

DIE PRAKTISCHE EINSETZBARKEIT DES CIDOC CRM IN INFORMATIONSSYSTEMEN IM
BEREICH DES KULTURERBES

Magisterarbeit im Fach Informationsverarbeitung

von

Markus Nix

Prof. Dr. Manfred Thaller
Professur für Historisch-Kulturwissenschaftliche Informationsverarbeitung
Universität zu Köln

Köln, 1. März 2004

Inhaltsverzeichnis

1	Einleitung.....	4
1.1	Metadaten.....	5
1.2	RDF.....	6
1.3	Semantic Web.....	7
2	Ontologien.....	9
2.1	Was ist eine Ontologie.....	9
2.2	Ein Beispiel.....	12
2.3	Arten von Ontologien	15
2.4	Ontologie-Sprachen und Ontologien.....	16
2.4.1	OWL	17
2.4.2	Topic Maps.....	19
2.4.3	Zusammenfassung.....	21
3	Das CIDOC Conceptual Reference Model.....	22
3.1	Ursprung.....	22
3.2	Zielsetzung.....	23
3.3	Verwendung als Ontologie.....	24
3.4	Der objektorientierte Aufbau des CRM.....	25
3.4.1	Grundbegriffe der Objektorientierung.....	25
3.4.1.1	Klassifikation und Identität.....	25
3.4.1.2	Vererbung.....	26
3.4.1.3	Polymorphie.....	26
3.4.1.4	Hierarchisierung.....	26
3.4.1.5	Ist-Ein-Relationen und Hat-Ein-Relationen.....	27
3.4.2	CRM Klassen.....	27
3.4.3	CRM Properties.....	29
3.4.4	Domain und Range.....	30
3.4.5	Vererbung.....	31
3.4.6	Die Modellierung des CRM.....	32
3.5	Weitere Anwendungen.....	36
4	Praktische Einsetzbarkeit des CRM.....	37

4.1	Das CRM in RDF.....	37
4.2	Das CRM in XML.....	39
4.2.1	Die Test-Daten.....	39
4.2.2	Speichern und Suchen - ein Versuch.....	40
4.2.3	Bearbeitung der Daten.....	41
4.2.4	Entwurf einer neuen DTD.....	42
4.3	Das CRM in Java.....	49
4.3.1	Aufbau der Klassenbibliothek.....	50
4.3.2	Stand der Klassenbibliothek.....	51
4.3.3	Klassen.....	52
4.3.4	Properties als Methoden.....	52
4.3.5	Erweiterung der Klassenbibliothek	54
4.3.6	Werkzeuge.....	54
4.3.6.1	Filesplitter.....	54
4.3.6.2	Objektprinter.....	56
5	Abschließende Bemerkungen.....	58
5.1	Erreichter Stand.....	58
5.2	Offene Fragen und folgende Schritte.....	58
5.3	Ausblick.....	61
6	Anhang A - Die DTD.....	63
7	Anhang B - Quellcode Beispiele.....	65
8	Anhang C - Inhalt der CD-ROM.....	72
9	Literaturverzeichnis.....	73
10	Erklärung.....	78

1 Einleitung

Es steht uns eine praktisch unbegrenzte Menge an Informationen über das World Wide Web zur Verfügung. Das Problem, das daraus erwächst, ist, diese Menge zu bewältigen und an die Information zu gelangen, die im Augenblick benötigt wird. Das überwältigende Angebot zwingt sowohl professionelle Anwender als auch Laien zu suchen, ungeachtet ihrer Ansprüche an die gewünschten Informationen. Um dieses Suchen effizienter zu gestalten, gibt es einerseits die Möglichkeit, leistungsstärkere Suchmaschinen zu entwickeln. Eine andere Möglichkeit ist, Daten besser zu strukturieren, um an die darin enthaltenen Informationen zu gelangen. Hoch strukturierte Daten sind maschinell verarbeitbar, sodass ein Teil der Sucharbeit automatisiert werden kann. Das Semantic Web ist die Vision eines weiterentwickelten World Wide Web, in dem derart strukturierten Daten von so genannten Softwareagenten verarbeitet werden.

Die fortschreitende inhaltliche Strukturierung von Daten wird Semantisierung genannt. Im ersten Teil der Arbeit sollen einige wichtige Methoden der inhaltlichen Strukturierung von Daten skizziert werden, um die Stellung von Ontologien innerhalb der Semantisierung zu klären. Im dritten Kapitel wird der Aufbau und die Aufgabe des CI-DOC Conceptual Reference Model (CRM), einer Domain Ontologie im Bereich des Kulturerbes dargestellt.

Im darauf folgenden praktischen Teil werden verschiedene Ansätze zur Verwendung des CRM diskutiert und umgesetzt. Es wird ein Vorschlag zur Implementierung des Modells in XML erarbeitet. Das ist eine Möglichkeit, die dem Datentransport dient. Außerdem wird der Entwurf einer Klassenbibliothek in Java dargelegt, auf die die Verarbeitung und Nutzung des Modells innerhalb eines Informationssystems aufbauen kann.

1.1 Metadaten

Metadaten sind Daten, die andere Daten beschreiben. In HTML-Dateien werden beispielsweise häufig Informationen über Inhalte des Dokuments im so genannten Header eingebettet. Sie dienen der Beschreibung des Seiteninhalts, auf die z.B. Suchmaschinen zurückgreifen. Es können aber noch weitere Informationen über das Dokument aufgeführt werden, wie z.B. Autor oder Erstellungsdatum, die nicht von Suchmaschinen genutzt werden. Selbstverständlich lassen sich nicht nur HTML-Dokumente mit Metadaten beschreiben. Typische Metadaten eines Buchs sind der Name des Autors, die Auflage, das Erscheinungsjahr und der Verlag. Nur normierte Metadaten lassen sich maschinell verarbeiten, zu diesem Zweck wurde eine Reihe von Formaten erstellt.

Ein Format, das diese Art Metadaten normiert, ist beispielsweise das *Maschinelle Austauschformat für Bibliotheken* (MAB).¹ Es ist vor allem im deutschen Bibliothekswesen gebräuchlich. Ein ähnliches, aber international verbreitetes Format ist MARC (MAchine-Readable Cataloging).² Ein weiteres Format bietet die Text Encoding Initiative (TEI),³ eine Organisation, das TEI-Konsortium, und ein gleichnamiges Dokumentenformat zur Kodierung für den Austausch von Texten. Dieses auf XML⁴ basierende Format hat sich zu einem De-facto-Standard innerhalb der Geisteswissenschaften entwickelt. Dennoch gibt es keinen offiziellen Standard, sondern in Gegenteil noch eine Vielzahl von weiteren Formaten.

Um Metadaten zu vereinheitlichen und damit nicht nur maschinenlesbar, sondern für möglichst viele Anwendungen zugänglich zu machen, existieren verschiedene Initiativen, darunter die Dublin Core Metadata Initiative (DC).⁵ Diese Initiative entwickelte den so genannten *Dublin Core*, ein Format, das aus einem Set von fünfzehn optional belegbaren Datenfeldern besteht, die der Beschreibung von Dokumenten und anderen Objekten dienen. Er kann auf verschiedene Arten implementiert werden, z.B. in XML oder RDF⁶ und ist damit kein Format, das sich auf eine bestimmte Richtung von Doku-

¹ MAB <http://www.ddb.de/index_txt.htm> unter dem Menüpunkt „DDB professionell“ (18.01.04).

² MARC <<http://www.loc.gov/marc/>> (2.01.04).

³ TEI <<http://www.tei-c.org/>>. (2.01.04), siehe auch Sperberg-McQueen/Burnard (2002).

⁴ *Extensible Markup Language* (XML) <<http://www.w3.org/XML/>> (27.02.04).

⁵ *Dublin Core* <<http://dublincore.org/>> (2.01.04).

⁶ *Resource Description Framework* (RDF) <<http://www.w3.org/RDF/>> (2.01.04), siehe auch Champin (6/2001).

menten oder Objekten spezialisiert, sondern als übergreifendes Metadatenformat eingesetzt werden soll.

Trotz der Standardisierung bleiben Metadaten immer Inhaltsbeschreibungen und lassen eine maschinelle Verarbeitung der Bedeutung dieser Inhalte nur sehr bedingt zu. Das heißt, maschinelles Suchen innerhalb von Metadaten ist zwar möglich und wird häufig angewendet, aber die semantische Auswertung des eigentlichen Dokuments bleibt Aufgabe des Menschen.

1.2 RDF

Der nächste Schritt zur Verwirklichung maschinenlesbarer Inhalte bestand in der Entwicklung des *Resource Description Framework* (RDF). Es wurde insbesondere für die Verwendung im *World Wide Web* (WWW) geschaffen und soll der Vereinheitlichung von Metadaten dienen, indem es eine Struktur zur Codierung, zum Austausch und zur Wiederverwendung von Metadaten bereitstellt. Das RDF ist eigentlich ein Modell und an keine Syntax gebunden, wird aber üblicherweise in XML repräsentiert. Daher kann man sagen, dass RDF eine XML-Anwendung ist. Zum Beispiel existiert der Dublin Core, der zuerst in ‚einfachem‘ XML implementiert wurde, auch in einer RDF-Version. Bestand das RDF zunächst nur aus der Definition einer einheitlichen Struktur, das heißt einem Syntax-Modell für Metadaten, und diente vornehmlich der Beschreibung von Ressourcen, entwickelt es sich mit der Einführung von RDF-Schema (RDFS)⁷ mehr und mehr zu einem eigenen Inhaltsformat.

RDFS definiert ein Set aus Kernklassen und deren Eigenschaften (so genannten Properties), das über die Beschreibung von Ressourcen hinaus die Darstellung von semantischen Relationen ermöglicht. Denn mit den Kernklassen lassen sich weitere, eigene Klassen frei definieren. Objekte als Instanzen dieser Klassen lassen sich miteinander in Beziehung setzen, wobei diese Beziehungen genau bestimmt sind. Das bedeutet, dass RDFS das Werkzeug ist, mit dem sich Ontologien in RDF beschreiben lassen. Ontologien sind ein weiterer wichtiger Schritt zur Semantisierung, wie im folgenden Abschnitt 1.3 ausgeführt wird.

⁷ RDFS <<http://www.w3.org/TR/rdf-schema/>> Einen guten kurzen Überblick auf RDFS gibt Rosin, (o.J.).

Eine RDF Anwendung ist RSS. Der Abkürzung werden zwar verschiedenen Bedeutungen zugeschrieben, aber die jeweilige Anwendung ist im Kern die gleiche (Voltz 2003; Miller et al. 2001).⁸ RSS ist ein RDF-Vokabular, das dazu genutzt wird, Inhalte von Webseiten maschinell auszuwerten und an registrierte Anwender Meldungen über einen *RSS-Reader* zu verschicken, der z.B. darüber informiert, dass der Seiteninhalt aktualisiert wurde. Die Funktion von RSS ist ähnlich der eines *Newsletters*, den eine Anwender abonniert; die technische Umsetzung unterscheidet sich jedoch. Dadurch, dass der Inhalt bei der Erstellung durch den Autor gekennzeichnet wird, muss gerade kein Newsletter mehr erstellt werden, sondern die gewünschten Informationen werden automatisch vom *RSS-Reader* abgerufen. RSS ist eine der ersten einfachen Anwendungen des Semantic Web.

1.3 Semantic Web

Das Semantic Web⁹ ist eine im Aufbau befindliche Erweiterung des heutigen WWW. Diese Erweiterung besteht in der semantischen Kodierung von Daten, sodass *Inhalte* von Dokumenten eben nicht nur von Menschen gelesen werden können, sondern auch Programmen, so genannten *Softwareagenten*, zugänglich sind.

Der stufenweise Aufbau des Semantic Web geht auf die Initiative des *World Wide Web Consortium* (W3C)¹⁰ zurück, wobei nach und nach sieben Schichten, aus denen das Semantic Web einmal bestehen soll, umgesetzt werden (Koivunen/Miller 2001). Die bereits existierenden Schichten setzen sich folgendermaßen zusammen: auf der untersten Stufe *Unicode*¹¹ und *Uniform Resource Identifiers* (URI), darauf aufbauend *XML*, *Namespaces*¹² und *XML-Schema*,¹³ darauf wiederum liegt die dritte Schicht aus *RDF* und *RDFS* bestehend und schließlich die *Web Ontology Language* (OWL). Letztere wurde am 10. Februar 2004 zusammen mit dem RDF als *W3C Recommendation* verabschiedet und hat somit einen offiziellen Status erreicht.

⁸ Die Abkürzung RSS steht für *Rich Site Summary* oder für *RDF Site Summary* oder auch für *Really Simple Syndication*, was mit der Entstehungsgeschichte der verschiedenen RSS-Versionen zusammenhängt.

⁹ *SemanticWeb* <<http://www.w3.org/2001/sw/>> (12.02.04), siehe auch Berners-Lee/Hendler/Lassila, (5/2001).

¹⁰ W3C <<http://www.w3.org/>> (27.02.04).

¹¹ *Unicode* <<http://www.unicode.org/>> (27.02.04).

¹² *Namespaces* <<http://www.w3.org/TR/REC-xml-names/>> (27.02.04).

¹³ *XML Schema* <<http://www.w3.org/XML/Schema>> (12.02.04).

OWL soll der Beschreibung von Ontologien dienen, die den Kern des Semantic Web bilden. Das zumindest ist das Bild, was sich zum derzeitigen Stand abzeichnet. Ob sich dieser Kern nicht doch auf einer höheren Ebene befindet, wird man erst sagen können, wenn die Entwicklung des Semantic Web abgeschlossen ist.

Mit OWL hat das W3C eine auf RDFS aufbauende Sprache entwickelt, die die Ausdrucksmöglichkeiten der Sprachen DAML und OIL in RDF vereinigt. Doch bevor dieser Punkt in Abschnitt 2.4.1 wieder aufgegriffen und detailliert besprochen wird, soll erläutert werden, was Ontologien sind.

2 Ontologien

2.1 Was ist eine Ontologie

Der Begriff *Ontologie* stammt aus der Philosophie. Dort bezeichnet er die allgemeine Lehre vom Sein bzw. dem Seienden, die neben der Erkenntnistheorie und der Logik eine der zentralen Einzeldisziplinen der Philosophie ist. Der Begriff taucht erstmals in der Philosophie des 17. Jahrhunderts auf, während die Seinslehre selbst auf den griechischen Philosophen Parmenides zurückgeht. Innerhalb dieser wird im Kern das Sein (Ontos) mit dem Denken (Logos) in Verbindung gebracht, was sich darauf begründen lässt, dass jede Prädikation eine Aussage über ein Sein impliziert (nach Kwiatkowski 1985:296 f.).

In Anlehnung an den philosophischen Begriff verwendet man den Begriff seit einigen Jahren in der Informatik, wobei in der Prädikation vermutlich die nächste und einzige Verbindung der beiden Wortbedeutungen liegt. Im Folgenden wird *Ontologie* immer im Sinne der Informationstechnologie verwendet.

Die vermutlich kürzeste und meistzitierte Definition des Begriffs stammt von Gruber (4/1993:1)¹⁴ und lautet: „*An ontology is an explicit specification of a conceptualization*“. Sie scheint auf den ersten Blick alles umfassend und dem entsprechend wenig aussagekräftig zu sein, was jedoch so nicht zutrifft.

Der Begriff der *Spezifikation* lässt sich als Definition oder als formale Darstellung verstehen. Da Grubers Definition keinerlei Restriktionen hinsichtlich der Art der Formalisierung macht, ist praktisch die Verwendung aller Mittel offen, solange es sich um eine explizite, das heißt eine offene und nachvollziehbare Darstellung handelt. Es gibt eine Reihe anderer Definitionen zum Begriff der Ontologie, die in diesem Punkt ausführlicher sind, aber bei genauerem Hinsehen nicht leichter einsehbar und auch nicht unbedingt exakter, denn sie werfen häufig neue Fragen auf und verlagern das Problem der exakten Bestimmung.¹⁵ Festhalten lässt sich, dass eine Ontologie oder ein Teil einer solchen immer eine formale Beschreibung liefert.

¹⁴ Diese Definition ist auch an anderen Stellen zu finden z.B. in Gruber (8/1993:1).

¹⁵ Guarino (10/96:2) bespricht ausführlich eine Reihe von Definitionen.

Der Begriff der *Konzeptualisierung* ist wesentlich schwieriger zu fassen. Dazu nochmals Gruber, der im folgenden Zitat seine Definition ausführt, wobei er von Ontologien für *Artificial Intelligence* (AI) Systeme spricht:

“For AI systems, what ‘exists’ is that which can be represented. When the knowledge of a domain is represented in a declarative formalism, the set of objects that can be represented is called the universe of discourse. This set of objects, and the describable relationships among them, are reflected in the representational vocabulary with which a knowledge-based program represents knowledge.” Gruber (4/1993:1)

Der wichtigste Punkt, der aus diesem Zitat hervorgeht, ist, dass Ontologien einen Ausschnitt der Wirklichkeit abbilden, wobei *Wirklichkeit* an dieser Stelle nicht philosophisch hinterfragt wird, sondern ganz pragmatisch mit Existenz gleichzusetzen ist. Ontologien transportieren Wissen in Form von beschreibbaren Relationen zwischen klassifizierten Objekten. Dieses Wissen lässt sich als *Konzeptualisierung* verstehen.

Der Begriff der *Konzeptualisierung* wird von Guarino (10/1996:2 f.) aufgenommen und kritisiert. Das Beispiel in Abbildung 1 soll dies illustrieren; es wurde aus Guarino übernommen und stellt Bauklötze dar, die auf einem Tisch liegen. Konzepte bestehen hier aus Beziehungen von Figuren, die mit natürlichsprachlichen Termen wie *auf* und *über* beschrieben werden. Guarino argumentiert, dass eine *Konzeptualisierung* nicht formal zu fassen sei, sondern vielmehr nur „intuitiv“ (10/1996:3) verstanden werden kann. So könne die linke Seite der Abbildung beispielsweise eine *Konzeptualisierung* darstellen, während die rechte Seite mit einer anderen Anordnung der Bauklötze auch eine andere *Konzeptualisierung* erfordert.

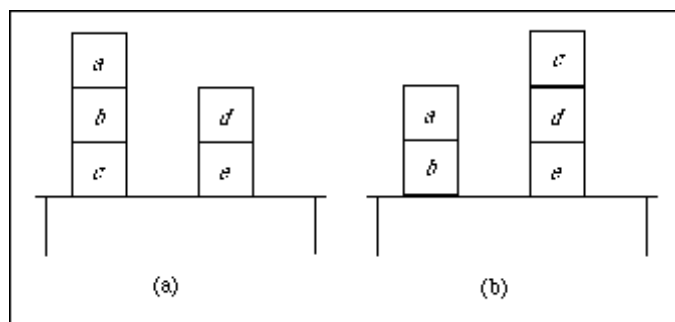


Abb. 1 Blocks on a table (Guarino, 10/1996:2)

(a) A possible arrangement of blocks.

(b) A different arrangement. Also a different conceptualization?

Guarino argumentiert weiter, dass dies zwar eine Möglichkeit wäre, der aber eine andere vorzuziehen sei, nämlich von *einem* Konzept zu sprechen, dass sich in verschiedene *states of affairs* teilen lässt. Hierbei wird deutlich, dass Guarino nicht den Begriff der Konzeptionalisierung kritisiert, sondern ihn vielmehr anwendet und die inhaltliche Definition eines Konzepts problematisiert.

Diese inhaltliche Bestimmung ist nicht unproblematisch und wirft bei der Entwicklung einer Ontologie diverse Fragen auf (Denny 11/2002; Noy/McGuinness o.J.). Zum Beispiel befindet sich in Abbildung 1 (b) *a* auf *b*. Die Frage ist nun, befindet sich *a* auch auf dem Tisch? Dem umgangssprachlichen Konzept folgend werden die meisten Befragten dem zustimmen, obwohl *a* sich nicht in der gleichen Weise auf dem Tisch befindet wie *b*. Diese inhaltliche Bestimmung eines Konzepts wie beispielsweise *auf* soll aber nach Gruber gerade durch die Spezifikation definiert werden. Was also ein Konzept umfasst, wird gegebenenfalls axiomatisch festgelegt. Obschon eine formale Bestimmung des Begriffs *Konzeptionalisierung* aussteht, sollten die wenigen Beispiele verdeutlicht haben, was ein Konzept im Sinne der Definition von Gruber ist.

Ein weiteres Zitat liefert nicht nur eine ebenfalls kurze und treffende Definition, die zur Klärung des Begriffs beitragen soll, sondern darüber hinaus eine ebenso treffende Kritik, die später konkretisiert wird.

„Unter einer Ontologie versteht man in der Informatik im Bereich Künstliche Intelligenz ein formal definiertes System von Dingen und/oder Konzepten und Relationen zwischen diesen Dingen. Zusätzlich enthalten Ontologien (zumindest implizit) Regeln. Ontologien haben mit der Idee des Semantic Web innerhalb der letzten Jahre einen Aufschwung erfahren, was jedoch nicht unbedingt zu einer Klärung des Begriffes Ontologie beigetragen hat. In vielen Fällen handelt es sich bei den als Ontologien bezeichneten Strukturen lediglich um kontrollierte Vokabularien wie Klassifikationen oder Thesauri.“¹⁶

Klassifikation bedeutet, dass beliebige Gruppen von Objekten in Klassen gefasst und geordnet werden. Eine Taxonomie ordnet Klassen, z.B. in einer hierarchischen Struktur. Ein bekanntes Beispiel für eine Taxonomie ist die Einteilung von Lebewesen in der Biologie. Eine Ontologie lässt sich als erweiterte Taxonomie verstehen, die entsteht, indem zwischen Instanzen der Klassen weitere Relationen definiert werden. Man kann also sagen, dass Thesauri, Klassifikationen, Taxonomien und Ontologien verschiedene

¹⁶ Wikipedia, 21.01.04 <[http://de.wikipedia.org/wiki/Ontologie_\(Informatik\)](http://de.wikipedia.org/wiki/Ontologie_(Informatik))> (17.02.04).

Techniken der Strukturierung von Konzepten oder Begriffen sind und dass bei Ontologien der Grad der Strukturierung am höchsten ist.

