Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation (IPD)

Ein Java-Rahmenwerk zur kooperativen, anreizbasierten Erstellung von Ontologien

Diplomarbeit

von

Thomas Much

15. November 2005 — 14. Juni 2006

Verantwortlicher Betreuer:	Prof. Dr.-Ing. K. Böhm
Betreuender Mitarbeiter:	Dipl.-Wirtsch.Inf. Conny Kühne

Ich erkläre hiermit, die vorliegende Arbeit selbstständig und ohne unzulässige fremde Hilfe angefertigt zu haben. Die verwendeten Literaturquellen sind im Literaturverzeichnis angegeben.

Karlsruhe, den 14. Juni 2006

Zusammenfassung

Ein Java-Rahmenwerk zur kooperativen, anreizbasierten Erstellung von Ontologien

Online-Enzyklopädien wie Wikipedia erhalten derzeit große Aufmerksamkeit — nicht nur in den Fachmedien, sondern auch in der gewöhnlichen Tagespresse. Das Interesse ist gerechtfertigt, kann doch mit solchen Systemen jedermann nicht nur Informationen abfragen, sondern auch neue Informationen nahezu beliebig hinzufügen.

Das Problem dieser Systeme ist, inwieweit die kooperativ gesammelten Informationen gesichertes Wissen darstellen. Daher wurden schon recht bald Bewertungssysteme eingeführt. Die generelle Problematik solcher Bewertungen, die Motivation zur Abgabe von Bewertungen und Angriffsmöglichkeiten (Szenarien zur gezielten Manipulation) werden bereits von anderen Arbeiten behandelt. Allerdings muss dabei die Infrastruktur der Testsysteme immer wieder komplett neu entworfen werden.

Ziel dieser Diplomarbeit ist es daher, ein Java-Rahmenwerk (Framework) zur Verfügung zu stellen, auf das Systeme zur kooperativen, anreizbasierten Erstellung von Ontologien aufsetzen können. Dabei soll das Rahmenwerk zum einen für die diversen Einsatzzwecke möglichst gut wieder verwendbar sein, zum anderen müssen sich einzelne Komponenten möglichst flexibel austauschen lassen.

Die Diplomarbeit umfasst zunächst die Analyse ausgewählter Systeme und Lösungsansätze, um einen gemeinsamen Nenner der Schnittstellen und Funktionalitäten zu finden. Die Implementierung des Java-Rahmenwerks definiert dann Schnittstellen (Interfaces) zum Erstellen der Ontologien, zur persistenten Speicherung, zur Ontologie-Evolution, zum Im-/Export der Ontologie, für eine Benutzerverwaltung sowie Schnittstellen für Anreiz-, Bewertungs-, Klassifikations- und Loggingmodule. Die implementierenden Komponenten sind austauschbar und werden beim Start der Anwendung dynamisch geladen. Die Diplomarbeit umfasst außerdem den lauffähigen Prototypen einer Web-Applikation, der das Rahmenwerk nutzt und der in dieser Ausarbeitung als konkretes Beispiel verwendet wird.

Abstract

A Java Framework for Cooperative, Incentive-based Creation of Ontologies

Online encyclopedias like Wikipedia raise a lot of interest these days, not only in specialised literature but also in daily news. For good reason, because with systems like these everyone can not only retrieve information but also add new and change existing data.

The question is to what extent the cooperatively collected information can be considered solid knowledge, so rating facilities were introduced soon. Ratings, their specific manipulation and malicious attacks are subject of other papers, but one problem remains: The infrastructure for test systems has to be developed anew each time.

This diploma thesis is about developing a Java framework as a basis for applications that allow cooperative, incentive-based creation of ontologies. The framework will be reusable for different purposes of ontology creation, and its components will be exchangeable easily. After an analysis of existing systems, the framework will define interfaces for ontology creation, persistent storage, import and export, ontology evolution, classification, logging, and of course for rating and incentive schemes. Implementing components are substitutable by declaration and will be loaded on startup dynamically. A prototype web application that uses the framework is provided as well.

While this thesis is written in German, Javadoc documentation is available in English [69].

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Als Tim Berners-Lee ab 1989 das World Wide Web (kurz WWW oder einfach „Web“) am CERN in Genf entwickelte, war im zugrunde liegenden HTTP-Protokoll nicht nur die HTTP-Methode *GET* zum Abfragen von Ressourcen eines Web-Servers vorgesehen, sondern bald auch die Methode *PUT*, mit der Daten direkt auf dem Server verändert werden können [13]. Die Idee dahinter war, dass ein jeder Informationen im Web genauso einfach lesen wie schreiben können sollte. Das Web sollte ein bidirektionales, kollaboratives Medium sein. Dieser Ansatz hatte sich aber zunächst nicht durchgesetzt, auch wenn mit Amaya [1] ein Web-Editor/Browser vom W3C angeboten wird, der das direkte Ändern von Server-Daten beherrscht. Das Web war lange Zeit ein eher statisches Medium, von wenigen geschrieben, von den meisten nur gelesen. Berners-Lee sieht die Mensch-zu-Mensch-Kommunikation durch einfach publizierbares Gemeinschaftswissen (shared knowledge) aber immer noch als wichtigen Punkt für die Weiterentwicklung des Web an [14, Kapitel 12].

Die freie Enzyklopädie „Wikipedia“ [101] ist ein Schritt in diese Richtung, auch wenn die Änderung der Server-Daten hier technisch anders realisiert ist[1]. Jeder Wikipedia-Nutzer kann die Artikel der Enzyklopädie nicht nur lesen, sondern auch ergänzen und korrigieren sowie neue Artikel anlegen. Durch die Kooperation der Nutzer entsteht so eine gemeinsam geschaffene Wissensbasis. Andere Web-Anwendungen erlauben es beispielsweise, in einer spezialisierten Wissensbasis für Urlaubsreisen Hotels, Urlaubsorte und -länder zu beschreiben, um nachfolgenden Reisenden Vorschläge zu machen, Warnungen auszusprechen und praktische Tipps zu geben.

Derartige Systeme weisen derzeit aber häufig zwei wesentliche Schwächen auf:

1. Wenn jeder Nutzer Änderungen einbringen darf, kann nicht ausgeschlossen werden, dass es zu böswilligen Attacken (Vandalismus), absichtlichen Falschdarstellungen oder schlicht falschen Daten aufgrund von Unwissenheit kommt. So wurden bei Wikipedia in jüngster Zeit diverse Biografien mit offenbar unwahren [43] oder zu einseitigen Behauptungen [44] manipuliert. Wie kann man solche Manipulationen unterbinden, ohne die rechtschaffenen Nutzer allzu sehr einzuschänken bzw. zu demotivieren?

2. Gerade die falschen Daten aufgrund von Unwissenheit zeigen, dass gut gemeinte Änderungen nicht automatisch qualitativ gute Änderungen sind. Wie kann man die objektive Qualität eines Beitrags ermitteln, d.h. wie kann man feststellen, welche der gesammelten Informationen gesichertes Wissen darstellen?

Eine zentrale Verwaltung und Kontrolle der Informationen durch Administratoren oder Domänenexperten [91] möchte man hierbei vermeiden. Schutz und qualitative Bewertung sollen nach Möglichkeit automatisch durchgeführt werden, die Entscheidungsgrundlage dafür muss aber wieder — wie die Informationen selbst — dezentral von den Nutzern kommen. Als Lösung bieten sich Bewertungen an, die Nutzer für die subjektive Qualität eines Beitrags vergeben können. Aus der Menge der Bewertungen kann das System dann geeignete Schlüsse ziehen.

Als Motivation zur konstruktiven Mitarbeit sollten den Nutzern Anreize gegeben werden, beispielsweise erweiterte Zugriffs- und Änderungsrechte, wenn die bisherigen Beiträge von den anderen Nutzern für gut befunden wurden. Die Frage, wie ein Bewertungs- und Anreizsystem ausgelegt sein muss, um konstruktive Mitarbeit zu erkennen und zu belohnen, ist Gegenstand aktueller Forschungen.

[1] Es wird nicht die *PUT*-Methode verwendet, sondern die ebenfalls in [13] beschriebene *POST*-Methode, mit der Formulardaten zum Server geschickt werden. Im Unterschied zur ersten Methode wird daraus dann erst auf dem Server die eigentliche Web-Seite generiert.

1.2 Zielsetzung

Ziel dieser Diplomarbeit ist der Entwurf und die Implementierung eines Java-Rahmenwerks zum kooperativen Aufbau einer strukturierten Wissensbasis, einer Ontologie (siehe Abschnitt 2.1). Das Rahmenwerk muss die Abgabe von Bewertungen durch die Nutzer des Systems vorsehen, ebenso die daraus resultierende Anreiz- und Rechtevergabe.

Da das Rahmenwerk zur Erforschung unterschiedlicher Bewertungs- und Anreizmechanismen genutzt werden wird, müssen sich diese Teile flexibel austauschen und die übrigen Teile möglichst gut wiederverwenden lassen. Wie das Rahmenwerk aufgeteilt ist und welche Bestandteile genau vorhanden sein sollten, muss zunächst analysiert werden.

Weil das Rahmenwerk hauptsächlich in Web-Applikationen zum Einsatz kommen wird, soll aufbauend auf dem Rahmenwerk auch ein lauffähiger Prototyp einer solchen Web-Applikation entwickelt werden.

Als Name des Rahmenwerks wird *ConsensusFoundation* („die Basis für Konsens“) gewählt.

1.3 Hinweise zum Aufbau dieser Arbeit und zur Notation

In Kapitel 2 werden zunächst die fachlichen Anforderungen ermittelt. Vor allem wird dabei analysiert, welche Art der Wissensrepräsentation zum Einsatz kommen soll und welchen Erfordernissen das Bewertungs- und Anreizsystem genügen muss. Kapitel 3 beschreibt danach die Richtlinien, die beim Entwurf der Schnittstellen und der Architektur des Systems beachtet werden sollen.

Nachdem die fachlichen und technischen Anforderungen vorliegen, kann Kapitel 4 den Entwurf vorstellen und diskutieren. Auch wenn eine funktionierende Implementierung des Rahmenwerks inklusive einer Beispiel-Anwendung explizites Ziel dieser Arbeit ist, wird der Quelltext des Projekts dabei nur vereinzelt aufgeführt. Sofern anhand der Implementation eine Entwurfsentscheidung nicht besser erläutert werden kann, tauchen Hinweise zur Implementation nur in den Fußnoten auf, damit ein Anwender des Rahmenwerks die passenden Einstiegspunkte in das Projekt findet. Weitere Hinweise zur konkreten Umsetzung finden sich im Anhang A sowie in der Javadoc-Dokumentation.

Kapitel 5 wertet anschließend aus, ob das fertige Rahmenwerk die gestellten Anforderungen erfüllt, und Kapitel 6 fasst die Ergebnisse der einzelnen Kapitel noch einmal zusammen. Abgerundet wird diese Arbeit durch einen ausführlichen Index, der vor allem für Anwender des Rahmenwerks gedacht ist, die während der Realisierung einer konkreten Applikation schnell die Diskussion zu einer bestimmten Schnittstelle, Klasse oder Architektur-Entscheidung finden möchten.

Zum Schluss noch einige Hinweise zur Notation:

Typen und Pakete werden in Schreibmaschinenschrift (Typewriter) dargestellt (`Typname` bzw. `de.paket.unterpaket`), ebenso Quelltext innerhalb des Fließtextes. Abgesetzte Listings sind kursiv gesetzt und mit Zeilennummern versehen:

```
Listings
als separater Absatz
```

Wenn eine `package`-Anweisung am Anfang des Listings angegeben ist, kann der vollständige Quelltext des gekürzten Listings im Projekt eingesehen werden. Ansonsten handelt es sich nur um ein Beispiel-Listing, das nicht im Projekt verwendet wird.

Normalerweise werden deutsche Fachbegriffe verwendet, die englischen werden aber in Klammern angegeben, falls sie geläufig sind (beispielsweise bei den Entwurfsmustern). Wenn ein Begriff einmal in normaler Schrift (beispielsweise OntologyManager) und ein anderes Mal in Schreibmaschinenschrift (entsprechend `OntologyManager`) erscheint, ist bei Letzterem eine bestimmte Schnittstelle bzw. Klasse gemeint, die als Quelltext vorliegt, und bei Ersterem allgemein irgendein Modul, das die gleichnamige Schnittstelle realisiert.

2 Fachliche Analyse

Dieses Kapitel wird die Entscheidungsgrundlage dafür liefern, welche Funktionalitäten ConsensusFoundation anbieten muss, in welche Bereiche das Rahmenwerk sinnvoll gegliedert wird und wie die Wissensbasis verwaltet werden soll. Dazu wird zunächst untersucht, was Wissen ist und wie es strukturiert und repräsentiert werden kann. Nach der Auswahl einer geeigneten Wissensrepräsentation erfolgt die Analyse bestehender Bewertungssysteme. Dabei werden nicht die Bewertungsalgorithmen betrachtet, sondern die Datenstrukturen, die zur Verwaltung der diversen Varianten im neuen Rahmenwerk nötig sein werden. Zum Schluss werden weitere generelle Eigenschaften von Mehrbenutzer-Anwendungen ermittelt und als Daten-Modell das Analyse-Schema entworfen, auf dem der Entwurf aufsetzen kann.

2.1 Wissensrepräsentation

Wissensrepräsentation bezeichnet die Form, in der Wissen formal abgebildet wird. Daher wird zunächst geklärt, was Wissen ist, und anschließend werden verschiedene Repräsentationsformen vorgestellt.

2.1.1 Wissen

Es gibt zahlreiche Ansätze zur Definition des Begriffs „Wissen“, aber keiner davon ist allgemein gültig für alle Kontexte. Im Alltag bezeichnet man mit Wissen *alle Kenntnisse im Rahmen alltäglicher Handlungs- und Sachzusammenhänge*, in der Philosophie *die begründete und begründbare (rationale) Erkenntnis im Unterschied zur Vermutung und Meinung oder zum Glauben* — Wissen kann demnach *primär durch systematische Erforschung (Experiment) oder deduzierende Erkenntnis gewonnen werden* [25].

In der Informatik kann ausgehend von einem *Signal* (einer zeitlichen Veränderung einer physikalischen Größe) eine *Nachricht* abstrahiert werden. Durch Interpretation der Nachricht in einem Kontext ordnet man ihr eine Bedeutung zu, die *Information*. Das Paar von Nachricht und zugeordneter Information wird *Datum* genannt. Sowohl die Kenntnis der Information des Datums als auch die Kenntnis von Interpretationsvorschriften zur Erzeugung solcher Informationen wird dann als *Wissen* zusammengefasst [36, Seite 1-4].

Die Unterscheidung zwischen Daten, Information und Wissen findet sich auch in der *Semiotik*[2] [36, Abschnitt 1.1.5] wieder. Danach werden die strukturellen Beziehungen zwischen den Zeichen, in denen die Daten kodiert sind, *Syntax* genannt. Die Beziehungen zwischen den Daten, die sich aus den Interpretationsvorschriften ergeben, heißt *Semantik*. Und wenn die Daten zu Gegenständen und Sachverhalten außerhalb der Datenmenge (z.B. in der realen Welt) in Beziehung gesetzt werden, sprechen wir von der *Pragmatik*. Wissen entsteht durch die Pragmatik, d.h. durch die Verknüpfung der semantisch angereicherten Daten mit der realen Welt. Abbildung 1 zeigt noch einmal die gerade beschriebenen Zusammenhänge. Im Weiteren werden wir diese Beschreibung von Wissen verwenden, da sie für unseren Anwendungsfall, das Wissensmanagement, gut geeignet ist.

Welche Arten der Organisation bzw. Repräsentation gibt es nun für solches Wissen? [63, Kapitel 4] benennt dafür unter anderem folgende Formen:

Systematik oder Begriffssystem: Die allgemeinste Form einer geordneten Zusammenstellung, der Oberbegriff für Systeme aus klar abgegrenzten Begriffen und deren Bezeichnungen. Die Begriffe sind durch Beziehungen miteinander verbunden, und die Struktur des Systems kann gewissen Regeln unterworfen sein.

[2] Lehre von den Zeichen

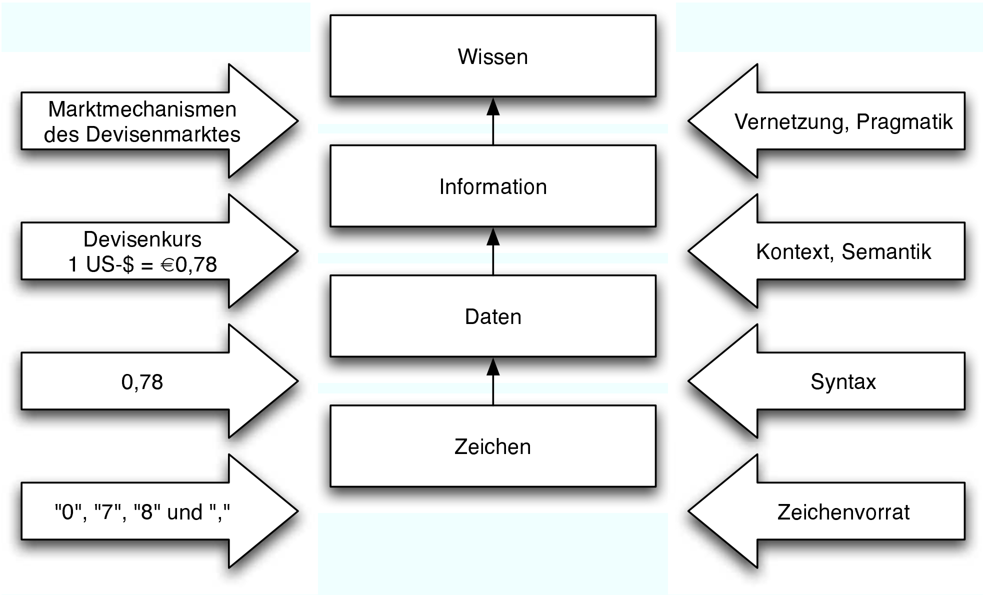


Abbildung 1: Begriffshierarchie für „Wissen“ nach [82]

Thesaurus: Enthält die Begriffe und Benennungen eines Sach- oder Fachgebiets bzw. einer Fachsprache, außerdem die Beziehungen zwischen ihnen (Ober- bzw. Unterbegriff, Verwandtschaft, Teil-Ganzes-Beziehung).

Klassifikation oder Taxonomie: Teilt die Wissensobjekte in Klassen oder Kategorien ein, die nach einem bestimmten System normalerweise hierarchisch geordnet sind.

Semantisches Netz: Besteht aus Begriffen (Konzepten) als Knoten eines Graphen, die durch benannte (typisierte) Kanten verbunden sind. Durch diese Beziehungen sollen Ausschnitte der Welt dargestellt werden, und man möchte Schlussfolgerungen (Inferenzen) aus dem vorhandenen Wissen ziehen können.

In fast allen oben genannten Definitionen taucht das Wort „Begriff“ auf. Was hat dieses Wort mit dem Wissen — den zweckorientierten, mit der realen Welt vernetzten Daten, die repräsentiert werden sollen — zu tun? Dazu sehen wir uns ein Konstrukt aus der Philosophie und Linguistik an, das *semiotische Dreieck* (siehe [91] und Abbildung 2). Wenn Sie z.B. das Wort „Baum“ lesen, ein Symbol oder eine Bezeichnung, bekommen Sie sofort eine Vorstellung davon, was dieses Wort bedeutet (entsprechend wird diese Vorstellung auch „Bedeutung“ oder „Begriff“ genannt) und welchen Zweck es in der realen Welt erfüllt (hier mit „Ding“ bezeichnet). Das Symbol ist somit ein Stellvertreter für den realen Baum, der vor unserem Fenster wächst, und mit unserem Wissen können wir die Bedeutung und den Zweck des Symbols erfassen. Symbole und deren Bedeutung können wir aber formal als Begriffe speichern und somit Wissen repräsentieren. Begriffe werden auch *Konzepte* oder *Klassen* genannt.

Im Wissensmanagement unterscheidet man zwischen *implizitem* bzw. *stillem Wissen* (tacit knowledge) und *explizitem Wissen* (explicit knowledge). Explizites Wissen liegt formalisiert vor. Das implizite Wissen kann dagegen nicht formal niedergeschrieben, sondern nur von Mensch zu Mensch weitergegeben werden, beispielsweise durch das Vorführen von Prozessabläufen. Es hat also sehr viel mit Erfahrung und Können zu tun. Ein Unternehmen wird deswegen großes Interesse daran haben, das implizite in explizites, formal fassbares Wissen zu konvertieren, um eine präzise, eindeutige Kommunikation des Wissens zwischen den Mitarbeitern — oder zwischen Mensch und Maschine — zu erreichen. Jegliches vom Rechner verarbeitbare Wissen ist per Definition explizit [91].

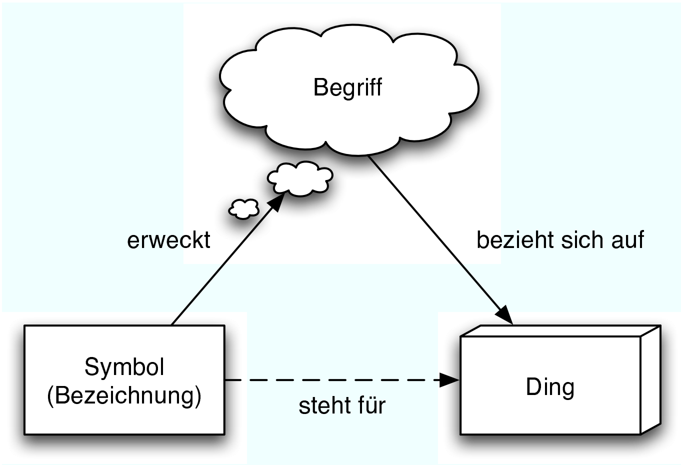


Abbildung 2: Semiotisches Dreieck nach [91]

Bei der Formalisierung des expliziten Wissens gibt es zwei Extreme, die vollständige Formalisierung und überhaupt keine Formalisierung. Letzteres ergibt kaum Sinn, da sich relevantes Wissen dann nur schwer wieder finden lässt, und Ersteres ist nicht praktikabel, da Wissen derzeit nicht automatisiert mit ausreichender Qualität formalisiert werden kann. Zudem ist der Aufwand für die manuelle Formalisierung zu hoch. Einen Kompromiss zwischen beiden Extremen stellen *Metadaten* dar. Dies sind Daten, mit denen der Zweck anderer Daten beschrieben wird. Entsprechend können sie eingesetzt werden, um ausgewählte Aspekte des Wissens zu formalisieren. Das konkrete Format der Metadaten spielt dabei eine untergeordnete Rolle [91], in Abschnitt 2.1.5 wird RDF als ein Beispiel vorgestellt. Das Bewertungssystem, das ConsensusFoundation anbieten wird, kann zur Forschung genutzt werden, wie die automatisierte Formalisierung unterstützt werden kann, indem das System aus den Bewertungen der Begriffe (Konzepte) und ihren Beziehungen durch die Anwender des Systems geeignete Schlüsse zieht.

Die Metadaten können wir also nutzen, um Daten mit Bedeutung zu versehen. Wenn Metadaten dazu verwendet werden, um auch die Beziehungen zwischen den semantisch angereicherten Daten zu beschreiben, kann man damit Wissen repräsentieren, was nun anhand einer Repräsentationsform, der Ontologien, beschrieben wird.

2.1.2 Ontologien

Ontologien sind formale Modelle einer Anwendungsdomäne, die dazu dienen, den Austausch und das Teilen von Wissen zu erleichtern. Sie machen sich dafür die verschiedenen im vorangegangenen Abschnitt vorgestellten Formen der Wissensorganisation und -repräsentation zu Nutze (Thesauri, Taxonomien, semantische Netze) [91]. Eine allgemeine Definition einer Ontologie ist schwierig. Je nach Ansatz lassen sich auch Klassifikationen und Thesauri als Vorläufer, Alternativen oder spezielle Formen von Ontologien auffassen [75].

Der Begriff stammt ursprünglich aus der Philosophie und ist eine *im 17. Jh. entstandene Bez. für die Lehre vom Seienden [...], die die formalen (obersten Strukturen und Gesetzlichkeiten) und materialen (inhaltl. Gliederung des Seienden) Prinzipien der Wirklichkeit begrifflich zu bestimmen sucht* [25].

In der Informatik ist das „Sein“ schwer formal zu fassen, daher behandeln Ontologien hier die Verarbeitung von explizitem Wissen, d.h. von formalisierten Begriffen (Konzepten) und ihren Beziehungen.

Entsprechend lautet eine bekannte Kurzdefinition wie folgt: *Eine Ontologie ist eine explizite Spezifikation einer Konzeptualisierung*[3] [39].

Etwas formaler lässt sich eine Ontologie O als 4-Tupel $\langle C, R, I, A \rangle$ definieren [29]. Die Elemente haben dabei folgende Eigenschaften:

- C ist eine Menge von Konzepten (Concepts), beispielsweise „Stadt“ und „Land“. Dies sind die Basiselemente der Ontologie. Sie werden normalerweise mit der „ist ein(e)“-Relation in einer Klassenhierarchie angeordnet. Beispielsweise kann das Konzept „Stadt“ Unterklasse des Konzepts „Siedlung“ sein, wohingegen zwischen „Stadt“ und „Land“ keine „ist ein“-Beziehung besteht.

- R ist eine Menge von Relationen (beispielsweise „liegt in“). Sie bilden die Verknüpfungen (Beziehungen) zwischen den Konzepten. Normalerweise werden zweistellige (binäre) Relationen eingesetzt, aber n-stellige Relationen sind ebenfalls abbildbar. Die „ist ein“-Relation ist Element dieser Menge.

