser Auflistung. Die Definition von Gruber entspricht genau betrachtet auch nicht den ersten zwei Ausprägungen, da sie den Begriff Ontologie allgemeiner definiert, als er tatsächlich in den Punkten 1. und 2. verwendet wird. Laut Guarino und Giaretta kann die Definition von Gruber aber so angepasst werden, dass sie dem am meisten verbreiteten Gebrauch des Begriffs Ontologie in der Informatik entspricht. Dies geschieht durch eine Einschränkung der Art, dass eine Ontologie als eine partielle Spezifikation einer Konzeptualisierung bezeichnet wird.

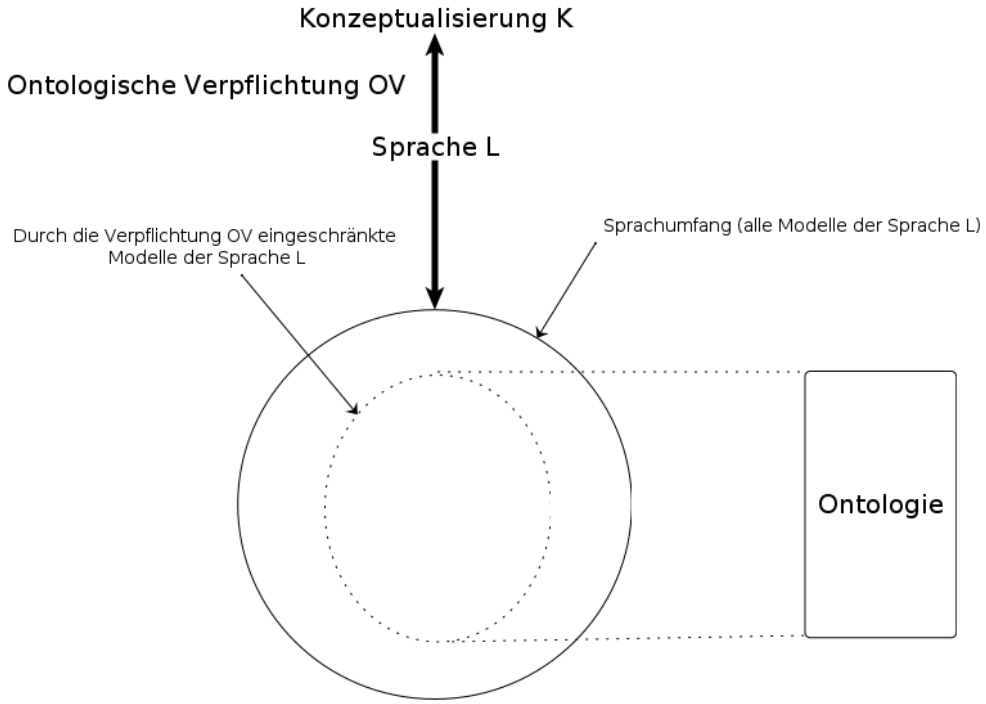


Abbildung 2.12: Zusammenhang der Begriffe: Konzeptualisierung, Ontologie und ontologische Verpflichtung nach [Gua98]

An dieser Stelle spielt die zur Spezifikation verwendete Sprache im Zusammenhang mit der ontologischen Verpflichtung eine wichtige Rolle. Eine Spezifikation einer Konzeptualisierung erfolgt in einer Sprache und ist somit

nicht sprachunabhängig. Der Umfang einer Sprache[52] legt fest, welche Sachverhalte mit ihr ausgedrückt werden können. Dies führt dazu, dass zwei in unterschiedlichen Sprachen verfasste Spezifikationen der gleichen Konzeptualisierung im Allgemeinen zu zwei zwar ähnlichen, aber unterschiedlichen Ontologien führen. Den Zusammenhang der hier eingeführten Begriffe stellt die Abbildung 2.12 dar.

Ergänzend zu der Definition des Ontologie-Begriffs von Gruber und der darauf folgenden Diskussion in diesem Kapitel, die für Charakterisierung und Verständnis des Ontologie-Begriffs eine wichtige Role spielt, kann Ontologie alternativ über ihre Aufgabe eingeführt werden. Folgende Definition von Sowa [Sow00] betont diesen Aspekt:

Definition 5 (Ontologie) [53] *Der Gegenstand einer Ontologie ist die Untersuchung der Kategorien der Dinge, die existieren oder in irgend einem Untersuchungsbereich D existieren können. Das Ergebnis einer solchen Untersuchung – genannt Ontologie – ist ein Katalog der Typen von Dingen, von denen eine Person annimmt, dass sie in D existieren und dabei eine Sprache L benutzt, um über D zu sprechen. Die Typen dieser Ontologie repräsentieren die Prädikate, Wortbedeutungen, Konzept- und Relationstypen der Sprache L, wenn sie benutzt wird, um über Themen im Bereich D zu diskutieren.*

Aufbau von Ontologien

Bedingt durch vielfältige Einsatzbereiche sind Ontologien bezüglich ihrer Konzeption und Umsetzung unterschiedlich aufgebaut. Nach Calero at al. [CRP06] lassen sich im Aufbau der meisten Ontologien die folgenden, grundlegenden Bestandteile identifizieren:

Konzept Ein Konzept ist eine gedankliche Zusammenfassung von Gegenständen oder vergegenständlichten Sachverhalten[54], die sich durch ge-

[52] Sie werden bei den formalen Sprachen, die für die Spezifikation einer Konzeptualisierung in der Regel verwendet werden, in Form eines Meta-Modells definiert.

[53] The subject of ontology is the study of the categories of things that exist or may exist in some domain. The product of such a study, called an ontology, is a catalog of the types of things that are assumed to exist in a domain of interest D from the perspective of a person who uses a language L for the purpose of talking about D. The types in the ontology represent the predicates, word senses, or concept and relation types of the language L when used to discuss topics in the domain D.

[54] Ob Vergegenständlichung in einer Ontologie ausgerückt werden kann, hängt von dem Umfang der gewählten Sprache ab.

meinsame Merkmale auszeichnen. Ein Beispiel aus der Domäne Universität ist das Konzept Student, das alle Person umfasst, die an einer Universität eingeschrieben sind und als ein gemeinsames Merkmal eine Matrikelnummer besitzen. Am Beispiel der Konzepte Student, Personen lässt sich bereits erkennen, dass die Konzepte mit Hilfe von Taxonomien organisiert werden und in diesem Fall Mechanismen wie Vererbung nutzen können.

Relation Eine Relation ist eine Beziehung zwischen Konzepten einer Domäne $K_1, K_2, \ldots K_n$ und kann formal als eine beliebige Teilmenge R des Kreuzprodukts der Mengen K ausgedrückt werden: $R \subset K_1 \times K_2 \times \ldots \times K_n$. In der Universitäts-Domäne kann die Relation eingeschrieben so definiert werden: $\forall a, b : (\text{eingeschrieben}(a, b) \rightarrow Person(a) \wedge Studienfach(b))$. Meistens werden in Ontologien binäre Relationen verwendet, insbesondere für eine Zuordnung von Attributen – auch Slots genannt – zu einem Konzept.

Axiom Bei einem Axiom handelt es sich um eine (formale) Aussage, die per Definition immer wahr ist. Axiome ermöglichen es, die entsprechende Domäne genauer zu beschreiben, indem mit ihnen Sachverhalte ausgedrückt werden können, die mit den beiden oberen Elementen einer Ontologie allein nicht zu beschreiben sind. Zum Beispiel könnte mit einem Axiom in der Univeritäts-Domäne ausgedruckt werden, dass ein Student einen Aufbau-Kurs nur dann belegen darf, wenn er vorher einen Grund-Kurs absolviert hat. Mit Hilfe von Axiomen kann die Konsistenz einer Ontologie geprüft werden und außerdem kann neues Wissen über die Domäne gewonnen werden, indem die vorhandenen Axiome[55] kombiniert werden.

Oft werden für die hier aufgelisteten Begriffe weitere Bezeichner synonym verwendet. So werden Konzepte alternativ auch als Begriffe, Entitäten, Klassen; Relationen als Beziehungen, Assoziationen, Rollen bezeichnet (vergleiche [CRP06], Seite 5).

An dieser Stelle ist es wichtig zu bemerken, dass die Häufigkeit der Verwendung der aufgelisteten Bestandteile mit der Reihenfolge in der Aufzählung abnimmt. So benutzen auch sehr informelle Ontologien – die hauptsächlich eine Sammlung von Begriffen einer Domäne darstellen – Konzepte. Da sie meist glossarartig aufgebaut sind, verzichten sie in der Regel auf die Verwendung von Relationen und Axiomen bzw. formulieren diese in der

[55] Bei Ontologien wird in der Regel nicht zwischen Axiomen und Schlussregeln unterschieden.

natürlichen Sprache.

Klassifikation von Ontologien

Eine Zusammenstellung, was in der Fachliteratur als eine Ontologie bezeichnet wird, findet sich in [SW01] und [GMZ07]. Die untersuchten Ausprägungen reichen von einfachen Katalogen, die zum Beispiel eine Auflistung von Gegenständen enthalten, über Taxonomien bis hin zu komplexen Theorien der formalen Logik. Sie unterscheiden sich in dem Grad der Formalisierung.[56]. Ergänzend zu der Klassifizierung nach dem Formalisierungsgrad können Ontologien nach ihrem Einsatzbereich klassifiziert werden. Dabei steht der Bereich, der durch eine Ontologie beschrieben wird, im Vordergrund. Guarino ([Gua98]) schlägt folgende hierarchische Klassifizierung vor:

Ontologie hoher Stufe [57] Diese Ontologien bilden die oberste Ebene und beschreiben allgemeine, Domänen-unabhängige Konzepte wie Raum, Zeit, Beziehung, Aktion, Objekt usw. Die Aufgabe einer *Ontologie hoher Stuffe* ist die Einigung auf generelle Konzepte und ihre Zusammenhänge, die bei der Modellierung der unterordneten Ontologien gelten sollen. Dadurch können Elemente einer *Domänen-Ontologie* entsprechend den Elementen in der *Ontologie hoher Stufe* klassifiziert und ihre Beziehungen auf Gültigkeit überprüft werden. Zum Beispiel kann in einer Generalisierungs-Hierarhie geprüft werden, ob alle Elemente Typen (bzw. Klassen) sind und keine Objekte.

Domänen-Ontologie Eine *Domänen-Ontologie* beschreibt das spezifische Vokabular einer Domäne (z.B. Universität). Die Grenzen einer Domäne können beliebig gewählt werden und eine Überlappung von *Domänen-Ontologien* ist erlaubt.

Aufgaben-Ontologie Diese Ontologien stehen auf der gleichen Ebene wie die *Domänen-Ontologien*. Eine *Aufgaben-Ontologie* beschreibt – wie der Name bereits andeutet – das Vokabular, das in Rahmen einer Aufgabe bzw. Aktivität (z.B. Prüfung) verwendet wird.

Anwendungs-Ontologie Diese Ontologie verknüpft Elemente der beiden oberen Ontologien und ist diesen hierarchisch untergeordnet. Aus den oberen Ontologien übernommene Elemente werden in einer *Anwendungs-Ontologie* mit zusätzlichen Informationen wie Rollen (z.B. Prüf-

[56] Die Einordnung von Ontologien in dieser Klassifikation hängt überwiegend von der Sprache, die eine Ontologie verwendet. Für weiterführende Informationen und die Rolle der gewählten Sprache siehe Kapitel 2.2.4 und insbesondere Abbildung 2.13

[57] Original: High Level Ontology

ling) angereichert, die die Elemente der *Domänen-Ontologie* bei der Ausführung einer Aktivität aus der *Aufgaben-Ontologie* annehmen.

Ontologie versus konzeptuelles und Meta-Modell

Als ein Modellierungs-Artefakt im Sinne der Software-Technik (siehe Kapitel 2.3.3) können Ontologien bei ihrer weiteren Verwendung zwei unterschiedliche Rollen einnehmen. Einerseits kann eine Ontologie vorgeben, welche Strukturen in der untersuchten Domäne oder im allgemeinen in der realen Welt existieren, und somit eine Art Grammatik für den entsprechenden Bereich vorgeben. Andererseits kann eine Ontologie eine Nachbildung einer Domäne darstellen und dadurch – betrachtet als Vokabular – ein Begriffswörterbuch (Glossar) festlegen.

In der Modellierung existieren zwei weitere Artefakte, die vergleichbare Aufgaben übernehmen. Es sind im ersten Fall Meta-Modelle und im zweiten Fall konzeptuelle Modelle. Die Nähe dieser Modelle zu Ontologien bzw. teilweise Überlappung ist eine mögliche Ursache für Missverständnisse bezüglich der Rolle der Ontologie und kann zu Verwechselungen zwischen diesen Modellen und Ontologien führen. Auch der Umstand, dass diese Artefakte oft in gleichen oder verwandten Modellierungs-Sprachen verfasst werden, trägt zusätzlich dazu bei. Trotz gewisser Gemeinsamkeiten existieren Kriterien, die hilfreich sind, Ontologien gegenüber Meta-Modellen und konzeptuellen Modellen abzugrenzen.

Von Meta-Modellen unterscheiden sich Ontologien dadurch, dass sie einen anderen Fokus haben. Ein Vergleich der Ontologie-Definition (auf Seite 39) mit der Definition eines Meta-Modells der *Object Management Group* (siehe [OMG06a]) macht deutlich, dass eine Ontologie überwiegend deskriptiv ist und sich auf den Untersuchungsbereich (Problemraum) richtet. Ein Meta-Modell ist dagegen präspektiv und richtet sich auf den Lösungsraum.

Konzeptuelle Modelle unterscheiden sich von Ontologien nicht im Fokus wie Meta-Modelle, sondern in dem Grad der Spezialisierung der Domäne oder – einer Analogie zur Optik folgend – in der Schärfe. Während Daten-Modelle projektspezifisch sind und somit eine durch die Projektanforderungen eingeschränkte und möglicherweise abweichende Modellierung der Domäne bieten, blendet eine Ontologie die projektspezifische Sicht aus (siehe auch [HK04]). Existiert zu einer Domäne eine Ontologie und wird in einem Projekt diese Domäne modelliert, so basieren Elemente des Daten-Modells auf den Konzepten aus der Ontologie und sie verfeinern diese in der Regel.

Ein weiterer wichtiger Aspekt, um Ontologien von konzeptuellen und Meta-Modellen abzugrenzen, ist die Voraussetzung, dass eine Ontologie gemeinsam von mehreren Beteiligten (Personen, Agenten, ...) entsprechend einer ontologischen Verpflichtung benutzt wird. Weder Daten noch Meta-Modelle verlangen das.

> Die Grenzen meiner Sprache bedeuten
> die Grenzen meiner Welt
>
> *(Ludwig Wittgenstein* Tractatus 5,6*)*

2.2.4 Ontologie-Sprachen

Das Spektrum von Wissensrepräsentation-Sprachen, die zum Verfassen einer Ontologie in Frage kommen, ist aktuell sehr breit. Auf Grund ihres unterschiedlichen Ursprungs und ihrer Einsatzbereiche (zum Beispiel *Semantic Web* oder *KI*-Forschung) unterscheiden sich diese Sprachen in ihren Eigenschaften, ihrer Präzision und Ausdruckskraft sehr stark. An dieser Stelle wird ein Überblick über aktuelle Sprachen zur Wissensrepräsentation gegeben[58].

Eine Konzeptualisierung eines Untersuchungsbereichs ist ein abstraktes Artefakt, das im Kopf eines Beobachters entsteht. Fokussiert er seine Gedanken auf bestimmte Elemente einer Domäne und Zusammenhänge zwischen diesen, bildet er zunächst abstrakte Modelle dieses Ausschnitts, die als Spezialfälle der zu Grunde liegenden Konzeptualisierung verstanden werden können. Erst um sich über diese Modelle mit anderen auszutauschen, sie festzuhalten und weiter zu bearbeiten, ist erforderlich, sie in die Form eines konkreten Artefakts zu gießen. Das erfordert also eine Form der Wissensrepräsentation (eine Sprache), in der die Modelle ausgedrückt werden können.

Es existieren vielfältige Möglichkeiten, das Wissen über eine Domäne festzuhalten. Da der Mensch beim Erstellen von Modellen eine Schlüsselrolle spielt, orientieren sich Wissensrepräsentations-Sprachen sehr stark an seinen kognitiven Eigenschaften und Theorien darüber, wie er die Welt wahrnimmt und verarbeitet. Basierend auf diesen Erkenntnissen bilden Elemente wie logische Aussagen, Regeln, Konzepte, Bilder, Analogien und Metaphern eine Grundlage dieser Sprachen.

Eine weitere wichtige Charakterisierung aller Wissensrepräsentations-Sprachen kann aus dem Begriff *Wissensrepräsentation* selbst abgeleitet werden. Davis et al. [DSS93] beschreiben Wissensrepräsentation anhand von folgenden fünf Aspekten:

- Die Wissensrepräsentation ist vereinfachend. Die Symbole sind Platzhalter für die Konzepte der realen Welt und können sie nicht ersetzen.

[58] Dieses Kapitel liefert einen kompakten Überblick. Eine detaillierte Beschreibung findet sich zum Beispiel in [Krz06] und [GDD06].

So können die Eigenschaften eines Gegenstands im Allgemeinen nicht automatisch und ohne Vorkenntnisse über die Domäne allein über die Kenntnis seines Symbols abgeleitet werden.

- Bei der Abbildung einer Domäne werden zwangsläufig bestimmte Sachverhalte hervorgehoben, indem sie repräsentiert werden und andere werden ausgeblendet, da sie entweder uninteressant sind oder mit der jeweiligen Sprache nicht ausgedrückt werden können.
- Eine Symbol-basierte Repräsentation schränkt inhärent – im Vergleich zu den repräsentierten Sachverhalten in der realen Welt – ein, welche Schlussfolgerungen abgeleitet werden können. Im Falle einer Ontologie bedeutet dies, dass die Wahl der Sprache bestimmt, welche ontologischen Folgerungen für einen gegebenen Gegenstandsbereich gemacht werden können.
- Die Repräsentation ermöglicht es, Wissen über eine Domäne auch maschinell zu verarbeiten und soll, um das zu ermöglichen, eine entsprechende Repräsentationsform besitzen.
- Da Modelle von Menschen erstellt und verwendet werden, soll es auch für diese Zielgruppe eine zugängliche Repräsentationsform geben.

Diese Eigenschaften sind sprachabhängig und können bei der Konzeption einer Sprache positiv oder negativ beeinflusst werden. Auch werden die oben erwähnten Elemente nicht von jeder Sprache angeboten, so dass sich im Laufe der Zeit in der Informatik sehr unterschiedliche Wissensrepräsentation-Sprachen mit unterschiedlichen Schwerpunkten entwickelt haben.

Klassifikation von Ontologie-Sprachen

Eine Zusammenstellung, was in der Fachliteratur als eine geeignete Wissensrepräsentationsform verwendet wird, um Ontologien zu beschreiben , findet sich bei [SW01] und [GMZ07]. Die untersuchten Ausprägungen reichen von einfachen Katalogen, die zum Beispiel eine Auflistung von Gegenständen enthalten, über Taxonomien bis hin zu komplexen Theorien der formalen Logik. Verbunden mit der Komplexität dieser Ausprägungen bieten die entsprechenden Wissensrepräsentationen entweder keine, schwache oder gut ausgeprägte Unterstützung für das automatische Schließen. Folgendes Bild zeigt eine eindimensionale Einordnung dieser Repräsenationsformen, wobei der Formalisierungsgrad von links nach rechts zunimmt:

Zu der ersten Gruppe links im Bild gehören kaum formalisierte, glossarartig aufgebaute Ontologien, die Konzepte einer Domäne auflisten und sie

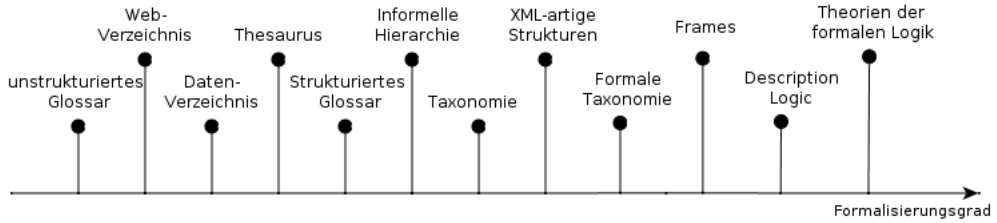


ABBILDUNG 2.13: Überblick über Wissensrepräsentationsformen für Ontologien nach [GMZ07]

in der natürlichen Sprache beschreiben. Thesaurus-artig aufgebaute Ontologien benutzen zusätzlich zu dem Vokabular auch Beziehungen zwischen den Konzepten. Oft werden mit deren Hilfe Synonyme und Ober-, Unterbegriffe (Taxonomien) besonderes ausgezeichnet. Die Beziehungen tragen in dieser Gruppe noch einen informellen Charakter, was insbesondere bei der Bildung von Ober- und Unterbegriffen zu beobachten ist. In einem Thesaurus wird noch nicht verlangt, dass diese Beziehung durchgehend transitiv verwendet wird. Die Wissensrepräsentationsformen weiter rechts (formale Taxonomien) spezifizieren die Beziehungen genauer und verlangen, dass zum Beispiel eine strikte Trennung von Klassen und Individual-Konzepten[59] vorgenommen wird. Frames führen als eine zusätzliche Strukturierungsmöglichkeit noch Eigenschaftsfächer (*Slots*, siehe auch Seite 47) ein, die in den meisten Frame-basierten Sprachen vererbt werden können. Die letzte Gruppe bilden Beschreibungsformen, die mit formalen Axiomen und Schlussregeln arbeiten.

Mehrere Untersuchungen ([Myl81], [BKK+02], [GDD06]) der existierenden Wissensrepräsentations-Sprachen haben leicht abweichende Klassifizierungen aufgestellt, die folgende Kategorien der Repräsentationsformen gemeinsam haben: Regel-, Struktur- und Logik-basiert. Später entstandene Untersuchungen von Baclawski at al. [BKK+02] und Gasevic at al. [GDD06] betrachten auch graphische Repräsentationsform als eine weitere Kategorie. Sie beinhaltet sowohl rein graphische Sprachen wie *UML*, die eine textuelle (*XML*-artige) Darstellung zwecks der persistenten Speicherung nutzen, als auch Sprachen, die eine graphische Repräsentationsform zusätzlich zu den bereits existierenden Sprachen der anderen Klassen anbieten.

[59] Individual-Konzepte sind vergleichbar mit den Instanzen bei der Objekt-orientierung

Regel-basierte Repräsentationsformen

Die vorwiegend für Expertensysteme im Bereich der kommerziellen *KI*-Anwendungen entwickelten Regel-basierten Ontologie-Sprachen basieren auf der Möglichkeit, *Wenn-Dann-Regeln* zu definieren. Mit deren Hilfe kann man Wissen für einen vorher festgelegten Aufgabenbereich – wie zum Beispiel Symptome und die dazu gehörigen medizinischen Diagnosen – beschreiben und effizient auswerten. In solchen Situationen zeigen in Regelbasierten Sprachen beschriebene Systeme ihre Vorteile und liefern in der Regel bessere Ergebnisse als menschliche Experten. Ihre Schwächen liegen in der Skalierbarkeit. Überschreitet eine zu beschreibende Domäne die Grenzen einer festgelegten Aufgabe, ist es im Allgemeinen schwierig, den Regelsatz so zu erweitern, dass weiterhin zuverlässige Ergebnisse geliefert werden.

Hauptsächlich unterscheiden sich Regel-basierte Sprachen voneinander dadurch, ob die Regeln deklarativ oder imperativ formuliert werden. In neueren Ansätzen, die im Rahmen des *Semantischen Webs* oder für eine deklarative Definition von Diensten entstanden sind, werden reine *Wenn-Dann-Regeln* mit Elementen der *Frame*-artigen Sprachen angereicht. Dadurch erreicht man, dass die Informationen über ein Konzept nicht mehr über Regeln verstreut sind, sondern mittels eines *Frames* gekapselt werden, was der Skalierbarkeit zu Gute kommt. Ein Beispiel für solche Sprachen ist *RuleML*.

Traditionell werden in diesem Bereich oft proprietäre Sprachen verwendet, die zum Beispiel auf Sprachen wie *Prolog* oder *Lisp* aufbauen und sie als formale Grundlage für den Aufbau des Regelsystems benutzen.

Struktur-basierte Repräsentationsformen

Die Struktur-basierten Ontologie-Sprachen legen ihren Schwerpunkt auf die Abbildung der strukturellen Gegebenheiten eines Gegenstandsbereichs und können komplementär zu den Logik-basierten Sprachen benutzt werden. Es handelt sich um eine Art der „objekt-orientierten“ Strukturierung und sie basiert auf unserer Wahrnehmung und Vorstellungen über menschliches Erkennen und Denken. Sie gilt als eine für Menschen zugängliche Form der Wissensrepräsentation [DSS93].

Struktur-basierte Sprachen unterteilt Mylopoulos [Myl81] in zwei Unterklassen: Netzwerk- und *Frame*-artige. Die ersten sind relativ einfach aufgebaut und bieten im Wesentlichen Knoten und Beziehungen als Sprachelemente an. Die zweite Unterklasse ermöglicht eine Unterstukturierung und

(wie der Name schon andeutet) basiert diese Klasse von Sprachen auf *Frames*. Sie stehen für Konzepte und dienen zur Identifikation und weiteren Charakterisierung von diesen mit Hilfe von *Slots* und *Facets*. Ein *Frame* muss benannt werden und wird über seinen Bezeichner eindeutig identifiziert. *Frames* können sowohl für einzelne Objekte (*instance Frame*), als auch für Klassen von Objekten (*class Frame*) stehen. *Slots* sind binäre Relationen zwischen *Frames* und einer Menge von Werten, die eine Eigenschaft von diesem *Frame* einnehmen kann.

In den *Frame*-artigen Sprachen können *Slots* – im Unterschied zu den Objekt-orientierten Sprachen[60] – unabhängig von *Frames* definiert und mehreren *Frames* zugewiesen werden. *Slots* sind wiederum selbst *Frames* und können mit weiteren *Slots* und *Facets* verknüpft werden. *Facets* dienen zur Charakterisierung einer Relation zwischen einem *Frame* und einem *Slot*. *Facets* sind ternäre Relationen zwischen den beiden und einem Wert-Typ. Dadurch werden Restriktionen wie zum Beispiel Kardinalität in der Beziehung zwischen einem *Frame* und einem *Slot* festgelegt.

Ein wichtiges Konstrukt in den *Frame*-artigen Ontologie-Sprachen ist die Möglichkeit, Hierarchien von *Frames* zu bilden. Die *instance Frames* eines in der Hierarchie unten stehenden *class Frame* sind gleichzeitig Instanzen aller[61] in der Hierarchie darüber stehenden *Frames*. Analog zur Vererbung in den Objekt-orientierten Sprachen gehören *Slots* eines hierarchisch höher stehenden *Frames* auch allen *Frames* in der Hierarchie darunter und werden dadurch vererbt. Ausnahme aus dieser Regel bilden so genannte *own Slots*, die nur Eigenschaften des entsprechenden *Frames* beschreiben.

Eine bekannte *Frame*-basierte Ontologie-Sprache ist zum Beispiel *Ontolingua*. Komplementarität zu den Logik-basierten Sprachen hat zu vielen Mischsprachen geführt. Ein sehr bekannter Vertreter dieser Gattung ist die Sprache *F-Logic*. Wie bereits erwähnt, unterstützen Struktur-basierte Sprachen den Versuch, einen Gegenstandsbereich entsprechend menschlicher Wahrnehmung zu beschreiben und sind in der Regel intuitiv zu benutzen. Das trifft im Besonderen zu, wenn eine graphische Schnittstelle angeboten wird (siehe Seite 49).

[60] Frames würden in diesem Fall – Klassen und Slots – Attributen einer Klasse entsprechen.

[61] Diese Beziehung ist transitiv.

Graphen-basierte Repräsentationsformen

Graphen-basierte Sprachen können in zwei Gruppen unterteilt werden. Die erste Gruppe bilden Sprachen, die grundsätzlich für eine visuelle Darstellung der Konzepte und ihrer Relationen eines Gegenstandsbereichs entwickelt wurden. Die zweite Gruppe beinhaltet Sprachen, die auf einer textuellen Sprache aufbauen und eine zusätzliche graphische Darstellungsform für sie anbieten. Sprachen der ersten Gruppe lassen sich in der Regel über Umwege (z.B. eine *XML*-artige Darstellung) maschinell bearbeiten. Die Zugänglichkeit der zweiten Gruppe der Graphen-basierten Sprachen für Maschinen hängt vor allem von der zu Grunde liegenden textuellen Sprache und ihrer Eignung für die maschinelle Verarbeitung ab. Im Bezug auf die Menschen-„Freundlichkeit" schneiden Graphen-basierte Sprachen besonderes gut ab, da sie in der Regel gezielt für den menschlichen Betrachter entwickelt wurden.

Die zweite Eigenschaft der Wissensrepräsentation-Sprachen – das Hervorheben von bestimmten Sachverhalten und Ausblenden von anderen (siehe Seite 45) – ist bei den Graph-basierten Sprachen besonderes ausgeprägt. Auf Grund ihrer visuellen Darstellung ermöglichen sie es, wichtige Elemente in den Vordergrund zu holen und feine Details auszulassen. Diese starke Neigung zur Fokussierung auf bestimmte Elemente eines Gegenstandsbereichs ist gleichzeitig eine Stärke und Schwäche dieser Gattung von Sprachen. Gegenüber der Übersichtlichkeit und schnellen Einstiegsmöglichkeit bei der Betrachtung der abgebildeten Domäne besteht bei den Graph-basierten Sprachen die Gefahr, dass andere, nicht weniger wichtige, aber nicht so auffallend dargestellte Elemente übersehen werden können. Textuelle Sprachen neigen deutlich weniger dazu, da das Wissen über eine Domäne sequentiell und somit gleichrangig dargestellt wird.

Die Abbildung 2.14 zeigt eine graphische Darstellung des englichen Satzes „Tom believes that Mary wants to marry a sailor" mit der Technik der *Konzeptuellen Graphen* [Sow01] modelliert. Dieses Beispiel zeigt, wie schnell sich die Bedeutung des Dargestellten einem Betrachter trotz der Komplexität[62] erschließt, ohne dass er die genaue Semantik der Sprache kennt. Diese Eigenschaft macht diese Klasse von Sprachen populär und erklärt, wieso auch für ursprünglich textuelle Sprachen eine graphische Darstellung angeboten wird. Weitere bekannte Graph-basierte Sprachen sind *Topic Maps*, *Semantische Netze*, *UML*. Sprachen wie *OWL* oder *RDF* besitzen ebenfalls eine graphische Schnittstelle.

[62] Es handelt sich genau genommen um drei verschachtelte *konzeptuelle Graphen*.

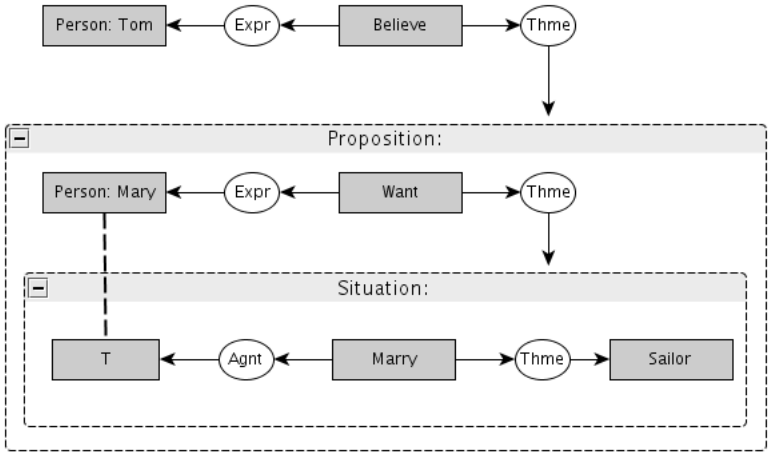


ABBILDUNG 2.14: Beispiel eines komplexen *Konzeptuellen Graphen*

Logik-basierte Repräsentationsformen

Logik-basierte Ontologie-Sprachen besitzen einen hohen Formalisierungsgrad und bauen auf den Grundlagen wie Aussagen- und Prädikatenlogik aus der klassischen Logik auf. Ontologien, die auf diesen Sprachen basieren, sind in Abbildung 2.13 auf der rechten Seite im Bereich der formalen Ontologien zu finden. Bekannte Vertreter dieser Sprachen sind *Knowledge Interchange Format (KIF)*, *General Ontological Language (GOL)* und *Description Logic*. Auch die wohl bekannteste und vom *World Wide Web Consortium (W3C)* standartisierte Ontologie-Sprache *Web Ontology Language (OWL)* kann der Klasse der Logik-basierten Sprachen zugeordnet werden, da zu ihren Wurzeln die Sprache *Ontology Interchange Language (OIL)* gehört, die auf der formalen Grundlage der Beschreibungslogik aufbaut, die wiederum eine Teilsprache der Prädikatenlogik erster Stufe ist.

Die Logik-basierten Ontologie-Sprachen unterscheiden sich untereinander vor allem in der Wahl des logischen Systems. Die mit der Sprache verbundene Ausdrucksstärke und Effizienz der computergestützten Schlussfolgerungen (Inferenz) hängt meistens direkt von dieser Wahl ab. So verwenden Sprachen, die Wert auf eine effiziente Auswertung der Schlussfolgerungen legen, Aussagenlogik. Sie können in diesem Fall inzwischen sehr effiziente *SAT-Solver*[63] verwenden, um in der Sprache formulierte Aussagen auf ihre Erfüllbarkeit zu testen. Entgegengesetzt zu dieser Eigenschaft der

[63] Es handelt sich um Programme, die darauf ausgerichtet sind, das Erfüllbarkeitsproblem der Aussagenlogik zu lösen.

Logik-basierten Sprachen verhält sich die Ausdrucksstärke. Somit ist sie im Falle der Aussagenlogik-basierten Sprachen eingeschränkt. Deswegen verwenden die meisten Logik-basierten Sprachen einen entscheidbaren Teil der Prädikatenlogik erster Stufe, der auch als Beschreibungslogik bezeichnet wird. Sie stellt einen guten Kompromiss zwischen der Inferenz und Ausdruckstärke dar. Ausdrucksstärkere Sprachen wie *KIF* nutzen die gesamte Prädikatenlogik erster Stufe.

Eine interessante Lösung findet sich bei der Sprache *OWL*. Sie verzichtet auf eine Festlegung, sondern bietet statt dessen eine Hierarchie aus drei Sprachen an: *OWL Lite*, *DL* und *Full*. Dabei ist zum Beispiel eine in *OWL Lite* formulierte Ontologie auch in der Teilsprache *OWL DL* gültig, was einen hierarchischen Aufbau von Ontologien mit unterschiedlichen Komplexitätsstufen ermöglicht.

Logik-basierte Ontologie-Sprachen vermeiden auf Grund ihrer formalen Grundlage Probleme wie Mehrdeutigkeiten und eignen sich gut für eine maschinelle Verarbeitung. Insbesondere Ontologien, mit denen Wissen aus schon modellierten Konzepten, Beziehungen und Aussagen über eine Domäne abgeleitet werden sollen (z.B. in *KI*), basieren oft auf den Logik-basierten Ontologie-Sprachen. Ein großer Nachteil dieser Sprachklasse ist die Tatsache, dass sich viele Sachverhalte aus der menschlichen Erfahrungswelt nur eingeschränkt oder mit einem großen Aufwand beschreiben lassen. Auch ihre ausgesprochen gute Eignung für die maschinelle Verarbeitung führt dazu, dass sie meistens für Menschen keine gut geeignete Repräsentationsform bieten.

2.3 Software-Entwicklungsprozesse

2.3.1 Überblick

In der Software-Technik wurde eine Vielzahl von Vorgehensmodellen entwickelt, die einen Software-Entwicklungsprozess verallgemeinert und idealisiert beschreiben und den Projektverantwortlichen als Leitfaden für die Abwicklung konkreter Projekte dienen. Der Auslöser für die intensive Entwicklung verschiedener Vorgehensmodelle war die *Software-Krise*. Das Wesen dieser Krise fasste Edsger W. Dijkstras treffend mit den folgenden Worten zusammen [Dij72]: „Solange es keine Maschinen gab, stellte die Programmierung kein Problem dar; als wir ein paar schwache Computer hatten, wurde Programmierung zu einem kleinen Problem, und nun seit-

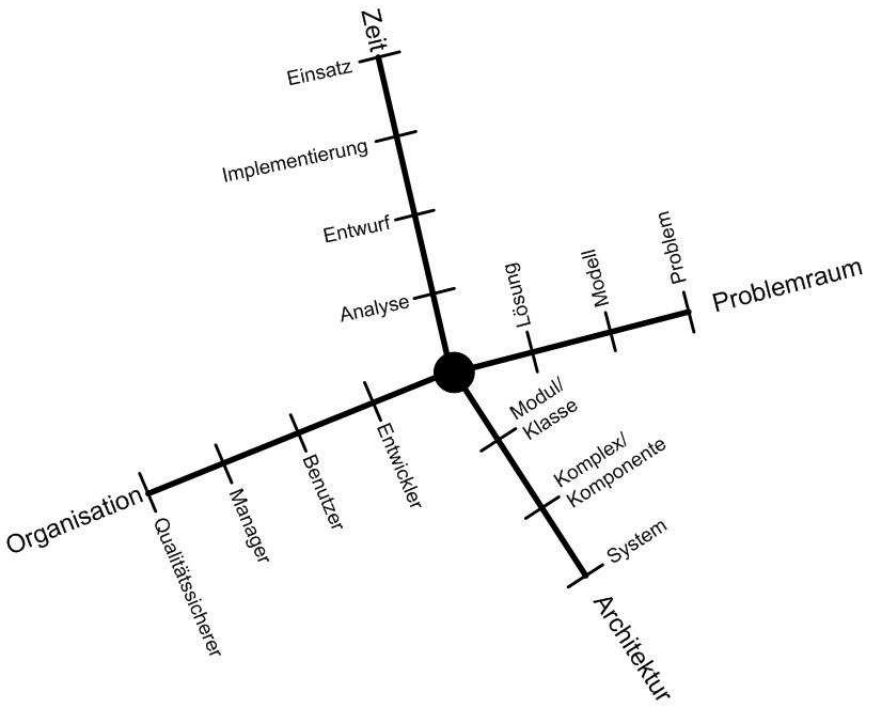


ABBILDUNG 2.15: Dimensionen eines Software-Prozesses [Hes96]

dem wir gigantische Computer haben, ist die Programmierung ein ebenso gigantisches Problem geworden.[64]“ Der Entwurf von präskriptiven Software-Entwicklungsprozessen sollte als Reaktion auf die Software-Krise die Komplexität der Software-Entwicklung beherrschbar machen.

Schematisch kann ein Software-Entwicklungsprozess mit einem mehrdimensionalen Modell basierend auf den folgenden Dimensionen beschrieben werden: Architektur, Raum, Zeit, Organisation (siehe Abbildung 2.15). Die ersten Vorgehensmodelle wie das *Wasserfall-Modell* legten den Schwerpunkt auf die Dimension Zeit, die für eine rechtzeitige Fertigstellung eines Projekts entscheidend ist. Um ein Software-Produkt nicht nur rechtzeitig, sondern auch in der von dem Kunden vorgegebenen Qualität auszuliefern, wurden weitere auf dem Wasserfall-Modell aufbauende Vorgehensmodelle entwickelt, die weitere Dimensionen stärker in ihr Modell einbeziehen. Als Beispiel dafür kann das *V-Modell*[65] dienen, bei dem die Dimen-

[64] Original: As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

[65] Dieses Modell ist in der Variante V-Model XT immer noch aktuell und ist quasi Stan-

sion *Raum* eine wichtige Rolle einnimmt. Neue Techniken wie die Objektorientierte Software-Entwicklung führten dazu, dass der strukturelle Aufbau eines Software-Produkts in den neueren Vorgehensmodellen besser berücksichtigt und sogar der zeitlichen Dimension übergeordnet wurde, wie zum Beispiel in dem evolutionären Vorgehensmodell *EOS*[66] [Hes96].

Eines der aktuellen Vorgehensmodelle *Rational Unified Prozess (RUP)* nimmt eine spezielle Rolle in der Landschaft der Vorgehensmodelle ein, da es mit dem Ziel entwickelt wurde, eine Vereinheitlichung der Software-Entwicklungsprozesse zu erreichen [Som07]. Einerseits bietet es ein umfangreiches Vorgehensmodell an und andererseits eine Möglichkeit, dieses Modell zu konfigurieren und an die vorliegende Situation in einem Projekt anzupassen. Obwohl *RUP* iterativ aufgebaut ist, ähnelt das Grundgerüst dieses Modells einem Wasserfallmodell sehr, das anerkannterweise als zu unflexibel gilt. Das ist einer der wichtigsten Kritikpunkte an diesem Vorgehensmodell [Hes01].

Die bisher angesprochenen Vorgehensmodelle können – vordergründig klassifiziert – als klassisch bezeichnet werden. Einen anderen Ansatz verfolgen agile Vorgehensmodelle, die als eine Alternative zu den klassischen Methoden – insbesondere für kleine und mittlere Projekte – entstanden sind. Zu den bekanntesten Vertretern dieser Gattung zählt *Extreme Programming* [Som07]. Der wesentliche Unterschied zur klassischen Vorgehensweise liegt bei den agilen Methoden darin, dass sie die Vorgaben zu den Dimensionen eines Software-Entwicklungsprozesses auf ein Minimum reduzieren. Stattdessen liefern sie eine generische Anleitung, wie in zyklisch ablaufenden, *kurzen* Zeitintervallen[67] meist lauffähige Teile eines Software-Produkts hergestellt werden sollen. Am besten kann das an dem agilen Vorgehensmodell *Scrum* [Pic07] veranschaulicht werden.

Die Abbildung 2.16 stellt sowohl Artefakte als auch den Prozess-Ablauf von *Scrum* dar. Alle Projekt-Aufgaben werden nach Prioritäten sortiert in dem *Product Backlog* aufgelistet. In jeder Iteration wird ein Teil davon in den *Sprint Backlog* übertragen, in kleinere Aufgaben unterteilt (*Sprint Backlog Items*) und innerhalb von maximal vier Wochen bearbeitet. In dieser Zeit wird der Fortschritt täglich in den *Daily Scrum meetings* kontrol-

dard bei aktuellen Projekten der öffentlichen Hand in Deutschland.

[66] Siehe für weitere Informationen das folgende Kapitel 2.3.2

[67] Meistens handelt es sich um wenige Monate. Diese Zeitabstände können nicht den Phasen der Zeit-Dimension gleichgesetzt werden, da bei agilen Methoden innerhalb eines solchen Zeitraums beliebige Tätigkeiten – von der Analyse bis zum Testen – durchgeführt werden können.

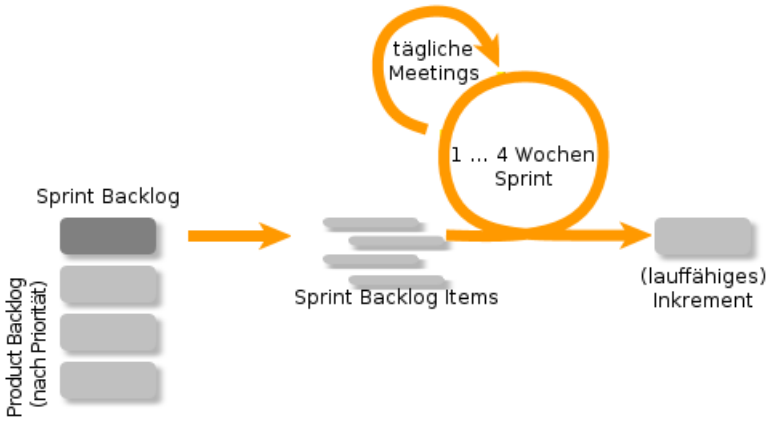


ABBILDUNG 2.16: Schematische Darstellung von *Scrum*

liert. Nach jeder Iteration wird das Ergebnis dem Auftraggeber präsentiert.

Vorteile der agilen Software-Entwicklung liegen in ihrer Kompaktheit und Flexibilität. So kann zum Beispiel das *Product Backlog* in *Scrum* jederzeit verändert werden. Bei dem nächsten *Sprint* können diese Änderungen bereits berücksichtigt werden. Nachteile liegen hauptsächlich auf der Management- und der vertraglichen Seite, da wegen der nur bedarfsweise vorhandenen und veränderbaren Spezifikation zeitliche und finanzielle Zusagen schwer zu machen sind [Som07].

2.3.2 Evolutionäre, objektorientierte Software-Entwicklung

Eines der Ziele[68] dieser Arbeit ist eine Beschreibung von Prozessen, mit deren Hilfe eine Nutzung von Ontologien in den typischen Software-Entwicklungsprozessen ermöglicht wird. Als Ausgangsbasis für diese Prozessbeschreibung dient das Modell zur *evolutionären, objektorientierten Software-Entwicklung (EOS)*. Dieses Vorgehensmodell wurde unter Berücksichtigung von Besonderheiten der objektorientierten Software-Entwicklung wie Zusammensetzung der Architektur eines Systems aus statischen Bausteinen entworfen [Hes05]. *EOS* bietet ein hohes Maß an Flexibilität und Skalierbarkeit sowohl für die Manager eines Projekts als auch für die beteiligten Entwickler, insbesondere in komplexen Projekten mit vielen Komponen-

[68] Für die Auflistung der Ziele siehe Kapitel 1.2.

ten und nebenläufig ablaufenden Entwicklungsprozessen [BHR+07b]. Das *EOS*-Vorgehensmodell beruht auf folgenden Leitgedanken [Hes96]:

- Objekt-orientierte Software-Entwicklung ist ein *hierarchischer, zyklischer Prozess*, der sich auf unterschiedlichen Ebenen der Software-Architektur in jeweils *analoger Weise* vollzieht.
- Software-Entwicklungsprozesse sind stark an die Software-Architektur gekoppelt: Jeder *System-Baustein* durchläuft seine eigene Entwicklung. *EOS* unterscheidet folgende, hierarchisch verschachtelte System-Bausteine: *System*, *Komponente*[69] und *Modul*.
- Der Entwicklungsprozess jedes Bausteins ist als *Zyklus* ausgelegt, der nach dem gleichen Schema aufgebaut ist. Die einzelnen Phasen dieses Zyklus sind *Analyse*, *Entwurf*, *Implementierung* und *Operativer Einsatz*[70].
- Phasen eines Zyklus und der hierarchische Aufbau eines Systems aus Bausteinen sind *orthogonal* zueinander und können beliebig miteinander gekoppelt werden. Dadurch entstehen die Aufgaben wie zum Beispiel *System-Analyse*, *Komponenten-Implementierung*, *Modul-Entwurf* usw.
- Die zeitliche Abfolge der Entwicklungszyklen für einzelne Bausteine wird *nicht durch einen übergeordneten Phasenplan bestimmt*, sondern entsprechend der aktuellen Projekt-Entwicklung durchgeführt und koordiniert.
- Bezüglich des Prozess-Managements ersetzt *EOS* die herkömmlichen Meilensteine durch *Revisionspunkte*. Sie sind Baustein-orientiert und geben den Entwicklungsstand eines Bausteins anhand seiner aktuellen Phase wieder. Dadurch ergibt sich der *Gesamtstatus* eines Projekts aus einer *Zusammenstellung der Zustände* aller in Entwicklung befindlichen Bausteine.

Im Unterschied zu den meisten klassischen Vorgehensmodellen, die auf der Dimension *Zeit* aufbauen, haben in *EOS* nicht irgendwelche Projektphasen die höchste Priorität, sondern die System-Bausteine und somit die Dimension *Architektur*. Die zeitliche Dimension wird in Form von Zyklen den System-Bausteinen untergeordnet. Dadurch kann ein nach *EOS* ablaufendes Software-Projekt viele Zyklen enthalten, die beliebig tief – bis hin zu jedem einzelnen Modul – gestaffelt werden können.

[69] Wobei eine Komponente rekursiv aus anderen Komponenten aufgebaut werden kann.
[70] *Operativer Einsatz* beinhaltet Tätigkeiten wie *Betrieb* und *Wartung* eines Bausteins.

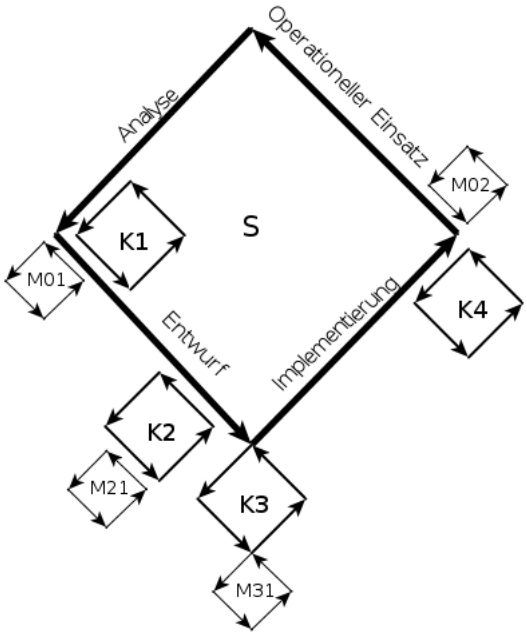


Abbildung 2.17: Fraktale Darstellung von *EOS*-Zyklen [Hes96]

Da das System selbst in *EOS* als ein Baustein betrachtet wird, besitzt es ebenfalls einen eigenen Zyklus. Auch wenn diese Konstruktion auf den ersten Blick nach dem klassischen Wasserfall-Modell aussieht, weist sie zwei entscheidende Unterschiede zu diesem Modell auf. Erstens ist die zeitliche Dimension als *echter* Zyklus angelegt, was eine evolutionäre Entwicklung des Systems ermöglicht. Zweitens spielen die System-Phasengrenzen eine untergeordnete Rolle und haben keinen vorschreibenden Charakter für die untergeordneten Zyklen von Komponenten und Modulen. Dadurch ist es in *EOS* möglich, dass ein Modul bereits in der Implementierung ist, während das System sich in der Analyse-Phase befindet. Für die zeitliche Koordination der Entwicklungsprozesse sorgen die Revisionspunkte eines Projekts.

Die Abbildung 2.17 veranschaulicht schematisch, wie die Gesamtstruktur eines *EOS*-Projekts bestehend aus den Bausteinen System S, Komponenten $K1 \ldots K4$ und Modulen $M01, M02, M21, M31$ aussehen kann. Diese Darstellung betont sowohl den hierarchischen, *fraktalen* Aufbau der Systemarchitektur als auch die zeitlich versetzte Ausführung von Entwicklungszyklen der einzelnen Bausteine. Auf diese Art wird zum Beispiel ver-

deutlicht, dass die Komponente $K4$ erst in der Implementierungsphase des Systems S analysiert und entwickelt wird.

In den Phasen Implementierung und Operativer Einsatz bietet *EOS* einen zusätzlichen System-Baustein – *Subsystem*. Ein Subsystem ist eine Zusammenfassung von Komponenten und Modulen, die zu einer ausführbaren Einheit – einem Inkrement – geformt wird. Bedingt durch unterschiedliche Entwicklungsstadien, in denen sich verschiedene Bausteine eines *EOS*-Projekts befinden können, bieten Subsysteme eine Möglichkeit, bereits im frühen Projekt-Entwicklungsstand lauffähige Inkremente auszuliefern oder die Funktionalität des zukünftigen Systems mittels Prototypen zu veranschaulichen.

Die Verschiebung des Schwerpunkts eines Entwicklungsprozesses in *EOS* von der Zeit-Dimension auf die Architektur eines Systems kommt nicht nur Software-Entwicklung entgegen, sondern kann auch in anderen Bereichen wie die Erstellung einer Ontologie ausgenutzt werden [BHR+07b]. Diese Möglichkeit wird an einer anderen Stelle dieser Arbeit im Abschnitt zur Homogenisierung der Software- und Ontologie-Entwicklungsprozessse aufgegriffen.

2.3.3 Modellierung von Software-Prozessen

Als Software-Prozess wird in der Software-Technik eine Menge von Tätigkeiten und damit zusammenhängenden Ergebnissen bezeichnet, durch die ein Software-Produkt entsteht [Som07]. Eine meist formale Vorgabe, wie diese Prozesse durchgeführt werden sollen, kann mit Hilfe eines präskriptiven Modells – eines Vorgehensmodells – beschrieben werden.

Ein Blick auf die Prozessmodellierung zeigt, dass dieser spezielle Bereich der Modellierung eine vergleichbare Entwicklung bezüglich der Nutzung von Meta-Modellen erfahren hat. Zunächst standen einzelne Vorgehensmodelle im Vordergrund, die unabhängig voneinander entwickelt und meist konkurrierend eingesetzt wurden. Die praktische Erfahrung mit diesen Modellen hat gezeigt, dass ein Vorgehensmodell in der Regel nur mit Anpassungen[71] im Kontext eines realen Software-Projekts anwendbar war. Als erste Reaktion auf diesen Anpassungsbedarf wurde im Rahmen des sehr umfangreichen Vorgehensmodells *RUP*[72] eine Möglichkeit angeboten, Anpassungen an diesem Vorgehensmodell im beschränkten Maße zu definieren.

[71] Als typische Abweichungen werden folgende Beispiele angesehen: abweichende Belegung der Rollen, Auslassen oder Einführen von neuen Arbeitsergebnissen usw.

[72] Siehe auch Kapitel 2.3.1

In einem nächsten Schritt und als Verallgemeinerung dieser Vorgehensweise entwickelte die *Object Management Group (OMG)* ein standardisiertes Meta-Modell zur Beschreibung von Prozessmodellen: *Software Process Engineering Meta-Model (SPEM)* [OMG08].

SPEM definiert eine Sprache, mit der verschiedene Vorgehensmodelle und ihre Anpassungen beschrieben werden können. Entsprechend den im vorherigen Abschnitt eingeführten Modellierungsräumen steht *SPEM* in der 2. Schicht. Mit ihr beschriebene Modelle – in diesem Fall Prozessmodelle – sind in der 1. Schicht zu finden und eine Anwendung eines Vorgehensmodells in einem realen Projekt findet sich dementsprechend in der 0. Schicht. Die 3. Schicht ist im Bereich der Prozessmodellierung noch nicht besetzt, da zur Zeit keine alternativen Meta-Modelle für die Beschreibung von Prozessmodellen existieren und eine Vereinheitlichung auf einer höher liegenden Schicht nicht notwendig ist.

Während *SPEM* eine Spezifikation ist, bietet *Eclipse Foundation* ein Projekt mit dem Namen *Eclipse Process Framework (EPF)* an, bei dem es sich um eine Umsetzung dieses Standards[73] handelt. Im Vergleich zu ihm bietet *EPF* das Werkzeug *EPF Composer*, mit dessen Hilfe Vorgehensmodelle nach *SPEM* definiert und in Form einer Webseite publiziert werden können. Die Beschreibung von *OBSE*-Prozessen basiert auf *SPEM* und wird mit Hilfe von *EPF Composer* durchgeführt.

SPEM ist modular aufgebaut. Für die Definition von Vorgehensmodellen wichtigste Pakete sind MethodContent, ProcessStructure und ProcessBehavior. Mit Hilfe des MethodContent-Pakets werden grundlegende, von einem spezifischen Vorgehensmodell unabhängige Elemente eines Prozesses beschrieben. Sie können in mehreren Modellen verwendet werden. ProcessStructure ist für die Beschreibung von einem Grundgerüst eines Vorgehensmodells vorgesehen.