Abschließend sollen nochmals die eingangs erwähnten Metadaten aufgegriffen werden, um einen prinzipiellen Unterschied zu Ontologien herauszustellen. Metadaten werden erstellt, um andere Daten zu beschreiben. Ontologien ‚typisieren‘ Daten und lassen so Aussagen über deren Inhalt zu. Sie ermöglichen eine Beschreibung von Daten, indem sie Objekte über definierte Verbindungen verknüpfen.

Newcomb nennt dies, die Unterscheidung einer *Ressource-Centric View* und einer *Subject-Centric View* (2003:40 f.). Newcomb verdeutlicht diesen Aspekt zwar anhand von Topic Maps, welche in 2.4.2 besprochen werden, er lässt sich aber auf Ontologien allgemein übertragen.

Metadaten sind Daten, die Daten beschreiben, das heißt Metadaten werden als etwas verstanden, das sich *um Daten herum* befindet, es sind gewissermaßen zusätzliche Daten. Bei der Suche nach einer Information werden Metadaten nach Stichwörtern zu diesem Subjekt durchsucht. Um aber Metadaten zu durchsuchen, müsste man Meta-Metadaten haben und um in diesen zu suchen, Meta-Meta-Metadaten und so weiter und so fort. Informationen werden hier als Datenressource verstanden, die Inhalte enthalten, das ist die *Ressource-Centric View*.

Ontologien fassen alle Daten als Objekte auf.¹⁷ Anstatt dass Topics von Metadaten ‚umfasst‘, das heißt beschrieben werden, kann jedes Topic mit anderen Topics verbunden werden, indem es über Assoziationen verknüpft wird. Einige Topics bestehen dabei aus Ressourcen, also aus Inhalten. Das wäre die *Subject-Centric View*. Der wichtige Punkt sind hierbei weniger die Begriffe, die Newcomb verwendet, als der, dass es zwei Perspektiven gibt, aus denen sich Daten sich betrachten lassen.

2.2 Ein Beispiel

Ein Beispiel soll die Nutzung von Ontologien veranschaulichen. Diese Nutzung beruht grundsätzlich auf der komplexen Struktur der Daten, sie soll es ermöglichen Informationen effektiver zu verwerten. Das kann dadurch geschehen, dass sich eine Suche ge-

¹⁷ Newcomb spricht in seinem Aufsatz über Topic Maps von Subjekten, daher auch die *Subject-Centric View*. Es mag widersprüchlich erscheinen, aber die Bezeichnung *Objekt* ist in der Übersetzung zumindest für diese Arbeit treffender.

zielter gestalten lässt, oder durch die Verbindung verschiedener Quellen, die weitere Schlüsse zulässt.

Ein gutes Beispiel hierzu findet man in Doerr (10/2001:3 ff.). Er vergleicht drei Dokumente, die dasselbe Ereignis beschreiben, nämlich die Jalta-Konferenz 1945, welche das Ende des Zweiten Weltkriegs besiegelte. Das erste Dokument ist ein Auszug aus dem Protokoll der Konferenz:

"The following declaration has been approved: The Premier of the Union of Soviet Socialist Republics, the Prime Minister of the United Kingdom and the President of the United States of America have consulted with each other in the common interests of the people of their countries and those of liberated Europe. They jointly declare their mutual agreement to concert..."¹⁸

Doerr schlägt folgenden Dublin Core Eintrag zu diesem Text vor:

Type: Text
Title: Protocol of Proceedings of Crimea Conference
Title.Subtitle: II. Declaration of Liberated Europe
Date: February 11, 1945.
Creator: The Premier of the Union of Soviet Socialist Republics
The Prime Minister of the United Kingdom
The President of the United States of America
Publisher: State Department
Subject: Postwar division of Europe and Japan (aus Doerr, 10/2001:4)

Das zweite Dokument ist die Abbildung 2, ein berühmtes Foto, das während der Konferenz entstand, ein Dublin Core Metadatensatz könnte so aussehen:

Type: Image
Title: Allied Leaders at Yalta
Date: 1945
Publisher: United Press International (UPI)
Source: The Bettmann Archive
Copyright: Corbis
References: Churchill, Roosevelt, Stalin (aus Doerr, 10/2001:4)

¹⁸ *Internet Modern History Sourcebook*. 1997.
<<http://www.fordham.edu/halsall/mod/1945YALTA.html>> (23.02.04)



Abbildung 2: Staatsoberhäupter der Alliierten in Jalta

Die beiden Metadatenätze beschreiben zwar Dokumente, die das gleiche Ereignis dokumentieren, aber bis auf die wenig aussagende Jahreszahl 1945 gibt es keine Gemeinsamkeiten darin, die es erlauben würden, beide Dokumente miteinander in Verbindung zu bringen. Dass hier das gleiche Ereignis beschrieben wird, bleibt also unerkannt.

Erst ein dritter Satz Metadaten liefert ein verbindendes Element. Der *Thesaurus of Geographic Names*¹⁹ (TGN) liefert folgenden Eintrag:

TGN Id: 7012124 Names: Yalta (C,V), Jalta (C,V)
Types: inhabited place(C), city (C)
Position: Lat: 44 30 N,Long: 034 10 E
Hierarchy: Europe (continent) <- Ukrajina (nation) <- Krym (autonomous republic)

Note: Located on S shore of Crimean Peninsula; site of conference between Allied powers in WW II in 1945; is a vacation resort noted for pleasant climate, & coastal & mountain scenery; produces wine, canned fruit & tobacco products.

In der Anmerkung findet man das fehlende Glied. Doerr nennt dies eine „integrierende“ (10/2001:4) Information, die benötigt wird, um das Problem der Verbindung von Begriffen wie *Crimea*, *Yalta* und *Krym* in verschiedenen Metadatenätzen zu lösen. Gleichzeitig lässt diese Information weitere Schlüsse zu. Dass aber der *Prime Minister of the United Kingdom* und *Churchill* identisch sein könnten, ist freilich mit den vor-

¹⁹ Getty Thesaurus of Geographic Names Online (TGN), 2000, <http://webapps.getty.edu/vow/TGN-FullDisplay?find=yalta&place=&nation=&prev_page=1&english=Y&subjectid=7012124> (23.02.04).

liegenden Daten nur eine Vermutung; der Beweis lässt sich nur mit einer weiteren integrierenden Information antreten.

Mit Hilfe einer Ontologie, lassen sich Daten derart verbinden, dass eine gezielte Suche nach solchen integrierenden Informationen möglich wird. Dieser Punkt wird in Abschnitt 5.2, nachdem das *Conceptual Reference Modell* (CRM) besprochen wurde, wieder aufgenommen und eingehender diskutiert.

2.3 Arten von Ontologien

Nach Guarino (1997) werden, neben dem Grad der Detaillierung, häufig vier Arten von Ontologien unterschieden, die unterschiedlich große Geltungsbereiche abdecken.

- *Top-Level Ontologies* beschreiben dabei generelle Konzepte, deren Gültigkeit weder auf ein bestimmtes Sachgebiet noch auf eine spezielle Anwendung beschränkt ist. Das sind Konzepte wie etwa *Raum*, *Zeit*, *Ereignis* oder *Materie*.
- *Domain Ontologies* und *Task Ontologies* beschreiben die Konzepte eines Sachgebiets oder einer bestimmten Zielsetzung. Ein solches Sachgebiet wäre beispielsweise der Bereich des Kulturerbes, in dem etwa die Unterscheidung bestimmter Konzepte wichtig ist, während die gleichen Konzepte in anderen Sachgebieten überhaupt keine Verwendung finden.
- *Application Ontologies* beziehen sich gleichzeitig auf eine Domain *und* ein Target und sind daher Spezialisierungen der beiden anderen Arten. Guarino führt darüber hinaus noch einen weiteren Typ, den der *Representation Ontologies*, ein, die als Metaontologien von Ontologie-Sprachen dienen sollen, hier aber nicht weiter besprochen werden.

Diese vier Typen lassen sich hierarchisch ordnen, das heißt Ontologien mit einem engeren Geltungsbereich lassen sich unter einen größeren Geltungsbereich, einem so genannten *Scope*, subsumieren. Allerdings lassen sich beispielsweise zwei oder mehrere Domain-Ontologien nicht zu einer Top-Level Ontologie zusammenfassen, denn die Verbindung von Ontologien bestünde in der Vereinigung von unterschiedlichen Konzepten, die möglicherweise gegensätzlich definiert sind (Guarino 6/1998 Kap. 2.4). Festhalten lässt sich, dass die Art einer zu erstellenden Ontologie von der beabsichtigten

Nutzung abhängt, dass also ihr Geltungsbereich schon in der Planung definiert oder zumindest bedacht werden muss.

2.4 *Ontologie-Sprachen und Ontologien*

Es gibt zwei verschiedene Ansätze zur Erstellung einer Ontologie. Zum einen lassen sich von Beginn an so genannte Ontologie-Sprachen verwenden, die ein Vokabular formal definieren, mit dem sich Konzepte beschreiben und miteinander in Beziehung setzen lassen. Die andere Möglichkeiten besteht darin, eine Ontologie zunächst theoretisch zu definieren und die formale Beschreibung weitestgehend offen zu lassen. Auf diese Weise wurde bei der Entwicklung des CRM verfahren. Hier wurden zuerst die Konzepte und deren Relationen festgelegt, aber nicht an eine bestimmte Sprache gebunden. Ein solches Modell lässt sich später in verschiedenen Sprachen formalisieren. Die Möglichkeiten der Verwendung sind zu einem späteren Zeitpunkt zwar auf die bereits bestehenden Konzeptualisierungen beschränkt, dagegen aber steht die Freiheit, beliebige Formate für diese *rein semantische* Schnittstelle zu verwenden. Verschiedene Formate können in diesem Zusammenhang beispielsweise objektorientierte oder relationale Datenbanken sein. Ein solches Modell bietet ein Schema auf einer äußerst hohen Ebene.

Der beabsichtigte Vorteil der ersten Variante, bei der das Vokabular bereits existiert, liegt darin, dass Anwendungen unterschiedliche Ontologien verarbeiten können sollen, wie etwa ein Browser HTML-Dateien mit den unterschiedlichsten Inhalten verarbeiten kann. Ein möglicher Nachteil wäre hier, dass eine Ontologie, die in einer bestimmten Sprache modelliert wird, eventuell auf Konstruktionen, die die Sprache nicht bietet, verzichten muss.

Von Seiten des W3C wird die Entwicklung des Semantic Web vorangetrieben, indem die Entwicklung von Ontologie-Sprachen unterstützt wird. Man möchte bald eine möglichst standardisierte Form erreichen und eine Sprache bereitstellen, mit deren Hilfe Entwickler an verschiedenen Ontologien in einem einheitlichen Format arbeiten können.

2.4.1 OWL

Die *Defense Advanced Research Projects Agency* (DARPA)²⁰ ist eine Organisation, welche militärische Forschungsprojekte für das *U.S. Department of Defense* durchführt. Das wohl bekannteste und erfolgreichste Projekt ist das ARPANET, aus welchem das heutige Internet hervorging. Eine ihrer Initiativen betreibt seit dem Jahre 2000 die Entwicklung der *DARPA Agent Markup Language* (DAML).²¹ DAML ist eine Ontologie-Sprache, die auf einer Erweiterung von XML und der Verknüpfung mit dem RDF besteht.

Die europäische Konkurrenz von DAML ist eine ähnliche Ontologie-Sprache namens OIL, was für *Ontology Inference Layer*²² steht. Die Initiative, die OIL entwickelt, geht auf ein IST (Information Society Technologies) Programm der Europäischen Union zurück.

Ein weiteres Komitee erstellte im Rahmen eines Standardisierungsverfahren aus den geeignetsten Merkmalen von DAML und OIL sowie SHOE²³ und noch einigen weiteren Sprachen *DAML+OIL*.²⁴ Während DAML+OIL noch von einer W3C Initiative weiterentwickelt wird, entstand daraus bereits ein neues Sprachformat. Das ist die jüngst als *W3C Recommendation* verabschiedete Web Ontology Language (OWL).

OWL wurde in drei Varianten entworfen, *OWL Lite*, *OWL DL* und *OWL Full*, wobei die Sprachkonstrukte aus OWL Lite eine Teilmenge von OWL DL sind, und die wiederum eine Teilmenge von OWL Full ist. Alle OWL Dokumente sind immer auch RDF Dokumente. Da aber die Versionen OWL Lite und OWL DL jeweils aus einem Subset des RDFS Vokabular bestehen, gilt die Umkehrung für sie nicht, sondern nur für OWL FULL, das heißt alle gültigen RDF Dokumente sind auch in OWL Full gültige Dokumente.

Die drei Versionen wurden eingeführt, um Anwendern die Möglichkeit zu geben, sich mit der komplexen Sprache nicht tiefer zu befassen, als für die vorgesehen Zwecke unbedingt notwendig. OWL Lite soll gerade die einfachsten Ansprüche an eine Ontolo-

²⁰ DARPA <<http://www.darpa.mil/index.html>> (28.01.04).

²¹ DAML < <http://www.daml.org/index.html> > (28.01.04).

²² OIL < <http://www.ontoknowledge.org/oil/> > (28.01.04).

²³ SHOE steht für Simple HTML Ontology Extensions, <<http://www.cs.umd.edu/projects/plus/SHOE/>> (28.01.04).

²⁴ *DAML+OIL Reference Description* <http://www.w3.org/TR/daml+oil-reference>> (27.02.04).

gie-Sprache erfüllen, während OWL DL den meisten Anwendungen genügen sollte. Der Einsatz von OWL Full als eine Erweiterung von RDFS wird eher selten notwendig sein.

Wo OWL anzusiedeln ist, verdeutlicht am ehesten ein Überblick über die verwendeten Technologien (nach McGuinness/van Harmelen 2004).

- Auf der untersten Stufe befindet sich XML, in der die grundlegende Syntax beschrieben ist. Darauf aufbauend liefert XML-Schema, weitere verfeinerte Möglichkeiten, XML-Dokumente zu definieren, da sich mit XML-Schema Datentypen definieren lassen. XML-Schema lässt sich als erweiterte Syntax verstehen.
- RDF ist ein objektorientiertes Modell, indem es Ressourcen (Objekte) definiert und einfache Konstrukte liefert, Relationen zwischen diesen Objekten zu beschreiben. RDF lässt sich somit als einfaches semantisches Modell verstehen, es bedient sich genau der syntaktischen Möglichkeiten die XML bietet.
- RDF Schema ist ein Vokabular, mit dem sich RDF Ressourcen in Form von Klassen und Properties beschreiben lassen; es ermöglicht die Bildung von Klassenhierarchien. Auch hier werden keine syntaktischen Veränderungen mehr am XML-Modell vorgenommen.
- OWL schließlich ist ebenfalls eine rein semantische Erweiterung, hier wird das Vokabular von RDFS weiter ausgebaut.

Es gibt eine Reihe von Anwendungen, für die OWL vorgesehen ist bzw. wofür sich OWL-Dokumente, die nicht zwingend Ontologien sein müssen, verwenden lassen (Heflin 2004). In Abschnitt 2.1 wurde mit dem Zitat schon darauf hingewiesen, dass auch Thesauri oder Klassifikationen oft als Ontologien bezeichnet werden, was nicht zutreffend ist. Eine Beispielanwendung, die eine ‚echte‘ Ontologie verwendet, ist der *Wine Agent*.²⁵

Der Wine Agent ist ein Informationssystem, das zu Demonstrationszwecken von der Stanford University entworfen wurde. Die Anwendung greift auf eine im WWW frei zugängliche Wissensbasis zu, die aus einer in DAML verfassten Ontologie besteht. Die Anwendung selbst ist nicht spektakulär, sie würde sich mit einer relationalen Datenbank leicht umsetzen lassen: Zu einem bestimmten Gericht wird der passende Wein ermittelt.

²⁵ KSL Wine Agent 1.0 <<http://www.ksl.stanford.edu/people/dlm/webont/wineAgent/>> (23.02.04).

Das Ergebnis der Anfrage beruht aber auf einer Argumentation, die von einem so genannten *Reasoner* durchgeführt wird, und nicht auf einer einfachen Tabellenabfrage. Dieser *Reasoner* setzt das Zusammenführen von Informationen um, wie es mit dem Beispiel der Jalta-Konferenz unter 2.2 angesprochen wurde.

Der Kern des *Reasoners* basiert auf einem Java-API²⁶ namens JTP.²⁷ Das besondere an diesem Argumentationsverfahren ist, dass die Information, auch wenn sie noch so einfach scheint, nirgendwo in dieser Form gespeichert ist, sondern sich aus den Informationen der Ontologie erschließen lässt. Damit soll gezeigt werden, dass es prinzipiell möglich ist, auf verschiedene dezentral erstellte und gespeicherte Dokumente zuzugreifen und Informationen zu generieren, die sich nur aus der Verbindung von anderen Informationen ergeben. Dies ist wohl die vielversprechendste Anwendung, die sich aus der Verwendung von Ontologien ergibt. Denn darauf beruht der Einsatz von Softwareagenten und besonders der von so genannten intelligenten Softwareagenten.

2.4.2 Topic Maps

Topic Maps existieren in zwei Ausführungen, jedoch auf eine ganz andere Art als die dargestellten OWL-Versionen. Die ursprüngliche, erste Version wurde in SGML²⁸ modelliert, sie ist sehr mächtig und durch ihre Komplexität schwer zu handhaben. Kurz nach Fertigstellung der Spezifikation setzte sich daher ein Teil der Autoren zum Ziel, Topic Maps ein zweites Mal für die Verwendung mit XML zu spezifizieren. Die Technologie sollte auf die wesentlichen Punkte reduziert und so für die Verwendung attraktiver gemacht werden, was durch die so genannten XML Topic Maps (XTM)²⁹ gelungen ist. Im Folgenden sind immer XML Topic Maps gemeint.

Der Begriff Topic Maps wird zweifach gebraucht, was zu Verwechslungen führen kann. Einerseits bezeichnet er die Spezifikation, die sich als Ontologie-Sprache verstehen lässt und gewissermaßen in Konkurrenz zu OWL steht. Andererseits kann *eine* Topic Map, das heißt ein der Spezifikation entsprechendes Dokument, verschiedene Verwendungen finden, unter bestimmten Voraussetzungen auch die als Ontologie. Mit

²⁶ API steht für *Application Programming Interface*, das ist eine Klassenbibliothek mit definierten Schnittstellen.

²⁷ JTP: *An Object-Oriented Modular Reasoning System* <<http://www.ksl.stanford.edu/software/JTP/>> (23.02.04).

²⁸ Die *Standard Generalized Markup Language* (SGML) wird allgemein als die Mutter aller Markup-Sprachen bezeichnet.

²⁹ Topic Maps werden ausführlich in Widhalm, Mück (2002) behandelt, XTM in Park (2003).

Topic Maps können demnach auch mehrere Ontologien gemeint sein. Einzelne Topic Maps lassen sich zusammenführen (*mergen*), um eine neue Topic Map zu bilden; das war eine entscheidende Forderung an das Design von Topic Maps. Das lässt aber nicht den Schluss zu, dass die mit Topic Maps beschriebenen Ontologien sich gleichfalls *mergen* lassen.

Topic Maps bestehen im Wesentlichen aus *Topics*, *Associations* und *Occurrences*. Topics können beliebig festgelegt werden und z.B. Abstrakta, aber auch konkrete Objekte repräsentieren. Assoziationen sind die Relationen, mit denen sich Topics verknüpfen lassen. Auch sie, das heißt ihr semantischer Gehalt, können frei definiert werden. Occurrences sind Instanzen von Topics. Eine Eigenschaft, die das Modell mächtig und ausgesprochen komplex macht, ist, dass Assoziationen wiederum als Topics definiert werden können und damit auch wieder verknüpfbar sind. Die Spezifikation der Topic Maps legt nur die Syntax fest, mit der Topics und Assoziationen definiert werden. Sie liefert also genau das Werkzeug, was zur Erstellung von Ontologien, nach der Definition von Gruber, benötigt wird. Denn obwohl die ursprüngliche Absicht, die dem Entwurf der Topic Maps zu Grunde lag, darin bestand, die Erstellung von Indizes, Glossaren und Thesauri zu erleichtern, gehen die Verwendungsmöglichkeiten, welche sich aus dem Modell entwickelten, weit darüber hinaus (Biezunski 2003:17).

Zurück zu den Umständen, unter denen eine Topic Map als Ontologie verwendet werden kann. Dazu möchte ich nochmals den Punkt aufgreifen, dass kontrollierte Vokabularien wie Klassifikationen oder Thesauri noch keine Ontologien darstellen, sondern erst die definierten Relationen von Objekten untereinander und die sich daraus ergebene Möglichkeiten. Der ursprünglichen Absicht folgend, lassen sich Topic Maps als einfache Indizes für Informationsressourcen verwenden, aber eben auch zur Wissensrepräsentation und damit zur Modellierung von Ontologien; vorausgesetzt, sie beziehen die Relationen der Ressourcen entsprechend mit ein (Obrst/Liu 2003:124 f.).

OWL und Topic Maps, so wurde weiter oben gesagt, konkurrieren miteinander. Tatsächlich zeigt Freese (2003:325), dass sich die meisten Strukturen von RDF und Topic Maps vereinbaren lassen und auch dass zumindest Interoperabilität zwischen diesen beiden Modellen besteht. Außerdem, so Freese weiter, besteht auf beiden Seiten Interesse die Modelle zusammenzuführen. Das schließt natürlich eine Annäherung an OWL mit ein.

2.4.3 Zusammenfassung

Das Problem, was sich für die Verwendung von Ontologie-Sprachen ergeben könnte, ist vergleichbar mit dem, was sich bereits mit dem Design von Topic Maps ergab. Die ursprüngliche Spezifikation ist ungeheuer mächtig, aber schwer zu beherrschen. Daher musste eine einfachere Version, die XML Topic Maps entwickelt werden. Beide Sprachen, OWL und Topic Maps sind mächtig und bieten viele Möglichkeiten, Ontologien zu erstellen. Vielleicht zu viele, als dass die daraus entstehenden Ontologien von einer breiten Masse von Anwendungen verarbeitet werden können. Dies ist aber genau der Kern der Vision vom Semantic Web. Ein Schritt, dieses Problem erst gar nicht entstehen zu lassen, sind die drei angesprochenen Versionen von OWL.