- I ist eine Menge von Instanzen (beispielsweise „Karlsruhe“ und „Deutschland“). Sie stellen die Elemente der Wissensbasis dar und sind konkrete Beispiele eines Konzepts („Karlsruhe“ ist eine Instanz von „Stadt“). Instanzen werden auch „Exemplare“ oder „Individuen“ genannt.

- A ist eine Menge von Axiomen mit allgemeingültigen Aussagen (beispielsweise „wenn eine Stadt in einem Land liegt und ein Fluss durch die Stadt führt, dann fließt der Fluss auch durch das Land“). Mit den Axiomen kann man die Informationen der Ontologie verifizieren oder aus ihnen per Inferenz neue Informationen schlussfolgern.

Die vier Mengen sind nicht notwendigerweise disjunkt.

Ontologien haben in den letzten Jahren stark an Interesse gewonnen, weil sie auch hinter der Idee des **semantischen Web** (Semantic Web) [100] stecken. Während bereits das „normale“ Web mit seinen in HTML [49] oder XHTML [103] formulierten Seiten unzählige Daten enthält, aus denen ein Mensch nützliche Informationen herauslesen kann, erkennt eine Maschine bei der Zeichenkette „3-89842-447-2“ nicht ohne weiteres, dass es sich hierbei um eine ISBN-Angabe handelt, selbst wenn die Angabe auf der Seite eines Online-Buchhändlers neben einem Buchtitel steht. Daher hat das W3C mit dem *Resource Description Framework* (RDF) eine formale Sprache zur Bereitstellung von Metadaten entwickelt, um den Daten im Web eine Bedeutung zuordnen und sie dann maschinell verarbeiten zu können [14, Kapitel 13].

RDF, das normalerweise in XML [30] formuliert wird, ist die Grundlage des semantischen Web. Es eignet sich jedoch nicht zur Beschreibung einer Ontologie, weil ein genormtes Vokabular fehlt, um beispielsweise Oberklassen-Unterklassen- und Typ-Instanz-Beziehungen zwischen den Ressourcen zu beschreiben. Dies ist mit den auf RDF aufbauenden Sprachen *RDF Schema* (RDFS) und *Web Ontology Language* (OWL) möglich, mit denen sich Ontologien formalisieren lassen. Ein anderer Ansatz zur Repräsentation von Ontologien im Web sind *Topic Maps* (Themenlandkarten), ein ISO-Standard. Während RDFS und OWL vor allem für die maschinelle Auswertung entwickelt wurden, sind Topic Maps für die menschliche Wissensverarbeitung gedacht. Sie sollen die einfache, für Menschen verständliche Strukturierung von und die ebenso einfache Navigation und Suche in einer Wissensbasis ermöglichen [98].

Die Interoperabilität von RDF und Topic Maps wird in [79] diskutiert und für realisierbar befunden. Ebenso wird als Ausblick gegeben, dass die Umwandlung der Daten und deren Semantik auch auf

[3] „An ontology is an explicit specification of a conceptualization.“

RDFS und OWL erweiterbar sein sollte. Entsprechend wurde in [24] bereits ein Vorschlag vorgestellt, wie Topic Maps in OWL formuliert werden können.

Das ConsensusFoundation-Rahmenwerk soll auf der Grundlage von Topic Maps entworfen werden, d.h. die Schnittstellen und Funktionalitäten sollen an bestehende Topic-Map-Implementierungen angelehnt werden. Bei speziellen Details muss abgewogen werden, ob sie explizit erwünscht sind oder ob sie die Interoperabilität mit anderen Repräsentationsformen unnötig erschweren. Ausschlaggebend für die Verwendung von Topic Maps war, dass bei ConsensusFoundation-Anwendungen die Menschen im Mittelpunkt stehen werden, die das Wissen kooperativ sammeln und gegenseitig bewerten, und dass eine Interoperabilität zu den W3C-Sprachen des semantischen Web generell gegeben ist. Topic Maps werden daher im folgenden Abschnitt ausführlicher vorgestellt, anschließend wird aber auch ein kurzer Überblick über RDF, RDFS und OWL gegeben.

2.1.3 Topic Maps

Topic Maps (Themenlandkarten) sind ein ISO-Standard [15] und werden normalerweise als „XML Topic Maps" (XTM) formuliert [77]. Es gibt weitere, auch herstellerspezifische Formate, beispielsweise die lineare Notation LTM [34]. Wir werden im Folgenden jedoch nur XTM betrachten, da XML [30] auch Grundlage für RDF und die darauf aufbauenden Sprachen des semantischen Web ist.

Die Idee der Topic Maps lässt sich zu „TAO" und „IFS" zusammenfassen, was für folgende Begriffe steht [78]:

Topics sind die Konzepte, Begriffe bzw. Themen in einer Topic-Map-Ontologie, beispielsweise „Stadt" oder „Karlsruhe". Es wird also zunächst kein Unterschied zwischen Konzepten (Klassen) und Instanzen gemacht, wie sie die Ontologie-Definition (wenn auch nicht zwingend) vorsieht. Topics können aber beliebig viele Typen besitzen, so kann z.B. dem Topic „Karlsruhe" der Typ „Stadt" zugewiesen werden — ein Typ ist selbst wieder ein Topic. Durch die Verwendung werden bestimmte Topics also als Klassen ausgezeichnet. Im Weiteren werden wir noch sehen, wie Oberklassen-Unterklassen-Beziehungen und Typ-Instanz-Beziehungen durch spezielle Relationen (Assoziationen) besser ausgedrückt werden können.

Occurrences („Vorkommen" oder „Auftreten") sind einem Topic zugeordnet und beschreiben Referenzen auf dieses Topic, die außerhalb der Topic Map bestehen (z.B. in einem Dokument im World Wide Web). Dies kann beispielsweise ein Kommentar zum oder ein Foto vom Topic-Gegenstand sein. Occurrences können auch einen Typ besitzen, der wieder durch ein Topic ausgedrückt wird. Hierdurch lassen sich bestimmte Arten von Occurrences (Bilder, Videos etc.) kategorisieren.

Assoziationen stellen die Beziehungen zwischen den Topics her, beispielsweise „Karlsruhe *liegt in* Deutschland" oder „Deutschland *gehört zur* EU." Die Assoziationen sind nicht gerichtet. Um trotzdem die verschiedenen Teilnehmer der Assoziation unterscheiden zu können, ordnet man die Teilnehmer (auch *Spieler* genannt) verschiedenen *Rollen* zu. Im letzten Beispiel könnte man als Rollen „Staat" und „Staatenverbund" verwenden. Topic-Map-Assoziationen können beliebig viele Spieler in beliebig vielen Rollen enthalten, man kann also problemlos n-stellige Relationen abbilden. Jede Assoziation und jede Rolle kann einen eigenen Typ besitzen, der wieder durch ein Topic definiert wird.

Identität: Ontologien enthalten Begriffe, die mit den Dingen der realen Welt verknüpft sind. Topics entsprechen dabei den Begriffen, und die Dinge der realen Welt werden bei einer Topic Map *Subjekte* genannt (engl. „Subject", was hier auch mit „Gegenstand" oder „Thema" übersetzt werden kann). Jedes Subjekt gibt es nur ein einziges Mal (so wie Sie nur ein einziges Mal existieren), und normalerweise wird ein Subjekt durch exakt ein Topic repräsentiert. Wenn nun aber mehrere Topic Maps exisitieren, kann darin durchaus jeweils ein separates Topic für das eine Subjekt vorhanden sein. Bei Buchverlagen in Deutschland, England und Frankreich würden die Topics „Buch", „book" und „livre" alle dasselbe Subjekt bezeichnen. Das Subjekt muss daher eine eindeutige *Subjekt-Identität* besitzen, die als URI formuliert wird. Bei einer adressierbaren Ressource (beispielsweise dem Dokument der XTM-Spezifikation) wird deren Adresse als Subjekt-Identität benutzt (hier „http://www.topicmaps.org/xtm/1.0/"). Die meisten Subjekte sind aber nicht direkt adressierbar, beispielsweise „Deutschland". Die Adresse „http://www.deutschland.de/" eignet sich hier nicht als Identität, denn dies ist sicher nicht die einzige Ressource zu Deutschland. In diesem Fall wird man eine eindeutige URI festlegen und als Identität des Subjekts definieren — bei Topic Maps spricht man von einem *Subject Indicator* (im ISO-Standard wird dies „Subject Descriptor" genannt). Zwei Topics, die dasselbe Subjekt repräsentieren (d.h. mit derselben Subjekt-Identität), werden als semantisch äquivalent zu einem fusionierten Topic betrachtet, das die Eigenschaften beider ursprünglicher Topics besitzt.

Damit solche Subjekt-Indikatoren portabel werden, d.h. in mehreren Ontologien — nicht notwendigerweise nur in Topic Maps — genutzt werden können, müssen sie zusammen mit ihrer Definition veröffentlicht werden. Man spricht dann von einem *Published Subject Indicator* (PSI).[4]

Facetten (Facets) dienen dazu, weitere Metadaten zu einem Topic anzugeben, beispielsweise in welcher Sprache ein Topic verfasst ist. Jede Facette enthält dabei einen Wert — Facetten sind also *Attribute* eines Topics. Weil man dies aber auch mit anderen Konstrukten (z.B. Assoziationen) realisieren kann, wurden Facetten nicht in den XTM-Standard übernommen. ConsensusFoundation wird die Idee der Attribute aber wieder aufgreifen, damit man attributierte Topics leichter nutzen kann.

Scope: Mit dem Scope (Gültigkeitsbereich) können zum einen gleichnamige Topics (Homonyme) unterschieden werden. Beispielsweise kann „Karlsruhe" sowohl für die Stadt als auch für das gleichnamige Schiff stehen, hier würde man als Scope entsprechend „Stadt" bzw. „Schiff" verwenden. Zum anderen kann damit die Gültigkeit von Topic-Namen eingeschränkt werden. Ein Topic kann z.B. sowohl den Namen „Buch" im Scope „Deutsch" als auch den Namen „book" im Scope „English" besitzen. Ein Scope wird wieder durch ein Topic angegeben.

Reifikation (Reification) ist ein wichtiges Konzept von Ontologien, das auch von Topic Maps unterstützt wird. Es wird damit die Vergegenständlichung von Dingen bezeichnet, um ihnen Eigenschaften zuweisen und Aussagen über sie treffen zu können. Beispielsweise kann man in der Topic Map erst dann eine Aussage über Karlsruhe treffen, wenn man ein entsprechendes Topic „Karlsruhe" erzeugt hat. Reifikation ist also das Erzeugen eines Topics für ein Ding, und das Ding (hier Karlsruhe) wird damit zum Subjekt (Gegenstand, Thema) des Topics. Interessant wird Reifikation dadurch, dass sie auf diverse Elemente der Topic Map angewendet werden kann, beispielsweise auch auf Assoziationen. Dadurch können Aussagen über Aussagen getroffen werden ([77] und [35, Abschnitt 5.3.4]).

2.1.4 Klassifikation bei Topic Maps

Topics und die übrigen Konzepte der Ontologie können mit einem Typ versehen werden, Topics sogar mit mehreren. Der Typ ist dabei selbst ein Topic. Beispielsweise kann dem Topic „Karlsruhe" der Typ „Stadt" gegeben werden. Der Typ wird also zur Einordnung eines Topics in eine oder mehrere Gruppen bzw. Kategorien genutzt, man spricht von Taxonomie oder Klassifikation [58]. Die freie Enzyklopädie Wikipedia [101] bietet die Klassifikation von Artikeln als Kategorien an, die — sofern vorhanden — am Ende eines Artikels angezeigt werden (siehe Abbildung 3).

[4] Für Personen existiert das FOAF-Projekt, mit dem sich jeder seine eigene Subjekt-Identität geben kann [93].

Abbildung 3: Klassifikation in Wikipedia mit Kategorien

Wie von Datenbanksystemen bekannt wird meistens strikt zwischen dem (Datenbasis-)Schema (in unserem Beispiel: den Typen) und der Datenbasis (hier: den Instanzen) unterschieden [61, Abschnitt 2.2.4]. Die Typen werden dabei durch speziell ausgezeichnete Klassen-Topics repräsentiert, wobei durch Ober- und Unterklassen eine Generalisierung bzw. Spezialisierung innerhalb des Typ-Systems erreicht wird — eine Klassenhierarchie entsteht. Instanz-Topics kommen dabei nur als Blätter der Klassenhierarchie vor, können also selbst nicht Typ für ein anderes Topic sein (siehe Abbildung 4 links).

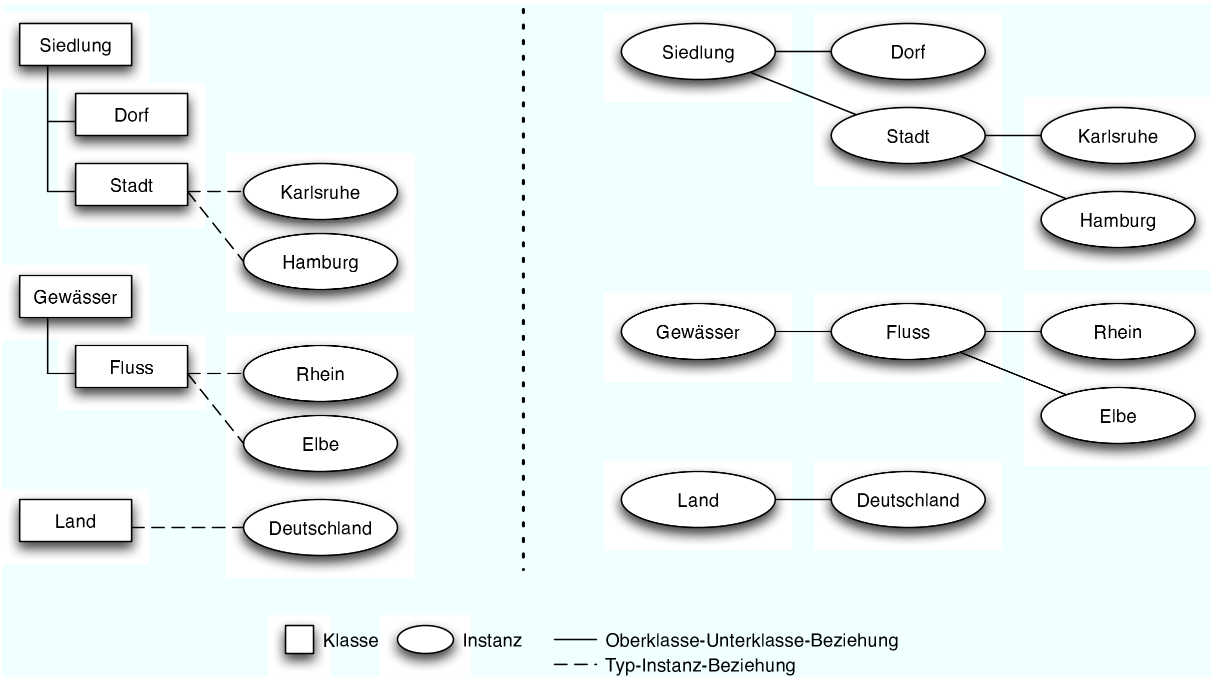


Abbildung 4: Topic-Hierarchie mit und ohne Unterscheidung von Klassen und Instanzen

Nun werden die Ontologien mit ConsensusFoundation aber in der Regel kooperativ, dezentral erstellt, und in diesem Fall gibt es eventuell keine zentrale Administration und kein zentrales Expertenwissen, die ein festes Typ-Schema vorgeben können. Ebenso kann es explizit erwünscht sein, dass das Typ-System dezentral von den Anwendern entwickelt wird. Während Experten Typen und Instanzen unterscheiden können, sind normale Anwender mit diesem Konzept leicht überfordert (beispielsweise mit der Frage, ob „Fluss“ eine Instanz von „Gewässer“ ist oder doch eher eine Unterklasse). Hier kann es sinnvoll sein, den Anwendern eine Kategorisierung bzw. Klassifikation zu ermöglichen, ohne zwischen Klassen und Instanzen zu unterscheiden — ein Instanz-Topic kann dann auch Typ eines anderen Topics sein (siehe Abbildung 4 rechts). Bei Bedarf könnte die Applikation nach einer gewissen Zeit anhand der Topic-Hierarchie und der Bewertungen entscheiden, welches Topic vermutlich eine Klasse und welches eine Instanz ist.

Wir haben gesehen, dass Topics mit anderen Topics typisiert werden können. Wie dies in **XTM** abgebildet wird, zeigt Listing 1. Darin ist eine vollständige Topic Map mit zwei Topics enthalten, wobei das eine („Stadt“) Typ des anderen („Karlsruhe“) ist. Die Beziehung wird über das `<instanceOf>`-Element hergestellt.

```
<?xml version="1.0" encoding="UTF-8"?>
<topicMap id="tmid1"
   xml:base="http://uka.de/ipd/dbis/consensus/beispiel1.xtm"
   xmlns="http://www.topicmaps.org/xtm/1.0/"
   xmlns:xlink="http://www.w3.org/1999/xlink">

  <topic id="id1">
    <baseName>
      <baseNameString>Stadt</baseNameString>
    </baseName>
  </topic>

  <topic id="id2">
    <instanceOf>
      <topicRef xlink:href="#id1"/>
    </instanceOf>
    <baseName>
      <baseNameString>Karlsruhe</baseNameString>
    </baseName>
  </topic>

</topicMap>
```

Listing 1: XTM-Topic-Map mit interner Typ-Instanz-Beziehung

Dieser Ansatz ist leicht zu nutzen, hat aber den Nachteil, dass nicht zwischen Klassen und Instanzen unterschieden werden kann — Instanz-Topics können daher auch wieder Typ für ein anderes Topic sein. Das kann zwar erwünscht sein, man sollte aber zumindest die Möglichkeit haben, eine explizite Klassenhierarchie aufzubauen. Und während man im obigen Beispiel „Stadt“ noch als Klasse erkennen könnte, weil das Topic keine Instanz eines anderen ist, funktioniert dieses Vorgehen nicht mehr, sobald man eine Klassenhierarchie benötigt.

Daher kann man bei Topic Maps auch einen anderen Ansatz verfolgen und die Oberklassen-Unterklassen- sowie die Klasse-Instanz-Beziehungen über explizite und passend typisierte Assoziationen abbilden. Die Topics für diese Typen werden dafür mit den Standard-PSIs für „superclass-subclass relationship“ und „class-instance relationship“ reifiziert (entsprechend auch die Topics für die benötigten Rollen). In Listing 2 wird die Beziehung zwischen den beiden Topics „Stadt“ und „Karlsruhe“

(Zeile 28-38) nun über das `<association>`-Element hergestellt (Zeilen 40-56). Das Listing zeigt aus Platzgründen nur die Klasse-Instanz-Beziehung, für Oberklassen-Unterklassen-Beziehungen müssten weitere Topics definiert und reifiziert werden (analog zu den Zeilen 7-26). Dieser Ansatz hat nebenbei den Vorteil, dass die Bewertung solcher Beziehungen leichter realisierbar wird, weil die Beziehung nun als explizite Assoziation vorliegt und nicht erst aufwändig extrahiert werden muss.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <topicMap id="tmid2"
3     xml:base="http://uka.de/ipd/dbis/consensus/beispiel2.xtm"
4     xmlns="http://www.topicmaps.org/xtm/1.0/"
5     xmlns:xlink="http://www.w3.org/1999/xlink">
6
7     <topic id="id_ClassInstance">
8       <subjectIdentity>
9         <subjectIndicatorRef xlink:href=
10            "http://www.topicmaps.org/xtm/1.0/core.xtm#class-instance"/>
11      </subjectIdentity>
12    </topic>
13
14    <topic id="id_Class">
15      <subjectIdentity>
16        <subjectIndicatorRef xlink:href=
17            "http://www.topicmaps.org/xtm/1.0/core.xtm#class"/>
18      </subjectIdentity>
19    </topic>
20
21    <topic id="id_Instance">
22      <subjectIdentity>
23        <subjectIndicatorRef xlink:href=
24            "http://www.topicmaps.org/xtm/1.0/core.xtm#instance"/>
25      </subjectIdentity>
26    </topic>
27
28    <topic id="id1">
29      <baseName>
30        <baseNameString>Stadt</baseNameString>
31      </baseName>
32    </topic>
33
34    <topic id="id2">
35      <baseName>
36        <baseNameString>Karlsruhe</baseNameString>
37      </baseName>
38    </topic>
39
40    <association id="id3">
41      <instanceOf>
42        <topicRef xlink:href="#id_ClassInstance"/>
43      </instanceOf>
44      <member id="id4">
```

```
      <roleSpec>
        <topicRef xlink:href="#id_Class"/>
      </roleSpec>
      <topicRef xlink:href="#id1"/>
    </member>
    <member id="id5">
      <roleSpec>
        <topicRef xlink:href="#id_Instance"/>
      </roleSpec>
      <topicRef xlink:href="#id2"/>
    </member>
  </association>

</topicMap>
```

Listing 2: XTM-Topic-Map mit expliziter Klasse-Instanz-Assoziation

Für die vollständige Beschreibung der Syntax des XTM-Formats sei auf die Spezifikation der XML Topic Maps verwiesen [77].

2.1.5 Resource Description Framework (RDF)

Das *Resource Description Framework* (RDF) ist eine Sprache, um Metadaten zu Ressourcen des World Wide Web hinzuzufügen. RDF ist dabei hauptsächlich zur maschinellen Verarbeitung der Informationen gedacht, nicht so sehr zur Anzeige für menschliche Nutzer. Ziel ist es, die Bedeutung von Ressourcen zwischen Anwendungen austauschbar zu machen, beispielsweise welche Zahl auf einer Web-Seite für den Buchpreis steht und welche Zeichenkette die ISBN enthält. Dazu müssen die Ressourcen eindeutig über eine URI identifizierbar sein [64].

Die Aussage über eine Ressource wird dabei als Beschreibung seiner Eigenschaften (Properties) realisiert. Dies erfolgt in Form eines Tripels ⟨Subjekt, Prädikat, Objekt⟩, beispielsweise

http://www.snailshell.de/uni/dipl/ hat einen Autor (Creator) und zwar Thomas Much

Das Subjekt ist hier die Web-Seite zu dieser Diplomarbeit. Das Prädikat ist „Autor (Creator)“ und wird durch die URI „http://purl.org/dc/elements/1.1/creator“ repräsentiert, das Objekt durch die URI des Autors („http://www.snailshell.de/foaf.rdf#me“). Dieses Tripel kann als RDF-Graph visualisiert werden, mit Knoten für Subjekt und Objekt sowie einem Pfeil für das Prädikat (siehe Abbildung 5). Mit RDF können auch Literale (Zahlen, Zeichenketten) beschrieben werden, die man im Graph als Rechteck darstellt [64, Abschnitt 2].

Während die Graph-Notation einfach zu lesen ist, wird in der Praxis normalerweise die XML-Syntax von RDF eingesetzt, genannt *RDF/XML* [11]. Listing 3 zeigt, wie der Graph aus obiger Abbildung in RDF/XML formuliert wird. Mit dem Element `<rdf:Description>` wird eine Ressource beschrieben, die Aussagen über die Eigenschaften stecken in den Unterelementen. Das dort verwendete `<dc:creator>`-Element ist über einen XML-Namensraum eingebunden, der die Beschreibung für die *Dublin Core*-Metadaten [26] referenziert [64, Abschnitt 3].