Das letzte Paket in dieser Auflistung – ProcessBehavior – verknüpft MethodContent und ProcessStructure miteinander. Dadurch wird es möglich, für alle Vorgehensmodelle verfügbare Elemente in ein spezielles Vorgehensmodell zu integrieren und somit das Prozessgerüst dieses Modells mit Inhalten zu füllen. Diese Modularisierung ermöglicht es, ein bereits mit *SPEM* beschriebenes Vorgehensmodell zu ergänzen oder zu überschreiben, ohne dass die komplette Modell-Beschreibung angepasst werden muss.

Die Abbildung 2.18 liefert einen Einblick in den Aufbau des MethodContent-Pakets. Symbole, die in den Klassen-Überschriften verwendet wer-

[73] Angestrebt ist eine volle Unterstützung der 2. Version des Standards.

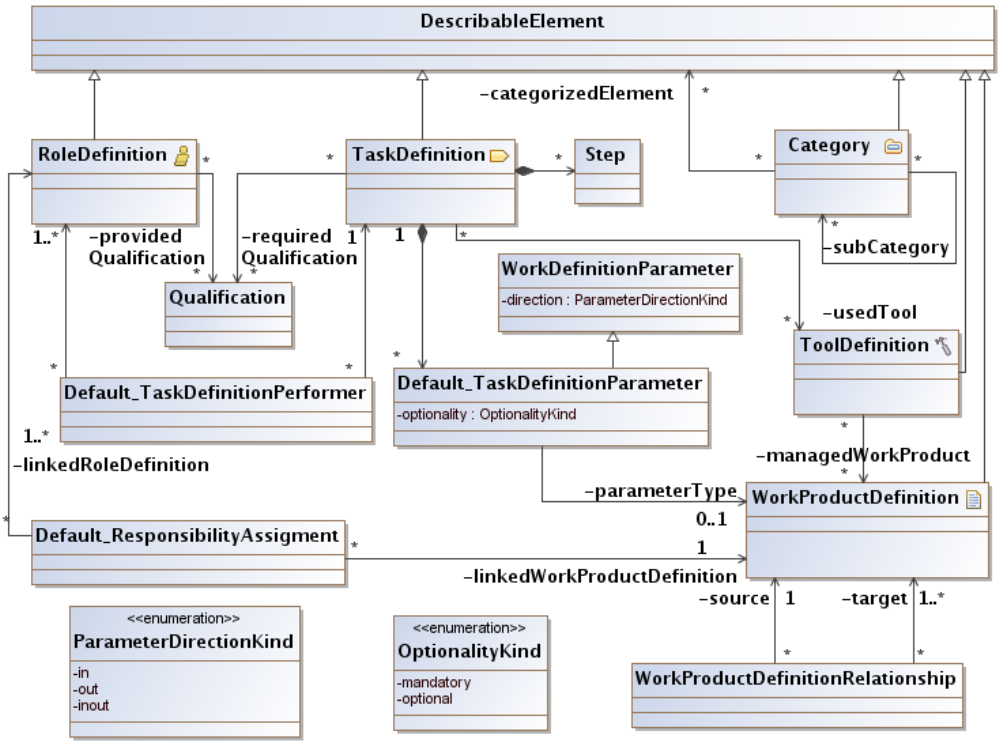


ABBILDUNG 2.18: Ausschnitt des *SPEM*-Prozess-Meta-Modells mit Schwerpunkt auf dem MethodContent-Paket (vgl. [OMG08])

den, entsprechen der graphischen Darstellung dieser Elemente im *EPF*. Auf diese Weise wird ein Zusammenhang zu den *EPF Composer*-Diagrammen im Kapitel 5 hergestellt. Namen der Meta-Klassen in diesem Paket, die wichtige Prozesselemente wie zum Beispiel Rolle definieren, werden mit dem Wort Definition ergänzt. Damit soll eine allgemeine Beschreibung einer Rolle suggeriert werden.

Im Mittelpunkt des *SPEM*-Meta-Modells steht die Meta-Klasse TaskDefinition. Sie definiert eine *Aufgabe* in einem Software-Entwicklungsprozess, die aus mehreren Schritten (Step) bestehen kann. Aufgaben in einem Software-Entwicklungsprozess können als Operationen angesehen werden, die zur Herstellung von Software dienen. Sie werden in *SPEM* entsprechend dieser Analogie mit Parametern (Default_TaskDefinitionParameter) verknüpft, die von einer Aufgabe konsumiert oder erzeugt[74] werden.

[74]Die Richtung wird von einem Objekt der Meta-Klasse ParameterDirectionKind vorgegeben. Sie können außerdem *verbindlich* oder *optional* sein, was durch OptionalityKind gesteuert wird.

Die Rolle der Parameter übernehmen in der Software-Entwicklung *Artefakte* (WorkProductDefinition). Mit diesem Begriff werden alle in einem Software-Projekt benötigten und entstehenden Dokumente, Modelle und Code-Fragmente bezeichnet. In der Regel entstehen Artefakte unter Verwendung von *Werkzeugen* (ToolDefinition). *SPEM* ermöglicht es zu definieren, welche Aufgaben Werkzeuge erfordern (usedTool) und für welche Artefakte diese Werkzeuge zuständig sind (managedWorkProduct). Die Artefakte werden in *SPEM* nicht isoliert voneinander betrachtet, sondern können über WorkProductDefinitionRelationship verknüpft werden. Diese Beziehung ermöglicht es, auf flexible Weise Artefakte aufeinander aufzubauen.

Ein weiteres wichtiges Element eines Vorgehensmodells ist eine *Rolle* (RoleDefinition), die eine Person bei ihrer Projekt-Teilnahme annehmen kann. Mit einer Rolle werden bestimmte Qualifikationen (Qualification) wie zum Beispiel *UML*-Kenntnisse verknüpft, die für das Bearbeiten einer Aufgabe in einem Software-Entwicklungsprozess erforderlich sind. Zusätzlich erlaubt *SPEM*, zuständige Rollen für Aufgaben (Default_TaskDefinitionPerformer) und Artefakte (Default_ResponsibilityAssigment) festzulegen.

Neben Aufgaben, Artefakten, Werkzeugen und Rollen weisen die meisten Vorgehensmodelle Strukturen zur zeitlichen und inhaltlichen Einordnung von Aufgaben auf. Dafür wird in der Software-Entwicklung der Begriff *Phase* verwendet. Das Diagramm in der Abbildung 2.18 enthält – auf den ersten Blick – keine entsprechende Meta-Klasse. Der Grund dafür ist die sehr unterschiedliche Auslegung dieses Begriffs in verschiedenen Vorgehensmodellen. Eine Phase kann einerseits fest mit Aufgaben verbunden sein und eine eindeutige zeitliche Einordnung besitzen wie in einem Wasserfall-artigen Vorgehensmodell. Andererseits kann es sich um eine Prioritätsgetriebene Sammlung von Aufgaben handeln, die in bestimmten Zeitabständen zyklisch abgearbeitet wird (agiles Vorgehensmodell *Scrum*).

Die vielfältige Verwendung des Phasen-Begriffs führte dazu, dass zunächst die starre Konstruktion einer Phase mit Hilfe von *Disziplinen*[75] in der aktuellen Version des Standards aufgegeben und allgemein mit Kategorien (Category) umgesetzt wurde. Auf diese Weise können Kategorien zur Strukturierung von Aufgaben aber auch anderer wichtiger Prozess-Elemente verwendet werden, die von der Meta-Klasse DescribableElement abgeleitet wurden. Diese Flexibilität kommt insbesondere dem Vorgehensmodell *EOS*[76] entgegen, da *SPEM* eine Einordnung von Phasen zu den Artefakten wie

[75] Diese Bezeichnung kommt aus dem Vorgehensmodell *RUP* und entspricht eher der Wasserfall-artigen Ausprägung dieses Begriffs [Hes01].

[76] Siehe Kapitel 2.3.2.

Komponenten oder Module erlaubt.

Beispiele von Vorgehensmodellen und anderen Prozessen, die mit *EPF* beschrieben wurden, können von der *EPF*-Webseite *http://www.eclipse.org/epf/* heruntergeladen werden. So wurden mit diesem Framework sowohl eine offene Version von *RUP* mit dem Namen *OpenUP* als auch mehrere Vertreter der agilen Vorgehensmodelle wie *Scrum* umgesetzt.

3

Ontologien in der Software-Technik und OBSE

Was eine Ontologie ist, wurde in dem einführenden Kapitel 2.2 vorgestellt. Darauf aufbauend beschäftigt sich dieser Abschnitt mit der Frage: *Wie* können Ontologien in der Software-Technik eingesetzt werden?

Zunächst folgt eine Übersicht über den aktuellen Forschungsstand, indem verschiedene *Szenarien* der Ontologie-Nutzung in der Software-Technik betrachtet werden. Anschließend wird die Idee der *Ontologie-basierten Software-Entwicklung* vorgestellt (Kapitel 3.2) und daran anknüpfend wird betrachtet, wie sich dieses Verfahren in das entsprechende Forschungsfeld einordnet und sich gegenüber anderen Einsatzmöglichkeiten für Ontologien abgrenzt.

3.1 Szenarien der Ontologie-Nutzung

Eine Betrachtung der aktuellen Publikationen zum Thema Ontologie in der Software-Technik zeigt ein breites Spektrum an Ansätzen auf. Grundsätzlich werden Ontologien dabei zur Repräsentation des strukturellen Aufbaus einer Domäne, zur Reduktion der konzeptuellen und terminologischen Ambiguität und zum Wissensaustausch über eine Domäne verwendet [CRP06]. Sie werden auf allen im Kapitel 2.2.3 vorgestellten Abstraktionsebenen eingesetzt und können dadurch sowohl einzelne Anwendungen als auch ganze Themenkomplexe der Software-Technik unterstützen.

Für eine Klassifikation der Ontologie-Nutzung ist es wichtig, zunächst zwischen zwei Einsatzbereichen zu unterscheiden: *Ontologie zur Beschreibung der Domäne Software-Technik* und *Ontologie als ein Artefakt.*

3.1.1 Ontologie zur Beschreibung der Domäne Software-Technik

Ontologien in diesem Bereich repräsentieren das Wissen über die Software-Technik entweder generisch oder fokussiert auf die einzelnen Teilgebiete dieser Disziplin. Zu den generischen Ansätzen kann unter anderem die ontologische Repräsentation von SWEBOK[1] – aufgestellt von Sicilia et al [SCG+05] – gerechnet werden. Bei Ansätzen wie diesem wird eine anerkannte Begriffssammlung als Ausgangsbasis für die Bildung einer ontologischen Struktur in Form von Kategorien benutzt.

Beispiele für spezifische Ontologien finden sich in der Literatur für die Bereiche Anforderungsanalyse, Software-Entwurf, -Qualität und -Wartung, Software-Entwicklungsprozesse und -Management. Zum Beispiel kann man im Bereich der Software-Qualität in der Arbeit von Boehm und In [BI96] die ersten Merkmale für eine entsprechende Ontologie erkennen. Die Autoren sprechen noch nicht explizit von einer Ontologie, sondern listen wichtige Konzepte aus dem Bereich Software-Qualität – *Qualitätsmerkmale* – auf und stellen Beziehungen zwischen diesen dar. Durch die Akzeptanz und Übernahme der Begriffe aus der Fachwelt kann diese Aufstellung nachträglich als eine Ontologie betrachtet werden.

Weitere aktuellere Ontologien im Bereich der Software-Qualität wurden in den Arbeiten von Guizzardi und Falbo aufgestellt. In [FG02] wird eine entsprechende Ontologie aus den existierenden Standards, Büchern und Konsultationen mit Experten abgeleitet. Sie behandelt Konzepte wie *Metrik*, *messbare* und *nicht messbare Qualitätseigenschaften*, *Relevanz* und *Anwendbarkeit* von diesen. Zusätzlich stellt diese Ontologie über die Konzepte *Artefakt* und *Paradigma* Querbezüge zu einer weiteren spezifischen Ontologie aus dem Bereich Software-Entwicklungsprozess her, wodurch die Abhängigkeit der Software-Qualität von der Art, wie die Software entwickelt wird, verdeutlicht wird. Eine weitere Arbeit zur Software-Qualität [BW05] legt ihren Schwerpunkt auf eine einheitliche Terminologie bei der qualitativen Beurteilung der Service-orientierten Architekturen.

Der exemplarisch gewählte Bereich der Software-Qualität[2] und die Beispiele für die generischen Ontologien zeigen, dass das Szenario, bei dem Ontologien zur Beschreibung der Domäne Software-Technik verwendet werden,

[1] *SWEBOK* ist ein von *IEEE* entwickeltes *Guide to the Software Engineering Body of Knowledge*, in dem Begriffe der Software-Technik glossarartig beschrieben werden. Siehe auch http://www.computer.org/portal/web/swebok.

[2] Eine Auflistung der weiteren spezifischen Ontologien ist in [CRP06] zu finden.

vor allem einer einheitlichen Terminologie und konsistenten Verwendung der Begriffe dient. Einzelne Software-Projekte beeinflussen diese Ontologien nur indirekt, in dem sie das bessere Verständnis der Software-technischen Grundlagen und Kommunikation fördern.

Eine vergleichsweise besondere Position in der Reihe der Ontologien zur Beschreibung der Domäne Software-Technik nimmt die umfassende Studie von Biolchini et al. [BMN+07] an. Die Autoren untersuchten mehrere Review-Verfahren zur Beurteilung wissenschaftlicher Publikationen im Bereich Software-Technik und entwickelten sowohl einen einheitlichen Prozess zur Durchführung der Reviews als auch eine Ontologie, die eine Klassifikation der Publikationsinhalte und ihrer Ergebnisse unterstützen soll. Auf diese Weise erstellten die Autoren indirekt eine Ontologie für die Software-Technik, die analog zu den oben vorgestellten Verfahren eine Vereinheitlichung und bessere Vergleichbarkeit – in diesem Fall der Review-Ergebnisse – erreichen soll. Biolchini et al. kombinieren neue, Review-spezifische Konzepte mit bereits vorhandenen Ontologien der Software-Technik.

3.1.2 Ontologie als Artefakt

Dieses Szenario spielt eine zentrale Rolle in der aktuellen Forschung im Zusammenhang mit Ontologien in der Software-Technik. Analog zu den anderen Artefakten wie Dokumentation, Modelle oder Code wird eine Ontologie in diesem Fall zur Herstellung oder während der Nutzung von Software-Produkten verwendet. Auf Grund der vielen in diesem Bereich existierenden Publikationen kann die Verwendung von Ontologien als Software-Artefakt in folgende Unterszenarien unterteilt werden (siehe auch [CRP06]):

- Ontologie als Artefakt *während der Software-Entwicklung*
  - zur Software-Herstellung
  - für die Software-Herstellung unterstützende Prozesse
- Ontologie als Artefakt *während der Laufzeit*
  - als Teil der Systemarchitektur
  - als Informationsquelle

Ontologie als ein Artefakt zur Software-Herstellung

Einen guten Einblick in dieses Szenario liefert Knublauch [Knu04], indem er an einem praktischen Beispiel aus dem Semantischen Web Wege aufzeichnet, wie eine Ontologie in jeder Phase der Software-Entwicklung ein-

gesetzt werden kann. Typischerweise konzentrieren sich detailliert ausgearbeitete Ansätze für dieses Szenario auf bestimmte Phasen des Software-Entwicklungszyklus. So schlagen Oliveira vor, Ontologien in der Analyse- und Entwurfsphase zu verwenden, um den Entwicklern Wissen über die Domäne zur Verfügung zu stellen, das von Domänen-Experten in die Form einer Ontologie gegossen wurde [OVRT06]. Die Autoren schlagen eine Kombination aus einer Aufgaben- und Domänen-Ontologie[3] vor, wobei die Aufgaben-Ontologie zur Erstellung von Anwendungsfällen in der Analyse-Phase und die Domänen-Ontologie hauptsächlich zum Erzeugen eines Datenbank-Schemas in der Entwurfsphase verwendet wird.

Ein anderes Konzept für die Wiederverwendung des während der Analyse-Phase gesammelten Wissens wird von Decker et al. [DRR+05] vorgestellt. Die Autoren benutzen *Semantische Wikis*[4], um zum Beispiel Anwendungsfälle eines Projekts festzuhalten und anderen Projekten zur Verfügung zu stellen.

Laut Cranefield und Pan [CP07] können *RFD*-basierte Ontologien im Bereich der Model-basierten Software-Entwicklung eine unterstützende Rolle einnehmen. Die Autoren schlagen vor, sowohl eine Analyse der *MOF*-basierten Modelle als auch Transformationen zwischen diesen auf der *RDF*-Ebene durchzuführen. Der zweite Punkt ist das Besondere an diesem Verfahren, da *OMG* für diesen Zweck, wie im Kapitel 2.1.4 vorgestellt, den *MOF QVT*-Standard vorsieht. Cranefield und Pan argumentieren, dass während Standard-konforme Implementierungen sich noch in der Entwicklung befinden, ausgereifte *RDF*-Werkzeuge wie *Jena rule language* eine Alternative bieten, um Transformationen im *MDA*-Umfeld durchzuführen. Um dies zu ermöglichen, stellen sie Transformationen zwischen *XMI* und *RDF* vor.

Für bestimmte Bestandteile eines Software-Produkts ist inzwischen ebenfalls eine Vielzahl an Verfahren verfügbar, die auf Ontologien als ein Artefakt zur Software-Herstellung aufbauen. So nutzen Furtado et al. [FS02] eine Ontologie, um Benutzerschnittstellen zu entwerfen. Sie erfassen zunächst alle für die Benutzerschnittstelle relevanten Konzepte einer Domäne mit Hilfe einer Ontologie und leiten daraus verschiedene Dialoge, Eingabe-

[3] Siehe Kapitel 2.2.3.

[4] Diese Wiki-Systeme unterscheiden sich von den bekanntesten Vertretern dieser Software-Systeme wie *Mediawiki* (das zu Grunde liegende Software-System von *Wikipedia*) dadurch, dass die Verknüpfungen zwischen den gespeicherten Inhalten annotiert werden. Dadurch entsteht eine semantische Klassifikation der Inhalte (siehe auch Beispiel auf der Seite 34).

masken usw. in Form von Modellen ab, die in einem weiteren Schritt zum Code transformiert werden.

In einem weiteren Verfahren schlagen Zimmer und Rauschmayer [ZR04] vor, Ontologien zum effizienten Umgang mit dem Quellcode einzusetzen. Ihre Idee basiert auf einer externen Annotierung der Code-Bestandteile mit Begriffen aus einer generischen, vorgegebenen Ontologie. Dadurch soll die bessere Navigation und semantische Suche im Quellcode ermöglicht werden, was vor allem für den Code-zentrierten agilen Software-Entwicklungsprozess Vorteile bringen soll.

Ontologie als Artefakt für die Software-Herstellung unterstützende Prozesse

Die Herstellung von Software ist in der Regel mit begleitenden Tätigkeiten wie Prozess-, Konfigurations-Management und Qualitätssicherung verbunden. Eine Ontologie kann in diesem Aufgaben-Bereich zum Beispiel verwendet werden, um das Wissen und die Erfahrungen, die während der Durchführung eines Projekts auf der Management-Seite gesammelt wurden, anderen Projekten zur Verfügung zu stellen. So schlagen Nour et al. [NHM00] gleich drei Ontologien vor. Eine, um Informationen über die Fähigkeiten zu sammeln, die zur Durchführung von bestimmten Aufgaben erforderlich waren. Eine weitere Ontologie, um die tatsächlich entstandene Prozess-Struktur aufzunehmen, nach der in einem Projekt vorgegangen wurde. Schließlich die dritte Ontologie über Artefakte, die in diesem Projekt entstanden sind. Diese Informationsquellen sollen einem Projekt-Manager einen Einblick in andere Projekte ermöglichen und ihn bei der Planung des aktuellen Projekts unterstützen.

Unter dem Namen *Kumbang* präsentieren Asikainen et al. ein Verfahren, mit dem verschiedene Konfigurationen eines (Software-)Produkts modelliert werden können [AMS07]. Dabei haben die Autoren Produktlinien im Blick, deren verschiedene Ausprägungen mittels einer Ontologie einheitlich erfasst werden sollen. Die Technik baut auf bereits existierenden Konfigurationsmöglichkeiten in diesem Bereich auf und verallgemeinert diese, indem sie Konstrukte für *Komponenten*, *Features*[5], *Attribute* und *Schnittstellen* zwischen diesen anbietet. Durch die weitgehende Verallgemeinerung wird außerdem erreicht, dass *Kumbang* sich auch auf andere konfigurierbare

[5]Mit *Featurs* werden in der Entwicklung von Produktlinien relevante Problemstellungen einer Domäne bezeichnet. Die Anforderungen an einem Produkt aus einer Produktlinie können mittels der gewählten *Features* festgelegt werden.

Produkte wie Computer oder Automobile erweitern lässt.

Ontologie als ein Teil der System-Architektur

In diesem Szenario werden Ontologien als eine zentrale Komponente eines Software-Produkts verwendet und beeinflussen sein Verhalten während der Laufzeit. Entsprechende Software-Systeme werden in der Literatur oft als *Ontologie-getrieben* bezeichnet. Typische Beispiele dafür, die außerdem als erste praktische Anwendungen von Ontologien in der Software-Technik angesehen werden können, sind *Wissens-basierte Systeme* (*Expertensysteme* bzw. *entscheidungsunterstützende Systeme*). Ein Teil ihrer Architektur bildet eine Wissensbasis, deren Umsetzung oft mittels einer Ontologie realisiert wird. Inzwischen existieren viele Systeme dieser Art. Sie unterscheiden sich in der Regel in der Domäne, die sie unterstützen, wie zum Beispiel Chemie oder Medizin. Als ein Vertreter der aktuellen Wissens-basierten Systeme mit einer Ontologie-Komponente kann *SAPPHIRE*[6] – ein System zur Überwachung und Vorhersage von Epidemien – angesehen werden.

Außerdem gewinnen Ontologien als Architektur-Bestandteil in Verfahren zur Integration von Modellen an Bedeutung. Dies ist ein komplexer Prozess, bei dem verschiedene Konflikte auftreten können und Entscheidungen für die eine oder andere Lösung getroffen werden müssen. Aus dieser Perspektive betrachtet, handelt es sich dabei um eine Unterart der Wissens-basierten Systeme, die sehr mit den entscheidungsunterstützenden Systemen verwandt sind. So stellen Bellström und Vöhringer eine allgemeine Vorgehensweise zur Integration von konzeptuellen Modellen vor, bei der sogar zwei unterschiedliche Ontologien eingesetzt werden [BV09]. Seinen Ursprung hat dieses Verfahren in der Integration von *CPL*-Modellen und es soll im *OBSE*-Prozess zur Bewältigung der Integrationsaufgaben eingesetzt werden.

Der Ablauf dieser Integration ist in Abbildung 3.1 schematisch[7] dargestellt. Für Konzepte stehende Elemente eines Modells werden in diesem Verfahren paarweise verglichen. Nach dem linguistischen Vergleich (linguistic comparison) wird bei zwei sprachlich unterschiedlichen Begriffen, eine Domänen-Ontologie eingesetzt, die zwei Modell-Elemente auf ihre semantische

[6] Das Akronym *SAPPHIRE* steht für Situational Awareness and Preparedness for Public Health Incidences and Reasoning Engines. Für weitere Informationen siehe www.phinformatics.org/ResearchProjects/SAPPHIRE/tabid/76/Default.aspx

[7] Vereinfachte Darstellung aus der Präsentation von Jürgen Vöhringer aus Klagenfurt (gemeinsamer Workshop in Bad Endorf).

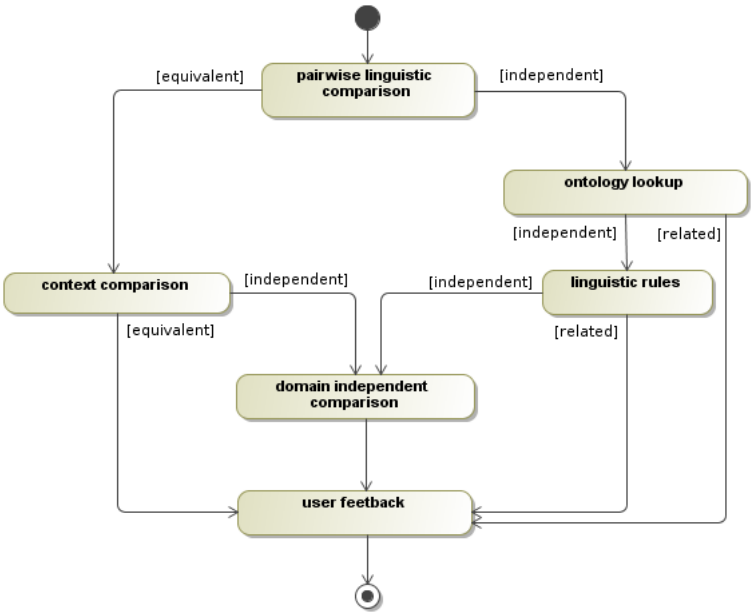


ABBILDUNG 3.1: Integrationsverfahren von Bellström und Vöhringer

Äquivalenz prüft. Sollten diese Elemente auf dieser hohen Ebene beim Vergleich noch als unabhängig klassifiziert werden, wird zunächst mit Hilfe von Wortbildungsregeln wie Teilwort usw. nach Gemeinsamkeiten gesucht (linguistic rules). Wenn dieser Vergleich ebenfalls keine Gemeinsamkeiten feststellt, wird die Umgebung der Elemente analysiert (context comparison). Schließlich wenn sie weiterhin als unterschiedlich erkannt werden, kann eine zweite, Domänen-unabhängige Ontologie befragt werden (domain independent comparison). Dabei handelt es sich um eine lexikalische Taxonomie *WordNet*[8]. Mit ihrer Hilfe lässt sich auf der rein sprachlichen Ebene die semantische Nähe der Bezeichner von Modell-Elementen bestimmen. Schließlich präsentiert das Verfahren seine Integrationsvorschläge dem Experten, der sie akzeptieren oder verwerfen kann.

Ergänzend zu den in diesem Bereich dominierenden Wissens-basierten Systemen werden immer häufiger Ontologien in die Architekturen von Systemen integriert, deren primäre Aufgabe nicht in der Wissensverarbeitung liegt. In ihrem Vorschlag für einen Ontologie-basierten Anwendungs-Server[9] setzen Oberle et al. [OESV04] eine Ontologie ein, um die Administration

[8] http://wordnet.princeton.edu
[9] Sehr verbreitet ist auch die englische Bezeichnung *Application Server*.

des Servers zu unterstützen und vor Allem die Validierung von Komponenten-Abhängigkeiten und Konfigurationen der Anwendungen durchzuführen. Dieser Ansatz wurde in späteren Arbeiten auf die Kommunikation in großen Software-Systemen sowie Konfiguration und Validierung in Web-Services ausgebaut [OLG+06].

Eine weitere Einsatzmöglichkeit von Ontologien als ein Teil der Architektur wird unter dem Namen *Ontology Driven Architecture (ODA)* vom *W3C*-Konsortium vorangetrieben. *ODA* wird als eine Erweiterung von *MDA* angesehen, bei der die Software-Entwicklung mit Techniken aus dem Bereich des *Semantischen Webs* angereichert werden soll [TPO+06]. Die Autoren gehen davon aus, dass der größte Anteil der benötigten Funktionalität eines Software-Produkts bereits irgendwo realisiert wurde und über Web-Services zur Verfügung gestellt werden kann. Sprachen wie *RDF* und *OWL* sollen bei einer vollständigen und exakten Beschreibung dieser Web-Services helfen und so eine Integration in die Software-Entwicklung ermöglichen.

Ontologie als Informationsquelle zur Laufzeit eines Software-Systems

Dieses Szenario wird von Systemen dominiert, die entweder eine semantische Auswertung von Information – zum Beispiel semantische Suche – oder das Verwalten von Ontologien ermöglichen. So kombinieren Finin et al. in ihrer semantischen Suchmaschine *Swoogle*[10] ca. 10000 Ontologien [FDP+05]. Diese Suchmaschine indiziert Web-Dokumente (Webseiten, Dateien), die Meta-Daten in *RDF* oder *OWL* anbieten und extrahiert zusätzlich die Beziehungen zwischen diesen Dokumenten. Zusätzlich klassifiziert sie die Suchergebnisse und ordnet sie entsprechend ihrer Ähnlichkeit ein. Eine Reihe weiterer Ansätze baut auf dieser Idee auf und verfeinert die Auswertung der in solchen Dokumenten vorhandenen Informationen [WHOS08].

Beim Erzeugen oder Verwalten von Ontologien kann es sich sowohl um Editoren wie *Protégé*[11] handeln als auch um Werkzeuge, die vorhandene Datenquellen mit Meta-Daten anreichern. Motiviert durch die Aussage von Mylopoulos, dass die drei wichtigsten Forschungsschwerpunkte im Bereich Datenbanken – *Geschwindigkeit*, *Geschwindigkeit* und *Geschwindigkeit* – den drei noch wichtigeren Schwerpunkten – *Semantik*, *Semantik* und *Semantik* – weichen [CRP06], entwickelten Rodríguez und Gómez-Pérez ein Verfahren für eine Transformation von relationalen Datenbanken zu Onto-

[10] http://swoogle.umbc.edu/
[11] http://protege.stanford.edu/

logien [RGP06]. Sie stellen eine entsprechende Sprache R_2O als ein Tool mit dem Namen *ODEMapster* zur Verfügung, mit dem sowohl eine Ontologie erzeugt werden kann als auch semantische Abfragen ausgeführt werden können. Für Anwendungen, die auf der transformierten Datenbank aufbauen, ermöglicht dieses Verfahren einen semantischen Zugang zu den gespeicherten Daten.

3.2 *OBSE*-Vorstellung

3.2.1 Einführung

Das im Mittelpunkt dieser Arbeit stehende *OBSE*-Verfahren ist ebenfalls ein Unterszenario für die Ontologie-Nutzung im Bereich der Software-Technik. Nach *OBSE* wird eine Domänen-Ontologie während der Software-Entwicklung eingesetzt. Das Verfahren kann somit in das Szenario *Ontologie als ein Artefakt zur Software-Herstellung* eingeordnet werden[12]. Wie von Knublauch demonstriert [Knu04], können Ontologien in diesem Bereich in verschiedenen Phasen der Software-Entwicklung hilfreich sein. *OBSE* unterstützt Software-Entwickler am stärksten in den frühen Projekt-Phasen – besonderes während der Anforderungsanalyse. Nach Uschold et al. können Ontologien in diesem Bereich unterschiedliche Rollen spielen [UJ99]:

Neutrales Autorensystem [13] Eine Ontologie wird in einer Sprache verfasst und kann in verschiedene andere Formate konvertiert und in unterschiedlichen Anwendungen verwendet werden.

Ontologie als Spezifikation Eine Ontologie stellt ein Vokabular bereit, das die Konzepte, Relationen und Axiome einer Domäne festlegt. Es steht den Software-Projekten zur Verfügung. Die Ontologie dient in diesem Fall als Basis für die Spezifikation und spätere Umsetzung dieser Domänen-spezifischen Anwendungen und erlaubt somit die Wiederverwendung von Elementen einer Domäne.

Einheitliche Informationsquelle Eine Ontologie ermöglicht einen einheitlichen Zugriff zu den heterogenen Informationsquellen, die in verschiedenen Repräsentationsformen verfasst und möglicherweise für Menschen oder Maschinen in ihrer ursprünglichen Form schwer oder nicht zugänglich sind.

[12] Siehe Kapitel 3.1.2.
[13] Original: neutral authoring

Ontologie-basierte Suche Eine Ontologie kann eine Suche nach Informationen in der Domäne genauer und effizienter machen (Analog zu der Suche im *Semantischen Web*).

Für *OBSE* ist ein Zusammenspiel der ersten drei Rollen besonders interessant. Mit Hilfe von Ontologien kann eine Wiederverwendungs-Infrastruktur aufgebaut werden, die es ermöglichen soll, Informationen über die Domäne aus verschiedenen Quellen (*Ontologie als einheitliche Informationsquelle*) zu gewinnen. Als Quellen können Artefakte dienen, die bereits in Projekten erstellt wurden oder weitere Modelle dieser Domänen, die möglicherweise unabhängig von Projekten entstanden sind.

Die Rolle einer Ontologie als *Spezifikation* würde es ermöglichen, das Wissen über die Domäne zentral festzuhalten und den Projekten als eine Referenz-Modellierung zur Verfügung zu stellen. Verwenden diese Projekte unterschiedliche Techniken (z.B. Programmiersprachen), ist an dieser Stelle die Rolle einer Ontologie als ein *neutrales Autoren-System* nützlich. So kann die Wiederverwendungs-Infrastruktur mit Hilfe einer Domänen-Ontologie eine neutrale Schnittstelle anbieten, um Ausschnitte aus Domänen-Modellen zwischen solchen Projekten auszutauschen.

Die *Ontologie-basierte Suche* spielt zwar für die Wiederverwendung eine untergeordnete Rolle, diese Funktion erleichtert aber einen Zugriff auf die Elemente einer Domäne und kann als eine hilfreiche Ergänzung angesehen werden.

3.2.2 Wiederverwendung auf der konzeptuellen Ebene

Wiederverwendung ist eine in der Software-Entwicklung etablierte Technik. Ihre Anwendung erstreckt sich auf verschiedene Phasen eines Software-Entwicklungsprozesses. Die Grundidee der Wiederverwendung im Kontext der Software-Technik besteht darin, bereits bekannte und in der Vergangenheit erfolgreich angewendete Konzepte, Artefakte und Werkzeuge beim Lösen neuer Probleme erneut einzusetzen. Durch diese Vorgehensweise sollen Kosten und Risiken bei der Entwicklung von neuen Software-Systemen reduziert werden.

Ein Artefakt, das während eines Software-Entwicklungsprozesses entsteht, ist in der Regel nicht automatisch für die Wiederverwendung geeignet. So ist zum Beispiel ein Code-Abschnitt – auch wenn er an vielen Stellen gebraucht werden könnte – zunächst nicht direkt wiederverwendbar[14].

[14]Eine Form der Wiederverwendung, zu der in diesem Fall oft intuitiv gegriffen wird, ist

Es erfordert zumindest eine Infrastruktur oder im allgemeineren Fall einen Prozess, die es ermöglichen, wiederverwendbare Teile zu definieren und zur Verfügung zu stellen. In dem Code-Beispiel wird das mit Hilfe von Operationen gelöst, die den Code verkapseln und ihn so an unterschiedlichen Stellen im Anwendungs-Code verwendbar machen.

Bereits an diesem kleinen Beispiel lässt sich erkennen, dass eine System-Entwicklung *mit* Wiederverwendung zunächst eine Entwicklung *für* Wiederverwendung erfordert. Prinzipiell ist die Wiederverwendung in allen Phasen der Software-Entwicklung möglich. Auf der Ebene der Programmierung gehört die Wiederverwendung inzwischen zum Standard und wird immer weiter durch neue Konzepte wie zum Beispiel *Objekt-*, *Aspekt-orientierte Software-Entwicklung*[15] oder *Komponenten-basierte Software-Entwicklung* vorangetrieben. Eine weitere Ebene, in der Wiederverwendung angestrebt wird, ist weitgehend von der jeweiligen Implementierung unabhängig und setzt bei Modellen in einem Software-Projekt, also beim Entwurf an. In diesem Bereich spielen bei der Wiederverwendung Entwurfsmuster und Code-Generatoren bis hin zur *Modell-basierten Software-Entwicklung* eine wichtige Rolle.

Ein weiterer, konsequenter Schritt ist die Einführung der Wiederverwendung auf der konzeptuellen Ebene. In diesem Fall wird angestrebt, den Aufwand bei der Analyse der Anforderungen und des zugehörigen Untersuchungsbereichs dadurch zu reduzieren, dass bei einer vorhergehenden Analyse bereits identifizierte Konzepte und Relationen einer Domäne wiederverwendet werden. Eine bekannte Technik in diesem Bereich ist die Entwicklung von *Produktlinien* [LSR07]. Sie ermöglichen es, zu einem Gegenstandsbereich unterschiedliche Anwendungen zu erstellen, die sich in ihrer Funktionalität voneinander unterscheiden[16]. Die Idee ist mit dem Zusammenstellen eines Wunsch-Fahrzeugs durch den Kunden in der Auto-Industrie vergleichbar. Mit Hilfe der *Produktlinien* können auch in der Software-Technik

die Wiederverwendung durch *copy&paste*, was allerdings berechtigter Weise als ein schlechter Programmierstil gilt.

[15] Anforderungen an einem Software-System beinhalten in der Regel Themenbereiche wie Effizienz oder Sicherheit, die quer zu der Kernfunktionalität eines Systems stehen. Für ihre Erfühlung werden Code-Fragmente verwendet, die an verschiedenen Stellen im System verstreut sind. Diese Situation erschwert die Wiederverwendung der Kern-Komponenten, insbesndere wenn sich die Anforderungen an der Effizienz oder Sicherheit ändern. Die *Aspekt-orientierte Software-Entwicklung* ermöglicht es, diese Fragmente zu kapseln und zentral zu definieren.

[16] Siehe auch Kapitel 3.1.2.

Kunden-spezifische Lösungen[17] umgesetzt werden, ohne dass der Untersuchungsbereich jeweils komplett neu analysiert werden muss.

Auf *Produktlinien* basierende Software-Entwicklung bietet eine Infrastruktur, welche die Wiederverwendung innerhalb einer Familie von funktional verwandten Anwendungen unterstützt. Das hat zur Folge, dass ein funktional anderes Software-System, das zum Teil auf den gleichen Konzepten und Relationen eines Gegenstandsbereichs basiert, von der Wiederverwendung mittels der *Produktlinien* nicht mehr profitiert. Diese Situation erfordert eine von der Funktion des Systems unabhängige, Domänen-zentrierte Art der Wiederverwendung. Um den Austausch von Anforderungen bzw. der in ihnen beschriebene Zusammenhänge zwischen verschiedenen Projekten zu ermöglichen, ist eine entsprechende Infrastruktur erforderlich, die einerseits die Sachverhalte einer Domäne in einer formalisierten Form zentral (außerhalb eines Projekts) aufnimmt und andererseits den anderen Projekten zur Verfügung stellt.

Um die Idee – Ontologien in der Software-Technik als eine Wissens-Austauschplattform zwischen mehreren Projekten zu verwenden – praktisch umzusetzen, wird als erstes eine Infrastruktur benötigt, die es ermöglichen soll, eine Domänen-Ontologie aufzubauen und den verschiedenen Projekten zur Verfügung zu stellen. Zweitens ist ein Prozess erforderlich, in dem geregelt wird, wie die Ontologie verwaltet wird und wie die Verknüpfungen mit den Software-Entwicklungsprozessen realisiert werden. Ein Vorschlag für eine entsprechende Methode mit dem Namen *Ontologie-basierte Software-Entwicklung (OBSE)*, die sowohl eine Infrastruktur als auch eine Prozessbeschreibung anbietet, wurde in Zusammenarbeit zwischen der Arbeitsgruppe Software-Technik an der Philipps-Universität Marburg und dem Institut für Angewandte Informatik an der Alpen-Adria-Universität Klagenfurt entwickelt ([BHR+07b], [BHR07a], [Bac08]).

3.2.3 *OBSE*-Grundstruktur

Ein Software-Entwicklungsprozess beschreibt Phasen, Aktivitäten, Rollen und Artefakte, die während der Entwicklung eines Software-Produkts im Rahmen eines Projekts verwendet werden bzw. anfallen[18]. Die *Ontologie-basierte Software-Entwicklung* erweitert diese Projekt-zentrierte Sicht um

[17] Ein typisches Beispiel für die Software-Systeme, die unter Verwendung von Produktlinien entwickelt wurden, sind die unterschiedlichen *Windows*-Varianten (*Home, Business, Enterprise, Ultimate*) der *Windows*-Version *Vista*.

[18] Siehe auch Kapitel 2.3.3

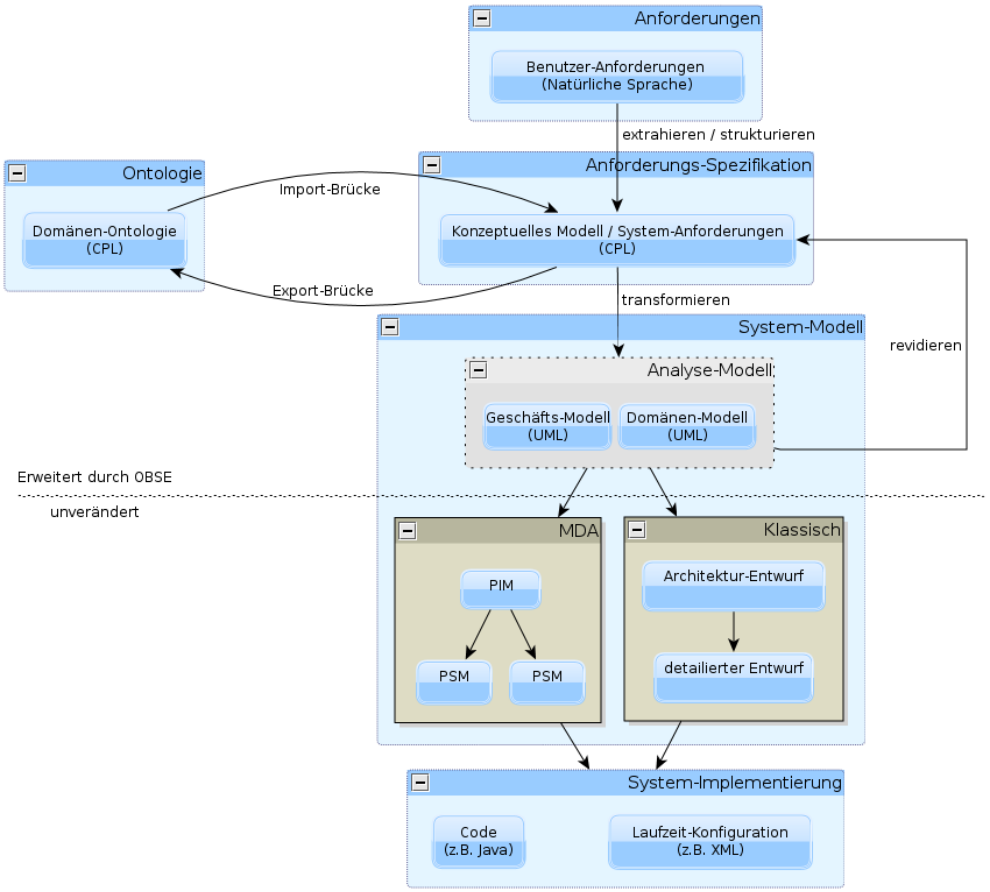


ABBILDUNG 3.2: Schematische Darstellung von *OBSE*

die Möglichkeit, Domänen-spezifische Teile von Software-Projekten auszulagern und wiederzuverwenden. Sie führt aus diesem Grund neue Projektübergreifende Elemente hinzu.

Die Abbildung 3.2 zeigt eine schematische Darstellung von *OBSE*. Auf der rechten Seite ist ein Software-Projekt repräsentiert, das stellvertretend für mehrere nach *OBSE* durchgeführte Projekte steht. Das Software-Projekt ist schematisch durch die Artefakte dargestellt, die typischerweise während der Entwicklung eines Software-Produkts entstehen.

Jede Software-Entwicklung beginnt mit einer Analyse der Anforderungen. Als Ergebnis dieser Aktivität werden für die System-Entwicklung wichtige Sachverhalte aus den Benutzer-Anforderungen extrahiert und in einer – im Vergleich zur natürlichen Sprache – formalisierten Form abgelegt.

An dieser Stelle steht einem Analysten eine breite Palette an Methoden wie zum Beispiel *Strukturierte Analyse*, *Anwendungsfall-Analyse* mit *UML* oder die *Conceptual Predesign-Methode* zur Verfügung. Die *Ontologie-basierte Software-Entwicklung* baut auf der letztgenannten auf. Für *CPM* spricht – neben der Unterstützung des Analysten durch die *NIBA*-Werkzeuge für eine natürlichsprachliche Analyse (siehe Kapitel 2.1.3) – auch die Eignung der *Conceptual Predesign Language* zur Beschreibung von Domänen-Ontologien (siehe Kapitel 4.1.3). So kann ein aus den Anforderungen gewonnenes konzeptuelles Modell[19] sowohl in der Projekt-Entwicklung als auch für einen Export in die Domänen-Ontologie verwendet werden.

Wegen dieser zentralen Stellung des konzeptuellen Modells ist es wichtig, dass dieses Modell mit den Fachverantworlichen abgestimmt wird und deren Vorstellungen über die Domäne nach Möglichkeit genau abgebildet werden. Diese Abstimmung wird durch die Verwendung von *CPL* im Rahmen der *Conceptual Predesign-Methode* gefördert (siehe Kapitel 2.1.3 und [MK02]).

Der auf die Anforderungsanalyse folgende Schritt auf der Projektseite ist eine Transformation des konzeptuellen Modells zu einem Analyse-Modell, das ein Bestandteil des System-Modells ist. Dabei steht die Bezeichnung System-Modell in der Software-Technik zusammenfassend für alle Modellierungs-Artefakte, die in einem Projekt für eine spätere System-Implementierung erstellt werden. Neben dem Analyse-Modell beinhaltet ein System-Modell weitere Modelle, die während des Projekt-Ablaufs mit Plattform- und Implementierungs-spezifischen Details angereichert werden. Im Falle der MDA sind das PIM und PSM[20]. Im klassischen Fall sind das die unterschiedlich detaillierten Entwürfe des Systems.

MDA schreibt die Verwendung von *UML* vor. Aber auch sonst wird eine überwiegende Anzahl an Software-Projekten Objekt-orientiert entwickelt. Aus diesem Grund wird das Analyse-Modell in der Regel in *UML* verfasst[21]. Um diese Projekte zu unterstützen und für sie einen Einstieg in die Ontologie-basierte Software-Entwicklung zu ermöglichen, bietet *OBSE* eine Möglichkeit, *CPL*-Modelle und insbesondere das Konzeptuelle Modell nach *UML* zu transformieren[22].

[19] Mit Hilfe von Modellen formalisierte Anforderungen werden in der Literatur oft auch als System-Anforderungen bezeichnet [Som07].

[20] Siehe auch Kapitel 2.1.1.

[21] Dabei wird in der Regel zwischen statischen (System-Modell) und dynamischen (Geschäfts-Modell) Aspekten des zukünftigen Systems unterschieden [CRP06].

[22] Vorausgehende Arbeiten sind [Ker01], [MK02] und [KVH+05]; die aktuelle Version der Transformation wird im Kapitel 4.2.4 vorgestellt.

Im weiteren Projekt-Verlauf werden die *UML*-Diagramme des System-Modells ausgearbeitet und verfeinert. Bei diesen Aktivitäten ist ein Szenario realistisch, bei dem sie nicht nur mit Implementierungs-spezifischen Details sondern auch mit weiteren Informationen über die Domäne angereichert werden. Diese in der Praxis übliche Vorgehensweise erfordert, dass die neuen Domänen-spezifischen Elemente des System-Modells wieder in das konzeptuelle Modell übertragen werden können. Moderne Techniken wie *MDA* sind darauf ausgerichtet, die Modelle eines Projekts durch Transformationen konsistent zu halten. Im klassischen Fall wird die Konsistenz ebenfalls vorausgesetzt. Deswegen wird in *OBSE* als Quelle für den Vergleich mit dem Konzeptuellen Modell das Analyse-Modell verwendet. Auch wegen seines höheren Abstraktionsgrads und fehlenden Platform- und Implementierungs-spezifischen Informationen eignet es sich gut für den Schritt revidieren, bei dem Elemente eines *UML*-Modells (in diesem Fall des Analyse-Modells) nach *CPL* konvertiert werden [Ruß07].

Die System-Implementierung wurde zum Zwecke einer geschlossenen Darstellung eines Software-Entwicklungszyklus in die schematische Darstellung (siehe Abbildung 3.2) aufgenommen. Der *OBSE*-Prozess bietet keine spezielle Unterstützung für diesen Schritt. Es ist die Aufgabe des Vorgehensmodells bzw. der gewählten Technologie wie zum Beispiel *MDA*, die Software-Entwickler beim Übergang von Modellen zum Code zu leiten.

Der wichtigste Bestandteil der *OBSE*-Infrastruktur – eine Domänen-Ontologie – befindet sich außerhalb der Projekte (siehe linke Seite in Abbildung 3.2). Zusammen mit den Import- und Export-Brücken[23] ermöglicht sie es, Teile[24] der konzeptuellen Modelle von einem oder mehreren Projekten in andere Projekte zu übertragen. Dabei bereitet ein konzeptuelles Modell des Projekts – analog zum Verkapseln von Code-Teilen als Operationen – Domänen-spezifische Elemente für den Export in die Domänen-Ontologie und dadurch für die spätere Wiederverwendung vor. Das eigentliche Übertragen von diesen Elementen übernehmen die Import- und Export-Brücken. Die Export-Brücke speist eine Domänen-Ontologie mit den Elementen aus den konzeptuellen Modellen. Die Import-Brücke versorgt dagegen Projekte mit Informationen über die Domäne und erlaubt es, Elemente (Konzepte, Beziehungen) in das konzeptuelle Modell des Projekts zu übernehmen.

[23] Die Bezeichnungen *Import* und *Export* wurden ausgehend von Projekten festgelegt. So steht *Import* für eine Übernahme von Elementen in ein Projekt und *Export* umgekehrt für die Übertragung von Elementen in die Domänen-Ontologie.

[24] Bedingt durch die unterschiedliche Aufgabenstellung überlappen sich die Gegenstandsbereiche von mehreren Projekten in einer Domäne in der Regel nur zum Teil.

3.2.4 Wiederverwendung mittels *OBSE*

Das Ziel von *OBSE* ist es einerseits, eine Domänen-Ontologie so aufzubauen, dass sie sich aus bereits in konkreten Projekten entwickelten und implementierten Modellen einer Domäne zusammensetzt. Andererseits soll die Anforderungsanalyse der neuen Projekte in dieser Domäne dadurch unterstützt werden, dass gewählte Elemente oder sogar ganze Modell-Ausschnitte aus der Domänen-Ontologie übernommen und in das konzeptuelle Modell eines Projekts integriert werden. Somit ist es besonderes für die Projekte geeignet, die sich eine *gemeinsame Domäne* teilen.

Wird eine bestimmte Technik für die Wiederverwendung eingesetzt, so ist dieser Einsatz zunächst mit einem zusätzlichen Aufwand verbunden, der investiert werden muss, um die entsprechenden wiederverwendungsfähigen Elemente aufzubereiten. Ein Blick auf das Beispiel der prozeduralen Wiederverwendung zeigt, dass ein Teil des Codes zunächst als eine Operation deklariert werden muss. Für den Rumpf der Operation muss aus diesem Grund ein *zusätzlicher Code* geschrieben werden. Insgesamt wird allerdings erwartet, dass diese Investition sich auszahlt. Dabei spielen zwei Faktoren eine wichtige Rolle: *Reduzierung des Gesamtaufwands* und *Qualitätsgewinn*.

Die *Reduzierung des Gesamtaufwands* wird durch eine wiederholte Verwendung der Operation an mehreren Stellen im Code erreicht. Aufwand für den Methoden-Rumpf fällt nur einmalig an. Danach erfordert jede Verwendung der Operation weniger Aufwand. Zusätzlich zu dem direkten Nutzen der Wiederverwendung, bringt sie in der Regel auch indirekt messbare Vorteile mit sich. Bei der prozeduralen Programmierung ist das zum Beispiel die verbesserte Lesbarkeit des Codes. Auch wenn eine Operation nur ein Mal aufgerufen wird und somit die direkte Aufwands-Bilanz nicht zu Gunsten der Wiederverwendung ausfällt, wird über bessere Lesbarkeit und damit verbundene bessere Wartbarkeit[25] eines Software-Produkts die gesamte Bilanz der Software-Qualität verbessert.

Übertragen auf die Wiederverwendung auf der konzeptuellen Ebene mittels *OBSE* entsteht zusätzlicher Aufwand vor allem durch die Verwendung der Export-Brücke. Das ist ein Schritt im Prozess, der bei einer klassischer Vorgehensweise nicht erforderlich wäre. Die Transformation von *UML* zu *CPL* (bezeichnet mit revidieren, siehe Abbildung 3.2) kann auch als ein Mehraufwand angesehen werden, allerdings sorgt dieser Schritt gleichzeitig dafür, dass das konzeptuelle Modell des Projekts auf dem aktuellen Stand

[25] Wartbarkeit gehört unter anderem zu den Qualitätsmerkmalen eines Software-Produkts [Som07].

gehalten wird.

Dem investierten Aufwand steht bei *OBSE* zunächst Zeitgewinn gegenüber, der sich durch einen Import (Import-Brücke) von Elementen aus der Domänen-Ontologie während der Anforderungsanalyse ergibt. Das trägt zur *Reduzierung des Gesamtaufwands* bei. Sie ist um so größer, je mehr Elemente beziehungsweise ganze Modell-Ausschnitte direkt in das konzeptuelle Modell eines Projekts übernommen werden können. Auch die Anzahl der Projekte, die die Import-Brücke anwenden, verbessert die Bilanz der Wiederverwendung, da dadurch die Anzahl der wiederverwendeten Elemente steigt. Zusätzlich führt diese Vorgehensweise zu einem strukturellen Gewinn, der sich aus der Vereinheitlichung der Modellierung gleicher Sachverhalte ergibt.

OBSE setzt auf einer Ebene auf, in der die Fehlerfreiheit bei der Übersetzung der Anforderungen und die Qualität der daraus erstellten Modelle für die spätere Produkt-Qualität eine entscheidene Rolle spielen. Bereits im Jahre 1979 machte Tom DeMarco darauf aufmerksam, dass die Kosten für die Fehlerbehebung um so höher sind, je früher dieser Fehler in einem Projekt gemacht wurde[26] [DeM79]. Interessanterweise wies er ebenfalls darauf hin, dass die meisten Fehler während der Anforderungsanalyse gemacht werden (siehe Abbildung 3.3).

Diese Besonderheit bei der Verteilung der Fehlerhäufigkeit und der Fehlerbehebungskosten auf die von ihm untersuchten Artefakte der Software-Entwicklung (Anforderungen, Code, Entwurf), führt dazu, dass die indirekten Auswirkungen der Wiederverwendung auf der konzeptuellen Ebene einen hohen Stellenwert bekommen, da sie in der Regel die Qualität des zukünftigen Software-Produkts nachhaltig beeinflussen.

Die Import-Brücke kann zusätzlich zur direkten Wiederverwendung von Elementen dazu beitragen, dass mögliche Lücken in den Anforderungen aufgedeckt werden. Da die Ontologie mit der wachsenden Zahl der exportierenden Projekte immer mehr und detailliertere Informationen über die entsprechende Domäne enthält und diese Informationen in der Form von Konzepten und Relationen über die Import-Brücke den Projekten zur Verfügung gestellt werden, ist zu erwarten, dass die im konzeptuellen Modell möglicherweise fehlenden oder inkonsistenten Elemente auffallen und zusätzlich mit dem Fachverantwortlichen diskutiert und ggf. übernommen werden. Wird zum Beispiel ein Sachverhalt in der Domänen-Ontologie anders modelliert als in dem konzeptuellen Modell eines Projekts, könnte das ein Hinweis auf

[26] Eine neuere Untersuchung [Car00] hat diese Einschätzung weitgehend bestätigt.