Es ist noch unklar inwieweit sich Ontologien mehrfach verwenden lassen oder anders ausgedrückt, wie sorgfältig sie entwickelt werden müssen, damit sie über einen längeren Zeitraum und vielfältig nutzbar sind. Zumindest legt die Unterscheidung der vier Typen nahe, dass Ontologien eines größeren Geltungsbereichs auch einen größere Beständigkeit haben sollten.

Es muss erwähnt werden, dass es neben OWL und den Topic Maps noch einige andere Ontologie-Sprachen gibt, die hier nicht behandelt werden.³⁰ Die wichtigsten Sprachen sind sicherlich OWL, schon allein durch die Unterstützung des W3C, und die Topic Maps durch ihre Nähe zum RDF, die vermuten lässt, dass die beiden Formate einmal zusammengefasst werden.

Zur Erstellung von Ontologien bieten verschiedene Universitäten, Open Source Initiativen sowie kommerzielle Anbieter Werkzeuge an. Solche Werkzeugen existieren für eine Reihe von Sprachen, sie ermöglichen die automatische oder halbautomatischen Erstellung von Ontologien. Daneben existieren ebenso einige Anbieter von fertigen Ontologien, wie z.B. KAON.³¹

³⁰ *Knowledge-Base Projects, Groups, and Related Material* <<http://www.cs.utexas.edu/users/mfkb/related.html>> (28.01.04).

³¹ Denny (11/2002) gibt einen Überblick auf verschiedene Werkzeuge zur Bearbeitung von Ontologien. KAON ist eine Open Source Initiative, die eine Reihe von Werkzeugen anbietet sowie verschiedene Domain-Ontologien. <<http://kaon.semanticweb.org/>>(5.02.04).

3 Das CIDOC Conceptual Reference Model

3.1 Ursprung

Die Herkunft des CRM liegt im *International Council of Museums (ICOM)*³² begründet, einer Organisation, welche sich die Bewahrung und den Fortbestand des natürlichen und kulturellen Welterbes zum Ziel gesetzt hat. Dabei werden im Rahmen dieser Bemühungen sowohl materielle als auch immaterielle Objekte mit einbezogen.

Das ICOM wurde 1946 gegründet, es stellt eine regierungsunabhängige *non-profit* Organisation dar, die formale Beziehungen zur UNESCO unterhält. ICOM arbeitet unter anderem im Auftrag der UNESCO, übernimmt aber auch beratende Funktionen etwa gegenüber dem *Economic and Social Council* der United Nations. Die Aktivitäten von ICOM richten sich auf die Interessen von Museen und beinhalten folgende Schwerpunkte:

- professioneller Austausch und Zusammenarbeit
- Verbreitung von Wissen und Steigerung des öffentlichen Bewusstseins für Museen
- Schulung von Mitarbeitern
- Weiterentwicklung von professionellen Standards
- Ausarbeitung und Förderung einer professionellen Ethik
- Bewahrung des Kulturerbes und Bekämpfung des illegalen Handels mit kulturellem Eigentum

Die Organisation setzt sich aus inzwischen 116 nationalen und 29 internationalen Komitees zusammen, von denen sich jedes den Studien eines bestimmten Typs von Museum oder einer museumsbezogenen Disziplin widmet.

³² *International Council of Museums (ICOM)*, 28.01.04, <<http://icom.museum/>> (28.01.04).

Eines dieser Komitees ist das *International Committee for Documentation of the International Council of Museums* (ICOM-CIDOC). Das ICOM-CIDOC ist mit 750 Mitgliedern in 60 Ländern der internationale Brennpunkt von Museen und verwandten Organisationen hinsichtlich ihrer Bemühungen und Interessen in Bezug auf Dokumentation.³³ Das ICOM-CIDOC ist wiederum in rund ein Dutzend Gruppen unterteilt, von denen eine die *Documentation Standards Working Group* (DSWG)³⁴ ist.

Aus der langjährigen Arbeit der DSWG, die sich um die Schaffung eines allgemeinen Datenmodells für Museen bemühte, entstand 1994 zunächst das *CIDOC Relational Data Model*³⁵ und daraus schließlich, bei dem Versuch aus dem relationalen Modell ein objektorientiertes zu fertigen, 1999 die erste abgeschlossene Version des CIDOC Conceptual Reference Model. Zur Zeit wird die Anerkennung des CRM als Standard (ISO/CD 21127) von der International Standard Organisation (ISO)³⁶ geprüft.

3.2 Zielsetzung

Das angestrebte Ziel des CRM besteht aus einem allgemeinen Datenformat, das dem Datenaustausch bzw. der Integration heterogener Daten dienen soll und zwar derart, dass keinerlei Informationsverlust in Kauf genommen werden muss. Dazu soll es möglich sein, neben den *reinen* Daten den entsprechenden Kontext mitzuliefern, in der Art, dass historische, geografische und theoretische Kontext-Informationen zur Verfügung stehen und damit den Hintergrund liefern vor dem einzelne Gegenstände zu betrachten sind, um die ursprüngliche Bedeutung und seinen Wert zu vergegenwärtigen.

Ebenfalls soll das Format Beschreibungen auf einem Niveau zulassen, die den Standards wissenschaftlicher Forschung genügen, was nicht ausschließen soll, dass das Format nicht auch der Öffentlichkeit zur Nutzung zugänglich gemacht wird und beispielsweise zur Präsentation oder zur Recherche verwendet werden kann.

Zusammenfassend lässt sich sagen, dass die Aufgabe des CRM ist, das Wissen von Museen zu repräsentieren, zu transportieren und auf einem hohen Level sowie für professionelle Zwecke als auch für Laien zugänglich zu machen.

³³ ICOM-CIDOC, 26.10.03, <<http://www.willpowerinfo.myby.co.uk/cidoc/>> (28.01.04).

³⁴ DSWG, 26.10.03, <<http://www.willpowerinfo.myby.co.uk/cidoc/wg1.htm#docwg>> (28.01.04).

³⁵ *CIDOC Relational Data Model* <<http://www.willpowerinfo.myby.co.uk/cidoc/model/relational.model/>> (26.02.04).

³⁶ International Organization for Standardization (ISO) <<http://www.iso.ch/iso/en/ISOOnline.frontpage>> (26.02.04).

3.3 Verwendung als Ontologie

Der Schwerpunkt dieser Arbeit liegt in der Untersuchung der Verwendbarkeit des CRM als Ontologie, weitere mögliche Verwendungen werden nicht untersucht, sondern nur kurz in Abschnitt 3.5 angesprochen.

Das CRM liefert Definitionen und eine formale Struktur, um die impliziten und expliziten Konzepte und Beziehungen, die innerhalb des Kulturerbe Sektors Verwendung finden, zu beschreiben. In diesem Sinne stellt es eine Domain Ontologie dar, die der folgenden Zielsetzung genügen soll:

"The CIDOC CRM is intended to promote a shared understanding of cultural heritage information by providing a common and extensible semantic framework that any cultural heritage information can be mapped to. It is intended to be a common language for domain experts and implementers to formulate requirements for information systems and to serve as a guide for good practice of conceptual modeling. In this way, it can provide the 'semantic glue' needed to mediate between different sources of cultural heritage information, such as that published by museums, libraries and archives." (Crofts u.a. 2002)

Die angestrebten Ziele sind noch nicht alle vollständig erreicht, und aus dem Zitat wird deutlich, dass sie über die Breitstellung einer Ontologie weit hinaus geht: Ein allgemeines semantisches Framework, welches alle Informationen eines Kulturerbe-Objektes beinhalten kann, verspricht gewissermaßen eine Container-Funktion. Dieser Teil wurde realisiert, und es ist, wie später gezeigt wird, gleichzeitig möglich, dass sich Informationen nahezu beliebig genau und strukturiert beschreiben lassen, so dass trotz der umfassenden Container Funktion nichts an der Exaktheit der Informationen verloren geht.

Das CRM soll ferner als Vermittlersprache zwischen Domain-Experten und Programmierern dienen können, weil es gleichzeitig ein formales *und* ein semantisches Modell ist. Als semantisches Modell liefert es eine Vorgabe dafür, was eine Dokumentation im Kulturerbe-Bereich leisten soll, weswegen es sich generell als Leitlinie für eine gute Dokumentation eignet. Unter diesen Voraussetzungen kann es auch der Mediation von heterogenen Quellen dienen, das heißt, es lässt sich als Metaschema verwenden, mit dem sich Quellen verschiedener Institute und/oder verschiedener Formate zusammenführen lassen.

Aus der geschilderten Intention resultieren zwei wesentliche Eigenschaften des CRM, die die besondere Funktion als Ontologie betreffen: Zum einen beschreibt das CRM eine Reihe von Relationen, anstatt ein allgemeines Datenformat vorzuschreiben.

Zum anderen konzentriert es sich stärker auf die Definition von Relationen als auf die von Klassen. Dies drückt sich in der Beschreibung einer relativ überschaubaren Menge von rund 80 Klassen und ca. 130 (bzw. 260) Relationen aus. Indem Instanzen der Klassen mittels der definierten Relationen miteinander verknüpft werden, lassen sich charakteristische Konzepte von Museen, Bibliotheken und Archiven beschreiben.

Da es sich beim CRM um ein objektorientiertes Modell handelt, sollen zum weiteren Verständnis kurz die wichtigsten Konzepte des objektorientierten Paradigmas dargelegt werden, bevor das CRM selbst eingehender beschrieben wird.

3.4 Der objektorientierte Aufbau des CRM

3.4.1 Grundbegriffe der Objektorientierung

Objektorientierung bzw. objektorientierte Programmierung ist *das* Paradigma der 90er Jahre. Es an dieser Stelle ausführlich zu behandeln, ist selbstverständlich nicht möglich, aber auch nicht notwendig. Das CRM besitzt eine ausgesprochen einfache Klassenhierarchie, zumindest aus der Sicht desjenigen, der mit den Begriffen objektorientierter Programmierung vertraut ist. Es sind nur einige wenige Grundlagen vonnöten, die in der Regel den Kern von objektorientierten Modellen bilden und die im Folgenden besprochen werden, um den Aufbau des CRM darzulegen.³⁷ Eine kurze Einführung in das objektorientierte Modell des CRM gibt Crofts (1998).

3.4.1.1 Klassifikation und Identität

Unter Klassifikation versteht man die Zuordnung von *Objekten* zu einer *Klasse*. Eine Klasse beschreibt die Eigenschaften einer Gruppe von Objekten. Das bedeutet, alle Objekte mit den durch die Klasse, beschriebenen Eigenschaften gehören zu dieser Klasse; solche Objekte nennt man Instanzen der Klasse, oder man sagt, sie sind Instanzen vom *Typ* der Klasse. Tatsächlich werden die Begriffe *Objekt* und *Instanz* allgemein und auch in dieser Arbeit synonym verwendet. Setzt man der Einfachheit halber voraus, dass von geometrische Figuren die Rede ist, wäre ein simples Beispiel etwa:

³⁷ Eine ausführlichere Darstellung dieser Grundlagen liefern Balzert 2001; Krüger 2003:143ff oder Goll/Weiß/Rothländer 2002:39-49.

- (1) *Zur Klasse der Vierecke gehören alle Objekte, die vier Ecken und vier Seiten haben.*

Identität bedeutet, dass jedes Objekt unabhängig neben anderen Objekten besteht und innerhalb der Beschränkung durch die Klassendefinition beliebige Formen annehmen kann. Die Klasse ist also immer ein abstraktes Konzept, ein Modell oder eine Definition, und das Objekt ist immer eine konkrete Instanz mit Eigenschaften, die diesem Konzept entsprechen.

3.4.1.2 Vererbung

Eine Tochterklasse *erbt* alle Eigenschaften ihrer Mutterklasse und beschreibt darüber hinaus noch weitere Eigenschaften. Eine Tochterklasse ist die Spezialisierung einer Mutterklasse und die Mutterklasse ist die Generalisierung der Tochterklasse.

- (2) *Alle Vierecke, deren gegenüberliegende Seiten zueinander parallel verlaufen, sind Parallelogramme.*

Die Klasse der Parallelogramme ist also eine Tochterklasse der Klasse der Vierecke. Ein Vorteil der Vererbung ist, dass eine Klasse nicht vollständig neu beschrieben werden muss, sondern nur deren erweiternde Eigenschaften, da sie ‚weiß‘, welche Eigenschaften ihre Mutterklasse hat.

3.4.1.3 Polymorphie

Polymorphie bedeutet *Vielgestaltigkeit*, sie drückt aus, dass ein Objekt vom Typ seiner Klasse und ebenso vom Typ seiner Mutterklasse(n) ist wie z.B. ein Parallelogramm auch ein Viereck ist. Ein Objekt hat also in gewisser Weise mehrere Gestalten. Polymorphie ist eng mit der Hierarchisierung verknüpft.

3.4.1.4 Hierarchisierung

Eine Klasse kann mehrere *direkte* Unterklassen haben oder *indirekte*, indem eine Tochterklasse wiederum eine Tochterklasse hat. Direkte Tochterklassen auf einer Ebene nennt man auch *Schwesterklassen*. Erweitert eine Klasse B eine Klasse A, und erweitert eine Klasse C die Klasse B, so erweitert C auch A. Die Klassen B und C sind beide

Tochterklassen der Mutterklasse A, das heißt, auch die Tochterklasse der zweiten Ebene besitzt alle Eigenschaften der höchsten Mutterklasse. Beispielsweise sind Rechtecke Parallelogramme, und sie sind auch Vierecke. Rechtecke sind in diesem Beispiel daher vom Typ dreier Klassen. Auf diese Weise lassen sich komplexe hierarchische Strukturen bilden, die sich in einem Ableitungsbaum darstellen lassen.

3.4.1.5 Ist-Ein-Relationen und Hat-Ein-Relationen

Objekte sind in der Regel komplex, das heißt, sie bestehen aus mehreren Objekten verschiedener Klassen. Beispielsweise gehören ein Auto, ein Motorrad und ein Fahrrad zur Klasse der Fahrzeuge. Das Auto hat als einziges dieser Objekte Türen oder anders ausgedrückt, es *hat* Objekte vom Typ Tür. Innerhalb der Objektorientierung würde man sagen, dem Auto wird in seiner Klassendefinition ein Objekt vom Typ Tür zugeschrieben. Ein Objekt vom Typ Auto *ist* also ein Fahrzeug, und es *hat* ein Objekt, was alle Eigenschaften der Klasse Tür besitzt, obwohl es selbst nicht zu dieser Klasse gehört. In der objektorientierten Programmierung kann ein Objekt neben anderen Objekten auch Datenfelder und Methoden *haben*. Der letzte Punkt ist für den Augenblick weniger wichtig, taucht aber in Abschnitt 4.2.4 wieder auf.

3.4.2 CRM Klassen

Die folgende Darstellung des CRM ist im Wesentlichen eine Zusammenfassung der Definition des Modells (Crofts u.a. 2002) einschließlich einiger Ergänzungen, die inhaltlich aus verschiedenen Dokumenten der CIDOC CRM-Homepage stammen.

Obwohl das CRM ein objektorientiertes Modell ist, lässt sich an Begriffen wie *Entity* und *Property* noch die Entwicklung aus dem *CIDOC Relational Data Model* erkennen, denn diese Bezeichnungen stammen aus dem *Entity-Relationship-Modell* (ERM oder auch ER), das häufig für die Modellierung von relationalen Schemata verwendet wird.

Das CRM besteht nur aus Klassen und Properties, wobei eine Klasse definiert, welche Properties ihr angehören. Ein Property beschreibt jeweils die semantische Relation zweier Instanzen bestimmter Klassen. Die Klassen des CRM sind streng hierarchisch aufgebaut, das heißt, es gibt genau eine Mutterklasse *E1 CRM Entity*, aus der alle

anderen Klassen abgeleitet sind. Alle Instanzen jeder beliebigen Klasse sind also immer auch Objekte vom Typ der Klasse *E1 CRM Entity*.

Eine CRM-Klasse beschreibt ein Konzept und umfasst so eine Kategorie von Gegenständen oder Begriffen, wobei an Stelle einer formalen Definition eine so genannte *Intention* tritt, die durch eine *Scope Note* beschrieben wird. Das Konzept, welches durch die *Scope Note* beschrieben wird, ist daher ein semantisches Konzept und kein formales, das heißt die *Scope Note* ist keine Definition im strengen Sinne, sondern eine natürlichsprachliche Beschreibung des Konzepts einer Klasse, die meist ein Beispiel einer Instanz dieser Klasse angibt. Die Funktion einer Instanz einer derart semantisch bestimmten Klasse lässt sich mit einem Substantiv vergleichen.

Klassen werden innerhalb des CRM auch *Entities* genannt, weswegen dem Namen einer Klasse, im allgemeinen ein Substantiv, jeweils ein *E* und zusätzlich eine Identifikationsnummer vorangestellt ist, also beispielsweise *E34 Inscription*. Hier scheint sich allerdings eine Ungenauigkeit eingeschlichen zu haben, denn auch Objekte einer Klasse werden als *Entities* bezeichnet, was bedeutet, dass die fundamentale Unterscheidung von Klassen und Objekten verwischt wird. Die Unterscheidung soll in dieser Arbeit immer deutlich zu erkennen sein.

Eine weitere Bezeichnung für Objekte innerhalb des CRM ist der Begriff der *Extension* im Gegensatz zur oben genannten *Intention*. Es ist Bestandteil der Definition, dass keine Klasse des CRM durch die Aufzählung von *Extensionen* definiert wird; die Menge der zu einer Klasse gehörigen Instanzen ist damit nicht beschränkt. Der Begriff *Extension* ist in diesem Zusammenhang synonym zu *Objekt* und *Instanz* und wird nur der Vollständigkeit halber erwähnt.

Neben der semantischen Beschreibung werden Klassen des CRM durch ihre *Properties* unterschieden und bestimmt; diese Beschreibung ist ein formales Konzept, was die Klasse im strengen Sinne definiert. Nahezu alle Klassen definieren eigene *Properties*, die grundsätzlich immer an die Tochterklassen vererbt werden. Die oberste Klasse der Hierarchie, *E1 CRM Entity* definiert ein freies Textfeld, das beliebige unstrukturierte Daten aufnehmen kann. Auch diese Eigenschaft wird vererbt, so dass alle CRM-Objekte über ein freies Textfeld verfügen und über eine bestimmte klassenspezifische Art und Anzahl von *Properties*. CRM Objekte *enthalten* keine weiteren Objekte.

Objekte müssen nicht alle ihre Properties benutzen. So kann es vorkommen, dass zwei Objekte unterschiedlicher Klassen die gleichen Properties verwenden, das heißt, sie werden auf die gleiche Art und Weise beschrieben. Die Entscheidung, welcher Klasse ein reales, physisches Objekt zuzuordnen ist, beruht daher zum einen auf dem semantischen Konzept, das die Scope Note beschreibt, und zum anderen wird sie bestimmt durch den gewünschten Grad der Beschreibung, also dadurch, welche Properties für eine angemessene Beschreibung benötigt werden.

3.4.3 CRM Properties

Ein Property steht in etwa für ein Attribut, das der Instanz einer Klasse zugeschrieben wird. Das heißt, dass Instanzen von CRM Klassen Properties im objektorientierten Sinne *haben*. Der ‚Wert‘ eines solchen Attributs wird durch ein anderes Objekt bestimmt. Ein Property ist vergleichbar mit einem Verb, das zwei Substantive miteinander verknüpft. Wobei Properties immer zwei Instanzen, *Domain* und *Range* (s.u.) verbinden. Sie können also weder alleine stehen noch von einer einzelnen Instanz verwendet werden. Beispiel (3) zeigt die formale Beschreibung eines Property, welches in der Klasse *E1 CRM Entity* definiert ist:

(3) *P1 is identified by (identifies): E41 Appellation*

Der Name eines Property beginnt immer mit einem *P*, gefolgt von einer Identifikationsnummer und einem Verb oder einer verbalen Phrase. In Klammern steht jeweils ein Name für die ‚Gegenrichtung‘, das heißt für den Fall, dass man Domain und Range gegeneinander austauschen möchte; bei Verben ist das meist schlicht die passive Form.

Auch ein Property wird über eine Scope Note definiert, die jeweils ein semantisches Konzept beschreibt und keine Definition im strengen Sinne darstellt. Der formale Teil der Definition bestimmt lediglich den Typ der Instanzen, zwischen denen ein Property auftreten darf. In Beispiel (3) ist das die Domain *E1 CRMEntity*, die selbst nicht mehr genannt wird, da die Definition in der Klasse stattfindet und der Range *E41 Appellation*. Properties sind *der* wesentliche Bestandteil einer Klasse und aller ihrer Tochterklassen, denn alle Properties werden ausnahmslos vererbt, was bedeutet, dass die Anzahl der Properties zunimmt, je tiefer sich eine Klasse in der Hierarchie befindet. Und je mehr

Properties einem Objekt einer Klasse zur Verfügung stehen, desto detaillierter lässt es sich beschreiben.

Die Verbindungen über Properties sind durch so genannte *Cardinality Constraints* beschränkt, das Konzept ist unter anderem aus dem Entity-Relationship-Modell bekannt. Es sieht die Beziehungen 1:1, 1:n und n:m vor. Ein klassische 1:n Beziehung ist z.B., dass eine Mutter mehrere Kinder haben kann, aber jedes Kind nur eine Mutter hat. Allerdings wird in der Definition des Modells (Crofts u.a. 2002) ausdrücklich darauf hingewiesen, dass diese Beschränkung sich allein auf die Semantik bezieht und nicht in die Implementierung des Modells mit einfließen soll. Es wird an dieser Stelle nicht recht deutlich, inwiefern sie eine Hilfestellung ist.

3.4.4 Domain und Range

Eine *Domain* übernimmt die Funktion eines grammatischen Subjekts. Es ist jeweils die Klasse, in der ein Property definiert ist. Dabei ist jedes Property genau für eine Domain definiert. Da Properties aber vererbt werden, bedeutet das, dass auch alle Tochterklassen einer Domain deren Properties besitzen und damit die Funktion als Domain übernehmen können.