```
<?xml version="1.0"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:dc="http://purl.org/dc/elements/1.1/">
```

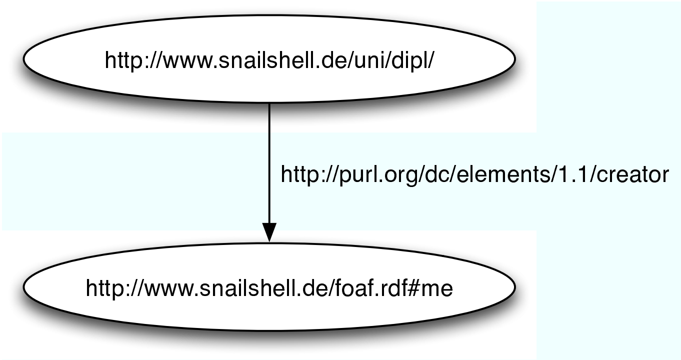


Abbildung 5: Graph für ein RDF-Tripel

```

  <rdf:Description rdf:about="http://www.snailshell.de/uni/dipl/">
    <dc:creator rdf:resource="http://www.snailshell.de/foaf.rdf#me"/>
  </rdf:Description>

</rdf:RDF>
```

Listing 3: XML-Syntax für RDF

RDF legt soweit also nur die Syntax fest, wie die Bedeutung einzelner Ressourcen beschrieben wird und wie Ressourcen über Aussagen verknüpft werden. Was nun noch fehlt, um auch einfache Ontologien beschreiben zu können, ist ein Vokabular, mit dem sich verschiedene Anwendungen auf gemeinsame Bedeutungen einigen sowie auf Regeln, wie dieses Vokabular eingesetzt wird. **RDF Schema** (RDFS) [17] definiert entsprechend ein Typsystem, das es erlaubt, Ressourcen hierarchisch in Klassen einzuteilen und Ressourcen als Instanzen von Klassen zu beschreiben [64, Abschnitt 5]. Listing 4 zeigt, wie mit den Elementen `<rdfs:Class>` und `<rdfs:subClassOf>` die Klasse „Stadt“ als Unterklasse von „Siedlung“ definiert (Zeile 8-12) und beides als neues Typ-Schema zur Verfügung gestellt wird. In dem Schema wird außerdem eine Eigenschaft (Property) jeder Siedlung definiert, nämlich dass sie eine Einwohnerzahl besitzt, die eine ganze Zahl sein muss (Zeile 16-19). Dabei sieht man gut, dass RDFS Klassen und Eigenschaften separat definiert und durch das Bezugselement (Domain) und den Wertebereich (Range) jeder Eigenschaft verknüpft.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
3  <rdf:RDF
4     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
6     xml:base="http://www.snailshell.de/schema/beispiel">
7
8    <rdfs:Class rdf:ID="Siedlung"/>
9
10   <rdfs:Class rdf:ID="Stadt">
11     <rdfs:subClassOf rdf:resource="#Siedlung"/>
12   </rdfs:Class>
13
14   <rdfs:Datatype rdf:about="&xsd;integer"/>
15
16   <rdf:Property rdf:ID="Einwohnerzahl">
```

```
      <rdfs:domain rdf:resource="#Siedlung"/>
      <rdfs:range rdf:resource="&xsd;integer"/>
   </rdf:Property>

</rdf:RDF>
```

Listing 4: Klassenhierarchie in RDF Schema

Listing 5 macht von diesem Typ-Schema Gebrauch und legt die Ressource „Karlsruhe“ als Instanz der Klasse „Stadt“ (Zeile 7) mit passender Einwohnerzahl[5] (Zeile 8-10) an.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
3  <rdf:RDF
4     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5     xmlns:bsp="http://www.snailshell.de/schema/beispiel#"
6     xml:base="http://www.snailshell.de/beispiel">
7
8    <bsp:Stadt rdf:ID="Karlsruhe">
9       <bsp:Einwohnerzahl rdf:datatype="&xsd;integer">
10         275049
11      </bsp:Einwohnerzahl>
12   </bsp:Stadt>
13
14 </rdf:RDF>
```

Listing 5: Instanz einer Klasse eines RDF-Schemas

Reifikation wird in RDF erreicht, indem ein ⟨Subjekt, Prädikat, Objekt⟩-Tripel durch ein `<rdf:Statement>`-Element ausgedrückt wird, das durch eine URI angesprochen werden kann. Dadurch kann in anderen Aussagen auf diese Aussage, d.h. auf das Tripel, verwiesen werden [64, Abschnitt 4.3].

2.1.6 Web Ontology Language (OWL)

Die *Web Ontology Language* (OWL[6]) geht in der Ausdrucksstärke über RDF Schema hinaus. So können mit OWL komplexere Zusammenhänge von Klassen (Schnitt, Vereinigung, Komplement) und die Kardinalitäten von Eigenschaften beschrieben werden. Dazu stehen drei Sprachversionen zur Verfügung, die im Folgenden kurz in aufsteigender Ausdrucksstärke beschrieben werden [65, Abschnitt 1.3]:

OWL Lite kann zur einfachen Umstellung von Taxonomien und Thesauri auf OWL genutzt werden. Klassifikation und Klassenhierarchien werden unterstützt, als Kardinalitätswerte werden aber nur 0 und 1 angeboten. OWL Lite kann als Erweiterung einer eingeschränkten RDF-Variante betrachtet werden.

[5] Stand: Ende 2005

[6] Das entsprechende Akronym müsste eigentlich WOL lauten, aber man hat sich schließlich für OWL entschieden — in Anlehnung an A A. Milnes einflussreiches Buch „Pu der Bär“, in dem die weise Eule (Owl) ihren Namen schreiben kann, aber leider einen Buchstabendreher einbaut [46].

OWL DL bietet eine hohe Ausdrucksstärke, erhält aber gleichzeitig die Berechenbarkeit und Entscheidbarkeit der Aussagen. Dazu werden die Sprachkonstrukte gewissen Einschränkungen unterworfen, beispielsweise können Klassen nur Unterklassen, nicht aber Instanzen anderer Klassen sein. Der Name „DL“ leitet sich von der Beschreibungslogik (Description Logics) ab, die eine entscheidbare Untermenge der Prädikatenlogik erster Stufe darstellt. Auch OWL DL ist eine Erweiterung einer eingeschränkten RDF-Variante.

OWL Full bietet alle Freiheiten der RDF-Syntax (und ist entsprechend eine Erweiterung von RDF), gibt aber keine Garantien zur Berechenbarkeit. Die Einschränkungen von OWL DL sind nicht vorhanden, hier können nun auch Klassen als Instanzen anderer Klassen beschrieben werden. [42, Abschnitt 4 R13] gibt dafür den Anwendungsfall der Klasse „Orang-Utan“ an, die viele Individuen als Instanzen besitzt, selbst aber Instanz der Klasse „Art“ ist. „Orang-Utan“ darf keine Unterklasse von „Art“ sein, weil sonst jedes Orang-Utan-Individuum eine Instanz von „Art“ wäre, was aus Sicht der Biologie keinen Sinn ergibt.

Da OWL Full eine Erweiterung von OWL DL und OWL DL eine Erweiterung von OWL Lite ist, ergeben sich daraus die folgenden Implikationen [65, Abschnitt 1.3]:

- O ist zulässige Ontologie in OWL Lite $\rightarrow$ O ist zulässige Ontologie in OWL DL
- O ist zulässige Ontologie in OWL DL $\rightarrow$ O ist zulässige Ontologie in OWL Full
- i ist gültige Schlussfolgerung in OWL Lite $\rightarrow$ i ist gültige Schlussfolgerung in OWL DL
- i ist gültige Schlussfolgerung in OWL DL $\rightarrow$ i ist gültige Schlussfolgerung in OWL Full

Wenn RDF-Dokumente nicht speziell für OWL Lite oder OWL DL geschrieben werden, gehören sie aufgrund der fehlenden Einschränkungen normalerweise zu OWL Full.

Angelehnt an die Beispiele in [90] wird in Listing 6 eine Ontologie in OWL konstruiert. Zunächst wird die Ontologie normalerweise mit dem `<owl:Ontology>`-Element allgemein beschrieben. Neben dem Namen und einem Kommentar können hier auch Versionierungs-Informationen angegeben und andere Ontologien importiert werden (Zeile 9-14, hier teilweise abgekürzt mit Auslassungszeichen dargestellt). Danach werden mit `<owl:Class>` die Klassen definiert. Die Eigenschaft „liegtIn“ (Zeile 33-36) ist im Wertebereich auf „Land“ festgelegt und erlaubt als Bezugselement „Thing“ — dies ist eine spezielle OWL-Klasse, die jede Instanz zulässt. Für die Klasse „Siedlung“ wird diese Eigenschaft mit `<owl:Restriction>` eingeschränkt, jede Siedlung liegt danach in exakt einem Land (Zeile 19-26). Die Instanzen schließlich werden in den Zeilen 38-47 angelegt, wobei bei der Instanz „Hamburg“ explizit angegeben wird, dass es sich um eine andere Instanz als „Karlsruhe“ handelt (was sich sonst nicht zwingend schlussfolgern lässt, wenn eine Instanz mehrere Namen besitzt). Durch solche zusätzlichen Aussagen können Fehler in der Ontologie besser entdeckt werden.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE rdf:RDF [<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">]>
3  <rdf:RDF
4     xmlns:owl ="http://www.w3.org/2002/07/owl#"
5     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
6     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
7     xmlns:xsd ="http://www.w3.org/2001/XMLSchema#">
8
9    <owl:Ontology rdf:about="">
10     <rdfs:label>Stadt Land Fluss</rdfs:label>
```

```
    <rdfs:comment>Eine Beispiel-Ontologie in OWL</rdfs:comment>
    <owl:priorVersion rdf:resource="..." />
    <owl:imports rdf:resource="..." />
  </owl:Ontology>

  <owl:Class rdf:ID="Land" />

  <owl:Class rdf:ID="Siedlung">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#liegtIn"/>
        <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
          1
        </owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:ID="Stadt">
    <rdfs:subClassOf rdf:resource="#Siedlung" />
  </owl:Class>

  <owl:ObjectProperty rdf:ID="liegtIn">
    <rdfs:domain rdf:resource="http://www.w3.org/2002/07/owl#Thing" />
    <rdfs:range rdf:resource="#Land" />
  </owl:ObjectProperty>

  <Land rdf:ID="Deutschland" />

  <Stadt rdf:ID="Karlsruhe">
    <liegtIn rdf:resource="#Deutschland" />
  </Stadt>

  <Stadt rdf:ID="Hamburg">
    <liegtIn rdf:resource="#Deutschland" />
    <owl:differentFrom rdf:resource="#Karlsruhe"/>
  </Stadt>

</rdf:RDF>
```

Listing 6: Ontologie in OWL

Wie schon bei RDF und RDFS wurden bei weitem nicht alle Möglichkeiten gezeigt, die OWL bietet. Es sollte aber deutlich geworden sein, wie die Sprachen aufeinander aufbauen und welche Ergänzungen jeweils genutzt werden können. Zur Vertiefung sei auf die angegebenen Quellen verwiesen.

Während RDFS und OWL sich gut für die maschinelle Wissensverarbeitung eignen, sind Topic Maps besser für die menschliche Wissensverarbeitung geeignet. Da Letztere vorrangiges Ziel von ConsensusFoundation ist, wurde bereits in Abschnitt 2.1.2 die Entscheidung für den Einsatz von Topic Maps und somit gegen RDFS bzw. OWL getroffen.

2.1.7 Werkzeuge

Es gibt zahlreiche Werkzeuge zum Betrachten und zum Bearbeiten von Ontologien. Da mit ConsensusFoundation nicht nur bestehende Ontologien betrachtet und bewertet, sondern auch neue Ontologien erstellt werden sollen, wurden einige vorhandene Editoren genauer untersucht, um die Anforderungen für die Benutzungsoberfläche eines Ontologie-Editors aufstellen zu können. Vor allem wurden die zwei folgenden, bekannten Ontologie-Editoren betrachtet:

Protégé kann OWL- und RDF-Ontologien laden und speichern und ist kostenlos verfügbar [94].

Ontopoly ist ein Topic-Map-Editor und Bestandteil der Ontopia Knowledge Suite (OKS). Er baut auf Ontopias „Web Editor Framework“ auf und verarbeitet Ontologien im XTM- und LTM-Format. Eine kostenfreie Testversion kann zeitlich beschränkt eingesetzt werden [76].

Aus Platzgründen werden die Eigenschaften dieser Editoren hier nicht beschrieben, sie sind aber in den Entwurf der Beispiel-Anwendung eingeflossen. Einen Überblick über zahlreiche weitere Ontologie-Werkzeuge liefert [80].

Während die Editoren die Datenstrukturen der erwähnten Formate erzeugen und bearbeiten können, fehlt ihnen ein Element, das kein Bestandteil der gängigen Ontologie-Sprachen ist, aber bei ConsensusFoundation von zentraler Bedeutung sein wird: Bewertungen.

2.2 Bewertungssysteme

Ontologien repräsentieren Wissen. Wenn aber — wie es für ConsensusFoundation-Anwendungen vorgesehen ist — eine Nutzergemeinschaft eine solche Wissensbasis kooperativ aufbaut, ist nicht klar, inwieweit die gesammelten Informationen gesichertes Wissen darstellen. Die Anwender können aus Unwissenheit oder leider auch absichtlich (böswillig) falsche Informationen eingeben. Man möchte also ermitteln, ob die Informationen qualitativ gut sind. Bei Ontologien gehört dazu vor allem auch die Frage, ob die Beziehungen zwischen den Konzepten korrekt sind: Stimmt die Klassifikation eines Konzepts, d.h. wurde die Klassenhierarchie korrekt aufgebaut? Wurden die Typ-Instanz-Beziehungen korrekt angewendet?

Bei der Enzyklopädie „Wikipedia“ kommt es derzeit nur auf die Qualität der Artikeltexte an, da Wikipedia keine Ontologie ist — die nötigen semantischen Informationen fehlen.[7] Die Qualitätssicherung erfolgt hier zum einen durch eine öffentliche Diskussion, die zu jedem Artikel abrufbar ist. Zum anderen können Administratoren bestimmte Seiten überwachen und bei erkennbar schlechten Änderungen alte Versionen zurückspielen — zu diesem Zweck wurden kürzlich auch die Qualitätssicherungsseiten eingeführt [102]. Aber auch bei echten Ontologien verlässt man sich derzeit meistens auf eine zentrale Kontrolle bzw. wenige Informationsquellen, die hier Domänenexperten genannt werden [91].

Solche zentralen Stellen zur Wissensaufbereitung bzw. zur Kontrolle möchte man aber vermeiden und stattdessen die Qualität der Informationen weitestgehend automatisch ermitteln. Bei vielen Online-Angeboten gibt es bereits eine Lösung, um die Qualität der angebotenen Daten dezentral zu bestimmten: Die Benutzer des Systems können die Daten bewerten. Die Idee dahinter ist, dass man die Anwender über die jeweiligen Daten abstimmen lässt und dass das System dann aus allen abgegebenen Bewertungen die Qualität eines Beitrags (oder eines Produkts) errechnet.

[7] Erweiterungen für ein semantisch untermauertes Wikipedia werden derzeit diskutiert [86].

Wie man die Anwender überhaupt dazu motiviert, Inhalte einzubringen und Bewertungen abzugeben, und wie man wahrheitswidrige Bewertungen vermeiden kann, wurde bereits an anderer Stelle untersucht ([18] und [48]) und ist nicht Bestandteil dieser Arbeit. ConsensusFoundation muss aber die dafür nötigen Bewertungen unterstützen und auch geeignete Anreize bieten können.

Daher werden im Folgenden ausgewählte Bewertungssysteme von Diskussionsforen, Nutzergemeinschaften und Handelsplattformen kurz vorgestellt, woraus sich dann die Anforderungen für ConsensusFoundation ergeben. Dabei werden nicht die Bewertungsalgorithmen untersucht, sondern *was* bewertet werden kann und welche Werte dafür zur Verfügung stehen.

2.2.1 Slashdot

Slashdot [89] ist eine der ältesten und sicherlich eine der bekanntesten Online-Nutzergemeinschaften für Computer- und Technikbegeisterte. Eingereichte Artikel werden von einem zentralen Redaktionsteam veröffentlicht, sofern der Artikel interessant und gut genug erscheint. Die Artikel können von allen Lesern kommentiert werden, ebenso können sie auch die Kommentare selbst wieder kommentieren.

Registrierte Leser können nach bestimmten Regeln zu Moderatoren werden und dürfen dann die Kommentare bewerten. Zum einen soll damit die Qualität eines Kommentars ermittelt werden, damit andere Leser qualitativ schlechte Kommentare ausfiltern können. Zum anderen wird dadurch das sogenannte *Karma* des Kommentators beeinflusst, das angibt, wie gut seine Beiträge sind. Das Karma ist unter anderem ein Faktor dafür, ob ein Leser zum Moderator ernannt wird.

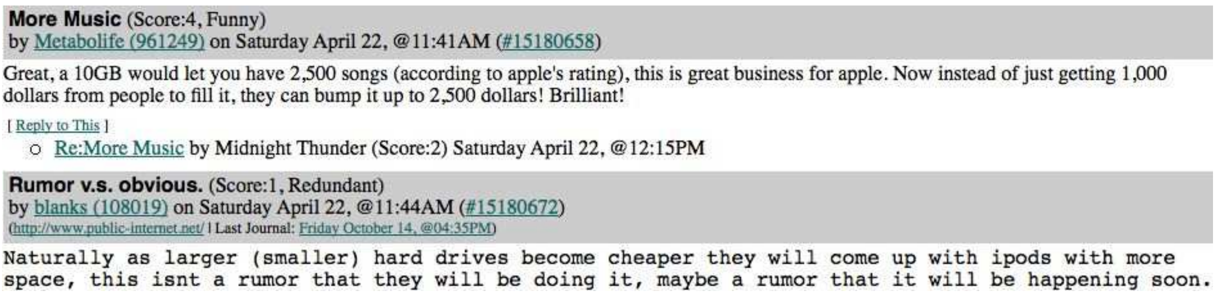


Abbildung 6: Artikel-Kommentare mit Punktevergabe auf slashdot.org

Die bewertbaren Elemente sind also einzig die Kommentare. Moderatoren können ihre Meinung zu einem Kommentar anhand einer festen Liste von beschreibenden Wörtern kundtun. Ein positives Wort verbessert die Bewertung, ein negatives verschlechtert sie. Jede Bewertung wird durch eine Punktzahl (den „Score“) auf der diskreten Skala von -1 bis 5 dargestellt (siehe Abbildung 6), wobei -1 der schlechteste und 5 der bestmögliche Score ist.

2.2.2 Amazon

Beim Online-Händler *Amazon* [2] ist für die Kunden die Qualität der angebotenen Produkte interessant. Daher können die Produkte von allen angemeldeten Nutzern auf einer diskreten Skala von null bis fünf Sternen bewertet werden, was den Qualitätsstufen „schlecht“ bis „sehr gut“ entspricht. Zusätzlich kann zu jeder Bewertung noch ein beliebiger Text als Kommentar abgegeben werden. Beides zusammen wird als „Rezension“ bezeichnet.

Nach einer redaktionellen Prüfung des Kommentars wird die Bewertung zusammen mit dem Durchschnitt aller Bewertungen für das Produkt unter dem Produkt angezeigt (siehe Abbildung 7). Während

Abbildung 7: Bewertung eines Produkts auf amazon.de

bei der Bewertung nur ganze Sterne vergeben werden können, wird der Durchschnitt bei Bedarf etwas genauer mit halben Sternen dargestellt.

Interessant ist die Möglichkeit, dass jeder Nutzer die Rezensionen dahingehend bewerten kann, ob sie hilfreich waren oder nicht. Dadurch werden böswillige Manipulationen oder unqualifizierte Rezensionen unwahrscheinlicher bzw. sie können von den Kunden besser als solche erkannt werden.

Abbildung 8: Bewertung eines Verkäufers auf amazon.de

Amazon ist nicht nur selber Händler, sondern auch eine Handelsplattform für fremde Verkäufer. Nur weil ein Kunde Amazon vertraut und dort einkauft, bedeutet dies nicht automatisch, dass er auch allen anderen Händlern vertraut. Daher können Kunden ebenfalls die Qualität (Zuverlässigkeit, Service) externer Händler bewerten (siehe Abbildung 8), wiederum in sechs Stufen von null bis fünf Sternen. Hier fällt die zeitliche Gewichtung der Bewertungen auf: Es wird zwar auch die Gesamtanzahl aller Bewertungen angezeigt, aber die durchschnittliche Bewertung wird nur aus den Bewertungen der letzten 12 Monate ermittelt. So hat ein Verkäufer die Möglichkeit, frühere schlechte Bewertungen durch guten Service auszugleichen, wodurch der Kunde diese Verbesserung wahrnehmen kann. Dies gilt natürlich auch umgekehrt, sollte der Service (und damit vermutlich auch die Bewertungen) schlechter werden. Da es für den Kunden einen großen Unterschied macht, ob ein Verkäufer 90% oder 99% positive Bewertungen erhalten hat, wird die durchschnittliche Bewertung nicht nur als grobe Sterne-Grafik, sondern zusätzlich als Prozentwert angezeigt.

2.2.3 eBay

eBay [27] ist die wohl bekannteste Online-Auktionsplattform. eBay vermittelt Kaufverträge zwischen Käufern und Verkäufern. Jeder registrierte Nutzer kann beide Rollen annehmen. Das Bewertungssystem soll hier Vertrauen auf beiden Seiten aufbauen, weshalb jeder Handelspartner die Möglichkeit hat, den anderen Partner nach erfolgreichem Abschluss einer Auktion zu bewerten. Hierzu stehen die drei

Stufen „positiv“, „neutral“ und „negativ“ zur Verfügung, die für das Bewertungsprofil mit den Werten 1, 0 und -1 berechnet werden (siehe Abbildung 9).

Abbildung 9: Bewertungen eines Nutzers auf ebay.de

Wieder fällt die zeitliche Gewichtung auf, wobei eBay zwischen den Bewertungen des letzten Monats, der letzten 6 und der letzten 12 Monate differenziert. Die Handelspartner haben vom Gegenüber so ein aktuelles Bild der Zuverlässigkeit und können Veränderungen gut nachvollziehen.

Zu jeder Bewertung kann ein kurzer Kommentar vergeben werden, der bei der detaillierten Auflistung der Bewertungen angezeigt wird. Außerdem ist es hier gezielt möglich, nur die Bewertungen von Käufern oder Verkäufern sowie alle selber abgegebenen Bewertungen anzeigen zu lassen.

2.2.4 Heise-Newsticker

Beim *Heise-Newsticker* [45] können — wie bei Slashdot — die Kommentare zu den Meldungen bewertet werden, damit andere Leser einen schnellen Überblick bekommen, welche Kommentare lesenswert sind und welche nicht. Im Gegensatz zu Slashdot kann hier aber jeder angemeldete Benutzer Bewertungen abgeben.

Zur Bewertung stehen die vier Stufen „$--$“ (völlig belangloser Beitrag), „$-$“ (unter dem Durchschnitt), „+“ (über dem Durchschnitt) und „++“ (unbedingt lesenswert) zur Verfügung (siehe Abbildung 10). Der Durchschnitt aller Bewertungen zu einem Kommentar wird als rote bzw. grüne Balkengrafik mit insgesamt 10 Stufen (fünf für Ablehnung und fünf für Zustimmung) dargestellt. Lässt man den Mauszeiger eine gewisse Zeit über der Bewertungsgrafik stehen, wird noch eine Prozentangabe der Ablehnung bzw. Zustimmung von -100% bis +100% angezeigt. Eine einmal abgegebene Bewertung kann nicht mehr geändert und auch nicht mehr zurückgezogen werden.

2.2.5 Consensus Builder

Der *Consensus Builder* ist quasi der Vorgänger dieser Arbeit. Der gleichnamige Prototyp hat die prinzipielle Machbarkeit der kooperativen und anreizbasierten Erstellung von strukturierten Datenbeständen mithilfe von Bewertungen gezeigt [85]. Sein Datenmodell mit Topics, Beziehungen und

Abbildung 10: Bewertung von Kommentaren zu Meldungen auf heise.de

Attributen basiert auf Topic Maps. Die Typhierarchie muss statisch vom Administrator vorgegeben werden, ebenso ist der von den normalen Anwendern vergebenen Typ einer Instanz statisch. Beziehungen können nur binär (zweistellig) sein. Da es sich um einen Prototypen handelt, wurde auf Wartbarkeit und Wiederverwendbarkeit kein Wert gelegt.

Bewertungen können für Topics, Beziehungen und Attribute abgegeben werden. Bei einem Topic wird damit der Typ des Topics bewertet, z.B. ob das Topic „Karlsruhe“ wirklich vom Typ „Stadt“ ist. Neben Zustimmung und Ablehnung kann eine Duplikat-Meldung abgegeben werden, wenn ein Topic synonym zu einem anderen ist. Allerdings ist daraus nicht zu erkennen, zu welchem Topic das bewertete synonym ist. Das Topic wird zwar nicht abgelehnt, aber doch in seiner Relevanz herabgestuft. Besser wäre hier eine Synonym-Beziehung zwischen zwei Topics, die unabhängig von den Topics selbst bewertet werden kann.

Beziehungen können mit Zustimmung und Ablehnung bewertet werden, ebenso Attribute. Bei Letzteren kommt noch die gerade beschriebene Duplikat-Meldung hinzu, wenn es sich um ein mengenwertiges Attribut handelt.

Jede Bewertung kann mit einem kurzen Kommentar versehen werden. Consensus Builder macht einen Unterschied zwischen der Korrektur einer abgegebenen Bewertung und einer Änderung der Bewertung aufgrund einer veränderten Bewertungsgrundlage (Letzteres wird „zeitliche Anpassung“ genannt). Ob dies sinnvoll ist, ist fraglich, denn durch den Kommentar zur Bewertung und einen automatisch gespeicherten Zeitstempel kann man Änderungen bereits gut nachvollziehen — und man spart sich einen unnötig schwer zu handhabenden Sonderfall ein. Alle Bewertungen zu einem Element der Wissensbasis können inklusive Kommentar, Bewerter und Zeitstempel in einer Historie eingesehen werden. Dazu gehören auch eventuelle Korrekturen bzw. Änderungen.

Jede Zustimmung, Ablehnung und Duplikat-Meldung ergibt jeweils einen Punkt. Die Punkte werden separat nach diesen drei Arten aufsummiert und können gegeneinander verrechnet werden.

2.3 Ergebnis der fachlichen Analyse

Nachdem in den vorangegangenen Abschnitten einige Möglichkeiten zur Wissensrepräsentation sowie ausgewählte bestehende Bewertungssysteme untersucht wurden, werden nun zunächst die Anforderungen an das ConsensusFoundation-Bewertungssystem aufgestellt. Anschließend wird mit dem Analyse-Schema beschrieben, wie das Wissen in ConsensusFoundation repräsentiert werden soll. Zum Schluss werden die übrigen Anforderungen (Benutzerverwaltung und Protokollierung) kurz zusammengefasst.