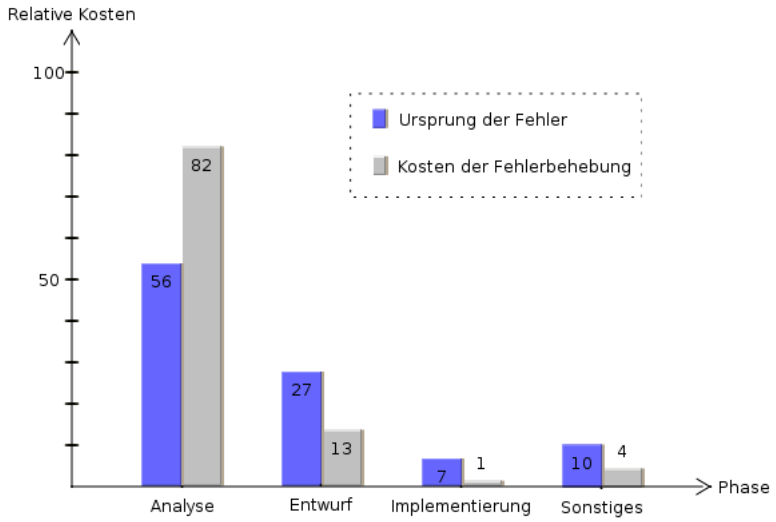


ABBILDUNG 3.3: Kosten und die Häufigkeit von Fehlern in den Software-Projekten [DeM79]

die fehlerhafte Modellierung sein. Wichtig ist in diesem Zusammenhang, dass ein Analyst bzw. System-Entwickler durch die Anwendung der Import-Brücke auf diese Fälle aufmerksam wird. Setzt er entsprechende Elemente im Software-Projekt anders als in der Domänen-Ontologie um, dann handelt es sich um eine projektspezifische Abweichung, die in diesem Fall bewusst hingenommen wurde und gegebenenfalls zu begründen wäre.

Von den anderen Verfahren aus dem Szenario *Ontologie als ein Artefakt zur Software-Herstellung* unterscheidet sich *OBSE* hauptsächlich dadurch, dass die konzeptuelle Wiederverwendung im Vordergrund steht. So ermöglicht zwar das Verfahren von Oliveira [OVRT06] das Erzeugen von Software-Artefakten wie Anwendungsfällen oder einem Datenbank-Modell aus einer Ontologie. Allerdings muss diese Ontologie speziell für das jeweilige Projekt erstellt werden und stellt somit ein zusätzliches Artefakt dar. Außerdem kann es vorkommen, dass beim Erstellen einer Domänen-Ontologie Zeit in Konzepte und Relationen investiert wird, die möglicherweise später in dem Projekt nicht mehr verwendet werden.

Der Gedanke der Wiederverwendung und eines Wissenstransfers zwischen den Projekten findet sich dagegen in der Arbeit von Decker at al [DRR+05]. Der Unterschied zu *OBSE* liegt in der verwendeten Infrastruktur. Während *Semantische Wikis*, welche Decker für den Aufbau der Ontologie vorschlägt, eine mit Meta-Daten versehene Verknüpfung der Beschrei-

bungen von Begriffen darstellen und somit zu den *strukturierten Glossaren* (siehe Abbildung 2.13 im Kapitel 2.2.4) gehören, baut die Domänen-Ontologie in *OBSE* auf der konzeptuellen Modellierungssprache *CPL*[27] auf. Diese Repräsentationsform ermöglicht eine direkte Integration in den Modellierungsprozess, die auf der Benutzung von Brücken und Transformationen zwischen *CPL* und *UML* basiert. Die Semantischen Wikis tragen dagegen eher einen informativen Charakter und werden in dem Modellierungsprozess indirekt verwendet [KVV06].

[27] Die Einordnung von *CPL* in die Landschaft der Wissensrepräsentatiosformen wird im Kapitel 4.1.3 behandelt.

4

OBSE-Infrastruktur: Prozess-Bausteine

Nachdem die grundlegenden Elemente[1] von *OBSE* (Konzeptuelles und Analyse-Modell, Domänen-Ontologie und Brücken) im Kapitel 3.2.3 vorgestellt wurden, geht dieses Kapitel ins Detail und zeigt, wie diese Bestandteile konzipiert sind, und wie die verschiedenen dafür eingesetzten Techniken aufeinander abgestimmt werden. Anhand eines durchgehenden Beispiels soll außerdem verdeutlicht werden, wie verschiedene Artefakte im Rahmen von *OBSE* entstehen, verwendet werden, und wie die Übergänge zwischen diesen funktionieren.

Während in Abbildung 3.2 (auf Seite 75) eine Prozess-orientierte Sicht auf die Grundstruktur von *OBSE* vorgestellt wurde, stellt das Diagramm in Abbildung 4.1 die Elemente der Infrastruktur aus der Sicht ihrer Modellierung dar.

Die *OBSE*-Infrastruktur baut auf der mehrschichtigen Meta-Modellierungs-Architektur auf[2]. Eine Grundlage für alle Infrastrukturelemente des *OBSE*-Prozesses bilden somit Meta-Modelle, die ihrerseits mit Hilfe von EMOF[3], der vereinfachten Version von MOF, beschrieben werden.

Die Modellierung der *OBSE*-Infrastruktur (Abbildung 4.1) kann – entsprechend der *MOF*-Architektur – in insgesamt drei Schichten (M1 bis M3) unterteilt werden. *EMOF/Ecore* bilden als ein *Super-Meta-Modell* die

[1] In diesem Kapitel werden mit *Elementen* zusammenfassend Bausteine der Infrastruktur wie Meta-Modelle, darauf aufbauende Modelle und Transformationen bezeichnet. Es sind *nicht* die Ausprägungen von diesen Modellen und deren Elemente gemeint.

[2] Siehe auch Kapitel 2.1, Seite 10.

[3] Genau betrachtet wird zwecks Werkzeug-Entwicklung eine *EMOF*-ähnliche Implementierung *ECore* als Meta-Modell verwendet. Sie ist im Rahmen der *Eclipse*-Plattform entstanden und bietet im Unterschied zu *EMOF* eine breite Werkzeugunterstützung. Die beiden Meta-Modelle unterscheiden sich dagegen minimal [Wil08].

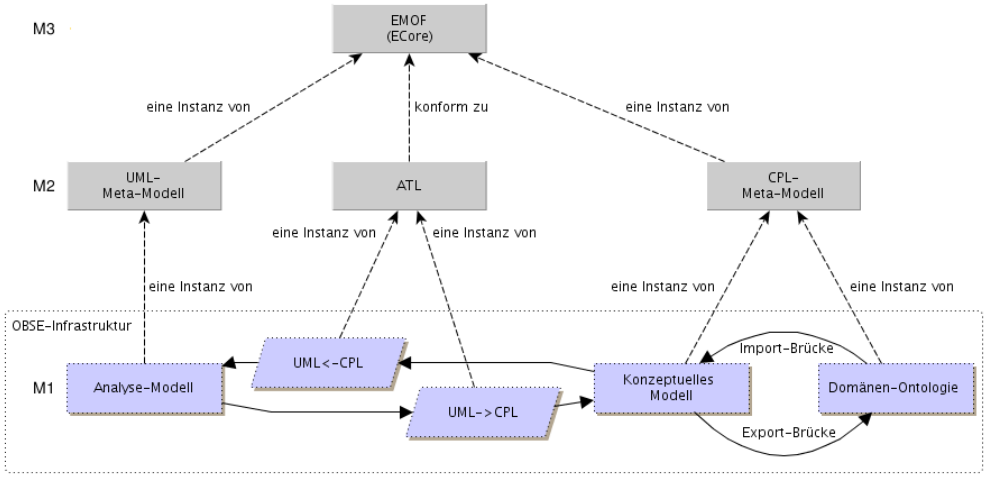


ABBILDUNG 4.1: *OBSE*-Infrastruktur

oberste Schicht dieser Architektur M3. Eine Ebene tiefer M2 stehen Meta-Modelle für Sprachen und Transformationen. Die Infrastruktur selbst stellt Modelle zur Verfügung, mit denen Konzepte einer Domäne und die zugehörigen Software-Objekte beschrieben werden. Entsprechend dieser Aufgabe stehen alle *OBSE*-Infrastruktur-Elemente auf der Ebene M1.

Vertikal lässt sich dieses Diagramm anhand der drei Meta-Modelle in der Ebene M2 in drei Stränge unterteilen. Auf der linken Seite steht das vom UML-Meta-Modell instantiierte Analyse-Modell. In der Mitte sind die Transformationen von *CPL* nach *UML* und zurück abgebildet. Sie bauen auf der *Atlas Transformation Language*[4] auf. Rechts stehen die auf dem CPL-Meta-Modell basierenden *OBSE*-Grundelemente. Dazu zählt neben dem konzeptuellen Modell auch die Domänen-Ontologie. Die Elemente in der Ebene M1 sind so angeordnet, dass der Abstraktionsgrad der Modelle in dieser Ebene von links nach rechts zunimmt.

Die Infrastruktur des *OBSE*-Prozesses setzt gezielt Techniken ein, die auf der MOF-Architektur aufbauen. Diese Vorgehensweise soll helfen, verschiedene Elemente des Prozesses, die ursprünglich unabhängig voneinander entwickelt wurden, auf eine einheitliche Basis zu stellen. So werden sowohl

[4] ATL besitzt ebenfalls ein von *ECore* instantiiertes Meta-Modell. Da sie aber weitere Bestandteile wie eine eigene *virtuelle Maschine* mitbringt und somit mehr als ein Meta-Modell für Transformationen ist, wird in dem Diagramm in der Abbildung 4.1 statt eine Instanz von die Beschriftung konform zu bei der Beziehung zu EMOF/Ecore verwendet. Für weitere Informationen siehe Kapitel 2.1.4

alle Modelle der *OBSE*-Infrastruktur (Rechtecke) als auch Transformationen zwischen diesen (Parallelogramme) konform zu dem *Super-Meta-Modell* EMOF/Ecore beschrieben.

Als eine von *OMG* standardisierte Modellierungssprache wurde UML ab der Version 2.0 vollständig mit MOF beschrieben. Auch ein ECore-basiertes Meta-Modell dieser Sprache wurde im Rahmen des *Eclipse UML2*-Projekts erstellt. ATL als eine Sprache für Modell-zu-Modell-Transformationen, die auf Ecore-basierten Modellen aufbaut, lässt sich ebenfalls problemlos in die *OBSE*-Architektur integrieren. Nur die vorgefundene Situation bei dem letzten Baustein in der Ebene M2 – dem CPL-Meta-Modell – erforderte eine Reihe an Anpassungen, um dieses Meta-Modell ebenfalls EMOF/Ecore-konform zu machen und somit eine Integration des konzeptuellen Modells und der Domänen-Ontologie zu ermöglichen.

4.1 *CPL*-Meta-Modell für die *OBSE*-Infrastruktur

Ursprünglich für die Kommunikation über Projektanforderungen mit Fachverantwortlichen entwickelt, besaß *CPL* zunächst nur ein skizzenhaftes Meta-Modell, das hauptsächlich zur Erklärung der Sprachelemente und in Präsentationen verwendet wurde. Dieser Teil des Meta-Modells wurde in *UML*[5] verfasst. Der vollständig zur Verfügung stehende Sprachumfang von *CPL* wurde mittels eines Datenbank-Modells festgelegt, das von allen bis dahin entwickelten *CPL*-Werkzeugen genutzt wurde.

Um die Konformität zur *MOF*-Architektur zu erreichen, bestand der wichtigste Schritt zunächst darin, alle Sprachkonstrukte von *CPL* einheitlich mittels EMOF/Ecore zu beschreiben. Diese Umschreibung von einem Datenbank- zu einem *EMOF/Ecore*-basierten Meta-Modell wurde gleichzeitig benutzt, um das Meta-Modell der Sprache zu revidieren[6] und Optimierungen, aber auch Anpassungen im Bezug auf *OBSE* durchzuführen. Die Änderungen reichten von einheitlichen englischen Bezeichnern bis hin zur Spezifikation von neuen Sprachelementen.

[5] Allerdings nur in der Version 1.x dieser Sprache, die noch nicht *MOF*-konform war.
[6] Siehe auch [Rußß07]

4.1.1 Meta-Modell-Anpassungen

Strukturierung der Sprachelemente

Wie bereits im Kapitel 2.1.3 vorgestellt, sind Tabellen für jedes Hauptmodellierungselement[7] die primäre Darstellungsform von *CPL*. Die Zeilen dieser Tabellen beinhalten jeweils ein Element des entsprechenden Typs und die Spalten charakterisieren diese Elemente. Diese tabellarische Darstellungsform liefert gleichzeitig einen Anhaltspunkt zur Klassifizierung der *CPL*-Sprachelemente:

Elemente 1. Klasse Bei diesen Elementen handelt es sich um Sprachelemente, die *alleinstehend* verwendet werden können und somit die Ausgangsbasis einer Tabelle bilden.

Elemente 2. Klasse Diese Sprachelemente dienten zur Charakterisierung der *Elemente 1. Klasse* und können nur im Verbund mit diesen auftreten. Strukturell bilden sie die Spalten einer Tabelle.

Diese Klassifizierung der Modellierungselemente findet sich nicht nur in *CPL*, sondern auch in anderen Modellierungssprachen. So kann im *UML*-Meta-Modell die Meta-Klasse Class (definiert *UML*-Klassen) als ein Element 1. Klasse und die Meta-Klasse Property (definiert Attribute/Assoziationsenden) als ein Element der 2. Klasse angesehen werden. Analog zu der Klassifikation in der Auflistung oben können die *UML*-Klassen für sich allein existieren, und die Attribute müssen einer *UML*-Klasse zugewiesen werden. Sie sind somit von ihr abhängig. In *UML* werden die Elemente 1. Klasse dadurch ausgewiesen, dass sie die Klasse PackageableElement spezialisieren. In *CPL* wird die Unterscheidung der Elemente explizit mit Hilfe der abstrakten Meta-Klassen ModelingElement und ModelingElementDependent gemacht. Wie die Namensgebung bereits andeutet, spezialisieren die Elemente 1. Klasse die Klasse ModelingElement und die *abhängigen* Elemente (2. Klasse) die Klasse ModelingElementDependent (siehe Abbildung 4.2[8]).

Diese Klassifizierung trägt nicht nur zur besseren Strukturierung des *CPL*-Meta-Modells und einer transparenten Anknüpfung an die primäre Darstellungsform bei, sondern ist auch für den *OBSE*-Prozess nützlich. Auf diese Weise können Merkmale eingeführt werden, die für alle Modellierungselemente einer Klasse verfügbar sein sollen. Dies wird zum Beispiel benutzt, um alle Elemente 1. Klasse mit Beschreibungen (Attribut description in der

[7] Dingtyp, Zusammenhangstyp, Operationstyp, Cooperationstyp

[8] Diese Abbildung zeigt alle Elemente 1. Klasse von *CPL* und der Übersichtlichkeit wegen nur die Elemente 2. Klasse, die einem ThingType untergeordnet sind.

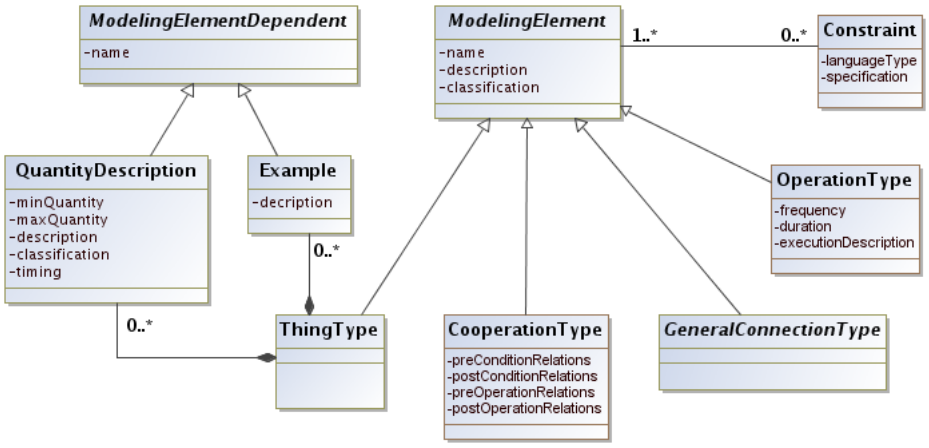


ABBILDUNG 4.2: *Elemente 1.* und *2. Klasse* in *CPL*

Meta-Klasse ModelingElement) oder Einschränkungen (Meta-Klasse Constraint) zu versehen.

Außerdem führt diese Klassifizierung zu einer hierarchischen Struktur mit wenigen Modellelementen, die als Wurzelelemente angesehen werden können. Ein Wurzelelement der Programmiersprache *Java* ist die Klasse Object. Alle weitere Klassen werden von dieser Klasse abgeleitet. Dadurch wird ermöglicht, bei Bedarf alle Klassen in *Java* einheitlich zu behandeln. Ähnlich verhält es sich mit den Modellierungselementen in der *CPL*. So ist es zum Beispiel für die Entwicklung von *OBSE*-Werkzeugen hilfreich, wenn bei einem Vergleich von zwei Glossaren über alle Elemente 1. Klasse einheitlich iteriert werden kann.

Generalisierung/Spezialisierung

Eine weitere wichtige Anpassung des *CPL*-Meta-Modells war die Einführung einer Möglichkeit zur *Generalisierung/Spezialisierung* von Dingtypen. Die Notwendigkeit für diese zusätzliche Beziehung ergab sich aus mehreren Gründen.

Auf der einen Seite stand die in der Praxis gesammelte Erfahrung, die es deutlich machte, dass die Fachverantwortlichen im Umgang mit der Generalisierung meist vertraut sind und diese von einer konzeptuellen Modellierungssprache erwarten. Auf der anderen Seite ist diese Beziehung aus der Sicht eines Entwicklers für die konzeptuelle Modellierung heute fundamental. Ein weiteres Argument ergab sich aus der anvisierten Verwendung

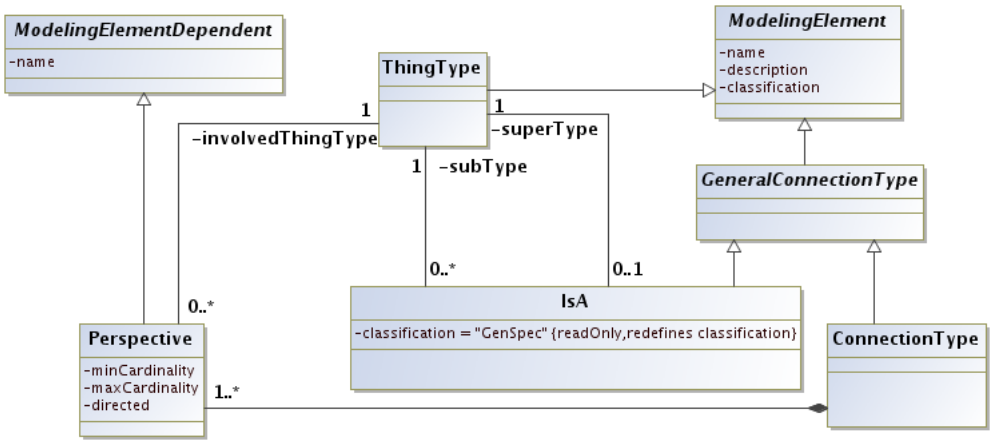


ABBILDUNG 4.3: *IsA*-Beziehung in *CPL*

von *CPL* als Ontologie-Sprache. Es ist eine Tatsache, dass die Aufstellung von Taxonomien zu den wichtigsten Modellierungstechniken beim Verfassen von (Domänen)-Ontologien zählt [Gua99]. Die Erweiterung des Meta-Modells um die Generalisierung trägt damit wesentlich zur Steigerung der Ausdruckskraft der Sprache *CPL* bei und wurde in die Sprache aufgenommen.

Bereits vor der Revision des *CPL*-Meta-Modells in Rahmen dieser Arbeit wurde Generalisierung in den Sprachumfang eingebaut. Bei dieser Lösung [MK02] handelte es sich um eine *verborgene Semantik* in dem *CPL*-Meta-Modell. Sie äußerte sich dadurch, dass die *IsA*-Beziehung nicht als ein eigenes Element in dem Meta-Modell spezifiziert wurde, sondern als Perspektiven (Perspective) eines Zusammenhangstyps (ConnectionType), die mit *speziellen Schlüsselworten* wie generalization und specialization markiert wurden[9]. Ein Problem dieser Lösung ist die mangelnde Transparenz. Jemand, der die Schlüsselworte nicht kennt, wird dieses Modellierungselement nicht als solches wahrnehmen, da es sich von einem typischen Zusammenhangstyp bis auf die Schlüsselworte nicht unterscheidet. Ein so definiertes Modellierungselement bringt keine durch das Meta-Modell differenzierte, explizit definierte Semantik mit sich. Aus diesem Grund wurde dieser spezielle Zusammenhangstyp in das neue *CPL*-Meta-Modell aufgenommen.

Die Generalisierung wurde mittels der Meta-Klassen GeneralConnectionType und IsA (siehe Abbildung 4.3) spezifiziert. Die abstrakte Meta-Klasse GeneralConnectionType dient als gemeinsame Oberklasse der Meta-

[9] für weiterführende Informationen siehe auch [KMZ04] und [Ruß07]

Klassen ConnectionType und IsA. Sie fasst die gemeinsamen Elemente dieser Beziehungen zusammen. Da ihre beiden Unterklassen zu den Elementen 1. Klasse zählen, spezialisiert GeneralConnectionType die Meta-Klasse ModelingElement. Das Feld classification, das vergleichbar zu *Stereotypen* in *UML* verwendet werden kann und jedem *Modellierungselement 1. Klasse* zur Verfügung steht, wird für die *IsA*-Beziehung neu definiert und auf einen konstanten Wert GenSpec gesetzt. Eine zusätzliche Klassifizierung dieser Beziehung ist auf Grund des speziellen Einsatzzwecks nicht erforderlich.

Behandlung von Synonymen

Die Sprache *CPL* bietet für ThingTypes eine spezielle Beziehung synonym, um im Rahmen eines Projekts synonym verwendete Begriffe auszuzeichnen. In einem natürlichsprachlichen Text werden oft aus stilistischen Gründen für ein Konzept der Domäne verwandte Begriffe verwendet. Diese Besonderheit der natürlichsprachlichen Texte wird in *CPL* unterstützt, was eine einfache Zuordnung synonym verwendeter Begriffe zueinander und ihre schnelle Identifikation im Anforderungsdokument ermöglicht.

Die im ursprünglichen Meta-Modell vorgefundene isSynonymeTo-Beziehung hat einen entscheidenden Nachteil. Alle dadurch verbundenen Synonyme sind gleichwertige Dingtypen. Aus diesem Grund können Synonyme unabhängig voneinander an verschiedenen Beziehungen beteiligt sein, die das *CPL*-Meta-Modell für einen ThingType erlaubt. Diese Freiheit kann einerseits, falls sie exzessiv ausgenutzt wird, zu einem überladenen Modell führen[10] und andererseits kann dadurch suggeriert werden, dass es sich (trotz Synonym-Beziehung) um verschiedene Konzepte handelt.

Diese Flexibilität bringt keine Vorteile mit sich und steigert nicht die Ausdruckskraft der Sprache. Die isSynonymeTo-Beziehung kann als eine Hülle betrachtet werden, mit der mehrere ThingTypes zusammengefast werden und nach außen als ein ThingType agieren. Deswegen macht es keinen Unterschied, welches der Synonyme eine Beziehung hat. Sie gilt automatisch für alle beteiligten Dingtypen, den sonst sind die Konzepte nicht synonym. Aus diesem Grund bietet das neue Meta-Modell eine Möglichkeit, ein ThingType als Repräsentanten (das Feld mainSynonyme ist in diesem Fall auf true gesetzt) auszuzeichnen, der alle Beziehungen zu dieser Synonyme-Gruppe auf sich bindet.

Um sicherzustellen, dass die Synonyme unter Berücksichtigung der

[10] Schlimmstenfalls kann ein solches Modell gleiche Beziehungen zu jedem Synonym mehrfach und somit viel Redundanz enthalten.

Trennung (in *Repräsentant* und *zugehörige Synonyme* verwendet werden, enthält das neue *CPL*-Meta-Modell eine mit dem ThingType verbundene, in *OCL* verfasste Einschränkung (siehe Listing 4.1). Zunächst wird dort in den Zeilen 6 und 7 festgelegt, dass innerhalb einer Synonym-Gruppe[11] nur ein Repräsentant (mainSynonyme) definiert werden darf. Zusätzlich unterbindet sie die Möglichkeit (ab Zeile 9), dass ein Nicht-Repräsentant eine Beziehung außer isSynonymeTo eingehen kann.

```
1  context ThingType inv Synonym
2
3  def:  reachable() : Collection(ThingType) =
4       self->closure(synonyme->select(synonyme <> null))
5
6  self.synonyme.reachable()->
7           select(s | s.mainSynonyme = true) -> size() <=1
8
9  self.synonyme -> forall (s : ThingType |
10          s.mainSynonyme = false implies s.superType.isEmpty() and
11          s.subType.isEmpty() and s.executingActor.isEmpty() and
12          s.involvedThingType.isEmpty() and s.parameter.isEmpty() and
13          s.perspectivedeterminer.isEmpty() and
14          s.connectionTypeDeterminer.isEmpty() and
15          s.conditionInvolvedThingType.isEmpty() and
16          s.operationTypeDeterminer.isEmpty() and
17          s.callingActor.isEmpty())
```

Listing 4.1: OCL-Einschränkungen für Synonyme

Einbau der Kommentare

Eine zusätzliche Anpassung des Meta-Modells der Sprache *CPL* wurde durchgeführt, um Modellelemente kommentieren zu können. Beinahe jede künstliche Beschreibungssprache sieht diese Möglichkeit vor. Ein Grund dafür ist, dass es zu den Informationen über einen Gegenstandsbereich, die in einem Modell abgebildet werden, in der Regel ergänzende Informationen gibt, die keine Elemente des Gegenstandsbereichs sind, sondern seine Elemente weiter charakterisieren. Sie stellen somit Zusatz-Informationen über diese Elemente dar. Dadurch stehen Kommentare inhaltlich auf einer ande-

[11] Die entsprechende Hülle wird mittels der Operation reachable() ermittelt, die alle über die Synonyme-Beziehung erreichbare Dingtypen in eine Menge aufnimmt.

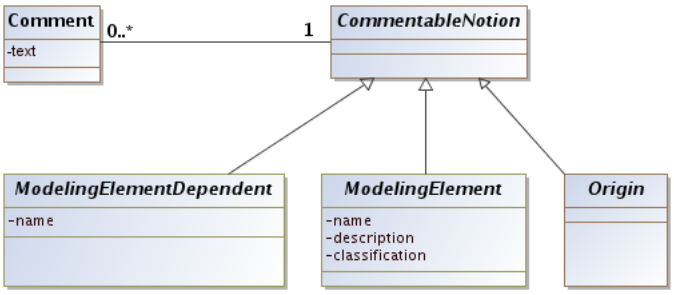


ABBILDUNG 4.4: *CPL*-Meta-Modell ergänzt um Kommentare

ren Ebene als die Modellelemente selbst und sollen von diesen abgegrenzt werden.

Primär ermöglicht es die Sprache *CPL*, alle ihre Elemente 1. Klasse und ausgewählte Elemente 2. Klasse mit Hilfe des Meta-Attributs description zu beschreiben. Um zu vermeiden, dass dieses Feld sowohl für die Beschreibung eines Elements als auch für das Festhalten der Meta-Informationen über dieses Element verwendet wird, wurde das *CPL*-Meta-Modell um die Meta-Klasse Comment ergänzt (siehe Abbildung 4.4).

CPL-Kommentare sind ein einfacher Text und werden eindeutig einem Element der Sprache zugeordnet. Kommentiert werden können alle Elemente 1. und 2. Klasse und Quellenangaben (Origin). Es ist erlaubt, mehrere Kommentare zu einem Modellierungelement zu formulieren.

4.1.2 Revidiertes *CPL*-Meta-Modell

Ergänzend zu der Beschreibung der wichtigsten Änderungen wird in diesem Abschnitt das gesamte *CPL*-Meta-Modell vorgestellt. Es ist in drei Bereiche unterteilt: allgemeine Meta-Modell-Elemente, Elemente für die statische und dynamische Modellierung. Die Verknüpfung dieser drei Bereiche wird über die zentralen Elemente des Meta-Modells ModelingElement und ModelingElementDependent hergestellt. Das ursprüngliche *CPL*-Meta-Modell steht zum Vergleich im Anhang A.

Die allgemeinen Meta-Modell-Elemente ermöglichen es, den *CPL*-Modellierungselementen zusätzliche Informationen zuzuordnen. Es sind die neu eingeführten Kommentare, Einschränkungen (Constraint) und die ausgebauten Quellenangaben (Origin). Die letzteren berücksichtigen nicht nur Quellen, die mit der Anforderungsanalyse verbunden sind (RequirementSource),

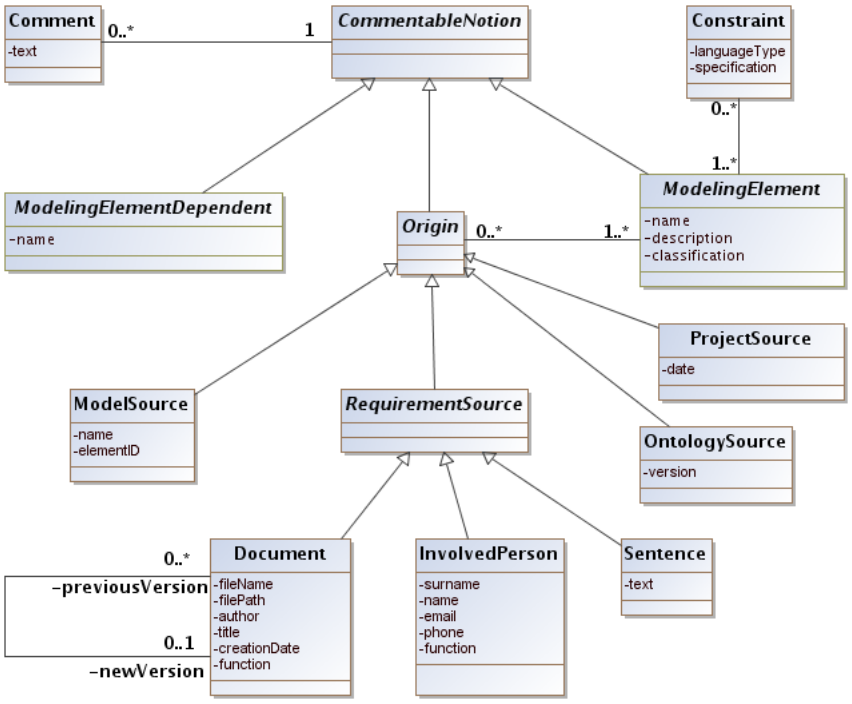


ABBILDUNG 4.5: Ausschnitt des revidierten *CPL*-Meta-Modells mit allgemeinen Elementen

sondern auch die im Rahmen des *OBSE*-Prozesses neu entstandenen Quellen: ModelSource, ProjectSource und OntologySource. Dieser Abschnitt des Meta-Modells wird in der Abbildung 4.5 dargestellt.

Im Mittelpunkt der statischen Modellierung steht das Meta-Modell-Element ThingType (siehe Abbildung 4.6). Für dieses Element können in *CPL* zusätzlich Beispiele (Example) und die typischen Größenordnungen[12] (QuantityDescription) angegeben werden. Vom GeneralConnectionType abgeleitete Elemente des Meta-Modells und die Meta-Klassen Perspective und AdditionalCardinality dienen in *CPL* zur Herstellung von Beziehungen zwischen den ThingType-Elementen. Die *IsA*-Beziehung wurde ausführlich in dem Kapitel 4.1.1 vorgestellt.

Für die mit Assoziationen in *UML* vergleichbaren Beziehungen steht in *CPL* die Meta-Klasse ConnectionType. Die Beziehung wird dadurch gebildet, dass einem ConnectionType mehrere Perspective-Elemente zugeordnet werden, die über die Assoziation involvedThingType auf den beteiligten

[12]Diese zeigt, wie viele Objekte von dieses Typs in einer Domäne in der Regel existieren.

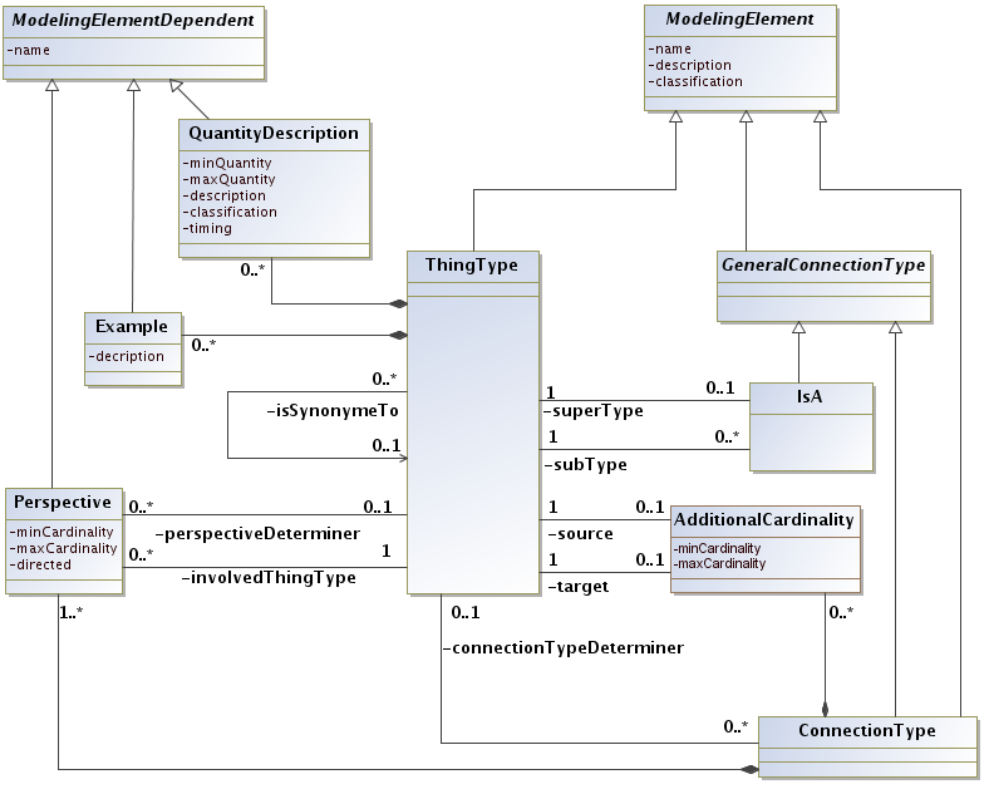


ABBILDUNG 4.6: Ausschnitt des revidierten *CPL*-Meta-Modells für die statische Modellierung

ThingType zeigen. Für eine zweistellige Beziehung werden einem ConnectionType zwei Perspektiven zugeordnet, für eine dreistellige – drei usw. Rekursive Beziehungen werden dadurch hergestellt, dass zwei Perspektiven eines Zusammenhangstypen auf den gleichen Dingtypen zeigen. Eine weitere Meta-Klasse, die ConnectionType mit ThingType verbindet, ist AdditionalCardinality. Sie wird in *CPL* zur Charakterisierung der mehrstelligen Beziehungen verwendet, indem die Kardinalitäten der beteiligten Dingtypen paarweise angegeben werden. Diese ergänzende Information sollte bei der Festlegung der eigentlichen Kardinalitäten minCardinality und maxCardinality in der Meta-Klasse *Perspective* hilfreich sein.

Die Vergegenständlichung wird in *CPL* durch Assoziationen mit dem Teilwort Determiner in ihrer Bezeichnung realisiert. So steht perspectiveDeterminer für eine Vergegenständlichung einer Perspekive. Verglichen mit *UML* handelt es sich in diesem Fall um eine *Rolle*, die diese Perspektive in der Beziehung übernimmt. Außerdem können noch Meta-Modell-Elemente

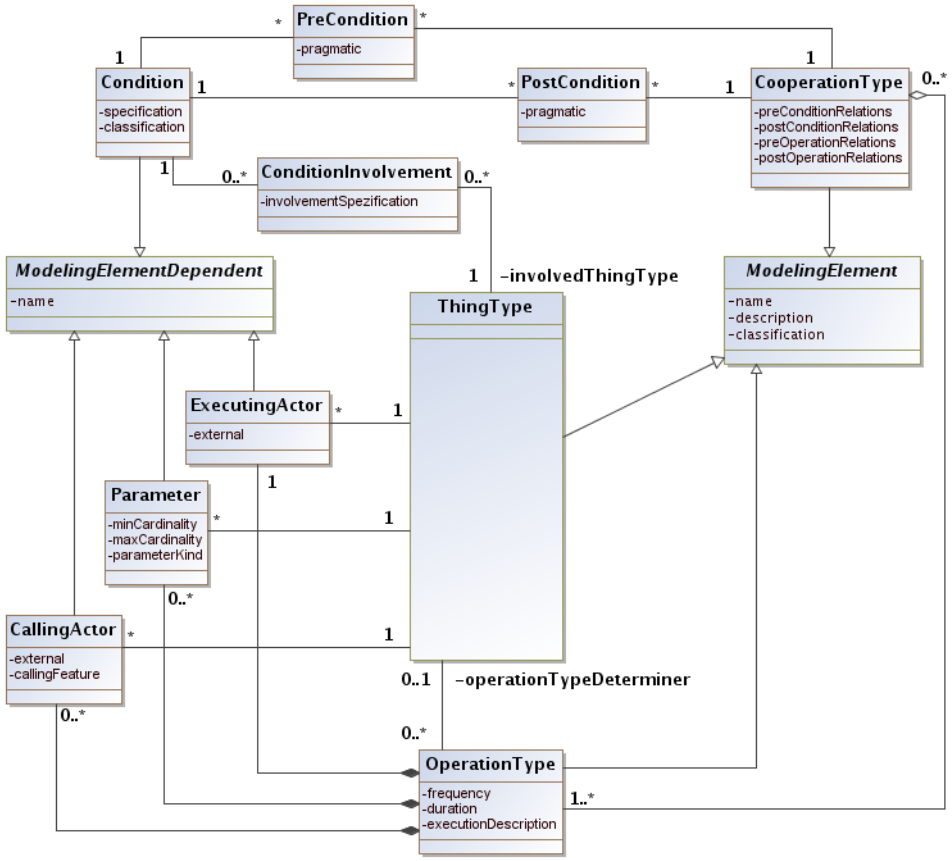


ABBILDUNG 4.7: Ausschnitt des revidierten *CPL*-Meta-Modells für die dynamische Modellierung

ConnectionType und operationType vergegenständlicht werden. Der erste Fall würde – in der Analogie zu *UML* – einer Assoziationsklasse entsprechen. Der zweite Fall (operationTypeDeterminer in der Abbildung 4.7) ist *CPL*-spezifisch und ermöglicht es, einen Vorgang mit einem ThingType stellvertretend in die statischen Modelle zu integrieren. Typischerweise wird ein operationTypeDeterminer verwendet, wenn in den Anforderungen ein Verb zusammen mit dem für diese Tätigkeit stehenden Substantiv wie *buchen* und *Buchung*, *überweisen* und *Überweisung* usw. verwendet wird und der Zusammenhang zwischen diesen im Modell festgehalten werden soll.

Die Grundlage der dynamischen Modellierung in *CPL* bilden die Meta-Klassen OperationType und CooperationType (siehe Abbildung 4.7). Ihnen werden mehrere Elemente 2. Klasse zugeordnet. Beim OperationType sind es Angaben zum Aufrufer der Operation (CallingActor), dem Ausführer der

Operation ExecutingActor und den Eingabe- und Ausgabe-Parametern Parameter.

Ein CooperationType setzt sich aus einem oder mehreren[13] OperationTypes zusammen. Für einen Kooperationstyp lassen sich Bedingungen (Condition) definieren, die in der Rolle einer Vor- (PreCondition) oder Nachbedingung (PostCondition) auftreten können. Auf diese Weise lassen sich mehrere Kooperationstypen zu einem Ablauf verknüpfen[14]. Da eine Spezifikation der Bedingungen oft im Zusammenhang zu den Dingtypen einer Domäne steht, stellt *CPL* en weiteres Meta-Modell-Element ConditionInvolvement zur Verfügung, um das Auftreten eines Dingtypen in einer Bedingung festzuhalten.

4.1.3 *CPL* als Ontologie-Sprache

Im Kapitel 2.1.3 wurde vorgestellt, wie *CPL* als ein Bestandteil der *Conceptual Predesign-Methode* im Rahmen des *NIBA*-Prozesses verwendet wird. Die Aufgabe, die diese Sprache dort übernimmt, zeigt, dass *CPL* ursprünglich mit dem Schwerpunkt entwickelt wurde, den Fachverantwortlichen und Software-Entwicklern eine möglichst barrierefreie Kommunikation über den zu analysierenden Untersuchungsbereich zu ermöglichen und damit den Wissensaustausch bei der Anforderungsanalyse zu fördern. Das ist eine der typischen Aufgaben für Sprachen zur konzeptuellen Modellierung. Eine scharfe Abgrenzung zwischen diesen Sprachen und solchen, die aktuell zur Beschreibung von Ontologien verwendet werden, existiert nicht. Ob eine Sprache als Modellierungssprache oder eine Sprache zur Beschreibung von Ontologien betrachtet werden kann, hängt überwiegend von der historischen Entwicklung dieser Sprache und ihrem anvisierten Einsatzzweck ab. Dabei ist es nicht ausgeschlossen, dass sich zum Beispiel eine auf Entwurf von Systemen spezialisierte Sprache auch für einen anderen Einsatzzweck verwenden lässt. So wird auch *UML* als eine geeignete Ontologie-Sprache angesehen ([Gui05] und [GDD06]).

Typischerweise bieten konzeptuelle Modellierungssprachen weitgehende Unterstützung, um Details über die zu modellierenden Elemente zu beschreiben. Diese Neigung zur Detailliertheit ist ein spürbarer Unterschied im Sprachumfang zwischen den Modellierungs- und Ontologie-Sprachen, der wiederum auf dem anvisierten Einsatzzweck beruht. Sprachen, die zur Beschreibung von Ontologien entwickelt wurden, sind für eine breite An-

[13] In diesem Fall werden sie parallel ausgeführt.
[14] Siehe auch Kapitel 2.1.3

wendbarkeit und übergreifende Wiederverwendbarkeit konzipiert. Modellierungssprachen wie *UML* sind dagegen entwickelt worden, um ein Software-System so detailliert zu entwerfen, wie das die spätere Implementierung erfordert. Um *UML* als Ontologie-Sprache zu verwenden, werden in der Regel spezielle *UML*-Profile[15] definiert, die die Möglichkeiten der Sprache in dieser Hinsicht einschränken bzw. an Ontologie-Sprachen wie *OWL* angleichen [BVEL04].

Um mit *CPL* Ontologien beschreiben zu können, ist es einerseits erforderlich, grundlegende Bestandteile einer Ontologie wie Konzepte und Beziehungen (siehe Seite 39) mit der Sprache ausdrücken zu können. Andererseits sollte die Detailliertheit von *CPL* den gängigen Ontologie-Sprachen bzw. *UML*-Ontologie-Profilen entsprechen, da in *CPL* verfasste Ontologien im *OBSE*-Prozess eine Schlüsselrolle beim Wissenaustausch innerhalb eines Untersuchungsbereichs spielen sollen.

Eine Analyse des *CPL*-Meta-Modells[16] und ein gleichzeitiger Abgleich mit der Klassifikation der Wissensrepräsentationsformen in der Abbildung 2.13 auf der Seite 46 gibt eine Orientierung bei der Einordnung von *CPL* in die Landschaft der Ontologie-Sprachen. Die Klasse ModelingElement ermöglicht es, mit Hilfe der Attribute name und description eine Domäne Glossar-artig zu beschreiben. Dies ist allerdings eine abstrakte Meta-Klasse, die nicht direkt verwendet wird, sondern nur ihre Unterklassen wie ThingType. Somit ist es im Vergleich zu reinen Glossaren möglich, die Domäne mit Konzepten und Relationen zu beschreiben. Damit rückt *CPL* bei der Einordnung in der Abbildung 2.13 weiter nach rechts in den Bereich der strukturierten Glossare.

Eine weitere Analyse der ebenfalls abstrakten Klasse GeneralConnectionType zeigt, dass die Beziehungen in *CPL* noch weiter formalisiert sind. Es wird zwischen Generalisierung/Spezialisierung und allgemeinen Beziehungen unterschieden. *CPL* weist die zur Bildung von Hierarchien erforderliche Generalisie-rung/Spezialisierung-Beziehung durch die Meta-Klasse IsA und die zugehörigen Meta-Assoziationen superType und subType auf und rückt damit in den Bereich der formalen Taxonomien.

[15] Mit Profilen steht in *UML* ein Mechanismus für die Änderung des Sprachumfang (gemäß dem *UML*-Meta-Modell) zur Verfügung. Ein *UML*-Profil ist gleichzeitig ein *UML*-Modell und eine Erweiterung des Meta-Modells. Es sind allerdings nur eine genauere Spezifikation oder Einschränkung der vorhandenen Elemente des *UML*-Meta-Modells erlaubt. Neue Elemente können auf diese Art nicht eingeführt werden.

[16] Siehe Abbildung 4.6 im Kapitel 4.1.2. Dieser Meta-Modell-Abschnitt spielt bei dem Vergleich eine wichtige Rolle, da die meisten Ontologie-Sprachen vorwiegend die statische Modellierung unterstützen.

Frames haben einerseits einige Gemeinsamkeiten mit der *Conceptual Predesign Language*, auf der anderen Seite gibt es entscheidende Unterschiede, die *CPL* von dieser Wissensrepräsentationsform deutlich abgrenzen. Genauso wie *Frame*-basierte Sprachen kann *CPL* der Gruppe der Struktur-basierten Sprachen zugeordnet werden. *Frames* können im Allgemeinen mit Hilfe von ThingTypes nachgebildet werden [KMZ04]. Auch die Unterscheidung zwischen *class* und *instance Frames* kann mit *CPL* ausgedrückt werden, indem die letzteren nicht auf ThingTypes, sondern auf die Meta-Klasse Example des *CPL*-Meta-Modells abgebildet werden. Allerdings dienen Examples in *CPL* hauptsächlich zur Veranschaulichung des zugehörigen ThingTypes und sind deswegen nicht so flexibel[17] zur Modellierung von Instanzen einsetzbar, wie das für das allgemeinere Modellierungselement *instance Frame* der *Frame*-basierten Sprachen der Fall ist.

Interessant ist die Situation bezüglich *Slots*. Da sie wiederum *Frames* sind, können sie analog mit ThingTypes nachgebildet werden. Eine wichtige Ausnahme bilden die *own Slots*, die nur Eigenschaften von *einem Frame* beschreiben. Für diese Einschränkung gibt es in dem Meta-Modell von *CPL* kein direktes Konstrukt. Indirekt kann sie durch die Zuordnung eines Constraints zum entsprechenden ThingType realisiert werden.

Für die beiden Beziehungstypen von *CPL* (IsA und ConnectionType) bieten die *Frame*-basierten Sprachen ebenfalls Äquivalente an. Die Charakterisierung der in Beziehung stehenden Elemente, die im Falle von *Frames* und *Slots* mit Hilfe von *Facets* stattfindet, ist in *CPL* durch das Konstrukt Perspective realisiert. Die Perspektiven sind in *CPL* analog zu den *Assoziationsenden* in *UML* realisiert und verbinden einen ConnectionType mit einem ThingType. Das hat zur Folge, dass eine typische zwei-stellige Beziehung zwischen zwei *ThingTypes* ebenfalls zwei Perspektiven besitzt. Dadurch können die Beziehungen in *CPL* – analog zu dem *Facet*-Element der *Frame*-basierten Sprachen – genauer charakterisiert werden.

CPL entspricht bezüglich des Formalisierungsgrads und der Ausdruckskraft weitgehend den *Frame*-basierten Ontologie-Sprachen [KMZ04] und kann zur Beschreibung einer Domänen-Ontologie verwendet werden. Zusätzlich zu der bereits diskutierten Überlegung, ob *CPL* als eine Ontologie-Sprache fungieren kann, ist natürlich auch die Frage berechtigt, ob eine neue Ontologie-Sprache erforderlich ist.

Abgesehen von der bereits erwähnten, eher praktisch motivierten Überlegung, *CPL* sowohl für die konzeptuellen Modelle eines Projekts, als auch

[17] Es ist keine weitere Beziehung bzw. Charakterisierung möglich.

für die zugehörige Domänen-Ontologie zu benutzen, sprechen noch folgende Gründe dafür:

- Die überwiegende Anzahl der Ontologie-Sprachen hat ihren Ursprung in der *KI*-Forschung und konzentriert sich sehr stark auf die maschinelle Verarbeitung und die Kommunikation mit und zwischen Agenten. Dies spielt in der Software-Technik in der Regel keine beziehungsweise nur eine untergeordnete Rolle. Auch die Erfahrung aus dem Bereich des *Semantischen Webs* zeigt, dass die Sprache *OWL* für nicht fachkundige Nutzer zu komplex ist [TPCB06]. Das führt zu einer auf einige wenige Elemente – wie zum Beispiel Bildung von Taxonomien – reduzierte Nutzung der Sprache. Als Ausweg wird auch im *Semantischen Web* an neuen, benutzerfreundlichen Sprachen bzw. Tools gearbeitet [TPCB06].
- Ein Kriterium, das bei der Evaluierung der Qualität eines Ontologie-Entwurfs eine wichtige Rolle spielt, ist laut Gruber [Gru93b] die minimale ontologische Verpflichtung. Um die Minimalität von Anfang an zu fördern und somit die Qualität einer Ontologie konstruktiv zu erhöhen, ist es erforderlich, dass die Notation einer Sprache nur die für die Beschreibung einer Ontologie benötigten Mittel zur Verfügung stellt. Das ist sicherlich nicht der Schwerpunkt von anderen in der Software-Technik verbreiteten Sprachen wie *UML*.

Zusammenfassend betrachtet ist *CPL* eine konzeptuelle Sprache, die wichtige Elemente zur Modellierung einer Domäne zur Verfügung stellt. Sie benutzt zu *UML* ähnliche Sprachelemente, was eine Transformation von *CPL* zu *UML* und zurück semi-automatisch mit einem kompakten Regelsatz ermöglicht[18]. Im Unterschied zu *UML* benutzt *CPL* einen reduzierten Sprachumfang, der den Struktur-basierten Ontologie-Sprachen entspricht. So wird verhindert, dass bestimmte Entwurfs-Entscheidungen (Klasse-/Attribut-Festlegung oder unterschiedliche Agregationstypen) bereits in den frühen Phasen getroffen werden – was einerseits hilft, die ontologische Verpflichtung der zu modellierenden Ontologien klein zu halten und andererseits sie für nicht-fachkundige Anwender zugänglicher macht. Diese Eigenschaften von *CPL* erlauben es, sie sowohl für die konzeptuelle Modellierung in einem Software-Projekt und für die Kommunikation mit Kunden bzw. Fachverantwortlichen als auch zur Beschreibung einer Domänen-Ontologie einzusetzen.

Die Vorstellung des revidierten Meta-Modells und die Betrachtung von

[18] Siehe Kapitel 4.2.2 und [Rußß07].

CPL als eine Sprache zur Beschreibung von Domänen-Ontologien schließen die vorbereitenden Tätigkeiten ab, die erforderlich waren, um die Elemente der *OBSE*-Infrastruktur definieren zu können. In den folgenden Kapiteln werden sie detailliert beschrieben. Zunächst werden die Infrastruktur-Elemente auf der Projekt-Seite in der Reihenfolge ihrer Verwendung in einem Projekt behandelt. Anschließend werden für den projektübergreifenden Wissensaustausch stehende Elemente auf der Seite der Domänen-Ontologie eingeführt.

4.2 Infrastruktur-Elemente auf der Projekt-Seite

4.2.1 Konzeptuelles Modell

Das Drehkreuz der *Ontologie-basierten Software-Entwicklung* ist das *konzeptuelle Modell*. Es verknüpft Anforderungen, System-Modell und die Domänen-Ontologie miteinander. Bedingt durch diese Aufgabe hat ein konzeptuelles Modell in *OBSE* einen oder mehrere Berührungspunkte zu den folgenden Sprachen: natürliche Sprache, *UML* und *CPL*. In den folgenden Abschnitten wird vorgestellt, wie ein konzeptuelles Modell im Rahmen eines Projekts verwendet wird. Dafür wird mit Hilfe eines Beispiels demonstriert, wie dieses Modell aus gegebenen Anforderungen aufgebaut und in *CPL* formuliert wird. Als weitere mit dem konzeptuellen Modell verknüpfte Bestandteile der *OBSE*-Infrastruktur werden die Transformationen nach *UML* und zurück vorgestellt.

Als Ausgangsbasis für das konzeptuelle Modell in *OBSE* dienen *Anforderungen* eines Software-Projekts. Dieser Begriff wird in der Software-Technik vielschichtig angewendet. In den meisten Fällen steht er übergreifend für alle Aussagen eines Kunden oder Anwenders bezüglich der gewünschten Funktionalität und Beschaffenheit des zukünftigen Software-Systems. So lassen sich Anforderungen eines Systems nach verschiedenen Kriterien klassifizieren. Zunächst können sie anhand ihres Abstraktions- und Detailliertheitsgrades in *Benutzer-* und *Systemanforderungen* unterteilt werden [Som07]. Eine andere weit verbreitete Klassifizierung nach der Zielsetzung unterscheidet zwischen den *funktionalen* und *nicht-funktionalen* Anforderungen. Diese beiden Klassifizierungen sind orthogonal zueinander. So können sowohl Benutzer- als auch Systemanforderungen in funktionale und nicht-funktionale Anforderungen unterteilt werden.

Im *OBSE*-Prozess werden zunächst in der natürlichen Sprache verfasste Anforderungen als Benutzer-Anforderungen bezeichnet. Das konzeptuelle Modell entsteht aus diesen durch eine weitere Spezifizierung und ihre Überführung nach *CPL*. Während die Benutzer-Anforderungen in der Regel eine Mischung aus den funktionalen und nicht-funktionalen Anforderungen darstellen, lassen sich mit *CPL* ausschließlich funktionale Anforderungen modellieren (vergleichbar zu der Anforderungsanalyse bei *UML*). Somit steht das Konzeptuelle Modell in *OBSE* für strukturierte, in *CPL* verfasste funktionale Systemanforderungen eines Software-Systems[19].

Die Vorgehensweisen und damit verbundene, eingesetzte Techniken zur Anforderungsanalyse sind ebenfalls vielfältig. *CPM* konzentriert sich auf den Bereich der funktionalen Anforderungen und bietet dafür eine neue, alternative Vorgehensweise zusammen mit den entsprechenden Tools und legt ihren Schwerpunkt auf den Übergang von den Benutzer- zu den Systemanforderungen. Somit wird davon ausgegangen, dass die Benutzeranforderungen mittels Interviews, Viewpoints, Szenarien[20] usw. bereits gesammelt sind und in der (strukturierten) natürlichen Sprache vorliegen.