Der *Range* eines Properties entspricht der Funktion eines grammatischen Objekts, er ist die Klasse, auf die sich ein Property bezieht. Anders ausgedrückt (um den Vergleich aus 3.4.3 wiederaufzunehmen) ist der Range der Wert, den ein Attribut annehmen kann. Jedes Property hat nur einen Range, der wiederum alle daraus abgeleiteten Klassen mit einschließt.

Eine Klasse kann grundsätzlich beide Funktionen ausüben. Sie kann Domain für das eine Property sein und Range für ein anderes oder sogar beides gleichzeitig wie beispielsweise die Klasse *E55 Type*.

Die Unterscheidung von Domain und Range ist rein konventionell, denn Properties lassen sich immer in zwei Richtungen lesen. Es ist daher allein eine Frage der Perspektive, welche Seite welche Funktion erfüllt. Ebenso ist es eine rein sprachliche Konvention, dass in der Definition des CRM die Domains *Properties* an ihre Tochterklassen vererben, während in der anderen Richtung, also auf Seite der Ranges, *Referenzen* vererbt werden. Tatsächlich sind Referenzen auch Properties, und die Unterscheidung stiftet zunächst Verwirrung. Da sie jedoch innerhalb des CRM gemacht wird, soll der

Begriff der Referenz hier erwähnt werden; benutzt wird er im Folgenden nur, wenn die Unterscheidung relevant ist.

Analog zur Grammatik vieler natürlicher Sprachen lassen sich mit den besprochenen Komponenten nun primitive ‚Sätze‘ nach dem Muster Subjekt-Verb-Objekt in der Form Domain-Property-Range bilden. Dabei ist weniger eine der beiden möglichen ‚Satzausagen‘ von Bedeutung, das heißt eine der Lesarten, als vielmehr die gesamte Relation der beiden Instanzen. Vom Sprachgebrauch her scheint es natürlich und einleuchtend zu sein, dass jede Subjekt-Objekt-Beziehung auch eine Objekt-Subjekt-Beziehung ist, und es sich somit um eine bidirektionale Relation handelt. Diese Auslegung ist jedoch nicht ganz unproblematisch, und es gibt gute Gründe, von zwei verschiedenen *unidirektionalen* Relationen auszugehen, die durch ein Property nur konzeptionell zusammengefasst werden. Bei der Umsetzung in RDF bzw. XML tritt dieses Problem wieder zu Tage und wird dort wieder aufgenommen.

3.4.5 Vererbung

Die Vererbung innerhalb des CRM entspricht der objektorientierten Vererbung wie sie bereits besprochen wurde. Es handelt sich also um Superklasse-Subklasse bzw. Mutterklasse-Tochterklasse-Beziehungen, die immer Ist-Ein-Relationen ausdrücken. Da Klassen formal durch ihre Properties beschrieben werden, heißt das, dass eine Tochterklasse alle Properties ihrer Mutterklasse erbt und einige eigene darüber hinaus definiert. Die Mutterklasse umfasst im Gegenzug alle Instanzen ihrer Tochterklasse(n) und noch einige mehr.

Darüber hinaus sieht das CRM die Vererbung bei Properties vor, was bedeutet, dass ein Property Subproperties haben kann. So ist z.B. *P14 carried out by (performed): E39 Actor* Subproperty von *P11 had participant (participated in): E39 Actor* beides sind Properties der Domain *E5 Event*. Dabei gilt, wenn das Property A einer beliebigen Klasse ein Subproperty B hat, ist es erforderlich, dass die Domain von A Superklasse der Domain von B ist und dass ebenso der Range von A Superklasse des Range von B ist. Da in den bearbeiteten Daten keine Vererbung von Properties vorkommt, sei dieses Merkmal hier nur am Rande erwähnt.

Das CRM verwendet Mehrfachvererbung, das bedeutet, dass eine Klasse mehr als eine direkte Mutterklasse haben kann. So erbt z.B. die Klasse *E81 Transformation* aus

den Klassen *E63 Beginnig of Existence* und *E64 End of Existence*, wobei nicht eine aus der anderen abgeleitet ist, sondern es sich um Schwesterklassen handelt.

3.4.6 Die Modellierung des CRM

Die Bestandteile des CRM sind, wie weiter oben schon gesagt wurde, ausschließlich Klassen und Properties bzw. Objekte als Instanzen der Klassen. Da alle Klassen nach dem gleichen Schema aufgebaut sind und sich formal nur durch ihre Properties unterscheiden, ist das Grundprinzip des CRM tatsächlich ein regelmäßiges und damit vergleichsweise einfaches Modell. Um das deutlich zu machen, soll der Aufbau von CRM-Objekten und auch deren Verknüpfung mit anderen Objekten anhand einiger Grafiken illustriert werden. Das allgemeinste CRM-Objekt ist das vom Typ *E1 CRM Entity*:

E1 CRM Entiy
freies Textfeld: z.B. ein Name
P1 is identified by (identifies): E41 Appellation P2 has type (is type of): E55 Type P3 has note: E62 String

Abbildung 3: CRMEntity-Objekt

Der Übersichtlichkeit halber sind nur die drei Properties dargestellt, welche die Klasse in ihrer Funktion als Domain definiert, darüber hinaus gibt es noch etwa zwanzig weitere Properties, die die Klasse in der Funktion eines Range definiert. Ein Objekt vom Typ *E1 CRM Entity* kann, wie aus der Grafik hervorgeht, über Verknüpfungen in der Domain-Range-Richtung mit den Klassen *E41 Appellation*, *E55 Type* und *E62 String* detaillierter beschrieben werden. Wobei die Klasse *E62 String* zu einer der wenigen Ausnahmen zählt, da sie zu den *Primitive Values* des CRM gehört. Das sind die drei Klassen, *E60 Number*, *E61 Time Primitive* und *E62 String*, die weitgehend unstrukturierte Daten aufnehmen können. Sie werden unter der abstrakten Klasse *E59 Primitive Value* subsumiert, die sich auf der gleichen Ebene wie die Klasse *E1 CRM Entity* befindet. Das heißt diese Klassen sind *nicht* in der CRM-Hierarchie enthalten, sie erben daher keine Properties und definieren auch keine eigenen, sie dienen als reine Container.

Ein Ausschnitt der Klassenhierarchie in einer Baumdarstellung soll dies veranschaulichen:³⁸

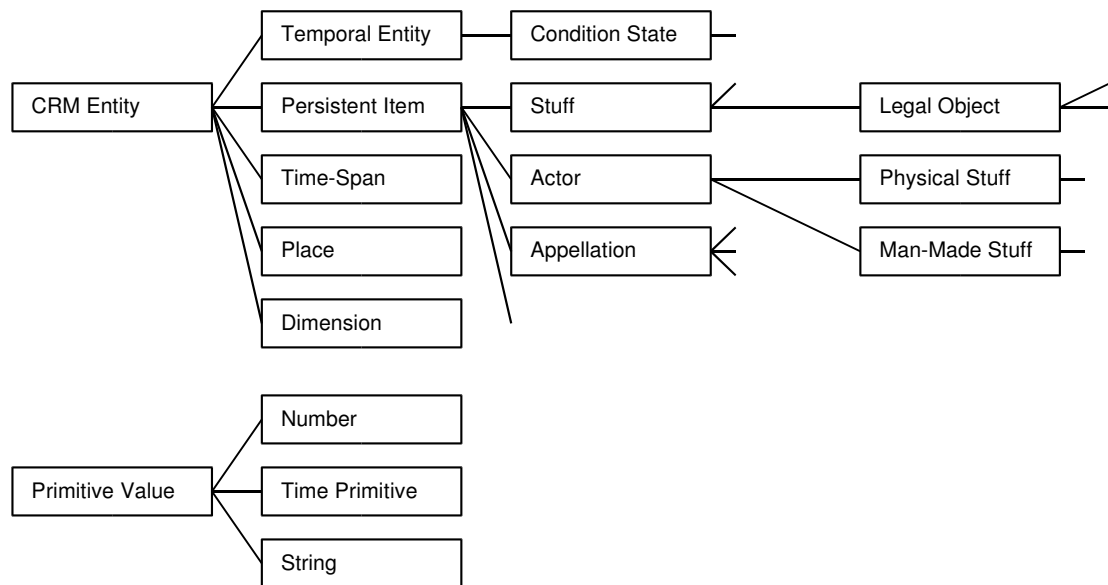


Abbildung 4: Ausschnitt der Klassenhierarchie

Die Linien zwischen den Kästchen sollen jeweils die Verbindung von Mutterklasse und Tochterklassen repräsentieren. Linien, die nicht auf ein weiteres Feld verweisen, sollen weitere Ableitungen aus der Klasse andeuten, die in diesem Ausschnitt nicht dargestellt sind. Klassen, von denen keine weitere Linie abgeht, haben keine Tochterklassen. Deutlich wird aus dieser Abbildung, dass die Klasse *E59 Primitive Value* und ihre Tochterklassen nicht zur Klassenhierarchie des CRM gehören.

Das folgende Beispiel zeigt zwei Objekte, die über das Property *P1 is identified by* (*identifies*) mit einander verknüpft sind. Wiederum sind der Übersichtlichkeit halber nicht alle Properties dargestellt.

³⁸ Üblicherweise werden solche Klassendiagramme mit Pfeilen dargestellt, z.B. in der Unified Modeling Language (UML), worauf an dieser Stelle aus Gründen der Übersichtlichkeit verzichtet wurde. Aus dem gleichen Grund wurde auf die Identifikationsnummern der Klassen verzichtet.

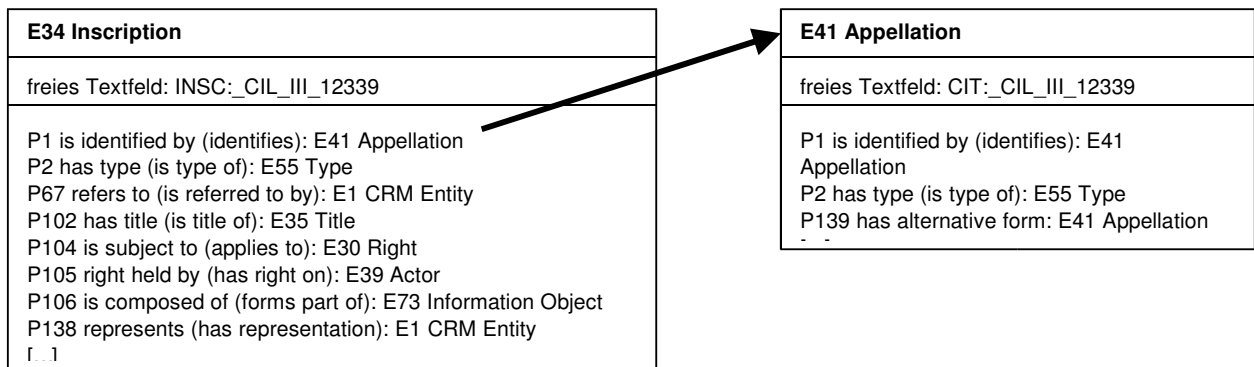


Abbildung 5: Verknüpfung von CRM-Objekten

Anzumerken ist an dieser Stelle, dass das P1-Property des Inscription-Objekts nicht das gleiche ist, wie das des Appellation-Objekts. Das Appellation-Objekt lässt sich ebenfalls über das eigene P1-Property mit einem weiteren Objekt weiter spezifizieren, was theoretisch eine unbegrenzt detaillierte Beschreibung zulässt. Eine solche detailliertere Beschreibung zeigt die folgende Abbildung, die die ‚zweite Ebene‘ mit einbezieht.

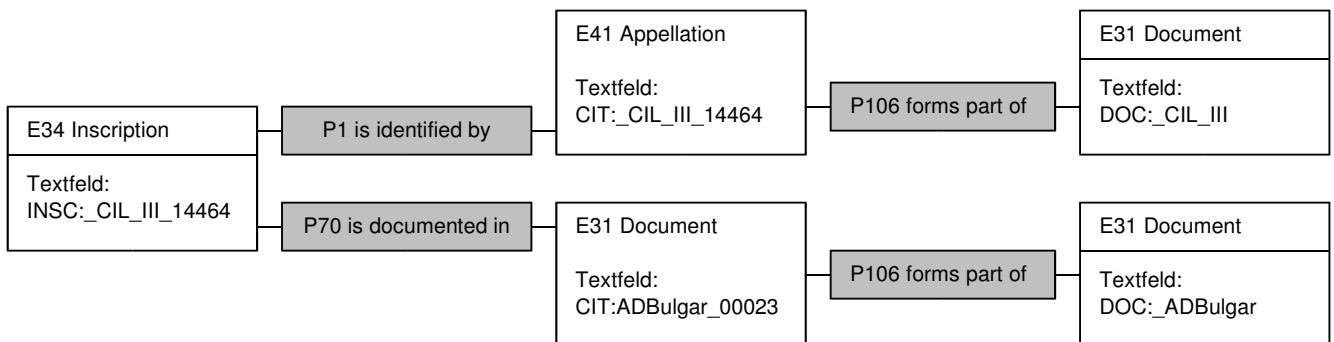


Abbildung 6: Erweiterte Verknüpfung von CRM-Objekten

Es gibt zwar keine ‚Ebenen‘ von Objekten im CRM, geht man aber bei der Betrachtung der Grafik vom *Inscription* Objekt aus, lässt sich durchaus sinnvoll sagen, dass es auf zweiter Ebene vom *Document* Objekt mit dem Textfeld *DOC._CIL_III* beschrieben wird, oder einfach auf zweiter Ebene verknüpft ist.

Aus dem bisher beschriebenen Modell ergeben sich einige Konsequenzen von selbst, andere wurden bei der Modellierung des CRM ausdrücklich beabsichtigt. Die wichtigsten Punkte sollen hier kurz referiert werden, eine genauere Ausführung geben Crofts u.a. (2002).

- Bei der Erstellung von Daten im CRM Format müssen die zugrundeliegenden Konzepte dem Anwender bekannt sein, damit er entscheiden kann, in welchen Geltungsbereich (*Scope*) ein reales Objekt einzuordnen ist. Das CRM liefert keine Möglichkeit, diese Entscheidung formal zu prüfen. Formal prüfen lässt sich nur die Verwendung der Properties. Das semantische Konzept ist nur maschinell verarbeitbar, weil es mit der formalen Definition verknüpft ist. Trotz dieser Verknüpfung darf nicht vergessen werden, dass es *zwei grundverschiedene* Konzepte sind, die eine Klasse beschreiben.
- CRM Konzepte sind semantische Konzepte, das heißt sie lassen sich nicht aus anderen Klassen oder Properties logisch ableiten. Logische Ableitung oder Vererbung kann sich immer nur auf die formale Beschreibung einer Klasse oder eines Property beziehen. CRM Konzepte lassen sich auch nicht zusammensetzen oder teilen, sie sind immer *primitive* Konzepte, die als abgeschlossene Einheiten bestehen.
- Damit die Integration von Daten gewährleistet ist, ist das CRM *monoton* modelliert. Das bedeutet, dass keine Konstruktion seine Gültigkeit verlieren soll, wenn weiteres Wissen hinzukommt, weder in Form von Daten noch in Form von neuen Konzepten.
- Das CRM liefert auch keine Mechanismen, die einmal produzierte Informationen verbessern würden oder den Wahrheitswert veralteter Daten überprüfen könnten. Wie der Wahrheitswert und damit die Qualität von Informationen überprüft wird, ist nicht die Aufgabe der Ontologie selbst, sondern die eines verarbeitenden Programms.
- Geschwisterklassen, also Klassen auf gleicher Ebene der Klassenhierarchie sind grundsätzlich nicht ausschließend, das heißt, ein physisches Objekt kann sehr gut Instanz zweier Geschwisterklassen sein. Dazu gibt es genau zwei Ausnahmen, also zwei Klassenpaare, deren Instanzen disjunkte Menge bilden. Das ist zum einen das Paar *E2 Temporal Entity* und *E77 Persistent Item* und das Paar *E18 Physical Stuff* und *E28 Conceptual Object*.
- Es werden nicht zwingend Komplemente definiert; beispielsweise folgt aus einem Property *former owner* nicht, dass es ein Property *current owner* geben muss.

- Das letzte dieser Merkmale sind die so genannten *Shortcuts*. Shortcuts sind bestimmte Properties, die eine direkte Verbindung zwischen Objekten, also zwischen Domain und Range, herstellen, während gleichzeitig eine solche Verbindung über viele Instanzen und Properties möglich ist:

“Shortcuts, Some properties are declared as ‘shortcuts’ of a path, that connects the same domain and range as the respective property, but leading through multiple properties and classes (normally one intermediate class). The declaration denotes, that all instances of this path can be seen as instances of the ‘shortcut’ property. The opposite is normally not true: It may not be possible to infer the path from the existence of an instance of the shortcut property. In some cases, it may be possible to infer a path with a hypothetical intermediate node which is uniquely defined by a property, domain and range instance.” (Crofts u.a. 2002)

Shortcuts sind also auch Properties, sie zeichnen sich nur dadurch besonders aus, dass sie einen *Pfad*, also die Verbindung von zwei Objekten über mehre Instanzen hinweg, direkt beschreiben. Shortcuts sollen bei der Verbindung von Quellen, wie in Abschnitt 2.2 beschrieben, helfen, indem mit Ihnen fehlende integrierende Informationen umgangen werden.

3.5 Weitere Anwendungen

Das CRM ist bisher vor allem ein theoretisches Modell. Zwar existiert eine Implementierung des Modells in *Telos* (Doerr 10/2001:7), in RDF und ebenso einige weitere Vorschläge zur Darstellung des Modells in anderen Formaten, so genannten Mappings, aber die Anwendung des Modells befindet sich noch in der Entwicklung. Diese Entwicklungen dienen hauptsächlich der Verwendung als Metadatenschema oder als Vorlage für Schemata von Datenbanken.³⁹

Zu den existierenden Mappings gehört unter anderem der angesprochene Dublin Core. Weitere Mappings bestehen für das *Encoded Archival Description* (EAD),⁴⁰ für das Datenmodell des *Art Museum Image Consortium* (AMICO)⁴¹ und für das Functional Requirements for Bibliographic Records (FRBR)⁴² Format. Auf der WWW-Seite des CRM findet sich eine Liste von Dokumenten, die diese Mappings detailliert beschreiben.⁴³ Darüber hinaus arbeitet die *CIDOC Working Group* des CRM zusammen

³⁹ Eine kurze Darstellung der Sprache *Telos* bieten Mylopoulos u.a. (1990).

⁴⁰ *Encoded Archival Description* <<http://lcweb.loc.gov/ead/>> (14.02.04).

⁴¹ *The Art Museum Image Consortium* (AMICO) <<http://www.amico.org/>>, eine Beschreibung des Datenmodells findet man unter: <<http://www.amico.org/AMICOLibrary/dataspec.html>> (14.02.04).

⁴² FEBR <<http://www.ifla.org/VII/s13/frbr/frbr.htm>>. (27.02.04).

⁴³ CRM Technical Papers <http://cidoc.ics.forth.gr/technical_papers.html> (14.02.04).

mit anderen Gruppen im Bereich der Standardisierung von Metadaten.⁴⁴ Im Bereich des Semantic Web existiert eine Zusammenarbeit mit *OntoWeb*,⁴⁵ die das CRM als einen vielversprechenden möglichen Standard für den Informationsaustausch verstehen.

⁴⁴ CRM Collaborations <<http://cidoc.ics.forth.gr/collaborations.html>> (14.02.04).

⁴⁵ *OntoWeb* <<http://ontoweb.aifb.uni-karlsruhe.de>> (14.02.04).

4 Praktische Einsetzbarkeit des CRM

Ein Informationssystem (IS) besteht aus Informationsressourcen, z.B. einer Datenbank, einem Index oder einer Ontologie, aus einer verarbeitenden Anwendung und einer Benutzerschnittstelle, also einer Ein- und Ausgabeanwendung. Bei der Verwendung einer Ontologie lässt sich weiter unterscheiden, ob die Ontologie zum Zeitpunkt der Entwicklung besteht und in Form von hartkodierte Daten in das Informationssystem integriert ist oder ob zur Laufzeit auf eine externe Ontologie zugegriffen wird (Guarino 6/1998:Kap.3). Bei der Verwendung des CRM steht die letztere Möglichkeit im Vordergrund. Eine Anwendung sollte das Modell derart implementieren, dass auf externe Ressourcen zugegriffen werden kann und zwar in der Form, dass für Daten unterschiedlichen (syntaktischen) Formats entsprechende Schnittstellen existieren. Dies ist möglich, weil das Modell als Metaschema eingesetzt werden kann, und es ist nötig, damit Ressourcen dezentral erstellt und bereitgestellt werden können. Schließlich ist vorgesehen, dass CRM Daten z.B. für ein Museum jeweils vor Ort von einem Kurator oder fachkundigen Mitarbeitern erstellt werden, um eine hohe Qualität zu gewährleisten.

Um ein einfaches Informationssystem zu entwickeln, standen für die vorliegende Arbeit Datensätze im RDF- und im XML-Format zur Verfügung.

4.1 Das CRM in RDF

Das Resource Description Framework, das weiter oben bereits angesprochen wurde, bietet die Möglichkeit, das objektorientierte Schema des CRM nachzumodellieren. Insbesondere Vererbung lässt sich innerhalb des RDF beschreiben. Eine offizielle Version der Beschreibung des CRM in RDF wurde mit der Schemasprache des RDF, der RDFS, erstellt und ist über die Homepage des CRM abrufbar.⁴⁶

Mit dieser Möglichkeit, die Klassenhierarchie darzustellen, nimmt man in Kauf zwei Inhaltsmodelle miteinander zu verknüpfen. Ob sich daraus ein weiterer Nutzen ergibt oder ob das Modell dadurch lediglich komplizierter und damit schwieriger zu verarbeiten ist, lässt sich schwerlich sagen, ohne die Möglichkeiten, die sich aus diesen Daten ergeben, praktisch zu testen. Eine Alternative zu dieser Möglichkeit wäre sicherlich, das

⁴⁶ *CIDOC CRM v3.4 Encoded in RDFS* <http://cidoc.ics.forth.gr/rdfs/cidoc_crm_v3.4.9.rdfs> (27.01.04).