2.3.1 Bewertungen

Die in Abschnitt 2.2 vorgestellten Bewertungssysteme werden in Tabelle 1 noch einmal systematisch zusammengefasst:

Bewertungssystem	*Was wird bewertet?*	*Wer bewertet?*	*Bewertungsstufen*
Slashdot	Kommentare zu Artikeln	Nutzer, die Moderatorenrechte erlangt haben	-1 - 5
Amazon	Produkte (und auch die Bewertungen selbst)	angemeldete Kunden (die Bewertungen selbst alle Nutzer)	0 - 5 Sterne
eBay	Handelspartner	der jeweilige Handelspartner	positiv, neutral, negativ
Heise-Newsticker	Kommentare zu Artikeln	angemeldete Nutzer	$--$, $-$, $+$, $++$
Consensus Builder	Topic-Typen, Beziehungen, Attribute	angemeldete Nutzer	Zustimmung, Ablehnung, Duplikat

Tabelle 1: Vergleich der Bewertungssysteme

ConsensusFoundation wird sich bei den bewertbaren Elementen an den Consensus Builder anlehnen. Die bewertbaren Kommentare und Produkte bei den anderen Bewertungssystemen können problemlos durch Topics repräsentiert werden. Die Handelspartner bei eBay entsprechen den Nutzern des Systems, die in allen vorgestellten Systemen auftauchen. Auch ConsensusFoundation soll Benutzer verwalten, die wie bei eBay von anderen Nutzern bewertet werden können. Sie sollen aber nicht als Topic repräsentiert, sondern durch eine eigene bewertbare Schnittstelle abgebildet werden.

Topic-Typen und synonyme Topics sollen als spezielle, bewertbare Beziehung realisiert werden. Da bei ConsensusFoundation die Klassenhierarchie, d.h. die zur Verfügung stehenden Typen, nicht von einem Administrator festgelegt, sondern kooperativ aufgebaut wird, müssen auch die Oberklassen-Unterklassen-Beziehungen bewertbar sein. Die Bewertungen selbst sollen allerdings nicht bewertbar sein, damit das Datenmodell einfacher gehalten werden kann.

Damit ergeben sich die in Abbildung 11 ersichtlichen Anwendungsfälle für das ConsensusFoundation-Bewertungssystem.

Die verschiedenen Systeme setzen zur internen Repräsentation der Bewertung diskrete Werte ein, allerdings wird kein einheitlicher Wertebereich verwendet. Bei den erwähnten Systemen kommen die Bereiche $-1..5$, $0..5$ und vermutlich $-2..2$ zum Einsatz. ConsensusFoundation könnte einen großen ganzzahligen Wertebereich nutzen, der alle denkbaren Skalen umfasst. Die Entscheidung fällt allerdings zugunsten des stetigen Intervalls $[-1, 1]$, das ebenso gut beliebig auf andere Intervalle (und auch auf diskrete Werte) abgebildet werden kann.

Aufbauend auf dem Bewertungssystem muss ConsensusFoundation ein austauschbares Anreizsystem vorsehen, mit dem Benutzer zur Mitarbeit an der Ontologie und zur Abgabe von Bewertungen motiviert werden können. Zum einen soll es möglich sein, für das Abgeben von Bewertungen (und für weitere Aktionen wie das Einbringen neuer Informationen) Punkte an den jeweiligen Anwender zu vergeben. Zum anderen soll aufbauend auf dem so gesammelten Punktestand als Anreiz eine dynamische Rechtevergabe erfolgen können, damit Anwender mit hohem Punktestand beispielsweise mehr Topics sehen dürfen als Anwender mit wenig Punkten. Wie sich unterschiedliche Punkte- und Rechtevergabe auf die Reaktionen der Anwender auswirkt, soll mithilfe von ConsensusFoundation erforscht werden, daher müssen diese Bereiche flexibel austauschbar sein.

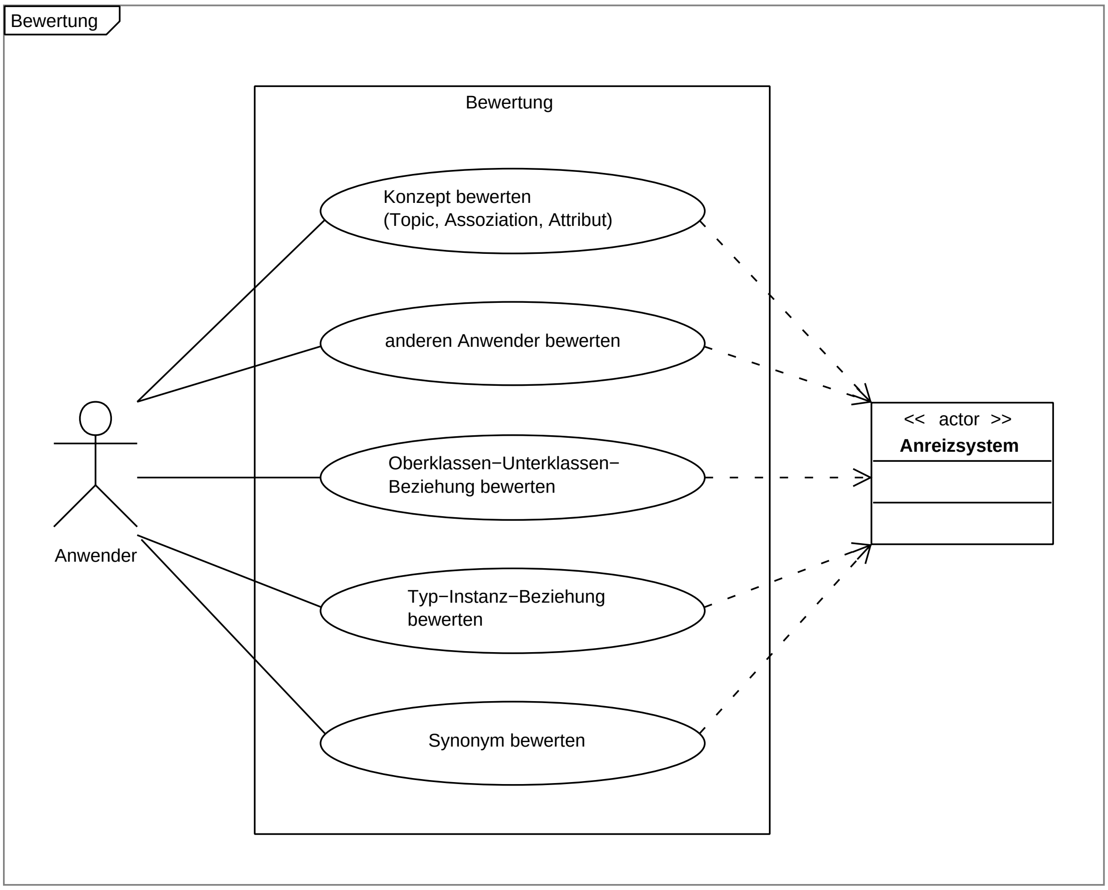


Abbildung 11: Anwendungsfälle für Bewertungen

2.3.2 Das Analyse-Schema

In Abschnitt 2.1.2 wurde das Topic-Map-Modell als Grundlage für ConsensusFoundation ausgewählt, d.h. es müssen Topics, Assoziationen und Occurrences sowie Attribute (Facetten) verwaltet werden. Hinzu kommen Bewertungen und Anwender (Benutzer), wie sie im vorangegangenen Abschnitt diskutiert wurden. Mit diesen Anforderungen wurde das Analyse-Schema als abstraktes Datenmodell für ConsensusFoundation entwickelt (siehe Abbildung 12). Die Fachklassen des Analyse-Schemas werden später im Entwurfs-Schema konkretisiert und verfeinert (siehe Abschnitt 4.1).

Benutzer können demnach Konzepte und Bewertungen anlegen, die jeweils eine Version besitzen. Das System muss später bei Änderungen die Versionen automatisch hochzählen und die Bewertungshistorie zur Verfügung stellen. „Konzept“ wird hier als Oberbegriff für Topics, Assoziationen und Attribute genutzt, also für die wichtigsten Elemente der Ontologie.

Jedes Topic kann beliebig viele Typen, Oberklassen, Unterklassen und Synonyme besitzen, und es kann von einem Topic beliebig viele Instanzen dieses Topic-Typs geben. Alle diese Zuordnungen müssen bewertbar sein, auch wenn der Entwurf hier einige Fälle zusammenfassen kann: Wenn beispielsweise Topic U das Topic O als Oberklasse besitzt, besitzt Topic O automatisch auch Topic U als Unterklasse — beide Beziehungen sollten aber ein und dieselbe Bewertung erhalten.

Ob das System es zulässt, dass ein Topic gleichzeitig Klasse und Instanz ist (wie es auch OWL Full erlaubt), und ob Instanzen selber wieder Typ sein können, hängt von der Konfiguration der Implementierung ab (siehe Abschnitt 4.2).

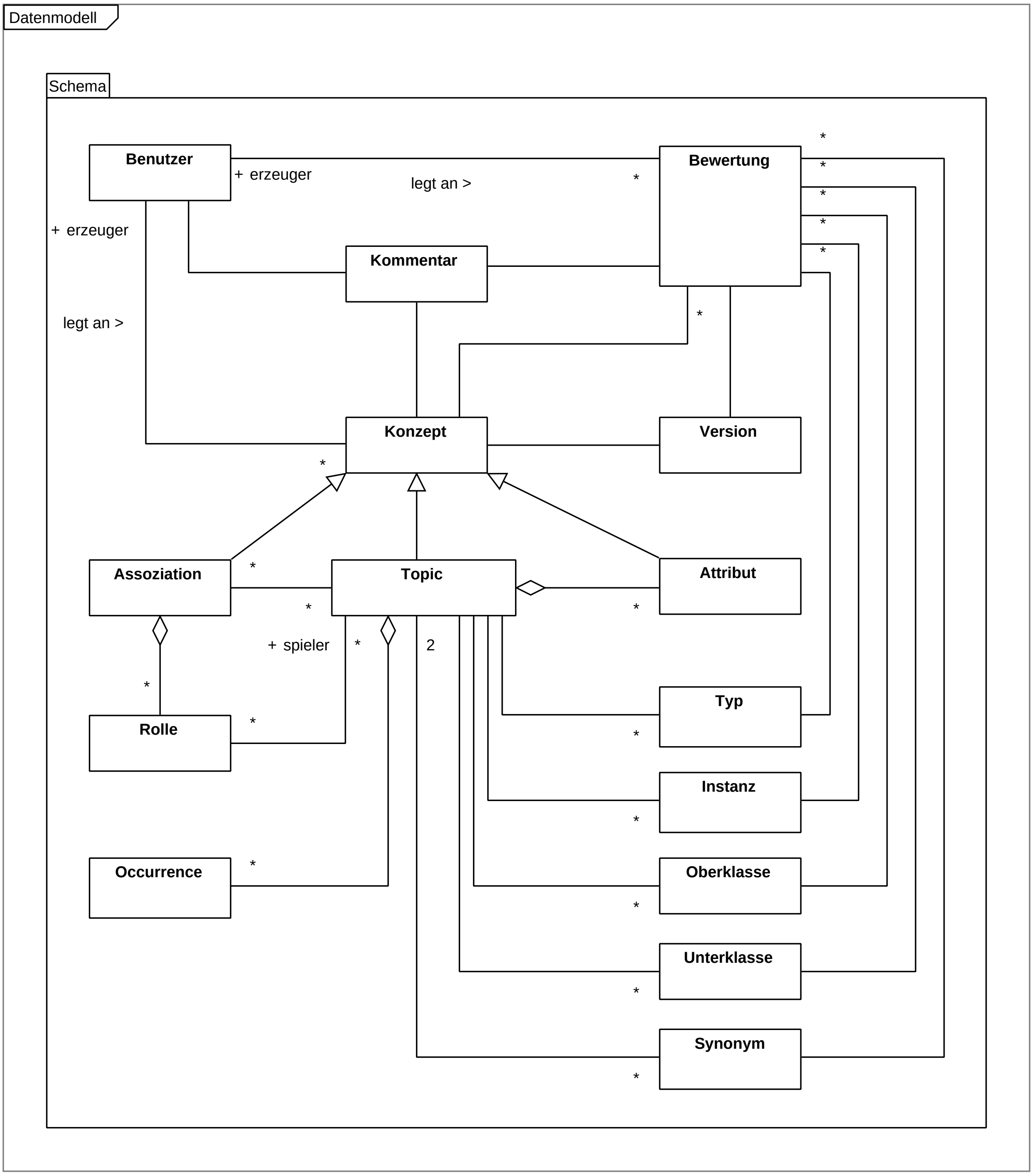


Abbildung 12: Analyse-Schema

Für die Verwaltung des Schemas soll eine eigene Komponente verantwortlich sein, die Ontologieverwaltung, denn die Klassen des Schemas bilden die Elemente in der Ontologie. Diese Komponente ist also für das Anlegen und Ändern der Elemente zuständig und muss zudem geeignete Abfragen der Elemente ermöglichen. Für komplexe, aufwändige Veränderungen zur Evolution der Ontologie sollten zumindest geeignete Schnittstellen zur Verfügung gestellt werden.

2.3.3 Benutzerverwaltung und Protokollierung

Zwei weitere Bereiche muss ConsensusFoundation abdecken: Die Benutzer müssen verwaltet werden, und alle Aktionen im System müssen möglichst einfach und flexibel protokollierbar sein.

Es muss also eine Benutzerverwaltung existieren, die alle gängigen Anwendungsfälle erfüllt: Ein Anwender muss sein eigenes Benutzerkonto verwalten können (anlegen, ändern, löschen), und er muss sich am System an- und abmelden können. Administratoren sollen den Anwendern Rollen zuweisen können, anhand derer bestimmte Rechte gewährt werden können. Außerdem soll es einen Super-Administrator geben, der alle Rechte besitzt und der andere Anwender zu Administratoren machen kann. Diese Anwendungsfälle sind in Abbildung 13 zusammengefasst.

Zur Vergabe von Berechtigungen werden Benutzer oft zu Gruppen zusammengefasst (beispielsweise Gäste, registrierte Anwender, Administratoren), denen die Rollen zugewiesen werden. Den Rollen sind dann bestimmte Rechte zugeordnet. Nicht alle Systeme verwenden jedoch Gruppen, sondern teilen ihren Anwendern direkt eine oder mehrere Rollen zu. Der Servlet-Container „Tomcat“ beispielsweise nutzt keine Gruppen, kann aber seine Rollen auf LDAP-Gruppen verweisen lassen [92]. ConsensusFoundation soll ebenfalls diesen Ansatz wählen und nur mit Rollen arbeiten, um den Applikationen nicht das Gruppen-Konzept aufzuzwingen.

Zur Protokollierung der Aktionen (Logging) muss eine Komponente exisitieren, die alle Aktionen vom System mitgeteilt bekommt. Die Aktionen müssen nach ihrer Art gefiltert werden können, um ganz gezielt nur bestimmte Aktionen mitzuschreiben. Die Komponente muss zudem einfach austauschbar sein, damit das Protokoll ohne allzu großen Aufwand nicht nur auf die Standard-Ausgabe, sondern auch in eine Datei oder Datenbank geschrieben werden kann.

Diese Anforderung für eine flexible Protokollierung ist insbesondere relevant, da ConsensusFoundation als Forschungsinstrument genutzt werden soll, um unterschiedliche Anreize für Benutzer zu testen, mit denen sie zur Abgabe ehrlicher Bewertungen und generell zur Mitarbeit an der Ontologie motiviert werden sollen. Die Reaktionen der Anwender auf die Anreize müssen protokolliert werden, um sie später auswerten und die Auswirkungen der unterschiedlichen Anreize vergleichen zu können.

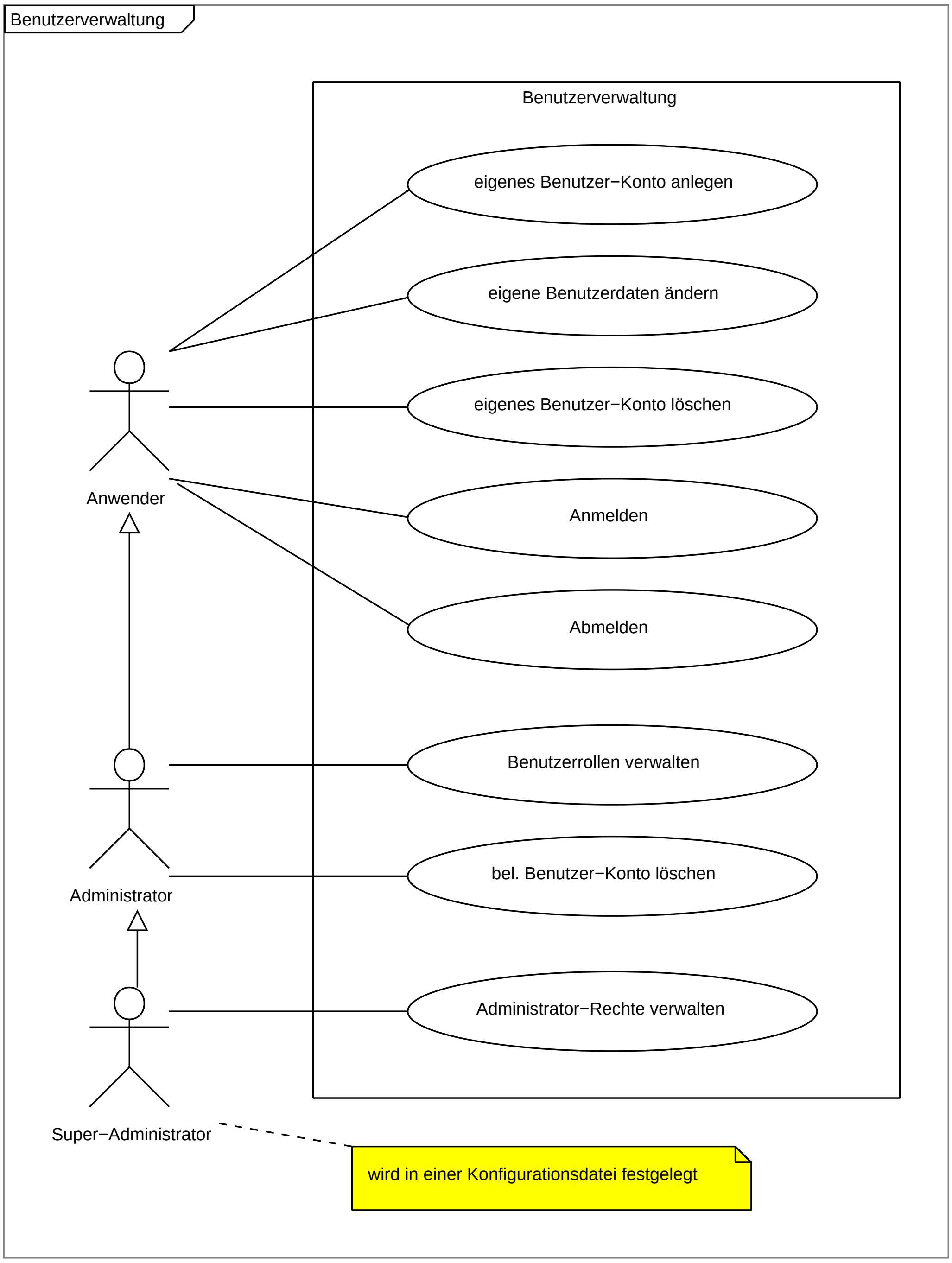


Abbildung 13: Anwendungsfälle der Benutzerverwaltung

3 Entwurfsrichtlinien

Beim Entwurf (und auch bei der Implementierung) sind gewisse Kriterien einzuhalten, damit das fertige Produkt nicht nur fachlich korrekt ist, sondern auch bezüglich Software-Technik und -Architektur gute Software darstellt. Was genau eine gute Qualität und Strukturierung ausmacht bzw. welche Voraussetzungen dafür gegeben sein müssen, wird in diesem Kapitel untersucht.

3.1 Grundlagen und Qualitätsaspekte

Die Aspekte von Software-Qualität werden in *äußere* und *innere Faktoren* aufgeteilt [66, Abschnitt 1.1]. Die äußeren Faktoren sind von den Anwendern der Software zu erkennen, wobei hierzu auch diejenigen zählen, die über den Einsatz eines Produkts entscheiden. Die inneren Faktoren betreffen vor allem Entwurfs- und Implementierungsaspekte wie die Lesbarkeit des Quelltextes oder die Modularität — und sind damit entscheidend für die Wartbarkeit einer Software. Während die äußeren Faktoren über die Akzeptanz der Software entscheiden, ergibt sich deren Qualität aus den inneren Faktoren.

[66, Abschnitt 1.2] definiert fünf äußere Qualitätsfaktoren, die *Schlüsselqualitäten* [66, Abschnitt 1.4]. Sie werden hier gleich in Bezug zu den Anforderungen aus Kapitel 2 gesetzt, die an das Consensus-Foundation-Rahmenwerk bestehen:

Korrektheit: *Korrektheit ist die Fähigkeit von Softwareprodukten, ihre Aufgaben exakt zu erfüllen, wie sie durch Anforderungen und Spezifikation definiert sind.* Dies wird auch als die *primäre Qualität* bezeichnet, denn eine Software soll natürlich den Zweck erfüllen, für den sie entwickelt wird. ConsensusFoundation muss also Ontologien verwalten können sowie den Benutzern erlauben, zu den Konzepten der Ontologie Bewertungen abzugeben etc. (siehe das Ergebnis der fachlichen Analyse in Abschnitt 2.3).

Robustheit: *Robustheit heißt die Fähigkeit von Softwaresystemen, auch unter außergewöhnlichen Bedingungen zu funktionieren.* Die geplanten Anwendungen von ConsensusFoundation werden normalerweise nicht unter wohldefinierten Laborbedingungen ablaufen, sondern häufig in öffentlichen Netzwerken, wo jederzeit eine Ressource ausfallen kann. Wie soll ConsensusFoundation reagieren, wenn mitten im Betrieb die Netzwerkverbindung zur Datenbank abbricht? Die Robustheit besagt hier unter anderem, dass zumindest eine sinnvolle Ausnahmebehandlung stattfinden muss, die programmatisch behandelt werden kann. Korrektheit und Robustheit zusammen kann man dann als *Zuverlässigkeit* eines Systems definieren.

Erweiterbarkeit: *Erweiterbarkeit bezeichnet die Leichtigkeit, mit der Softwareprodukte an Spezifikationsänderungen angepasst werden können.* ConsensusFoundation soll entwickelt werden, um unterschiedliche Anreizmechanismen erforschen zu können. Es ist daher von elementarer Bedeutung, dass die Anpassung bestehender Applikationen an neue Algorithmen zur Punkt- und Anreizvergabe sowie zur Protokollierung der Aktionen (zur späteren Auswertung) mit minimalem Aufwand möglich ist.

Wiederverwendbarkeit: *Die Wiederverwendbarkeit von Softwareprodukten ist die Eigenschaft, ganz oder teilweise für neue Anwendungen wiederverwendet werden zu können.* Während Consensus-Foundation einerseits leicht erweiterbar sein muss, muss andererseits das Rahmensystem selbst in möglichst vielen Anwendungen ohne Änderungen einsetzbar sein. Dies ist ja gerade einer der Gründe für die Entwicklung des Rahmensystems: Es soll nicht ständig das Rad neu erfunden (und beispielsweise die Ontologie-Verwaltung neu programmiert) werden müssen.

Kompatibilität (Verträglichkeit): *Kompatibilität ist das Maß der Leichtigkeit, mit der Softwareprodukte mit anderen verbunden werden können.* Das ConsensusFoundation-Rahmenwerk darf beispielsweise nicht zu viele Annahmen über die Umgebung machen, in der es eingesetzt wird — dies kann eine Web-Applikation auf einem Server sein, aber genauso gut eine Desktop-Anwendung.

Oben wurde bereits der wichtige innere Qualitätsfaktor *Modularität* erwähnt, der sich aus den drei letztgenannten äußeren Faktoren Erweiterungsfähigkeit, Wiederverwandbarkeit und Kompatibilität ergibt. Um genauer zu fassen, was es heißt, eine Software modular zu gestalten, definiert Meyer in [66, Kapitel 2] dazu fünf Kriterien und fünf Prinzipien. Die Prinzipien folgen aus den Kriterien, daher werden nur erstere hier zitiert:

1. **Sprachliche Moduleinheiten**
   Moduln[8] *müssen zu syntaktischen Einheiten der benutzten Sprache passen.*

2. **Wenige Schnittstellen**
   Jeder Modul sollte mit möglichst wenig anderen kommunizieren.

3. **Schmale Schnittstellen (Lose Kopplung)**
   Wenn zwei Moduln überhaupt miteinander kommunizieren, sollten sie so wenig Information wie möglich austauschen.

4. **Explizite Schnittstellen**
   Wenn zwei Moduln A und B kommunizieren, dann muß das aus dem Text von A oder B oder beiden hervorgehen.

5. **Geheimnisprinzip**
   Jede Information über einen Modul sollte modulintern sein, wenn sie nicht ausdrücklich als öffentlich erklärt wird.

Während Meyer diese Prinzipien im Weiteren dazu verwendet, um die Grundlagen des objektorientierten Entwurfs (OOD) und der objektorientierten Programmierung (OOP) mit Klassen, abstrakten (aufgeschobenen) Klassen, Generizität und Vererbung zu motivieren, werden wir sehen, dass uns viele dieser Faktoren und Prinzipien auch bei höheren Abstraktionsstufen begegnen, wenn es um die Entwicklung einer Architektur mit Komponenten und Rahmenwerken geht. Auf die Grundlagen der Objektorientierung wird hier nicht weiter eingegangen, dafür sei auf andere Stellen verwiesen (neben [66] beispielsweise [74, Kapitel 2]). Da aber auch ConsensusFoundation letztendlich programmiert werden muss, werden im Folgenden zunächst noch einige Richtlinien vorgestellt, die sich für Klassen und Schnittstellen (Interfaces) — die sprachlichen Moduleinheiten bei Java — aus den genannten Prinzipien ergeben.