Beispiel einer Anforderungsbeschreibung

Folgender Textausschnitt ist an die Anforderungsbeschreibung angelehnt, die nach mehreren Interviews mit den zukünftigen Nutzern des *SpeSemOnline*-Systems[21] erstellt wurde. Das web-basierte Software-System soll eine elektronische Anmeldung zu den Sprechstunden eines Professors ermöglichen. Es bestand aus einer öffentlichen Webseite für alle Sprechstunden-Interessenten, die das Anmelden, Benachrichtigen und Verwalten von eigenen Sprechstunden ermöglichte. Außerdem stand ein passwortgeschützter Bereich für die Anbieter der Sprechstunden zur Verfügung, um Zeiträume der Sprechstunden zu definieren und Anmelde-, Nachzügler- und Sperr-Listen zu verwalten.

„Ein Professor bietet mehrere Sprechstunden an. Eine Sprech-

[19] Siehe auch Abbildung 3.2 in der *OBSE*-Vorstellung auf der Seite 75.

[20] Viewpoints und Szenarien sind stukturierte Techniken der Anforderungsanalyse. Viewpoints ermitteln die Anforderungen an ein Software-System ausgehend von den beteiligten Personen (Aktoren) und konzentrieren sich auf ihre Interaktion. Szenarien leiten die Anforderungen aus den Beschreibungen der Situationen bei der zukünftigen Systemnutzung ab. Für weiterführende Informationen siehe [Som07].

[21] *SpeSemOnline* ist ein Akronym für den Namen: *Sprechstunden- und Seminarverwaltung Online*. Der Ausschnitt aus den Anforderungen basiert auf der zugehörigen Projektdokumentation [HB08].

stunde wird in Slots eingeteilt. Von dem zuständigen Professor werden die Sprechstundenanfang-, Sprechstundenendzeit und die Slotlänge (z.B. 15 Minuten) festgelegt. Studenten können sich in genau einen freien Slot einer Sprechstunde eintragen, der für sie durch einen Slot-Eintrag reserviert wird. Möchte sich nun ein Student für eine bereits komplett belegte Sprechstunde anmelden, so kann er sich in die Warteliste eintragen, was ebenfalls mit einem entsprechenden Eintrag gekennzeichnet wird. Ein Student wird durch seine E-Mail-Adresse identifiziert. Die Warteliste wird von dem Professor verwaltet. Um sich im System anmelden zu können, sollte er einen Account besitzen.“

Diese Beschreibung ist entsprechend der Vorgaben an die Benutzeranforderungen[22] sehr allgemein verfasst und enthält noch keine Umsetzungsspezifischen Details. Auch wenn sie von einem Software-Entwickler dieses Projekts erstellt wurde, ist sie ebenfalls einem zukünftigen Anwender zugänglich und lässt noch verschiedene technische Umsetzungen der Anforderungen zu.

Von Anforderungen zu einem konzeptuellen Modell

Die meisten Techniken, um diese Beschreibung in die Systemanforderungen zu überführen, basieren auf der *grammatikalischen Analyse*. Diese wird zum Beispiel in der *Objekt-orientierten Software-Entwicklung* verwendet, um Objekte bzw. ihre Klassen zu identifizieren und in Objektlisten abzulegen. Die grammatikalische Analyse kann analog im Rahmen von *OBSE* verwendet werden. Dabei kommen Substantive als Kandidaten für Dingtypen und Verben für Zusammenhangs- bzw. Operationstypen[23] in Frage. So bietet sich an, die Substantive Professor und Sprechstunde als Dingtypen und das Verb anbieten als Zusammenhangstyp zu modellieren. Diese Analyse-Technik wird in *OBSE* mittels der bereits im Kapitel 2.1.3 beschriebenen *NIBA*-Werkzeuge weitgehend automatisiert [FKM+02]. Eine standardmäßige Auswertung des ersten Satzes der obigen Anforderungsbeschreibung durch den *NIBA-Interpreter* ist in der Tabelle 4.1 dargestellt.

[22] Siehe [Som07]

[23] Grammatikalische Analyse ist vor allem für die statische Modellierung gut geeignet. Dynamik eines Systems aus einem natürlichsprachlichen Text abzuleiten, erfordert in der Regel zusätzlichen Interpretationsaufwand. So lassen sich aus dem Verb bietet an in der obigen Anforderungsbeschreibung ergänzend zu der statischen Beziehung mehrere Operationen wie zum Beispiel anlegen, bearbeiten ableiten, die der Benutzer eines Systems implizit erwartet.

Klassifikation	Wort
Quantor	Ein
Dingtyp	Professor
Spezifikator	kann
Quantor	mehrere
Dingtyp	Sprechstunden
Zusammenhangstyp	anbieten

TABELLE 4.1: Automatischer Vorschlag des *NIBA-Interpreters*

Der erste Schritt beim Übergang von den Benutzer- zu den Systemanforderungen ist die Klassifikation der Elemente und Herstellung der Beziehungen zwischen diesen. Erwartungsgemäß wurden Substantive auf Dingtypen abgebildet. Bei den Verben unterscheidet der Interpreter zwischen den *Modal-*, *Hilfs-* und *Vollverben*. Aus diesem Grund werden die beiden vorkommenden Verben unterschiedlich klassifiziert. Beim Vorhandensein von zwei Dingtypen in einem Satz wird ein Vollverb in der Regel als Zusammenhangstyp identifiziert, so wie in dem Beispiel oben das Verb anbieten.

Der zweite Schritt besteht in der weiteren Spezifikation der erkannten Beziehung mittels zugehörigen Kardinalitäten, deren Beschreibung in den Benutzeranforderungen noch einen Spielraum enthält und möglicherweise von Projekt zu Projekt unterschiedlich ausgelegt werden kann. Anhaltspunkte für die Kardinalität eines in der Beziehung beteiligten Dingtypen bieten *Qantoren* und *Spezifikatoren*. Sie werden kombiniert ausgewertet, um Informationen über mögliche Kardinalitäten zu gewinnen.

In dem vorliegenden Beispiel (im ersten Satz) kann der Wort-Verbund ein Professor (wie in diesem Fall vermutlich gemeint) als *genau einer* und in einer anderen Konstellation als *einer von mehreren* interpretiert werden. An dieser Stelle macht der Interpreter keinen Eintrag in das *CPL*-Modell. Lücken, die der Interpreter hinterlässt, sind gute Anhaltspunkte für weitere Abstimmung mit den Fachverantwortlichen und sollen beim Übergang zu den Systemanforderungen vervollständigt werden.

Die Kombination der Worte kann und mehrere wird dagegen als Kardinalität 0..N interpretiert [FKM+02] und dem Dingtyp Sprechstunde zugeordnet. Technisch gesehen bedeutet das, dass eine Instanz der Meta-Klasse Perspective erzeugt wird und ihre Attribute minCardinality und maxCardinality auf 0 bzw. N gesetzt werden. Diese Instanz wird mit den Instanzen für Sprechstunde und bietet an der Meta-Klassen ThingType (Meta-Assoziation

Name	beteiligter Dingtyp	Perspektive	Kardinalität	Rolle	Quelle
bietet an	Professor	bietet an	1	Sprechstunden-Verantwortlicher	Anforderungs-beschreibung auf der Seite 100
	Sprechstunde	wird angeboten von	0..N		

TABELLE 4.2: Ausschnitt des konzeptuellen Modells für den ersten Satz der Anforderungsbeschreibung.

involvedThingType) und ConnectionType (Aggregation zur Perspective) verbunden. Wie diese Situation mittels *CPL* dargestellt wird, zeigt die Tabelle 4.2.

Die Abbildung 4.8 präsentiert ergänzend dazu eine mit dem Schwerpunkt auf Beziehungen angelegte, graphische Übersicht des konzeptuellen Modells, das aus dem oben präsentierten Ausschnitt der Benutzeranforderungen entstanden ist.

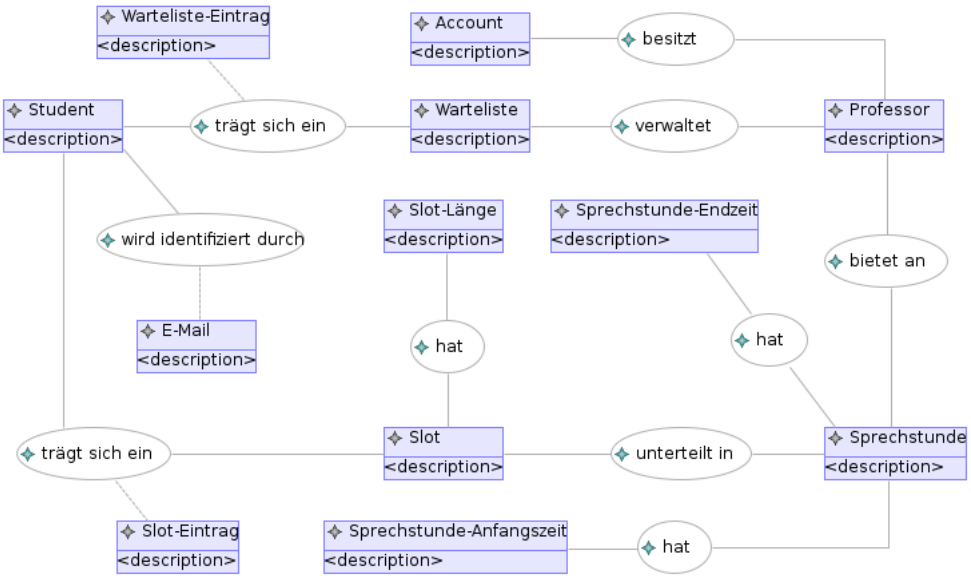


ABBILDUNG 4.8: Graphische, schematische Darstellung aus den Anforderungen extrahierter Ding- und Zusammenhangstypen

Diese Beispiele zeigen, dass ein so verfasstes konzeptuelles Modell im Vergleich zu den Benutzeranforderungen deutlich mehr Struktur aufweist und weitere Details enthält und somit nach Sommerville [Som07] als ein

berechtigtes Mittel zur Beschreibung der Systemanforderungen angesehen werden kann. Auch die Lücken in dem Modell lassen sich in dieser Form besser als in der natürlichsprachlichen Beschreibung der Anforderungen erkennen und in einer Diskussion mit dem Fachverantwortlichen klären und ergänzen. Die zugrunde liegende Struktur (ThingType und ConnectionType, Perspective) des konzeptuellen Modells bietet außerdem eine gute Ausgangsbasis für weitere Modelle im Projekt, die aus diesem Modell mittels Transformationen abgeleitet werden können.

4.2.2 Verwendeter Transformationsansatz

Um ein konzeptuelles Modell an weitere Entwicklungsschritte eines Software-Projekts anzubinden, stellt die *OBSE*-Infrastruktur zwei Transformationen[24] zur Verfügung: CPL→UML und UML→CPL.

Bei der Suche nach einem geeigneten Mechanismus für eine Umsetzung der CPL→UML- und UML→CPL-Transformationen wurde der Schwerpunkt auf die relationalen oder Graphen-basierten Ansätze gesetzt[25], da sie im Vergleich zu der direkten Manipulation und Struktur-Ansätzen auf bekannten formalen Methoden aufbauen und somit eine bessere Grundlage für die Transformationen bereitstellen. Einen sowohl theoretischen als auch in Form eines Werkzeugs weit entwickelten Transformationsansatz aus dem Bereich der Graphtransformationen bietet das *Tiger EMF Transformationsprojekt* [BEK+06]. Die Vorteile dieses Ansatzes liegen vor allem in seinen umfassend erforschten theoretischen Grundlagen. Als ein entscheidender Nachteil für die Integration in die *OBSE*-Infrastruktur hat sich erwiesen, dass die zugehörigen Werkzeuge vorerst nur *endogene*[26] Transformationen unterstützen. Die formalen Grundlagen für die Umsetzung von *bidirektionalen* und *exogenen* Transformationen werden aktuell erforscht [EEE+07]. Die entsprechenden Werkzeuge konnten allerdings zu dem Zeitpunkt der Beschreibung von CPL→UML- und UML→CPL-Transformationen noch nicht eingesetzt werden.

Im Bereich der relationalen Ansätze nimmt *MOF QVT* durch seine Standardisierung und damit verbundene Zukunftssicherheit einen besonde-

[24] Oft wird für den Begriff Transformation alternativ die dem englischen stammende Bezeichnung *Mapping* verwendet.

[25] Für weitere Informationen zu Transformationsansätzen siehe Kapitel 2.1.4.

[26] Die Eingabe- und die Ausgabe-Sprache sind in diesem Fall identisch. Im Gegensatz zu *endogenen* Transformationen erlauben *exogene* Transformationen, unterschiedliche Sprachen für Ein- und Ausgabe zu benutzen.

ren Stellenwert ein. Aus diesem Grund wurden im Bereich der relationalen Ansätze nur auf diesen Standard aufbauende Implementierungen betrachtet. Es stehen sowohl kommerzielle wie die im *CASE*-Tool *Borland Together* integrierte Unterstützung für Transformationen als auch offene Lösungen wie *Atlas Transformation Language (ATL)* zur Verfügung. Für die letztere spricht nicht nur ihre offene Implementierung, sondern auch ihre bessere Reife im Vergleich zu den konkurrierenden Lösungen [FH06]. Auch die Aufnahme von *ATL* in das *Eclipse Model to Model (M2M)*-Projekt ist eine zusätzliche Bestätigung, dass diese Implementierung zur Zeit eine ausgereifte und leistungsfähige Lösung darstellt. Zudem ermöglicht es diese Integration in *Eclipse*, den Transformationsmechanismus mit den weiteren Elementen wie *Ecore*-Meta-Modell und *UML*-Tolls zu verbinden und so zu einem mächtigen Werkzeug zu entwickeln.

4.2.3 Eingabe- und Ausgabe-Sprachen der Transformationen

Bei den beiden Transformationen handelt es sich um partielle Transformationen. Sie vermitteln zwischen zwei Sprachen, die sich, wie in der Abbildung 3.2 vorgestellt, horizontal auf unterschiedlichen Ebenen befinden. Durch diese Einordnung im *OBSE*-Prozess und insbesondere durch ihren Aufgabenbereich unterscheiden sich die Sprachen in ihrem Abstraktionsgrad und ihrer Detailliertheit. Diese Unterschiede führen unter anderem zu bestimmten Elementen in den Meta-Modellen dieser Sprachen, die keine Entsprechung in der jeweils anderen Sprache besitzen, weil sie wie zum Beispiel im Falle von *CPL* für Kommunikation mit den Fachverantwortlichen oder wie im Falle von *UML* für spätere System-Implementierung vorgesehen sind.

Ein Überblick über Untersprachen, die basierend auf *UML* und *CPL* definert und im Rahmen der *OBSE*-Infrastruktur benutzt werden, ist in Abbildung 4.9 dargestellt. Auf der linken Seite sind die *CPL*-Untersprachen *CPL System* und *CPL Ontology* abgebildet. Die erste Sprache wird benutzt, um die Tansformationen innerhalb eines Projekts (Analyse-Modell) zu definieren. *CPL Ontologie* ist die Grundlage für die Beschreibung von Export- und Import-Brücken. Wie die Proportionen der Kreise in der Abbildung 4.9 andeuten, ist die Überlapung zwischen *CPL System* und *CPL Ontology* groß. Sie unterscheiden sich nur in wenigen Details, die bei der Beschreibung der Sprachelemente erläutert werden. Außerdem decken sie den Sprachumfang von *CPL* zum überwigenden Teil ab. Hauptsächlich für

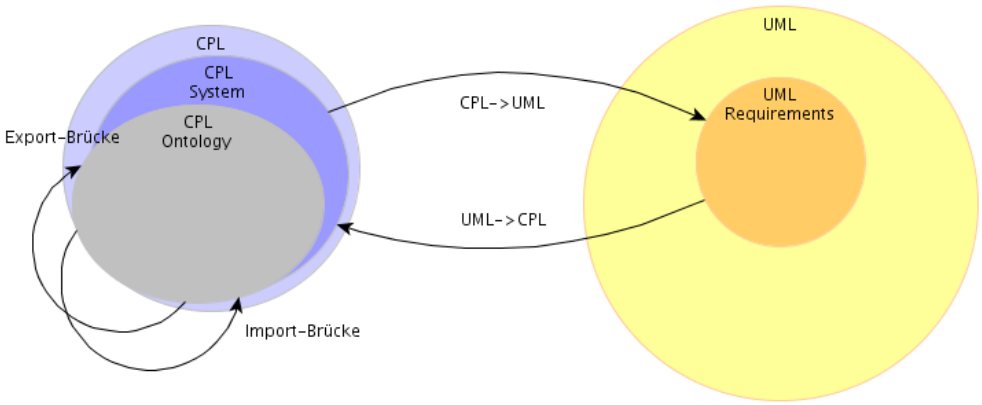


Abbildung 4.9: Eingabe- und Ausgabe-Sprachen der Transformationen

das Verständnis der Anforderungen bestimmte Elemente von *CPL* gehören nicht zu diesen Untersprachen. *CPL Ontologie* verzichtet noch zusätzlich auf die projektspeziefische Elemente.

Auf der anderen Seite (Abbildung 4.9, rechts) dient *UML Requirements* als das Ziel und die Quelle für die Transformationen zwischen *CPL* und *UML*. Das ist ein relativ kleiner Teil von *UML*, der für die Abbildung von Anforderungen aus einem konzeptuellen Modell mit *UML* erforderlich ist. Es handelt sich hauptsächlich um die Sprachelemente, die in *UML* für Beschreibung von Klassendiagrammen verwendet werden.

Im Folgenden wird detailliert erläutert, auf welchen Ausschnitten des jeweiligen Meta-Modells die Transformationen definiert sind. Das bedeutet allerdings *nicht*, dass die Transformationen ausschließlich Elemente aus den Sprachen *CPL System* oder *UML Requirements* als Eingabe akzeptieren, sondern, dass sie bei der Transformation berücksichtigt werden und nur Elemente aus diesen Sprachen erzeugt werden können. Als Quelle für die Transformationen können dagegen beliebige *CPL*- bzw. *UML*-Modelle dienen. Aus diesem Grund und wegen der besseren Lesbarkeit werden die Namen der Untersprachen im folgenden Text nicht explizit verwendet. Bei Transformationen wird implizit *CPL System* und *UML Requirements* angenommen, bei den Brücken *CPL Ontologie*.

Für die Transformationen relevanter Ausschnitt des *CPL*-Meta-Modells

Die überwiegende Anzahl der *CPL*-Meta-Modell-Elemente wird von den Transformationen unterstützt. Aus diesem Grund werden in diesem Abschnitt nur die Elemente beschrieben, die ausgehend von dem *CPL*-Meta-Modell (vergleiche Abschnitt 4.1.2) bei den Transformationen ausgelassen werden. Der so beschriebene Ausschnitt dient sowohl als Eingabe-Sprache für die CPL→UML- als auch als Ausgabe-Sprache für die UML→CPL-Transformaß-tion.

Da *UML* im Vergleich zu *CPL* eine deutlich größere Ausdruckskraft besitzt, findet sich nur für wenige *CPL*-Meta-Model-Elemente keine adäquate Entsprechung in *UML*. Bei den meisten ausgelassenen Elementen handelt es sich damit in erster Linie um Elemente, die in der Sprache *CPL* für die Kommunikation mit einem Fachverantwortlichen und für das bessere Verständnis des Gegenstandsbereichs vorgesehen sind. Dazu gehören QuantityDescription, AdditionalCardinality, Example oder auch die jedem ModelingElement zur Verfügung stehende textuelle Beschreibung description. Die letzternannte könnte zwar als Kommentar an das entsprechende *UML*-Element angehängt werden, diese Lösung würde aber entsprechende Diagramme mit Kommentaren überladen, die auf der Ebene eines Analyse-Modells nicht mehr wesentlich zur Systementwicklung beitragen.

Die *CPL*-Elemente vom Typ QuantityDescription werden benutzt, um die typische Anzahl der Objekte von diesem Typ in einer Domäne zu beschreiben. Interessanterweise existiert in *UML* ein ähnliches Konstrukt mit dem Namen MultiplicityElement. Allerdings trägt eine Angabe von QuantityDescription in *CPL* einen informativen und in *UML* einen verbindlichen Charakter. Aus diesem Grund, und um dadurch mögliche Missverständnisse auszuschließen, wurde auf die entsprechende Transformationsregel verzichtet.

AdditionalCardinality ist ein Element der Sprache *CPL*, mit dem mehrstellige Beziehungen dem Fachverantwortlichen besser zugänglich gemacht werden sollen. Es wird benutzt, um Kardinalitäten zwischen *je zwei* an der Beziehung beteiligten ThingTypes festzuhalten. Dieses Modellierungselement macht deutlich, auf welcher Grundlage die allgemeinen Kardinalitäten einer mehrstelligen Beziehung entstanden sind. Ein entsprechendes Element existiert in *UML* nicht. Einerseits aus diesem Grund, und weil die Informationen aus den AdditionalCardinality-Elementen in die Kardinalitäten der Beziehung eingeflossen sind und somit für die weitere Systementwicklung

keine entscheidene Rolle mehr spielen, wird auf eine Transformation verzichtet.

Obwohl Example in *CPL* im Wesentlichen das gleiche Konzept wie die Instance in *UML* darstellt, unterscheiden sie sich in der tatsächlichen Umsetzung sehr stark. In *UML* bilden die Instance-Elemente die Struktur der entsprechenden Klasse nach und zeigen, wie die entsprechenden Attribute belegt werden. *CPL* knüpft dagegen die Examples an ThingTypes, die selbst keine innere Struktur besitzen, und die Beschreibung eines Examples erfolgt in natürlicher Sprache. Diese Unterschiede sprechen gegen eine Transformation von Example nach Instance.

In einem Analyse-Modell verlieren Synonyme, die in einem konzeptuellen Modell für die Kommunikation mit dem Fachverantwortlichen und im Zusammenhang mit den Anforderungsdokumenten hilfreich sind, ihre Relevanz. Auf dieser Ebene ist es entscheidend, welcher Dingtyp aus einer Synonym-Gruppe als eine Klasse oder ein Attribut umgesetzt wird. Dafür ist die vorgestellte Änderung an dem *CPL*-Meta-Modell (siehe 4.1.1) nützlich. Im Falle einer existierenden isSynonymeTo-Beziehung werden bei der Transformation nur als mainSynonyme ausgezeichnete ThingTypes berücksichtigt. Die Beziehung selbst und über sie verbundene weitere Dingtypen sind kein Gegenstand der Transformationen.

Die Meta-Klasse CallingActor wird vorerst nicht von Transformationen berücksichtigt, da die aktuelle Version sich auf die Konstrukte konzentriert, die der statischen Modellierung zuzuordnen sind. Mit CallingActor ist es möglich, Operationsaufrufe analog zu den *Interaktionsdiagrammen* von *UML* zu modellieren. Einige Untersuchungen zur Transformation von dynamischen Aspekten eines *CPL*-Modells wurden bereits durchgeführt [KVH+05]. Eine entsprechende Umsetzung mit *ATL* steht noch aus.

Für die Transformationen relevanter Ausschnitt des *UML*-Meta-Modells

Die Ausgabesprache für die CPL→UML-Transformation und die Eingabesprache der UML→CPL-Transformation nutzen einen Ausschnitt des umfangreichen *UML*-Meta-Modells, der in dieser Sprache haupsächlich für die Beschreibung der Elemente eines *UML*-Klassen-Diagramms verwendet wird. Zusätzlich zu dieser Eingrenzung werden Elemente des Meta-Modells wie zum Beispiel ParameterDirectionKind und VisibilityKind[27], die zu den Entwuf-

[27] VisibilityKind legt die Sichtbarkeit von *UML*-Elementen und ParameterDirectionKind die Form des Zugriffs auf die Parameter einer *UML*-Operation fest.

spezifischen Elementen zugerechnet werden können und keine Entsprechung auf der Ebene eines *CPL*-basierten konzeptuellen Modells besitzen, aus der Spache der Transformationen ausgenommen.

Eine Zusammenstellung der *UML*-Meta-Modell-Elemente, die von den Transformationen benutzt werden, ist in der Abbildung 4.10 dargestellt. Dieses Diagramm setzt sich aus mehreren Diagrammen der *UML*-Spezifikation [OMG05b] zusammen. Das zentrale Element dieses Diagramms bildet die Meta-Klasse Class. Die für das *UML*-Klassendiagramm typische Möglichkeit, Klassen durch ihre Attribute (Property) und Operationen (Operation) zu beschreiben, gehört zum Gegenstand der Transformationen. Auch bekannte Beziehungen wie Generalisierung (Generalization)[28] und allgemeine Form der Assoziation (Association) findet sich in dem für die Transformationen relevanten *UML*-Sprachausschnitt wieder. Die Assoziationsenden einer Assoziation, die ebenfalls mit der Meta-Klasse Property festgelegt werden, können als StructuralFeature mit den Multiplizitäten MultiplicityElement versehen werden.

Die Operationen können zusätzlich zu den Parametern über Vor- (precondition), Nach- (postcondition) und Rumpf-Bedingungen (bodyCondition) verfügen, mit denen sich die Operationen spezifizieren lassen. Die letztgenannte ist eine Besonderheit von *UML*, die auf der üblichen, späteren Umsetzung der *UML*-Modelle in eine *Objekt-orientierte Sprache* beruht. Durch Bildung von Unterklassen können Operationen überladen und mittels *Polymorphie* Kontext-abhängig aufgerufen werden.

UML schreibt vor, dass die Vor- und Nachbedingungen unabhängig vom Kontext, in dem sie aufgerufen werden, für alle gleichnamigen Operationen einer Spezialisierungskette gelten. Dies entspricht auch der Behandlung dieser Bedingungen auf der konzeptuellen Ebene, wofür *CPL* entsprechende Konstrukte PreCondition und PostCondition anbietet. Anders verhält es sich mit den Rumpf-Bedingungen, die in *UML* speziell für polymorphe Operationen vorgesehen sind. Sie sind eine spezielle Nachbedingung, die allerdings in unterschiedlichen Klassen einer Spezialisierungskette unterschiedlich ausfallen darf.

Eine Alternative bei der Umsetzung der Transformationen wäre der Verzicht auf die Berücksichtigung von Rumpf-Bedingungen. Allerdings würde in diesem Fall auch die für die konzeptuelle Ebene wichtige Information über diese Operation verloren gehen. Deswegen beinhaltet der *UML*-Sprachausschnitt der Transformationen das bodyCondition-Element.

[28] Generalisierung ist an der Meta-Klasse Classifier angehängt.

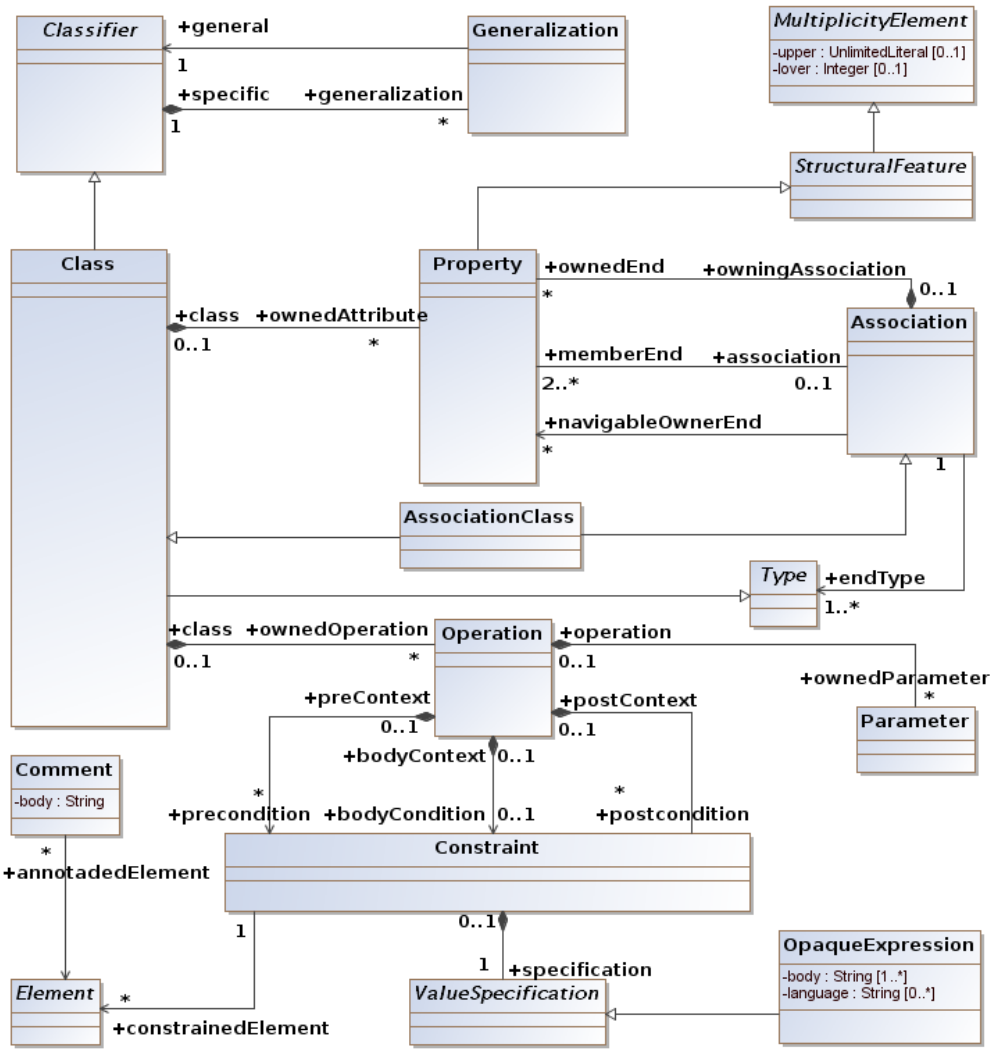


Abbildung 4.10: Elemente des *UML*-Meta-Modells, die von CPL→UML- und UML→CPL-Transformation verwendet werden.

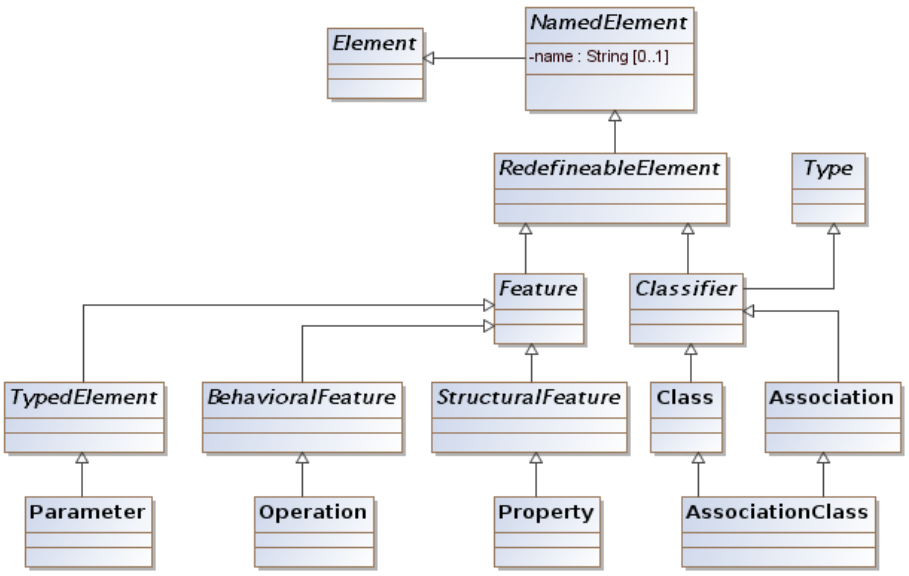


ABBILDUNG 4.11: Benennung von *UML*-Meta-Modell-Elementen

Zusätzlich zu den Operationen können in *UML* alle Sprachelemente vom Typ Element mit Constrains versehen werden. Analog werden auch Kommentare behandelt, die auf die in *CPL* neu eingeführte Meta-Klasse Comment (siehe Kapitel 4.1) abgebildet werden.

Die Abbildung 4.11 stellt eine Hierarchie aus *abstrakten* Meta-Klassen dar, die die Einordnung von *UML*-Meta-Modell-Elementen wie Property, Class usw. im Rahmen des *UML*-Meta-Modells zeigen. Diese abstrakten Meta-Klassen gehören nicht direkt zum Gegenstand der Transformationen, da keine Modellelemente von diesem Typ in Modellen vorkommen können. Trotzdem spielt diese Hierachie im Übersetzungsprozess eine wichtige Rolle, da mittels der abstrakten Meta-Klassen Attribute wie name den eigentlichen Meta-Klassen zugeordnet werden. Diese Hierarchie in der Kombination mit der Abbildung 4.10 verdeutlicht außerdem, für welche Elemente die Beziehungen in der Abbildung 4.10 gelten. Zum Beispiel stehen Kommentare (Comment) allen von Element abgeleiteten Meta-Klassen zur Verfügung (annotadedElement) und werden von den Transformationen berücksichtigt.

Analog zu den *CPL*-Modellen können *UML*-Modelle, die von *UML* nach *CPL* transformiert werden sollen, Elemente enthalten, die auf Meta-Modell-Elementen basieren und die in den beiden Diagrammen dieses Abschnitts nicht dargestellt sind. Die UML→CPL-Transformation kann problemlos mit diesen Modellen umgehen, allerdings werden ausschließlich Ele-

mente transferiert, die in den Abbildungen 4.10 und 4.11 vorgestellt wurden. Die CPL→UML-Transformation kann ausschließlich die vorgestellten Elemente erzeugen.

4.2.4 CPL→UML-Transformation

Ist die Abstimmung mit den Projektverantwortlichen durchgeführt worden und stehen die Systemanforderungen fest, besteht der nächste Schritt in der Ableitung des entsprechenden *Analyse-Modells*, das den weiteren Schritten im Projekt als Ausgangsbasis dient. Auf Grund der weiten Verbreitung der Objekt-orientierten Vorgehensweise und der damit oft verbundenen Verwendung der Modellierungssprache *UML* wurde zunächst eine Transformation von *CPL* nach *UML* entwickelt, die im Folgenden vorgestellt wird[29].

Bereits im Rahmen von *CPM* entstand eine Transformations-Unterstützung[30] von *CPL* nach *UML*. Es handelte sich allerdings um eine in *Java* geschriebene *direkte Manipulation*, die auf *Vorschriften* und *Empfehlungen* aufbaute. Die Vorschriften wurden für die Situationen definiert, in denen eine eindeutige Entscheidung (z.B. ob Klasse oder Attribut) anhand der ergänzenden Informationen (wie zum Beispiel der beteiligten Relationen) möglich ist. Die Empfehlungen schlagen in einer gegebenen Situation die jenige Variante vor, die auf Grund der ergänzenden Informationen als wahrscheinlicher erscheint. Als Eingabe-Modell diente eine Datenbank mit *CPL*-Elementen und als Ausgabe wurden Diagramme in einem proprietären graphischen *UML*-Editor erzeugt.

Dieses Transformations-Regelwerk konnte aus mehreren Gründen nicht in den *OBSE*-Prozess integriert werden. Zunächst baute es auf veralteten Versionen von *CPL* und *UML*[31] auf. Da die Transformationen nicht mit Hilfe von Meta-Modellen definiert wurden, hätten sie nur mit einem sehr hohen Aufwand in die *OBSE*-Infrastruktur integriert werden können. So hätte man zwei zusätzliche Transformationen definieren müssen, die einerseits zwischen der Datenbank und den aktuellen Meta-Modell-basierten *CPL*-Modellen und andererseits zwischen der graphischen und *XMI*-Darstellung von *UML*-Modellen vermitteln sollten. Eine Umsetzung der beiden Transformationen hätte sicherlich mehr Zeit in Anspruch ge-

[29] Als Ergänzung sind weitere verbreitete Modellierungssprachen wie *Business Process Execution Language (BPEL)* und *XML Process Definition Language (XPDL)* vorstellbar (siehe Ausblick im Kapitel 8).

[30] Für weiterführende Informationen siehe [MK02] und [Ker01].

[31] Es wurden Diagramme basierend auf der *UML* Version 1.x generiert.

***CPL*-Meta-Klasse**	***UML*-Meta-Klasse(n)**
ThingType	AssociationClass oder Class oder Property
Perspective	Property
ConnectionType	Association
IsA	Generalization
OperationType	Operation
CooperationType	keine direkte Entsprechung
ExecutingActor	keine direkte Entsprechung
Parameter	Parameter
Condition	Constraint mit OpaqueExpression
PreCondition	Constraint mit OpaqueExpression
PostCondition	Constraint mit OpaqueExpression
Costraint	Constraint mit OpaqueExpression
Comment	Comment

TABELLE 4.3: Übersicht über die Meta-Klassen-Abbildung für die CPL→UML-Transformation

nommen als die CPL→UML-Transformation mittels *ATL* neu zu definieren.

Vorteilhaft für die Überarbeitung der vorhandenen Transformationsvorschrift erwies sich außerdem die Möglichkeit, einige Informationen aus der vorhandenen Beschreibung wie die Regel-Bedingungen wiederzuverwenden oder als Anhaltspunkte für neue Regeln zu nutzen. Das sprach zusätzlich für die Neudefinition der Transformation und gegen die Einbettung der vorhandenen Transformation in die *OBSE*-Infrastruktur. Die eigens für den *OBSE*-Prozess entwickelte UML→CPL-Transformation wurde von Anfang an mit *ATL* umgesetzt. Das ermöglichte außerdem, im Weiteren auf der dort gesammelten Erfahrung im Umgang mit dieser Sprache aufzubauen. Durch die Umsetzung der CPL→UML-Transformation mit *ATL* wurde der gesamte Transformationsansatz vereinheitlicht.

Die Tabelle 4.3 präsentiert einen Überblick darüber, wie die Meta-Klassen des *CPL*-Meta-Modells auf die Meta-Klassen des *UML*-Meta-Modells abgebildet werden. Abstrakte Klassen der beiden Meta-Modelle berücksichtigt diese Tabelle nicht, da von diesen Klassen keine Modellierungselemen-

te direkt instanziiert werden können, die als Eingabe oder Ausgabe der Transformation fungieren können. Die entsprechenden Transformationsregeln werden in den folgenden Kapiteln ausführlich beschrieben.

Transformationsregeln für den *CPL*-Zusammenhangstyp

Die im Kapitel 4.1 eingeführte Gruppierung der Sprachelemente in Elemente 1. (ModelingElement) und 2. Klasse (ModelingElementDependent) erwies sich als vorteilhaft bei der Definition der *ATL*-Regeln. Die Sprache bietet einen zweistufigen Mechanismus basierend auf Regeln (Rules) und diesen untergeordneten, so genannten Lazy Rules. Die Ersten werden benutzt, um die Hauptmodellierungselemente der Sprache *CPL* zu transformieren. Lazy Rules werden dagegen bei Bedarf aufgerufen, um die abhängigen Elemente – falls vorhanden – zu transformieren.

Diese Differenzierung führt zu wenigen Haupt-Regeln, da der Mustervergleich allein auf das zu transformierende Element eingeschränkt werden kann. Dadurch müssen keine Muster für die Hauptmodellierungselemente mit zugeordneten bzw. fehlenden Elementen 2. Klasse (und allen möglichen Kombinationen!) einzeln erstellt werden. Diese Aufgabe übernehmen die Lazy Rules, die einen eigenen Mustervergleich durchführen. So wird die Identifikation und Transformation von Elementen 2. Klasse auf sie ausgelagert.

Die beiden Spezialisierungen der abstrakten Klasse GeneralConnectionTyp (IsA und ConnectionType mit den zugehörigen Perspektiven) können ohne Einschränkung auf die Generalisierung (Generalization) bzw. Assoziationen (Association) mit den zugehörigen Property-Elementen abgebildet werden. Die entsprechenden Regeln werden in dem Listing 4.2 vorgestellt.

Durch den vergleichbaren Aufbau der Generalisierung in *CPL* und *UML* fällt die Transformationsregel IsA2Generalization relativ einfach aus. Bei der Erzeugung von Assoziationen (ConnectionType2Association-Regel, Zeile 10) müssen in *UML* die entsprechenden Assoziationsenden ebenfalls generiert werden. Dies erfolgt mit Hilfe der Elemente 2. Klasse Perspective, die einem ConnectionType zugeordnet sind. Sie werden in der Perspective2Property-Regel generiert und dann der Assoziation zugewiesen. Da es mindestens zwei sind, werden sie in Zeile 19 als eine Menge behandelt (collect). Die Hilfsoperation getModel ist erforderlich, um die erstellte Assoziation dem *UML*-Modell zuzuordnen.

```
1  rule IsA2Generalization {
2
3      from isa : CPL!IsA
4
5      to gen : UML!Generalization (
6                          general <- isa.superType,
7                          specific  <- isa.subType )
8  }
9
10 rule ConnectionType2Association {
11
12     from ct : CPL!ConnectionType (
13                     ct.connectiontypeDeterminer.oclIsUndefined() and
14                     not thisModule.connectToAttribute(ct))
15
16     to assoc : UML!Association (
17                      name <- ct.name,
18                      memberEnd <- ct.connectsThrough ->
19                            collect(e | thisModule.Perspective2Property(e)))
20
21     do { thisModule.getModel().packagedElement <- assoc; }
22 }
23
24 unique lazy rule Perspective2Property {
25
26     from persp: CPL!Perspective
27
28     to property: UML!Property (
29                     name <- persp.name,
30                     lower <- persp.minCardinality,
31                     upper <- persp.maxCardinality,
32                     type <- persp.involvedThingType )
33 }
34
35 helper def : getModel() : UML!Model =
36                     UML!Model.allInstances()->asSequence()->first();
```

LISTING 4.2: Transformationsregel für die Beziehungen in *CPL*

Assoziationsklassen sind in *UML* sowohl von Assoziationen als auch von Klassen abgeleitet (siehe Abbildung 4.11). Deswegen werden sie nicht innerhalb der ConnectionType2Association-Regel, sondern in dem folgenden

Abschnitt zusammen mit anderen Dingtypen behandelt. Aus diesem Grund verhindert die Bedingung in der Zeile 13, dass für einen vergegenständlichten Zusammenhangstyp eine Assoziation angelegt wird. Die zweite Beziehung in dieser Regel sorgt dafür, dass für ein Attribut ebenfalls keine Assoziation erstellt wird, da die Attribute in *UML* zwar mit Hilfe von Proporties, aber ohne zugehörige Assoziation definiert werden.

Transformationsregeln für den *CPL*-Dingtyp

Zunächst kann die zweistufige Regelkonstruktion zu der Annahme führen, dass die CPL→UML-Transformation ausgehend von den fünf[32] Hauptregeln (Rules) für die Hauptmodellierungselemente definiert werden kann, die um mehrere Lazy Rules ergänzt werden. Ein Ergebnis der auf diese Weise aufgebauten Transformation führt in der Tat zu einem gültigen *UML*-Modell, bei dem allerdings alle Dingtypen auf *UML*-Klassen abgebildet werden[33]. Was bei einem konzeptuellen *CPL*-Modell eine gewünschte Darstellung der Sachverhalte eines Gegenstandsbereichs war, ist im Bereich der Systemmodellierung untypisch. Mit Blick auf die später folgende Implementierung spielen auf der Ebene des Analyse-Modells, die das Ergebnis der CPL→UML-Transformation sein soll, Konzepte wie *Datenkapselung* und *Objektorientierung* eine wichtige Rolle, wodurch es notwendig wird, zwischen Klassen und Attributen zu unterscheiden.

Eine Klasse hat die Kontrolle über ihre Attribute und gibt einen Zugriff auf diese entsprechend der Sichtbarkeitsregeln nach außen frei. Somit soll ein Dingtyp beim Übergang von einem konzeptuellen zu einem Analyse-Modell entweder auf eine Klasse oder ein Attribut einer Klasse transformiert werden. Es handelt sich dabei um eine *Entwurfsentscheidung*, die an dieser Stelle getroffen wird und von einem System zu einem anderen unterschiedlich ausfallen kann. Aus diesem Grund wurden *mehrere Regeln* für die Transformation eines Dingtyps entwickelt, um eine situationsabhängige Abbildung auf Klassen, Assoziationsklassen oder Attribute zu ermöglichen. Welche dieser Regeln zur Anwendung kommt, wird mittels *Muster* bestimmt, die die speziellen Situationen wie zum Beispiel das Vorhandensein einer Vergegenständlichung eines Zusammenhangstypen identifizieren und somit ein Zeichen für eine passende Abbildung (in diesem Fall auf eine

[32] Der Zusammenhangstyp gliedert sich in zwei Beziehungen: IsA und ConnectionType wie in dem Listing 4.2 vorgestellt.

[33] Begründet dadurch, dass für einen Dingtypen in diesem Fall nur eine Hauptregel zur Verfügung steht, die automatisch eine Klasse erzeugen wird.

Assoziationsklasse) liefern.

Die angesprochene Entwurfsentscheidung treffen – analog zu den klassischen Projekten - Entwickler (Architekten) des zukünftigen Systems. Die Anhaltspunkte dafür liefert insbesondere die Umgebung eines Dingtyps, auf die die Mustererkennung angewendet wird. Zur Umgebung zählen alle Beziehungen, in die dieser Dingtyp (DT1) involviert ist. Für eine Modellierung als Attribut ist mindestens ein nicht reflexiver Zusammenhangstyp erforderlich, der eine Zuordnung zu einem Dingtyp (DT2) herstellt. So kann DT2 zu einer Klasse umgewandelt werden und den Dingtyp (DT1) als ein Attribut aufnehmen. Die typische Umgebung eines Attributs ist somit eine einzelne Beziehung zu einem weiteren Dingtypen (in diesem Fall DT2). Wird sie größer, indem der Dingtyp (DT1) in weitere Beziehungen (insbesondere Generalisierung) involviert wird, ist das ein Zeichen dafür, dass er als eine Klasse und nicht als Attribut modelliert werden muss. Der Grund dafür ist, dass *UML* keine Beziehungen zu den Attributen erlaubt, so dass diese verloren gehen würden, und das entstehende *UML*-Modell sich dadurch semantisch zu stark von dem Eingabe-Model entfernen würde.

In den meisten Fällen kann eine Entscheidung, ob ein Dingtyp auf eine Klasse oder ein Attribut abgebildet wird, automatisch anhand seiner Umgebung getroffen werden. Dafür verwendet die CPL→UML-Transformation sieben Muster, die bestimmen, welche Transformationsregel (ThingType2Class, ConnectionTypeDeterminer2AssociationClass oder ThingType2Attribute) angewendet werden soll.

Diese Regeln und die für ihre Aktivierung zuständigen Muster sind in der Tabelle 4.4 aufgelistet[34]. Sie sind so konzipiert, dass in den ersten fünf Mustern geprüft wird, ob eine (Assoziations-)Klasse erzeugt werden muss. Erst wenn das nicht der Fall ist, wird das Muster sechs ausgewertet. Das siebte Muster ist von der Entscheidung, ob eine Klasse oder ein Attribut erzeugt wird, unabhängig und behandelt Dingtypen, die als Perspektiven-Bezeichner eingesetzt werden.

Bevor diese Muster ausführlich vorgestellt werden, möchte ich an dieser Stelle einen Einblick geben, wie die automatische Erzeugung von Klassen und Attributen von einem Entwickler beeinflusst werden kann. Eine ohne diesen manuellen Eingriff ablaufende CPL→UML-Transformation wird in jeder Situation, auf die das sechste Muster zutrifft, ein Attribut erzeugen. Das kann dazu führen, dass in einigen Situationen Attribute erzeugt werden,

[34] Die Muster wurden unter Berücksichtigung der vorherigen direkten Transformation [MK02] entwickelt.

die zwar zu einem gültigen *UML*-Modell führen, aber aus konzeptueller Sicht betrachtet keine Attribute sind.

Die Beispielsituation in der Abbildung 4.12 demonstriert, dass eine von dem Automatismus abweichende Entscheidung sinnvoll sein kann. Nach der Transformation wird dieses *CPL*-Modell zu einem *UML*-Modell mit der Klasse LKW mit dem Attribut PKW transformiert. Sicherlich ist das ein wohlgeformtes *UML*-Modell. Konzeptuell stehen beide Fahrzeugarten allerdings auf gleicher Ebene und eine Modellierung als Attribut, die eine gewisse Unterordnung voraussetzt, ist hier inhaltlich problematisch[35]. Dieser semantische Unterschied kann nicht anhand der Umgebung eines Dingtypen – und somit mit Hilfe der Muster – identifiziert werden.

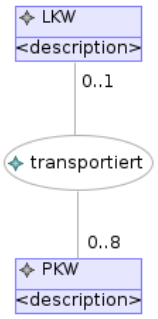


ABBILDUNG 4.12: PKW: Attribut oder Klasse?

Zusätzlich können weitere nicht-funktionale Eigenschaften des zu entwickelnden Systems wie zum Beispiel Erweiterbarkeit diese Entscheidung beeinflussen[36]. Diese Punkte sprechen für eine *semi-automatische* Umsetzung der Transformationsregel, die es einem Entwickler ermöglichen soll, diese Entwurfsentscheidung manuell zu treffen.

Allerdings werden die Vorschläge eines Entwicklers nur in solchen Situationen berücksichtigt, in denen sowohl eine Klasse als auch ein Attribut erzeugt werden können. Muss ein Dingtyp auf eine Klasse abgebildet werden, um eine Situation adäquat nach *UML* zu transformieren, wird die Entscheidung des Entwicklers außer Acht gelassen. Deswegen sind die Muster, die eine Transformationsregel zur Erzeugung einer (Assoziations-)Klasse vorschlagen, verbindlich, da sie nicht mit Hilfe eines Attributs nachgebildet

[35] Gleichzeitig zeigt dieses Beispiel, dass die Transformation in den Situationen ein unerwünschtes Ergebnis liefert, in denen das Modell wie in diesem Beispiel unvollständig ist. So kann davon ausgegangen werden, dass in einem vollständigen Modell die Dingtypen PKW und LKW in weitere Beziehungen involviert sind und möglicherweise sogar zu einem übergreifenden Dingtyp Fahrzeug generalisiert werden.

[36] Wenn es absehbar ist, dass ein Konzept der Domäne später in dem System eine wichtige Rolle spielen wird und somit seine Umgebung ausgebaut wird, ist es im Hinblick auf Erweiterbarkeit sinnvoll, entsprechenden Dingtyp auf eine Klasse abzubilden, auch wenn er zum aktuellen Zeitpunkt laut den Entscheidungsmuster einem Attributen entspricht.

werden können.

Gesteuert wird der Semi-Automatismus durch das Feld classification in der Meta-Klasse ModelingElement des *CPL*-Meta-Modells[37]. Bei einem ThingType wird das Feld classification während der CPL→UML-Transformation ausgewertet. Steht dort class, erzeugt die Transformation eine Klasse, steht dort attribute, wird entsprechend ein Attribut erzeugt, es sei denn, dies würde den Mustern 1-5 aus der Tabelle 4.4 widersprechen, die die Erzeugung einer Klasse vorschreiben. Wenn bei einem zu transformierenden ThingType keine Situation entsprechend der Tabelle 4.4 vorliegt und eine Vorgabe in dem Feld classification fehlt, wird standardmäßig eine Klasse erzeugt.

Jede Beziehung in dem *CPL*-Meta-Modell, die an die Meta-Klasse ThingType geknüpft ist, kann für die Entscheidung, ob dieser ThingType zu einem Attribut oder zu einer Klasse transformiert wird, eine wichtige Rolle spielen. Aus diesem Grund sind die Muster der CPL→UML-Transformation ausgehend von den Beziehungen in der Umgebung eines ThingTypes aufgebaut.

Das erste Muster sorgt dafür, dass eine *Generalisierung* vollständig übertragen wird, wodurch alle Elemente dieser Kette als Klassen abgebildet werden müssen. *Muster zwei* und *drei* behandeln die *Vergegenständlichung von Beziehungen*, die in *UML* mittels Assoziationsklassen modelliert wird. Um diese Situation adäquat übertragen zu können, ist es erforderlich, alle beteiligten ThingTypes zu Klassen zu transformieren. Mit dem *vierten Muster* wird sichergestellt, dass in *reflexive und mehrstellige* ConnectionTypes involvierte ThingTypen (die somit nicht auf ein Attribut transformierbar sind) auf Klassen abgebildet werden. *Muster fünf* behandelt *Operationen*. Sie werden in *UML* – analog zu Attributen – einer Klasse zugeordnet. Aus diesem Grund müssen alle Dingtypen, die für das Ausführen einer oder mehreren Operationen (OperationType) zuständig sind, auf eine Klasse abgebildet werden.

Muster sechs steht für eine Situation, in der ein ThingType zu einem *Attribut* transformiert werden *kann*. Der typische Fall tritt bei $N = 1$ auf, bei dem ein ThingType bis auf eine Anbindung an einen ConnectionType alleinstehend ist. Die Einschränkung der Kardinalität in diesem Muster sorgt dafür, dass ein für ein Attribut in Frage kommender ThingType eindeutig dem anderen verbundenen ThingType zugeordnet ist. Da er in nur eine Be-

[37] Bei der UML→CPL-Transformation wird dieses Feld ebenfalls benutzt, um die Zuordnung eines zu transformierenden Elements festzuhalten. Falls ein Element durch Rücktransformation erzeugt wird, entspricht die Belegung dieses Feldes der ursprünglichen Verwendung des Elements im *UML*-Modell.

Nr.	Muster	Regel
1	Ein ThingType ist ein Element in einer IsA-Hierarchie. In diesem Fall existiert mindestens eine IsA-Beziehung, bei der entweder superType oder subType auf denThingType gesetzt sind.	ThingType2Class
2	Ein ThingType ist eine Vergegenständlichung von einem ConnectionType. In diesem Fall ist der connectionTypeDeterminer des ConnectionTypes auf den ThingType gesetzt.	ConnectionType-Determiner2-AssociationClass
3	Ein ThingType ist in mindestens eine vergegenständlichte Beziehung involviert. In diesem Fall ist er über eine Perspective mit einem (ConnectionType) verbunden, bei dem connectionTypeDeterminer gesetzt ist.	ThingType2Class
4	Ein ThingType, der in einer reflexiven bzw. mehrstelligen Beziehung beteiligt ist. Zwei Perspective-Elemente eines ConnectionTypes zeigen über das Merkmal involvedConnectionType auf den gleichen ThingType bzw. der ConnectionType besitzt mehr als zwei Perspektiven	ThingType2Class
5	Ein ThingType ist für das Ausführen eines Operationstypen verantwortlich. In diesem Fall existiert mindestens ein ExecutingActor, der mit diesem ThingType verknüpft ist.	ThingType2Class
6	Ein ThingType T hat N Zusammenhangstypen (ConnectionType), die zu jeweils unterschiedlichen ThingTypes führen und deren Perspective-Elemente auf der Seite von T in dieser Beziehung beim Feld maxCardinality auf den Wert 1 gesetzt sind. Außerdem können die Muster 1 – 5 nicht auf T angewendet werden und das Feld classification ist für $N = 1$ leer oder für $N \geq 1$ auf den Wert attribute gesetzt.	ThingType2-Attribute
7	Ein ThingType wird ausschließlich als Vergegenständlichung einer Perspektive verwendet. Dann zeigt der entsprechende perspectiveDeterminer auf einen ThingType, der keine weitere Beziehungen hat.	keine Transformation

TABELLE 4.4: Entscheidungsmuster bei der Transformation eines ThingTypes und durch sie zur Anwendung kommende Regel.

ziehung involviert ist, liegt es nahe, diesen thingType zu einem Attribut zu transformieren.