CRM direkt in XML Schema darzustellen, denn RDF bzw. RDFS beruht auf der Technologie von XML Schema. Damit wäre es grundsätzlich möglich, die Klassenhierarchie des CRM darzustellen, ohne das RDF zu bemühen und die beiden Inhaltsmodelle miteinander zu verbinden. Andererseits beruhen Topic Maps und OWL, so wie die Vorgänger von OWL, auf RDF, daher ist es ebenso gut möglich, dass es aus Gründen der Interoperabilität sinnvoller ist, gerade RDF oder eine der Ontologie-Sprachen zu verwenden.

In Abschnitt 4.2.4 soll gezeigt werden, dass auch die Verwendung von XML-Schema nicht unbedingt notwendig ist, sondern eine Darstellung in XML, die über eine Document Type Definition (DTD) definiert ist, ausreichen kann. Grundsätzlich ist eine einfache Umsetzung, sofern sie das Gleiche leistet wie eine aufwendige, von Vorteil, denn je einfacher die Darstellung ist, desto leichter lassen sich Daten lesen und verarbeiten, wodurch sie weniger fehleranfällig sind.

Eine Besonderheit der Implementierung in RDF stellt die Behandlung der Properties dar, denn Properties wurden aus folgendem Grund immer zweifach implementiert:

„RDF does not allow to instantiate properties beginning from a range value. Therefore, each CRM property is represented as two RDFS properties. For instance ‘P2 has type (is type of)’ is represented as: ‘P2F.has_type’ for the domain to range direction and ‘P2B.is_type_of’ for the range to domain direction.“⁴⁷

An dieser Stelle wird die Menge der Properties praktisch verdoppelt, auch wenn zwei Properties in der RDF-Fassung die gleiche Relation beschreiben sollen wie das ursprüngliche Property des CRM. Dass die Umsetzung des bidirektionalen Konzepts von Properties nicht ganz unproblematisch ist, wurde schon bei der Beschreibung von Domain und Range (vgl. 3.4.4) angedeutet und soll im folgenden Abschnitt 4.2 mit der Besprechung der XML-Daten eingehender behandelt werden. Dies bietet sich unter anderem deswegen an, weil die Struktur der RDF-Daten sehr ähnlich derjenigen der XML-Daten ist, vermutlich diente eines der Formate als Vorlage zur Erstellung des anderen. Das selbe Vorgehen, eben die Verdoppelung der Properties, taucht also in der XML-Version des CRM wieder auf. Abgesehen von dieser Verdopplung der Properties haben die beiden Ansätze jedoch wenig gemeinsam. Des Weiteren soll die Umsetzung

⁴⁷ ICS-FORTH, 20.11.02, “CIDOC CRM v3.4 Encoded in RDFS”
<http://cidoc.ics.forth.gr/rdfs/cidoc_crm_v3.4.9.rdfs> (27.01.04).

des CRM in RDF nicht weiter vertieft werden, da die XML-Daten im praktischen Teil der Arbeit im Vordergrund standen.

4.2 Das CRM in XML

Die Modellierung einer Klassenhierarchie, beispielsweise die des CRM, ist in XML nicht möglich, da sich objektorientierte Vererbung mit einer DTD nicht formulieren lässt. Schon aus diesem Grund muss sich die Definition der beiden Formate grundsätzlich voneinander unterscheiden.

XML besteht aus ineinander verschachtelten Elementen, die sich sehr gut dazu eignen, Hat-Ein-Relationen auszudrücken. Es existiert in XML aber kein Sprachkonstrukt, welches dafür vorgesehen ist, Ist-Ein-Relationen zu beschreiben. Möglicherweise wurde daher gar nicht versucht eine DTD zu entwickeln, welche XML-Dokumente derart einschränkt, dass sie dem CRM entsprechen. Die offizielle DTD des CRM⁴⁸ ist daher auch *nicht* zur Überprüfung der richtigen Anwendung bzw. Umsetzung des Modells gedacht, dafür existiert eigens ein Tool,⁴⁹ welches die Daten in RDF konvertiert, sondern sie stellt die Definition eines Dokumenten-Typs dar, welcher allein dem Datentransport dienen soll.

Trotzdem spricht einiges dafür XML-Daten im CRM Format zu erstellen: XML Daten eignen sich unter anderem hervorragend zum Transport strukturierter Daten. Außerdem lassen sie sich mit einer Vielzahl von Werkzeugen bearbeiten. Für diese Arbeit lag der ausschlaggebende Grund allerdings schlicht im Format der vorliegenden Test-Daten.

4.2.1 Die Test-Daten

Um die praktische Einsetzbarkeit des Modells zu testen, lagen zunächst zwei Datensätze in XML vor (ca. 10,6 und 11,2 MB), zu denen später noch ein dritter im RDF Format (ca. 15,8 MB) kam. Die Daten für diese Arbeit wurden alle von Herrn Martin Doerr, dem Vorsitzenden der *CIDOC CRM Special Interest Group*,⁵⁰ zur Verfügung gestellt.

⁴⁸ ICS-FORTH, 20.11.02, "CIDOC CRM DTD" <http://cidoc.ics.forth.gr/docs/xml_to_rdfs/new-cidoc.dtd> (27.01.04).

⁴⁹ CIDOC CRM, 23.01.03, <http://cidoc.ics.forth.gr/docs/mapping_tool_4_12_02.zip> (27.01.04).

⁵⁰ CIDOC CRM SIG (Special Interest Group) <http://cidoc.ics.forth.gr/who_we_are.html> (14.02.04).

Der erste Datensatz (im Folgenden LVPA-Daten) bestand aus einer XML-Datei in CRM-Form, die aus einem Auszug der *LVPA-Datenbank* in Wien erstellt wurde. VBI ERAT LVPA⁵¹ ist ein von der EU und der UNESCO gefördertes Projekt, das aus einer Webplattform mit Anbindung an wissenschaftliche Datenbanken besteht, wobei eine der verwendeten Datenbanken die besagte in Wien ist. Das Ziel des Projektes besteht darin, über die Webplattform römische Steindenkmäler aus einer Reihe von Provinzen des ehemaligen römischen Reiches in ihrem architektonischen Kontext zu präsentieren. Es soll ebenso interessierten Laien wie Wissenschaftlern, aber auch Schulen und Museen wissenschaftlich fundierte Informationen benutzergerecht vermitteln und Bild- und Schriftdenkmäler über ein mit der Zeit wachsendes, online verfügbares Bildarchiv zugänglich machen. Zur Zeit enthält das Projekt 6021 Steindenkmäler und 5772 illustrierende Bilder. Einen detaillierteren, dennoch kurzen Überblick über das Projekt bietet Schaller (6/2003).

Der zweite Datensatz (im Folgenden CIL-Daten) ist ein Auszug aus Daten des Corpus Inscriptionum Latinarum (CIL), ebenfalls im XML-Format. Das Corpus Inscriptionum Latinarum wurde im Jahre 1853 unter der Leitung von Theodor Mommsen gegründet und ist seit dem die maßgebliche Dokumentation des epigraphischen Erbes der römischen Antike. Es erfasst die lateinischen Inschriften aus dem gesamten Raum des ehemaligen Imperium Romanum in geographischer und systematischer Ordnung. Seit Anfang 1994 wird das Projekt von der Berlin-Brandenburgischen Akademie der Wissenschaften (BBAW)⁵² geführt. Heute liegen vom Corpus Inscriptionum Latinarum 17 Bände mit ca. 180.000 Inschriften vor.

Der dritte Datensatz (im Folgenden RDF-Daten) scheint nach Stichproben inhaltlich mit den CIL-Daten identisch zu sein, ist aber durch das RDF-Format bedingt, anders strukturiert.

4.2.2 Speichern und Suchen - ein Versuch

Die einfachste Form eines Informationssystems lässt sich mit einer primitiven Suchfunktion über eine Datenressource realisieren. Das geplante Vorgehen bestand also darin, die Dateien zunächst in eine handlichere Größe zu zerlegen, sie in einer Datenbank

⁵¹ VBI ERAT LVPA, 28.01.04, <<http://www.ubi-erat-lupa.org/>> (28.01.04).

⁵² Die Informationen zum Corpus Inscriptionum Latinarum entstammen der WWW-Seite der BBAW <<http://www.bbaw.de/forschung/cil/index.html>> (27.01.04).

zu speichern und ein einfaches Programm zu schreiben, mit der sich eine elementare Suche bewerkstelligen lässt. Im weiteren Vorgehen sollten die Suchmöglichkeiten verfeinert und insbesondere verknüpft werden, um so herauszufinden, wie sich aus den semantischen Relationen Nutzen ziehen lässt.

Die Struktur der Daten soll hier nicht im Detail besprochen werden, es stellte sich aber bei genauerem Hinsehen heraus, dass es sich um drei verschiedene Ausführungen handelt, also jeder Datensatz in einem leicht verschiedenen Format vorlag.⁵³ Die Aufgabe bestand also zunächst darin, sich für ein Format zu entscheiden, das heißt zu erkennen, welches Format dem Modell genügt. Zwar beschreiben die RDF-Daten das Modell am besten, da sie jedoch später vorlagen, lag der Schwerpunkt dieser Arbeit von Beginn an in der Verarbeitung von XML-Daten. Die beiden Datensätze in XML hingegen entsprechen in einigen Punkten nicht der Definition des CRM, was leider erst im Verlauf der Arbeit deutlich wurde. Vor einer möglichen Arbeit *mit* dem Daten, mussten diese also zunächst selbst bearbeitet werden.

4.2.3 Bearbeitung der Daten

Die Transformation der beiden XML-Datensätze in ein gültiges CRM-Format war ohne größeren Aufwand nicht möglich. Sie enthalten zwar interessanteres Material, da unterschiedliche Objekte, nämlich verschiedene Steindenkmäler an unterschiedlichen Orten, beschrieben werden, ließen sich aber leider nicht verwenden.

Da der RDF-Datensatz zwar im Wesentlichen dem CRM entspricht, auf sämtliche Sprachkonstrukte des RDF jedoch verzichtet werden sollte, bot es sich an, die RDF-Elemente aus diesem Datensatz schlicht zu entfernen, ohne dabei inhaltliche Änderungen an den Daten vorzunehmen. Bis auf wenige kleine weitere Änderungen, beispielsweise die Transformation des Attributs *rdf:about* in das einfache Attribut *content*, ließen sich diese RDF-Daten problemlos transformieren. Damit dienten sie als direkte Vorlage für das im folgenden Abschnitt 4.2.4 entwickelte XML-Format. Es erlaubt, beliebig viele einzelne CRM-Objekte ungeordnet gewissermaßen ‚aneinandergereiht‘ in eine Datei zu schreiben.

⁵³ Ein viertes Format, das ebenfalls der offiziellen DTD genügt, findet sich in Doerr/Karvasonas (2001).

Durch die entwickelte DTD bedingt (vgl. Anhang A) müssen Properties in einer geordneten Reihenfolge auftreten. Diese Reihenfolge war in den RDF-Daten nicht vorgesehen und musste von Hand geändert werden, was für die ursprüngliche rund 15,8 MB große Datei mit einigen tausend Inschriften nicht möglich war.

Die erstellte Testdatei ist mit 184 KB deutlich kleiner. Sie enthält statt den mehreren tausend genau 186 Inschriften, von denen jede durch ein CRM-Objekt repräsentiert ist. Insgesamt werden diese 186 physikalischen Objekte von 1256 CRM-Objekten beschrieben. Die Kürzung der Daten ist insofern irrelevant, als dass die Inschriften in nahezu der gleichen Form beschrieben sind und sich der Aufbau der CRM-Objekte also ständig wiederholt.

4.2.4 Entwurf einer neuen DTD

Wie weiter oben angesprochen, dient die bestehende offizielle DTD nicht der Validierung des Modells, sondern nur dem Transport der Daten. Es werden darin keine Klassen beschrieben, was im objektorientierten Sinne auch nicht möglich ist.

Sie beinhaltet keine sinnvolle Beschränkung der Möglichkeiten, sondern enthält nicht mehr als eine Aufzählung von Elementen, die den CRM-Properties entsprechen sollen. Eine Umsetzung von CRM-Klassen in XML-Konstrukte gibt es nicht. Zur Beschreibung von Klassen dient einzig das Element `<in_class>`, das einen beliebigen Klassennamen zum Inhalt haben kann. Außerdem wird gerade dieses `in_class`-Element auf zwei sehr unterschiedliche Arten verwendet, es lassen sich damit sowohl Hat-Ein- als auch Ist-Ein-Beziehungen ausdrücken, so dass es beim Parsen des Dokuments praktisch nicht möglich ist, die aktuelle Funktion zu bestimmen. Des Weiteren wird ein XML-Entity definiert, das eine nahezu unbeschränkte Verschachtelung der Elemente erlaubt, was an dieser Stelle und auf diese Art und Weise nicht im Sinne des CRM ist.

Innerhalb der bestehenden Document Type Definition können CRM-Daten also grundsätzlich dargestellt werden, nur leider auch noch weit mehr darüber hinaus. Darin liegt vermutlich der Grund, dass die drei zur Verfügung stehenden Datensätze alle voneinander abweichend strukturiert sind.

Die DTD, die im Folgenden vorgestellt wird, existiert bisher nur als Ansatz, sie ist gewissermaßen ein Subset einer vollständigen Ausführung, es sind darin lediglich die-

jenigen Elemente definiert, die die Klassen und Properties beschreiben, welche in den Beispieldaten Verwendung fanden. Nichtsdestotrotz entsprechen die gegen diese DTD validierten XML-Daten dem CRM.

Die DTD soll ermöglichen, ein einzelnes oder beliebig viele CRM-Objekte in XML darzustellen, und gleichzeitig die falsche Verwendung von Elementen unterbinden. Des Weiteren soll sie möglichst ohne Konstrukte, die nicht bereits im CRM enthalten sind, auskommen; das heißt XML-Elemente und -Entities sollen nur CRM-Entities und -Properties beschreiben. Demnach wird ein einziges Element definiert, welches nicht dem CRM entstammt: das Element `<CRMSet>`. Es übernimmt eine reine Containerfunktion und ist das so genannte Wurzelement des Dokuments. Die Einführung eines Namespace ist nicht notwendig, da sämtlich Elemente und Attribute des Modells sehr spezielle Name haben und eine Verwechslung mit anderen Modellen nahezu ausgeschlossen ist. Außerdem ist die Vermischung verschiedener XML-Modelle auch für die Zukunft nicht angestrebt.

Mit Hilfe der DTD werden Domain und Range eines jeden Property eindeutig bestimmt, womit gewährleistet ist, dass Properties ihrer Definition entsprechend benutzt werden. Damit wiederum ist garantiert, dass jedes CRM-Objekt der Definition des CRM entsprechend weiter verknüpfbar ist und je nach Anforderung spezifiziert werden kann.

Die DTD ist allerdings *nicht* dazu geeignet, die Klassenhierarchie ausdrücken. Das ist an dieser Stelle aber auch nicht notwendig, denn XML-Daten verarbeiten keine Informationen. Es gibt keinen ‚XML-Prozessor‘ und keinen ‚XML-Compiler‘, der auf die Struktur der Klassenhierarchie zurückgreifen müsste. XML Daten dienen nur der (sortierten) Speicherung oder dem Transport, und dazu ist es unbedingt notwendig, dass die XML-Daten nicht gegen das CRM verstoßen, das genau lässt sich mit der Validierung gegen die beschriebene DTD sicherstellen.

XML-Dokumente, die sich gegen diese DTD validieren lassen, sind folgendermaßen aufgebaut:

1. Für jedes *Klassenelement*, also XML-Elemente, die Objekte vom Typ einer CRM-Klasse repräsentieren, lassen sich unstrukturierte, das heißt nicht spezifizierte Informationen als Wert des Attributs `content` einfügen.

2. Relationen zu anderen Entities werden einzig mit denen der Klasse zugehörigen Properties ausgedrückt, das heißt Klasselemente enthalten immer nur Property-Elemente.
3. Klasselemente enthalten jedes Property-Element höchstens einmal, das heißt mehrere Verknüpfungen über das gleiche Property werden innerhalb eines Elements ausgedrückt.
4. Kein Entity einer beliebigen Klasse enthält ein anderes Entity, wie das teilweise in den unterschiedlichen Varianten der ursprünglichen Datensätzen der Fall war.

```

<CRMSet>                                <!-- reiner Container, Wurzelement für XML -->
  <class1 content="">                    <!-- erstes CRM-Entity, hier Domain -->
    <prop1>                              <!-- zur Klasse gehöriges Property -->
      <class24 content=""/>              <!-- Klasse aus dem Range des jeweiligen Property-->
    </prop1>
    <prop2>                              <!-- weitere Properties -->
      <class2 content=""/>
      <class8 content=""/>
    </prop2>
    <propN>
      <classN content=""/>
    </propN>
  </class1>
  <class2 content="">                    <!-- nächstes Entity/Objekt -->
    <propX>
      <classX content=""/>
    </propX>
    <propN>                              <!-- Property -->
      <classN content="">                <!-- Range UND Domain -->
        <prop2>                          <!-- Property -->
          <class2 content=""/>            <!-- Range UND Domain -->
        <prop1>                          <!-- Property -->
          <class24 content=""/>          <!-- Range -->
        </prop1>
          <class3 content=""/>
          <class8 content=""/>
        </prop2>
      </classN>
    </propN>
  </class2>
</CRMSet>

```

Abbildung 7: Generisches Datenbeispiel

Damit entspricht in jedem Fall ein gültiges XML-Dokument dieses Formats der CRM-Definition, auch wenn die DTD keine Klassenhierarchie zu beschreiben vermag. Die objektorientierte Definition des CRM sagt tatsächlich viel mehr aus, aber es ist, wie

ei-

te

nn

schon erwähnt, nicht notwendig, das CRM in einer DTD oder einem XML-Schema darzustellen, um Dateien in einer Form zu strukturieren, die dem Modell voll genügt.

Um das zu verdeutlichen, muss man die Unterscheidung der Ist-Ein- und der Hat-Ein-Relation wieder aufnehmen. Die CRM-Properties werden allesamt durch Hat-Ein-Relationen innerhalb ihrer Klassendefinition beschrieben. Diese Relationen sind in XML, wie es oben gezeigt wurde, einfach abzubilden, indem einem Element ein Kind-element untergeordnet wird. Dazu nochmals ein Beispiel, das der CRM-Definition (Crofts u.a. 2002) entnommen ist:

(4) *The capital of Italy (E53 Place) is identified by Rome (E48 Place Name)*

E53 Place hat ein Property *is identified by*, über dieses Property wird dem Objekt *E53 Place* das Attribut *E48 Place Name* zugeordnet. Dies bedeutet aber nicht, dass *E53 Place* dieses Objekt der Klasse *E48 Place Name* ‚hat‘, sondern dass ihm in der durch das Property definierten Art ein solches Objekt als Attribut zugeschrieben wird.

Tatsächlich drücken viele der CRM-Properties im semantischen Sinne Hat-Ein-Relationen aus, aber eben nicht im objektorientierten. Beispielsweise hat *E70 Stuff* ein Property *P43 has dimension*, welches sich auf ein Objekt der Klasse *E54 Dimension* bezieht. Es ist eine wichtige Unterscheidung, dass ein Objekt vom Typ *E70 Stuff* keinesfalls ein Objekt *E54 Dimension* hat. Zwar gibt es grundsätzlich in objektorientierten Modellen auch die Möglichkeit, dass ein Objekt ein anders *hat*, aber diese Relation kommt im CRM *nicht* vor: Kein Objekt einer Klasse enthält ein anderes Objekt. Objekte enthalten immer nur Properties, das heißt CRM Klassen definieren immer *primitive* Datentypen im objektorientierten Sinne. Da demnach Objekte im CRM überhaupt nicht aus verschiedenen Objekten bestehen dürfen, muss diese Möglichkeit auch nicht beschrieben werden, weswegen das problematische `in_class`-Element hinfällig ist.

Wie in 3.4.1.5 dargestellt wurde beschreiben Ist-Ein-Relation innerhalb des CRM Vererbung. Vererbung lässt sich in XML nicht darstellen, indem eine bestimmte Klasse um zusätzliche Eigenschaften erweitert wird. Möglich ist jedoch ein andere Art und Weise, nämlich dass die Beschreibung aller Eigenschaften der Mutterklasse in der Tochterklasse wiederholt wird. Damit bleibt die eindeutige Zuordnung aller Objekte zu einer bestimmten Klasse möglich, was nichts anderes bedeutet, als dass auf diese Art die

Klasse definiert wird. Verloren geht in der Tat die Beschreibung der Relation der Klassen untereinander, aber nicht die Relation selbst. Sie bleibt bestehen und gültig.

Abbildung 8 zeigt einen Ausschnitt aus der vorläufigen DTD, in dem drei Elemente definiert werden, die jeweils eine Klasse repräsentieren.

```
<!ELEMENT E1_CRM_Entity (P1F_is_identified_by?, P2F_has_type?, P3_has_note?)>
<!ATTLIST E1_CRM_Entity
  content CDATA #REQUIRED
>
<!ELEMENT E31_Document (P1F_is_identified_by?, P2F_has_type?, P3_has_note?,
P106B_forms_part_of?)>
<!ATTLIST E31_Document
  content CDATA #REQUIRED
>
<!ELEMENT E41_Appellation (P1F_is_identified_by?, P2F_has_type?, P3_has_note?,
P70B_is_documented_in?, P106B_forms_part_of?)>
<!ATTLIST E41_Appellation
  content CDATA #REQUIRED
>
```

Abbildung 8: Ausschnitt aus der DTD

Die Klasse *E1 Document* stellt eine Erweiterung der Klasse *E1 CRMEntity* dar, indem sie die ursprüngliche Klassendefinition um das Property *P106 forms part of* erweitert. Es lassen sich an dieser Stelle leider keine XML-Entities (das sind selbstdefinierte Mengen) verwenden, da die Reihenfolge der Properties in der Definition relevant ist.

Würde man beispielsweise aus dem Element `<E31_Document>` ein Entity ableiten, um dies in der Definition des Elements `<E41_Appellation>` zu verwenden, stünde dort das Element *P106* vor dem Property *P70*. Die im jeweiligen Element neu zu definierenden Property-Elemente würden also einfach angehängt. Das aber soll genau vermieden werden, da es zwingend ist, die Reihenfolge bei der Benutzung der Elemente einzuhalten, und die Sortierung nach Identifikationsnummern dazu äußerst hilfreich ist. Damit ist die Definition der Klassen abgeschlossen, es fehlt noch eine angemessene Darstellung der Properties einschließlich der Ranges.