Der Begriff der **Schnittstelle** tauchte schon mehrfach auf. [83, Seite 44] definiert diesen Begriff wie folgt: *Eine Schnittstelle definiert einen Vertrag, der für den Fall des »vertragsgemäßen« Aufrufs die ebenfalls »vertragsgerechte« Ausführung einer bestimmten Aktion garantiert.* Bei Java können damit je nach Kontext eine einzelne Methode oder eine Teilmenge bzw. alle Methoden einer Klasse oder Java-Schnittstelle (Interface) gemeint sein. Es geht also darum, dass die Methoden genau das korrekt ausführen, was der Programmierer implementiert hat und was der Aufrufer der Methode erwartet. Um diese Korrektheit für beide Seiten sicherzustellen, wurde für die Programmiersprache „Eiffel" das *Programmieren als Vertragsabschluss* (Design by Contract) entwickelt, dessen grundlegende Ideen auch beim Einsatz anderer Sprachen berücksichtigt werden sollten. Die Korrektheit wird durch drei Arten von Zusicherungen dokumentiert und geprüft [67, Abschnitt 9.2]:

[8] Während die zitierte Quelle von „der Modul" (Singular) und „die Moduln" (Plural) spricht, wird in dieser Arbeit „das Modul" bzw. „die Module" verwendet.

Vorbedingungen einer Methode müssen vom Aufrufer der Methode garantiert werden, unter anderem indem er gültige Werte als Parameter übergibt.

Nachbedingungen müssen von der Methode selbst nach jedem korrekten Aufruf garantiert werden.

Klassen-Invarianten müssen von der Klasse für jedes neu erzeugte Objekt sowie nach jeder Ausführung einer öffentlichen Methode garantiert werden.

Auch wenn von Java das Programmieren als Vertragsabschluss nicht direkt unterstützt wird, kann man dies teilweise mit der `assert`-Anweisung nachbilden. Ansonsten sollten die Bedingungen zumindest mittels Javadoc dokumentiert sein.

Alle diese Punkte, die Prinzipien und Zusicherungen, zielen auf eines ab: Die Abhängigkeiten zu einer konkreten Implementierung sollen verringert oder am besten sogar ganz eliminiert werden, um eine bessere Qualität der Software zu erreichen. Module (Klassen) sollen nur auf wohldefinierte Schnittstellen zugreifen, die von der Implementierung abstrahieren. [50, Seite 37] spricht von *schüchternem Quelltext*, dessen Module nicht zu viel von sich preisgeben und keine Annahmen über die innere Struktur anderer Module machen.

Das Ergebnis ist eine Entkopplung der Module. Klassen, die nur von anderen Schnittstellen, nicht aber von deren konkreter Implementierung abhängen, können später leichter ausgetauscht werden. Änderungen an einer Implementierung, die aber die Schnittstelle unverändert lassen, ziehen nicht zwingend Änderungen an anderen Klassen, welche die Schnittstelle verwenden, nach sich. Ziel muss es also sein, ConsensusFoundation so weit wie möglich über abstrakte Schnittstellen zu beschreiben und diese Schnittstellen *stabil* zu halten.

[32, Seite 23-30] fasst dies zu zwei Prinzipien des objektorientierten Entwurfs zusammen:

1. *Programmiere auf eine Schnittstelle hin, nicht auf eine Implementierung.*

2. *Ziehe Objektkomposition der Klassenvererbung vor.* Klassenvererbung wird auch als *White-Box-Wiederverwendung* bezeichnet, weil Unterklassen oft auf die Implementationsdetails der Oberklasse zugreifen können — aber dies widerspricht dem Geheimnisprinzip und sollte ja gerade durch die Abstraktion mit Schnittstellen verhindert werden. Bei der Objektkomposition bzw. *Black-Box-Wiederverwendung* werden dagegen nur die wohldefinierten Schnittstellen angesprochen.

Wir begeben uns nun eine Stufe höher und fassen Klassen zu größeren funktionalen Einheiten zusammen — dies wird schließlich zu Komponenten führen (siehe Abschnitt 3.3). Auch hier wird man versuchen, die Abhängigkeiten dieser größeren Einheiten untereinander minimal zu halten und ausschließlich mit wohldefinierten Schnittstellen zu arbeiten. [50, Abschnitt 8] beschreibt dies als *Orthogonalität* der Komponenten, d.h. sie sollen unabhängig von der inneren Struktur der anderen Komponenten sein. Solche in sich abgeschlossenen Komponenten führen wieder zu einer besseren Entkopplung des Gesamtsystems und machen dieses besser beherrschbar.

Letztlich führt diese Zusammenfassung dazu, eine **Architektur** des Softwaresystems zu entwerfen. [83, Seite 1] definiert eine Software-Architektur wie folgt: *Die Software-Architektur ist die grundlegende Organisation eines Systems, dargestellt durch dessen Komponenten, deren Beziehungen zueinander und zur Umgebung, sowie die Prinzipien, die den Entwurf und die Evolution des Systems bestimmen.* Fünf fundamentale Entwurfsprinzipien einer Softwarearchitektur sind in [81, Abschnitt 5.2] beschrieben:

Abstraktion: Beim Entwurf werden die Bausteine (Komponenten) idealisiert und vereinfacht, um die Komplexität zu verringern und das System beherrschen zu können. Auch bei ConsensusFoundation werden die von der fachlichen Analyse identifizierten Teilbereiche zunächst nur grob mit ihren Verantwortlichkeiten und Beziehungen beschrieben, worauf dann später pro Komponente ein Feinentwurf folgt.

Kapselung: Was wir bei den Klassen als Geheimnisprinzip gesehen haben, gilt auch für die Komponenten einer Architektur — sie sollten definierte Verantwortlichkeiten und Schnittstellen besitzen.

Modularität: Auch bei Komponenten ist eine lose Kopplung für die Modularität von elementarer Bedeutung. Hier muss insbesondere auf die richtige Granularität geachtet werden, damit es nicht zu viele Abhängigkeiten gibt (bei zu kleinen Komponenten) oder eine Komponente nicht erst noch weiter unterteilt werden muss (wenn die ursprüngliche zu groß ist).

Hierarchie: Bei den Klassen entstanden die Hierarchien durch Vererbung („ist ein") und Komposition („besteht aus"). Bei Architekturen wird vor allem die letztgenannte, die strukturelle Hierarchie behandelt, und auch bei ConsensusFoundation wird die in der Quelle genannte Abstufung „Schichtenarchitektur – Rahmenwerk (Framework) – Komponenten" eingesetzt werden.

Konzeptuelle Integrität (Einheitlichkeit): Entwurfsentscheidungen sollen durchgängig getroffen und Speziallösungen vermieden werden.

Wie lässt sich nun eine gute Architektur qualitativ erkennen? [81, Abschnitt 5.1.5] gibt zumindest fünf Kriterien an, mit denen sich der korrekte Entwurf formal feststellen lässt, was ein gutes Indiz für eine gute Qualität ist:

Kopplung: Es soll möglichst wenig Beziehungen zwischen den einzelnen Bausteinen geben, denn eine solche lose Kopplung reduziert die Komplexität des Systems.

Kohäsion: Sind die Komponenten unabhängig und dienen sie einem einzigen, wohldefinierten Zweck? Die Bestandteile einer Komponente sollten nicht mehrere Verantwortlichkeiten vermischen.

Zulänglichkeit: Kann der Baustein alle seine Anforderungen mit einer möglichst kleinen Schnittstelle erfüllen?

Vollständigkeit: Deckt der Baustein mit seinen Schnittstellen alle Aufgaben aus seinem Verantwortungsbereich ab?

Einfachheit: Sind die Schnittstellen angemessen, d.h. wurde ein guter Kompromiss zwischen Zulänglichkeit und Vollständigkeit gefunden?

Diese Fragen bzw. Kriterien sind in der Auswertung des fertigen Rahmensystems zu prüfen, zudem werden dort noch andere Aspekte wie die Performanz berücksichtigt.

Wir gehen nun noch einmal auf die Stufe der Klassen zurück und sehen uns ausgehend von dort einige Konzepte zur Strukturierung an, was uns zu immer größeren Einheiten und schließlich zur gewünschten Architektur einer typischen ConsensusFoundation-Anwendung bringt. Die folgenden Abschnitte dienen dabei auch zur Begriffsklärung und Abgrenzung der Konzepte untereinander.

3.2 Entwurfsmuster

Bei der Softwareentwicklung treten immer wieder ähnliche Probleme auf, die entsprechend zu ähnlichen Lösungsansätzen führen. Beim objektorientierten Entwurf bedeutet dies, dass immer wieder ähnliche Klassen mit entsprechenden Beziehungen entwickelt werden, bestimmte Strukturen tauchen also wiederholt auf. Damit solche wiederkehrenden Muster bewusst und gezielt eingesetzt werden können, um von Anfang an einen besseren Entwurf zu erreichen, wurde in [32] erstmals ein Katalog von *Entwurfsmustern* (Design Patterns) zusammengestellt, der im Folgenden kurz vorgestellt wird.

Ein Muster besteht demnach aus

- einem möglichst aussagekräftigen **Musternamen**,
- der **Problembeschreibung**, in der diskutiert wird, in welchem Kontext das Muster sinnvoll angewendet werden kann,
- der **Lösung**, die abstrakt beschreibt, aus welchen Elementen und Beziehungen der Entwurf besteht, und
- den **Konsequenzen**, die aufzeigen, welche Vor- und Nachteile der Entwurf hat.

Anhand ihrer Aufgabe werden die Muster aufgeteilt in

- **Erzeugungsmuster**
- **Strukturmuster** und
- **Verhaltensmuster**.

Erzeugungsmuster dienen zum Erzeugen von Objekten. Ein Beispiel ist die *Abstrakte Fabrik (Abstract Factory)*, häufig auch nur Fabrik (Factory) genannt, die Schnittstellen zum Erzeugen verwandter oder voneinander abhängiger Objekte bietet, ohne die konkreten Klassen zu nennen [32, Seite 107 ff.]. Das vermutlich bekannteste Entwurfsmuster überhaupt, das *Einzelstück (Singleton)*, gehört ebenfalls in diese Kategorie. Dieses Muster stellt sicher, dass es von einer Klasse genau ein Exemplar gibt und ein globaler Zugriffspunkt darauf existiert [32, Seite 157 ff.].

Strukturmuster behandeln die Zusammensetzung von Klassen und Objekten. In diese Kategorie fällt z.B. die *Fassade (Facade)*, die eine einheitliche Schnittstelle zur einfacheren Benutzung einer Menge von Schnittstellen eines Subsystems bietet [32, Seite 212 ff.]. Dadurch können auch die Abhängigkeiten zwischen Subsystemen minimiert werden.

Verhaltensmuster schließlich beschreiben die Zusammenarbeit zwischen Klassen und Objekten. Ein typischer Vertreter dieser Kategorie ist der *Beobachter (Observer)*. Hiermit können bei der Zustandsänderung eines beobachteten Objekts (des „Subjekts“) beliebig viele abhängige Objekte (die Beobachter) benachrichtigt werden [32, Seite 287 ff.]. Weil dies über abstrakte Schnittstellen beschrieben wird, sind Subjekt und Beobachter trotzdem nicht zu eng gekoppelt.

ConsensusFoundation soll die Entwurfsmuster des Buches nutzen, wo es sinnvoll ist. Beim Entwurf wird dazu auf die jeweils verwendeten Muster hingewiesen.

Neben den Entwurfsmustern, die Entwurfsprobleme mittlerer Granularität behandeln, lassen sich *Architekturmuster* identifizieren, die unter anderem die Zerlegung eines Gesamtsystems in Subsysteme beschreiben [83, Abschnitt 17.3]. Ein bekanntes Architekturmuster ist „Model/View/Controller“ (MVC), das die Trennung von Darstellung, Steuerung und Datenmodell vorsieht (siehe Abschnitt 3.6).

3.3 Komponenten

Während Entwurfsmuster die Struktur und die Zusammenarbeit von Klassen und Objekten beschreiben, gibt es in Softwaresystemen größere logische, funktionale Einheiten, die sich aus Klassen zusammensetzen und die dementsprechend mithilfe von Mustern entworfen werden können. Diese Einheiten, die oben Module oder Subsysteme genannt wurden, werden nun genauer als *Komponenten* (Components) definiert:

Komponenten sind modulare Teile eines Systems, die ihren Inhalt und somit komplexes Verhalten transparent kapseln und in ihrer Umgebung als austauschbare Einheiten mit klar definierten Schnittstellen auftauchen. [83, Seite 48]

Ein *Subsystem* ist laut dieser Quelle eine spezielle Komponente, die aus anderen Komponenten und Klassen besteht und zur Zerlegung von größeren Systemen in Untereinheiten dient. Im Unterschied zu Komponenten werden Subsysteme nicht direkt angesprochen, sondern interagieren mit anderen Komponenten.

Da Komponenten die Bausteine einer Software-Architektur sind, gelten für sie die fundamentalen Entwurfsprinzipien aus Abschnitt 3.1. Entsprechend haben sie dieselben Ziele: Modularität, Kapselung, Austauschbarkeit und Wiederverwendbarkeit. Wir haben bereits gesehen, dass die Komponenten eines Systems orthogonal, d.h. unabhängig voneinander, sein sollten, weil dies die genannten Ziele unterstützt. Dies ergibt sich auch aus der Forderung nach Kohäsion, nach der jede Komponente in sich abgeschlossen sein soll.

Der Wunsch nach Kohäsion führt dann zur Komponente als funktionale Einheit, d.h. sie wird zu einem wohldefinierten Zweck entworfen. [83, Abschnitt 4.3] ergänzt daher die Entwurfsprinzipien um die *Trennung von Zuständigkeiten* (Separation of Concerns, SoC), nach der für jeden Aspekt der Problemstellung genau ein Element des Systems verantwortlich ist. Beispielsweise sollte die Protokollierung (das Logging) in einer einzigen Komponente realisiert sein, auch wenn sie dann natürlich von anderen Teilen des Systems angesprochen wird.

Zur Identifikation von Komponenten wird in [83, Kapitel 5] der Begriff der *Software-Kategorie* eingeführt, ein Wissens- oder Sachgebiet, in dem ein Teil der Software realisiert werden soll. Ziel ist es, dass genau eine Komponente für genau eine Kategorie verantwortlich ist — dann hat man die Trennung von Zuständigkeiten erreicht. Dies gilt aber explizit nicht für zusammengesetzte Komponenten, also solche, die ihrerseits aus weiteren Komponenten bestehen. Für nicht zusammengesetzte, einfache Komponenten wird daher zur Unterscheidung auch der Begriff *Modul* genutzt.

Für ConsensusFoundation hat die fachliche Analyse diverse Kategorien identifiziert: Ontologieverwaltung, Benutzerverwaltung, Bewertung, Rechtevergabe, Protokollierung — sie sollen jeweils von einer einfachen Komponente (einem Modul) realisiert werden. Beim Entwurf werden zur Benennung der ConsensusFoundation-Einheiten die Begriffe Komponente und Modul synonym verwendet.

3.4 Klassenbibliotheken

Eine *Klassenbibliothek* (auch objektorientierte Software-Bibliothek genannt) ist eine Sammlung von verwandten und wiederverwendbaren Klassen, die nützliche und allgemeine Funktionalität zur Verfügung stellen [32, Seite 36]. In [31] wird gezeigt, wie solche Bibliotheken robust und flexibel entworfen werden können, wofür viele der in Abschnitt 3.1 vorgestellten Konzepte (u.a. Schnittstellen und Zusicherungen) genauer untersucht werden. Als Vorteile von Klassenbibliotheken nennt [81, Abschnitt 8.3.2] die drei folgenden:

- Sie sind ein *Strukturierungsmittel* durch die Kapselung ähnlicher und zusammengehöriger Funktionalitäten.
- Sie unterstützen die *Wiederverwendung* von Softwarebausteinen.
- Sie werden als *Binärcode* ausgeliefert, d.h. der Quelltext muss nicht veröffentlicht werden.

Der letzte Punkt zeigt, dass es bei Klassenbibliotheken vor allem um die Code-Wiederverwendung geht. Eine bestimmte Anwendungsarchitektur wird von den Bibliotheken nicht festgelegt.

ConsensusFoundation muss daher mehr als eine Klassenbibliothek sein, denn neben der Funktionalität soll auch die Architektur des Systems in einem gewissen Rahmen vorgegeben werden. Allerdings wird ConsensusFoundation bei der Implementierung auf andere Klassenbibliotheken zurückgreifen, um nicht alle Grundfunktionalitäten selber realisieren zu müssen.

3.5 Rahmenwerke

ConsensusFoundation soll als Rahmenwerk (Framework) realisiert werden. Was genau heißt das, und wie unterscheidet sich ein Rahmenwerk von einer Klassenbibliothek? Dazu betrachten wir zunächst zwei Definitionen:

1. *Ein Software-Framework stellt typischerweise ein halbfertiges Architekturgerüst für einen (komplexen) Anwendungsbereich dar, das auf die Bedürfnisse und Anforderungen einer konkreten Anwendung aus diesem Anwendungsbereich angepasst werden kann.* [83, Seite 395]
2. Ein Rahmenwerk bietet einen Applikationsrahmen, der die Ablaufhoheit besitzt. [81, Seite 226]

Daraus ergeben sich drei wesentliche Eigenschaften eines Rahmenwerks, die in [83, Abschnitt 20.1] wie folgt zusammengefasst werden:

Umkehrung des Kontrollflusses: Normale Bibliotheksroutinen werden von einer Anwendung aufgerufen, übernehmen aber niemals den Hauptkontrollfluss der Anwendung, was auch als *Call-Down-Prinzip* bezeichnet wird. Rahmenwerke dagegen legen für die Anwendung die Reihenfolge der Operationen ganz oder zumindest teilweise fest, was dem *Call-Back-Prinzip* entspricht — der Entwickler erweitert das Rahmenwerk um Module (Klassen), die dann vom Rahmenwerk zu gegebener Zeit aufgerufen werden. Diese *Umkehrung des Kontrollflusses* (Inversion of Control, IoC) wird auch *Hollywood-Prinzip* genannt [32, Seite 369]: „Don't call us, we'll call you.“ (Rufen Sie uns nicht an, wir melden uns bei Ihnen.)

Vorgabe einer konkreten Anwendungsarchitektur: Die Architektur sollte zum großen Teil definiert sein und nur an bestimmten Punkten flexibel gehalten werden, wo dann die speziellen Funktionalitäten der jeweiligen Applikation ergänzt werden können.

Anpassbarkeit durch Variationspunkte: Die Variations- oder *Erweiterungspunkte* eines Rahmenwerks sind genau die Stellen, an denen die spezielle Funktionalität einer konkreten Anwendung ergänzt wird. Das Rahmenwerk gibt dabei nur den Typ (die Schnittstelle) der Erweiterung vor und ist somit unabhängig von einer bestimmten Implementierung.

Die Abgrenzung zu den Klassenbibliotheken liegt also darin, dass ein Rahmenwerk nicht zur Code-, sondern zur *Architektur-Wiederverwendung* eingesetzt wird. [32, Seite 37-39] spricht auch von Entwurfswiederverwendung, grenzt dies aber sogleich gegen Entwurfsmuster ab: Entwurfsmuster sind abstrakter, kleiner und weniger spezialisiert als Rahmenwerke.

Es gibt zwei grundlegende Arten von Rahmenwerken, die *objektorientierten* sowie die *komponentenbasierten Rahmenwerke* [83, Abschnitt 20.2]:

- Objektorientierte Rahmenwerke sind über die Struktur ihrer Klassen und deren Verantwortlichkeiten definiert. Eine Nutzung des Rahmenwerks erfolgt über die Spezialisierung abstrakter Oberklassen (bei einem *White-Box-Rahmenwerk*) oder über die Komposition fertiger Klassen (bei einem *Black-Box-Rahmenwerk*). In der Praxis werden beide Varianten oft vermischt.

- Komponentenbasierte Rahmenwerke dagegen legen den Fokus auf die Schnittstellen ihrer Komponenten und definieren deren Interaktion. Die Komponenten sind dabei die Variationspunkte des Rahmenwerks, und die konkreten Implementationen der Komponenten-Schnittstellen werden vom eigentlichen Rahmenwerk verwaltet.

Mischformen beider Arten sind möglich.

Da Abschnitt 3.3 bereits mehrere Komponenten für ConsensusFoundation identifiziert hat, wird das Rahmenwerk zunächst komponentenbasiert entworfen. Zusätzlich ist aber eine Standard-Implementierung der Komponenten erwünscht, von deren Klassen spezialisierte Unterklassen abgeleitet werden können, was einem objektorientierten Rahmenwerk entspricht. ConsensusFoundation wird also eine Mischform aus komponentenbasiertem und objektorientiertem Rahmenwerk werden.

Wie [83, Seite 400-401] und [81, Seite 291] bemerken, kommen innerhalb eines Gesamtsystems mittlerweile häufig mehrere Rahmenwerke zum Einsatz, die dementsprechend nicht mehr alle die uneingeschränkte Ablaufhohheit besitzen können. Daher muss die Interaktion zwischen den Rahmenwerken geregelt werden. Komponentenbasierte Rahmenwerke haben hier einen Vorteil, weil sie die Interaktion zunächst nur beschreiben, nicht aber konkret implementieren, und somit leichter auf eine Kooperation auszulegen sind.

ConsensusFoundation muss dies berücksichtigen, denn es soll — gerade innerhalb von Web-Applikationen — zusammen mit anderen Rahmenwerken eingesetzt werden, die beispielsweise speziell und ausschließlich für die Steuerungsschicht entwickelt worden sind. Das ConsensusFoundation-Rahmenwerk wird daher zunächst passiv darauf warten, dass es den Kontrollfluss übergeben bekommt, um dann seine applikationsspezifischen Erweiterungen innerhalb der Anwendung zu steuern (beispielsweise die Module für spezielle Anreizmechanismen).

3.6 Mehrschichtige Architektur

Auf oberster Ebene der Software-Architektur wird man versuchen, das Gesamtsystem in mehrere Schichten aufzuteilen, um auch hier die Zuständigkeiten klar zu definieren (beispielsweise Darstellung oder Datenhaltung). Innerhalb einer Schicht können dann diverse Komponenten oder ein jeweils passendes Rahmenwerk genutzt werden, wobei die Elemente einer Schicht beliebig miteinander kommunizieren dürfen. Zwischen den Schichten sollte die Kommunikation aber nur über klar definierte Schnittstellen erfolgen. Dies hat wieder dieselben Gründe wie bei Klassen, Komponenten und Rahmenwerken — Modularität, Wartbarkeit, Wiederverwendbarkeit —, nur mit noch höherer Granularität.

Während man bei Desktop-Anwendungen manchmal noch auf eine Zwei-Schichten-Architektur (Anwendungs- und Datenhaltungsschicht) trifft, sollte man bereits bei solchen Anwendungen besser eine

Drei-Schichten-Architektur einsetzen, um in der Anwendungsschicht die Zuständigkeiten für die Darstellung und für das Fachkonzept klar zu trennen [7, Seite 371-373].

Zur Entkopplung von Darstellung und Fachkonzept bietet sich das „Model/View/Controller"-Muster [32, Abschnitt 1.2] an. Die Steuerung (der Controller) nimmt dabei Benutzereingaben entgegen, ruft damit das Fachkonzept (das Model) auf und reicht dessen Rückgabe an die Darstellung (die View) weiter. Die Darstellung ist somit unabhängig vom Fachkonzept und kann (dann meistens zusammen mit der Steuerung) problemlos ausgetauscht werden. Ebenso ist es denkbar, dass einem Model mehrere Controller und Views zugeordnet werden, in denen die Daten des Models unterschiedlich aufbereitet werden.

Bei der Drei-Schichten-Architektur ist die Darstellungsschicht also für zwei Bereiche verantwortlich (View und Controller), was wiederum besser in zwei einzelne Schichten aufgeteilt werden sollte, in eine Darstellungs- und eine Steuerungsschicht (auch Fachkonzept-Zugriffsschicht genannt). Ebenso sollte die Fachkonzept- von der Datenhaltungsschicht entkoppelt werden, damit die Geschäftslogik (das Fachkonzept) nicht von einer bestimmten Implementierung zur Datenhaltung abhängt. Hier wird man daher noch eine Datenhaltungs-Zugriffsschicht einführen. Diese Zugriffsschichten dienen unter anderem dazu, die Datentypen der unteren Schicht in geeignete Datentypen für die darüber liegende Schicht umzuwandeln. Nach [7, Seite 375-376] ergibt sich damit die in Abbildung 14 ersichtliche Mehrschicht-Architektur.