In dem allgemeineren Fall[38] ist die Situation ähnlich, allerdings ist jetzt ein für ein Attribut in Frage kommender Dingtyp mit verschiedenen weiteren Dingtypen über jeweils genau einen Zusammenhangstyp verbunden. Bei der Sprache *CPL* kann eine solche Konstellation durchaus einem Attribut entsprechen, das zur Charakterisierung von mehreren anderen Dingtypen verwendet wird. Zum Beispiel kann ein Dingtyp *Titel* dafür verwendet werden, um die entsprechende Eigenschaft den verschiedenen Konzepten einer Domäne zuzuordnen, die einen Titel besitzen (Vorlesung, Diplomarbeit, usw). Die Transformation zu einem Attribut bedeutet in diesem Fall, dass der entsprechende ThingType *allen* mittels ConnectionType verbundenen ThingTypes als Attribut zugewiesen wird.

Die Transformation ist aktuell so konfiguriert, dass der typische Fall mit $N = 1$ standardmäßig zu einer Abbildung auf ein Attribut führt. Soll ein Attribut – wie in dem allgemeinen Fall – mehreren Klassen zugeordnet werden, dann erwartet die Transformation von *CPL* nach *UML* in dem Feld classification eine Entscheidung des Entwicklers. Wird sie auf attribute gesetzt, dann wird auch in dem allgemeinen Fall ein Attribut erzeugt, ansonsten eine Klasse. Der Grund für den vorgesehenen Eingriff eines Entwicklers liegt darin, dass ein gemeinsam verwendetes Attribut nicht automatisch von einem an mehreren gleichartigen Beziehungen beteiligten Dingtypen unterschieden werden kann.

Eine weitere Beziehung, die von einem ThingType ausgeht, ist perspectiveDeterminer. Im Unterschied zu den bisher in den Mustern verwendeten Beziehungen eines ThingType beeinflusst sie nicht die Entscheidung, ob ein Attribut oder eine Klasse erzeugt wird. Mit perspectiveDeterminer wird in *CPL* ein Konzept ausgedrückt, das in *UML* im Wesentlichen den *Rollen* einer Assoziation entspricht. Während die Rollen in *UML* allerdings als Name (ein Attribut) des zugehörigen Assoziationsendes Property festgehalten werden, sind diese Rollen in *CPL* ein ThingType und können somit in alle Beziehungen involviert werden, die das *CPL*-Meta-Modell für einen ThingType vorsieht. *CPL* ist an dieser Stelle flexibler als *UML*.

Bei der Transformation wird ein ThingType, bei dem das Feld perspectiveDeterminer gesetzt ist, zunächst wie jeder andere ThingType behandelt und entsprechend den Mustern in der Tabelle 4.4 analysiert. Die durch perspectiveDeterminer definierte Beziehung wird nachträglich mit Hilfe einer *Lazy Rule* transformiert. Wird ein dem perspectiveDeterminer zugehöriger ThingType zu einem Attribut transformiert, enthält das konzeptuelle Mo-

[38] N ist in diesem Fall die Anzahl der Zusammehangstypen.

dell eine Situation, die dem Rollen-Konzept von *UML* entspricht. In diesem Fall ist es ausreichend, nur den Namen des in perspectiveDeterminer angegebenen ThingType auf den Namen des Property-Elements zu übertragen. Das Property-Element selbst wird bei der Transformation der zugehörigen Beziehung erzeugt.

Wenn ein für eine Rolle stehender ThingType nicht dem Muster für ein Attribut entspricht und auf eine Klasse abgebildet wird, muss trotzdem die oben beschriebene Zuweisung des Names an das entsprechende Property-Element durchgeführt werden. Obwohl in *UML* die Klasse, die diesen ThingType repräsentiert, unabhängig von der Beziehung ist, wird durch das Zuweisen des Namens die Übertragung der Information über die Rolle sichergestellt.

Ein perspectiveDeterminer kann zu einer speziellen Situation führen, bei der ein ThingType weder zu einem Attribut noch zu einer Klasse transformiert wird. Für die Erkennung einer solchen Situation sorgt das *siebte Muster* in der Tabelle 4.4. In diesem Fall wird – wie in den Fällen zuvor – der Name des ThingType auf den Namen von Property übertragen. Eine Transformation zu einer Klasse ist zwar technisch machbar, bringt für das *UML*-Modell allerdings keine Vorteile, da keine Zuordnung zur Property mit den Mitteln von *UML* hergestellt werden kann und in dem Modell lediglich eine alleinstehende Klasse erzeugt wird.

Im Listing 4.3 werden die Regeln gezeigt, die bei der Transformation eines Dingtyps entsprechend der Mustern in der Tabelle 4.4 eine Klasse oder ein Attribut erzeugen. Ergänzend dazu existiert die Regel ConnectionTypeDeterminer2AssociationClass, um Assoziationsklassen zu erstellen. Sie ist weitgehend analog zu der Regel für Assoziationen aus dem Listing 4.2 aufgebaut. Der Unterschied besteht darin, dass statt einer Assoziation eine Assoziationsklasse erstellt wird. Die Attribute werden einer Assoziationsklasse analog zum folgenden Listing 4.3 zugeordnet. Diese kombinierte Vorgehensweise bei der Transformation einer Assoziationsklasse entspricht ihrer Definition in *UML* als Spezialisierung von Class und Association (siehe Abbildung 4.11).

```
1  rule ThingType2Class {
2
3      from tt : CPL!ThingType (
4          not thisModule.isConnDeterminer(tt) and -- einschlaegige Regel
5          not thisModule.isAttribute(tt) and -- ist ein Attribut moeglich?
6          not thisModule.isOnlyPerspectiveDeterminer(tt))
7          -- keine alleinstehende Klasse fuer reine PerspectiveDeterminer
8
9      to c : UML!Class (
10         name <- tt.name,
11         ownedAttribute <- tt.seenThrough -- sammele Attribute
12             -> select( per | thisModule.isAttribute(thisModule.
13             getOpossitePerspective(per).looksAt))
14             -> collect(per | thisModule.ThingType2Attribute(thisModule.
15             getOpossitePerspective(per).looksAt)))
16
17     do { thisModule.getModel().packagedElement <- c; }
18 }
19
20 lazy rule ThingType2Attribute {
21
22     from tt : CPL!ThingType
23
24     to attr : UML!Property (
25         name <- tt.name,
26         upper <- 1 )
27 }
28
29 helper def : isAttribute (tt : CPL!Thingtype) : Boolean =
30     if not thisModule.isToBeClass(tt) and -- schliesse Muster 1-5 aus
31         not (tt.classification = 'class') and -- Semi-Automatismus
32         thisModule.canBeAttribute(tt) -- pruefe Muster 6
33         then true else false endif;
34
35 helper def : hasMultiDigitConnection (tt : CPL!ThingType) : Boolean =
36     tt.seenThrough -> select(per | per.belongsTo.connectsThrough
37             -> asSequence().size() > 2).notEmpty();
```

LISTING 4.3: Regel zur Erzeugung von Klassen und Attributen

Ein Attribut ist in *UML* ein von der Klasse, zu der es gehört, abhängiges Element. Um sicherzustellen, dass bei der Transformation kein Attribut unabhängig von einer Klasse erzeugt wird, wird es mit der Hilfe einer unter-

geordneten Regel *Lazy Rule* erstellt, die von der Transformationsregel für die Klassen aufgerufen wird.

Zunächst wird bei der Transformation eines Dingtyps (in der Zeile 4) geprüft, ob es sich um eine Vergegenständlichung eines Zusammenhangstyps (connectionDeterminer) handelt. Wenn nicht, dann wird geprüft, ob aus diesem Dingtypen ein Attribut erzeugt werden kann. Die Hilfsoperation isAttribute besteht aus mehreren Regeln, die die Muster aus der Tabelle 4.4 überprüfen und den Semi-Automatismus berücksichtigen (siehe Zeile 30).

Die Hilfsoperation hasMultiDigitConnection zeigt stellvertretend, wie die Mehrstelligkeit einer Beziehung aus dem Muster vier überprüft wird. Für ein Attribut liefern solche Hilfsfunktionen für die Muster 1 – 5 false und für das Muster sechs true. In diesem Fall wird auf die Transformation zu einer Klasse verzichtet. Ansonsten wird noch geprüft, ob der Dingtyp ausschließlich als eine Vergegenständlichung einer Perspektive agiert. Ist es der Fall, dann ist – entsprechend dem Muster sieben – eine Transformation nicht erforderlich. Wenn alle drei Bedingungen in der Transformationsregel für eine Klasse false[39] liefern, wird eine Klasse erzeugt. Ihr wird der Name des Dingtypen zugewiesen und die zugehörigen Attribute (Zeilen 11 bis 15) werden erstellt.

Zunächst sorgt der Filter in Zeile 12 dafür, dass alle mit diesem Dingtypen in Beziehung stehenden Dingtyp auf ihre Eignung für ein Attribut überprüft werden. Die Menge der Attribute wird in Zeile 14 nur für diese Elemente erstellt und jeweils die untergeordnete Regel ThingType2Attribute aufgerufen. Da die Prüfungen auf ein Attribut bereits durchgeführt wurden, ist diese Regel vergleichsweise einfach aufgebaut und konvertiert jeden ihr übergebenen ThingType zu Property. Bei dem Attribut wird sein Name und die Multiplizität gesetzt. Durch das Muster sechs ist garantiert, dass für diese die obere Grenze bei 1 liegen muss.

Die vorgestellte Regel ThingType2Class enthält außerdem Code, um den entsprechenden Klassen Operationen zuzuweisen. Dieser Code ist algorithmisch dem von einem Attribut analog und wurde aus Platzgründen im Listing 4.3 ausgelassen. Die entsprechende untergeordnete Regel, die dazu dient, Operationen selbst zu erzeugen, wird in dem folgenden Abschnitt erläutert.

[39] In diesem Fall liegt keine Assoziationsklasse, kein Attribut und keine reine Perspektivenbezeichnung vor.

Transformationsregeln für den *CPL*-Operations- und Kooperationstyp

Die Regeln für einen Operations- und Kooperationstyp hängen stark zusammen. Ein Operationstyp liefert Informationen, die für das Erzeugen einer *UML*-Operation (Operation) erforderlich sind. Ein Kooperationstyp ergänzt diese um die optionalen Informationen über die Vor- und Nachbedingungen dieser Operation. Analog zu *UML*-Operationen wird ein Operationstyp höchstens einem Dingtyp zugeordnet, der für seine Ausführung – realisiert durch die Meta-Klasse ExecutingActor – zuständig ist. Diese Gemeinsamkeit und die Tatsache, dass das Muster fünf in Tabelle 4.4 eine Abbildung dieses Dingtyps auf eine Klasse sicherstellt, vereinfachen die Transformation eines Operationstyps. Diese Transformation wird zusammen mit der Abbildung eines Kooperationstyps strukturell in der Abbildung 4.13 dargestellt.

Der Ausgangspunkt für die Transformation ist die Meta-Klasse OperationType. Sie wird auf die Operation des *UML*-Meta-Modells abgebildet. Die Zuordnung zu der zuständigen Klasse (Class) des *UML*-Modells wird mittels des Konstrukts ExecutingActor zusammen mit ThingType[40] hergestellt. Die Parameter sind in *CPL* fast identisch zu denen in *UML* definiert. Sie werden – da sie optional sind – während der Transformation mittels einer *Lazy Rule* von der Kombination Parameter/ThingType auf Parameter/Type abgebildet. Genauso optional und somit *Lazy Rule*-basiert ist die Auswertung des CooperationType, bei der mehrere Vor- (PreCondition) und Nachbedingungen (PostCondition) auf Constraint von *UML* mit entsprechenden Rollen precondition und postcondition abgebildet werden. Dabei dient Constraint als ein Container für die eigentliche Bedingung, die mit Hilfe der abstrakten Klasse ValueSpecification ausgedrückt werden kann.

Da ValueSpecification in dem *UML*-Meta-Modell nicht nur für eine Beschreibung von Vor- und Nachbedingungen, sondern auch für eine Festlegung von Multiplizitäten und Vorgabewerten von Parametern verwendet wird, besitzt sie eine Reihe von Klassen, die ValueSpecification weiter spezifizieren. Für die Übertragung von Bedingungen von *CPL* nach *UML* eignet sich insbesondere die Klasse OpaqueExpression[41]. Sie ermöglicht es, eine Liste mit Bedingungen zu formulieren und eine korrespondierende Liste von Sprachen anzugeben, in denen diese Bedingungen verfasst sind. Als Spra-

[40] Die Darstellung des ThingType c wird hier der Vollständigkeit halber mit aufgeführt. Dieser ThingType wird nicht im Rahmen dieser Regel, sondern zusammen mit allen anderen ThingTypes transformiert (siehe Listing 4.3).

[41] Siehe auch den *UML*-Sprachausschitt in Abbildung 4.10.

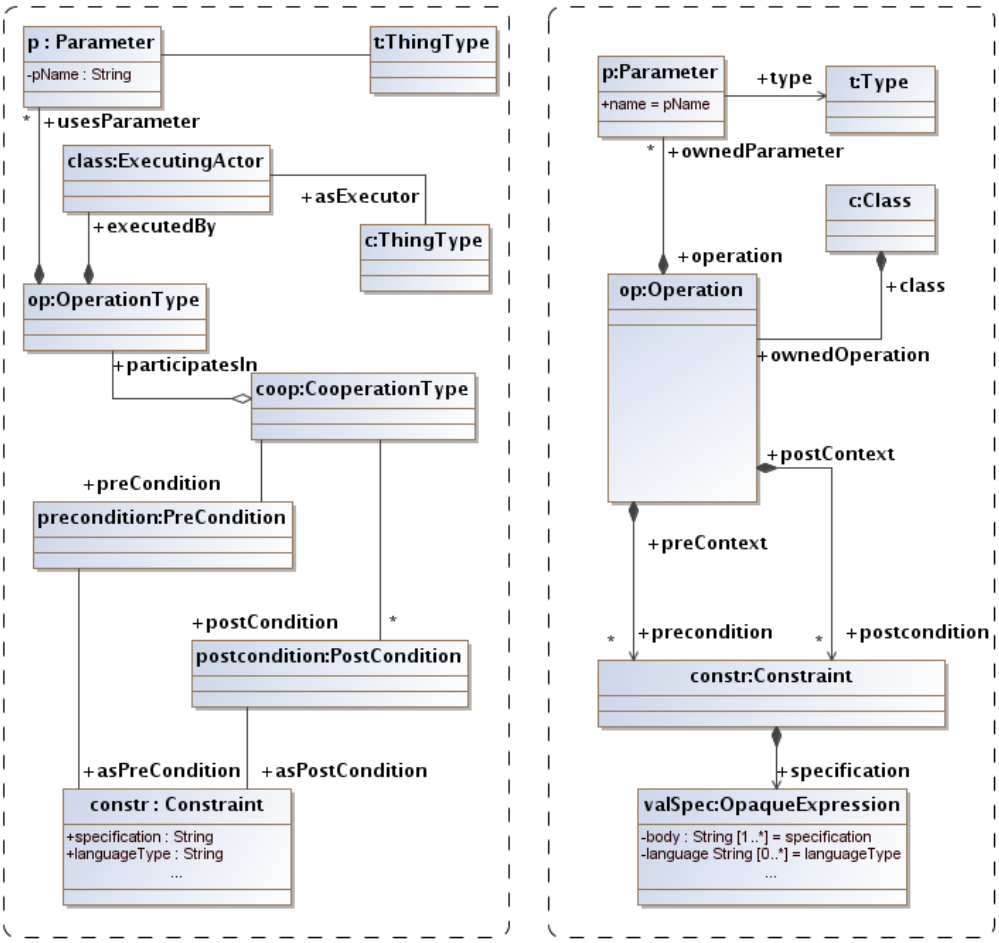


Abbildung 4.13: Visualisierung der Transformationen für die Operations- und Kooperationstypen

chen können sowohl formale Sprachen wie *OCL* als auch Programmiersprachen wie *Java* bis hin zur natürlichen Sprache verwendet werden.

In *CPL* werden die Bedingungen typischerweise in natürlicher Sprache verfasst, um sie für Fachverantwortliche zugänglich zu halten. Trotzdem ist in *CPL* die Möglichkeit vorgesehen, andere Sprachen zu benutzen und analog zu *UML* deren Typ (languageType in der Meta-Klasse Condition) anzugeben. Im Unterschied zu *UML* erlaubt *CPL* keine Listen aus Bedingungen und Sprachen, sondern nur die Eingabe von *einer* Bedingung und *der* zugehörigen Sprache. Für die Transformation ist dieser Unterschied unwesentlich. Die Daten aus *CPL* werden als erstes Element in die Listen von

OpaqueExpression eingetragen.

4.2.5 Analyse-Modell

Im Rahmen des *OBSE*-Prozesses besteht die Aufgabe der CPL→UML-Transformation darin, einen Entwickler beim Übergang von einem konzeptuellen Modell zum ersten System-Modell durch möglichst weitgehende Automatisierung zu unterstützen. Als Ergebnis dieser Transformation entsteht ein dem konzeptuellen Modell nahe stehendes *UML*-Modell. Der Schwerpunkt dieses Modells liegt – vergleichbar mit dem konzeptuellen Modell – auf der Darstellung der Elemente einer Domäne und deren Beziehungen. Es ist noch von dem späteren Entwurf des zukünftigen Systems unabhängig. Die ersten Entwurfsentscheidungen wie die Bildung von Attributen und ihre Zuordnung zu den Klassen gehören zwar zum Entstehungsprozess dieses Modells, sind allerdings nur in einzelnen Fällen durch die spezifische Umsetzung des zukünftigen Systems begründet. Hauptsächlich sind sie auf die gewählte Objekt-orientierte Paradigma und die Wahl der Modellierungssprache zurückzuführen.

Für System-Modelle dieser Art hat sich in der Literatur der Begriff *Analyse-Modell* etabliert [CRP06]. Viele Verfahren, die ihren Schwerpunkt auf die statische Modellierung einer Domäne legen, sprechen alternativ von einem *Domänen-Modell*. In dieser Arbeit wird ein Domänen-Modell als eine statische Komponente des Analyse-Modells angesehen und diesem untergeordnet. Das hängt damit zusammen, dass es im *OBSE*-Prozess ergänzend zu der statischen Modellierung einer Domäne möglich ist, dynamische Sachverhalte einer Domäne zu modellieren[42]. Dieser Ausschnitt des Gegenstandsbereichs wird im *Geschäfts-Modell* festgehalten (siehe Abbildung 3.2).

Betrachtet man die Entwicklung eines Software-Produkts nach *OBSE* als Ganzes, dient das Analyse-Modell in diesem Prozess als ein Bindeglied zwischen den neuen, durch die *OBSE* hervorgerufenen Infrastruktur-Elementen und Bestandteilen der konventionellen Software-Entwicklung, die – ausgehend von dem Analyse-Modell – für eine Implementierung des Software-Produkts benötigt werden. An dieser Stelle nutzt *OBSE* ein in der Software-Entwicklung bekanntes Modell, auf dem sowohl *MDA* als auch klassische Formen der Software-Entwicklung aufbauen [CRP06].

Abbildung 4.14 setzt die Modellierung der Beispiel-Situation dieses Kapitels fort und präsentiert das Analyse-Modell für das Sprechstundenver-

[42] Auch wenn die aktuellen Werkzeuge dies nicht durchgehend unterstützen.

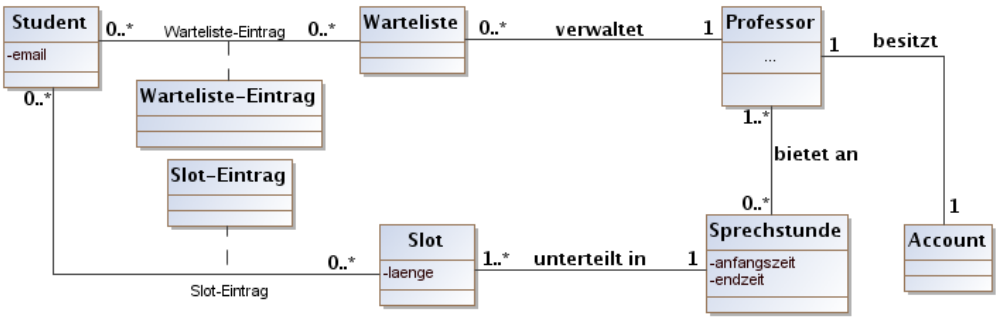


ABBILDUNG 4.14: Analyse-Modell des Projekt-Beispiels

waltungssystem. Dieses UML-Modell ist aus dem in der Abbildung 4.8 präsentierten *CPL*-Diagramm durch eine Anwendung der CPL→UML-Transformation entstanden.

Bis auf die Account-Klasse auf der rechten Seite des Diagramms wurden alle Elemente automatisch und ohne Vorgaben eines Entwicklers generiert. Die Klasse Account wurde gezielt mit betrachtet, um eine sich von dem Automatismus unterscheidende Transformation zu demonstrieren. Standardmäßig entspricht diese Klasse dem Attribut-Muster der Transformation. Allerdings handelt es sich in der vorliegenden Situation um einen Ausschnitt der Anforderungen, in dem das Konzept Account nicht weiter beschrieben wurde. Ein Entwickler kann sich – falls er wie in diesem Fall von unvollständigen Informationen ausgeht – für eine Umsetzung in eine Klasse entscheiden, die für weitere Veränderungen und Erweiterungen offen ist. Diese Modellierung kann auch als ein Diskussionspunkt bei weiteren Abstimmungen mit den Fachverantwortlichen dienen.

4.2.6 UML→CPL-Transformation

Das im vorhergehenden Abschnitt vorgestellte Analyse-Modell stellt eine Ausgangsbasis für die im weiteren Projekt-Ablauf anstehenden Verfeinerungen und damit verbundenen Entwurfs-Entscheidungen dar. Zum Beispiel kann die Assoziation unterteilt in in eine Komposition umgewandelt werden, um einem Entwickler zu signalisieren, dass Slot-Elemente abhängig von der sie beinhaltenden Sprechstunde implementiert werden sollen. Größere Veränderungen als in diesem kleinen Beispiel werden sich sicherlich bei der Anwendung von Entwurfsmustern ergeben. In der Regel werden durch diese Anpassungen weiterentwickelte Modelle im Projektverlauf zunehmend

Entwurfs- und Plattform-spezifischer und können als eine Verfeinerung von bereits erfassten Sachverhalten einer Domäne zwecks technischer Umsetzung angesehen werden.

In der Praxis kann man durchaus erwarten, dass parallel zu diesen Verfeinerungen auch relevante konzeptuelle Veränderungen bzw. Ergänzungen stattfinden. Die Gründe dafür können unterschiedlicher Natur sein. Zum Beispiel können neue oder veränderte Anforderungen, die – meistens aus Zeitgründen – ohne vorhergehende Analyse direkt umgesetzt werden, dazu führen aber auch nicht adäquat modellierte Sachverhalte eines konzeptuellen Modells, die in der Entwurfsphase nachträglich korrigiert werden.

OBSE soll den Projektbeteiligten kein starres, Wasserfall-artiges Korsett aufzwingen, das verlangen würde, dass alle vorhergehenden Schritte abgeschlossen und die erstellten Artefakte einwandfrei sind und als Maßstab für die darauf folgenden Schritte/Artefakte dienen. Im Gegenteil – begünstigt durch das Vorgehensmodell *EOS* (siehe Kapitel 2.3.2) – soll der *OBSE*-Prozess eine zyklische Software-Entwicklung sowohl auf der System-Ebene als auch auf der Ebene der einzelnen Module ermöglichen. Um das zu unterstützen, bietet der *OBSE*-Prozess eine Möglichkeit, in den späteren Phasen entstandene Änderungen z.B. nach einem Entwicklungszyklus in das konzeptuelle Modell zu übertragen. Dieser Schritt ist wichtig, um einerseits den zukünftigen Zyklen ein konzeptuelles Modell zu Verfügung zu stellen, das gewonnene Erkenntnisse aus späteren Phasen der vorhergegangenen Zyklen enthält. Andererseits können auf diese Weise konzeptuelle Erkenntnisse, die nach der Anforderungsanalyse gemacht werden, mit Hilfe der Domänen-Ontologie an weitere Projekte weitergegeben werden.

Um den Entwickler beim Übertragen von Änderungen aus einem Analyse-Modell in das zugehörige konzeptuelle Modell zu unterstützen, wurde speziell für *OBSE* die UML→CPL-Transformation entwickelt[43]. In ihrer aktuellen Version verwendet sie die gleichen Teilsprachen von *UML* und *CPL* wie die CPL→UML-Transformation [44]. Als Quelle für die UML→CPL-Transformation wurde gezielt das Analyse-Modell eines Projekts bestimmt. Dadurch entsteht eine kompakte, fest definierte Schnittstelle zwischen der *OBSE*-Infrastruktur und den daran anknüpfenden Elementen der Software-Entwicklung.

Technisch gesehen kann die Transformation auf ein beliebiges *UML*-Modell auch aus einer späteren Phase heraus angewendet werden[45]. Eine

[43] Für ausführliche Beschreibung der Transformation siehe auch [Ruß07].
[44] Siehe Kapitel 4.2.3.
[45] Durch die Einschränkung des *UML*-Sprachumfangs sorgt sie sogar dafür, dass viele

***UML*-Meta-Klasse**	***CPL*-Meta-Klasse(n)**
Class	ThingType
AssociationClass	ThingType
Property	ThingType
Association	ConnectionType
Generalization	IsA
Operation	OperationType und CooperationType
Parameter	Parameter
Constraint	Constraint oder PostCondition oder PreCondition
OpaqueExpression	keine direkte Entsprechung
Comment	Comment

TABELLE 4.5: Übersicht über die Meta-Klassen-Abbildung für die UML→CPL-Transformation

automatische Trennung zwischen konzeptuellen und Entwurfs-bezogenen Veränderungen eines Modells ist im allgemeinen Fall unmöglich. Deswegen soll durch die angestrebte Nähe des Analyse-Modells zum konzeptuellen Modell vermieden werden, dass bei der Rücktransformation neben konzeptuellen auch Entwurfsentscheidungen[46] auf die konzeptuelle Ebene übertragen werden.

Die UML→CPL-Transformation kann zusätzlich zu ihrer primären Aufgabe im *OBSE*-Prozess verwendet werden, um weitere Quellen zu erschließen, die für den Aufbau der Domänen-Ontologie verwendet werden können. Auf diese Weise können Erkenntnisse aus früheren bzw. von *OBSE* abgekoppelten, unter Verwendung von *UML* durchgeführten Projekten einfließen. Zum Beispiel können mit Hilfe der UML→CPL-Transformation Modelle, die aus den so genannten *open source-Projekten*[47] stammen, als eine weitere Quelle für die Domänen-Ontologie fungieren.

Diese Transformation wurde analog zu der CPL→UML-Transformation in *ATL* definiert. Einen kompakten Überblick über die Abbildung der Elemente bietet Tabelle 4.5. Um die Eigenschaften der Transformatio-

Entwurfs-spezifische Elemente bei der Transformation ausgelassen werden.

[46] Entwurfsentscheidungen sollen nicht an dem Analyse-Modell durchgeführt werden.

[47] Natürlich unter der Berücksichtigung der Lizenz-Bestimmungen.

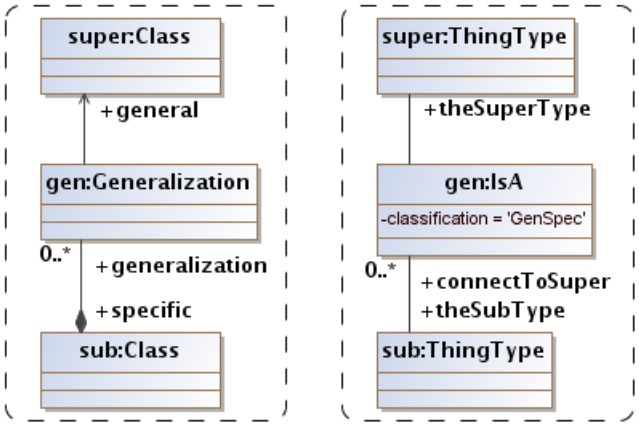


ABBILDUNG 4.15: Visualisierung der IsA-Transformation (Generalisierung) von *UML* nach *CPL*

nen wie Beschränktheit oder Stabilität positiv zu beeinflussen bzw. leichter überprüfen zu können, wurden die Regeln der beiden Transformationen nach Möglichkeit durch Vertauschen von from- und to-Bereichen einer Transformation-Regel definiert. Insbesondere bei den Regeln, die strukturell ähnliche Sprachelemente transformierten, war das ohne weiteres möglich. Ein gutes Beispiel dafür ist die Generalisierung in *UML* und ihr Gegenstück in *CPL* – die IsA-Beziehung.

Die Abbildung 4.15 zeigt die graphische Darstellung der *UML*-zu-*CPL*-Richtung. Durch das Vertauschen der Blöcke[48] entsteht die Regel für die Rücktransformation, die bereits im Code-Format im Listing 4.2 auf der Seite 115 vorgestellt wurde. Weitere Transformationsregeln, die auf diese Weise definiert werden können, sind die Regeln für Zusammenhangstypen/Assoziationen, vergegenständlichte Zusammenhangstypen/Assoziationsklassen, Operations- und Kooperationstypen/Operationen mit Bedingungen[49].

Deutliche strukturelle Unterschiede weisen die Sprachen *CPL* und *UML* insbesondere im Bereich von – aus *UML*-Sicht – Klassen, Attributen und Operationen auf. Während Dingtypen und Operationstypen in *CPL* zu den Elementen 1. Klasse gehören und somit auf der gleichen Ebene stehen, handelt es sich bei ihren Gegenstücken in *UML* (Klassen sowie Attributen und Operationen) um strukturell untergeordnete Sprachelemente. Die-

[48] Links steht bei diesem Diagramm ein Muster, das in einem Modell gesucht wird. Rechts ist das zu generierende Muster im Ausgabe-Modell dargestellt.

[49] Gemeint ist ausschließlich die Regel, um Operationen zu erzeugen. Ihre Zuordnung zu den Klassen kann nicht durch das Vertauschen realisiert werden.

ser Unterschied führt dazu, dass unterschiedliche Transformationsregeln erzeugt werden mussten und eine Definition der Transformationsregel durch das Vertauschen von from- und to-Abschnitten nicht möglich war. In der UML→CPL-Transformation entstanden neue Regeln, die zum Beispiel dafür sorgen, dass ein *UML*-Attribut wieder an eine Kombination aus Ding-, Zusammenhangstypen und den zugehörigen Perspektiven übersetzt wird (siehe Listing 4.4).

```
1  rule Attribute2ThingType {
2
3      from attr : UML!Property(attr.association.oclIsUndefined()
4                  and attr.opposite.oclIsUndefined())
5
6      to tt  CPL!ThingType(name <- attr.name),
7          con : CPL!ConnectionType(connectsThrough <- persp_class,
8                  connectsThrough <- persp_class),
9          persp_class : CPL!Perspective(looksAt <- attr.class,
10             maxCardinality <- 1)
11         persp_attr : CPL!Perspective(looksAt <- tt,
12             minCardinality <- attr.lower, maxCardinality <- attr.upper)
13 }
```

LISTING 4.4: Regel zur Transformation eines Attributs

Da ein Property-Element in *UML* sowohl für Attribute als auch für die Festlegung von Assoziationsenden verwendet wird, erfolgt zunächst in dem from-Abschnitt ein Ausschluss von Assoziationsenden attr.association.oclIsUndefined() und binären Assoziationen[50] attr.opposite.oclIsUndefined().

Anschließend werden in dem to-Abschnitt ein ThingType tt für das Attribut und der entsprechende ConnectionType erzeugt. Ein ThingType für die Klasse, die das Attribut attr enthält, wird unabhängig davon während der Anwendung der Transformationsregel für Klassen erstellt. Eine Verknüpfung des ConnectionTypes con mit dem ThingType der Klasse erfolgt in der Attribute2ThingType-Regel. Dafür dient das persp-class-Element.

Um die Beispielsituation vom Anfang dieses Abschnitts zu vervollständigen, wird im Folgenden ein Beispiel für eine mögliche Änderung an dem Analyse-Modell vorgestellt, die anschließend mit Hilfe der UML→CPL-Transformation in das konzeptuelle Modell übertragen werden kann. Das

[50] Sie können in *UML* alternativ ohne Angabe von Assoziationsenden direkt mit Hilfe der opposite-Beziehung aufgebaut werden.

im Abschnitt 4.2.5 vorgestellte Klassendiagramm (siehe Abbildung 4.14) enthält mindestens einen realistischen Anhaltspunkt für konzeptuelle Veränderungen. Die Kandidaten dafür sind die Klassen Student, Professor und Account.

Eine kleine Ergänzung der Anforderungen, die verlangen würde, dass Studenten ebenfalls einen Zugang zu dem System bekommen sollen, würde dazu führen, dass eine zusätzliche Assoziation besitzt zwischen den Klassen Student und Account entstehen würde. Nicht ausgeschlossen ist es auch, dass zur Vermeidung der Redundanz die Klassen Student und Professor zu einer neuen Klasse Person generalisiert werden und die Assoziation besitzt zu dieser Klasse verschoben werden wird (siehe Abbildung 4.16)[51]. Auch gemeinsame Attribute wie zum Beispiel E-Mail (email), Vor- und Nachname können – falls sie benötigt werden – in die Oberklasse Person verlagert werden.

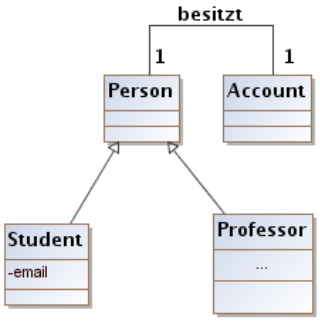


Abbildung 4.16: Änderung des System-Modells

Eine Transformation des Analyse-Modells mit diesem veränderten Ausschnitt führt zu einem neuen konzeptuellen Modell, das auch auf der Systemebene durchgeführte Änderungen enthält.

4.3 Infrastruktur-Elemente auf der Ontologie-Seite

In diesem Kapitel werden die letzten zwei Elemente der *OBSE*-Infrastruktur – Domänen-Ontologie und Import-/Export-Brücken – vorgestellt. Bedingt durch ihre Aufgabe, Wissen über eine Domäne zwischen den beteiligten

[51] Auf der Entwurfsebene sind diese und ähnliche Lösungen für vergleichbare Situationen in der Praxis verbreitet. Aus der ontologischen Sicht sind sie meistens problematisch, da die temporären und permanenten Konzepte einer Domäne in eine Klassen-Hierarchie eingeordnet werden. So handelt es sich in dem Beispiel bei den Objekten der Klassen Student und Professor um Rollen, die einer Person temporär zugewiesen werden können. Solche Situationen sind gute Anhaltspunkte für weitere Modell-Verbesserungen auf der konzeptuellen bzw. Domänen-Ontologie-Ebene.

Projekten zu vermitteln, stehen diese Infrastruktur-Elemente außerhalb der Projekte und werden von diesen gemeinsam verwendet.

4.3.1 Domänen-Ontologie

Betrachtet man die konzeptuellen Modelle der *OBSE*-Projekte als anwendungsspezifische Konzeptualisierungen eines Domänen-Ausschnitts, passt die bekannte Bezeichnung einer Ontologie als *„shared conceptualisation“* von Gruber [Gru93a] optimal auf die Rolle, die die Domänen-Ontologie in dem *OBSE*-Prozess spielt. Dementsprechend sammelt sie Konzepte und Beziehungen aus solchen Domänen-Ausschnitten, integriert sie zu einer gemeinsamen, anwendungsunabhängigen Sicht auf die Domäne und entspricht somit dem, was mehrheitlich unter einer Ontologie in der Software-Technik verstanden wird.

Für die meisten Ontologien[52] in der Software-Technik gilt, dass sie bereits vor ihrem ersten Gebrauch weitgehend vollständig modelliert und gefüllt sind. In diesem Punkt schlägt *OBSE* einen neuen Weg ein und verzahnt die Entwicklung einer Domänen-Ontologie mit ihrer Verwendung in Projekten. In diesem Fall findet der Wissens-Transfer bidirektional statt und die Entwicklung der Domänen-Ontologie verläuft evolutionär.

Eine derartige Umstellung führt zu einer Verschiebung der Schwerpunkte von Entwurfskriterien einer Ontologie, wie sie von Gruber aufgestellt wurden [Gru93b]. So wird die Vollständigkeit der Domänen-Ontologie in *OBSE* nicht mehr angestrebt, da sie nur insoweit eine Domäne abbilden kann, als das der Modellierung der Gegenstandsbereiche durch die bisherigen Projekte entspricht. Bezüglich der Herstellung der minimalen ontologischen Verpflichtung trägt der *OBSE*-Ontologie-Entwicklungsprozess *aktiv* bei. Diese Ontologie-Eigenschaft (siehe auch Kapitel 2.2.3) hängt überwiegend davon ab, wie konsistent die Elemente einer Ontologie verwendet werden. So beschreibt der *OBSE*-Prozess nicht nur, wie eine Ontologie entwickelt wird, sondern sorgt mit der Import-Brücke dafür, dass schon in einem frühen Projekt-Stadium Begriffe einer Domäne den Projekten zur Verfügung gestellt werden.

Dass die Exporte aus den mehreren in das *OBSE*-Prozess involvierten Projekten als eine Ontologie und nicht nur als ein integriertes konzeptuelles Modell bezeichnet werden, dafür spricht einerseits ihre Funktion als gemeinsames Vokabular und andererseits der Verzicht auf die Übernahme

[52] Siehe auch Szenarien in dem Kapitel 3.1.

projektspezifischer Elemente. Dafür sorgt in erster Linie, dass die Sprache der Domänen-Ontologie dem reduzierten *CPL*-Sprachumfang der Transformationen entspricht (siehe Kapitel 4.2.3). So werden auf die Anforderungsanalyse bezogene Elemente wie QuantityDescription und AdditionalCardinality sowie Synonyme, welche die Herstellung der minimalen ontologischen Verpflichtung stören, ausgefiltert[53]. Zusätzlich dazu erlaubt die Export-Brücke, gezielt projektspezifische Modell-Elemente vom Export auszunehmen.

Eine auf diese Weise entstandene Domänen-Ontologie ist in Abbildung 4.17 aus der Vogelsperspektive gezeigt. Sie ist durch die zeitversetzte Anwendung der Brücken aus drei Projekten entstanden, die an zwei unterschiedlichen Fachbereichen (Mathematik und Informatik und Erziehungswissenschaften) der Philipps-Universität Marburg zur Unterstützung unterschiedlicher Abläufe (Schein- und Raumverwaltung, Organisation der fortgeschrittenen Praktika und Sprechstundenverwaltung) eingesetzt wurden[54]. Bereits bei so wenigen, unterschiedlichen Projekten enthält diese Domänen-Ontologie Konzepte wie Termin, E-Mail oder Student, die in allen drei Projekten verwendet wurden. Auch die Konzepte des Beispiels aus dem Kapitel 4.2 sind in die Domänen-Ontologie eingeflossen. Sie sind zur Veranschaulichung rot markiert.

Eine Strukturierung der Domänen-Ontologie kann auf zwei Ebenen erfolgen: inhaltlich und technisch. Die inhaltliche Strukturierung bezieht sich auf die Konzepte der Domäne und ihre Zugehörigkeit zueinander. Der erste Ansatz mit Hilfe einer zunächst zweidimensionalen Struktur bestehend aus Gegenstands(unter)bereichen und Interessengruppen[55], die als eine Matrix über den gesamten Untersuchungsbereich gelegt wurde [And08], hat sich bei späteren Analyse und Anwendung als zu starr erwiesen und wurde verworfen. Eines der wichtigsten Argumente dafür war der Umstand, dass die für diese Achsen verwendeten Begriffe, wie zum Beispiel „Fachbereich“ (als Gegenstandsbereich), selbst Konzepte in der Domänen-Ontologie sind und somit keine zusätzliche strukturelle Ebene darstellen. Solche Situationen können und sollen mit Hilfe der Ding- und Zusammenhangstypen ausgedruckt werden.

[53] Die einzige Ausnahme stellt das Meta-Modell-Element Example dar. Es wurde aus umsetzungstechnischen Gründen aus der Sprache der Transformationen rausgenommen. Für die Beschreibung einer Domäne ist es weiterhin relevant und für den Export in die Domänen-Ontologie zugelassen.

[54] Diese Projekte werden später im Evaluationskapitel dieser Arbeit (siehe Kapitel 7) ausführlich vorgestellt und analysiert.

[55] Eine Interessengruppe steht für eine funktionale Sicht auf die Gegenstandsbereiche. Zum Beispiel Raumverwaltung oder Vorlesungsplannung an einem Fachbereich.

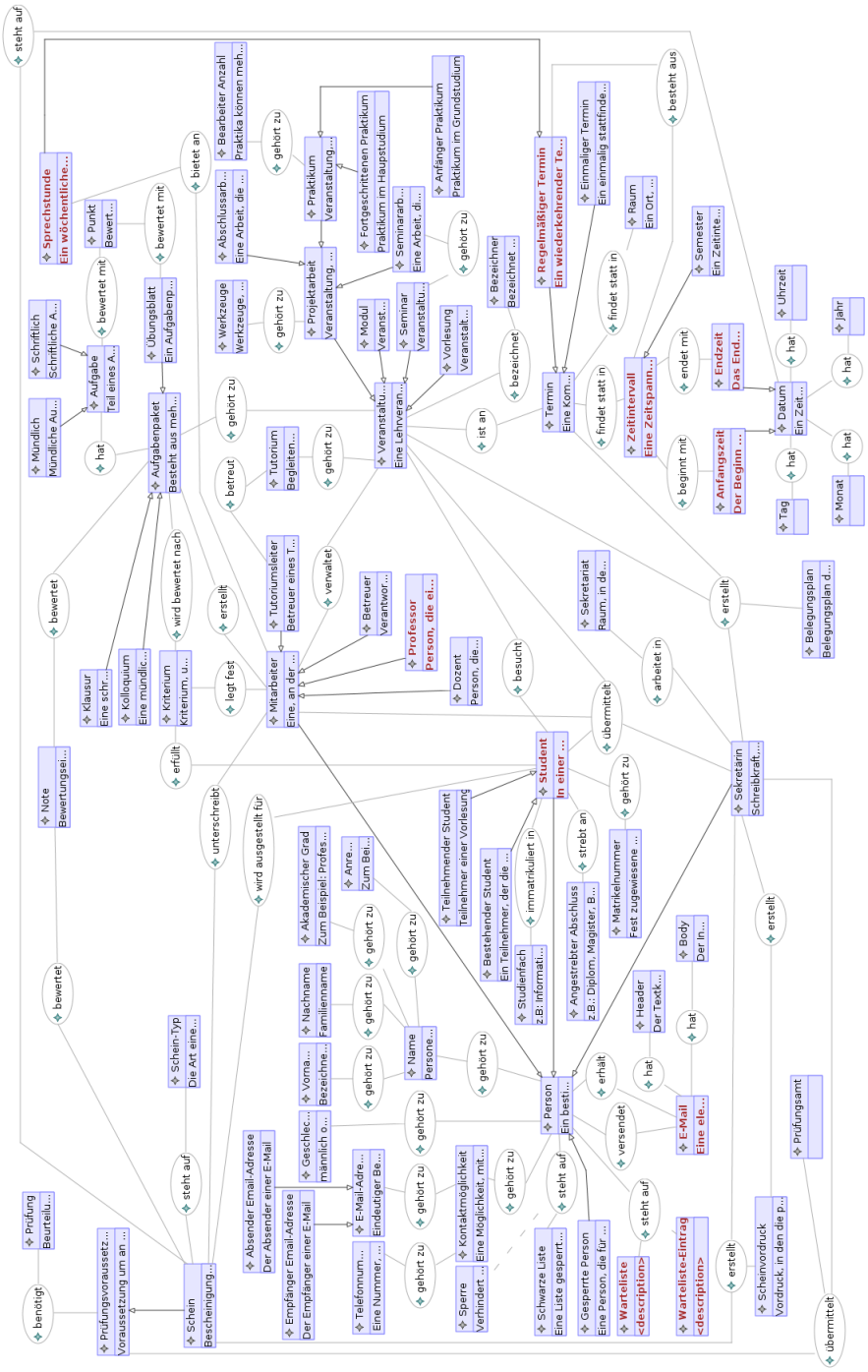


ABBILDUNG 4.17: Einblick in die Domänen-Ontologie (Universität)

Zur technischen Strukturierung, die bei der Anwendung von Brücken hilfreich ist, kann in *CPL* auf die Quellenangaben[56] zugegriffen werden. Auf diese Art wird analog zu dem sonst verbreiteten Konzept *Namespace* eine Zugehörigkeit der Modellelemente zu den einzelnen Projekten beziehungsweise der Domänen-Ontologie hergestellt.

Bezüglich der hierarchischen Klassifizierung der Ontologien nach Guarino[57] kann keine eindeutige Zuordnung zu einer Ebene angegeben werden. Vielmehr kann die Domänen-Ontolgie in dem *OBSE*-Prozess zwischen den Ebenen *Anwendungs-Ontologie* und *Domänen-Ontologie* eingeordnet werden. Der Grund dafür ist ihre dynamische Entwicklung. Geboren aus einem Projekt steht sie einer Anwendungs-Ontologie mit der spezifischen Sicht auf die Domäne sehr nahe. Mit jedem weiteren Export wird die Nutzung der Konzepte durch die Bildung von Oberklassen und Verlagerung von Beziehungen zu diesen verallgemeinert. Auf diese Art findet die Verschiebung in Richtung Domänen-Ontologie entsprechend dieser Klassifizierung statt, was auch die gleichlautende Bezeichnung in dem *OBSE*-Prozess rechtfertigt. Auf den *OBSE*-Prozess wirkt sich vorteilhaft aus, dass diese Entwicklung selbstverstärkend abläuft. Das bedeutet, je mehr Projekte an der Ontologie teilnehmen, desto allgemeiner wird sie. Gleichzeitig lässt sich eine allgemeinere Ontologie vielseitiger und in mehr Projekten wiederverwenden.

4.3.2 Import- und Export-Brücke

Für die Wiederverwendung zuständige Import- und Export-Brücken werden in *OBSE* als *Prozesse* definiert. Während sie in dem folgenden Kapitel 5 als atomare Einheiten verwendet werden, werden in diesem Abschnitt die internen Abläufe vorgestellt, die während ihrer Anwendung ausgeführt werden.

Betrachtet als Operationen, die auf *CPL*-Modelle angewendet werden, lassen sich die Import- und Export-Brücke wie folgt definieren:

[56] Meta-Klasse Origin des *CPL*-Meta-Modells (siehe Kapitel 4.1.2).
[57] Siehe Kapitel 2.2.3.

```
1 importBruecke(
2         in domaenenOntologie : CPLModell,
3         inout konzeptuellesModell :CPLModell) : void
4
5 exportBruecke(
6         in konzeptuellesModell : CPLModell) : CPLModell
```

LISTING 4.5: Operationsrümpfe der Import- und Export-Brücke

Die Darstellung im Listing 4.5 verwendet *UML*-Syntax für Operationen. CPLModell ist ein abstrakter Datentyp, der sowohl für die konzeptuellen *CPL*-Modelle der Projekte als auch für das *CPL*-Modell der Domänen-Ontologie steht. Mit in und inout wird die Richtung der Parameterverwendung festgelegt.

Die Import-Brücke hat als Parameter zwei *CPL*-Modelle: Domänen-Ontologie und konzeptuelles Modell eines Projekts. Auf die Domänen-Ontologie wird *lesend* zugegriffen (in). Ein konzeptuelles Modell wird während des Imports sowohl *gelesen* als auch *verändert* (inout), indem Elemente aus der Domänen-Ontologie in das konzeptuelle Modell integriert werden.

Die Export-Brücke benötigt nur einen Parameter konzeptuellesModell, der in dieser Operation *nicht verändert* wird in. Außerdem besitzt sie eine Rückgabe ebenfalls von Typ CPLModell, die einen Ausschnitt aus dem eingegebenen konzeptuellen Modell darstellt, der für eine Integration in die Domänen-Ontologie gedacht ist. Die interne Umsetzung dieser Brücken wird in den folgenden Abschnitten erläutert.

Import-Brücke

Vor der Anwendung der Import-Brücke unterscheiden sich nach *OBSE* laufende Projekte kaum[58] von anderen typischen Software-Projekten. Erst ihre Anbindung über die Import-Bücke an die Domänen-Ontologie eröffnet die Möglichkeit, von den bereits durchgeführten Projekten zu profitieren.

Die Voraussetzungen für die Import-Brücke sind neben der *nicht leeren* Domänen-Ontologie eine *durchgeführte* Anforderungsanalyse, die als ein konzeptuelles Modell vorliegt. In diesem Modell vorhandene Elemente der 1. Klasse dienen als Anhaltspunkte für das Abgrenzen und Darstellen der geeigneten Ausschnitte aus der Domänen-Ontologie. Somit wird die Import-

[58] *NIBA*-Tools und das konzeptuelle Modell sind die wesentlichen Unterschiede. Wobei ihre Verwendung auch in den nicht nach *OBSE* durchgeführten Projekten sinnvoll sein kann.

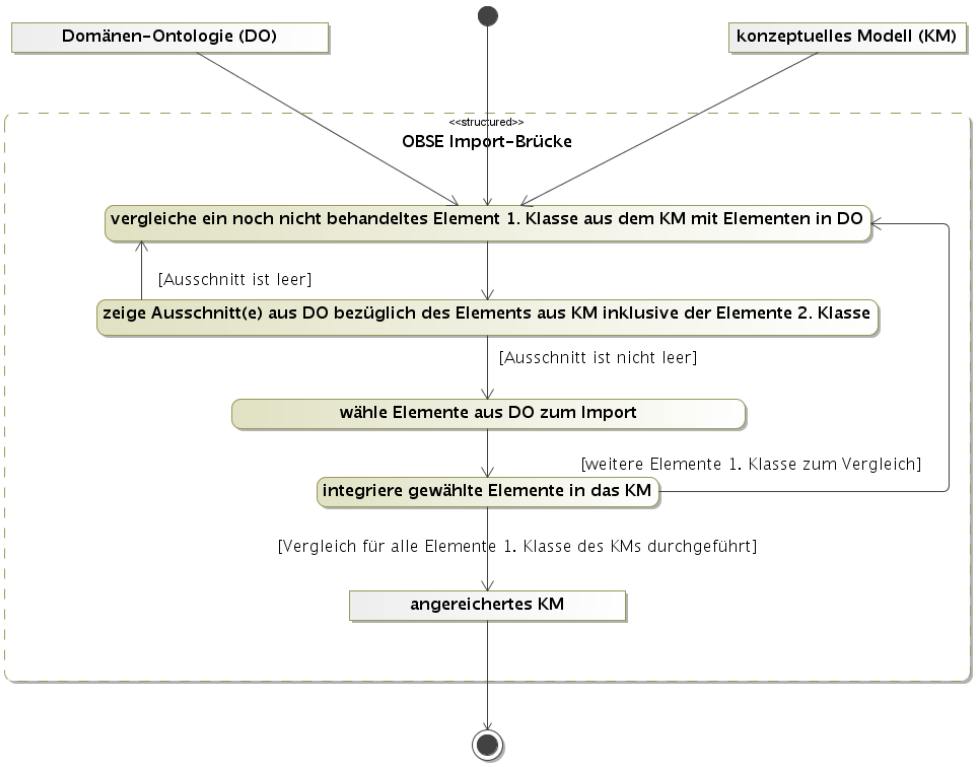


ABBILDUNG 4.18: Ablauf der Import-Brücke

Brücke sequentiell entsprechend der Anzahl dieser Elemente ausgeführt. Ihr Ablauf ist in Abbildung 4.18 dargestellt[59].

Für jedes Element 1. Klasse aus dem konzeptuellen Modell wird ein entsprechender Ausschnitt aus der Domänen-Ontologie angezeigt. Dieser Ausschnitt enthält sowohl das gesuchte Element als auch seine nähere Umgebung. Der *Radius* dieser Umgebung wird über die Tiefe der Beziehungen definiert, die verfolgt werden sollen. Bei den Dingtypen reicht es in der Regel, dass nur die direkten Zusammenhangstypen und die Dingtypen, zu denen sie führen, angezeigt werden. Anhand dieser Informationen kann ein Entwickler seine Modellierung des Gegenstandsbereichs mit der entsprechenden Modellierung in der Domänen-Ontologie vergleichen und falls erforderlich einzelne Elemente oder den gesamten Ausschnitt in sein Modell übernehmen. Da *CPL* eine Quellenangabe für alle Elemente 1. Klasse unterstützt, wird den zu importierenden Elementen zusätzlich ein Origin-

[59] Siehe auch [BHR07a].

Element vom Typ OntologySource zugewiesen. Für die Integration der gewählten Elemente ist in *OBSE* das Integrationsverfahren von Bellström und Vöhringer vorgesehen[60]. Damit kann die Ausführung der letzten Aktivität der Import-Brücke zum Teil[61] automatisieren. Mit Hilfe der Algorithmen zur Ähnlichkeitssuche in Modellen läßt sich die erste Artivität der Brücke automatisieren.

Vorteile der sequentiellen Abarbeitung der Elemente liegen darin, dass die Übersichtlichkeit bei dem Vergleich der Modellierungen auf der Ontologie- und Projektseite zunimmt und die Komplexität der Integration – analog zum Prinzip „Teile und Herrsche“ – verringert wird. Auf diese Weise können auch sehr große Domänen-Ontologien als Quelle benutzt werden, ohne dass dadurch der Aufwand bei der Ausführung der Import-Brücke zunimmt. Somit ist er hauptsächlich an die Größe des Projekts (in der Form seiner Anforderungsmodellierung) gebunden.

Export-Brücke

Die Export-Brücke ermöglicht es umgekehrt, Modellelemente aus einem Projekt zur Integration in die Domänen-Ontologie zu Verfügung zu stellen. Ihre Eingabe – das konzeptuelle Modell – stellt eine projektspezifische Sicht auf die Domäne dar. Begünstigt durch die Sprache *CPL* enthält dieses Modell noch kaum Entwurfsentscheidungen (siehe auch Kapitel 4.2.5) und ist somit der entsprechenden Domänen-Ontologie[62] sehr nah. Trotzdem kann es nicht ausgeschlossen werden, dass ein Projekt sowohl projektspezifische Elemente als auch eine abweichende Modellierung der Sachverhalte verwendet. Aus diesem Grund bietet die Export-Brücke eine Möglichkeit, eine Auswahl zu treffen, welche Elemente des konzeptuellen Modells in die Domänen-Ontologie exportiert werden sollen. Ausgewählt können sowohl Elemente 1. als auch 2. Klasse, wobei die Elemente 2. Klasse nur mit ihren „*Besitzern*“ – den entsprechenden Elementen 1. Klasse – exportiert werden können. Das ist die erste Aktivität in der Abbildung 4.19.

Im vorhergehenden Kapitel 4.3.1 wurde festgelegt, welche Elemente des *CPL*-Meta-Modells als projektspezifisch angesehen werden. Die Elemen-

[60] Für weitere Informationen siehe Kapitel 3.1.2 und [BV09]

[61] Das Verfahren arbeitet semi-automatisch.