Ein Range, so wurde in Abschnitt 3.4.4 gesagt, hat in etwa die Funktion eines grammatischen Objekts. Es ist die Klasse, auf die sich ein Property bezieht, bzw. eine ihrer Tochterklassen. Ranges lassen sich mit einer einfachen Aufzählung aller zugehöri-

ei-
te
nn

gen Klassen beschreiben. Und da hierbei keine Reihenfolge eingehalten werden muss, lassen sich in diesem Fall XML-Entities verwenden (Abbildung 9).

```
<!ENTITY % R55_Type "E55_Type|E56_Language|E57_Material|E58_Measurement_Unit">
```

Abbildung 9: Entity-Definition

Mit der Darstellung der Ranges ist im Grunde auch die Beschreibung der Properties gelöst. Sie lassen sich nun nahezu unverändert aus der CRM-Definition übernehmen. In Abbildung 10 ist wiederum ein Ausschnitt der vorläufigen DTD zu sehen, der die Definition einiger Properties zeigt.

```
<!ELEMENT P1F_is_identified_by (%R41_Appellation;)+>  
<!ELEMENT P1B_identifies (%R1_CRM_Entity;)+>  
<!ELEMENT P2F_has_type (%R55_Type;)+>  
<!ELEMENT P2B_is_type_of (%R1_CRM_Entity;)+>
```

Abbildung 10: Property-Definition

Die Namen der Properties wurden sowohl in der RDF-Fassung als auch in der offiziellen DTD aus jeweils einer Hälfte des Namens wie er in der CRM-Definition festgelegt ist, übernommen und um ein zusätzliches *F* für die Domain-Range-Richtung und ein *B* für die Inversion erweitert. Diese Erweiterung um einen Buchstaben erscheint zunächst redundant und daher nicht empfehlenswert, denn die Richtung eines Properties geht schließlich aus der Semantik des Namens deutlich hervor. Es stellte sich jedoch heraus, dass diese vermeintlich doppelte Markierung, insbesondere bei der Programmierung, durchaus hilfreich ist.

An dieser Stelle möchte ich auf das bereits angesprochene Problem der Verdopplung der Properties zurückkommen. In Abbildung 10 ist zu sehen, dass die zweifache Ausführung der Properties auch in dieser Version der DTD übernommen wurde. Der Grund dafür war zunächst der, dass die ursprünglichen Daten nicht mehr als erforderlich verändert werden sollten.

Es ist jedoch eine grundlegende Frage, ob es überhaupt nötig ist, jede Relation zweifach zu definieren, oder ob nicht die Definition des CRM bloß eine Relation definiert, wo in Wirklichkeit zwei sind. Die Unterscheidung von Domain und Range spricht für die letztere These, denn auch wenn die Unterscheidung laut Definition rein konventionell ist, ist sie von Bedeutung.

Angenommen, man möchte von einem Objekt des Typs *E1 CRM Entity* die Verbindung mit einem Objekt durch das Property *is identified by (identifies)* herstellen, dann ist es notwendig, dass auf der anderen Seite ein Objekt vom Typ *E41 Appellation* steht und nicht ein weiteres *E1 CRM Entity*. Die Richtung bestimmt also zumindest die mögliche Anwendung.

Ein weiteres, nicht so schwerwiegendes, aber dennoch bedeutendes Argument für die Gliederung eines Properties in zwei Relationen ist die Lesbarkeit der Dokumente. Für den Benutzer ist es angenehmer, wenn auch mehrfach verkettete Verknüpfungen immer in eine Richtung beschrieben werden können, so dass es sich in gewohnter Weise von links nach rechts lesen lässt.

Eine stärkeres Argument bietet das folgende Beispiel, das zwar die ursprünglich intendierte Relation darstellt, sich aber zumindest mit einer DTD nicht sinnvoll beschreiben lässt.

(5) *E1 CRM Entity: P1 is identified by (identifies): E41 Appellation*
E41 Appellation: P1 is identified by (identifies): E1 CRM Entity

Mit einem Vorgriff auf die Umsetzung in Java,⁵⁴ einer von Grund auf objektorientierten Programmiersprache, mit der sich also alle objektorientierten Strukturen modellieren lassen sollten, möchte ich die Diskussion abschließen: Das Beispiel (5), ließe sich in Java zwar umsetzen, wäre aber tatsächlich das gleiche wie Beispiel (6), da in unterschiedlichen Klassen definierte Methoden per definitionem nicht gleich sind, auch wenn sie den gleichen Namen tragen und die gleiche Funktion erfüllen.

(6) *E1 CRM Entity: P1 is identified by: E41 Appellation*
E41 Appellation: P1 identifies: E1 CRM Entity

Ein CRM-Entity wird auch in Java jeweils von einem (Java-)Objekt repräsentiert, während Properties in Form von Methoden implementiert wurden. Mit Hilfe der Methoden werden auch die Java-Objekte miteinander verbunden, dazu müssen in Java alle Objekte, die sich gegenseitig ‚kennen‘ sollen auch gegenseitig miteinander bekannt gemacht werden. Damit ließe sich abschließend sagen, dass es durchaus sinnvoll ist, von *einem* bidirektionalen Property als Konzept zu sprechen, gewissermaßen einem generi-

⁵⁴ Java <<http://java.sun.com/>> siehe auch: Krüger (2003) und Goll/Weiß/Rothländer (2002).

schen Property, und hiervon die Realisierung in Form eines unidirektionalen *Property*s und einer unidirektionalen *Referenz* zu unterscheiden (vgl. Kap. 3.4.4).

Damit sind alle Elemente, die benötigt werden, das CRM angemessen in XML darzustellen, beschrieben worden, ohne dass dabei ein genuin objektorientiertes Konstrukt verwendet wurde. Möglich ist das, weil die Anwendung der Klassenhierarchie auf einer anderen Ebene stattfindet, nämlich auf der verarbeitenden Ebene, also einem Programm, welches Daten und Modell verbinden kann. Eine Anwendung, z.B. ein Java-Programm innerhalb eines Informationssystems, das Daten einliest und sie in einer irgendeiner Form bearbeitet, ‚kennt‘ die Klassenhierarchie und kann sie benutzen.

Zum Schluss der Besprechung des XML-Teils noch eine Bemerkung zur Möglichkeit der Darstellung des CRM mit XML-Schema. Diese Darstellung ist grundsätzlich möglich, und sie ist vermutlich in einigen Punkten angemessener oder eleganter als die vorgeschlagene Lösung mit Hilfe der DTD. Zu Beginn der Arbeit mit den XML-Dateien war jedoch nicht klar, dass deren Verwendung auf eine vollständige Überarbeitung der DTD hinauslief und schließlich konnte gezeigt werden, dass die DTD das leistet, was eine Definition leisten soll.

Es lässt sich festhalten, dass das geplante Vorgehen, nämlich Speicherung der Dateien in eine XML-Datenbank und eine erweiterte Suche über einfache semantische Verknüpfungen, mit Daten im vorgeschlagenen Format durchaus umzusetzen ist, aber verschiedene Nachteile mit sich bringt auf die ich in Abschnitt 4.3.6.1 zurückkommen werde.

Mit dem beschriebenen Format und dem transformierten Datensatz ist *ein* Teil der Voraussetzungen für eine Verwendung der CRM-Daten geschaffen worden. Eine Anwendung jedoch, die auf die objektorientierte Struktur des CRM zurückgreifen soll, z.B. für eine Suchanfrage, benötigt eine andere Repräsentation des Modells.

4.3 Das CRM in Java

In Java lässt sich die objektorientierte Struktur des CRM in einer Klassenbibliothek problemlos nachmodellieren. Der Sinn einer solchen Bibliothek besteht darin, CRM-Objekte in Java darzustellen und damit auch in Java verarbeitbar zu machen. Da Java eine Programmiersprache ist, bietet sie, anders als XML, nicht nur die Möglichkeit, die Klassenhierarchie abzubilden, sondern es lassen sich auch beliebige Werkzeuge er-

stellen, welche auf die Hierarchie zurückgreifen. Außerdem ist es möglich, CRM-Objekte direkt in Form von Java-Objekten zu speichern oder zu transportieren ohne den Umweg über ein anderes Datenformat. Schließlich bietet sich an, eine Schnittstelle zu dem im Abschnitt 2.2.1 angesprochenen JTP-API zu schaffen.

Das CRM ist zwar ein theoretisches Modell, das keinerlei Vorgaben in Bezug auf das Datenformat oder die Darstellung macht, aber die Implementierung mit einer objekt-orientierten Programmiersprache kommt dem Modell sicherlich am nächsten, indem jeweils ein Java-Objekt ein CRM-Objekt repräsentiert.

In der in Abschnitt 3.4.2 besprochenen Darstellung wurde bereits darauf aufmerksam gemacht, dass es sich immer um zwei Konzepte handelt, die in einer Klasse vereint werden. Das semantische Konzept wird jeweils durch die Scope Note beschrieben und das formale durch die Bestimmung der zur Klasse gehörigen Properties. Während das semantische Konzept nicht weiter formalisierbar ist, als bereits geschehen, lässt sich das formale Konzept in Java eins zu eins verwirklichen.

4.3.1 Aufbau der Klassenbibliothek

Die Klassen- und Property-Namen des CRM müssen wie bereits in XML auch in Java angepasst und werden. Klassennamen dürfen in Java keine Leerzeichen enthalten, weswegen bei Namen, die aus mehreren Wörtern bestehen, die Wörter üblicherweise ohne Leerzeichen aneinander geschrieben werden, wobei jedes neue Wort mit einem Großbuchstaben beginnt. Also statt `E1 CRM Entity`, heißt es `E1CRMEntity`. Interfaces (s.u.) tragen die gleichen Namen wie diejenigen Klassen, die direkt aus ihnen abgeleitet werden, jedoch ohne das führende *E* und die Nummerierung. Die Klasse `E1CRMEntity` wird demnach aus dem Interface `CRMEntity` abgeleitet (vgl. Anhang B).

Bei Properties wird im Wesentlichen genauso wie bei Klassen verfahren, allerdings wurde die zweifache Ausführung der Properties konsequent fortgesetzt. Das führende *P* im Namen wurde jeweils durch ein *F* bzw. *B* ersetzt. Hinzu kommt, dass es in Java üblich ist, Methoden- und Variablenamen mit Kleinbuchstaben zu beginnen, daher heißt es `f1IsIdentifiedBy` statt `F1 is identified by`.

Es wurde weiter oben gesagt, dass sich das CRM problemlos in Java übertragen ließe, was mit einer kleinen Einschränkung auch zutreffend ist. Diese Einschränkung

betrifft die in Java verwendeten Interfaces, die an sich kein Problem darstellen, sondern zur Lösung eines solchen beitragen.

Es gibt im CRM Mehrfachvererbung, das heißt, dass eine Klasse aus mehreren Klassen gleichzeitig abgeleitet sein kann. Mehrfachvererbung ist in Java nicht vorgesehen und nur über einen Umweg zu erreichen. Dieser Umweg besteht aus einem Interface. Interfaces definieren die Struktur einer Klasse, ohne dabei Code zu enthalten. Eine Klasse, die ein Interface implementiert, ist verpflichtet, alle Methoden dieses Interface zu implementieren. Eine Klasse kann mehrere Interfaces implementieren und muss dann allen Interfaces genügen, das heißt sie kann gewissermaßen aus mehreren Interfaces erben. Auf diese Art und Weise wird die Mehrfachvererbung ermöglicht. Dies ist die einzige Aufgabe der Interfaces.

Von den beschriebenen Unterschieden abgesehen, ist die Klassenbibliothek genauso regelmäßig wie ihre Vorlage, das CRM selbst, aufgebaut und lässt sich in wenigen Punkte zusammenfassen:

1. Alle Klassen erweitern genau eine Klasse. Mit zwei Ausnahmen: `ElCRMEntity`, die Superklasse aller CRM-Klassen ist und `E59PrimitiveValue`.
2. Einige Klassen implementieren zusätzlich ein Interface gleichen Namens.
3. Ein dritter Typ von Klasse implementiert zusätzlich ein zweites Interface, womit Mehrfachvererbung ermöglicht wird.
4. CRM Properties werden in Java als Methoden implementiert, und zwar immer zwei Methoden für ein Property

4.3.2 Stand der Klassenbibliothek

Die Klassenbibliothek umfasst zwar bereits alle Klassen des CRM, aber nicht in ihrer vollen Funktionalität. Das ließ sich im Bearbeitungszeitraum leider nicht bewerkstelligen. Zur Zeit sind die Klassen des CRM in der Version 3.4 (Crofts u.a. 2002) enthalten und alle Methoden-Signaturen für die Properties der Domain-Range-Richtung nicht aber deren Inversion. Eine Signatur ist in diesem Zusammenhang der Methodenkopf, das bedeutet, die Methoden sind nur definiert, nicht aber ausprogrammiert worden. Dabei ist die Programmierung der Methoden ausgesprochen einfach, kurz und immer gleich, sie muss jedoch rund 260 mal vorgenommen werden. In den beschriebenen Testdaten kom-

men nur etwa zwanzig Methoden zur Anwendung, die exemplarisch ausprogrammiert sind, so dass sich die Anwendbarkeit demonstrieren lässt.

Für diese Klassenbibliothek, das CRM-API, existiert eine standardisierte HTML-Dokumentation, die den aktuellen Stand der Bibliothek dokumentiert.

4.3.3 Klassen

Eine CRM-Klasse in Java entspricht ihrer formalen Definitionen wie in Abschnitt 3.4.2 besprochen, das heißt sie definiert alle Properties und zwar in Form von Methoden. Das objektorientierte Modell wird vollständig unterstützt, es müssen daher keine besonderen Strategien wie in XML verfolgt werden um Vererbung zu beschreiben. Genau wie im zugrundeliegenden Modell definiert daher die Klasse `ElCRMEntity` ein Textfeld, in Java eine `String`-Variable namens `content`, das an alle Tochterklassen vererbt wird. Neben dieser Variablen definiert die Klasse Methoden für die drei Properties der Domain-Range-Richtung und bisher zwei Methoden für Properties in die entgegengesetzte Richtung. Zusätzlich wird zu jeder Methode ein `Vector`⁵⁵ definiert, in dem die Relationen zu anderen Objekten gespeichert werden können, dazu mehr im folgenden Abschnitt 4.3.4.

Selbstverständlich könnten in Java noch weitere Methoden innerhalb einer Klasse implementiert werden, deren Funktionen weit über die des CRM hinausgehen. Inwieweit das nötig ist, wird sich im Verlauf der Verwendung zeigen. Voraussichtlich werden Erweiterungen der Funktionalität allerdings in zusätzliche Hilfsklassen ausgelagert, so dass die Klassen im Kern der Bibliothek kaum mehr an Funktionen bieten, als dies für CRM-Klassen vorgesehen ist. Vorläufig existieren drei zusätzliche Methoden (abgesehen von denen, welche standardmäßig aus der Klasse `Object`, der Wurzel der Java-Klassenhierarchie, geerbt werden). Das sind die Methoden: `toString()`, `log()` und `logAll()`, die jeweils unterschiedlich detaillierte Beschreibungen eines Objekts liefern und für Prüfzwecke vorgesehen sind (siehe Anhang B).

4.3.4 Properties als Methoden

Properties werden in Java durch jeweils zwei Methoden repräsentiert. Verknüpfungen werden immer von einem Objekt aus hergestellt, das heißt, ein Java-Objekt ruft eine seiner Methoden auf, der als Argument jeweils ein anderes Objekt übergeben wird. In der

⁵⁵ Ein `Vector` ist eine Speicherstruktur in Java, die beliebige Objekte aufnehmen kann.

Methode wird zunächst eine Referenz des Argument-Objekts in einem Vector des aufrufenden Objekts gespeichert; im Anschluss daran übergibt das aufrufende Objekt dem Argument-Objekt eine Referenz auf sich selbst. Damit sind die beiden Objekte gegenseitig *registriert*. Der Aufbau einer Methode ist seiner Funktion entsprechend einfach, wie Abbildung 11 zeigt, in der die beiden Methoden des Property *P2 has type (is type of): E55 Type* abgebildet sind:⁵⁶

```
/**
 * f2 has type E55 Type
 */
public void f2HasType(E55Type type) {
    f2.add(type);
    type.b2.add(this);
}

/**
 * b2 is type of: E1CRMEntity
 */
public void b2IsTypeOf(E1CRMEntity crmEntity){
    b2.add(crmEntity);
    crmEntity.f2.add(this);
}
```

Abbildung 11: Java-Methoden

Die Vektoren, welche die Objekte aufnehmen, werden immer mit dem ersten Teil des entsprechenden Methodennamens bezeichnet. Im Beispiel ist das *f2* für die primär Richtung (*forward*) und *b2* für die Inversion (*backward*). Die Verwendung der Methoden ist mit der Registrierung abgeschlossen, im weiteren Verlauf werden nur noch die gespeicherten Referenzen auf die Objekte benutzt.

Solange die Java-Objekte in einer Anwendung existieren, also zur Laufzeit des Programms, enthalten alle Vektoren eines Objekts nur Referenzen auf die in ihnen gespeicherten Objekte. Jedes Java-Objekt wird nur einmal erzeugt, aber beliebig oft referenziert und solange im Speicher vorgehalten, wie es benötigt wird. In dem Augenblick aber, in dem ein Objekt z.B. in eine Datei gespeichert wird, werden alle damit verknüpften Objekte und die mit diesen Objekten verknüpften Objekte usw. ebenfalls gespeichert. Dieser Umstand ist der Serialisierung von Objekten in Java zu verdanken, die erlaubt, dass Objekte *vollständig* gespeichert werden können. Bei der Deserialisierung, das ist

⁵⁶ Die beiden Methoden stehen hier nur zum Vergleich beieinander, sie sind nicht in der gleichen Klasse definiert, sondern in den Klassen *E1CRMEntity* und *E55Type*.

das Einlesen gespeicherter Java-Objekte, wird man einen Mechanismus brauchen, der identische CRM-Objekte erkennen und zusammenführen kann.

Man darf keinesfalls vergessen, dass diese Java-Objekte mehr als nur eine Darstellung von CRM-Objekten sind. Sie sind in gewisser Weise zwei *Arten* von Objekten gleichzeitig; und damit ist an dieser Stelle *nicht* die Polymorphie gemeint. Es ist durchaus möglich, dass es mehrere Java-Repräsentationen eines CRM-Objekts gibt, die sowohl identisch als auch verschieden sein könnten.

4.3.5 Erweiterung der Klassenbibliothek

Die Klassenbibliothek ist ein erster Entwurf. Die Unterteilung in mehrere kleine Packages⁵⁷ beispielsweise schien zu Beginn der Arbeit sinnvoll, weil sie eine Orientierung in den 80 Klassen bietet. Sie birgt aber auch Nachteile, insbesondere für die Datenkapselung. Aus der bisherigen Arbeit lassen sich zwei Punkte zu einer sinnvollen Grundstruktur festhalten.

Erstens kann man davon ausgehen, dass der Kern der Bibliothek, der die CRM-Klassen enthält, noch wachsen wird. Daher sollte dieser Kern in *einem* Package zusammengefasst werden und möglichst keine anderen Klassen enthalten. So kann die Menge der Klassen zunehmen und bildet gleichzeitig eine Einheit. Zweitens lassen sich dadurch, dass die Kernklassen alle gleich aufgebaut sind, definierte Schnittstellen schaffen, welche einen Zugriff von außen auf die Inhalte der Objekte erlauben. Diese Schnittstellen sollten in einem weiteren Package zusammengefasst werden.

Es gibt noch zahlreiche weitere Stellen die ausgearbeitet werden können. Dass z.B. Objekte nahezu beliebig miteinander verknüpfbar sind, wirft ein Problem auf. Objekte *müssen* Zugriff auf die Datenfelder anderer Objekte haben, während dieser Zugriff gleichzeitig so eng wie möglich beschränkt sein sollte, damit die gleichen Objekte geschützt sind.

4.3.6 Werkzeuge

Die in folgenden beschriebenen Programme sind keine Anwendungen im herkömmlichen Sinne mit einer Reihe von Funktionen, sondern kleinere Werkzeuge, die dazu dienen,

⁵⁷ Ein Package ist in Java Gliederungsebene einer Bibliothek, mit Packages lassen sich u.a. Sichtbarkeit und Zugriffsmöglichkeiten beschränken.

die Datensätze zu bearbeiten oder Klassen zu entwickeln, die in einer größeren Anwendung zum Einsatz kommen könnten.

4.3.6.1 Filesplitter

CRM-Instanzen sind nicht hierarchisch geordnet. Sie sind über ihre Properties mit anderen Objekten verknüpfbar und beschreiben sich auf diese Weise gegenseitig. So entsteht ein sehr komplexes unbegrenztes Netz, in dem alle Verbindungen immer gleichrangig und in beide Richtungen lesbar sind. In der XML-Darstellung ist eine hierarchische Struktur jedoch unumgänglich. Daher wird ein (physisches) Objekt immer von der zweiten XML-Ebene beginnend beschrieben.

Eine XML-Datei beliebig viele CRM-Objekte in Form von Schwisterelementen auf der zweiten Ebene enthalten. Das *Filesplitter*-Programm zerlegt die großen Dateien in kleinere Dateien, sodass jedes Mal ein CRM-Objekt der obersten Ebene (das ist die zweite XML Ebene) in eine Datei geschrieben wird. Nützlich ist dies für die Speicherung von XML-Dateien in einer XML-Datenbank,⁵⁸ in der viele kleinere Dateien wesentlich schneller zu verarbeiten sind als wenige große (Meier o.J.). Die CRM-Dateien im XML-Format lassen sich in einer Datenbank gezielt durchsuchen. Auf diese Weise sollte mit einer einfachen Suchmaske als grafische Benutzerschnittstelle ein Informationssystem in seiner simpelsten Form erstellt werden.