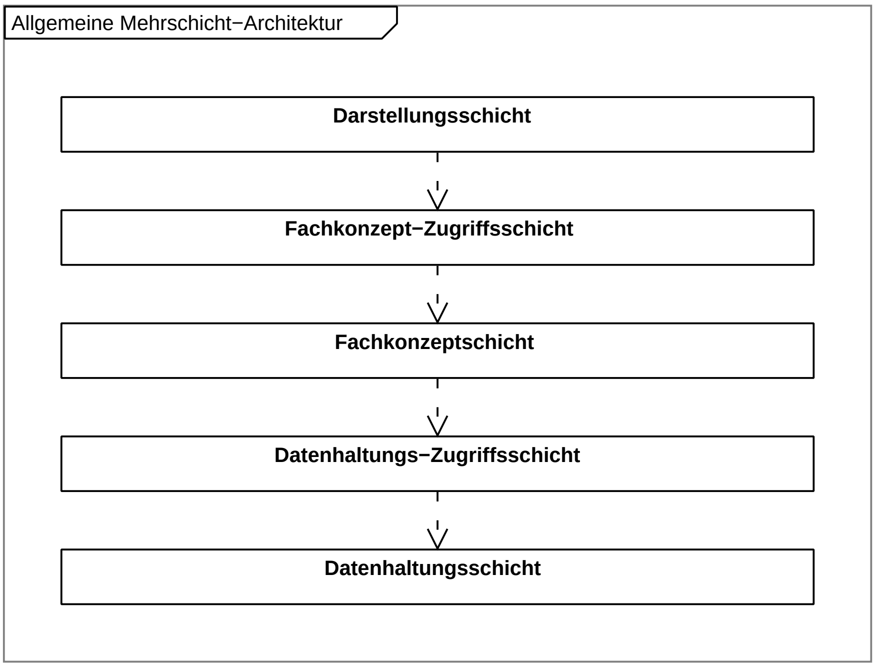


Abbildung 14: Allgemeine Mehrschicht-Architektur nach [7]

ConsensusFoundation soll hauptsächlich in Web-Applikationen eingesetzt werden, daher muss es sich in die dort üblichen Architekturen integrieren lassen. Die Firma Sun beschreibt in [88, Abschnitt 1.3] diverse Anwendungs-Szenarien für J2EE-Applikationen, die dort vorgestellte J2EE-Mehrschicht-Architektur zeigt Abbildung 15. Innerhalb des Web-Containers, der die Darstellungs- und Steuerungsschicht umfasst, wird entsprechend ein Rahmenwerk eingesetzt, das auf Steuerung und Darstellung von Web-Applikationen spezialisiert ist (typische Vertreter sind *Struts* [4] und *JavaServer Faces* [12]). Das ConsensusFoundation-Rahmenwerk wird in der Fachkonzeptschicht genutzt, die hier als EJB-Container bezeichnet ist. Allerdings darf sich ConsensusFoundation nicht auf den Einsatz zusammen mit

Enterprise JavaBeans (EJB) festlegen, denn J2EE-Applikationen lassen sich auch ohne EJB realisieren [56].

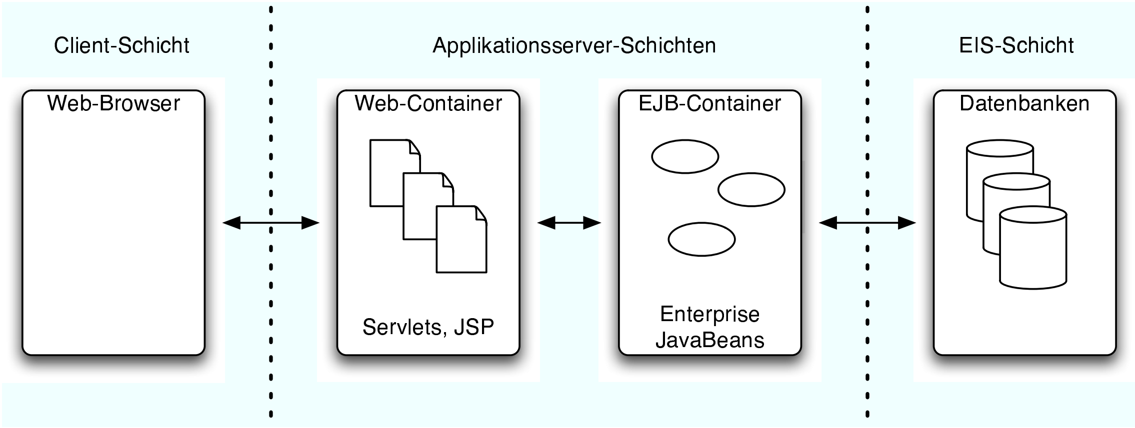


Abbildung 15: J2EE-Mehrschicht-Architektur nach [88]

Das erwähnte „Model/View/Controller"-Muster, das ursprünglich nur mit grafischen Benutzungsoberflächen bei Desktop-Anwendungen genutzt wurde (beispielsweise bei Smalltalk oder bei der Swing-Bibliothek von Java), lässt sich ebenfalls bei Web-Applikationen einsetzen [6, Abschnitt 1.4.6]. Servlets übernehmen dabei die Steuerung (Controller), JSPs die Anzeige (View), und ConsensusFoundation ist wieder Teil der Geschäftslogik (Model). Die meisten Rahmenwerke für die Darstellungs- und Steuerungsschicht realisieren das MVC-Muster oder eine Variante davon.

3.7 Fazit

ConsensusFoundation soll zunächst als komponentenbasiertes Rahmenwerk entworfen werden, d.h. die Schnittstellen und die Interaktionen werden spezifiziert. Zusätzlich soll eine Standard-Implementierung der Schnittstellen zur Verfügung gestellt werden, wobei von den realisierten Klassen durchaus Spezialisierungen gebildet werden können. Es wird also eine Mischform zwischen komponentenbasiertem und objektorientiertem Rahmenwerk entstehen.

Beim Entwurf der Schnittstellen und Klassen sind die in diesem Kapitel gesehenen Richtlinien zu beachten. Wenn möglich, sollen Entwurfsmuster genutzt werden. Beispielsweise bietet es sich an, flexibel austauschbare Module wie die Anreizmechanismen lose gekoppelt als Beobachter an das Rahmenwerk anzubinden, die Konzepte der Ontologie von einer Fabrik erzeugen zu lassen etc.

Das ConsensusFoundation-Rahmenwerk soll ausschließlich in der Fachkonzept-Schicht (im Model) zum Einsatz kommen. Abhängigkeiten zu bestimmten J2EE-Technologien sind zu vermeiden, aber die Anbindung an andere Rahmenwerke (vor allem für die Darstellungs- und Steuerungsschicht) muss möglich sein.

4 Entwurf und Implementierung

Aufbauend auf den Anforderungen der fachlichen Analyse und der Entwurfsrichtlinien wird in diesem Kapitel der Entwurf der Schnittstellen des ConsensusFoundation-Rahmenwerks vorgestellt und diskutiert. Die identifizierten Bereiche werden dabei als separate, austauschbare Komponenten realisiert, die von einer zentralen Komponente verwaltet werden und über die auch der Zugriff auf die anderen Komponenten erfolgt. Während bei der Analyse deutsche Begriffe verwendet wurden, wird bei Entwurf, Implementierung und Quelltext-Dokumentation nun wie üblich Englisch als Sprache zur Benennung von Klassen, Schnittstellen etc. gewählt. Aus der Ontologie-Verwaltung wird somit der OntologyManager, aus der Komponente zur Verwaltung der Bewertungen der RatingManager etc. Die Schreibweise ohne Bindestrich orientiert sich an den gleichnamigen Java-Schnittstellenbezeichnungen (Interfaces).

Abbildung 16 zeigt die Komponenten und deren Beziehungen untereinander im Überblick. Zur besseren Einordnung in ein Gesamtsystem sind die Komponenten dabei bereits auf Knoten verteilt, die bei einer typischen, auf ConsensusFoundation basierenden Web-Applikation anzutreffen sind. Die Entscheidung für die aufgeführten Fremdkomponenten wird in Abschnitt 4.3 erläutert.

Das ConsensusFoundation-Rahmenwerk ist in die folgenden Java-Pakete (Packages) aufgeteilt:

de.uka.ipd.consensus.foundation enthält die zentrale Verwaltung — die Klasse `ConsensusFoundation` — sowie diverse abstrakte Basisimplementierungen, die in den anderen Paketen genutzt werden (siehe Abschnitt 4.4).

de.uka.ipd.consensus.foundation.schema definiert alle Schnittstellen für das Datenmodell (Schema) von ConsensusFoundation (siehe Abschnitt 4.1). Diese Schnittstellen dienen den Komponenten dazu, die Ontologie-Daten zu verarbeiten.

de.uka.ipd.consensus.foundation.ontology ist das Paket für den OntologyManager, der für die Erzeugung und Verwaltung der Ontologie-Daten verantwortlich und entsprechend eng an die Realisierung der Schema-Schnittstellen gekoppelt ist (siehe Abschnitt 4.5).

de.uka.ipd.consensus.foundation.query enthält die QueryEngine, die als Aggregat des OntologyManagers zur Abfrage der Ontologie-Daten verwendet wird (siehe Abschnitt 4.5.1).

de.uka.ipd.consensus.foundation.evolution definiert den EvolutionManager, der für Änderungen an der Ontologie verantwortlich ist, die mehr als ein Konzept betreffen (siehe Abschnitt 4.6).

de.uka.ipd.consensus.foundation.rating definiert die Schnittstellen für das Bewertungssystem, den RatingManager (siehe Abschnitt 4.7). Hierbei geht es um die Abgabe der Bewertungen, nicht um deren Auswertung.

de.uka.ipd.consensus.foundation.scoring ist für das Anreizsystem, den IncentiveManager, verantwortlich. Hier kann die Auswertung der Bewertungen (und der übrigen Benutzer-Aktionen) durchgeführt werden, was eine Vergabe von Punkten als Anreiz zur Folge haben kann (siehe Abschnitt 4.8).

de.uka.ipd.consensus.foundation.rights ist das Paket für die dynamische Rechtevergabe, den DynamicRightsManager (siehe Abschnitt 4.9).

de.uka.ipd.consensus.foundation.user enthält die Schnittstellen für die Benutzerverwaltung, den UserManager (siehe Abschnitt 4.10).

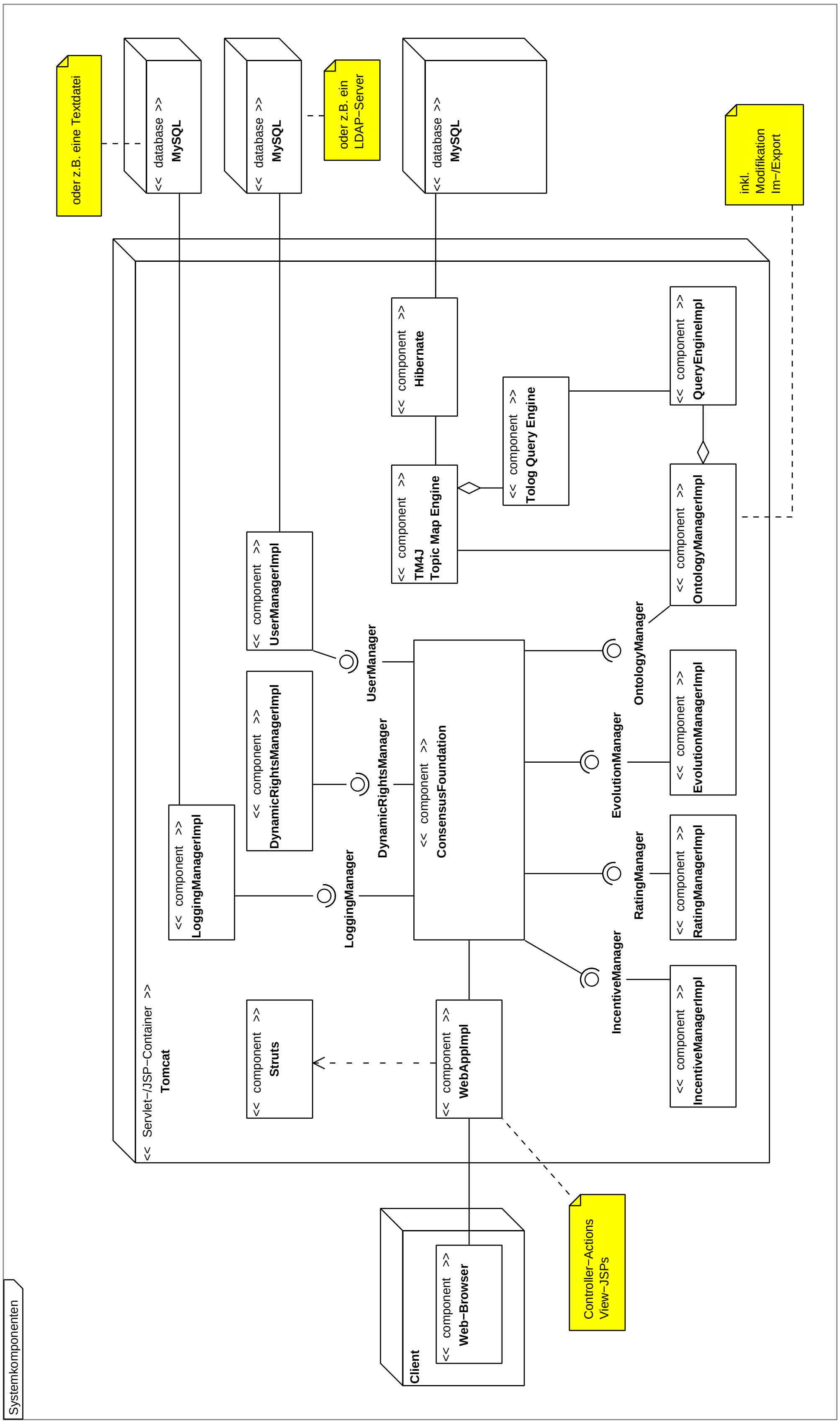


Abbildung 16: Komponenten- und Verteilungsdiagramm von ConsensusFoundation

de.uka.ipd.consensus.foundation.logging schließlich kümmert sich um die Protokollierung der Aktionen im Rahmensystem, wofür nur eine einzige Schnittstelle — der LoggingManager — definiert wird (siehe Abschnitt 4.11).

Die einzelnen Komponenten werden in den angegebenen Abschnitten genauer besprochen.

Die Standard-Implementierung aller Schnittstellen findet sich im Paket `de.uka.ipd.consensus.impl`. Die folgenden Abschnitte gehen bei Bedarf darauf ein. Eine detaillierte Beschreibung der Schnittstellen und Implementierungen erfolgt aber nicht in dieser Arbeit, sondern als Javadoc-Dokumentation. Das Paket `de.uka.ipd.consensus.test` schließlich enthält einige Testklassen. Alle erwähnten Pakete samt ihrer Klassen und Schnittstellen sind im JAR-Archiv `ConsensusFoundation.jar` enthalten, das somit das ConsensusFoundation-Rahmenwerk darstellt — das fertige Produkt dieser Arbeit, mit dem darauf aufbauende Anwendungen realisiert werden können.

Zur Demonstration des Rahmenwerks wurde die Beispiel-Anwendung *ConsensusFoundation Demo* entwickelt, deren Steuerungs- und Darstellungs-Klassen das Eclipse-Projekt (siehe Abschnitt A.1) im Paket `de.snailshell.consensus.demo` (und in Unterpaketen davon) enthält.

4.1 Das Entwurfs-Schema

Das Analyse-Schema (siehe Abschnitt 2.3.2) hat die für das ConsensusFoundation-Rahmenwerk notwendigen Fachklassen identifiziert und die strukturellen Zusammenhänge zwischen ihnen aufgezeigt. Im Entwurfs-Schema werden diese Fachklassen nun so zu Schnittstellen verfeinert, dass sie alle für den praktischen Einsatz relevanten Methoden enthalten, wobei eine Fachklasse durchaus auf mehrere Schnittstellen aufgeteilt werden kann. Einige der Fachklassen können auch sinnvoll zu einer Schnittstelle zusammengefasst werden. Die Schnittstellen des resultierenden Entwurfs-Schemas im Paket `de.uka.ipd.consensus.foundation.schema` zeigt Abbildung 17. Wenn im Weiteren von „Schema“ ohne genauer spezifizierende Angaben die Rede ist, ist damit dann immer dieses Entwurfs-Schema gemeint.

Auffälligster Unterschied zum Analyse-Schema sind die `xxxable`-Schnittstellen, in denen die Methoden zur Beschreibung der Beziehungen zusammengefasst werden, die in derselben Fachklasse enden. Beispielsweise spezialisieren alle Schnittstellen, deren korrespondierende Fachklassen eine Beziehung zur Fachklasse „Bewertung“ besitzen, die Schnittstelle `Rateable` („bewertbar“). Die jeweiligen Methoden müssen dadurch nur einmal deklariert werden und nicht wiederholt in jeder bewertbaren Schnittstelle.

Eine Besonderheit stellen auch `Synonym`, `SuperSubclass` und `TypeInstance` dar. Diese Schnittstellen behandeln spezielle Assoziationen des Rahmenwerks, Typ-Instanz- und Oberklasse-Unterklasse- (siehe Abschnitt 4.2) sowie Synonym-Beziehungen zwischen jeweils zwei Topics. Aus Sicht einer Applikation ist dabei nicht der Zugriff auf die Assoziation selbst interessant, sondern auf die beiden beteiligten Topics und auf die Bewertungen der Assoziation. Entsprechend sind diese Schnittstellen bewertbar und delegieren die Bewertungsmethoden an diejenigen der Assoziation. Weil somit aus Sicht einer an der Bewertung interessierten Anwendung kein Unterschied zwischen der Assoziation und einer dieser Schnittstellen besteht (sie alle erben von `Rateable`), entspricht dieser Entwurf am ehesten dem Entwurfsmuster „Dekorierer“ (Decorator), auch bekannt als „gebundener Umwickler“ (Wrapper) [32]. Da [32, Seite 204] auch von einer *Hülle* spricht, werden diese Schnittstellen im Folgenden auch als *Assoziationshüllen* bezeichnet.

Die übrigen Schnittstellen des Schemas werden nun kurz mit ihren wichtigsten Eigenschaften vorgestellt. Eine detaillierte Beschreibung der einzelnen Methoden findet sich in der Javadoc-Dokumentation.

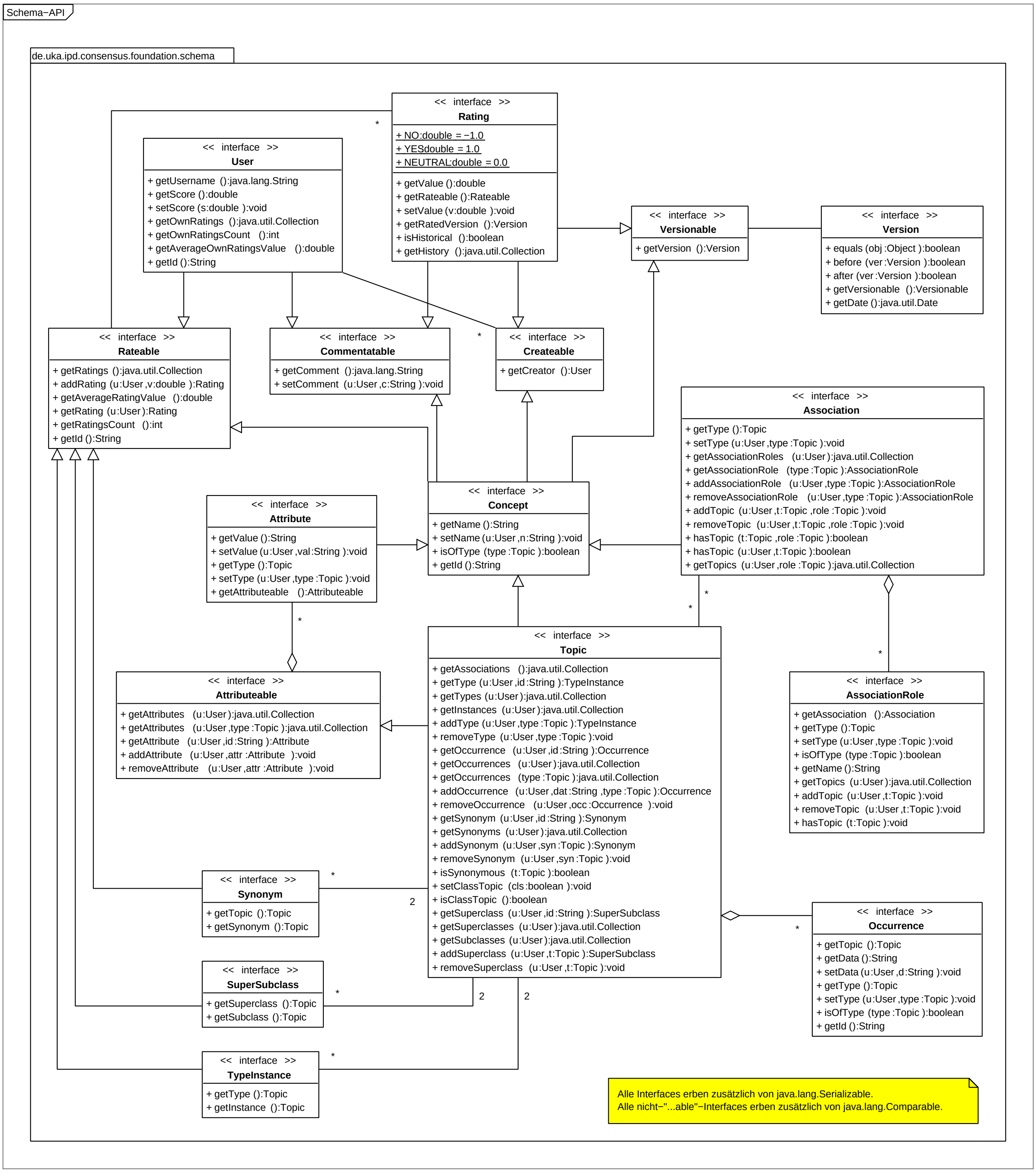


Abbildung 17: Das Paket de.uka.ipd.consensus.foundation.schema

Topic ist das zentrale Konzept einer ConsensusFoundation-Ontologie, die aufgrund der fachlichen Analyse an das Topic-Map-Konzept angelehnt ist, ohne auf zu spezielle Details einzugehen. Als wichtiges Topic-Map-Konzept haben wir in Abschnitt 2.1.3 die *Occurrences* gesehen, die bei ConsensusFoundation über die `Topic`-Schnittstelle zur Verfügung stehen. Ansonsten ist diese Schnittstelle für die Verwaltung von Typen und Instanzen, Ober- und Unterklassen sowie die Synonyme des Topics zuständig.

Association repräsentiert eine Beziehung zwischen beliebig vielen Topics der Ontologie (n-stellige Assoziationen), auch wenn im Normalfall nur zweistellige Beziehungen verwendet werden. Jede Assoziation kann in beliebige *Rollen* unterteilt sein (dargestellt durch die Schnittstelle `AssociationRole`), die jeweils beliebig viele Topics als *Spieler* dieser Rolle enthalten können.

Attribute speichert für ein Topic einen Wert unter einem Namen oder Typ. Jedes Topic verwaltet eine Liste solcher Attribute, in denen Metadaten zum Topic abgelegt sein können. Sie entsprechen der Idee der *Facetten* (facets) mit Werten (facet values).

Version kennzeichnet eine bestimmte Version eines Konzepts oder einer Bewertung. Versionen lassen sich zeitlich ordnen und vergleichen, daher kann man auch das Datum erfragen, an dem die Version angelegt wurde.

User repräsentiert einen von ConsensusFoundation verwalteten Benutzer, der einen eindeutigen Benutzernamen sowie einen Punktestand (Score) besitzt. Neben den Bewertungen, die andere Anwender zu diesem Benutzer abgegeben haben, können über diese Schnittstelle alle Bewertungen (und deren Anzahl und Mittelwert) abgefragt werden, die dieser Benutzer vergeben hat.

Commentatable wird von den Schnittstellen spezialisiert, die eine Zeichenkette als Kommentar verwalten wollen. Wenn man mit ConsensusFoundation eine Anwendung ähnlich Wikipedia [101] realisieren möchte, würde man bei einem Topic in diesem Kommentar den gesamten Text des zugehörigen Artikels ablegen.

Rateable stellt ein bewertbares Element der Ontologie dar. Man kann alle Bewertungen dieses Elements abfragen oder eine einzelne Bewertung, die ein bestimmter Benutzer abgegeben hat. Zudem steht die Anzahl aller Bewertungen und deren arithmetisches Mittel zur Verfügung.

Rating verwaltet eine Bewertung zu einem Konzept oder Benutzer. Der Wert jeder Bewertung liegt im stetigen Intervall $[-1, 1]$ und kann je nach Anwendung in beliebige andere Bereiche bzw. diskrete Werte transformiert werden. ConsensusFoundation schlägt drei diskrete Werte zur Zustimmung, Ablehnung und neutralen Bewertung vor. Neben dem Wert und der Version der Bewertung kann zusätzlich die Version des bewerteten Elements abgefragt werden, wodurch man erkennen kann, ob ein Nutzer eine alte Version des Elements bewertet hat. Außerdem ist die Historie verfügbar, d.h. wie (und wann) ein Benutzer — der Erzeuger dieser Bewertung — die Werte im Laufe der Zeit geändert hat.

4.2 Klassifikation bei ConsensusFoundation

In Abschnitt 2.1.4 wurden zwei Arten der Klassifikation bzw. Typisierung von Topics vorgestellt. Zum einen kann es eine Klassenhierarchie mit speziell ausgezeichneten Klassen-Topics geben, und nur diese Klassen-Topics dürfen Typ eines Instanz-Topics sein. Zum anderen kann ganz auf Klassen verzichtet werden, dann kann jedes Instanz-Topic auch wieder Typ eines anderen Topics sein.

Mit der ConsensusFoundation-Standard-Implementierung lassen sich beide Arten der Klassifikation realisieren. Hierdurch kann zum einen das bewährte Typ-System mit Klassen und Instanzen genutzt werden, zum anderen kann aber auch erforscht werden, wie sich ein freies Typ-Schema auf die Qualität der Ontologie auswirkt. Die Auswahl, ob Klassen als Typen verwendet werden, erfolgt durch den Schlüssel `cfimpl.schema.usesclasses` in der Konfigurationsdatei (siehe Abschnitt A.3).