[62] Fur die Elemente des konzeptuellen Modells, deren Meta-Elemente wie QuantityDescription oder AdditionalCardinality nicht zu dem Sprachumfang der Domänen-Ontologie gehören, wird die Entscheidung, dass sie von dem Export ausgeschlossen werden, automatisch getroffen.

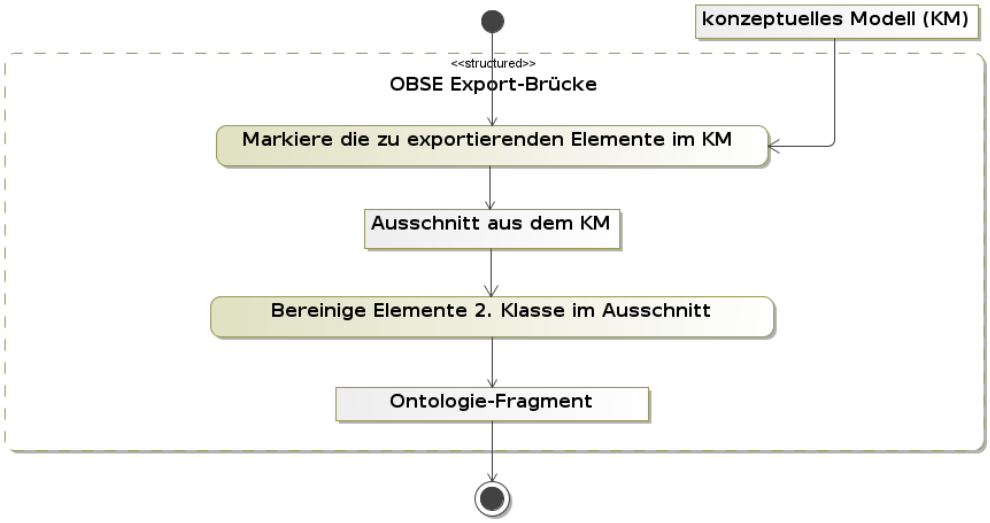


ABBILDUNG 4.19: Ablauf der Export-Brücke

te wie AdditionalCardinality können nicht in die Domänen-Ontologie übertragen werden und werden beim Export automatisch ausgefiltert. Dieser Schritt läuft voll automatisch ab. Er wird mit Hilfe einer trivialen Modell-zu-Modell-Transformation realisiert, die alle Elemente mit Ausnahme der zu ausfilternden Elemente auf sich selbst abbildet. Dabei entsteht ein gültiges Modell, da nur die voneinander unabhängigen Elemente 2. Klasse des *CPL*-Meta-Modells betroffen sind. Da der Export bedingt durch spätere Änderungen an dem konzeptullen Modell möglicherweise wiederholt ausgeführt werden muss, wird die getroffene Auswahl mit Hilfe des *CPL*-Meta-Modell-Elements ProjectSource festgehalten. Synonyme werden vom Export dadurch ausgeschlossen, dass bereits im ersten Schritt nur Repräsentanten (mainSynonyme) markiert werden können. Diese Bereinigung ist die zweite Aktivität der Export-Brücke und liefert ein so genanntes Ontologie-Fragment.

Definition 6 (Ontologie-Fragment) *Ein Ontologie-Fragment ist ein Ausschnitt des konzeptuellen Modells eines Projekts, der für die Integration in die Domänen-Ontologie vorgesehen ist.*

Ontologie-Fragment wird an dieser Stelle zusätzlich zu den Infastrukturelementen definiert, da es im *OBSE*-Prozess als ein eigenständiges Artefakt auftritt. Dieses Artefakt übernimmt die Aufgabe, Änderungsvorschläge

an die zugehörige Domänen-Ontologie zu kapseln und einem Ontologie-Entwickler zu Verfügung zu stellen. Er kann Ontologie-Fragmente sammeln, analysieren, zusammenführen, bevor Änderungen an der Domänen-Ontologie durchgeführt werden. Dass eine Export-Brücke keine Integration von Ontologie-Fragmenten in die Domänen-Ontologie vorsieht, liegt an den Zuständigkeiten im *OBSE*-Prozess, die unter Anderem im folgenden Kapitel festgelegt werden.

5 OBSE-Prozessbeschreibung

Nach der ausführlichen Vorstellung der *OBSE*-Infrastruktur-Elemente widmet sich dieses Kapitel der Beschreibung von Rollen und Prozessen der Ontologie-basierten Software-Entwicklung. Als Ausgangsbasis für ihre Definition dient das im Kapitel 2.3.2 vorgestellte evolutionäre Vorgehensmodell *EOS*[1]. Es bestimmt weitgehend, wie die Entwicklung eines Software-Projekts ablaufen soll. Um eine Anknüpfung an die Prozesse zur Ontologie-Entwicklung herzustellen, die neu im Rahmen von *OBSE* entstanden sind, wird dieses Vorgehensmodell stellenweise verändert. Diese Änderungen werden im Folgenden erläutert.

5.1 Eingliederung der Infrastruktur-Elemente

Bis auf das Analyse-Modell kommen *OBSE*-Infrastruktur-Elemente in dem *EOS*-Vorgehensmodell nicht vor. Um sie in einer Prozessbeschreibung zu verwenden, werden sie zunächst entsprechend dem *SPEM*-Meta-Modell[2] klassifiziert.

Neben dem Analyse-Modell gehören weitere Modelle der Infrastruktur – konzeptuelles Modell und die Domänen-Ontologie – zu den Artefakten (WorkProductDefinition) des *OBSE*-Prozesses. Transformationen zwischen *CPL* und *UML* und die beiden Brücken-Spezifikationen sind dagegen Aufgaben (TaskDefinition), die unter der Verwendung von Artefakten (Default_TaskDefinitionParameter) verarbeitet werden und nur von bestimmten Rollen ausgeführt werden dürfen (Default_TaskDefinitionPerformer).

Ein nach *EOS* durchgeführtes Software-Projekt läuft zyklisch in vier

[1] Für weitere Informationen über *EOS*-Modell siehe Kapitel 2.3.2 und [Sar03].
[2] Siehe Kapitel 2.3.3.

Phasen Analyse, Entwurf, Implementierung und operationaller Einsatz ab. Durch die Eingliederung in den *OBSE*-Prozess werden auf der Projekt-Seite die Phasen Analyse und operationeller Einsatz angepasst.

In der Analyse-Phase werden auf mehreren Baustein-Ebenen (System, Komponenten oder Modul) jeweils Anforderungen analysiert und modelliert. Bis auf die Bedingung, dass diese Analyse objektorientiert durchgeführt werden soll, macht *EOS* keine weiteren verbindlichen Vorgaben zu den zu verwendenden Techniken [Sar03]. Dies ändert sich in Verbindung mit dem *OBSE*-Prozess. Das Ergebnis der Anforderungsanalyse soll ein *CPL*-basiertes konzeptuelles Modell sein[3], dass anschließend in ein *UML*-basiertes Analyse-Modell überführt wird.

Für die beiden Modelle und die Transformationen zwischen diesen ist – wie auch ursprünglich für die Anforderungsanalyse im *EOS*-Prozess – die Rolle des *Systemanalytikers*[4] zuständig. Zusätzlich bestimmt ein Systemanalytiker, welche Teile des konzeptuellen Modells für eine Integration in die Domänen-Ontologie zur Verfügung gestellt werden und welche Elemente aus der Domänen-Ontologie in das konzeptuelle Modell übernommen werden. Aus diesem Grund liegt die Anwendung der Export- und Import-Brücke ebenfalls in der Zuständigkeit des Systemanalytikers. Das Diagramm in der Abbildung 5.1 stellt diese Rolle in der *SPEM*-Notation dar.

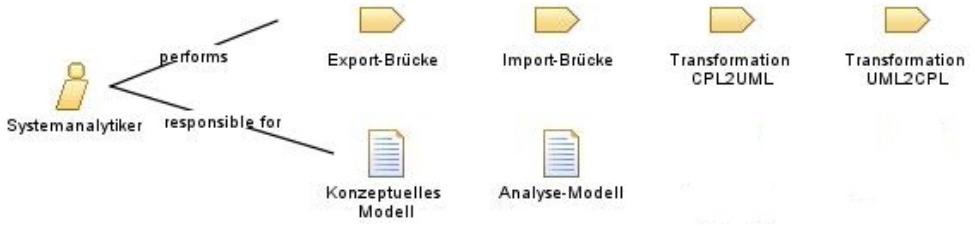


ABBILDUNG 5.1: Zuständigkeiten und Aufgaben der erweiterten Systemanalytiker-Rolle

Die Entwicklung der Domänen-Ontologie findet außerhalb der Projekte statt. Aus diesem Grund sind die Brücken so konzipiert, dass die Zuständigkeiten zwischen den Verantwortlichen in den Projekten und auf der Ontologie-Seite klar geregelt sind. Diese Trennung basiert darauf, dass die

[3]Dieses kann mit Hilfe der *NIBA*-Werkzeuge erstellt werden.

[4]Bis jetzt wurde in dieser Arbeit zwischen einzelnen Rollen auf der Projekt-Seite nicht unterschieden und die Systemanalytiker wurden verallgemeinert als Software-Entwickler bezeichnet.

Brücken, die zu den Aufgaben eines Projekts gehören und unter Zuständigkeit eines Systemanalytikers stehen, die Domänen-Ontologie zwar verwenden, sie aber nicht direkt verändern dürfen. Die Import-Brücke hat per se nur Lese-Zugriff, die Export-Brücke initiiert lediglich Veränderungen an der Domänen-Ontollogie durch eine Auslieferung von Ontologie-Fragmenten. Sie stellen einen Änderungsvorschlag dar. Für den Umfang seiner Einarbeitung in die Domänen-Ontologie sind allerdings die Verantwortlichen auf der Ontologie-Seite zuständig.

5.2 Ontologie-Entwicklungsprozess

Die Entwicklung einer Domänen-Ontologie ist eine komplexe Aufgabe. Sie verläuft in *OBSE* parallel zur Software-Entwicklung in Projekten und ist langzeit-orientiert. Analog zur Software-Entwicklung kann Ontologie-Entwicklung ebenfalls in verschiedene Phasen unterteilt werden, die mehrere Aufgaben zusammenfassen. Aus der Vogelperspektive betrachtet sind die typischen Aufgaben, die bei der Entwicklung einer Domänen-Ontologie bewältigt werden müssen, die Analyse der Domäne und die Identifikation der zu modellierenden Konzepte, Nachbildung dieser Konzepte und ihrer Beziehungen in der Ontologie, Veröffentlichung und Nutzung der Domänen-Ontologie [CRP06]. Entsprechend dem *SPEM*-Meta-Modell sind diese Aufgaben den Rollen zuzuordnen, die für sie zuständig sind. Phasen, Aufgaben und Rollen des Ontologie-Entwicklungsprozesses werden in diesem Kapitel vorgestellt.

5.2.1 Rollen

Im Unterschied zu den Software-Entwicklungsmodellen ist in der Literatur noch keine allgemein anerkannte Festlegung zu finden, welche Rollen normalerweise an der Ontologie-Entwicklung beteiligt sind. Das liegt hauptsächlich daran, dass die Einsatzgebiete einer Ontologie und somit Anforderungen an ihre Herstellung sehr unterschiedlich sind[5]. Ausgehend von den zentralen Aufgaben der Ontologie-Entwicklung kommen folgende Rollen häufig vor ([VPST05], [CRP06]):

Wissensingenieur [6] Personen, die diese Rolle übernehmen, sind für die Analyse der Domäne zuständig. Sie identifizieren Konzepte und Be-

[5] Siehe Kapitel 3.1

[6] Original: Knowledge Engineer

ziehungen in dieser Domäne und geben diese zur Modellierung in einer Ontologie-Sprache an einen Ontologie-Entwickler weiter.

Ontologie-Entwickler Ein Ontologie-Entwickler baut aus den Vorschlägen des Wissensingenieurs eine Domänen-Ontologie und verwaltet diese. Außerdem bietet er in der Regel Hilfe bei der Verwendung der Domänen-Ontologie durch die Benutzer an.

Domänen-Experte Ein Domänen-Experte unterstützt den Wissensingenieur bei der Analyse der Domäne und ist im Vergleich zu diesen Experten aus dem entsprechenden Fachbereich. Zusätzlich gehört zu den Aufgaben eines Onotologie-Entwicklers eine Kontrollfunktion über Änderungen an der Domänen-Ontologie und die damit verbundene Freigabe von neuen Versionen der Domänen-Ontologie.

Die *OBSE*-Projekte tragen durch die Verwendung von Export-Brücken maßgeblich zu der Entwicklung der Domänen-Ontologie bei. Die Analyse der Domäne und Identifikation der zu modellierenden Konzepte wird im *OBSE*-Prozess weitgehend an die Projekte ausgelagert. Bezogen auf die Rollen hat diese Verlagerung zur Folge, dass die Rolle eines Wissensingenieurs in dem Ontologie-Entwicklungsprozess nicht mehr direkt, sondern „virtuell“ durch eine Vereinigung der beteiligten Systemanalytiker ausgeübt wird.

Die Rollen des Ontologie-Entwicklers und Domänen-Experten sollten dagegen durch reale Personen besetzt werden. Ein Ontologie-Entwickler muss im Umgang mit Wissensrepräsentationssprachen vertraut sein. Ein Domänen-Experte soll mit der Domäne vertraut sein und eine *verbindliche* Auskunft über zugehörige Konzepte und Beziehungen geben können. Die mit diesen Qualifikationen (siehe Qualifications im *SPEM* Meta-Modell) verbundenen Aufgaben werden Phasen zugeordnet und mit diesen im folgenden beschrieben.

5.2.2 Phasen

Bedingt durch ihre historische Entwicklung wird unter einer Ontologie oft eine *statische* Abbildung einer Domäne verstanden, die vor ihrer ersten Anwendung erstellt wird. Szenarien wie *Ontologie zur Beschreibung der Domäne Software-Technik* aber auch die meisten Ansätze aus dem Szenario *Ontologie als Artefakt* gehen von dieser Sicht auf eine Ontologie aus (zum Beispiel [FG02], [BMN+07], [OVRT06] [FS02] usw.). Dem entsprechend legen diese Verfahren ihren Schwerpunkt auf die Ontologie-Erstellung und

vernachlässigen Prozesse, um die entstandene Ontologie später zu aktuallisieren und zu revidieren. Was für Domänen, die wenigen Veränderungen unterworfen sind, ein gewöhnlicher Weg bei der Entwicklung einer Domänen-Ontologie ist, ist für die meisten Domänen der Software-Entwicklung nicht mehr praktikabel. Diese Domänen basieren oft auf sozialen oder wirtschaftlichen Gebilden, die eher selten die dafür erforderliche Stabilität bieten und selbst vielen Änderungen unterworfen sind. Ein typisches Beispiel ist die Universität. Allein der Umstieg auf die Master- und Bachelor-Studiengänge hat diese Domäne weitgehend und nachhaltig verändert.

Aus *OBSE*-Sicht sind Ontologie-Entwicklungsprozesse zyklich und orientieren sich an einer ebenfalls zyklischen Sicht auf die Software-Entwicklung. Analog zum *EOS*-Modell[7] verläuft die Entwicklung einer Domänen-Ontologie in *OBSE* in vier Phasen: *Kollation*, *Konsolidierung*, *Revision*, *Ontologie-Einsatz*.

Wie bereits beschrieben, sind die Quellen für die Inhalte einer Domänen-Ontologie in *OBSE* die Exporte aus den Projekten. Das hat zur Folge, dass der Entstehung einer Ontologie zuerst eine Zeitspanne vorsteht, in der entsprechende Vorschläge aus den Projekten gesammelt werden. Zu Beginn des ersten Zyklus ist die Domänen-Ontologie leer, in späteren Durchläufen stellen die exportierten Ontologie-Fragmente Ergänzungen und Anpassungsvorschläge dar. Eine Zusammenfassung der Kollationsphase ist in der Tabelle 5.1 dargestellt.

Ein Übergang in die nächste Phase – Konsolidierung – kann anhand einer Kombination der Faktoren *Zeit* und der *Anzahl der eingereichten Ontologie-Fragmente* erfolgen. Über die Zeit kann gesteuert werden, dass auch bei wenigen Ontologie-Fragmenten eine Aktualisierung der Domänen-Ontologie stattfindet. Die Fragmente können wichtige Änderungen enthalten und es darf nicht allein von der Aktivität der anderen Projekte abhängen, wann sie integriert werden. Umgekehrt, wenn viele Ontologie-Framente aus den Projekten exportiert werden, ist das ein Indiz dafür, dass die aktuelle Version der Domänen-Ontologie vorzeitig aktualisiert werden muss.

Die Aktualisierung beginnt mit einer Auswahl der Ontologie-Fragmente, die in die neue Version der Domänen-Ontologie einfließen sollen. Typischerweise werden das alle eingereichte Fragmente sein. Allerdings kann es aus verschiedenen Gründen sinnvoll sein, Einreichungen zu Paketen zu gruppieren und sie zeitversetzt zu integrieren. Dafür ist dieser Schritt in der Phase Konsolidierung vorgesehen (siehe Tabelle 5.2). Anschließend gehört

[7] Siehe Kapitel 2.3.2 und [Sar03]

beteiligte Rollen	Systemanalytiker, Ontologie-Entwickler
Eingabe-Artefakte	Konzeptuelle Modelle der Projekte
Ziele	Sammeln der Veränderungsvorschläge für die Domänen-Ontologie
Aktionen und Teilschritte	1. Führe die Export-Brücke im Rahmen eines Projekts aus 2. Sammele Änderungsvorschläge von den beteiligten Projekten
Ausgabe-Artefakte	Ontologie-Fragmente
Werkzeuge	*CPL*-Editoren, Export-Brücke

TABELLE 5.1: Zusammenfassung der Kollationsphase

es zu der gemeinsammen Aufgabe des Ontologie-Entwicklers und Domänen-Experten, Konzepte und Beziehungen der Fragmente zu analysieren und zu entscheiden, welche Teile davon die Domäne richtig abbilden und daher integriert werden sollen.

Der nächste Schritt in dieser Phase gehört zu den Aufgaben des Ontologie-Entwicklers. Er soll die Ontologie-Fragmente auf die Integration in die Domänen-Ontologie vorbereiten, indem er überflüssige Konzepte und Beziehungen entfernt und andere entsprechend der Kommunikation mit dem Domänen-Experten möglicherweise überarbeitet. Das Ergebnis dieser Phase ist eine Auswahl der überarbeiteten Ontologie-Fragmente für die Integration.

Die dritte Phase *Revision* beinhaltet hauptsächlich die Durchführung der Integration der einzelnen Ontologie-Fragmente (siehe Tabelle 5.3) und ist eine Aufgabe des Ontologie-Entwicklers. Ist sie abgeschlossen, soll die neue Version der Domänen-Ontologie von Domänen-Experten abgenommen werden, bevor sie als neue Version veröffentlicht wird. Dieser Schritt kann Nacharbeiten an der Domänen-Ontologie nach sich ziehen, die von dem Ontologie-Entwickler duchzuführen sind.

Die letzte Phase eines Ontologie-Entwicklungszyklus ist ihr *operationeller Einsatz*. Sie dauert solange an, bis eine neuere Version der Domänen-Ontologie veröffentlicht wird. In dieser Zeit wird die Domänen-Ontologie

beteiligte Rollen	Ontologie-Entwickler, Domänen-Experte
Eingabe-Artefakte	Ontologie-Fragmente
Ziele	Auswahl und Anpassung der Ontologie-Fragmente für die nächste Aktualisierung der Domänen-Ontologie.
Aktionen und Teilschritte	1. Bestimme Ontologie-Fragmente, die in die neue Version der Domänen-Ontologie enfließen sollen. 2. Kontrolliere, dass keine projektspezifischen Lösungen enthalten sind. 3. Bereite Ontologie-Fragmente auf die Integration vor.
Ausgabe-Artefakte	Auswahl der Ontologie-Fragmente für die Integration
Werkzeuge	*CPL*-Editoren

Tabelle 5.2: Zusammenfassung der Konsolidierungsphase

beteiligte Rollen	Ontologie-Entwickler, Domänen-Experte
Eingabe-Artefakte	Auswahl der Ontologie-Fragmente für die Integration, Domänen-Ontologie
Ziele	Veröffentlichung einer neuer Version der Domänen-Ontologie.
Aktionen und Teilschritte	1. Integriere die ausgewählten Fragmente in die Domänen-Ontologie 2. Kontrolliere die entstandene Ontologie und gebe sie anschließend frei.
Ausgabe-Artefakte	neue Version der Domänen-Ontologie
Werkzeuge	*CPL*-Editoren, Integration auf der *CPL*-Ebene

Tabelle 5.3: Zusammenfassung der Revisionsphase

beteiligte Rollen	Systemanalytiker
Eingabe-Artefakte	Domänen-Ontologie
Ziele	Weitergabe der in der Ontologie erfassten Elemente an die Projekte.
Aktionen und Teilschritte	1. Führe die Import-Brücke aus.
Ausgabe-Artefakte	Konzeptuelles-Modell
Werkzeuge	Import-Brücke, *CPL*-Editoren

Tabelle 5.4: Zusammenfassung des Ontologie-Einsatzes

bezüglich der Änderungen stabil gehalten und entspricht weitgehend der klassischen Verwendung der Ontologie, wie in den beispielhaft erwähnten Verfahren oben. In dieser Phase wird sie von Projekten verwendet, indem diese die Import-Brücke anwenden. Dafür sind allein die Systemanalytiker zuständig (siehe Tabelle 5.4). Der operationelle Einsatz schließt den Entwicklungs-Zyklus einer Domänen-Ontologie ab.

Es ist wichtig zu beachten, dass die vorgestellten Phasen sich auf *eine Version* der Domänen-Ontologie beziehen. Die Ontologie kann sich insgesamt betrachtet – und meistens wird dies der Fall sein – gleichzeitig in mehreren Phasen befinden. So befindet sich ihre aktuelle veröffentlichte Version in der Phase operationeller Einsatz. Gleichzeitig wird der Nachfolger dieser Version entwickelt und ist zum Beispiel in der Phase Analyse. Währenddessen stellen Projekte weitere Konzepte zur Integration in die Ontologie zur Verfügung, die den Anfang für die Nach-Nachfolger-Version legen, die sich dementsprechend in der Kollation-Phase befindet.

5.2.3 Verzahnte Projekt- und Ontologie-Entwicklungszyklen

Die zyklische Entwicklung der Domänen-Ontologie und Projekte verläuft in *OBSE* unabhängig voneinander und wird lose über die Import- und Export-Brücken gekoppelt. Diese Verbindung ist schematisch in Abbildung 5.2 dargestellt.

Auf der Projekt-Seite beginnt ein neuer Entwicklungsprozess mit der Analyse-Phase. Am Anfang dieser Phase wird die Import-Brücke angewen-

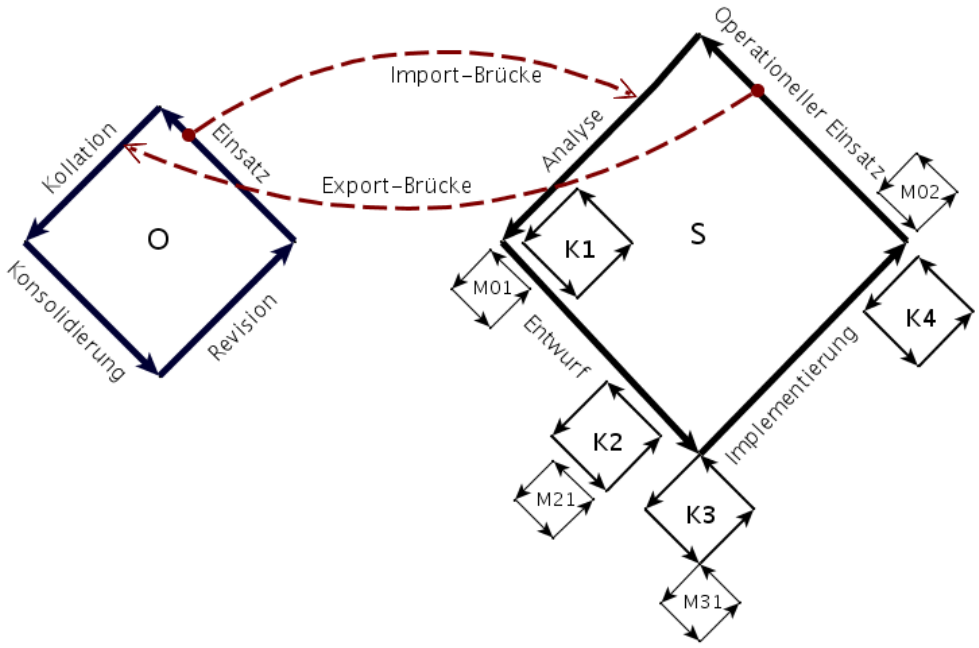


ABBILDUNG 5.2: Verknüpfung der Projekt- und Ontologie-Entwicklungszyklen

det, um die vorhandenen Informationen über die Domäne in das Projekt zu holen.

Elemente des Gegenstandsbereichs, die während des gesamten Software-Entwicklungszyklus erarbeitet wurden, bieten sich an, am Ende der operationellen Einsatz-Phase[8] als ein Ontologie-Fragment exportiert zu werden, da diese Elemente an dieser Stelle im Prozess nicht mehr verändert werden.

Für den Import ist es erforderlich, dass eine Version der Domänen-Ontologie vorhanden ist, die sich in der Phase Ontologie-Einsatz befindet. Das ist – aus der Prozesssicht gesehen – die Andockstelle für die Import-Brücke. Da die Domänen-Ontologie während der Zeit des Ontologie-Einsatzes stabil gehalten wird, spielt die weitere zeitliche Einordnung der Andockstelle keine Rolle.

Die Export-Brücke ist mit der Kollationsphase auf der Ontologie-Seite verknüpft. Durch ihre Anwendung wird diese Phase gestartet, falls noch kei-

[8]Da die Brücke auf der *CPL*-Ebene arbeitet, hilft an dieser Stelle die UML→CPL-Transformation, falls während der Software-Entwicklung zu Inkonsistenzen zwichen dem konzepuellen und Entwurfs-Modellen gekommen ist.

ne Ontologie-Fragmente vorliegen. Das ist immer bei einer neuen Domänen-Ontologie oder beim Übergang in die Analyse-Phase der Ontologie-Entwicklung der Fall, bei dem alle eingereichte Fragmente übernommen werden. Ansonsten ergänzt jeder Export aus einem Projekt die vorhandene Menge der Änderungsvorschläge und trägt zu dem Fortschritt dieser Phase bei[9].

5.2.4 Anpassungen für die agile Software-Entwicklung

Aus dem Manifest zur agilen Software-Entwicklung [BBvB+01], das die Verschiebung der Schwerpunkte bei der Entwicklung von Software im Vergleich zu den klassischen Methoden postuliert, lassen sich mehrere Prinzipien lesen. Dazu zählen vor allem Einfachheit, Kundennähe, Zweckmäßigkeit und gemeinsamer Code-Besitz, worauf die gewählte Vorgehensweise ausgerichtet werden soll. Die darauf aufbauenden Vorgehensmodelle wie *Scrum*[10] werden in vielen Projekten erfolgreich eingesetzt und haben sich inzwischen etabliert [Pic07].

Die Erfahrung und die Kritik[11] an dem bekannten und umfangreichen Vorgehensmodell *RUP* zeigen, dass kein Entwicklungsprozess optimal in allen Software-Projekten angewendet werden kann und die Wahl des Vorgehensmodells sich vorrangig an den Projektvorgaben orientieren soll. Für den *OBSE*-Prozess, der auf mehreren unterschiedlichen Projekten einer Domäne aufbaut, hat das die Konsequenz, dass es – realistisch betrachtet – zu einer Mischung aus klassisch und agil durchgeführten Projekten kommen kann.

Ihr Zusammenspiel stellt für das *OBSE*-Verfahren kein Problem dar, da sich die Prinzipien der agilen Software-Entwicklung in erster Linie auf die Projekte auswirken. Dort muss auf Grund des gemeinsamen Code-Besitzes vor allem die Zuständigkeit für die beiden Brücken neu geregelt werden. Dieses Prinzip betont einerseits die Ausrichtung der agilen Entwicklungsmethoden auf ein möglichst frühzeitig lauffähiges System, gilt aber andererseits genau so für alle weitere Artefakte eines Projekts, insbesondere wenn diese zur Erstellung von Code verwendet werden. In einem agil durchgeführten Projekt erstreckt sich somit der gemeinsame Besitz auch auf das konzeptuelle Modell des Projekts.

Übertragen auf das Vorgehensmodell *Scrum*, in dem nur drei Rollen

[9] Durch die Anzahl der eingereichten Ontologie-Fragmente. Siehe Kapitel 5.2.2.
[10] Siehe auch Kapitel 2.3.1.
[11] Siehe zum Beiispiel [Hes01].

Product Owner, *Team*, und *Scrum Master* vorgesehen sind, hat eine Integration in den *OBSE*-Prozess zur Folge, dass die *Product Owner* die Pflege des konzeptuellen Modells übernimmt und somit für die Brücken zuständig ist. So wird die Import-Brücke am Anfang eines Sprints gebraucht. In diesem Zeitabschnitt beschäftigen sich die Software-Entwickler mit der Analyse der ihnen zugewiesenen Backlog Items, was durch die Informationen aus der Domänen-Ontologie – analog zu Anforderungsanalyse – unterstützt werden kann. Die Anwendung der Export-Brücke bietet sich am Ende eines Sprints an, nachdem die entstandenen Inkremente dem Auftraggeber präsentiert wurden. Als eine Begutachtung und Abnahme ist diese Präsentation für die Stabilität der entstandenen Lösung wichtig und ist eine Voraussetzung für die Export-Brücke.

5.2.5 Vergleich mit den existierenden Methodologien

Für den gesamten *OBSE*-Prozess (verschränkte Ontologie- und Software-Entwicklung) finden sich in der Forschung kaum vergleichbare Ansätze (siehe auch Kapitel 3.2). Dagegen existieren für die reine Ontologie-Entwicklung bereits einige Methodologien, die an dieser Stelle betrachtet und mit der in diesem Kapitel vorgestellten Vorgehensweise verglichen werden.

Einen Rahmen für die Vergleiche von Software-Entwicklungsprozessen liefert der *IEEE 1074*-Standard, indem er ein Grundgerüst eines Prozesses für den Software-Lebenszyklus vorgibt. Bei einem Vergleich werden Prozesse der jeweiligen Vorgehensmodelle entsprechend dem Standard eingeordnet und dreistufig mit *unterstützt*, *teilweise unterstützt* und *nicht unterstützt* bewertet. Da die Prozesse zur Ontologie-Entwicklung viele Gemeinsamkeiten zu denen der Software-Entwicklung aufweisen, leiteten die Autoren Fernández-López und Gómez-Pérez von diesem Standard ein entsprechendes Grundgerüst für den Ontologie-Entwicklungsprozess ab, das sich analog zu dem *IEEE 1074*-Standard zu Vergleichen heranziehen lässt [FLGP02].

Der Hauptprozess der Ontologie-Entwicklung wird danach in drei Abschnitte unterteilt: *Vorbereitung*, *Entwicklung* und *Einsatz*. Diese bestehen wiederum aus Unterabschnitten, die die typischen Aufgaben des Entwicklungsprozesses wie *Entwurf*, *Nutzung* usw. beinhalten[12].

Bekannte Methodologien für die Entwicklung einer Ontologie sind vor allem *METHONTOLOGY* und *On-To-Knowledge (OTK)*. *METHONTOLOGY* ist eine umfassend beschriebene Methodologie, die ihren Schwer-

[12]Siehe die beiden linken Spalten in der Tabelle 5.5.

punkt auf die Wiederverwendung von bereits existierenden Ontologien setzt und nicht auf die Entwicklung einer Domänen-Ontologie von Grund auf. *METHONTOLOGY* bietet Mechanismen zur Umstrukturierung der existierenden Ontologien und ihre Integration, Evaluation usw. Die wichtigste Phase dieser Methodologie ist die Konzeptualisierungsphase, in der ausgehend von den existierenden Ontologien eine neue kreiert wird [CRP06].

On-To-Knowledge hilft beim Erstellen einer Ontologie, die auf die Aufgaben der späteren Wissensmanagement-Werkzeuge ausgerichtet ist. Die Anforderungen an die Ontologie bei dieser Vorgehensweise werden ähnlich zu der Anwendungsfall-Analyse der Software-Technik mit Hilfe von Nutzungs-Szenarien gesammelt. Anschließend werden diese Szenarien analysiert und daraus Konzepte und Beziehungen für die Ontologie abgeleitet. *OTK* sieht die Verwendung von existierenden Ontologien vor. Als Ergebnis wird eine anwendungsspezifische Ontologie erstellt [CRP06].

Weitere Methodologien – ausgewählt wegen ihrer Nähe zu dem Ontologie-Entwicklungsprozess in *OBSE* – sind *KACTUS*, *DILIGENT* und *UPON*. Charakteristisch für *KACTUS* ist der *bottom-up* Aufbau der Domänen-Ontologie, bei dem analog zu *OBSE* die Projekte einer Domäne als Lieferanten für das Wissen über die Domäne dienen [CRP06]. *DILIGENT* ist eine durch die Initiative des Semantischen Webs inspirierte Methodologie, die ihren Schwerpunkt auf das verteilte Arbeiten an einer Ontologie legt [VPST05]. *UPON* überträgt etablierte Techniken und Methoden aus der Software-Entwicklung auf die Ontologie-Entwicklung und passt sie dem neuen Aufgabenbereich an [NMN09].

Tabelle 5.5 zeigt einen Vergleich der vorgestellten Ontologie-Entwicklungsprozesse entsprechend dem *IEEE 1074*-Standard angewendet auf die Ontologien. Die Werte in dieser Tabelle wurden bis auf den *OBSE*-Ontologie-Entwicklungsprozess aus den Veröffentlichungen [CFLGP03], [VPST05] und [NMN09] übernommen.

Die Bewertung für *OBSE* wurde aus den folgenden Überlegungen abgeleitet. Der Ontologie-Entwicklungsprozess bietet keine Schritte für eine Vorabuntersuchung einer Domäne und für die Machbarkeitsstudie an. Die Aufgaben des ersten Unterprozesses wurden somit mit – belegt. Die Entwicklung einer Domänen-Ontologie nach *OBSE* wurde dagegen sowohl bezüglich der notwendigen Infrastruktur als auch der Prozesse in dieser Arbeit detailliert beschrieben. Angefangen beim Sammeln der „Anforderungen“ als Ontologie-Fragmente[13] in der Kollationsphase bis hin zur Implementierung

[13] Ontologie-Fragmente sind keine Anforderungen, ersetzen diese aber in gewissem Sinne.

Unterprozesse	Aufgaben	METH-ONTO-LOGY	OTK	KACTUS	DILIGENT	UPON	OBSE
Vorbereitung	Umgebungsstudie	–	–	–	–	P	–
	Machbarkeitsstudie	–	+	–	+	–	–
Entwicklung	Anforderungen	+	+	+	P	+	+
	Entwurf	+	P	P	+	+	+
	Implementierung	+	+	+	P	+	+
Einsatz	Installation	–	–	–	+	–	+
	Nutzung	–	–	–	P	–	+
	Support	–	–	–	+	–	–
	Pflege	P	P	–	+	P	+
	Stilllegung	–	–	–	+	–	P

TABELLE 5.5: Vergleich der Ontologie-Entwicklungsprozesse

(Legende: + steht für unterstützt, – für nicht unterstützt und P für die partielle Unterstützung)

der neuen Version in der Revisionsphase ist die Entwicklung der Domänen-Ontologie in *OBSE* vorgegeben. Zusätzlich beschreibt der *OBSE*-Ontologie-Entwicklungsprozess, wie die entstandene Ontologie mit Hilfe der Import-Brücke benutzt wird[14].

Die Aufgabe Pflege ist mit + belegt, da *OBSE* durch die zykliche Ontologie-Entwicklung dafür sorgt, dass Veränderungsvorschläge laufend aufgenommen und die bestehende Ontologie damit weiterentwickelt wird, was für die kontinuierliche Pflege der Domänen-Ontologie sorgt. Stilllegung tritt zwangsläufig ein, wenn eine neue Version veröffentlicht wird. Diese Aufgabe wurde aber bis jetzt nicht intensiv betrachtet und wird aus diesem Grund mit P belegt.

Alle Methodologien bieten eine weitgehende Unterstützung für den Unterprozess Entwicklung. Nur *DILIGENT* beschreibt nicht genau, wie eine Ontologie in diesem Prozess implementiert wird. Diese Methodologie legt ihren Schwerpunkt auf die Prozesse der kolloborativen Ontologie-Entwicklung ohne die Vorgabe einer Ontologie-Sprache, da sich diese Methodologie haupsächlich auf die heterogene Entwicklergemeinde des *Semantischen Webs* richtet.

Die Aufgaben des Unterprozesses Einsatz zeigen deutlich, dass die meisten existierenden Prozesse auf die einmalige Erstellung einer Ontologie ausgerichtet sind und darüber hinaus keine weitere Unterstützung anbieten. Nur *DILIGENT* und *OBSE* gehen in diesem Bereich einen Schritt weiter und beschreiben die entsprechenden Prozesse, allerdings für zwei unter-

[14]Installation wird durch ständigen (Weiter-)Aufbau ersetzt.

schiedliche Zielgruppen. Während *DILIGENT* Entwickler im *Semantischen Web* anspricht und ihnen eine Moglichkeit anbietet, ihre Anstrengungen zu koordinieren, richtet sich der *OBSE*-Ontologie-Entwicklungsprozess an Software-Entwickler und berücksichtigt analog zu *UPON* deren Bedürfnisse und ihre Erfahrung.

Darüber hinaus bieten die meisten Methodologien zur Ontologie-Entwicklung keine Werkzeuge an, die bei ihrer Durchführung eine technische Unterstützung leisten würden [CRP06]. Das trifft in der obigen Auswahl auf *KACTUS*, *DILIGENT* und *UPON* zu. Somit existiert bis jetzt kein Ontologie-Entwicklungsprozess, der nicht nur die Entwicklung, sondern auch deren Einsatz beschreibt und die beiden Teilprozesse durch ein Werkzeug unterstützt. Um diese Lücke zu schließen und das Arbeiten mit *OBSE* praktisch auszuprobieren, wurde im Rahmen dieser Arbeit das *OBSE-Tool* entwickelt.

6 Ein Werkzeug zur OBSE-Prozessunterstützung

Eine umfassende Werkzeugunterstützung ist eine unerlässliche Voraussetzung für den *OBSE*-Prozess, seine Akzeptanz und Verbreitung. Das hier beschriebene *OBSE*-Tool wurde parallel zu der Prozessbeschreibung entwickelt und ist noch nicht vollständig. Auch in seinen Vorversionen war es sowohl beim Konzipieren als auch beim Evaluieren des *OBSE*-Prozesses hilfreich.

6.1 Konzeption

Das primäre Ziel bei der Entwicklung des *OBSE-Tools* war, verschiedene Werkzeuge (Editoren, Transformatoren, Wizards, usw.) für das Arbeiten mit den Elementen der *OBSE*-Infrastruktur zur Verfügung zu stellen. Da diese Elemente alle miteinander verknüpft sind, sollen die einzelnen Werkzeuge unter einer gemeinsamen Oberfläche angeboten werden, um ganzheitliche Arbeitsabläufe zu unterstützen. Sie sollen folgende zentrale Anwendungsfälle abdecken: Anlegen und Verwalten von Ontologie- und Software-Entwicklungs-Projekten, Anlegen und Bearbeiten (grafisch und tabellarisch) von *CPL*-Modellen für eine Domänen-Ontologie und konzeptuelle Modelle der Projekte, Anlegen und Bearbeiten von *UML*-Modellen, Ausführung von Transformationen zwischen *CPL* und *UML*, Ablegen dieser Modelle in einer relationalen Datenbank, Ausführung von Brücken.

Eine IST-Analyse bereits vorhandener Werkzeuge ergab, dass es abgesehen von der *UML*, zu der bereits eine breite Palette von Werkzeugen existiert, an der Unterstützung für die sonstigen Elemente der *OBSE*-Infrastruktur mangelte. Lediglich für die ursprüngliche Version von *CPL* existierten mehrere Datenbank-basierte Prototypen, die auf verschiedenen

Plattformen (*Web-basiert* in *PHP* und *Stand-alone* in *Java*) aufbauten, was ihre Integration und Wiederverwendung im *OBSE*-Tool erschwerte bzw unmöglich machte.

Eine Besonderheit, die die Wahl der passenden Architektur beeinflusste, war die anvisierte Vorgehensweise bei der Werkzeug-Entwicklung. Als eine komplexe Anwendung, die in Kooperation mit dem Institut für die Angewandte Informatik an der Alpen-Adria-Universität Klagenfurt entstehen sollte, verlangte die Entwicklung des *OBSE*-Tools eine dezentralisierte Vorgehensweise, bei der Werkzeuge für die einzelnen Elemente der Infrastruktur getrennt voneinander entwickelt werden können. Trotzdem sollte die gewählte Architektur ihre nachträgliche Integration unter einer gemeinsamen Oberfläche – Lego-Baustein-artig – ermöglichen.

In den letzten Jahren sind mehrere Plattformen entstanden, die derartige Vorgehensweise ermöglichen. Die bekanntesten, nicht-kommerziellen Vertreter sind *NetBeans*[1] und *Eclipse*[2]. Wegen der größeren Verbreitung und besonderes der sehr fortgeschrittenen Unterstützung der Meta-Modell-basierten Entwicklung von Editoren fiel die Wahl auf *Eclipse*. Entscheidend war dafür, dass *Eclipse* im Unterschied zu *NetBeans* auf einer Implementierung[3] der speziell für die *Service*-basierten Anwendungen[4] entwickelten *OSGi*-Standards (*Open Services Gateway initiative*)[5] basiert.

OSGi kann als eine Schicht angesehen werden, die auf der virtuellen Maschine von *Java* aufsetzt und die Zusammenarbeit von *PlugIns* steuert. Diese Komponenten-basierte Plattform erlaubt es, einzelne *PlugIns* zur Laufzeit dynamisch zu installieren, zu starten und ihre Ausführung zu beenden. Außerdem bringt sie eine ausgereifte Versions- und Abhängigkeitsverwaltung mit, die es ermöglicht, für die Ausführung eines *PlugIns* erforderliche andere *PlugIns* ebenfalls automatisch in der richtigen Version zu laden oder zu entladen. Eine auf *OSGi*-basierende Anwendung bringt in der Regel diese Laufzeit-Schicht und eine Reihe von den miteinander verknüpften *PlugIns* mit.

OSGi-Implementierungen können vielseitig, zum Beispiel in modu-

[1] http://netbeans.org/

[2] http://www.eclipse.org/

[3] Das Gründgerüst von *NetBeans* ist ähnlich zu dem *OSGi*-Standard aufgebaut und wird in den zukünftigen Versionen dem Standard angeglichen, um Kompatibilität zu den anderen *OSGi*-basierten Plattformen wie *Eclipse* zu erhöhen.

[4] In diesem Fall werden die Bausteine des Systems, die ihre Dienste (*Services*) anderen Bausteinen des Systems anbieten, als *Bundle* oder *PlugIn* bezeichnet.

[5] www.osgi.org

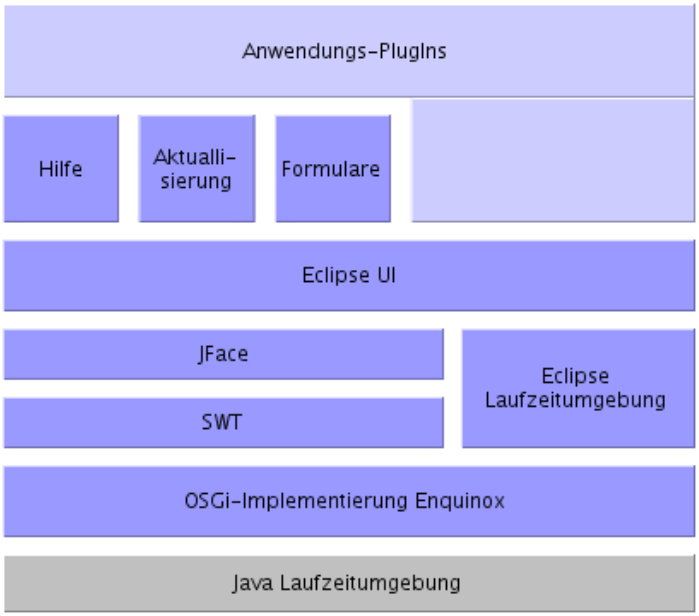


ABBILDUNG 6.1: *Rich Client Plattform* von *Eclipse* [Elb10]

lar aufgebauten eingebetteten Systemen, vernetzten Infrastrukturen oder *Desktop*-Anwendungen eingesetzt werden. Mit der *Rich Client Plattform (RCP)*[6] zielt *Eclipse* auf die Entwicklung von den Letzteren und bringt zusätzlich zu der *OSGi*-Implementierung mit dem Namen *Equinox* viele weitere Komponenten mit, die eine Entwicklung von modularen *Desktop*-Anwendungen erleichtern und beschleunigen. Sie sind in der Abbildung 6.1 zwischen der *Java-Laufzeitumgebung* und *Anwendungs-PlugIns* dargestellt.

So bietet *Eclipse* für den Aufbau der graphischen Benutzerschnittstelle sowohl eine Bibliothek von plattformunabhängigen Gestaltungselementen *Standard Widget Toolkit (SWT)* als auch eine Komponente *JFace*, die es ermöglicht, *SWT*-Elemente mit Daten aus den zugehörigen *Java*-Objekten zu befüllen. Zusammen mit *Eclipse Laufzeitumgebung*, die die Verwaltung des Lebenszyklus einer *Service*-basierten Anwendung übernimmt, bilden *SWT* und *JFace* eine Grundlage jeder *Eclipse*-basierten *RCP*-Anwendung. Eine Vorform davon ist die *Eclipse UI*, die als eine noch funktionslose, leere Oberfläche einer *RCP*-Anwendung betrachtet werden kann.

[6] Mit *Rich Client* sind *Desktop*-Anwendungen gemeint, die einige Eigenschaften der *Web*-Anwendungen wie Vernetzbarkeit, zentrale Aktualisierung und Konfigurierbarkeit mitbringen.

Die drei *PlugIns* (*Hilfe*, *Aktualisierung* und *Formulare*) in der oberen linken Ecke in der Abbildung 6.1 sind optional. Da sie aber in den meisten Anwendungen gebraucht werden, gehören sie standardmäßig zu *RCP*. Die obere Schicht der *RCP*-Plattform bilden Anwendungs-*PlugIns*, die die eigentliche Funktionalität einer *RCP*-Anwendung bereitstellen.

Folgende Vorteile ergeben sich aus der Verwendung der *Eclipse RCP*-Plattform (vgl. [Elb10]):

- Durch *Eclipse UI* und die darunter liegenden Elemente der Benutzerschnittstelle wird ein einheitliches, durchdachtes und vielen Entwicklern bekanntes Bedienkonzept zur Verfügung gestellt. Jede *RCP*-Anwendung, sei es eine *Java-Entwicklungsumgebung* oder das *OBSE*-Tool, bietet einheitliche Bedienelemente an. Das trifft insbesondere auch auf die einzelnen *PlugIns* einer Anwendung zu, unabhängig davon, wann und wo sie entwickelt wurden.
- Die meisten benötigten Elemente der Benutzerschnittstelle werden bereitgestellt, was die Erstellung einer Anwendung enorm beschleunigt.
- Die Plattform ist konsequent auf die Modularisierung und Erweiterbarkeit ausgelegt. *PlugIns* können jeder Zeit einer Anwendung hinzugefügt und aus dieser entfernt[7] werden.
- Es stehen ausgereifte und in *Eclipse* nahtlos integrierte Werkzeuge – realisiert ebenfalls als *PlugIns* – zum Erstellen von *RCP*-Anwendungen zur Verfügung.

Zusätzlich ermöglichen *Eclipse*-Frameworks wie *Eclipse Modeling Framework (EMF)* und *Graphical Modeling Framework (GMF)* eine Meta-Modell-getriebene Entwicklung von baumartigen, tabellarischen und graphischen Editoren, die ausgehend von dem Meta-Modell einer Sprache weitgehend generiert werden können. Besonderes für die Entwicklung der *CPL*-Werkzeugunterstützung spielten diese Frameworks eine entscheidende Rolle.

Im Überblick betrachtet besteht das *OBSE*-Tool aus folgenden funktionalen Komponenten[8] (siehe Abbildung 6.2):

Editoren Meta-Modell-basierte, tabellarische und graphische Editoren für die Sprachen *UML* und *CPL*.

[7] Natürlich unter der Berücksichtigung der Abhängigkeiten, die in diesen *PlugIns* festgelegt wurden.

[8] An dieser Stelle wird der Begriff Komponente und nicht *PlugIn* verwendet, da die Darstellung nicht granular genug ist und diese Komponenten aus mehreren *PlugIns* gebildet werden.

Transformatoren Werkzeuge, die die Tranformationen zwischen den Sprachen *UML* und *CPL* ermöglichen.

Export/Import-Brücken Unterstützung der Kopplung von Projekten und der zugehörigen Domänen-Ontologie (inkl. Integration).

Prozessleitfaden Beschreibung des *OBSE*-Prozesses, die Projektbeteiligte in ihre Aufgaben einführt und Verknüpfungen zu den erforderlichen Werkzeugen und Artefakten herstellt.

Prozess-Editor Dieser Editor ermöglicht es – falls erforderlich – die vorgegebene *OBSE*-Prozessbeschreibung an die Besonderheiten des jeweiligen Projektes anzupassen. Zum Beispiel, um die agile Vorgehensweise abzubilden.

Datenbankanbindung Stellt Funktionalität zur Verfügung, um *UML*- und *CPL*-Modelle statt in *XML*-Dateien in einer relationalen Datenbank abzulegen.

automatisierte Anforderungsanalyse Werkzeuge aus dem *NIBA*-Projekt zur Überführung von natürlichsprachigen Anforderungen nach CPL.

Die Abbildung 6.2 zeigt gleichzeitig, mit Hilfe welcher Frameworks bzw. Hilfsmittel[9] diese Komponenten erstellt werden. So wird die *Datenbankanbindung* unter Verwendung von *Teneo* und *Hibernate* realisiert. Dazu werden für *ECore*-basierte Meta-Modelle des *OBSE*-Tools zunächst mit *Teneo* Annotationen erstellt, die später von *Hibernate* für die objektrelationale Abbildung[10] der *Java*-Klassen eines Modells auf die entsprechenden Tabellen einer relationalen Datenbank verwendet werden.

Im Bereich der Modellierungswerkzeuge ermöglichen es die Frameworks *EMF* und *GMF*, ausgehend aus den *ECore*-basierten Meta-Modellen Editoren zu generieren, die grundlegende Elemente zum Editieren von darauf aufbauenden Modellen anbieten. Während *EMF* nur anhand eines Meta-Modells einen vollwertigen, Baum-artigen Editor generieren kann, benötigt *GMF* zum Generieren eines graphischen Editors weitere Angaben über das Aussehen der Modellierungselemente und Beziehungen. Sie werden in deklarativer Form der Generierungsvorschrift beigesteuert. Da *CPL* für eine tabellarische Darstellung konzipiert wurde, bleibt auch der automatisch mit dem *EMF*-Framework generierte Editor nicht unverändert, sondern wird um eine zusätzliche tabellarische Sicht auf die Modellierungselemente er-

[9] Angaben in Klammern. *NIBA*-Werkzeuge werden komplett an der Universität Klagenfurt konzipiert und entwickelt. Aus diesem Grund wird auf die genaue Beschreibung dieser Werkzeuge verzichtet.

[10] *Object-relational mapping (ORM)*

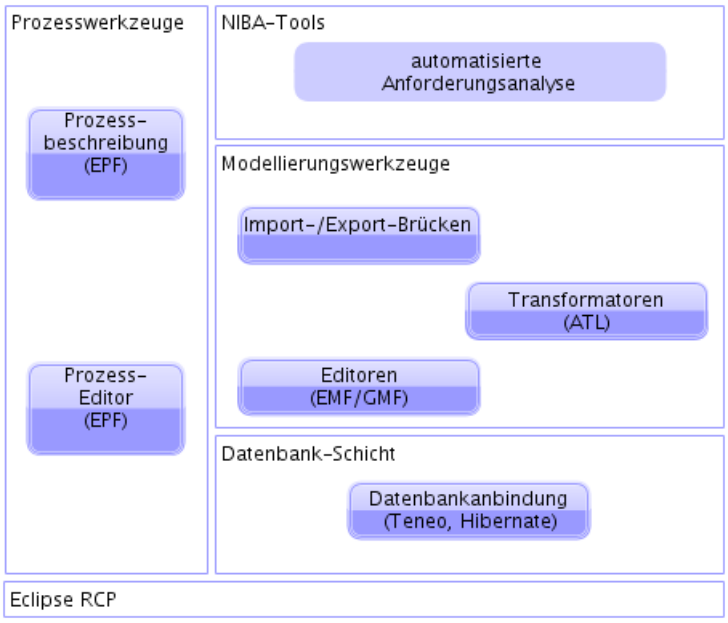


Abbildung 6.2: Überblick über das *OBSE*-Tool

gänzt. Transformationen basieren auf der Sprache *ATL*, für die bereits eine *Eclipse*-Implementierung existiert [11]. Die Brücken bauen auf dem Integrationsansatz von Bellström und Vöhringer[12] auf.

Die Komponenten im Bereich Prozesswerkzeuge werden mit Hilfe des *Eclipse Process Frameworks (EPF)* erstellt. Sein Vorläufer wurde ursprünglich von *Rational* entwickelt, um Anpassungen am *RUP* zu ermöglichen. Nach der Übernahme dieses Unternehmens durch *IBM* wurden die Möglichkeiten dieses Frameworks erweitert und für Modellierung anderer Vorgehensmodelle verallgemeinert[13]. Außerdem erfolgte seine Integration in *Eclipse*, was eine Komposition der Modellierungswerkzeuge mit der *OBSE*-Prozessbeschreibung unter einer Benutzeroberfläche ermöglichte. Der Editor von *EPF*, mit dem der *OBSE*-Prozess nach dem *SPEM*-Modell beschrieben wurde, kann gleichzeitig benutzt werden, um die mit dem *OBSE*-Tool ausgelieferte Prozessbeschreibung zu verändern.