Die Abfragesprache der eXist XML-Datenbank basiert auf XPath,⁵⁹ einer Sprache, die die Navigation in XML-Dokumenten ermöglicht. Die Datenbank erweitert die Abfragemöglichkeiten mit XPath zwar um wesentliche Eigenschaften, trotzdem gestaltet sich die Generierung von komplexen Suchanfragen, so genannten *nested queries*, bei denen Ergebnisse einer Suche mit einer neuen Anfrage durchsucht werden, als relativ aufwendig. Hinzu kommt, dass man in den wenigsten Fällen nach einem einzelnen Objekt sucht, sondern der Verkettung von mehreren Objekte nachgehen möchte und somit viele Suchanfragen in Folge generieren wird.

Ein anderer Grund wiegt schwerer, von der Nutzung einer XML-Datenbank abzusehen, nämlich die eingangs erwähnte Strukturierung der Daten. Tauchen viele Inschriften in einem Sammelband auf, wie z.B. in den Testdaten, enthält jedes *E34 Inscription-*

⁵⁸ Für diesen Versuch wurde die *eXist Open Source XML Database* benutzt, <<http://exist.sourceforge.net/>> (14.02.04).

⁵⁹ *XML Path Language (XPath)* <<http://www.w3.org/TR/xpath>> (14.02.04).

Objekt einen Verweis in Form eines Objekts auf diesen Sammelband. In der XML-Datenbank erscheinen damit viele identische Einträge, das heißt Elemente, für diesen Band. Eine Speicherung in einer relationalen oder einer objektorientierten Datenbank kann dieses Problem lösen, indem Referenzen erstellt werden. XML ist sinnvoll für den Transport und zur Bereitstellung von CRM-kodierten Daten, jedoch nicht für die Speicherung.

4.3.6.2 Objektprinter

Die Funktion des Programms *Objektprinter* besteht in der Demonstration der Instanziierung von Objekten. Es bietet ein Dialogfenster, in welchem sich die XML-Quelldatei und das Zielverzeichnis für die Ausgabe bestimmen lassen. Ferner ist es möglich, eine detaillierte Ausgabe *aller* Objekte oder nur derjenigen der ersten Ebene der XML-Datei zu erzeugen. Die Ausgabe besteht aus einer einfachen Textdatei, die eine Repräsentation der aus der Quelldatei eingelesenen Objekte enthält. Produziert wird diese Ausgabe durch den Aufruf der `logAll()`-Methode, der entsprechenden Objekte. Ziel der Demonstration ist es, die Funktion der Klasse `CRMContentHandler` zu zeigen. Diese Klasse, die sich in einem Xerxes SAX Parser⁶⁰ einsetzen lässt, ist der Kern der Generierung von Java-Objekten im CRM Format.

Beim Einlesen des XML-Dateien werden vom Xerxes Parser so genannte *Callback*-Methoden aufgerufen, wenn bestimmte Ereignisse auftreten. Solche Ereignisse sind beispielsweise der Beginn eines XML-Elements, der am öffnenden Tag erkannt wird, oder das Schließen eines Elements. Der `CRMContentHandler`⁶¹ verwendet zwei so genannte *Stacks*. Auf einem der Stacks werden bei einem öffnenden Tag CRM-Objekte abgelegt, auf dem anderen hingegen Methoden (bzw. Properties). Bei einem schließenden Tag wird, falls es sich um ein Objekt handelt, die oberste Methode des Methoden-Stacks aufgerufen und das Objekt selbst als Argument übergeben. Anschließend wird in jedem Fall das letzte Objekt bzw. die letzte Methode von dem entsprechenden Stack gelöscht. Die erzeugten Objekte werden während des Programmablaufs gespeichert und am Ende des Parsens ausgegeben. Der Quelltext der Klasse `CRMContentHandler` ist in Anhang B

⁶⁰ SAX steht für *Simple API for XML* <<http://xml.apache.org/xerces2-j/index.html>> (17.11.03). Ich habe die Version: *Xerces2 Java Parser 2.5.0* benutzt.

⁶¹ Vgl. Anhang B.

abgedruckt. Des Weiteren sei für eine detaillierte Beschreibung zur Verarbeitung von XML mit Java auf McLaughlin (2002) verwiesen.

Ein noch ungelöstes Problem, was sich bei diesem Verfahren stellt, ist, dass beim Einlesen der XML-Daten weit mehr Objekte erzeugt werden, als tatsächlich existieren. Das liegt an der im vorigen Abschnitt angesprochenen Struktur der XML-Daten. Daher muss ein Mechanismus entwickelt werden, der den gesamten Datenbestand verwaltet und gleiche Objekte zusammenführt.

5 Abschließende Bemerkungen

5.1 Erreichter Stand

Zusammenfassend lässt sich sagen, dass grundlegende Schritte zur Verwendung des CRM vollzogen sind, obwohl der erreichte Stand noch keine praktikable Anwendung darstellt. Die vorgestellte DTD ermöglicht es, Objekte, die durch das CRM beschrieben werden, korrekt in einfachem XML abzubilden. Mit dem Entwurf der Klassenbibliothek konnte gezeigt werden, dass die Übertragung des CRM in eine objektorientierte Programmiersprache möglich ist. Des Weiteren konnte mit Hilfe des Objektprinter-Programms gezeigt werden, dass sich unter Verwendung dieser Klassenbibliothek Java-Objekte im CRM-Format aus CRM kodierten XML-Daten generieren lassen. Dieser Stand lässt es nicht zu, weitreichende Aussagen über die Verwendung des Modells als Ontologie zu machen; er ist die Grundlage zur Verarbeitung in Java.

5.2 Offene Fragen und folgende Schritte

Da der erreichte Stand nicht in einer abgeschlossenen Anwendung besteht, sondern die Vorbereitung dazu leistet, gibt es eine Reihe von Schritten, die sich in direktem Anschluss an diese Arbeit anbieten. Alle folgenden Schritte sollten aber zunächst darauf zielen, die wichtigste Frage zu klären: inwieweit lässt sich mit den in Java kodierten CRM-Objekten ‚schließen‘ und damit ein Informationszuwachs produzieren?

Dazu soll nochmals das Beispiel der Jalta-Konferenz aus Abschnitt 2.2 bemüht werden, an dem deutlich wird, dass dieses *Schließen* oder *Argumentieren* tatsächlich nur sehr bedingt möglich ist. Gerade in dem von Doerr dargestellten Beispiel tritt ein massives Problem zu Tage. Der TGN-Eintrag liefert nur eine Bemerkung, eine Notiz, die nicht ausreichend formalisiert ist. Im CRM könnte man für einen solchen Vermerk das Property *PI Has Note* verwenden. *PI Has Note* kann aber eine beliebige Zeichenkette enthalten. Um diese Art Information auszuwerten, müssen Daten *interpretiert* werden. Man muss daher sehr genau unterscheiden, ob man von Schließen oder Interpretation spricht. Selbst der Ausdruck des *Schließen* ist vielleicht nicht recht zutreffend, besser wäre es vermutlich, von Auswerten zu sprechen, doch zunächst zur Interpretation. Eine maschinelle Interpretation ist nicht möglich, sondern zum derzeitigen Stand der Künstli-

chen Intelligenz ein Paradox. Wird eine inhaltlich gleiche Information bloß in verschiedenen Schreibweisen kodiert oder gar in verschiedenen Sprachen, funktioniert die maschinelle Verarbeitung nicht mehr. Das CRM sieht zwar ein Property vor, mit dem sich Informationen zur Sprache kodieren lassen, diese müssten aber zur maschinellen Verarbeitung auch erstellt worden sein und zudem eine Übersetzung liefern, die in der Argumentation verarbeitet werden kann. Nur wenn alle benötigten Daten in formalisierter Form zu Verfügung stehen, lassen sie sich maschinell verarbeiten. Allein die beiden Schreibweisen *Jalta* und *Yalta* können ein erhebliches Problem verursachen.

Eine *Auswertung* von CRM-Daten ist prinzipiell möglich. CRM-Objekte lassen sich in einem Graphen darstellen: Als Knoten, die über ungerichtete Kanten, das wären die Properties, mit anderen Knoten verbundenen sind. In einem derartigen Graphen lässt sich jede bestehende Verbindung finden. Ein solches Vorgehen ist bekannt und die Effektivität ist hierbei eine Frage der Suchstrategien bzw. der verarbeitenden Algorithmen. Sie bestimmen, wie schnell und mit welcher Qualität ein solches System arbeitet, denn das Problem, was sich stellt, ist die Menge der zu verarbeitenden Varianten. Sucht man beispielsweise nach der *Jalta-Konferenz*, das wäre ein Objekt vom Typ *E5 Event*, so gibt es über fünfzig Möglichkeiten, das heißt Properties, dieses Objekt mit anderen Objekten direkt zu verknüpfen.

Existiert ein *E5 Event*-Objekt, was sich über den Namen *Jalta-Konferenz* identifizieren lässt, ist es trivial, dieses Objekt zu finden und auch alle damit direkt verbundenen Objekte auszugeben, um weitere Informationen zu erhalten. Genauso einfach wäre es, alle *E41 Appellation*-Objekte zu finden, deren Inhalt *Jalta-Konferenz* ist und die damit das gesuchte *E5 Event* identifizieren. Nicht wesentlich komplizierter aber umfangreicher wird die Suche nach allen *E41 Appellation*-Objekten, die ein Property *P139 has alternative form* mit dem Inhalt *Yalta-Conference* oder *Jalta-Konferenz* oder diesen in Namen einer anderen Sprache oder in einer anderen Schreibweise besitzen.

Spätestens hier wird die Suche unübersichtlich und es schleicht sich wieder ein Informations-Overkill ein, ähnlich dem, den eine herkömmliche Suche im WWW liefert. Denn es werden bei einer solchen Suche nicht nur die gewünschten *E5 Events* ausgegeben, sondern *alle* Objekte zu denen Verknüpfungen bestehen. Hier wäre also eine Eingabemaske vonnöten, mit deren Hilfe sich eine detaillierte Suchanfrage erstellen lässt, die wenige hochwertige Ausgaben liefert. Das heißt, man sollte dem Anwender die

Möglichkeiten geben, Suchbegriffe semantisch miteinander zu verbinden. Man denke an einfache UND-Verknüpfungen, wie man sie in gängigen Suchmaschinen verwendet. Mit den Properties des CRM ließen sich Verknüpfungen erstellen, die eine differenzierte Suche ermöglichen.

Einer der ersten Schritte in diese Richtung wäre es, die Java-Objekte in einer objektorientierten Datenbank zu speichern, um die objektorientierten Strukturen des Modells voll auszuschöpfen. Auf diese Weise würde sich zeigen, wie komplex sich Suchanfragen sinnvoll gestalten lassen. Nach der Umsetzung einer einfachen Suche lassen sich Werkzeuge entwickeln, die eine Navigation erlauben, schließlich bedeutet Navigieren in diesem Zusammenhang nichts anderes als: Suchen von einem bestimmten Punkt aus. Als Voraussetzung dafür, dass komplexere Suchanfragen möglich werden, müssen erst alle Klassen der Bibliothek fertiggestellt werden, das heißt alle Methoden in beiden Richtungen müssen implementiert sein. Denn um in komplexen Strukturen zu suchen, werden die Objekte, die diese Strukturen bilden, benötigt.

Doch all dies wäre die Nutzung des CRM als Schema für eine Datenbank. Zwar ist diese Verwendung durchaus nützlich, aber es ist keine maschinelle Nutzung der Ontologie im Sinne einer Argumentation oder eines Schließens. Die Nutzung einer Ontologie soll aber gerade das maschinelle Schließen (*Reasoning*) ermöglichen. Wie weiter oben beschrieben, ist es grundsätzlich ausführbar, den Pfad einer Verbindung von zwei Knoten zu finden, auch über mehrere Kanten und Knoten hinweg. Wichtig ist, dass dies immer ein semantisch gehaltvoller Weg ist, und ein Mensch in der Lage ist, diesen Weg anhand der Semantik der Verknüpfungen gezielt zu verfolgen. Das einzig mögliche Vorgehen für eine Maschine ist jedoch, alle Möglichkeiten auszuprobieren, bis sie an ein bestimmtes Ziel gelangt, das die Suche beendet. Da die Semantik für die Maschine unzugänglich bleibt, lässt sich das Problem, *welcher* Weg sinnvoll ist, im voraus nicht lösen. Aus dieser Vielzahl von sich ergebenden Kombinationen entsteht ein neues Problem, nämlich, wie weit oder wie tief die Suche gehen soll, und wann die Suche abgebrochen wird, auch wenn sie zu keinem Ergebnis führt. Denn da die Verknüpfung von Objekten über Properties unbegrenzt möglich ist, ist auch der Suchraum unbegrenzt.

Dies ist ein ernstzunehmendes Problem, denn der in Abschnitt 2.4.1 angesprochene *Wine Agent* verarbeitet eine recht kleine Datenmenge und beansprucht dafür schon einige Sekunden. Da die Menge der potenziellen Verknüpfungen im Verhältnis zu den

Objekten exponential zunimmt, muss der Suchraum soweit wie möglich beschränkt werden, und selbst dann ist noch nicht klar, ob die Verarbeitung von größeren Datenmengen realistisch ist. Die Entwicklung eines Reasoners ist keine triviale Angelegenheit, daher wäre es möglicherweise von Nutzen die RDF-Version des CRM soweit zu entwickeln, das eine Schnittstelle zum JTP-API geschaffen wird. Da das JTP in Modulen aufgebaut ist, ließe sich das CRM wenigstens zu Testzwecken darin integrieren.

Andere sinnvolle Erweiterungen wären Schnittstellen für die Verarbeitung von CRM-Daten anderer Formaten. So, wie das Einlesen aus einer XML Datei geschieht, sollte es auch mit RDF oder den Daten einer relationalen Datenbank ermöglicht werden.

Schließlich sollte es Werkzeuge geben, mit denen sich CRM-Daten intuitiv und syntaktisch fehlerfrei erstellen lassen. Eine besondere Aufgabe ist dabei, eine Hilfestellung für die Kategorisierung von realen Objekten zu schaffen. Für eine solche Anwendung bieten sich Scalable Vector Graphics (SVG)⁶² an, eine XML-basierte Technologie zur Erstellung von Grafiken. Die Generierung solcher Grafiken geschieht über XML-Transformationen,⁶³ denn SVG ist ein XML Format. Die große Stärke dieser Technologie liegt in der Darstellung von schematischen, flächigen Grafiken, die zudem skalierbar sind, das heißt, sie lassen sich ohne Qualitätsverluste nahezu beliebig ein- und auszoomen, was für die Darstellung von komplexen Verkettungen von Objekten ein ausgesprochenes Vorteil ist. Die Visualisierung von CRM-Objekten wäre natürlich nicht nur für die Erstellung, sondern auch für das Durchsuchen und Navigieren in CRM-Daten geeignet.

5.3 Ausblick

Bis zu welchem Grad die Ausschöpfung von Ontologien gelingt, das heißt welche Möglichkeiten sich aus der maschinellen Verarbeitung von Informationen ergeben, wird sich erst in der Praxis zeigen. RSS-Reader sind die Vorläufer dieser Art von Programmen, sie nutzen die Möglichkeiten von RDF zwar nur im Ansatz, aber das mit Erfolg. Die Erwartungen an die Nutzung von Ontologien gehen weit darüber hinaus. Wenn sich auch nur ein Teil der Vision von Berners-Lee/Hendler/Lassila (5/2001) erfüllt, werden die daraus entstehenden Möglichkeiten ausgesprochen umfassend sein.

⁶² *Scalable Vector Graphics (SVG)* <<http://www.w3.org/Graphics/SVG/>> (6.01.04).

⁶³ Diese Transformationen lassen sich mit deren *Extensible Stylesheet Language Transformations (XSLT)* automatisieren. <<http://www.w3.org/TR/xslt>> (6.01.04).

RDF und RDFS scheinen im Bereich der Semantisierung augenblicklich die wichtigsten Schlüsseltechnologien zu sein. Sie liefern die Möglichkeiten, semantische Konzepte zu formalisieren und somit Inhalte transportierbar zu machen. XML und XML-Schema stellen die Syntax dazu bereit und sind damit wichtige Voraussetzungen. Da nahezu alle Ontologie-Sprachen RDF verwenden und die Vereinigung einiger Sprachen und Modelle schon zu beobachten war, wird sich vermutlich eine dieser Sprachen oder einige wenige auf der Basis von RDF durchsetzen. Nichtsdestotrotz bleibt es auch unter Verwendung der besten Sprache eine aufwendige Arbeit, durchdachte Ontologien zu erstellen. Vermutlich werden Ontologien hinsichtlich der Qualität ihrem Geltungsbereich entsprechend erstellt. Application-Ontologies könnten von Anwendungen automatisch erstellt und manuell nachbearbeitet werden, während Domain- und Top-Level Ontologien erst nach sorgfältiger Entwicklung eingesetzt werden. Insbesondere Top-Level Domains werden vermutlich vielfach und über einen längeren Zeitraum genutzt, da die darin beschriebenen Konzepte fundamentaler Art sind.

Das CRM ist eine durchdachte Domain-Ontologie, deren Entwicklung im wesentlichen abgeschlossen ist. Es wäre ein interessanter Vergleich, das Modell in Form von Topic Maps und in OWL zu realisieren, und sicherlich wäre der Bereich des Kulterbes durch diese Umsetzung mit an der Spitze der Entwicklung des Semantic Web vertreten. Selbst wenn die Erwartungen einer ‚intelligenten‘ Nutzung von Inhalten durch Softwareagenten nicht erfüllt werden, bleiben für das CRM die Anwendungsmöglichkeiten als Metadatenschema, was die Mediation heterogener Quellen ermöglicht und als Ontologie, mit der eine gezielte Suche nach Inhalten möglich ist.

6 Anhang A - Die DTD

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
  CRMSet ist ein reines XML-Container Element und gehoert nicht zum eigentlichen CRM.
-->
<!ELEMENT CRMSet (%R1_CRM_Entity;)+>

<!-- CRM-Entities:
- Es wurden nur die für die Testdaten benoetigen Klassen beruecksichtigt
  und darin auch nur die benoetigten Properties. Die vorliegende DTD ist
  also keine vollstaendige Beschreibung des CRM. Sie kann lediglich die
  Beispieldaten validieren; dadurch laesst sich allerdings gewaehrleisten,
  dass diese der CRM Definition genuegen.
- Der Quantifier fuer Property-Elemente" ist in der Entity-Definition IMMER
  "?".
- Die Property-Elemente sind nach "P-Nummern" sortiert, diese Reihenfolge
  MUSS in Dokumenten beibehalten werden!(aus diesem Grund lassen sich hier
  leider keine Entities verwenden).
-->
<!ELEMENT E1_CRM_Entity (P1F_is_identified_by?, P2F_has_type?, P3_has_note?)>
<ATTLIST E1_CRM_Entity
  content CDATA #REQUIRED
>
<!ELEMENT E31_Document (P1F_is_identified_by?, P2F_has_type?, P3_has_note?,
P106B_forms_part_of?)>
<ATTLIST E31_Document
  content CDATA #REQUIRED
>
<!ELEMENT E34_Inscription (P1F_is_identified_by?, P2F_has_type?, P3_has_note?,
P70B_is_documented_in?, P150F_shows_characters?, P151F_has_transcription?)>
<ATTLIST E34_Inscription
  content CDATA #REQUIRED
>
<!ELEMENT E41_Appellation (P1F_is_identified_by?, P2F_has_type?, P3_has_note?,
P70B_is_documented_in?, P106B_forms_part_of?)>
<ATTLIST E41_Appellation
  content CDATA #REQUIRED
>
<!ELEMENT E62_String (#PCDATA)>

<!-- XML-Entities fuer Ranges:
- Das R steht fuer Range, ein Range ist immer eine Klasse einschliesslich
  ihrer Tochterklassen, "Vererbung" wird hier jeweils durch Aufzaehlung
  beschrieben.
- In der Elementeliste steht an erster Stelle immer die Klasse, aus der
  alle folgenden abgeleitet sind.
- Die folgenden Klassen sind allein der Uebersichtlichkeit halber nach "E-
  Nummern" sortiert.
-->

<!ENTITY % R1_CRM_Entity "E1_CRM_Entity | E2_Temporal_Entity | E3_Condition_State |
E4_Period | E5_Event | E6_Destruction | E7_Activity | E8_Acquisition_Event | E9_Move |
E10_Transfer_of_Custody | E11_Modification_Event | E12_Production_Event |
E13_Attribute_Assignment | E14_Condition_Assessment | E15_Identifier_Assignment |
E16_Measurement_Event | E17_Type_Assignment | E18_Physical_Stuff | E19_Physical_Object |
E20_Biological_Object | E21_Person | E22_Man-Made_Object | E24_Physical_Man-Made_Stuff |
E25_Man-Made_Feature | E26_Physical_Feature | E27_Site | E28_Conceptual_Object |
E29_Design_or_Procedure | E30_Right | E31_Document | E32_Authority_Document |
E33_Linguistic_Object | E34_Inscription | E35_Title | E36_Visual_Item | E37_Mark |
E38_Image | E39_Actor | E40_Legal_Body | E41_Appellation | E42_Object_Identifier |
E44_Place_Appellation | E45_Address | E46_Section_Definition | E47_Spatial_Coordinates |
E48_Place_Name | E49_Time_Appellation | E50_Date | E51_Contact_Point | E52_Time-Span |
E53_Place | E54_Dimension | E55_Type | E56_Language | E57_Material |
E58_Measurement_Unit | E63_Beginning_of_Existence | E64_End_of_Existence |
E65_Creation_Event | E66_Formation_Event | E67_Birth | E68_Dissolution | E69_Death |
E70_Stuff | E71_Man-Made_Stuff | E72_Legal_Object | E73_Information_Object | E74_Group |
E75_Conceptual_Object_Appellation | E77_Persistent_Item | E78_Collection |
E79_Part_Addition | E80_Part_Removal | E81_Transformation | E82_Actor_Appellation |
E83_Type_Creation | E84_Information_Carrier">

<!ENTITY % R31_Document "E31_Document | E32_Authority_Document">
```

```

<!ENTITY % R41_Appellation "E41_Appellation | E42_Object_Identifier |
E44_Place_Appellation | E49_Time_Appellation | E75_Conceptual_Object_Appellation |
E82_Actor_Appellation">

<!ENTITY % R55_Type "E55_Type | E56_Language | E57_Material | E58_Measurement_Unit">

<!ENTITY % R73_Information_Objekt "E73_Information_Objekt | E29Design_or_Procedure |
E31_Document | E32_Authority_Document | E33_Linguistic_Object | E34_Inscription |
E35_Title | E36_Visual_Item | E37_Mark | E38_Image">

<!-- Properties:
- Quantifier sollen ausschliesslich hier gesetzt werden, damit sind sie an
einer Stelle fuer alle Klassen bearbeitbar.
-->
<!ELEMENT P1F_is_identified_by (%R41_Appellation;)+>
<!ELEMENT P1B_identifies (%R1_CRM_Entity;)+>
<!ELEMENT P2F_has_type (%R55_Type;)+>
<!ELEMENT P2B_is_type_of (%R1_CRM_Entity;)+>
<!ELEMENT P3F_has_note (E62_String)+>
<!ELEMENT P70F_documents (%R1_CRM_Entity;)+>
<!ELEMENT P70B_is_documented_in (%R31_Document;)+>
<!ELEMENT P106F_is_composed_of (%R73_Information_Objekt;)+>
<!ELEMENT P106B_forms_part_of (%R73_Information_Objekt;)+>
<!ELEMENT P127F_has_broader_term (%R55_Type;)+>
<!ELEMENT P127B_has_narrower_term (%R55_Type;)+>
<!ELEMENT P137F_exemplifies (%R55_Type;)+>
<!ELEMENT P137B_is_exemplified_by (%R1_CRM_Entity;)+>
<!ELEMENT P150F_shows_characters (E62_String)><!-- war urspruenglich: (#PCDATA)> -->
<!ELEMENT P151F_has_transcription (E62_String)><!-- war urspruenglich: (#PCDATA)> -->

```

7 Anhang B - Quellcode Beispiele

Im Folgenden ist das Interface `CRMEntity` abgedruckt sowie die Klassen `E1CRMEntity` und den `CRMContentHandler`. Diese Dateien mit ausführlicheren Kommentaren sowie weitere Informationen enthält die in Anhang B beschriebene CD-ROM.