Mit `Topic.isClassTopic()` ist es möglich abzufragen, ob das Topic als Klassen-Topic erzeugt wurde (siehe Abschnitt 4.5). Ist dies der Fall, können mit `getSuperclasses()` und `getSubclasses()` die Ober- und Unterklassen des Topics ermittelt werden, wodurch eine Applikation die Klassenhierarchie aufbauen kann. Die Wurzeln der Hierarchie liefert `QueryEngine.getClassRoots()`. Die Instanzen eines Klasse-Topics gibt `getInstances()` zurück. Ist das Topic dagegen keine Klasse, liefert `getTypes()` alle Klassen-Topics, die das Instanz-Topic als Typ besitzt. Die Beziehungen zwischen Ober- und Unterklasse bzw. Typ und Instanz (siehe Abbildung 4 links) werden dabei intern mit den *Published Subject Indicators (PSI)* [77, Abschnitt 2.3] „superclass“, „subclass“ und „superclass-subclass relationship“ bzw. „class“, „instance“ und „class-instance relationship“ abgebildet.

Wenn ConsensusFoundation für die Klassifikation ohne Klassen konfiguriert ist (siehe Abbildung 4 rechts), wird die Topic-Hierarchie mit den Topic-Methoden `getTypes()` und `getInstances()` ermittelt. Die Wurzeln der Hierarchie liefert dann `QueryEngine.getInstanceRoots()`, und die übrigen oben vorgestellten Klassen-Methoden haben keine Wirkung. Intern werden die Beziehungen aber trotzdem (und ausschließlich) mit den PSIs „superclass“, „subclass“ und „superclass-subclass relationship“ abgebildet. ConsensusFoundation baut also technisch gesehen eine reine Klassenhierarchie (ohne Instanzen) auf. Dies hat einen einfachen Grund: Die Typ-Instanz-PSIs sind nicht für Hierarchien definiert, und andere Topic-Map-Werkzeuge sollen trotzdem die mit ConsensusFoundation erstellten Ontologien verarbeiten können.

4.3 Technische Rahmenbedingungen

Neben dem Entwurf der Schnittstellen muss bedacht werden, welche fremden Bibliotheken und Produkte — und welche Versionen davon — bei der Implementierung zum Einsatz kommen sollen, da dies Einfluss auf den Entwurf haben kann. Folgende Vorgaben waren gegeben:

- Die verwendeten Werkzeuge und Bibliotheken sollen lizenzfrei nutzbar und idealerweise als Open Source [38, Seite 230 ff.] verfügbar sein.
- Als Persistenzschicht soll *Hibernate* [47] zum Einsatz kommen.
- Die Beispiel-Anwendung soll mit einem gängigen Rahmenwerk für die Darstellungs- und Steuerungsschicht realisiert werden, z.B. mit *Struts* [4].
- Das gesamte Projekt muss sich unabhängig von einer konkreten Entwicklungsumgebung übersetzen lassen, weshalb *Ant* [3] als Build-Werkzeug genutzt werden soll.
- ConsensusFoundation-Anwendungen sollen in praxisrelevanten Java-Server-Umgebungen einsetzbar sein.

Mit diesen Vorgaben fiel die Entscheidung zugunsten der folgenden Software:

- Durch die fachliche Entscheidung, das Java-Rahmenwerk am Topic-Map-Konzept zu orientieren, bietet sich die Bibliothek *Topic Maps for Java (TM4J)* [95] an. Diese Bibliothek besitzt alle Schnittstellen zur Verarbeitung einer Ontologie, auch wenn ConsensusFoundation davon abstrahiert, um einfachere Schnittstellen anzubieten, und fehlende Konzepte wie die dynamische Rechteverwaltung und das Bewertungssystem ergänzen muss. Die TM4J-Implementierung kümmert sich bereits um die nötigen Konsistenzprüfungen innerhalb der Ontologie, kann im XTM-Format [77] im- und exportieren und bietet die SQL-ähnliche Abfragesprache „Tolog“. Eingesetzt wird Version 0.9.8.

TM4J realisiert ebenfalls das *Common Topic Map Application Programming Interface* (TMAPI) [96], eine standardisierte Programmierschnittstelle zur Bearbeitung von Topic Maps. Leider erwies sich die TM4J-Implementierung der TMAPI als zu fehlerbehaftet. Alternative Implementierungen bieten entweder keine passende Persistenzschicht oder sind nur kommerziell verfügbar. ConsensusFoundation nutzt daher die normale TM4J-Programmierschnittstelle.

- Hibernate ist bereits bei TM4J als Persistenzschicht vorgesehen, wenn auch nur in Version 2. Dies wird aber beibehalten, um TM4J nicht anpassen zu müssen.

- Da die *Java Enterprise Edition 5 (Java EE 5)* [53] erst vor kurzem fertiggestellt wurde, setzen die meisten produktiven J2EE-Umgebungen, die neue Versionen generell eher langsam akzeptieren, noch auf J2EE 1.4. Da zudem TM4J mit Java 1.4 entwickelt wurde, passt sich ConsensusFoundation hier an und wird ebenfalls mit Java 1.4 realisiert. Damit entfällt zwar die etwas höhere Typsicherheit durch generische Datentypen (Generics) bei Java 5.0, aber die Server-Basis für den Einsatz ist deutlich höher. Diese Entscheidung betrifft nicht nur die Implementierung des Rahmenwerks, sondern natürlich auch die Schnittstellen, und ist deshalb besonders relevant, weil Quelltexte, die die erweiterte Java 5.0-Syntax nutzen, nicht abwärtskompatibel zu Java 1.4 sind [52].

- Struts wird immer noch häufig bei Web-Anwendungen eingesetzt, auch wenn die Darstellungsschicht Defizite im OO-Konzept aufweist [6, Abschnitt B.2.9]. Aufgrund der weiten Verbreitung wird die Beispiel-Anwendung dennoch mit einer aktuellen Struts-Version (1.2.7 oder neuer) umgesetzt. Die Entwurfsrichtlinien haben bei der Mehrschichtarchitektur bereits gefordert, dass das Rahmenwerk keine Abhängigkeiten zur Darstellungs- und Steuerungsschicht besitzen darf, weshalb hier später problemlos andere Rahmenwerke (beispielsweise *JavaServer Faces* [12]) zum Einsatz kommen können.

4.4 Zentrale Verwaltung — der ConsensusFoundationManager

Das Paket `de.uka.ipd.consensus.foundation` (siehe Abbildung 18) enthält die Klasse `ConsensusFoundation`, welche die Initialisierung des Rahmenwerks übernimmt, sowie Schnittstellen, um einfach auf die Module des Rahmenwerks zugreifen zu können.

`ConsensusFoundation` ist eine der wenigen Klassen im Rahmenwerk, die konkret (d.h. nicht abstrakt) implementiert sind. Sie liest die Konfigurationsdatei ein, lädt die darin deklarierten Klassen dynamisch nach und erzeugt jeweils ein Exemplar davon. Wenn bei einer neuen Web-Applikation nur auf vorhandene Implementierungen zurückgegriffen wird, reicht also folgender Konstruktor-Aufruf aus, um ConsensusFoundation nutzen zu können:

```
ConsensusFoundationManager cfm = new ConsensusFoundation();
```

Durch den parameterlosen Konstruktor wird die Standard-Konfigurationsdatei `ConsensusFoundation.properties` (siehe Abschnitt A.3) als Ressource vom Klassenpfad (Classpath) geladen. Es ist aber möglich, als Parameter den Namen einer beliebigen Datei zu übergeben, die dann ebenfalls vom Java-Klassenlader (Classloader) als Ressource auffindbar sein muss. Bei Web-Anwendungen liegen solche Ressourcen normalerweise im Verzeichnis `/WEB-INF/classes` [23, Abschnitt SRV.9.5].

Die Klasse ist absichtlich nicht als Fabrik (Factory) oder Einzelstück (Singleton) [32] realisiert! Das Rahmenwerk darf nicht festlegen, wieviele Ontologien gleichzeitig verarbeitet werden. Da aber `ConsensusFoundation` für den Normalfall ausgelegt ist, dass zur selben Zeit immer nur auf eine Ontologie zugegriffen wird, entscheidet die jeweilige Anwendung anhand der Anzahl der Exemplare, wieviele Ontologien gleichzeitig genutzt werden können. Jedes Exemplar benötigt eine eigene Konfigurationsdatei, damit jede Ontologie in einem separaten Teil des Hintergrundspeichers abgelegt wird.

Die Beispiel-Anwendung *ConsensusFoundation Demo* erzeugt ein einzelnes Exemplar in einem speziellen Struts-Plugin, das von Struts beim Initialisieren der Web-Applikation ausgeführt wird (siehe Abschnitt A.4). Die Referenz auf dieses Exemplar wird im Servlet-Kontext unter einem bestimmten, implementationsabhängigen Namen abgelegt, unter dem die Klassen der Steuerungsschicht die Referenz später wieder erfragen können (siehe Listing 7).

```
package de.snailshell.consensus.demo;
...
public class ConsensusFoundationPlugin
        implements org.apache.struts.action.PlugIn {
    private String config = ConsensusFoundation.CF_PROPERTIES;
    ...
    public void init(ActionServlet servlet, ModuleConfig config)
                                              throws ServletException {
        try {
            ConsensusFoundation cf =
                new ConsensusFoundation( this.config );
            servlet.getServletContext().setAttribute(
                Constants.CONTEXT_ATTR_CONSENSUSFOUNDATION, cf );
        } catch (Exception e) {
            throw new UnavailableException(
                "Could␣not␣initialize␣ConsensusFoundation\n" + e );
        }
    }
}
```

Listing 7: Struts-Plugin zum Initialisieren von ConsensusFoundation

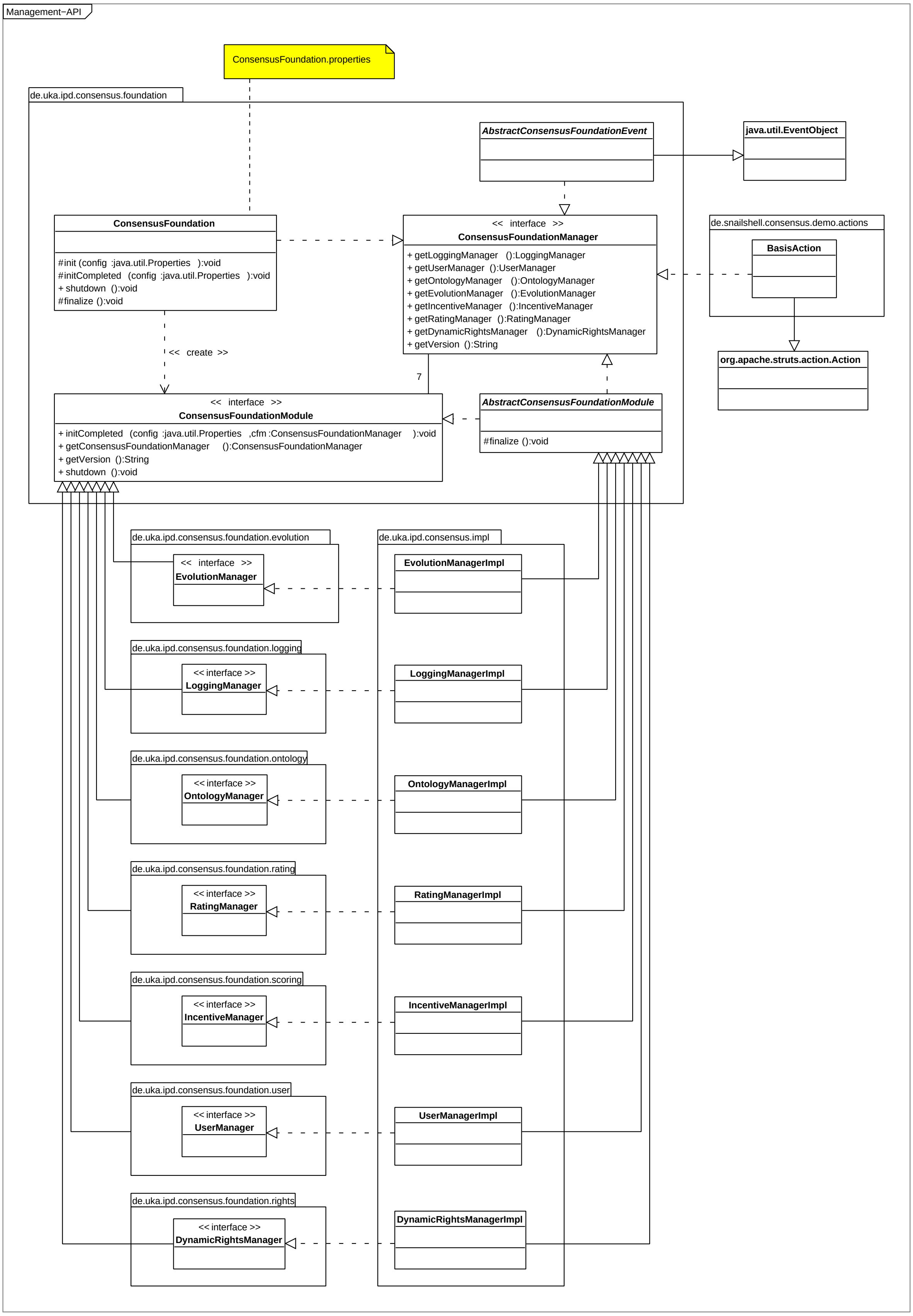


Abbildung 18: Das Paket de.uka.ipd.consensus.foundation

Im Konstruktor werden die Module in der folgenden Reihenfolge geladen und initialisiert:

1. LoggingManager
2. UserManager
3. OntologyManager
4. EvolutionManager (optional)
5. RatingManager (optional)
6. IncentiveManager (optional)
7. DynamicRightsManager (optional)

Die Reihenfolge ist entscheidend, da später initialisierte Module typischerweise auf den vorhergehenden aufbauen. Um die Abhängigkeiten bei der Initialisierung deutlich zu machen, bekommen alle Module an ihre `init()`-Methode die bis dahin garantiert vorhandenen Module als Argumente übergeben. So erhält der UserManager die Referenz auf den LoggingManager, der OntologyManager sowohl LoggingManager als auch UserManager, und alle folgenden Module können auf die ersten drei zwingend vorhandenen Module als `init()`-Parameter zugreifen.

Ob die optionalen Module geladen werden, hängt davon ab, ob für sie vollqualifizierte Klassennamen in der Konfigurationsdatei angegeben sind. Alle dort angegebenen Klassen müssen über den Klassenpfad erreichbar sein. Bei Web-Applikationen befinden sich mindestens das oben erwähnte Verzeichnis `/WEB-INF/classes` für einzelne Klassen sowie das Verzeichnis `/WEB-INF/lib` für JAR-Archive auf dem Klassenpfad.

Nachdem alle vorhandenen Module mit ihrer `init()`-Methode angelegt wurden, wird anschließend die `initCompleted()`-Methode in jedem dieser Module aufgerufen. `initCompleted()` wird von der Schnittstelle `ConsensusFoundationModule` deklariert, die jedes Modul implementieren muss. Die Idee hinter dieser zweistufigen Initialisierung ist einfach: Nachdem die Module grundlegend eingerichtet wurden, erhalten alle Module noch einmal die Gelegenheit, abschließende Einrichtungen vorzunehmen, die von anderen Modulen abhängen. Das bedeutet für jedes Modul, dass nach `init()` alle seine Methoden wohldefinierte Rückgaben liefern müssen. Die wenigen Ausnahmen, bei denen ein Modul dies nicht einhalten kann, sind in der API-Dokumentation entsprechend beschrieben. Genutzt wird `initCompleted()` beispielsweise, um Beobachter (Observer, speziell bei Java auch Listener genannt) [32] an anderen Modulen zu registrieren. So meldet sich der LoggingManager als Beobachter beim RatingManager an, um Änderungen in den Bewertungen protokollieren zu können.

Beim Beenden der Anwendung werden die geladenen Module in umgekehrter Ladereihenfolge wieder freigegeben. Dies geschieht mit `ConsensusFoundation.shutdown()`, welches die `shutdown()`-Methoden in allen aktiven Modulen aufruft. Die Beispiel-Anwendung installiert dazu einen `ServletContextListener`, der vom Servlet-Container beim Herunterfahren der Web-Applikation benachrichtigt wird (siehe Listing 8 und [23, Abschnitt SRV.14.2.12]).

```
package de.snailshell.consensus.demo;
...
public class ConsensusFoundationServletContextListener
        implements ServletContextListener {
    ...
    public void contextDestroyed(ServletContextEvent event) {
        ConsensusFoundation cf = ConsensusFoundationUtils.
```

```
            getConsensusFoundationManager( event.getServletContext() );
        if (cf != null) {
            cf.shutdown();
        }
    }
}
```

Listing 8: ServletContextListener zum Beenden von ConsensusFoundation

Zudem stellt `shutdown()` sicher, dass die Methode nur einmal effektiv ausgeführt wird — weitere Aufrufe haben dann keine Wirkung. Dies ist wichtig, denn `finalize()` [37, Abschnitt 12.6] ruft diese Methode sicherheitshalber auf, falls der explizite — und vorzuziehende[9] — Aufruf von `shutdown()` in der Anwendung vergessen wird.

Die abstrakte Klasse `AbstractConsensusFoundationModule` dient als Oberklasse für alle Module. Auch diese Klasse überschreibt `finalize()` und ruft darin sicherheitshalber die eigene `shutdown()`-Methode auf. Außerdem implementiert die Klasse die Schnittstelle `ConsensusFoundationManager`, mit der jedes Modul einfachen Zugriff auf alle anderen Module hat.

Des Weiteren kann `ConsensusFoundationManager` auch von Klassen außerhalb des Rahmenwerks implementiert werden, gedacht ist dies vor allem für die Steuerungsschicht. Hier ist dies am Beispiel der Klasse `BasisAction` gezeigt, mit der alle Struts-Aktionen der Web-Applikation ebenso einfach alle ConsensusFoundation-Module ansprechen können.

Abschließend sei noch erwähnt, dass es theoretisch möglich ist, `ConsensusFoundation` durch eine eigene Implementierung zu ersetzen und dadurch eine vollständig andere Initialisierung des Rahmenwerks zu realisieren. Normalerweise wird es aber ausreichen, die bestehende Klasse zu vererben und den eigenen Initialisierungscode nach dem Aufruf des Oberklassen-Konstruktors einzufügen. Durch Überschreiben der Methoden `init()` und `initCompleted()` stehen außerdem zwei Einstiegspunkte (Hooks) für die Zeitpunkte unmittelbar vor und nach dem Laden der Module zur Verfügung.

Die letzte Klasse in diesem Paket, `AbstractConsensusFoundationEvent`, ist die Oberklasse aller in diesem Rahmenwerk definierten Ereignis-Klassen. Durch die implementierte `ConsensusFoundationManager`-Schnittstelle besteht auch in den Ereignis-Objekten einfacher Zugriff auf alle Module.

[9] “you should never depend on a finalizer to update critical persistent state ... provide an explicit termination method” [16, Seite 21]

4.5 Verwaltung der Ontologie — der OntologyManager

Der `OntologyManager` (siehe Abbildung 19) ist die Fabrik (Factory) [32] für die Konzepte des ConsensusFoundation-Rahmenwerks. Ein aggregiertes Modul, die QueryEngine, ist für Abfragen der Konzepte zuständig (siehe Abschnitt 4.5.1). Und da der OntologyManager auch für die Persistierung der Daten verantwortlich ist, behandeln weitere Schnittstellen des Pakets `de.uka.ipd.consensus.foundation.ontology` den Im- und Export (siehe Abschnitt 4.5.2) sowie Transaktionen (siehe Abschnitt 4.5.3).

Zum Erzeugen von Topics stehen drei Methoden zur Verfügung:

createTopic() legt ein neues Topic mit einem Namen, einem Erzeuger und optional mit einem Typ, also als Instanz eines anderen Topics, an. Wenn ConsensusFoundation Klassen verwendet (siehe Abschnitt 4.2), muss der Typ ein Klassen-Topic sein. Normalerweise ist dies die Methode der Wahl zum Erzeugen neuer Topics.

createClassTopic() erzeugt ein Klassen-Topic, optional als Unterklasse eines anderen Klassen-Topics. So kann bereits beim Erzeugen eine Klassenhierarchie aufgebaut werden, auch wenn dies später noch über die Topic-Schnittstelle problemlos möglich ist.

createInternalTopic() erzeugt ein Topic, das nicht als normales Topic in der Anwendung sichtbar ist. Eine Applikation kann dies einsetzen, um wenige, spezielle Typen zu verwalten, die nicht durch Anwender modifizierbar sein sollen, beispielsweise eine Oberklasse für alle Assoziationstypen. Die Anwendung muss sich die Kennungen dieser Topics merken, um sie später gezielt abfragen zu können.

`createAssociation()` und `createAttribute()` erzeugen die entsprechenden Konzepte, wobei auch hier optional ein Topic als Typ angegeben werden kann. Die `deleteXXX()`-Methoden versuchen dagegen, ein Konzept aus der Ontologie zu löschen. Wenn dies aber zu einer Inkonsistenz führen würde, weil ein Konzept noch an anderen Stellen referenziert wird, wird das Konzept nicht gelöscht, sondern man erhält eine Ausnahme vom Typ `OntologyManagerException`. Für das Löschen mit Aufrechterhaltung der Konsistenz ist der EvolutionManager zuständig (siehe Abschnitt 4.6), der auf die hier zur Verfügung stehenden Methoden für die einzelnen Schritte des konsistenten Löschens zurückgreifen kann.

Änderungen an der Ontologie, die mit dem OntologyManager oder mit den Schnittstellen des Schemas durchgeführt werden, können in einem Beobachter (Observer, bei Java auch Listener genannt) [32] vom Typ `OntologyListener` erfolgen, der am OntologyManager registriert wird. Den Methoden des Listeners wird ein `OntologyEvent`-Objekt übergeben, welches das von der Änderung betroffene Konzept enthält (und gegebenenfalls auch den alten Wert, wenn ein solcher vorhanden ist). Die Standard-Implementierungen des Incentive- (siehe Abschnitt 4.8) und des LoggingManagers (siehe Abschnitt 4.11) machen hiervon Gebrauch und melden sich selbst als Beobachter an.

Als Kernmodul ist der OntologyManager nicht optional und somit immer vorhanden. Wenn in der Konfigurationsdatei keine andere Klasse mit dem Schlüssel `cf.ontologymanager` deklariert ist, wird die Standard-Implementierung `de.uka.ipd.consensus.impl.OntologyManagerImpl` verwendet.

Normalerweise wird man mit einer persistenten Datenhaltungsschicht („Backend“) arbeiten. Die von ConsensusFoundation verwendete TM4J-Bibliothek sieht dafür Hibernate vor, zusammen mit einem von Hibernate unterstützten Datenbasismanagementsystem (hier MySQL). Um dieses Backend zu verwenden, muss der Konfigurationsschlüssel `cfimpl.ontology.backend` auf den Wert `hibernate` gesetzt werden, ansonsten wird das nicht persistente Memory-Backend verwendet. Letzteres sollte nur zu Testzwecken genutzt werden.

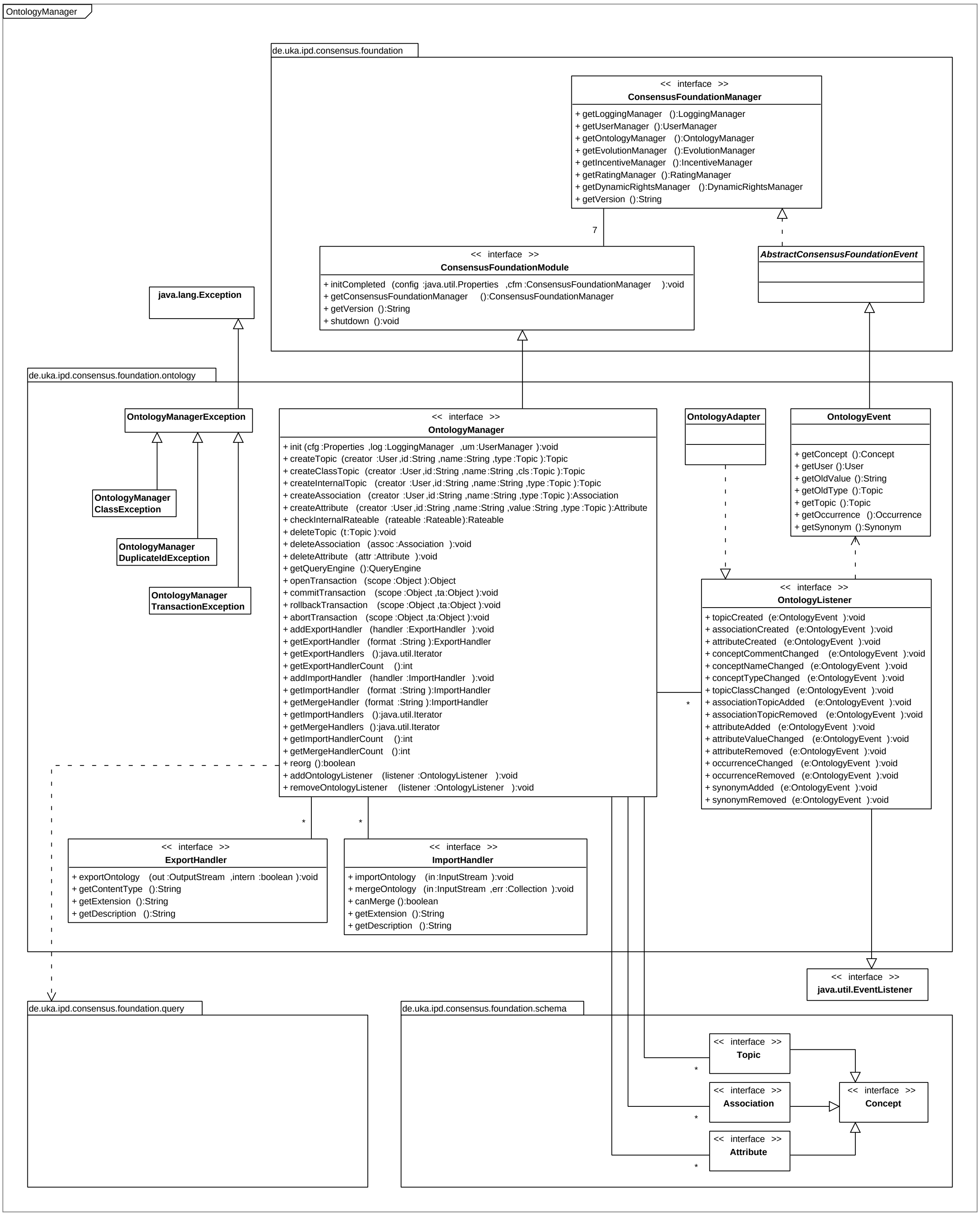


Abbildung 19: Das Paket de.uka.ipd.consensus.foundation.ontology

4.5.1 Abfragen mit der QueryEngine

Die Schnittstelle `de.uka.ipd.consensus.foundation.query.QueryEngine` (siehe Abbildung 20) definiert die Methoden zur Abfrage von Konzepten aus der Ontologie. Die Methoden könnten auch Teil des OntologyManagers sein, da die Abfrage-Routinen Kenntnis der OntologyManager-Implementierung haben müssen. Aus Gründen der Übersichtlichkeit ist die QueryEngine aber als separates Modul realisiert, das man über die OntologyManager-Methode `getQueryEngine()` erhält. Aufgrund der engen Kopplung werden der OntologyManager und die QueryEngine normalerweise zusammen ausgetauscht, aber bei Bedarf kann die QueryEngine separat in der Konfigurationsdatei mit dem Schlüssel `cf.queryengine` ausgewechselt werden. Ist der Schüssel nicht definiert, wird die Standard-Implementierung `de.uka.ipd.consensus.impl.QueryEngineImpl` verwendet.