[11] Siehe auch Kapitel 2.1.4
[12] Siehe Kapitel 3.1.2 und [BV09]
[13] Siehe Kapitel 2.3

6.2 PlugIn-Architektur

Zur Strukturierung von Objekt-orientierten Software-Systemen stehen Klassen – als kleinster Baustein – und Pakete – als darüber liegende hierarchische Struktur – zur Verfügung. Im Unterschied zu Klassen, bei denen eine Schnittstelle nach außen mit Hilfe der Operationen und ihren Sichtbarkeitseinstellungen genau definiert werden kann, bieten Pakete lediglich eine Möglichkeit, enthaltene Klassen als Paket-intern oder nach außen sichtbar zu kennzeichnen. Im Bezug auf die Strukturierung eines Software-Systems können PlugIns als eine Erweiterung des Paket-Mechanismus angesehen werden. Sie kapseln ein oder mehrere Pakete und ermöglichen es, funktionale Schnittstellen genau zu definieren und Sichbarkeiten festzulegen. Im *Eclipse*-Umfeld spielen bei dieser Art der Strukturierung neben PlugIns noch Erweiterungspunkte und Erweiterungen eine wichtige Rolle:

PlugIn Ein *Eclipse*-PlugIn beinhaltet Pakete und eine MANIFEST.MF-Datei[14], die deklarativ im *XML*-Format Abhängigkeiten, Erweiterungspunkte, Erweiterungen und die Sichtbarkeit von Paketen nach außen festlegt. Zusätzlich wird einem PlugIn ein eindeutiger Name zugeordnet und in der Regel eine Klasse definiert, die die Steuerung des PlugIn-Lebenszykus (initialisieren, starten, stoppen) übernimmt.

Erweiterungspunkt Ein Erweiterungspunkt deklariert eine Schnittstelle eines *Eclipse*-PlugIn nach innen, über die zusätzliche Funktionalität in dieses PlugIn integriert werden kann.

Erweiterung Eine Erweiterung ist eine Schnittstelle nach außen, über die Funktionalität eines PlugIns zur Verfügung gestellt wird[15].

In Abhängigkeit davon, ob PlugIns Erweiterungspunkte, Erweiterungen oder beides anbieten, können sie die Rolle eines *Hosts*, *Extenders* oder beide einnehmen. Das Zusammenspiel der PlugIns in dem Erweiterungsmechanismus von *Eclipse* lässt sich besonders gut mit einem Beispiel zur Erweiterung einer Benutzeroberfläche verdeutlichen. Ein Host-PlugIn, das die Oberfläche einer Anwendung implementiert, stellt einen Erweiterungspunkt vom Typ ActionSets bereit. Über diesen im *Eclipse*-System vordefinierten Erweiterungspunkt können Menü-Einträge oder Symbole in der Werkzeugleiste mit darunterliegender Funktionalität hinzugefügt werden.

[14] In früheren Version von *Eclipse* wurde der Inhalt von MANIFEST.MF noch auf zwei Dateien plugin.xml und MANIFEST.MF verteilt.

[15] Umgangsprachlich kann ein Erweiterungspunkt als Steckdose und eine Erweiterung als ein Stecker bezeichnet werden.

Während seiner Laufzeit beobachtet dieser Host, welche PlugIns (*Extender*) ihre Erweiterungen für diesen Erweiterungspunkt angemeldet haben und integriert ihre Einträge in die Oberfläche der Anwendung. So kann ein Host-PlugIn erweitert werden, ohne dass es verändert werden muss. Im *OBSE*-Tool werden auf diese Weise zum Beispiel Transformationen in die gemeinsame Oberfläche integriert.

Einen besonderen Stellenwert in der Infrastruktur des *OBSE*-Prozesses nimmt die Sprache *CPL* ein. So wie sie dort die Rolle eines Drehkreuzes ausübt, das verschiedene Artefakte miteinander verknüpft, übernimmt das *CPL*-Meta-Modell (cpl.ecore) und das zugehörige obsetool.cpl.metamodell-PlugIn die entsprechende Rolle in dem *OBSE*-Tool. Dieses PlugIn steht im Mittelpunkt der PlugIn-Architektur des Tools, die mittels des *UML*-Komponentendiagramms in der Abbildung 6.3 dargestellt ist. Das metamodel-PlugIn stellt als Extender über die Erweiterung org.eclipse.emf.ecore.generated_package[16] einen Zugriff auf das Meta-Modell anderen PlugIns zur Verfügung. Wie der Name dieser Erweiterung andeutet, handelt es sich dabei um eine automatisch generierte Implementierung, die alle Zugriffe auf die Elemente eines *Ecore*-Modells übernimmt. Die Erstellung der nötigen *Java*-Klassen und Schnittstellen übernimmt das *EMF*-Framework. Auf diese Weise wird zum Beispiel die Klasse ThingType des *CPL*-Meta-Modells auf das gleichnamige Interface und seine Implementierung abgebildet. Das wird für jedes Element des Meta-Modells durchgeführt. Zusätzlich erstellt *EMF* eine zentrale Klasse *CPLFactoryImpl*, mit der einzelne Meta-Modell-Elemente erstellt werden können.

Alle PlugIns, die auf dem *CPL*-Meta-Modell aufbauen, benutzen das obsetool.cpl.metamodel-PlugIn. Dadurch wird sichergestellt, dass unabhängig davon, ob ein *CPL*-Modell in einem graphischen, tabellarischen Editor oder während einer Transformation entstanden ist, es von diesen gemeinsam benutzt werden kann. So baut jede Komponente des *OBSE*-Tools[17] mindestens eine «use»-Beziehung zum PlugIn das *CPL*-Meta-Modells auf.

Die *CPL*-Editoren (linke obere Ecke der Abbildung) teilen ein gemeinsames PlugIn obsetool.cpl.edit, das die grundlegende Funktionalität zum Editieren von *CPL*-Modellen anbietet. Spezifische Repräsentation – tabellarisch oder graphisch – wird mit Hilfe der PlugIns obsetool.cpl.editor und obsetool.cpl.diagram realisiert. Diese PlugIns integrieren sich in die gemeinsame Oberfläche (realisiert durch das PlugIn obsetool.application) entspre-

[16] Erweiterungen und Erweiterungspunkte werden in diesem Diagramm der Übersichtlichkeit wegen nicht dargestellt.

[17] vgl. auch die Abbildung 6.2

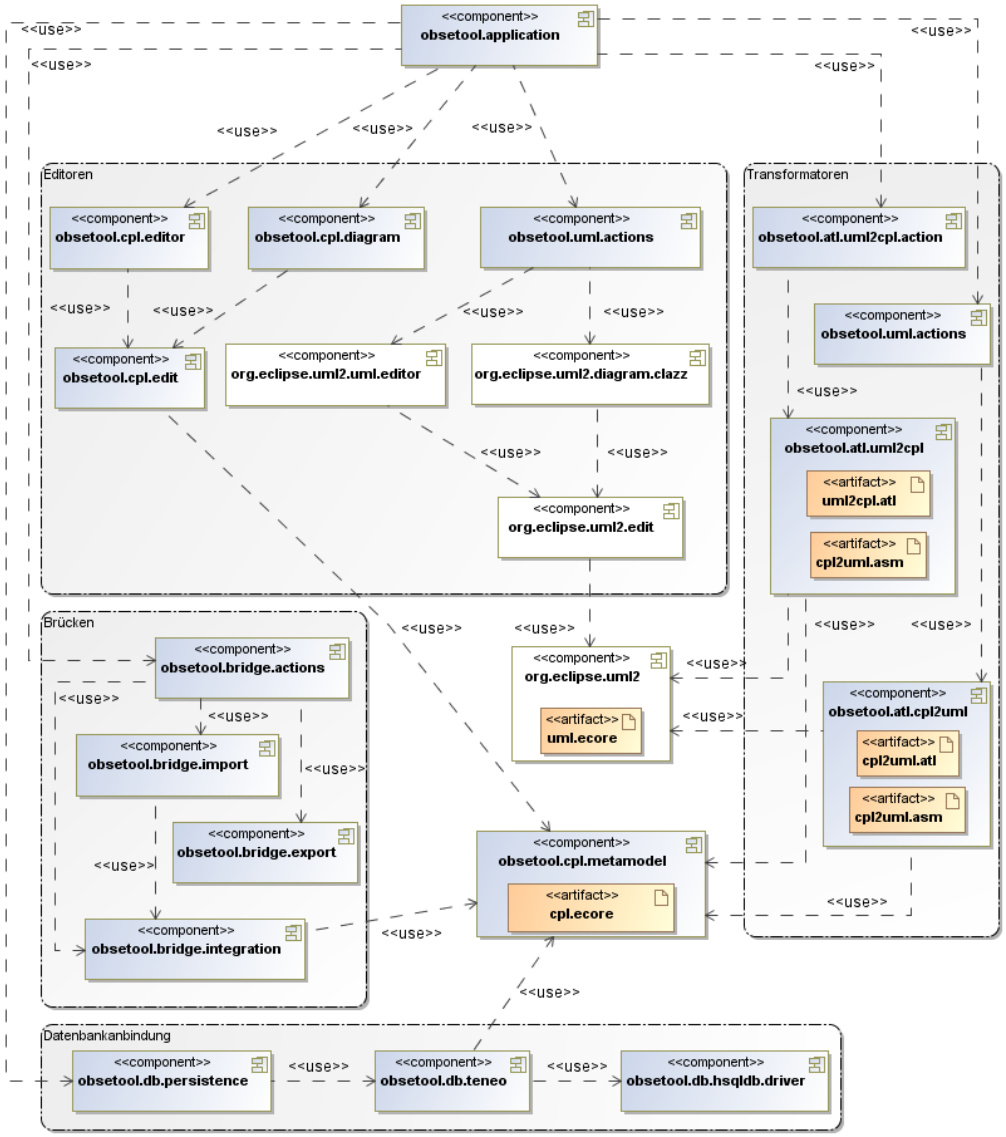


Abbildung 6.3: PlugIn-Architektur des OBSE-Tools

chend dem vorherigen ActionSets-Beispiel.

Bei den PlugIns für die Imoprt- und Export-Brücke liefert das obsetool.bridge.actions-PlugIn die Menü-Einträge für die Oberfläche des *OBSE*-Tools. Die PlugIns obsetool.bridge.import und obsetool.bridge.export stellen Wizards entsprechend den Aktivitätsdiagrammen in dem Kapitel 4.3.2 zur Verfügung. Die Ausführung von Integrationsaufgaben für *CPL*-Modelle ist in das PlugIn obsetool.bridge.integration ausgelagert, das auf dem *CPL*-Meta-Modell aufbaut. Da die Import-Brücke Integrationsschritte beinhaltet, greift sie auf das obsetool.bridge.integration-PlugIn zu. Ansonsten, um die Ontologie-Fragmente oder andere *CPL*-Modelle zu integrieren, sieht das obsetool.bridge.actions eine direkte Nutzung des obsetool.brdge.integration-PlugIns vor.

Die Datenbankanbindung erfolgt weitgehend automatisch mit Hilfe des *Teneo*-Frameworks. Als eine Objekt-Relationale-Brücke ermöglicht *Teneo*, Klassen aus dem obsetool.cpl.metamodel-PlugIn auf entsprechende relationale Datenbanktabellen abzubilden. Da verschiedene relationale Datenbanken zum Teil unterschiedlichen Funktionsumfang mitbringen, baut das obsetool.db.teneo-PlugIn auf Treibern für die zu verwendende Datenbank auf. Für diesen Erweiterungspunkt können verschiedene Treiber registriert werden, so dass jede relationale Datenbank, die einen jdbc-Treiber mitbringt[18], zum Ablegen von *CPL*-Modellen verwendet werden kann. Getestet wurde das *OBSE*-Tool mit der kompakten *HSQLDB*-Datenbank, wofür bereits ein entsprechendes PlugIn in der Architektur existiert.

Eine zu dem obsetool.cpl.metamodel-PlugIn vergleichbare Rolle in der Architektur übernimmt das org.eclipse.uml2-Plugin, das das *UML*-Meta-Modell zur Verfügung stellt. Es wird von dem baumartigen (org.eclipse.uml2.uml.editor) und einem Editor für die *UML*-Klassendiagramme (org.eclipse.uml2.diagram.clazz) verwendet. Dieses PlugIn und die darauf aufbauenden Editoren-PlugIns sind in der Abbildung 6.3 farblich abgesetzt, um zu verdeutlichen, dass es sich nicht um eigene Implementierung handelt, sondern dass sie von der *Eclipse*-Plattform in das *OBSE*-Tool integriert wurden. Auch die Namensgebung betont diesen Unterschied. Auf Grund ihres Ursprungs bringen sie eine andere Menü-Stuktur mit, als sie in dem *OBSE*-Tool gewünscht ist. Das PlugIn obsetool.uml.actions wird deswegen zwischen den *UML*-Editoren und der *OBSE*-Applikation zwischengeschaltet und übernimmt die Konvertierung.

[18] Das trifft zur Zeit auf alle gebräuchlichen, relationalen Datenbanken zu.

Die Transformatoren sind aus jeweils zwei PlugIns für jede Transformationsrichtung zusammengesetzt. Analog zu den anderen Komponenten liefern entsprechnde actions-PlugIns Erweiterungen wie Menü-Einträge für die Integration in die Oberfläche des *OBSE*-Tools. Die PlugIns obsetool.atl.uml2cpl und obsetool.atlcpl2uml stellen die eigentliche Funktionalität der Transformationen bereit. Sie beinhalten zwei Artefakte, die neben dem *Java*-Code eine entscheidende Rolle für die Umsetzung einer Transformation spielen. In den Dateien cpl2uml.atl und uml2cpl.atl befinden sich Transformationsregeln in der für Menschen lesbaren Form. Die Dateien cpl2uml.asm und uml2cpl.asm sind ihre maschineninterpretierbaren Gegenstücke, die während einer Transformation von der Virtuellen Maschine des *ATL*-Frameworks ausgeführt werden[19].

Das obsetool.application-PlugIn ist im Unterschied zu allen anderen PlugIns in der Abbildung 6.3 als eine eigenständige *RCP*-Anwendung deklariert und ist ausführbar. Außerdem werden hier Aufgaben-übergreifende Elemente des *OBSE*-Tools implementiert. Zum Beispiel bietet obsetool.application eine Möglichkeit, *OBSE*-Projekte oder Projekte zum Verwalten der Domänen-Ontologie zu erstellen und zu bearbeiten. Die wichtigste Funktion dieses PlugIns liegt aber darin, als Host Anknüpfpunkte für die Dienste der darunterliegenden PlugIns anzubieten und sie einem Benutzer unter einer gemeinsamen Oberfläche zur Verfügung zu stellen.

Was das Diagramm in der Abbildung 6.3 nicht zeigt, ist die Menge an PlugIns, die von der *Eclipse*-Plattform zusätzlich aktiviert werden, um die Ausführung der vorgestellten PlugIn-Struktur zu ermöglichen. Abhängig von der Konfiguration des *OBSE*-Tools handelt es sich dabei um bis zu 190 zusätzliche PlugIns. Eine derart komplexe Anwendung im Rahmen dieser Arbeit herzustellen, war nur unter Verwendung von ausgereiften und weitgehend auf der Modell-getriebenen Vorgehensweise aufbauenden *Eclipse*-Werkzeugen möglich. Ein Einblick in die Implementierung des *OBSE*-Tools und Verwendung dieser Werkzeuge wird in dem kommenden Kapitel anhand von ausgewählten Beispielen gegeben.

6.3 Umsetzung

Für einen Einblick in die Umsetzung sind Editoren für die Sprache *CPL* besonders interessant. Einerseits sind das Komponenten des Tools, die im *OBSE*-Prozess am meisten und an verschiedenen Stellen gebraucht wer-

[19] Siehe Kapitel 2.1.4

den. Andererseits handelt es sich dabei um komplexe Werkzeuge, deren Umsetzung eine automatische Generierung vom Code und seine manuelle Anpassung beinhaltet.

6.3.1 Tabellarischer Editor für *CPL*

Die wichtigsten Schritte bei der Implementierung des tabellarischen Editors waren:

1. Erstellen der *Ecore*-Version des *CPL*-Meta-Modells (cpl.ecore).
2. Ableitung des *Generator-Modells* (cpl.genmodel).
3. Generieren der obsetool.cpl.metamodel-, obsetool.cpl.metamodel.edit- und obsetool.cpl.metamodel.editor-PlugIns.
4. Anpassungen an dem generierten Code.

Die Notwendigkeit des ersten Schrittes hängt damit zusammen, dass die Grundlage des *EMF*-Frameworks *Ecore*-Modelle bilden. *Ecore* kann als eine Vereinfachung von *UML* angesehen werden, bei der nur ausgewählte Sprachkonstrukte übernommen wurden. Diese Auswahl wurde vor allem mit Blick auf effiziente Generierung vom *Java*-Code getroffen. Die für die Entwicklung des tabellarischen Editors notwendige Konvertierung des *CPL*-Meta-Modells zu dem entsprechenden *Ecore*-Modell konnte wegen der Nähe von *Ecore* und *UML* problemlos durchgeführt werden. Lediglich bidirektionale Assoziationen und Assoziationsklassen, die unter *Ecore* nicht unterstützt werden, müssen in zwei unidirektionale bzw. normale Klassen umgewandelt werden. Ansonsten unterscheidet sich das entstandene *Ecore*-Modell inhaltlich[20] nicht von dem im Kapitel 4.1.2 präsentierten *CPL*-Meta-Modell.

Im zweiten Schritt wird ein Generator-Modell erstellt, das sich im wesentlichen aus Einstellungen des Generators und einer Verknüpfung auf das im ersten Schritt erzeugte *Ecore*-Modell für die Sprache *CPL* zusammensetzt. Das Generator-Modell ist nützlich, da es während der Editor-Entwicklung durchaus vorkommen kann, dass die Generierung vom Code wiederholt ausgeführt wird. In diesem Fall müssen die Einstellungen für den Generierungsvorgang nicht jedes Mal neu eingegeben werden. Dazu gehören zum Beispiel Präfixe der zu generierenden Pakete, Pfade und Schalter,

[20] Was sich noch unterscheidet aber für den Informationsgehalt des Modells keine Rolle spielt, ist die abweichende Benennung der Sprachelemente in *Ecore*. So wird eine Klasse in *Ecore* mit *EClass* statt *Class* bezeichnet.

die bestimmte Funktionalität in dem generierten Code erlauben oder unterdrücken.

Die Code-Generierung im dritten Schritt ist ein vollautomatisierter Vorgang. *EMF* erstellt deutlich mehr Code als die konventionellen Generatoren der *UML*-Werkzeuge. Nicht nur Code für die statische Struktur wird erstellt, sondern auch Code für das Anlegen, Bearbeiten und Darstellen von Modell-Elementen im Rahmen einer *RCP*-Anwendung. Zunächst wird für jede Klasse des *CPL*-Meta-Modells ein Interface und die zugehörige Implementierung als *Java*-Klasse angelegt. Für einen Dingtypen sind das zum Beispiel ThingType und ThingTypeImpl (siehe Beispiel in dem Listing 6.1).

Für die weitere Verwendung in den Editoren ist es wichtig, dass diese Klassen indirekt von EObject – *EMF*-Pedant für java.lang.Object – abgeleitet sind. Diese Klasse ist wiederum von dem Interface Notifier abgeleitet (siehe Zeile 6). Das hat zur Folge, dass die Modell-Elemente entsprechend dem *Observer*-Entwurfsmuster registrierte Beobachter informieren, wenn Änderungen an diesen durchgeführt wurden. Das ist ein für *OBSE*-Tool unverzichtbares Verhalten. So werden Änderungen in einem Editor vom anderen wahrgenommen. Dieser Editor kann auf diese Änderungen reagieren und seine Anzeige aktualisieren. Die Darstellung eines Modells in beiden Editoren bleibt so konsistent.

Mit dem Interface CplFactory und seiner Implementierung CplFactoryImpl wird eine Vereinheitlichung der Vorgehensweise bei der Erzeugung von Modellelementen erreicht. Realisiert wird dies mit Hilfe des *Factory*-Entwurfsmusters, was die Flexibilität des generierten Codes erhöht und möglichen späteren Änderungen an dem zu Grunde liegenden *Ecore*-Modell entgegenkommt.

```
1  public interface Thingtype extends ModelingElement ...
2
3  public class ThingtypeImpl extends ModelingElementImpl
4          implements Thingtype ...
5
6  public interface EObject extends Notifier ...
7
8  public interface CplFactory extends EFactory ...
9
10 public class CplFactoryImpl extends EFactoryImpl
11         implements KcpmmFactory ...
12 }
```

LISTING 6.1: Einblick in die generierte Code-Struktur des tabellarischen Editors

Viele Hilfsfunktionen wie Änderungen rückgängig machen, Kopieren und Einfügen, „drag and drop“ werden ebenfalls in den generierten Code integriert[21]. Ist der *CPL*-Editor nach der Generierung voll einsetzbar? Diese Frage kann mit ja beantwortet werden, allerdings ist dieser Editor generisch und berücksichtigt die Besonderheiten einer Sprache nicht. Es ist zwar möglich, Elemente anzulegen und baumartig über das Modell zu navigieren, jedoch fehlt die für die Sprache *CPL* wichtige tabellarische Darstellung noch in diesem Stadium.

Damit der Editor Anpassungen im generierten Code bei einer erneuten Generierung nicht überschreibt, besitzen Methoden eine spezielle Annotation. Standardmäßig wird das Schlüsselwort `@generated` verwendet, um den automatisch entstandenen Code zu kennzeichnen. Wird eine Methode angepasst, reicht es mit `@generated NOT` zu signalisieren, dass sie bei der nächsten Generierung nicht mehr verändert werden darf.

Im *OBSE*-Tool kommunizieren Editoren und Modell-Elemente nicht direkt miteinander, sondern über zwei spezielle Klassen: `IContentProvider` und `ILabelProvider`. Die erste wird benutzt, um über den Inhalt eines Modells zu navigieren und die zweite, um ein Modell-Element darzustellen (meistens als Name eines Elements versehen mit einem Symbol). Auf diese Art können analog zu dem *Model-View-Controler*-Entwurfsmuster mehrere Sichten auf ein *CPL*-Modell angeboten werden. Gleichzeitig ist die Implementierung eines neuen Providers der richtige Einstiegspunkt, um den generierten Code

[21] Für weiterführende Informationen zum *EMF*-generierten Code und seinem Design siehe [BSM+03].

zu erweitern.

```
1  tableViewerOfThingtype.setContentProvider
2      (new AdapterFactoryContentProvider(adapterFactory) {
3          public Object [] getElements(Object object){
4              return ((Model)object).getContain().toArray();
5          }
6          public void notifyChanged(Notification notification){
7              switch (notification.getEventType()){
8                  case Notification.ADD:
9                  case Notification.ADD_MANY:
10                     if ( notification.getFeature() != CplPackage
11                         .eINSTANCE.getModel_Contain()) return;
12             }
13             super.notifyChanged(notification);
14         }
15 });
16
17 tableViewerOfThingtype.setLabelProvider
18     (new AdapterFactoryLabelProvider(adapterFactory));
```

LISTING 6.2: Registrierung neuer Provider für den tabellarischen Editor

Die Listing 6.2 zeigt einen Ausschnitt der Methode createPages() aus der Klasse *CplEditor*, in dem der tabellarischen Darstellung (tableViewerOfThingtype) ContentProvider (Zeilen 2-16) und LabelProvider (Zeilen 18-19) zugewiesen werden. Als Beispiel wird an dieser Stelle die Aktivierung des Benachrichtigungsmechanismus beim Hinzufügen von neuen Elementen gezeigt (Zelen 6-14). Dadurch werden alle *Observer* beim Erzeugen von neuen Modell-Elementen (Bedingung in der Zeile 10) informiert. Ergänzend dazu müssen noch viele weitere Anpassungen vorgenommen werden, um generierte ContentProvider von *CPL*-Modellierungselementen wie Dingtyp zu erweitern und die Spalten der Tabelle und ihre Inhalte festzulegen. Diese Änderungen können anhand des beiliegenden Quellcodes des *OBSE*-Tools verfolgt werden (qgenerated NOT-Methoden in den PlugIns obsetool.cpl.metamodel, obsetool.cpl.metamodel.edit und obsetool.cpl.metamodel.editor).

6.3.2 Grafischer Editor für *CPL*

Plugins, die Elemente die Elemente des *CPL*-Meta-Modells (cpl.ecore in obsetool.cpl.metamodel) und allgemeine Bearbeitungsfunktionen (obsetool.cpl.-metamodell.edit) bereitstellen, werden gemeinsam von dem tabellarischen und grafischen Editor verwendet. Die Entwicklung des grafischen Editors baut somit auf den ersten drei Schritten bei der Entwicklung des tabellarischen Editors auf. Daran anknüpfend wurden noch folgende Schritte durchgeführt:

1. Erstellen des *Grafik-Modells* (static.gmfgraph)
2. Erstellen des *Werkzeug-Modells* (static.gmftool)
3. Kombinieren der *Grafik-*, *Werkzeug-* und des *CPL-Ecore*-Modelle zu einem *Mapping-Modell* (static.gmfmap)
4. Ableiten und Konfigurieren des *Generator-Modells* (static.gmfgen)
5. Generieren der obsetool.cpl.metamodel.diagram-PlugIns.

Die deklarative Entwicklung des Editors beginnt mit der Festlegung von grafischen Figuren im Grafik-Modell. Hier wird zum Beispiel deklariert, dass ein Dingtyp als ein Viereck dargestellt werden soll, das im Kopf seinen Namen und im Rumpf seine Beschreibung enthält (siehe Listing 6.3). Zusätzlich beinhaltet das Grafik-Modell Definitionen von Knoten und Kanten des Diagramms (Zeile 13). Gleichzeitig wird an dieser Stelle eine Verknüpfung zwischen grafischen Figuren und Knoten und Kanten des zukünftigen Diagramms vorgenommen.

Ein grafischer Editor bietet in der Regel neben der Diagrammfläche eine Sammlung von Werkzeugen, die verschiedene Manipulationen von Elementen eines Diagramms erlauben. Dazu zählen in erster Linie Werkzeuge zum Anlegen von neuen Modellelementen. Die Werkzeuge werden vorerst unabhängig von Knoten und Kanten festgelegt, für die sie später verwendet werden. Eine Deklaration aus dem Werkzeug-Modell steht im Listing 6.3 ab Zeile 15. Sie legt fest, wie ein Werkzeug-Symbol aussehen soll, das später für das Erstellen eines Dingtypes verantwortlich sein soll.

Der zentrale Schritt bei der Entwicklung eines graphischen Editors mit *GEF* ist die Verknüpfung von Grafik-, Werkzeug- und Ecore-Meta-Modell. Ein Mapping-Modell verbindet somit grafische Elemente eines Modells mit Meta-Modell-Elementen, die sie repräsentieren, und mit Werkzeugen, die Manipulationen eines Diagramms erlauben. Die minimale Definition in dem Mapping-Modell steht in dem Listing 6.3 ab der Zeile 23. In der Regel wird eine Verknüpfung dieser Art mit weiteren Angaben versehen, die eine Ver-

wendung des definierten Elements in dem Diagramm steuern. Graphischer Editor nutzt diese Möglichkeit, um Einschränkungen zu definieren. Zum Beispiel wird dadurch verhindert, dass die IsA-Beziehung bidirektional zwischen zwei Dingtypen angelegt wird. Diese Einschränkungen werden in *OCL* ausgedrückt.

```
<!-- static.gmfgraph -->
<descriptors name="ThingTypeFigure">
    <actualFigure xsi:type="gmfgraph:Rectangle"
        name="ThingTypeFigure">
        <foregroundColor xsi:type="gmfgraph:ConstantColor"
            value="lightBlue"/>
        <backgroundColor xsi:type="gmfgraph:RGBColor"
            red="230" green="230" blue="255"/>
        ...
    </actualFigure>
</descriptors>

<nodes name="ThingType" figure="ThingTypeFigure"/>

<!-- static.gmftool -->
<tools xsi:type="gmftool:CreationTool" title="ThingType"
    description="Create a new ThingType">
    <smallIcon
        xsi:type="gmftool:BundleImage" path="icons/thingtype.png"/>
    <largeIcon xsi:type="gmftool:DefaultImage"/>
</tools>

<!-- static.gmfmap -->
<nodes>
    <ownedChild>
        <domainMetaElement href="cpl.ecore#//ThingType"/>
        <tool xsi:type="gmftool:CreationTool"
            href="static.gmftool#//@palette/@tools.0/@tools.0"/>
        <diagramNode href="static.gmfgraph#ThingType"/>
    </ownedChild>
</nodes>
```

Listing 6.3: Einblick in die Definition der Diagrammelemente

Analog zum *EMF* legt *GEF* für die Code-Generierung erforderlichen Informationen getrennt von der eigentlichen Deklaration des Editors in dem Generator-Modell ab. Aus diesem Modell wird der Code generiert.

Nachträgliche manuelle Anpassungen sind auch im *GEF* generierten Code möglich, waren allerdings für die Umsetzung des graphischen Editors nicht erforderlich. Der Grund dafür sind die vielfältigen Möglichkeiten bei der deklarativen Beschreibung der Elemente des Diagramms. Allerdings führt diese Vielfalt zu einer längeren Einarbeitungszeit in das Framework. Insbesondere eine Abschätzung, wie eine deklarativ beschriebene Figur später im Diagramm aussieht, lässt sich hauptsächlich durch Ausprobieren herausfinden, was mit wiederholter Code-Generierung verbunden ist. Dieser Nachteil wurde durch die ausgereifte Werkzeug-Unterstützung und schnelle Code-Generierung mehr als kompensiert und hat die Entwicklung des graphischen Editors im Rahmen dieser Arbeit erst ermöglicht.

6.3.3 Ergebnis

Beide Editoren sind in dem Screenshot des *OBSE*-Tools in der Abbildung 6.4 zu sehen. Im mittleren Bereich des Bildes ist ein kleines *CPL*-Beispiel im grafischen Editor (oben) und dem tabellarischen Editor (unten) dargestellt. Links davon ist ein *Navigator*, in dem verschiedene Dateien der *OBSE*-Projekte ausgewählt werden können. Rechts von den Editoren stehen weitere nützliche Werkzeuge *Outline* und *Properties*, die einen Benutzer bei der Bearbeitung der *CPL*-Modelle und Sprachelemente unterstützen. *Outline* zeigt eine hierarchische, baumartige Darstellung des gleichen *CPL*-Modells. *Property* ermöglicht es, alle möglichen Informationen über ein Sprachelement einzugeben. In dem Screenshot sind das die Felder des Dingtyps Book.

Die Version des *OBSE*-Tools, die im Rahmen dieser Arbeit entwickelt wurde, kann als „Proof of Concept“ bezeichnet werden. Das Ziel bei der Entwicklung von Komponenten und den zugehörigen PlugIns war zu zeigen, dass die entsprechenden Werkzeuge sich als Teil einer *RCP*-Anwendung realisieren lassen und mit passenden Frameworks wie *EMF* ihre Implementierung sehr effizient und mit einem hohen Grad an Stabilität erstellt werden kann. Trotz diesem frühen Stadium und noch laufender Weiterentwicklung – es fehlt noch eine *RCP*-kompatible Umsetzung des Integrationsverfahrens aus Klagenfurt – hat sich das *OBSE*-Tool in dem praktischen Einsatz während der Evaluierung (siehe nächstes Kapitel) des Prozesses bewährt und ist zur Zeit das benutzerfreundlichste Werkzeug für die *CPL*-Modellierung.

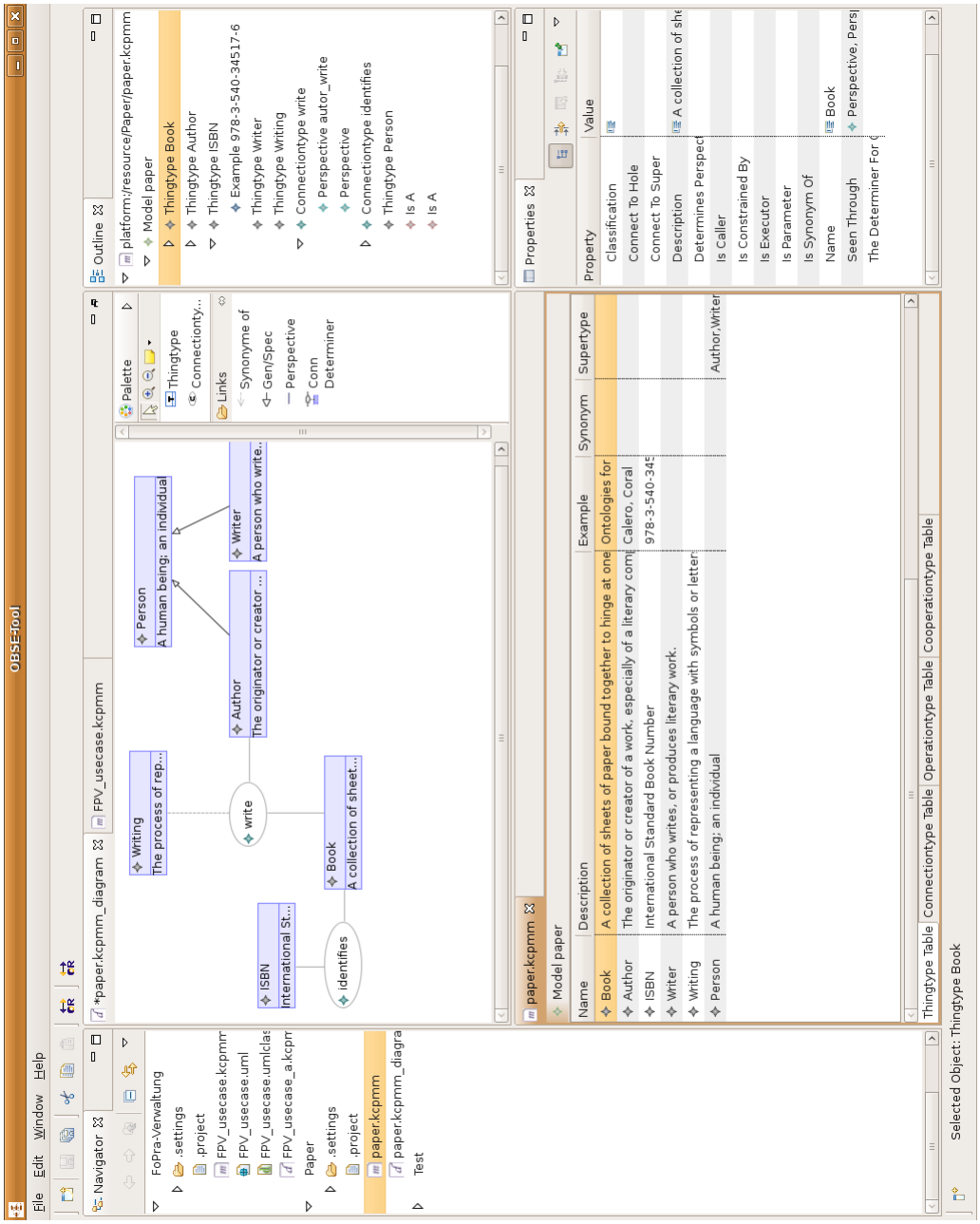


ABBILDUNG 6.4: Ein Screenshot des *OBSE*-Tools

7 Evaluierung

Software-Entwicklungsprozesse sind komplexe Gebilde und ihr Wirkungsgrad hängt von vielen Faktoren ab. Eine umfassende Analyse eines Prozesses, die neben der Prozess-Definition auch den Grad der Anwendung der vorgeschriebenen Aktivitäten und ihren Nutzen beurteilt, erfordert den Einsatz des Prozessmodells in einem oder besser noch in mehreren Unternehmen inklusive anschließender Zertifizierung nach bekannten Standards wie *ISO 9000* oder dem *Capability Maturity Model*. Für die erste Analyse des *OBSE*-Prozesses mit dem Ziel, das Zusammenspiel seiner Infrastruktur-Elemente und Prozessabläufe zu untersuchen, ist diese Art der Evaluierung allerdings überdimensioniert und zu unscharf. Um vor dem tatsächlichen Einsatz des Prozesses im Feld zu beurteilen, inwieweit der Prozess funktioniert und die gesetzten Ziele bezüglich der Wiederverwendung und Verbesserung der Modell-Qualität erfüllt werden, wurde eine spezielle Versuchsanordnung zur Evaluierung des *OBSE*-Prozesses entworfen, die in diesem Kapitel zusammen mit den entstandenen Ergebnissen vorgestellt wird.

7.1 Einführung

Die Idee der Evaluierung basiert auf einem Vergleich der Ergebnisse bereits abgeschlossener Projekte einer Domäne mit den Ergebnissen einer Emulation, die den Ablauf dieser Projekte unter *OBSE* nachbildet. Mit Hilfe der Emulation soll die Zeit eingespart werden, die für eine wiederholte, verzahnte Ausführung dieser Projekte erforderlich wäre. Der Schwerpunkt bei der Beurteilung der Artefakte des *OBSE*-Prozesses soll auf konzeptuellen Modellen und der Domänen-Ontologie liegen, da insbesondere die frühen Phasen der Software-Entwicklung durch *OBSE* beeinflusst werden. Die herangezogenen Projekte stellen einerseits mit ihren Anforderungen einen festen Bezug zu der Realität her. Andererseits sollen ihre konzeptuellen Modelle

eine Vergleichsbasis liefern, die zur Beurteilung der Qualitätsunterschiede zu den aus der Emulation entstandenen konzeptuellen Modellen verwendet wird.

7.2 Versuchsanordnung

Die Grundlage bilden folgende an der Philipps-Universität Marburg abgeschlossene Projekte:

Scheinverwaltung Das Projekt hatte zum Ziel, Prozesse zur Unterstützung der Lehre wie Übungsbetrieb und Schein-Vergabe an dem Fachbereich Mathematik und Informatik zu automatisieren. Es wurde in kleinen Teilprojekten mit im Schnitt sechs Projektmitgliedern entwickelt. Die Teilprojekte hatten eine Dauer von sechs Monaten. Die gesamte Entwicklungsdauer betrug zwei Jahre. Scheinverwaltung ist eine Web-Anwendung auf der Basis von *JSP*, *Java* und *MySQL*. Sie wurde inkrementell entwickelt. Zu der Projektdokumentation gehört ein Analyse-Dokument, das unter anderem die Anforderungen an das künftige System enthält. Insgesamt wurde die Anwendung in sechs Bereiche Account, Semester, Veranstaltung, Schein, Student und Termin unterteilt, zu denen insgesamt 30 Anwendungsfälle identifiziert wurden.

SpSemOnline Mit der Sprechstunden- und Seminarverwaltung Online[1] wurde eine elektronische Anmeldung zu den Sprechstunden von Dozenten und Seminaren am Fachbereich Erziehungswissenschaften realisiert. Es war ein kleines Projekt, das innerhalb eines Jahres von zwei Teilnehmern durchgeführt wurde. Als Ergebnis entstand eine Web-Anwendung auf der Basis von *PHP* und *MySQL*. Die Entwicklung folgte keinem Vorgehensmodell. Für die Evaluierung wurden eine Anforderungsliste und der entstandene Code herangezogen. Die Anforderungsliste ist in vier Bereiche Sprechstunde, Dozenten, Nachrücker und Sonstiges mit insgesamt 48 natürlichsprachlich beschriebenen Punkten gegliedert.

FoPra-Verwaltung Die Aufgabe der FoPra-Verwaltung[2] ist es, die Prozesse um die Präsentation und Vergabe von Fortgeschrittenen-Praktika am Fachbereich Mathematik und Informatik zu automatisieren. Das

[1] Dieses Projekt diente bereits als Quelle für die Beispiele im Kapitel 4.2.
[2] FoPra ist ein Akronym für Fortgeschrittenen-Praktikum.

System wurde in zwei Teilprojekten zum Teil Modell-getrieben mit *Struts*, *Java* und *MySQL* entwickelt. Im ersten Teilprojekt wurde konventionell nach einem *RUP*-Ableger *OpenUP* entwickelt. Im zweiten Teilprojekt kam *Scrum* zum Einsatz. Die gesamte Projektdauer betrug ein Jahr, die durchschnittliche Zahl der Projektmitglieder lag bei fünf. Die Funktionalität des Systems wurde aus 20 Anwendungsfällen zu den Bereichen System, Benutzer und Administrator abgeleitet.

Diese Projekte wurden ausgewählt, weil sie alle in einer Domäne – Universität am Beispiel der Philipps-Universität Marburg – liegen und somit für den Wissensaustausch nach dem *OBSE*-Prozess geeignet sind. Gleichzeitig unterscheiden sie sich in mehreren Punkten voneinander und ermöglichen dadurch eine facettenreiche Untersuchung des *OBSE*-Prozesses. Folgende zentrale Fragestellungen können mit Hilfe dieser Projekte angesprochen werden:

- Die Gegenstandsbereiche der Projekte befinden sich in unterschiedlichen Bereichen der Universitäts-Domäne. Können die Projekte auch in diesem Fall von dem Wissenstransfer profitieren?
- Unterschiedliche Programmiersprachen und Techniken bei der Anwendungsentwicklung machen es in der Regel unmöglich, Wiederverwendung auf der Entwurfs- oder Implementierungsebene einzusetzen. Ist dies auf der konzeptuellen Ebene mit *OBSE* gelungen?

Unabhängig von den Unterschieden in Projekten war bei der Evaluierung außerdem die Frage wichtig, ob und inwieweit mit Hilfe der Domänen-Ontologie die Qualität der Modelle in den Projekten verbessert werden konnte.

Der Ablauf der Evaluierung ist in dem Aktivitätsdiagramm in Abbildung 7.1 dargestellt. Die zeitliche Abfolge der Projekte wurde bei der Evaluierung beibehalten. So wurde das Projekt Scheinverwaltung als erstes und die Projekte SpSemOnline und FoPra-Verwaltung wurden anschließend weitgehend parallel durchgeführt.

Als vorbereitenden Schritt für die Evaluierung, bei dem keine Analyse stattfindet sondern zunächst die Domänen-Ontologie aufgebaut wird, wurde das Projekt Scheinverwaltung verwendet. Um möglichst viele Konzepte und Beziehungen aus der Domäne zu gewinnen, wurden sowohl die Anforderungen als auch der Code der Scheinverwaltung analysiert und als konzeptuelles Modell modelliert[3]. Auf das konzeptuelle Modell der Scheinverwaltung wurde dann entsprechend dem *OBSE*-Prozess die Export-Brücke angewendet,

[3]Für weitere Informationen siehe auch [Hor10].

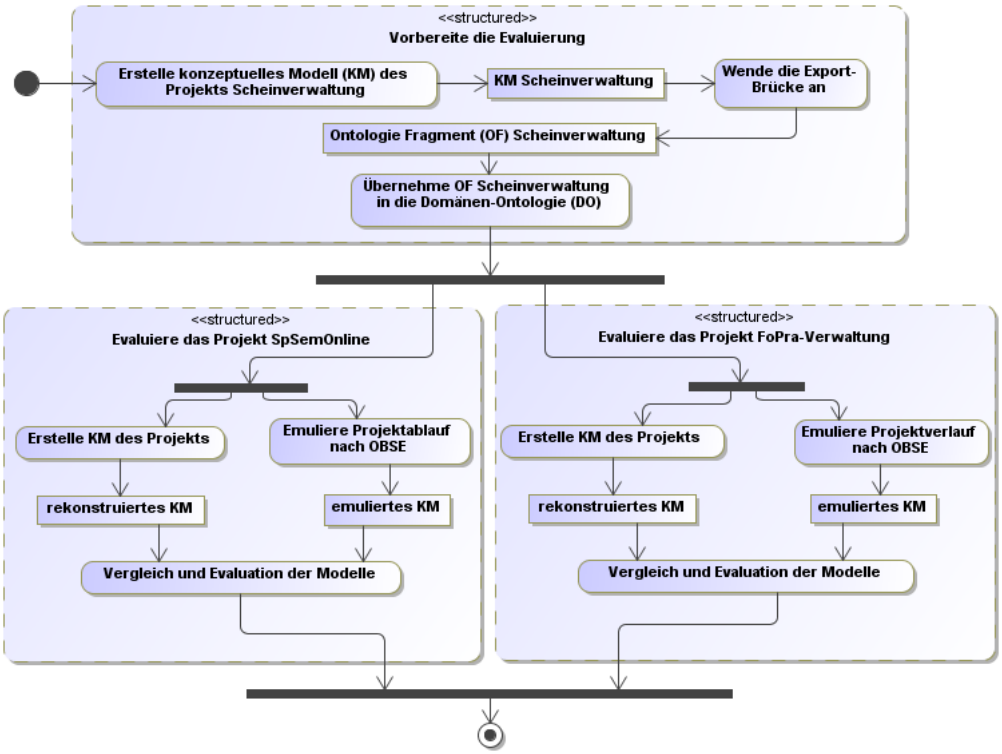


ABBILDUNG 7.1: Evaluierung des *OBSE*-Prozesses

die ein Ontologie-Fragment zur Verfügung stellt. Da die Domänen-Ontologie zu diesem Zeitpunkt leer ist, wird dieses Fragment in die Ontologie ohne Integration übernommen.

Die Evaluierung der Projekte SpSemOnline und FoPra-Verwaltung wird parallel und auf die gleiche Weise durchgeführt. Für jedes Projekt werden zwei konzeptuelle Modelle erstellt:

rekonstruiertes KM Dieses Modell soll nur Konzepte und Beziehungen enthalten, die tatsächlich in dem jeweiligen Projekt umgesetzt wurden. Aus diesem Grund wurden diese konzeptuelle Modelle in der Aktivität *Erstelle KM des Projekts* aus dem Code (SpSemOnline) oder *UML*-Modellen (FoPra-Verwaltung) rekonstruiert. Diese strukturierte Aktivität ist in Abbildung 7.2 links detailliert dargestellt. Im Falle von FoPra-Verwaltung reichte es auf Grund der Modell-basierten Entwicklung aus, nur das *UML*-Klassendiagramm einzubeziehen. Beim *SpSemOnline*-Projekt standen keine Modelle zur Verfügung.

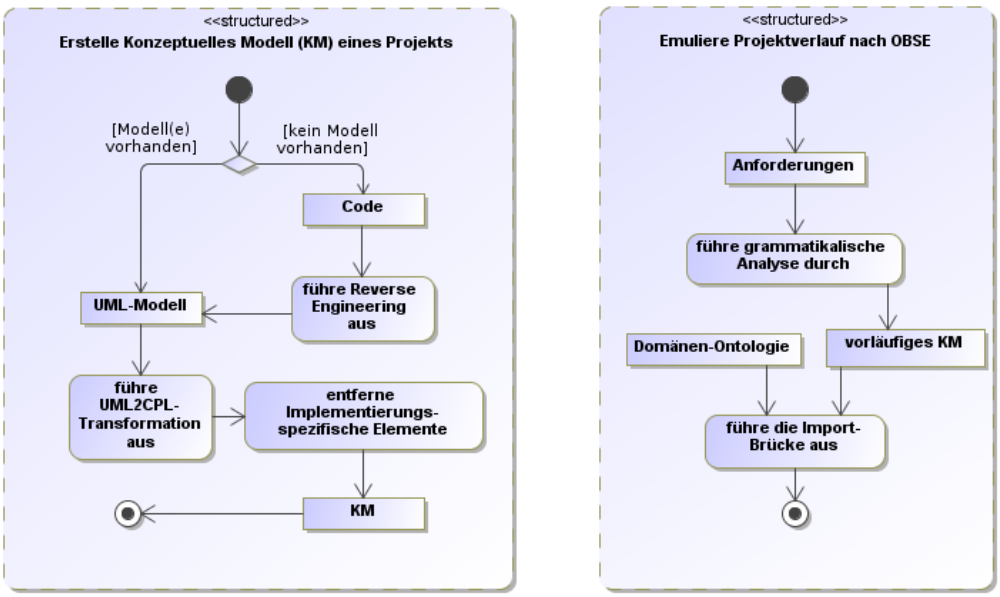


ABBILDUNG 7.2: Unterdiagramme des Evaluierungsablaufs

emuliertes KM Dieses Modell soll durch die Emulation der Analyse-Phase des *OBSE*-Prozesses (siehe Abbildung 7.2 rechts) entstehen. Die Quellen für dieses Modell sind nur Anforderungen der Projekte und die Domänen-Ontologie. Dieses Modell wird somit unabhängig von der existierenden Projektrealisierung erstellt.

Schließlich werden diese zwei Modelle verglichen (siehe Abbildung 7.1). An dieser Stelle werden qualitative Unterschiede in der Modellierung ohne und mit OBSE untersucht.

Die Beurteilung der Modell-Qualität in diesem Schritt basiert auf der Klassifikation von Fieber, Huhn und Rumpe [FHR08]. Die Autoren stellen eine Taxonomie auf, die auf der obersten Ebene zwischen der *inneren* und *äußeren* Qualität eines Modells unterscheidet. Dabei werden Qualitätsmerkmale, die ausschließlich das untersuchte Modell betreffen, der inneren Qualität zugeordnet. Die äußere Qualität enthält Merkmale, die sich im Bezug auf andere Artefakte (meist Modelle) auswirken. Dieser Bereich wird zusätzlich in zwei Unterbereiche anhand der *horizontalen* und *vertikalen* Beziehungen zwischen Modellen aufgeteilt. Äußere horizontale Qualitätsmerkmale wirken sich auf Modelle aus, die in der gleichen Entwicklungsebene liegen. Ein Beispiel dafür ist ein Anforderungsmodell und ein zugehöriges, ergänzendes

Aktivitätsdiagramm. Äußere vertikale Merkmale betreffen zugehörige Modelle auf verschiedenen Ebenen, wie im Falle von *OBSE* ein konzeptuelles und Analyse-Modell.

Die Taxonomie von Fieber, Huhn und Rumpe ist sehr breit gefächert, um alle Aspekte des Feldes Modellqualität abzudecken. Für die Evaluierung des *OBSE*-Prozesses spielen folgende Qualitätsmerkmale eine wichtige Rolle, die anhand von entdeckten Unterschieden in den untersuchten Modellen identifiziert wurden:

Präzision Nach diesem inneren Qualitätsmerkmal sollen alle relevanten Elemente der Domäne entsprechend der Detailliertheit des Modells wiedergegeben werden. Die Relevanz der Elemente ergibt sich aus den Anforderungen an das künftige System.

Einfachheit Einfachheit ist ein weiteres, inneres Qualitätsmerkmal, das besagt, dass die Modelle nicht komplizierter sein sollen als die Sachverhalte, die sie repräsentieren. Zum Beispiel soll die Verwendung von nicht relevanten Elementen vermieden werden.

konzeptionelle Integrität Dieses äußere horizontale Qualitätsmerkmal verlangt, dass gleiche Sachverhalte auf gleicher Modellierungsebene auf die gleiche Art und Weise modelliert werden.

Verfolgbarkeit Verfolgbarkeit ist ein äußeres vertikales Merkmal. Dabei ist es wichtig, dass alle Elemente der unteren Ebenen zu ihrem Ursprung in den höheren Ebenen vervolgt werden können.

Korrektheit Korrektheit als ein äußeres vertikales Qualitätsmekmal bedeutet, dass das Modell die Anforderungen in den Modellketten präzise abbildet.

Änderbarkeit Änderbarkeit ist ein weiteres, äußeres, vertikales Qualitätsmerkmal, das sich wiederum aus den Merkmalen Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit zusammensetzt. *Wartbarkeit* ist der Aufwand, der betrieben werden muss, um im Modell Fehler zu beheben sowie Spezifikationsänderungen zu modellieren. *Erweiterbarkeit* ist ein Maß für die Leichtigkeit, mit dem das System erweitert werden kann. *Wiederverwendbarkeit* ist die Eigenschaft des Modells, für andere Systeme oder andere Varianten des Systems z.B. in einer Produktlinie wiederverwendet werden zu können.

Neben dem speziell für die Evaluierung des *OBSE*-Prozesses vorgesehenen Vergleich der konzeptuellen Modelle lieferte noch eine Stelle in dem Evaluierungsablauf interessante Erkenntnisse über den *OBSE*-Prozess. Das ist die Anwendung der Import-Brücke in der Aktivität Emuliere Projektver-

lauf nach OBSE (siehe Abbildung 7.2 rechts). An dieser Stelle konnte die Auswirkung der Wiederverwendung auf der konzeptuellen Ebene untersucht werden.

7.3 Ergebnisse

Im diesem Abschnitt werden die Ergebnisse der Evaluierung zusammenfassend[4] vorgestellt. Zunächst kommen Fälle, die bei der Ausführung der Import-Brücke identifiziert wurden. Anschließend kommt der Vergleich der konzeptuellen Modelle. Bei der Betrachtung der Beispiele soll beachtet werden, dass sie aus Projekt-Artefakten entstanden sind und weder korrigiert noch optimiert wurden. Auch die Domänen-Ontologie wurde bei der Evaluierung nur anhand von Daten aus einem Projekt aufgebaut und ihre Qualität entspricht weitgehend seiner Modellierung. Dies sollte eine praxisnahe Beurteilung des *OBSE*-Prozesses ermöglichen und zeigen, dass bereits am Anfang der Domänen-Ontologie-Entwicklung[5] der *OBSE*-Prozess zu qualitativen Verbesserungen von Projektmodellen beiträgt.

7.3.1 Auswirkungen der Import-Brücke

Nachdem jeweils ein konzeptuelles Modell für die Projekte SpSemOnline (23 Dingtypen und 32 Zusammenhangstypen) und FoPra-Verwaltung (30 Dingtypen und 26 Zusammenhangstypen) aus den Projektanforderungen erstellt wurde, wurde auf diese Modelle die Import-Brücke angewendet. Zu diesem Zeitpunkt stand bereits eine Domänen-Ontologie[6] (77 Dingtypen und 55 Zusammenhangstypen) aus der Vorbereitung der Evaluierung zur Verfügung.

Insgesamt führte die Anwendung der Import-Brücke zu vier Veränderungen in dem konzeptuellen Modell des *SpSemOnline*-Projekts und zu sechs Veränderungen in dem Modell der *FoPra-Verwaltung*. Zwei Anpassungen im *SpSemOnline*-Modell und genau so viele in dem *FoPra-Verwaltung*-Modell stellen Fälle der Wiederverwendung dar. Dabei konnten mehrere

[4] Eine ausführliche Beschreibung der Fälle findet sich in [Hor10].

[5] Mit weiteren Entwicklungsstuffen der Domänen-Ontologie ist davon auszugehen, dass ihre Vollständigkein und Korrektheit zunimmt, und sie ihre Wirkung auf die Projektmodelle noch besser entfalten kann.

[6] Siehe auch Abbildung 4.17

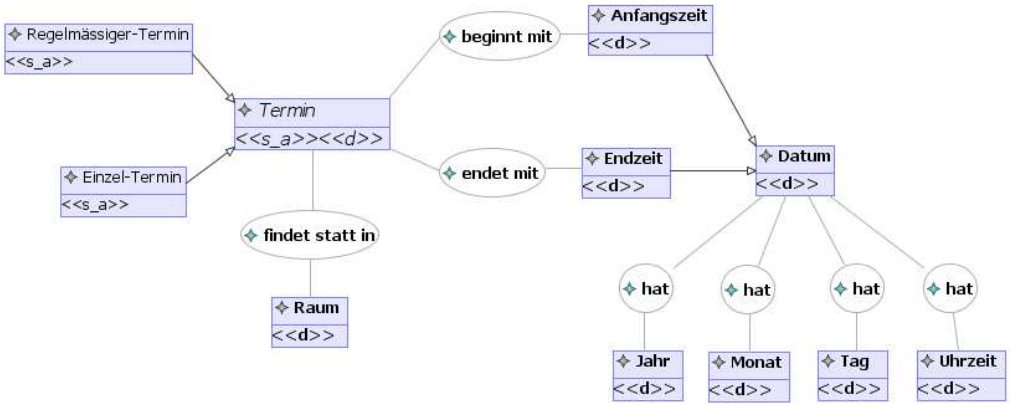


ABBILDUNG 7.3: Beispiel für die Wiederverwendung [Hor10]

Konzepte und Beziehungen aus der Domänen-Ontologie übernommen werden.