```
package crm;

import crm.persistent.appell.E41Appellation;
import crm.persistent.made.E55Type;

public interface CRMEntity {

    /**
     * f1 is identified by (identifies): E41 Appellation
     */
    public void f1IsIdentifiedBy(E41Appellation appell);

    /**
     * f2 has type (is type of): E55 Type
     */
    public void f2HasType(E55Type type);

    /**
     * f3 has note: E62 String
     * The class "crm.E62String" is not related to the class java.lang.String!
     */
    public void f3HasNote(E62String string);

    /**
     * b137 is exemplified by (exemplifies): E1 CRM Entity
     */
    public void b137Exemplifies(E55Type type);
}
// eof
```

```

package crm;

import crm.persistent.appell.E41Appellation;
import crm.persistent.made.E55Type;
import crm.persistent.legal.E31Document;

import java.util.Vector;
import java.io.Serializable;

public class E1CRMEntity implements CRMEntity, Serializable {

    public E1CRMEntity() {
    }

    public Vector f1 = new Vector();
    public Vector f2 = new Vector();
    public Vector f3 = new Vector();
    public Vector b70 = new Vector();
    public Vector b137 = new Vector();
    public Vector b138 = new Vector();
    public String content = "none";

    /**
     * f1 is identified by (identifies): E41 Appellation
     */
    public void f1IsIdentifiedBy(E41Appellation appell) {
        f1.add(appell);
        appell.b1.add(this);
    }

    /**
     * f2 has type E55 Type
     */
    public void f2HasType(E55Type type) {
        f2.add(type);
        type.b2.add(this);
    }

    /**
     * f3 has note: E62 String
     */
    public void f3HasNote(E62String string) {
        f3.add(string);
        string.b3.add(this);
    }

    /**
     * b70 is documented in: E31 Document
     */
    public void b70IsDocumentedIn(E31Document document) {
        b70.add(document);
        document.f70.add(this);
    }

    /**
     * P137 is exemplified by (exemplifies): E1 CRM Entity
     */
    public void b137Exemplifies(E55Type type) {
        b137.add(type);
        type.f137.add(this);
    }

    /**
     * This method returns in addition to the toString() method of java.lang.Object the
     * the "content" field of this object.
     * @return A string representation of this object and its "content"
     */
    public String toString() {
        return getClass().getName() + '@' + Integer.toHexString(hashCode()) +
            " content: " + content;
    }

    /**
     * This method returns a string representation of this object including its field
     * names and simple values in form of vector size or string representation. If for
     * example its "types" field contains four objects the method will return a string
     * like: "types: 4"
     */
}

```

```

* @return A string representation of this object and its fields including simple
* values.
*/
public String log() {
    StringBuffer buf = new StringBuffer();
    buf.append(this.toString() + "\n");
    buf.append("is identified by: " + f1.size() + "\n");
    buf.append("has types: " + f2.size() + "\n");
    buf.append("has notes: " + f3.size() + "\n");
    buf.append("is documented in: " + b70.size() + "\n");
    buf.append("exemplifies: " + b137.size() + "\n");
    return buf.toString();
}

/**
* This method works like log() except that it returns instead of simple values
* explicit values.If for example the object's "types" field contains three objects
* it will return the string representation of these three objects.
*
* @return An explicit String representation of this object and its fields.
*/
public String logAll() {
    StringBuffer buf = new StringBuffer();
    buf.append(this.toString() + "\n");
    buf.append("is identified by: " + f1.toString() + "\n");
    buf.append("has types: " + f2.toString() + "\n");
    buf.append("has notes: " + f3.toString() + "\n");
    buf.append("is documented in: " + b70.toString() + "\n");
    buf.append("exemplifies: " + b137.toString() + "\n");
    return buf.toString();
}
}
// eof

```

```

package app;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.LinkedList;
import java.util.Properties;
import org.xml.sax.Attributes;
import org.xml.sax.ContentHandler;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;
// Dieser Import MUSS das gesamte Package umfassen wg. class.forName()!
import crm.*;

// Es gibt noch keine Fehlerbehandlung, wie z.B. das Auftauchen von Elemente
// in der falschen Reihenfolge. Das sollte grundsätzlich durch die Validierung
// gegen die DTD abgefangen werden.
public class CRMContentHandler implements ContentHandler {

    private Locator locator;
    private Properties classProps = new Properties();
    private Properties propProps = new Properties();
    private E62String e62String; // Dies ist KEIN Java String!
    private String CLASS_E62_STRING = "crm.E62String";
    private String LOCALNAME_E62_STRING = "E62_String";
    private String CONTENT = "content";
    private String CRMSET = "CRMSet";
    private StringBuffer charBuf = new StringBuffer();
    private LinkedList objStack = new LinkedList();
    private LinkedList metStack = new LinkedList();
    private LinkedList printList = new LinkedList();
    private boolean logVerbose = false;

    public void setDocumentLocator(Locator aLocator) {
        this.locator = aLocator;
    }

    // Der Konstruktor
    public CRMContentHandler(File aDir, boolean logVerbose) {
        this.logVerbose = logVerbose;
        // lade Properties
        try {
            classProps.load(new FileInputStream(new File("res\\class-names")));
            propProps.load(new FileInputStream(new File("res\\property-names")));
        }
        catch (IOException ex) {
            ex.printStackTrace(System.out);
        }
    }

    public void startDocument() {
        // nichts zu tun
        System.out.println("start parsing document");
    }

// Fortsetzung nächste Seite

// Schreibt eine String Representation aller Java-Objekte/CRM-Objekte der
// obersten Ebene in ein File.
public void endDocument() {
    System.out.println("end parsing document");
    E1CRMENTity crm;
    E59PrimitiveValue prim;

```

```

Object obj;
try {
    FileWriter fw = new FileWriter(new File("CRM-Objects.txt"));
    StringBuffer buf = new StringBuffer();
    int count = 0;
    while (printList.size() > 0) {
        count++;
        obj = printList.getFirst();
        printList.removeFirst();
        buf.append("*** " + count + " ***\n");
        if (obj instanceof E1CRMEntity) {
            crm = (E1CRMEntity)obj;
            buf.append(crm.logAll() + "\n");
        }
        else if (obj instanceof E59PrimitiveValue) {
            prim = (E59PrimitiveValue)obj;
            buf.append(prim.logAll() + "\n\n");
        }
    }
    fw.write(buf.toString());
    fw.close();
}
catch (IOException ex) {
}
}

public void startElement(String namespaceURI, String localName, String qName,
Attributes atts) {
    Object obj;
    // entity/class handling
    String currentElement = classProps.getProperty(localName);
    if (currentElement != null) {
        try {
            obj = Class.forName(currentElement).newInstance();
            if (currentElement.equals(CLASS_E62_STRING)) {
                e62String = (E62String)obj;
                objStack.add(e62String);
            }
            else {
                E1CRMEntity crm = (E1CRMEntity)obj;
                crm.content = atts.getValue(CONTENT);
                objStack.add(crm);
            }
        }
        catch (IllegalAccessException ex) {
            System.out.println(ex);
        }
        catch (InstantiationException ex) {
            System.out.println(ex);
        }
        catch (ClassNotFoundException ex) {
            System.out.println(ex);
        }
    }
    // property/method handling
    else if ((currentElement = propProps.getProperty(localName)) != null) {
        obj = objStack.getLast();
        Method[] methods = obj.getClass().getMethods();
        for (int i = 0; i < methods.length; i++) {
            if (methods[i].getName().equals(currentElement)) { // matching method found!
                metStack.add(new MethodObj(methods[i], obj));
                break;
            }
        }
    }
    else if (localName != CRMSET) {
        System.out.println("Parsing error @ startElement: unexpected element!");
    }
}
// Fortsetzung nächste Seite
public void endElement(String namespaceURI, String localName, String qName) {
    if (propProps.getProperty(localName) != null) {
        if (metStack.size() != 0) {
            metStack.removeLast();
        }
        else {
            System.out.println("Parsing error @ endElement: metStack was null! @ line: "
                + locator.getLineNumber());
        }
    }
    else if (classProps.getProperty(localName) != null) {
        if (objStack.size() != 0) {

```

```

Object[] args = {objStack.getLast()};
objStack.removeLast();
MethodObj mo;
if (localName.equals(LOCALNAME_E62_STRING)) {
    if (e62String != null) {
        e62String.contentOfElement = charBuf.toString();
        charBuf.delete(0, charBuf.length());
        e62String = null;
    }
}
try {
    if (metStack.size() > 0) {
        mo = (MethodObj)metStack.getLast();
        mo.method.invoke(mo.obj, args);
    }
}
catch (InvocationTargetException ex) {
}
catch (IllegalArgumentException ex) {
}
catch (IllegalAccessException ex) {
}
if (objStack.size() == 0 && logVerbose == false) {
    printList.add(args[0]);
}
else if (logVerbose == true) {
    printList.add(args[0]);
}
}
else {
    System.out.println("Parsing error @ endElement objStack was null! @ line: "
        + locator.getLineNumber());
}
}
else if (localName != CRMSET) {
    System.out.println("Parsing error @ endElement: unexpected element!");
}
}

// Die Bearbeitung von Zeichendaten (Elementinhalt) findet nur in E62_String-Elementen
// statt, bzw. in E59_PrimitiveValue und deren Tochterklassen.
public void characters(char[] ch, int start, int length) throws SAXException {
    if (e62String != null) {
        String str = new String(ch, start, length);
        if (str != null) {
            str = str.trim();
        }
        if (str != null) {
            charBuf.append(str);
        }
    }
}
}

```

// Fortsetzung nächste Seite

```

public void ignorableWhitespace(char[] ch, int start, int length) throws SAXException
{
    // nichts zu tun
}

public void processingInstruction(String target, String data) throws SAXException {
    // nichts zu tun
}

public void skippedEntity(String name) throws SAXException {
    // nichts zu tun
}

public void startPrefixMapping(String prefix, String uri) throws SAXException {
    // nichts zu tun
}

public void endPrefixMapping(String prefix) throws SAXException {
}

```

```
// nichts zu tun
}

class MethodObj {

    Method method;
    Object obj;

    public MethodObj(Method m, Object o) {
        method = m;
        obj = o;
    }
}
// eof
```

8 Anhang C - Inhalt der CD-ROM

- Die Datei inhalt.txt. Sie beschreibt die folgenden Inhaltsangaben.
- Die original Testdaten und die modifizierten Beispieldaten
- Die DTD CRMSet
- Das CRM API bestehend aus Klassenbibliothek, Quellcode und JavaDocs
- Das Programm Objektprinter, samt Quellcode, Hilfsdateien und Beschreibung
- Das Programm Filesplitter, samt Quellcode und Beschreibung

9 Literaturverzeichnis

Diese Bibliographie enthält alle Angaben zu zitierten Quellen und zur verwendeten Literatur, bei denen der Autor bekannt und somit die Kurzzitierweise möglich ist. Bei Internetquellen wo dies nicht der Fall ist, verweise ich jeweils in einer Fußnote auf die entsprechende URL der Quelle bzw. der verwendeten Literatur.

Balzert, Heide. 2001. *Objektorientierung in 7 Tagen*. Spektrum, Heidelberg, u.a.: Akademie Verlag.

Bekiari, Chrissoula; Martin Doerr. 10/1999. *Documentation and Reasoning on Parts and Potential Wholes. Technical Report 260, ICS-FORTH*.
<ftp://ftp.ics.forth.gr/tech-reports/1999/1999.TR260.Documentation_Reasoning_Parts_Potential_Wholes.ps.gz> (09.02.04).

Biezunski, Michel. 2003. *Introduction To The Topic Maps Paradigm*. in Park, 2003.

Berners-Lee, Tim; James Hendler, Ora Lassila. 5/2001. *The Semantic Web*.
<<http://www.scientificamerican.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21&catID=2>> (Datum).

----- . 9/1998. *Semantic Web Road map*. <<http://www.w3.org/DesignIssues/Semantic.html>> (Datum).

Champin, Pierre-Antoine. 6/2001. *RDF Tutorial*. <<http://citeseer.nj.nec.com/513949.-html>> (9.02.04).

Crofts, Nick; Martin Doerr. ohne Jahr. *Electronic esperanto – The Role of the oo CIDOC Reference Model*.
<http://cidoc.ics.forth.gr/docs/doerr_crofts_ichim99_new.pdf> (09.02.04).

----- . 1998. *CIDOC Conceptual Reference Model, Introduction to object oriented modelling*. <<http://cidoc.ics.forth.gr/docs/introduction2oo.htm>> (Datum).

----- . 1999. *Implementing the CIDOC CRM with a relational database*.
<http://cidoc.ics.forth.gr/docs/Implementing_the_CIDOC_CRM.pdf> (09.02.04).

- ; u.a. (Hgg.). 11/2002. *Definition of the CIDOC object-oriented Conceptual Reference Model and Crossreference Manual. Version 3.4.* <http://cidoc.ics.forth.gr/crm_definition_documents/cidoc_crm_3.4/definition_cidoc_cross_referenced_3-4.html> (27.01.04).
- Denny, Michael. 11/2002. *Ontology Building: A Survey of Editing Tools* <<http://www.xml.com/pub/a/2002/11/06/ontologies.html>> (Datum).
- Doerr, Martin; Jane Hunter, Carl Lagoze. ohne Jahr. *Towards a Core Ontology for Information Integration.* <<http://jodi.ecs.soton.ac.uk/Articles/v04/i01/Doerr/doerr-final.pdf>> (09.02.04).
- . 1998. *Data Example of the CIDOC Reference Model, Epitaphios GE34604.* <http://cidoc.ics.forth.gr/docs/crm_example_1.pdf> (09.02.04).
- . 2000. *Mapping of the Dublin Core Metadata Element Set to the CIDOC CRM.* <<http://citeseer.nj.nec.com/doerr00mapping.html>> (09.02.04).
- ; Iraklis Karvasonas. 5/2001. *Converting object documentation into a CRM-compatible XML form using Data Junction 7.5.* <http://cidoc.ics.forth.gr/docs/data_index_mapping.pdf> (27.01.04).
- . 7/2001. *Mapping format for data structures of the CIDOC CRM.* <<http://cidoc.ics.forth.gr/docs/mappingdtd.pdf>> (09.02.04).
- . 10/2001. *The CIDOC CRM – an Ontological Approach to Semantic Interoperability of Metadata.* <http://cidoc.ics.forth.gr/docs/crm-ontology_preprint.pdf> (09.02.04).
- Freese, Eric. 2003. *Topic Maps and RDF.* in Park, 2003:283-325.
- Geser, Guntram; Seamus Ross. 2003. *DigiCULT Thematic Issue 3, Towards a Semantic Web for Heritage Resources.* Salzburg: Salzburg Research Forschungsgesellschaft oder als PDF-Datei unter <http://www.digicult.info/downloads/thematic_issue_3_low.pdf>
- Goll, Joachim; Cornelia Weiß, Peter Rothländer. 2002. *Java als erste Programmiersprache.* Stuttgart u.a.: B.G. Teubner.

Gruber, Thomas. 4/1993. *A Translation Approach to Portable Ontology Specifications*.
<ftp://ftp.ksl.stanford.edu/pub/KSL_Reports/KSL-92-71.ps.gz> (Datum).

Gruber, Thomas. 8/1993. *Toward Principles for the Design of Ontologies Used for Knowledge Sharing (1993)*. <<http://citeseer.ist.psu.edu/gruber93toward.html>>
(09.02.04).

Guarino, Nicola. 10/96. *Understanding, Building, And Using Ontologies*.
<<http://ksi.cpsc.ucalgary.ca/KAW/KAW96/guarino/guarino.html>> auch als PDF-
Datei unter <<http://citeseer.nj.nec.com/guarino97understanding.html>> (9.02.04).

-----, 1997. *Semantic Matching: Formal Ontological Distinctions for Information Organization, Extraction, and Integration*.
<<http://gunther.smeal.psu.edu/guarino97semantic.html>> (Datum).

-----, 6/1998. *Formal Ontology and Information Systems*. <<http://www.loa-cnr.it/Papers/FOIS98.pdf>> (Datum).

Heflin, Jeff. 2004. *OWL Web Ontology Language Use Cases and Requirements*.
<<http://www.w3.org/TR/2004/REC-webont-req-20040210/>> (Datum).

Holzner, Steven. 2001. *XML Insider*. München: Markt+Technik.

Krüger, Guido. 2003³. *Handbuch der Java-Programmierung*. München u.a.: Addison
Wesley.

Koivunen, Marja-Riitta; Eric Miller. 2001. *W3C Semantic Web Activity*.
<<http://www.w3.org/2001/12/semweb-fin/w3csw>> (Datum).

Kwiatkowski, Gerhard (Hg.). 1985. *Schülerduden. Die Philosophie*. Mannheim u.a.: Bi-
bliographisches Institut.

McGuinness, Deborah L.; Frank van Harmelen. 2004. *OWL Web Ontology Language Overview*. <<http://www.w3.org/TR/2004/REC-owl-features-20040210/>>
(11.02.04).

- McIlraith, Sheila A.; Tran Cao Son, Honglei Zeng. 2001. *Semantic Web Services*. <<http://citeseer.nj.nec.com/mcilraith01semantic.html>> (9.02.04).
- McLaughlin, Brett. *Java & XML*. 2002². Köln u.a.: O'Reilly Verlag.
- Meier, Wolfgang. ohne Jahr. *eXist: An Open Source Native XML Database*. <<http://citeseer.nj.nec.com/551828.html>> (9.02.04).
- Miller, Eric et al. 24.01.01. *W3C RSS 1.0 News Feed Creation How-To*. <<http://www.w3.org/2001/10/glance/doc/howto#htmlrss>> (Datum).
- Mylopoulos, John; u.a. 1990. *Telos: Representing Knowledge About Information Systems*. <<http://citeseer.nj.nec.com/mylopoulos90telos.html>> (Datum).
- Newcomb, Steven. 2003. *A Perspective on the Quest for Global Knowledge Interchange*. in Park, 2003:31-50.
- Noy, Natalya; Deborah McGuinness. ohne Jahr. *Ontology Development 101: A Guide to Creating Your First Ontology*. <<http://www.ksl.stanford.edu/people/dlm/papers/ontology101/ontology101-noy-mcguinness.html>> (11.02.04).
- Obrst, Leo; Howard Liu. *Knowledge Representation, Ontologie Engineering, And Topic Maps*. in Park, 2003:103-148.
- Park, Jack (Hg.). 2003. *XML Topic Maps. Creating and Using Topic Maps for the Web*. Boston u.a.: Addison Wesley.
- Rosin, Andre. ohne Jahr. *RDF-Schema* <http://www.dbis.informatik.hu-berlin.de/lehre/WS0203/SemWeb/artikel/3/Rosin_RDF-Schema_v4.pdf> (Datum).
- Salathé, Marcel. 2001. *SVG - Scalable Vector Graphics für professionelle Einsteiger*. München: Markt+Technik.

Schaller, Kurt. 6/2003. *UBI-ERAT-LUPA. Auf der Internet-Fährte der römischen Wölfin*. <<http://mailbox.univie.ac.at/elisabeth.trinkl/forum/forum0603/27lupa.htm>> (Datum).

Sperberg-McQueen, C.M.; Lou Burnard (Hgg.). 2002. *TEI P4: Guidelines for Electronic Text Encoding and Interchange*. Oxford, u.a.: Text Encoding Initiative Consortium.

van der Vlist, Eric. 2003¹. *XML-Schema*. Köln u.a.: O'Reilly Verlag.

Volz, Rainer. 2003. *RSS kurzgefasst*.
<http://www.vrtpj.de/content/istandards/rssguide_de.html> (Datum).

Widhalm, Richard; Thomas Mück. 2002. *Topic Maps*. Berlin u.a.: Springer-Verlag.

Zajac, Rémi. ohne Jahr. *Engineering Ontologies vs. Linguistic Ontologies for Web Applications*. <<http://www.carc.aist.go.jp/nlwww/mmac/resources/pdf/zajac.pdf>> (Datum).

10 Erklärung

Hiermit versichere ich, dass ich diese Magisterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen meiner Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen sind, habe ich in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht.

Dasselbe gilt sinngemäß für Tabellen, Karten und Abbildungen.

Unterschrift