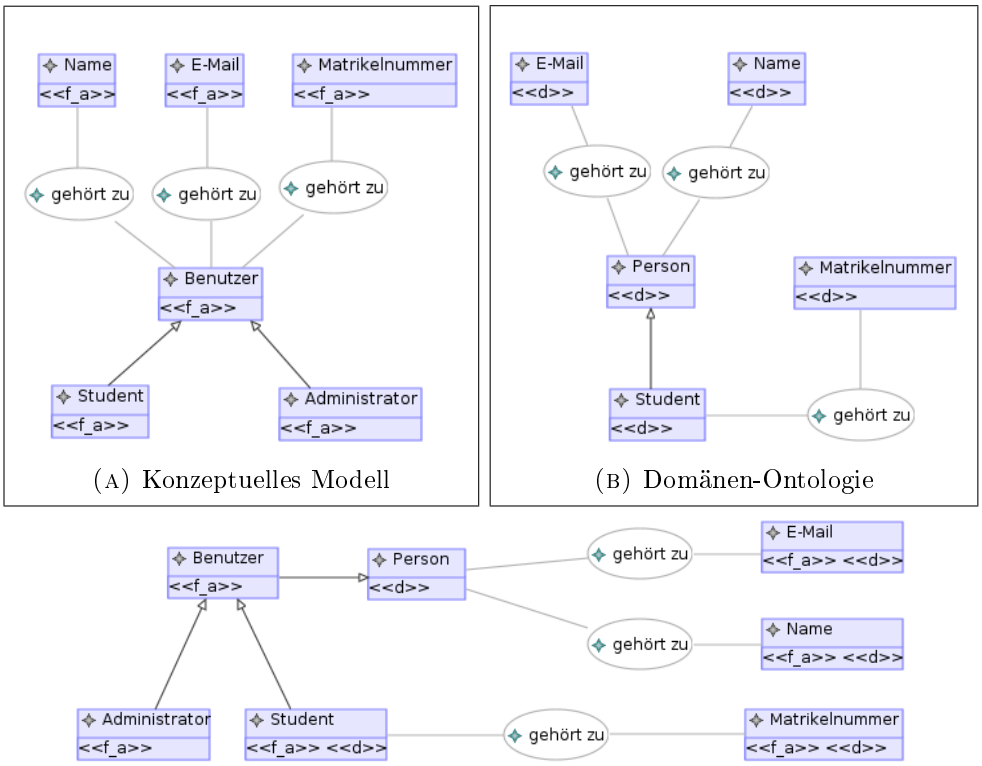
Ein gutes Beispiel dafür liefert das Konzept Termin (siehe Abbildung 7.3). So wurde der Dingtyp Termin im Falle von *SpSemOnline* um acht weitere Konzepte mit den zugehörigen Beziehungen erweitert, die für seine Umsetzung erforderlich sind[7].

Neben dem allgemein gebräuchlichen Konzept Termin wurden in der vorliegenden Universitätsdomäne noch weitere Konzepte wie Name oder Bezeichner in den Anforderungen nicht genauer spezifiziert. In einer weniger vertrauten Domäne als der untersuchten kann das zu einer falschen bzw. unvollständigen Umsetzung der Anforderungen führen. Die Import-Brücke hilft in diesen Situationen in zweierlei Hinsicht. Zunächst liefert sie Hinweise auf Lücken in den Anforderungen. Ergänzend ermöglicht sie eine bereits erprobte und von einem Domänen-Experten freigegebene[8] Modellierung des Sachverhalts in das Projekt zu übernehmen.

Weitere Anpassungen in den konzeptuellen Modellen der Projekte (zwei bei *SpSemOnline* und vier bei *FoPra-Verwaltung*) wurden auf Grund der qualitativen Verbesserungen dieser Modelle durchgeführt. Ein Beispiel dafür ist in Abbildung 7.4 dargestellt. Im Diagramm *(A)* steht ein Ausschnitt

[7] Buchstaben in den spitzen Klammern visualisieren den Ursprung eines Elements: s_a steht für die *SpSemOnline*-Anforderungen, d für die Domänen-Ontologie. Aus der Domänen-Ontologie hinzugefügte Elemente sind fett hervorgehoben.

[8] Diese Prüfung findet in der Revidierungs-Phase des Ontologie-Entwicklungszyklus statt (siehe Kapitel 5.2.2).



(A) Konzeptuelles Modell

(B) Domänen-Ontologie

(C) Resultierendes konzeptuelles Modell

ABBILDUNG 7.4: Verbesserung der Qualitätsmerkmalle Korrektheit und Erweiterbarkeit

des konzeptuellen Modells der *FoPra-Verwaltung*. Entsprechend den Anforderungen wurden hier die Konzepte Name, E-Mail und Matrikelnummer dem Konzept Benutzer zugeordnet. Der entsprechende Auschnitt der Domänen-Ontologie im Diagramm *(B)* zeigt, dass die Konzepte Name und E-Mail dem Konzept Person und das Konzept Matrikelnummer dem Konzept Student zugeordnet sind.

Mit Hilfe dieses Domänen-Ontologie-Auschnitts konnte das Anforderungsmodell der *FoPra-Verwaltung* bezüglich der Qualitätsmerkmalle Korrektheit und Erweiterbarkeit verbessert werden (Abbildung 7.4 (C)). Insbesondere die Verlagerung des Konzepts Matrikelnummer zum Student verbesserte die Modellqualität stark. Neben dem korrekten Gebrauch dieses Konzepts wurde die Erweiterbarkeit des Systems erhöht, da in dem resultierenden Modell weitere Personengruppen, die keine Matrikelnummer besitzen, als Benutzer agieren können. Außerdem können diesem Modell

weitere Personengruppen hinzugefügt werden, die keine Benutzer sind.

Weitere Beispiele für die Veränderungen auf Grund der Modellqualität führten außerdem zu Verbesserung der konzeptuellen Integrität, indem einige Konzepte in den resultierenden konzeptuellen Modellen richtig benannt wurden oder vermeintliche Synonyme wie Dozent und Professor anhand von zusätzlichen Informationen aus der Domänen-Ontologie auseinander gehalten werden konnten.

Projekt	**vor dem Import**		**nach dem Import**	
	TT	CT	TT	CT
SpSemOnline	23	32	44	37
FoPra-Verwaltung	30	26	41	36

TABELLE 7.1: Quantitative Veränderung der konzeptuellen Modelle

Tabelle 7.1 zeigt, wie sich die Größe der Modelle nach der Anwendung der Import-Brücke geändert hat. TT steht in dieser Tabelle für die Anzahl der Dingtypen und CT für die Anzahl der Zusammenhangstypen. Den größten Zuwachs verzeichnete das Projekt *SpSemOnline*. Das kann vor allem darauf zurückgeführt werden, dass dieses Projekt ohne Verwendung eines Vorgehensmodells durchgeführt wurde und seine Anforderungsbeschreibung durch fehlende Formalisierung viele Lücken aufwies. Aber auch ein nach den Vogehensmodellen *OpenUP* und *Scrum* durchgeführtes Projekt *FoPra-Verwaltung* konnte deutlich von der Anwendung der Import-Brücke profitieren.

Zusammenfassend betrachtet kann die Import-Brücke neben ihrer primären Funktion als Mittel zum Wissenstransfer von der Domänen-Ontologie zu den Projekten auch als eine in den Prozess integrierte konstruktive Qualitätssicherungsmaßnahme betrachtet werden. Die Evaluierung hat gezeigt, dass die Import-Brücke in dieser Funktion vor allem die Korrektheit, Erweiterbarkeit und konzeptuelle Integrität eines konzeptuellen Modells verbessert. Ein interessantes Ergebnis der Evaluierung an dieser Stelle war die Beobachtung, dass mehr als die Hälfte aller Modellanpassungen mit qualitativen Verbesserungen verbunden waren. Das zeigt, dass Wissenstransfer nicht allein anhand von Elementen-Übernahme stattfindet, sondern überwiegend in der Form der erprobten und geprüften Modellmuster.

7.3.2 Vergleich der konzeptuellen Modelle

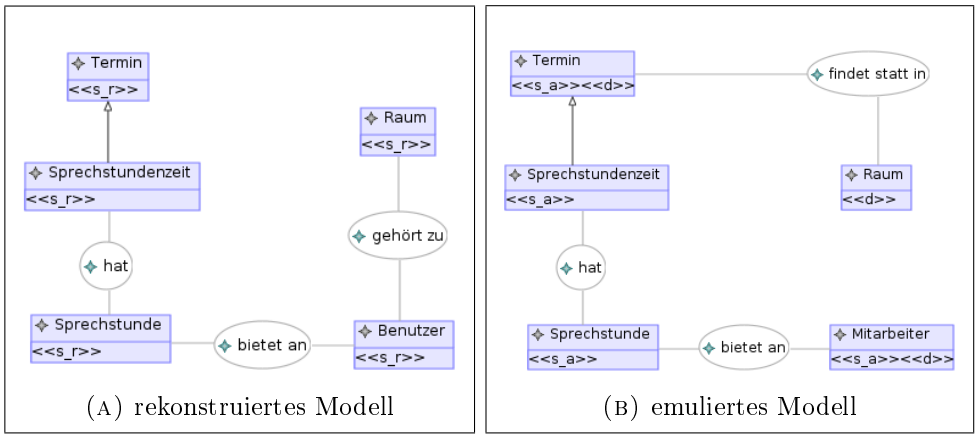
In diesem Schritt des Evaluierungsprozesses sollen Ergebnisse der *OBSE*-Emulation mit den Ergebnissen der durchgeführten Projekte *SpSemOnline* und *FoPra-Verwaltung* verglichen werden. Der Vergleich wurde ebenfalls auf der Ebene der konzeptuellen Modelle durchgeführt (siehe Seite 180). Das aus dem *PHP*-Code gewonnene Modell des Projekts *SpSemOnline* beinhaltet insgesamt 58 Dingtypen und 51 Zusammenhangstypen. Das aus dem *UML*-Klassen-Diagramm mit Hilfe der CPL2UML-Transformation gewonnene Modell der *FoPra-Verwaltung* besteht aus 37 Dingtypen und 40 Zusammenhangstypen. Der Umfang des *FoPra-Verwaltung*-Modells entspricht weitgehend seinem emulierten Pendant[9]. Der signifikante Unterschied bei dem *SpSemOnline* ist einerseits auf die fehlende Objekt-orientierung bei der Nutzung der Programmiersprache *PHP* zurückzuführen. Andererseits wurde bei diesem Projekt deutlich, dass die Anforderungen und die Implementierung sich stark voneinander unterscheiden. So wurden neue Elemente bei Bedarf während der Implementierung ergänzt.

Generell wurde bei diesem Vergleich festgestellt, dass die Konzepte, die während der Emulation mittels der Import-Brücke in die Anforderungsmodelle importiert wurden, in den rekonstruierten Modellen ebenfalls gefunden werden konnten. Somit waren sie für die Umsetzung dieser Projekte erforderlich. Vorteil des *OBSE*-Prozesses lag in diesem Fall darin, dass sie bereits in einer frühen Phase der Software-Entwicklung ergänzt wurden und ihre Modellierung mit den Fachverantwortlichen abgestimmt werden konnte. Bei einer späteren, während der Implementierung durchgeführten Integration der fehlenden Konzepte wurde zum Teil eine qualitativ schlechtere Modellierung dieser Konzepte beobachtet.

Ein gutes Beispiel dafür liefert das Konzept Raum des *SpSemOnline*-Projekts. Während der Emulation wurde dieses Konzept aus der Domänen-Ontologie importiert und dem Konzept Termin zugeordnet (Abbildung 7.3 und 7.5 rechts). In dem realen Projekt wurde die Implementierung ebenfalls um das Konzept Raum ergänzt. Allerdings wurde Raum dem Konzept Benutzer zugeordnet (Abbildung 7.5 links).

Auf diese Weise sind in dem Projekt *SpSemOnline* zwei homonyme Konzepte Raum zusammengefallen. Erstens ein Raum eines Benutzers bzw. Mitarbeiters des Fachbereichs und zweitens ein Raum, in dem ein Termin stattfindet. Diese Modellierung impliziert, dass die Sprechstunden immer in dem Raum des Benutzers stattfinden, was nicht immer der Realität ent-

[9] Siehe Spalte nach dem Import in der Tabelle 7.1.



(A) rekonstruiertes Modell (B) emuliertes Modell

ABBILDUNG 7.5: Einordnung des Konzepts Raum

sprechen kann. Aus diesem Grund stellt die Modellierung in dem emulierten konzeptuellen Modell eine deutlich flexiblere Lösung zur Verwaltung von Sprechstunden dar. Bezüglich der Qualitätsmerkmale ist diese Lösung besser wart- und wiederverwendbar, da sie keine versteckten Zusammenhänge beinhaltet, die ihrer Wiederverwendung oder nachträglichen Änderung in dem Weg stehen.

Neben der Änderbarkeit wirkt sich die Modellierung des Konzepts Raum in dem rekonstruierten Modell ebenfalls auf den Qualitätsmerkmal Verfolgbarkeit aus. Während im *OBSE*-Prozess die Quelle des Elements die Domänen-Ontologie ist, taucht das Konzept Raum in dem rekonstruierten Modell einfach auf.

Vergleichbare Fälle konnten beim Konzept Name beobachtet werden. Oft wird dieses Konzept in den Anforderungen nicht genau spezifiziert. In dem emulierten Modell wird Name nach einem Import um die Konzepte Vorname und Nachname ergänzt. In dem rekonstruierten Modell sind sie sowohl in dem Projekt *SpSemOnline* als auch *FoPra-Verwaltung* zu finden. Allerdings werden diese Konzepte in beiden Projekten unterschiedlich umgesetzt. In *SpSemOnline* wurden Bestandteile dem Konzept Name zugewiesen. In der *FoPra-Verwaltung* wurde auf die Verwendung vom Konzept Name verzichtet und seine Bestandteile wurden dem Konzept Person direkt zugeordnet. Diese unterschiedliche Modellierung eines gleichen Sachverhalts der Domäne widerspricht dem Qualitätsmerkmal der konzeptuellen Integrität. Außerdem wurden im Projekt *SpSemOnline* einem Namen weitere Bestandteile wie Anrede und Titel zugeordnet, was aus konzeptueller Sicht problematisch ist. Solche Modellierung widerspricht auch den Qualitätsmerk-

malen Präzision und Korrektheit. Das gilt auch für die *FoPra-Verwaltung*, da in ihrem Modell das in den Anforderungen geforderte Konzept *Name* ausgelassen wurde.

Ein weiteres Beispiel dafür, dass ein nach *OBSE* entstandenes konzeptuelles Modell im Bezug auf die Qualitätsmerkmale Änderbarkeit und Verfolgbarkeit besser abschneidet, liefert die Modellierung einer Ankündigungs-E-Mail aus dem Projekt *FoPra-Verwaltung*. Im Unterschied zu dem emulierten fehlte in dem rekonstruierten konzeptuellen Modell die entsprechende Modellierung. Auf den ersten Blick deutete das darauf hin, dass das Versenden von Ankündigungs-E-Mails nicht umgesetzt wurde. Eine genauere Analyse des *FoPra-Verwaltungs*-Codes zeigte, dass Ankündigungs-E-Mails indirekt, verteilt auf mehrere Operationen umgesetzt wurde. Im Allgemeinen gelten solche Realisierungen als schlecht wart- und erweiterbar. Außerdem können sie selbst nicht wiederverwendet werden und erschweren eine Wiederverwendung von beteiligten Operationen.

In dieser Untersuchung anhand der realen Projekte konnte beobachtet werden, dass die Anforderungen durchaus Lücken aufweisen oder die Domäne ungenau beschreiben können. Das führt zwangsläufig dazu, dass ein aus den Anforderungen entstandenes Modell während der Projekt-Umsetzung vervollständigt und angepasst wird. Das konnte in den beiden Projekten *SpSemOnline* und *Fopra-Verwaltung* beobachtet werden. Ohne *OBSE*, so wie die Projekte tatsächlich umgesetzt worden waren, führte dies zu verschiedenen Modellierungen von gleichen Sachverhalten, deren Ursprung nicht mehr festgestellt werden konnte. Die neuen fehlenden Konzepte wurden zum Teil erst im Code eingeführt. Die Simulation des *OBSE*-Prozesses für diese Projekte zeigte dagegen, dass diese Lücken und problematischen Stellen mit *OBSE* bereits auf einer höheren Ebene ausgeräumt werden konnten. Dabei wurde in der Regel die bestehende Modellierung aus der Domänen-Ontologie übernommen, wodurch sowohl Zeit für die Entwicklung eigener Lösungen erspart als auch die Qualität dieser Modelle verbessert werden konnte.

8 Zusammenfassung

8.1 Fazit

In dieser Arbeit wurde ein Prozess zur *Ontologie-basierten Software-Entwicklung* ausgearbeitet. Als erstes wurde eine Infrastruktur entworfen, welche die Anbindung einer Domänen-Ontologie an die typische Software-Entwicklung ermöglicht. Um verschiedene in die Infrastruktur integrierte Techniken unter ein Dach zu bringen, wurde ein Meta-Modell-basierter Aufbau mit *EMOF/Ecore* auf der obersten Ebene ausgewählt. Diese Vorgehensweise ermöglichte es, Editoren für die Sprachen *CPL* und *UML* einheitlich zu entwickeln und Transformationen entsprechend dem *MOF QVT*-Standard zu definieren.

Um eine Integration der konzeptuellen Modellierungssprache *CPL* in die *OBSE*-Infrastruktur zu ermöglichen, wurde im Rahmen dieser Arbeit für sie ein *EMOF/Ecore*-konformes Meta-Modell erstellt. Gleichzeitig wurde an der Verbesserung der Sprache gearbeitet, indem problematisch umgesetzte Sprachkonstrukte wie zum Beispiel Generalisierung neu definiert wurden. Auch die Tauglichkeit von *CPL* als eine Ontologie-Sprache wurde analysiert und ihr Meta-Modell wurde für diesen Zweck angepasst. Dadurch wurde erreicht, dass sowohl für die Domänen-Ontologie als auch für die konzeptuellen Modelle der Projekte bis auf kleine Modifikationen die gleiche Sprache verwendet wurde. Zusätzliche Transformationen zu einer weiteren Ontologie-Sprache wurden auf diese Weise vermieden, die *OBSE*-Infrastruktur und zugehörigen Prozesse konnten dadurch kompakter gestaltet werden.

Ausgehend von den Infrastruktur-Elementen wurde in dieser Arbeit ein zyklischer Prozess zur Ontologie-Entwicklung vorgestellt. Als Teil dieses Prozesses wurden neue Rollen eingeführt und mit den zugehörigen Aufgaben verknüpft. Der *OBSE*-Prozess unterscheidet sich von den typischen Vor-

gehensmodellen dadurch, dass er selbst auf einem anderen Vorgehensmodell aufbaut, das auf der Projektseite für die eigentliche Software-Entwicklung verwendet wird. In dieser Arbeit wurden *OBSE*-Prozesse in das *EOS*-Vorgehensmodell integriert und ergänzend dazu wurde eine Einbindung in das agile Vorgehensmodell *Scrum* skizziert.

Für die Unterstützung des *OBSE*-Prozesses wurde das *OBSE*-Tool entwickelt. Seine Architektur wurde PlugIn-basiert aufgebaut, was eine verteilte Entwicklung und eine spätere Integration der funktionalen Bestandteile ermöglichte. In der aktuellen Version erlaubt das Tool bereits das Verwalten und Bearbeiten von *CPL* und *UML*-Modellen, Transformationen zwischen diesen sowie das Anlegen von Projekten für die Software- und Ontologie-Entwicklung. Für die vollständige Unterstützung des *OBSE*-Prozesses fehlt noch eine Umsetzung der Brücken.

Die durchgeführte Evaluation hat wichtige Erkenntnisse über den *OBSE*-Prozess geliefert und die Erwartung bestätigt, dass mit Hilfe einer Domänen-Ontologie Wissen zwischen den beteiligten Projekten in dieser Domäne ausgetauscht werden kann. Eine signifikante Anzahl von Konzepten und Beziehungen konnte während des Imports aus dem Projekt Scheinverwaltung in die Projekte SpSemOnline und FoPra-Verwaltung übertragen werden. Zusätzlich zu der direkten Wiederverwendung wurde in dem Evaluierungsprozess beobachtet, dass mit Hilfe der Import-Brücke die Qualität der konzeptuellen Modelle insbesondere im Bezug auf das Modellqualitätsmerkmal Änderbarkeit verbessert werden konnte.

8.2 Ausblick

Im *OBSE*-Prozess wird ein breites Spektrum an aktuellen Techniken eingesetzt, die einen Spielraum für weitere Entwicklung lassen. So können sowohl weitere Vorgehensmodelle an die Ontologie-Entwicklung angebunden werden als auch die Erkenntnisse aus der aktuellen Ontologie-Forschung im Ontologie-Entwicklungsprozess berücksichtigt werden. Im Folgenden werden Ideen für zukünftige Arbeiten entsprechend der Gliederung des Infrastruktur-Kapitels dieser Arbeit präsentiert. Anschließend werden Vorschläge für den möglichen Ausbau des *OBSE*-Tools skizziert.

Weiterentwicklung der Sprache *CPL*

Ein Sprachkonstrukt, das *CPL* bei ihrem Einsatz als eine Ontologie-Sprache noch fehlt, ist die Komposition von Ontologien. Mit ihrer Hilfe wäre es möglich, eine hierarchische Ontologie unter Verwendung einer anderen aufzubauen. Einige Ontologie-Sprachen wie *OWL* realisieren dies mit Hilfe von Namensräumen und erlauben es, eine Ontologie in eine andere zu importieren, da ihre Elemente durch Namensräume eindeutig gekenzeichnet sind. Dieses Konzept kann für die Sprache *CPL* übernommen werden. Diese Spracherweiterung würde helfen, Domänen-Ontologien zu kombinieren und im *OBSE*-Prozess gemeinsam zu benutzen.

Zusätzlich kann für die Sprache *CPL* ein *UML*-Profil erstellt werden. Das Vorhandensein dieses Profils würde keine direkte Auswirkung auf den *OBSE*-Prozess haben, sondern vielmehr für die Verbreitung der Sprache *CPL* sorgen. Mit Hilfe des *UML*-Profils können *CPL*-Modelle dann auch mit *UML*-Werkzeugen erstellt und bearbeitet werden. Außerdem werden aktuell – auf Grund der starken Verbreitung von *UML* – viele Verfahren für Modellmanipulationen auf der Basis von UML-Profilen entwickelt, die dann gegebenenfalls in dem *CPL*-Umfeld eingesetzt werden können.

Ausbau des *OBSE*-Prozesses auf der Projektseite

Die Liste der Vorgehensmodelle, die mit *OBSE* zusammenarbeiten, kann weiter ausgebaut werden, um Projekten, die nach diesen Vorgehensmodellen entwickelt werden, einen Zugang zu der Domänen-Ontologie zu ermöglichen. Die Schnittstelle zwischen der Ontologie-Entwicklung auf der einen und Software-Entwicklung auf der anderen Seite wurde in *OBSE* mit Import- und Export-Brücken gezielt klein gehalten. Ein weiteres Vorgehensmodell muss somit nur an den Stellen angepasst werden, wo diese Brücken verwendet werden. Die dafür erforderliche Integration des konzeptuellen Modells wird – im Falle der Objekt-orienterten Vorgehensweise mit *UML*-Modellierung – bereits durch den *OBSE*-Prozess mit Hilfe der Transformationen zwischen den Sprachen *CPL* und *UML* unterstützt.

Der Übergang von dem konzeptuellen zu dem Analyse-Modell ist eine weitere Stelle, an der es sich lohnen würde, den *OBSE*-Prozess zu erweitern. Denkbar wären zusätzliche Transformationen zu den Modellierungssprachen wie *Business Process Execution Language (BPEL)* und *XML Process Definition Language (XPDL)*[1], um die Entwicklung von Prozess- und Service-

[1]Bei den Sprachen *BPEL* und *XPDL* handelt es sich um sich ergänzende Sprachen,

basierten Software-Systemen zu unterstützen. Auf diese Weise können diese Systeme sowohl als Quelle für die Informationen über eine Domäne dienen als auch von der statischen Modellierung in der Domänen-Ontologie profitieren.

Die Entwicklung von weiteren Transformationen ist nicht nur mit Hinblick auf moderne Techniken interessant. In vielen Unternehmen werden sehr oft Software-Lösungen eingesetzt, die vor Jahrzehnten mit damaligen Technologien entwickelt wurden und für die heutzutage kaum Spezialisten zu finden sind. Im Banken-Sektor sind zum Beispiel *Cobol*-Programme verbreitet, welche die Abwicklung wichtiger Geschäftsprozesse übernehmen. Für die Migrationsprojekte wäre es sicherlich von Vorteil, Informationen über die Domänen dieser Geschäftsprozesse zu gewinnen. Eine Transformation zwischen *Cobol* und *CPL* vorausgesetzt, kann der *OBSE*-Prozess an dieser Stelle für den Wissenstransfer von den „alten“ zu den „neuen“ Projekten sorgen.

Ausbau des *OBSE*-Prozesses auf der Ontologie-Seite

Die Verwendung von Domänen-Ontologien stellt einen guten Ansatzpunkt für die konstruktiven Qualitätsverbesserungsmaßnahmen in dem *OBSE*-Prozess dar. Verbesserungen an dieser Stelle werten nicht nur dieses Artefakt auf, sondern werden über die Import-Brücke an die beteiligten Projekte weitergegeben. Die in dieser Arbeit beschriebenen Prozesse zur Herstellung einer Domänen-Ontologie können um weitere Schritte zur Qualitätsherstellung ergänzt werden. Zum Beispiel kann die Domänen-Ontologie in der Revisionsphase mit Hilfe des Verfahrens von Guizzardi [Gui05] auf ontologische Fehler untersucht werden.

Im Hinblick auf die agile Software-Entwicklung können Auswirkungen einer Erweiterung des gemeinsamen Besitzes auf die Domänen-Ontologie untersucht werden. In diesem Fall würden die Projektbeteiligten zusätzlich die Rolle eines Ontologie-Entwicklers übernehmen und ihre Fragmente direkt in die Domänen-Ontologie integrieren. Diese weitreichende Veränderung in der Entwicklung der Domänen-Ontologie würde zu einem Wiki-artigen Entwicklungsprozess führen, der bereits in mehreren Wissensaustauschplattformen eingesetzt wird, deren bekanntester Vertreter *Wikipedia* ist.

deren Modelle *Web-Services* beschreiben. Diese Modelle können durch eine spezielle *Workflow-Maschine* ausgeführt werden und somit definierte Dienste anbieten.

Prozessunterstützung durch das *OBSE*-Tool

Um die Unterstützung des *OBSE*-Prozesses zu vervollständigen, sollte das *OBSE*-Tool um das PlugIn für die Integration von *CPL*-Modellen erweitert werden. Das Integrationsverfahren wird zur Zeit in Klagenfurt entwickelt [BV09]. Der nächste Schritt ist die Umsetzung dieses Verfahrens als *Eclipse*-PlugIn. An dieser Stelle kann *Atlas Model Weaver (AMW)*[2] aus dem *Eclipse Modeling Framework* als Teil der Implementierung eingesetzt werden, um die Entwicklung des PlugIns zu beschleunigen.

Der grafische Editor kann weiter ausgebaut werden, indem für weitere Elemente des Meta-Modells Diagramm-Elemente entwickelt und umgesetzt werden. Ein guter Kandidat dafür ist das Meta-Modell-Element Perspective-Determiner. Dieser Ausbau würde es ermöglichen, Rollen, die Perspektiven in einer Beziehung spielen, grafisch zu visualisieren.

Der bisherige Schwerpunkt bei der Entwicklung des *OBSE*-Tools war auf die Funktionalität ausgerichtet, um die Umsetzbarkeit zu zeigen. Die nächste Ausbaustufe des Tools kann dazu benutzt werden, die Benutzerfreundlichkeit zu erhöhen. Dafür können Editoren besser miteinander verknüpft werden. Zusätzlich können in die beiden Editoren Filter eingebaut werden, die bestimmte Elemente der Modelle ausblenden können, um die Übersichtlichkeit zu erhöhen.

[2] http://www.eclipse.org/gmt/amw/

Anhang

A Ursprügliches *CPL*-Meta-Modell

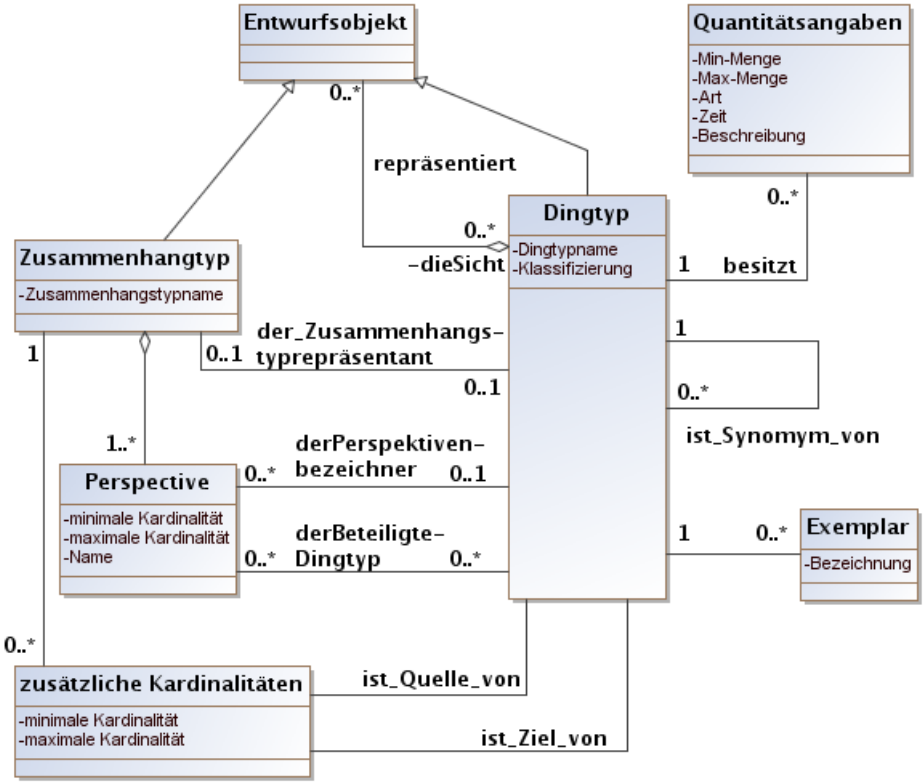


Abbildung A.1: Ausschnitt des ursprüglichen *CPL*-Meta-Modells für die statische Modellierung

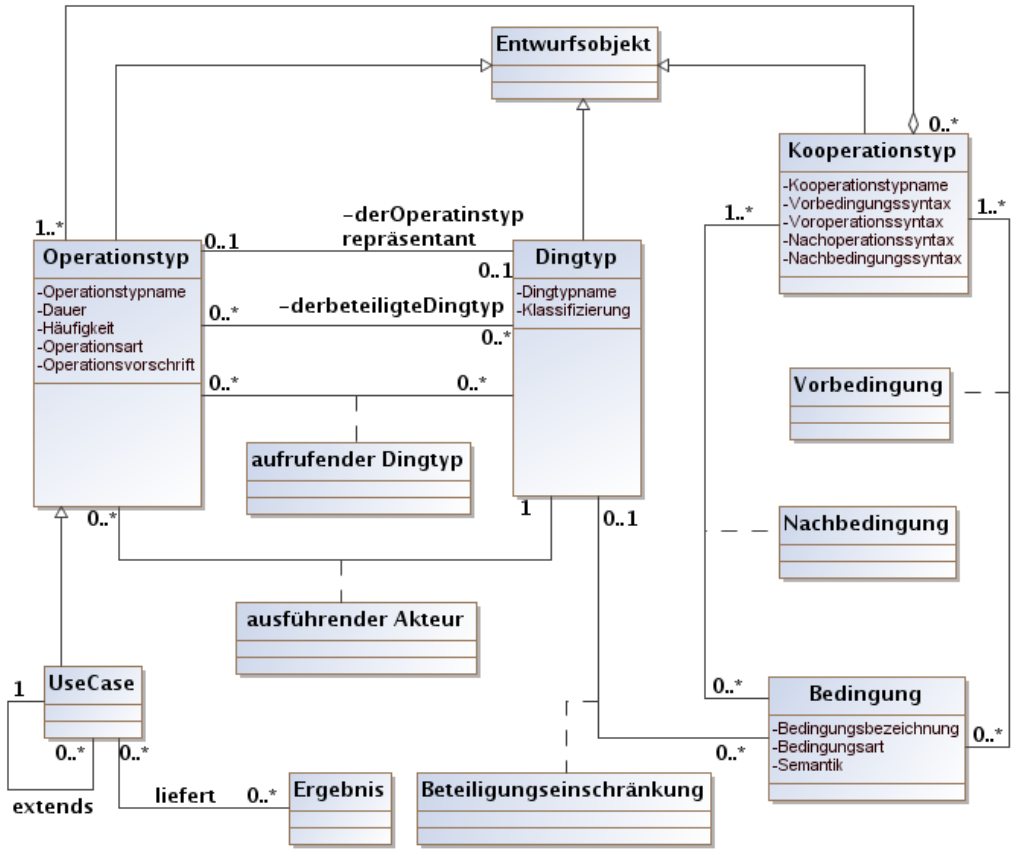


Abbildung A.2: Ausschnitt des ursprüglichen *CPL*-Meta-Modells für die dymamische Modellierung

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AMS07] Timo Asikainen, Tomi Männistö und Timo Soininen: *Kumbang: A domain ontology for modelling variability in software product families.* Advanced Engineering Informatics, 21(1):23 – 40, 2007.

[And08] Nils Andersch: *Modularisierung von Ontologien in der Softwareentwicklung.* Diplomarbeit, Philipps-Universität Marburg, 2008.

[Bac08] Andrej Bachmann: *Methoden- und Werkzeugunterstützung für ontologiebasierte Software-Entwicklung (OBSE).* In: Thomas Kühne, Wolfgang Reisig und Friedrich Steimann (Herausgeber): *Modellierung*, Band 127 der Reihe *LNI*, Seiten 217–220. GI, 2008.

[BBvB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland und Dave Thomas: *Manifesto for Agile Software Development*, 2001. `http://www.agilemanifesto.org/`.

[BEK+06] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer und Eduard Weiss: *Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework.* In: *Proceedings of Model Driven Engineering Languages and Systems (MoDELS'06)*, Band 4199 der Reihe *LNCS*, Seiten 425–439, 2006.

[BHR07a] Andrej Bachmann, Wolfgang Hesse und Aaron Ruß: *Coupling ontology and software development processes - a rendez-vous approach.* In: C.R.G. Farias G. Guizzardi (Herausgeber): *2nd Brazilian Workshop on Ontologies and Metamodels for Software and Data Engineering (WOMSDE)*, LNCS. Springer, 2007.

[BHR+07b] Andrej Bachmann, Wolfgang Hesse, Aaron Ruß, Christian Kop, Heinrich C. Mayr und Jürgen Vöhringer: *A Practical Approach to Ontology-based Software Engineering.* In: Manfred Reichert, Stefan Strecker und Klaus Turowski (Herausgeber): *EMISA*, Band P-119 der Reihe *LNI*, Seiten 129–142. GI, 2007.

[BI96] Barry Boehm und Hoh In: *Identifying Quality-Requirement Conflicts.* IEEE Softw., 13(2):25–35, 1996.

[BJRV03] Jean Bézivin, Frédéric Jouault, Peter Rosenthal und Patrick Valduriez: *ATL Transformation-based Model Management Framework.* In: *Management Framework, Research Report, Atlas Group, INRIA and IRIN*, 2003.

[BKK+02] Kenneth Baclawski, Mieczyslaw M. Kokar, Paul A. Kogut, Lewis Hart, Jeffrey E. Smith, Jerzy Letkowski und Pat Emery: *Extending the Unified Modeling Language for ontology development.* Software and System Modeling, 1(2):142–156, 2002.

[BLHL01] Tim Berners-Lee, James Hendler und Ora Lassila: *The Semantic Web.* Scientific American, May 2001.

[BMN+07] Jorge Calmon de Almeida Biolchini, Paula Gomes Mian, Ana Candida Cruz Natali, Tayana Uchôa Conte und Guilherme Horta Travassos: *Scientific research ontology to support systematic review in software engineering.* Advanced Engineering Informatics, 21(2):133–151, 2007.

[BSM+03] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick und Timothy J. Grose: *Eclipse Modeling Framework.* Addison Wesley Professional, 2003.

[Bun77] Mario Bunge: *Ontology I: The Furniture of the World.* In: *Treatise on Basic Philosophy*, Seiten 15–19, Berlin, Heidelberg, 1977. Springer.

[BV09] Peter Bellström und Jürgen Vöhringer: *Towards the Automation of Modeling Language Independent Schema Integration.* In: *Proceedings of the International Conference on Information, Process, and Knowledge Management eKNOW '09*, Seiten 110–115. Springer, 2009.

[BVEL04] Sara Brockmans, Raphael Volz, Andreas Eberhart und Peter Löffler: *Visual modeling of OWL DL ontologies using UML.* In: *The Semantic Web – ISWC 2004*, Seiten 198–213. Springer, 2004.

[BW05] Steffen Bleul und Thomas Weise: *An Ontology for Quality-Aware Service Discovery.* In: *In Proc. First International Workshop on Engineering Service Compositions (WESC'05*, Seiten 35–42, 2005.

[Car00] Joseph J. Carr: *Requirements engineering and management: the key to designing quality complex systems.* The TQM Magazine, 12(6):400–407, 2000.

[CFLGP03] Oscar Corcho, Mariano Fernández-López und Asunción Gómez-Pérez: *Methodologies, tools and languages for building ontologies: where is their meeting point?* Data Knowl. Eng., 46(1):41–64, 2003.

[CH03] Krzysztof Czarnecki und Simon Helsen: *Classification of Model Transformation Approaches*. In: *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[Che76] Peter Pin Shan Chen: *The Entity-Relationship Model: Toward a Unified View of Data*. ACM Transactions on Database Systems, 1:9–36, 1976.

[CP07] Stephen Cranefield und Jin Pan: *Bridging the gap between the model-driven architecture and ontology engineering*. International Journal of Human-Computer Studies, 65(7):595 – 609, 2007.

[CRP06] Coral Calero, Francisco Ruiz und Mario Piattini: *Ontologies for Software Engineering and Software Technology*. Springer, Berlin, Heidelberg, 2006.

[DeM79] Tom DeMarco: *Structured analysis and system specification*. Yourdon computing series. Yourdon, Upper Saddle River, NJ, 1979.

[Dij72] Edsger W. Dijkstra: *The Humble Programmer*. ACM, 15(10):859–866, October 1972.

[DRR+05] Björn Decker, Eric Ras, Jörg Rech, Bertin Klein und Christian Hoecht: *Self-organized Reuse of Software Engineering Knowledge supported by Semantic Wikis*. In: *In Workshop on Semantic Web Enabled Software Engineering (SWESE), held at the 4th International Semantic Web Conference ISWC*, 2005.

[DSS93] R. Davis, H. Shrobe und P. Szolovits: *What is knowledge representation?* AI Magazine, 14(1):17–33, 1993.

[EEE+07] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann und Gabriele Taentzer: *Information Preserving Bidirectional Model Transformations*. In: *FASE*, Seiten 72–86, 2007.

[Elb10] Ralf Elbert: *Eclipse RCP Buch*, 2010. `http://www.ralfebert.de/rcpbuch/`.

[FDP+05] Tim Finin, Li Ding, Rong Pan, Anupam Joshi, Pranam Kolari, Akshay Java und Yun Peng: *Swoogle: Searching for knowledge*

on the Semantic Web. In: *In AAAI 05 (intelligent systems demo*, Seiten 1682–1683. The MIT Press, 2005.

[FG02] Ricardo De Almeida Falbo und Giancarlo Guizzardi: *An Ontological Approach to Domain Engineering.* In: *In Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, Seiten 351–358. ACM Press, 2002.

[FH06] Mario Friske und Konrad Hilse: *Evaluation von Transformationsmaschinen in der modellbasierten Qualitiätssicherung.* In: Christian Hochberger und Rüdiger Liskowsky (Herausgeber): *GI Jahrestagung*, Band 94 der Reihe *LNI*, Seiten 205–209. GI, 2006.

[FHR08] Florian Fieber, Michaela Huhn und Bernhard Rumpe: *Modellqualität als Indikator für Softwarequalität: eine Taxonomie.* Informatik Spektrum, 31(5):408–424, 2008.

[FKM+00] Günther Fliedl, Christian Kop, Heinrich C. Mayr, Willi Mayerthaler und Christian Winkler: *Linguistically Based Requirements Engineering: The Niba-Project*, 2000.

[FKM+02] Günther Fliedl, Christian Kop, Heinrich C. Mayr, Willi Mayerthaler und Christian Winkler: *The NIBA Workflow. From textual requirements specifications to UML-schemata*, 2002.

[FLGP02] Mariano Fernández-López und Asunción Gómez-Pérez: *Overview and analysis of methodologies for building ontologies.* Knowl. Eng. Rev., 17(2):129–156, 2002.

[Fow04] Martin Fowler: *UML konzentriert.* Addison-Wesley, München, 3. Auflage, 2004, ISBN 3-8273-2126-3.

[FS02] Elizabeth Furtado und Kênia Soares Sousa: *An Ontology-Based Method for Universal Design of User Conceptual Interfaces Using Scenarios.* In: Costin Pribeanu und Jean Vanderdonckt (Herausgeber): *TAMODIA*, Seiten 25–31. INFOREC Publishing House Bucharest, 2002.

[GDD06] Dragan Gaševic, Dragan Djuric und Vladan Devedžic: *Model Driven Architecture and Ontology Development.* Springer, Berlin, Heidelberg, 1. Auflage, 2006.

[GG95] Nicola Guarino und P. Giaretta: *Ontologies and Knowledge Bases: Towards a Terminological Clarification.* Towards Very Large Knowledge Bases: Knowledge Building and Knowledge

Sharing, Seiten 25–32, 1995.

[GGKH03] Tracy Gardner, Catherine Griffin, Jana Koehler und Rainer Hauser: *A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard*, July 2003.

[GMZ07] Fausto Giunchiglia, Maurizio Marchese und Ilya Zaihrayeu: *Encoding Classifications into Lightweight Ontologies.* J. Data Semantics, 8:57–81, 2007.

[GN87] Michael R. Genesereth und Nils J. Nilsson: *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann Publishers, 1987.

[Gru93a] Thomas R. Gruber: *A Translation Approach to Portable Ontology Specifications.* Knowledge Acquisition, 5:199–220, 1993.

[Gru93b] Thomas R. Gruber: *Toward Principles for the Design of Ontologies Used for Knowledge Sharing.* In: *Formal Ontology in Conceptual Analysis and Knowledge Representation.* Kluwer Academic Publishers, 1993.

[Gua95] Nicola Guarino: *Formal ontology, conceptual analysis and knowledge representation.* International Journal of Human and Computer Studies, 43(5-6):625–640, 1995.

[Gua97] Nicola Guarino: *Understanding, building and using ontologies.* International Journal of Human-Computer Studies, 46(2-3):293–310, 1997.

[Gua98] Nicola Guarino: *Formal Ontology and Information Systems.* In: *Proceedings of the 1st International Conference on Formal Ontologies in Information Systems (FOIS'98)*, Seiten 3–15, Trento, Italy, June 1998. IOS Press.

[Gua99] Nicola Guarino: *The Role of Identity Conditions in Ontology Design.* In: *COSIT '99: Proceedings of the International Conference on Spatial Information Theory: Cognitive and Computational Foundations of Geographic Information Science*, Seiten 221–234, London, UK, 1999. Springer-Verlag.

[Gui05] Giancarlo Guizzardi: *Ontological foundations for structural conceptual models.* CTIT PhD Thesis Series, 05-74, 2005.

[HB08] Lars Hechmann und Khalid Ballafkir: *SpeSemOnline – Projektdokumentation.* Technischer Bericht, Philipps-Universität Marburg, 2008.

[HC67] Peter Haggett und Richard J. Chorley: *Models in geography.*

Methuen; distributed in the U.S.A. by Barnes Noble, London, 1967.

[Hes96] Wolfgang Hesse: *Theory and practice of the software process - a field study and its implications for project management.* In: C. Montangero (Herausgeber): *Software Process Technology, 5th European Workshop, EWSPT 96*, Band 141 der Reihe *LNCS*, Seiten 241–256. Springer, 1996.

[Hes01] Wolfgang Hesse: *Dinosaur Meets Archaeopteryx? Seven Theses on Rational's Unified Process (RUP).* In: *Proc. CAiSE'98/LFIP 8.1 Int. Workshop on Evaluation of Modelling Methods in System Analysis and Design (EMMSAD'01)*, 2001.

[Hes05] Wolfgang Hesse: *Ontologies in the Software Engineering Process.* In: Richard Lenz, Ulrich Hasenkamp, Wilhelm Hasselbring und Manfred Reichert (Herausgeber): *EAI*, Band 141 der Reihe *CEUR Workshop Proceedings*, 2005.

[HK04] Wolfgang Hesse und Barbara Kzensk: *Ontologien in der Softwaretechnik.* In: *Proc. Workshop Ontologien in der und für die Softwaretechnik bei der Modellierung 2004*, 2004.

[HM08] Wolfgang Hesse und Heinrich C. Mayr: *Modellierung in der Softwaretechnik: eine Bestandsaufnahme.* Informatik Spektrum, 31(5):377–393, 2008.

[Hor10] Benjamin Horst: *Evaluierung von Software-Prozessen zur Ontologie-basierten Software-Entwicklung.* Diplomarbeit, Philipps-Universität Marburg, 2010.

[JK06] Frédéric Jouault und Ivan Kurtev: *On the architectural alignment of ATL and QVT.* In: *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, Seiten 1188–1195, New York, NY, USA, 2006. ACM.

[Ker01] Arno Kersche: *Mapping KCPM To A UML-Modell.* Diplomarbeit, Alpen-Adria-Universität Klagenfurt, 2001.

[KMZ04] Christian Kop, Heinrich C. Mayr und Tatjana Zavinska: *Using KCPM for Defining and Integrating Domain Ontologies.* In: *WISE Workshops*, Band 3307 der Reihe *Lecture Notes in Computer Science*, Seiten 190–200. Springer, 2004.

[Knu04] Holger Knublauch: *Ontology-Driven Software Development in the Context of the Semantic Web: An Example Scenario with Protege/OWL.* In: David S. Frankel, Elisa F. Kendall und De-

borah L. McGuinness (Herausgeber): *1st International Workshop on the Model-Driven Semantic Web (MDSW2004)*, 2004.

[Krz06] Barbara Krzensk: *Beschreibungstechniken für Ontologien in der Softwaretechnik*. Diplomarbeit, 2006.

[KVH+05] Christian Kop, Jürgen Vöhringer, Martin Hölbling, Thomas Horn, Heinrich C. Mayr und Christian Irrasch: *Tool Supported Extraction of Behavior Models*. In: *ISTA*, Seiten 114–123, 2005.

[KVV06] Markus Krötzsch, Denny Vrandečić und Max Völkel: *Semantic MediaWiki*. In: *The Semantic Web - ISWC 2006*, Band 4273 der Reihe *Lecture Notes in Computer Science*, Seiten 935–942, Heidelberg, DE, 2006. Springer.

[Kü02] Thomas Kühne: *The Role of Metamodeling*. In: *in MDA, International Workshop in Software Model Engineering (in conjunction with UML'02)*, 2002.

[LSR07] Frank J. van der Linden, Klaus Schmid und Eelco Rommes: *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, Secaucus, NJ, USA, 2007.

[MCF03] Stephen J. Mellor, Anthony N. Clark und Takao Futagami: *Model-Driven Development*. IEEE SOFTWARE, 20(05):14–18, 2003.

[MK02] Heinrich C. Mayr und Christian Kop: *A User Centered Approach to Requirements Modeling*. In: *In: M. Glinz, G. Müller-Luschnat (Hrsg.): Modellierung 2002 - Modellierung in der Praxis - Modellierung für die Praxis*, Seiten 75–86. Springer, 2002.

[MK06] Heinrich C. Mayr und Christian Kop: *Benutzerzentrierte Modellierung*. Alpen-Adria Universität Klagenfurt, orkshop Modellierung 2006Workshop Modellierung 2006 Auflage, März 2006.

[Myl81] John Mylopoulos: *An overview of Knowledge Representation*. SIGMOD Rec., 11(2):5–12, 1981.

[NHM00] Philip Nour, Harald Holz und Frank Maurer: *Ontology-based Retrieval of Software Process Experiences*. In: *ICSE Workshop on Software Engineering over the Internet*, 2000.

[NMN09] Antonio Nicola, Michele Missikoff und Roberto Navigli: *A software engineering approach to ontology building*. Inf. Syst.,

34(2):258–275, 2009.

[OESV04] Daniel Oberle, Andreas Eberhart, Steffen Staab und Raphael Volz: *Developing and Managing Software Components In An Ontology-Based Application Server.* In: *In 5th International Middleware Conference, LNCS*, Seiten 459–477. Springer, 2004.

[OLG+06] Daniel Oberle, Steffen Lamparter, S. Grimm, D. Vr, S. Staab und A. Gangemi: *A.: Towards Ontologies for Formalizing Modularization and Communication in Large Software Systems.* Journal of Applied Ontology, Seiten 163–202, 2006.

[OMG05a] OMG: *MOF QVT Final Adopted Specification.* Object Management Group, June 2005.

[OMG05b] OMG: *UML 2.0 Superstructure Specification, Version 2.0.* Technischer Bericht, Object Management Group, 2005.

[OMG06a] OMG: *Meta Object Facility (MOF) Core Specification Version 2.0.* Object Modeling Group, 2006.

[OMG06b] OMG: *Ontology Definition Metamodel, Sixt Revised Submission.* Object Modeling Group, 2006.

[OMG07] OMG: *Unified Modelling Language 2.1.2 Infrastructure Specification.* Specification Version 2.1.2, Object Management Group, November 2007.

[OMG08] OMG: *Software Process Engineering Meta-Model, version 2.0*, April 2008.

[OSU08] Peter Ohrstrom, Henrik Schaerfe und Sara L. Uckelman: *Jacob Lorhard's Ontology: A 17th Century Hypertext on the Reality and Temporality of the World of Intelligibles.* In: *ICCS '08: Proceedings of the 16th international conference on Conceptual Structures*, Berlin, Heidelberg, 2008. Springer.

[OVRT06] Kathia Oliveira, Karina Vilela, Ana Regina Rocha und Guilherme Horta Travassos: *Use Ontologies in Software Development Environments*, Kapitel 10. Springer, 2006.

[Pic07] Roman Pichler: *Scrum: Agiles Projektmanagement erfolgreich einsetzen.* dpunkt, Heidelberg, 2007, ISBN 978-3-89864-478-5.

[RGP06] Jesús Barrasa Rodríguez und Asunción Gómez-Pérez: *Upgrading relational legacy data to the semantic web.* In: Les Carr, David De Roure, Arun Iyengar, Carole A. Goble und Michael Dahlin (Herausgeber): *WWW*, Seiten 1069–1070. ACM, 2006.

[Ruß07] Aaron Ruß: *Übersetzung von UML-Diagrammen für die*

Ontologie-basierte Software-Entwicklung. Diplomarbeit, Philipps-Universität Marburg, 2007.

[Sar03] Siar Sarferaz: *Methoden und Werkzeugunterstützung für evolutionäre, objektorientierte Software-Projekte.* Dissertation, Philipps-Universität Marburg, 2003.

[SCG+05] Miguel Ángel Sicilia, Juan José Cuadrado, Elena García, Daniel Rodríguez und José R. Hilera: *The Evaluation of ontological representations of the SWEBOK as a revision tool*, 2005.

[Sei03] Ed Seidewitz: *What Models Mean.* IEEE Softw., 20(5):26–32, 2003, ISSN 0740-7459.

[Smi04] Berry Smith: *Ontology: Philosophical and Computational.* Oxford: Blackwell, 2004.

[Som07] Ian Sommerville: *Software Engineering.* Addison-Wesley, Reading, MA, 8. Auflage, 2007.

[Sow00] John F. Sowa: *Knowledge representation: logical, philosophical and computational foundations.* Brooks Cole Publishing Co., Pacific Grove, CA, USA, 2000.

[Sow01] John F. Sowa: *Conceptual Graph Examples*, 2001. http://www.jfsowa.com/cg/cgexampw.htm.

[Sta73] Herbert Stachowiak: *Allgemeine Modelltheorie.* Springer, 1973.

[Sta03] Standish Group: *CHAOS Report 2003.* Technischer Bericht, The Standish Group, 2003.

[SW01] Barry Smith und Christopher Welty: *FOIS introduction: Ontology—towards a new synthesis.* In: *FOIS '01: Proceedings of the international conference on Formal Ontology in Information Systems*, Seiten 3.–9., New York, NY, USA, 2001. ACM.

[TPCB06] Valentin Tablan, Tamara Polajnar, Hamish Cunningham und Kalina Bontcheva: *User-friendly ontology authoring using a controlled language.* In: *In Proceedings of LREC 2006 - 5th International Conference on Language Resources and Evaluation. ELRA ELDA*, 2006.

[TPO+06] Phil Tetlow, Jeff Z. Pan, Daniel Oberle, Evan Wallace, Michael Uschold und Elisa Kendall: *Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering.* W3C Working Draft Working Group Note 2006/02/11, W3C, März 2006. `http://www.w3.org/2001/sw/BestPractices/SE/ODA/`.

[UJ99] Mike Uschold und Robert Jasper: *A Framework for Understanding and Classifying Ontology Applications.* In: *IJCAI-99 Workshop on Ontologies and Problem-Solving Methods (KRR5)*, Seiten 16–21, 1999.

[VPST05] Denny Vrandecic, H. Sofia Pinto, York Sure und Christoph Tempich: *The DILIGENT Knowledge Processes.* Journal of Knowledge Management, 9(5):85–96, 2005.

[WHOS08] Jan Wielemaker, Michiel Hildebr, Jacco Van Ossenbruggen und Guus Schreiber: *Thesaurusbased search in large heterogenous collections.* In: *Proc. The Semantic Web - ISWC 2008 7th International Semantic Web Conferenc.* Springer-Verlag, 2008.

[Wil08] Edward D. Willink: *Adapting EMOF/EssentialOCL/QVT to Ecore/MDT-OCL/EQVT.* In: *Eclipse OMG Symposium*, 2008.

[ZR04] Christian Zimmer und Axel Rauschmayer: *Tuna: Ontology-Based Source Code Navigation and Annotation.* In: *Workshop on Ontologies as Software Engineering Artifacts (OOPSLA)*, 2004.

Index

Lebenslauf

Andrej Bachmann (geb. Rodionov)

09.1983-05.1993	Mittelschule Nr. 4, Scheskasgan
09.1993-05.1994	Informatikstudium, Moskau
08.1995-06.1997	Ludwig-Geißler-Schule (Abitur), Hanau
11.1997-08.1998	Wehrdienst, Fuldatal
10.1998-03.2005	Informatikstudium, Marburg
seit 11.2002	Externer Berater, Deutsche Bank, Eschborn
10.2005-09.2009	Wissenschaftlicher Mitarbeiter, FB 12, Philipps-Universität Marburg