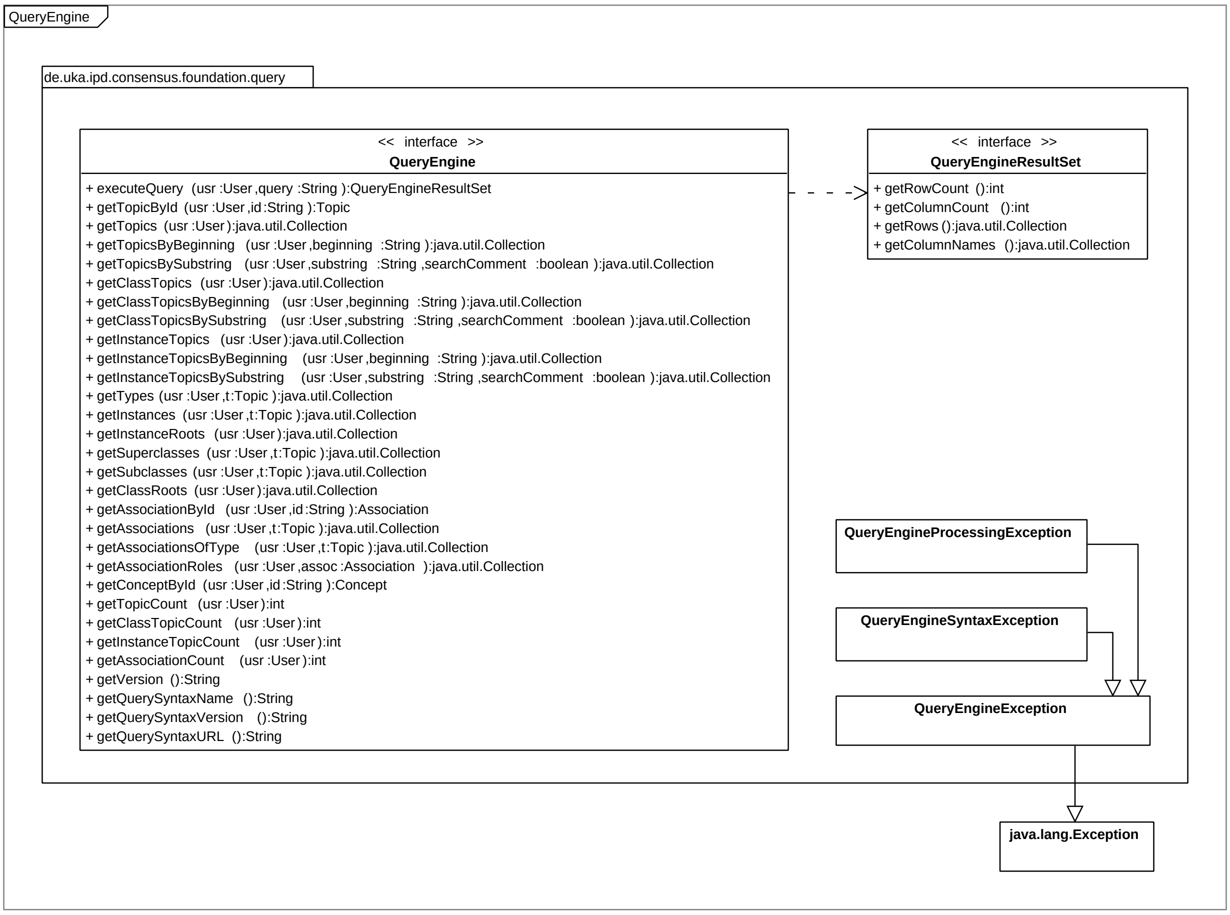


Abbildung 20: Das Paket de.uka.ipd.consensus.foundation.query

Topics können anhand ihrer Kennung (ID), der Anfangsbuchstaben ihres Namens sowie einer Teilzeichenkette gefunden werden. Bei Letzterem ist außerdem möglich, nicht nur im Namen, sondern auch in der Beschreibung, d.h. im Kommentar, suchen zu lassen. Assoziationen können ebenfalls anhand der Kennung gesucht werden. Zusätzlich können alle Assoziationen, in denen ein gegebenes Topic eine Rolle spielt, und alle Assoziationen eines gegebenen Typs ermittelt werden. Attribute werden von der QueryEngine nicht behandelt, da sie nur innerhalb eines Topics genutzt werden, wo sie auch abgefragt werden können. Mit weiteren Methoden ist es möglich, spezielle Assoziationen wie Typ-Instanz-Beziehungen und Oberklassen-Unterklassen-Beziehungen auszuwerten (diese Methoden werden von den entsprechenden Topic-Schnittstellen als Delegationsmethoden [32] genutzt). Für Applikationen ist außerdem wichtig, alle Klassen-Topics ohne Oberklassen (`getClassRoots()`) und alle Instanz-Topics ohne Typ (`getInstanceRoots()`) abfragen zu können, um Wurzel-Topics zur Anzeige der entsprechenden Hierarchien zu haben. Die Rückgabe der letztgenannten Methode ist abhängig davon, ob ConsensusFoundation für die Nutzung von Klassen-Topics konfiguriert wurde. Dies wird in Abschnitt 4.2 genauer beschrieben. Alle diese Abfragen beachten automatisch den DynamicRightsManager (siehe Abschnitt 4.9).

Außerdem erlaubt ConsensusFoundation die Abfrage von Konzepten über eine frei formulierte Abfrage-Zeichenkette. Das Rahmenwerk definiert dafür keine eigene Abfragesprache, sondern reicht die Abfragen direkt an eine Fremdbibliothek weiter. Bei der Standard-Implementierung bringt TM4J bereits die Abfragesprache „Tolog“ mit, die von ConsensusFoundation angesprochen wird. Damit der Anwender bzw. die Applikation weiß, in welcher Syntax die Anfrage gestellt werden muss, lässt sich die verwendete Sprache (Syntax, URL der Online-Dokumentation) ermitteln und beispielsweise in der Anwendung anzeigen. Die von TM4J realisierte Version 1.0 von Tolog ist in [33] spezifiziert. Auf [97] findet sich eine allgemeine Einführung in die neuere Version 1.1, die aber problemlos mit TM4J genutzt werden kann. Listing 9 zeigt eine beispielhafte Tolog-Anfrage, mit der alle Assoziationen ermittelt werden, die Rollen und Spieler besitzen.

```
select $A, $R, $P
from association($A), association-role($A, $R), role-player($R, $P)
```

Listing 9: Tolog-Abfrage für alle Assoziationen mit Rollen und Spielern

Die Ausführung einer solchen Abfrage erfolgt mit `executeQuery()`, wobei die Standard-Implementation des DynamicRightsManagers derartige Abfragen nur Administratoren gestattet. Als Ergebnis erhält man ein `QueryEngineResultSet`, mit dem man einfach über alle Zeilen und Spalten des Ergebnisses iterieren kann. Die Standard-Implementation dieser Schnittstelle ist „träge“ (lazy) realisiert, wodurch erst dann die Werte von TM4J abgefragt werden, wenn sie von der Anwendung benötigt werden. Trotz des Namens kann dadurch ein Performanzgewinn erreicht werden, falls die Applikation nicht alle Daten der Abfrage wirklich nutzt. Obwohl die Tolog-Abfrage natürlich TM4J-Objekte zurückliefert, versucht das Rahmenwerk, diese so weit wie möglich in ConsensusFoundation-Objekte umzuwandeln (aus einer TM4J-Assoziation wird eine ConsensusFoundation-Assoziation etc.). Nur wenn das Rahmenwerk kein analoges Objekt anbieten kann, wird das Original-Objekt zurückgegeben. Das Ergebnis einer Tolog-Abfrage in der Beispiel-Anwendung zeigt Abbildung 21.

4.5.2 Im- und Export

Mit `ImportHandler` und `ExportHandler` stehen zwei Schnittstellen zur Verfügung, um austauschbare Module zum Import und Export der Ontologie zu realisieren. Die Anwendung kann entweder selbst Exemplare der Module erzeugen und mit `addImportHandler()` bzw. `addExportHandler()` beim OntologyManager registrieren. Einfacher und flexibler ist es jedoch, die Klassennamen aller gewünschten Module in der Konfigurationsdatei als Werte der Schlüssel `cf.importhandler` bzw. `cf.exporthandler` zu deklarieren (siehe Abschnitt A.3). Es wird von allen dort aufgeführten Klassen vom OntologyManager automatisch ein Objekt erzeugt und kann anschließend als entsprechendes Modul abgefragt und genutzt werden.

Eine Anwendung kann entweder alle Module abfragen, um dem Benutzer eine Liste aller Im- und Exportformate anzuzeigen, oder aber gezielt ein Modul anhand der Dateinamenserweiterung des gewünschten Formats. Standardmäßig wird von ConsensusFoundation das XTM-Format [77] durch die Klassen `de.uka.ipd.consensus.impl.XTMImportHandler` und `XTMExportHandler` unterstützt.

`importOntology()` wird vom OntologyManager beim Start verwendet, um eine externe Ontologie zu laden — eine eventuell vorhandene Datenbasis wird dabei ersetzt! Die Standard-Implementierung

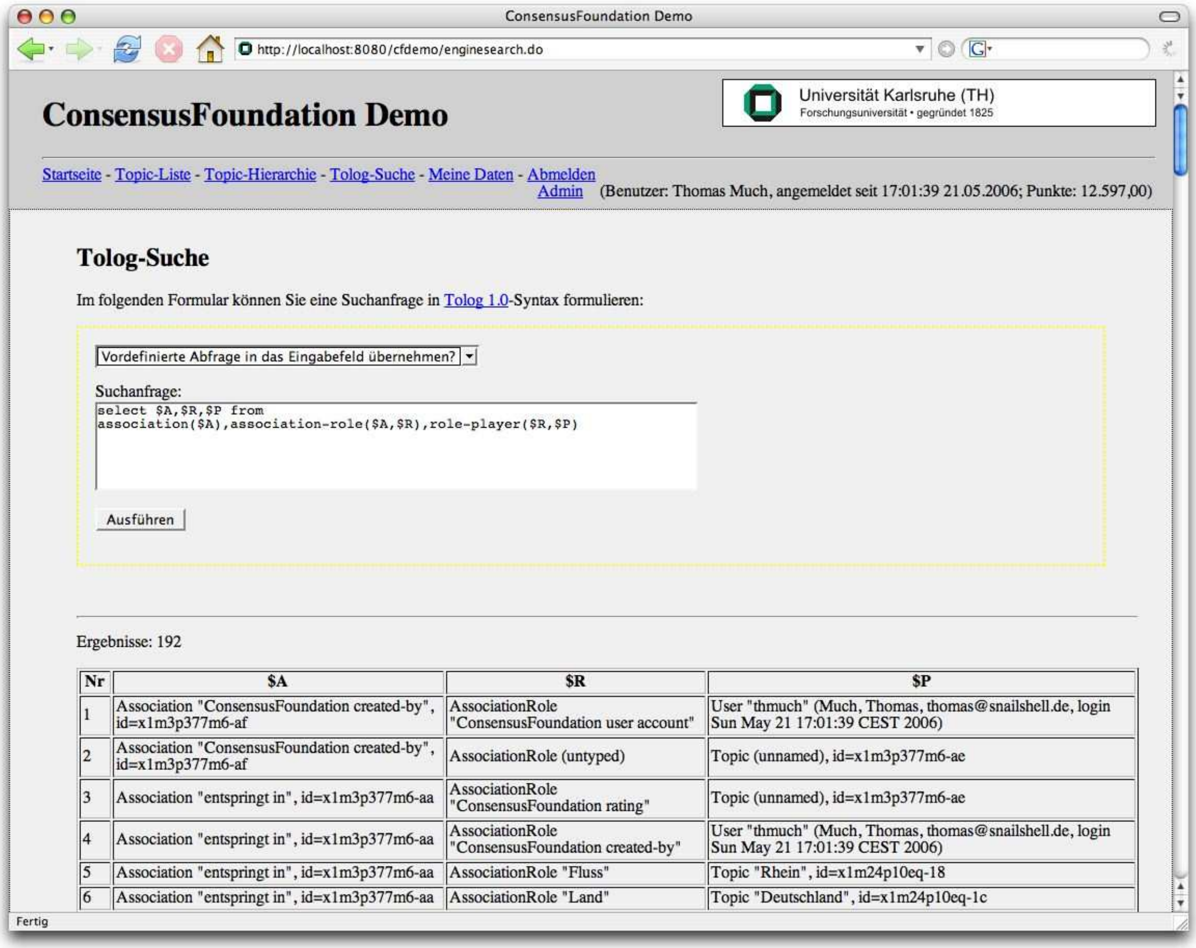


Abbildung 21: Tolog-Abfrage in der ConsensusFoundation-Beispielanwendung

wertet dafür in der Konfigurationsdatei den Schlüssel `cfimpl.ontology.file` aus, dessen Wert die zu importierende Datei angibt. Die Datei muss als Ressource über den Klassenpfad geladen werden können. Wichtig ist dies vor allem für Testzwecke, wenn nach dem Start immer eine definierte Datenbasis vorhanden sein soll.

Mit `canMerge()` kann ein ImportHandler signalisieren, ob er eine Ontologie zur aktuellen hinzuladen kann. Dabei wird allerdings nur ein technisches Zusammenführen vorgenommen, ein fachliches Verschmelzen von Konzepten wird vom EvolutionManager (siehe Abschnitt 4.6) behandelt. Das technische Zusammenführen erkennt beispielsweise doppelt vergebene Kennungen (IDs) als Fehler, beachtet aber die Semantik der Konzepte nicht. Weil das Laden (Import) und das technische Zusammenführen (Merge) sehr ähnlich ablaufen, wird beides über die Import-Schnittstelle abgewickelt.

`exportOntology()` ist entsprechend für den Export der Ontologie zuständig, wobei angegeben werden kann, ob interne ConsensusFoundation-Konzepte (beispielsweise Benutzer-Topics, Versionsinformationen oder — falls sie in der Ontologie gespeichert würden — Bewertungen) mit exportiert werden sollen. Dadurch kann man eine schlanke, auf die wesentlichen Konzepte reduzierte Ontologie erzeugen, die sich im Falle des XTM-Exports mit anderen Topic-Map-Werkzeugen besser verarbeiten lässt.[10]

Sowohl beim Import als auch beim Export muss vom Aufrufer ein offener Datenstrom (`java.io.InputStream` bzw. `OutputStream`) übergeben werden. Die Schnittstellen des Rahmenwerks zwingen den Im-

[10] Das Ausfiltern der internen Konzepte erledigt die Klasse `de.uka.ipd.consensus.impl.XTMExportInternalFilter`.

plementierungen der Module dadurch keine bestimmte Form der Speicherung auf (Text- oder Binärdaten, lokale Datei oder Netzwerkressource etc.).[11]

4.5.3 Transaktionen

Transaktionen (TA), die eine Folge von Operationen zu einer logischen Einheit zusammenfassen und dabei gewisse Eigenschaften garantieren, betreffen nicht nur Datenbasismanagementsysteme, sondern generell Anwendungen, bei denen mehrere Benutzer gleichzeitig auf einer gemeinsamen Datenbasis arbeiten, im hier relevanten Fall insbesondere (verteilte) Web-Applikationen: „Die Bedeutung des Transaktionskonzepts geht jedoch weit über den Einsatz im Rahmen von DBS hinaus. Es stellt ein zentrales Paradigma der Informatik dar, das zur sicheren Verwendung unterschiedlichster Betriebsmittel eingesetzt werden kann. [...] Das Transaktionskonzept ist somit auch von zentraler Bedeutung für die sichere Abwicklung von Geschäftsvorgängen im Internet (Electronic Commerce) [...].“ [41, Seite 391]

ConsensusFoundation, dessen Anwendungen vermutlich häufig Web-Applikationen sein werden (entweder eigenständig oder als Teilsystem eines größeren Gesamtsystems), muss bei der Sicherung seiner Daten entsprechend auch die sogenannten ACID-Eigenschaften für eine Transaktion garantieren können [41, Abschnitt 13.1]:

Atomarität (Atomicity): Die Transaktion soll aus Anwendersicht unteilbar verlaufen, so dass sie entweder ganz oder gar nicht ausgeführt wird.

Konsistenz (Consistency): Nach Abschluss der Transaktion muss sich die Datenbasis wieder in einem konsistenten (widerspruchsfreiem) Zustand befinden.

Isolation: Wenn mehrere Transaktionen gleichzeitig laufen, beispielsweise beim Mehrbenutzerbetrieb, müssen die parallelen Zugriffe anderer Benutzer unsichtbar bleiben. Es darf also keine unerwünschten Nebenwirkungen geben.

Dauerhaftigkeit (Durability): Die Dauerhaftigkeit bzw. Persistenz erfolgreicher Transaktionen wird garantiert, d.h. auch nach zukünftigen Fehlern muss das Ergebnis der Transaktion Bestand haben.

Das Rahmenwerk realisiert allerdings keine eigene Transaktionsverwaltung, sondern verlässt sich auf die Routinen der eingesetzten Persistenzschicht. Zur Entkopplung vom konkreten Persistenz-Modul stellt ConsensusFoundation aber eine abstrakte Schnittstelle zur Steuerung der Transaktionen zur Verfügung. Diese enthält, wie bei Transaktionen üblich, Methoden zum Beginn einer neuen Transaktion (Begin of Transaction, BOT), zur erfolgreichen Beendigung (Commit) sowie zum Abbruch einer Transaktion (Rollback) [41, Abschnitt 13.2].

Die wenigen Methoden zur Transaktionssteuerung sind im OntologyManager angesiedelt und nicht in eine separate Komponente ausgelagert. Dies ist sinnvoll, weil eine Anwendung damit an der Komponente die Transaktionen steuert, die sich intern auch um die Persistenz kümmert. Außerdem wird man beispielsweise bei verteilten J2EE-Anwendung gar keine lokale Transaktionssteuerung mit den ConsensusFoundation-Methoden durchführen, sondern übergeordnete Transaktionsmechanismen verwenden, beispielsweise deklarativ mit den *Container-Managed Transactions (CMT)* bei EJB oder programmatisch mit der *Java Transaction API (JTA)* [55, Seite 23 und 210-213].

[11] Wie in der Beispiel-Anwendung Dateien über einen Web-Browser hoch- bzw. heruntergeladen werden, kann in den Klassen `de.snailshell.consensus.demo.actions.MergeAction` und `ExportAction` eingesehen werden.

Die Standard-Implementierung reicht die Aufrufe von `openTransaction()` etc. direkt an die entsprechenden TM4J-Methoden weiter. Beim Hibernate-Backend beginnt und beendet TM4J eine Transaktion mit derselben Granularität wie die intern ebenfalls benötigte Hibernate-Session (der Gültigkeitsbereich für die Identitäten von Objekten, die mit Hibernate abgefragt wurden). Dadurch kann eine Anwendung leicht eine Transaktion pro Anfrage (Request) realisieren, was in [9, Abschnitt 5.2.2] auch als „session-per-request" bezeichnet wird.[12] Für die dort ebenfalls erwähnten Applikations-Transaktionen sollten die oben beschriebenen übergeordneten Transaktionsmechanismen zum Einsatz kommen, wobei die ConsensusFoundation-Schnittstelle schon darauf vorbereitet ist, ein beliebiges Objekt zur Kennzeichnung eines beliebigen Gültigkeitsbereichs einer Transaktion zu verarbeiten. Beim Einsatz des Memory-Backends sollten Sie beachten, dass dieses nicht transaktionsfähig ist, die Methodenaufrufe bleiben hier wirkungslos.

[12] Die Beispiel-Anwendung setzt dies in dem Servlet-Filter [23, Abschnitt SRV.6] `de.snailshell.consensus.demo.ConsensusFoundationFilter` um.

4.6 Komplexe Änderungen — der EvolutionManager

Die meisten Änderungen der Ontologie werden über die Schnittstellen des Schemas oder des OntologyManagers durchgeführt. Dabei handelt es sich um lokale oder konstruktive Änderungen, d.h. entweder ist nur ein Konzept davon betroffen (z.B. wenn das Attribut eines Topics oder der Kommentar eines Konzepts geändert wird) oder ein Konzept wird neu zur Ontologie hinzugefügt, so dass die bestehenden Konzepte gar nicht davon betroffen sind.

Manche Veränderungen sind aber destruktiv, entfernen also Informationen aus der Ontologie, oder betreffen mehrere Konzepte, wodurch die Änderungsoperation aufwändiger wird oder vielleicht gar nicht fehlerfrei durchgeführt werden kann. Solche Anwendungsfälle fasst die Schnittstelle `EvolutionManager` im Paket `de.uka.ipd.consensus.foundation.evolution` zusammen (siehe Abbildung 22). Von der Funktionalität her könnte der EvolutionManager Teil des OntologyManagers sein, aber aus Gründen der Übersichtlichkeit und weil die erwähnten komplexen Änderungen erwarten lassen, dass es verschiedene Implementierungen geben wird, die zur Erprobung ausgetauscht werden sollen, sind die hier besprochenen Methoden in einer eigenen, separaten Komponente zusammengefasst.

Es gibt zahlreiche Anwendungsfälle, wie eine Ontologie verändert werden kann. [59] erwähnt über 80 grundlegende Änderungsoperationen (z.B. das Ändern oder Hinzufügen einer Oberklasse), die — sofern sie für ConsensusFoundation relevant sind — von den Schema- bzw. OntologyManager-Schnittstellen abgedeckt werden, sowie zahlreiche darauf aufbauende komplexe Operationen (beispielsweise das Verschmelzen von Klassen oder das Verschieben eines Teilbaums der Klassenhierarchie). Der EvolutionManager bietet derzeit nur für einen Teil der denkbaren komplexen Operationen Methoden an, vor allem für diejenigen, die sich unmittelbar aus anderen Schnittstellen des Rahmenwerks selbst ergeben. In Zukunft ist sicherlich mit einer Ergänzung der Schnittstelle zu rechnen (am besten als Spezialisierung der existierenden), wenn sich in der Praxis weitere Methoden als sinnvoll bzw. wünschenswert herausstellen.

Die Methoden des EvolutionManagers können von einem Benutzer explizit aufgerufen werden, falls die Anwendung dies erlaubt. Alternativ könnte die Applikation einen Beobachter am Bewertungssystem (siehe Abschnitt 4.7) registrieren, der automatisch beispielsweise Konzepte löscht, wenn ausreichend viele negative Bewertungen vorhanden sind, oder Topics zusammenführt, wenn ein Synonym ausreichend oft positiv bewertet wurde.

Im Folgenden werden die Methoden kurz vorgestellt, und es werden jeweils die Fragen aufgeworfen, die bei der Implementierung zu beachten sind. Wenn Klassen-Topics verwendet werden sollen, muss sichergestellt sein, dass ConsensusFoundation überhaupt für die Verwendung von Klassen konfiguriert ist (siehe Abschnitt 4.2).

deleteTopic() löscht ein einzelnes Topic. Handelt es sich um ein Instanz-Topic, muss dieses aus den Assoziationen entfernt werden, in denen es eine Rolle spielt. Falls eine Assoziation nicht mit der verminderten Spielerzahl existieren darf, muss gegebenenfalls auch die Assoziation gelöscht werden.

Handelt es sich dagegen um ein Klassen-Topic, sollten normalerweise auch dessen Instanzen gelöscht werden. Unterklassen verbleiben bei dieser Methode aber in der Ontologie — hier ist zu klären, wie diese neu in der Hierarchie verankert werden, beispielsweise könnten sie alle Oberklassen des zu löschenden Topics übernehmen. Zu klären ist außerdem, was mit Konzepten passiert, die dieses Topic als Typ verwenden: Werden die Konzepte gelöscht, oder erhalten sie als Typ eine Oberklasse des zu löschenden Topics?

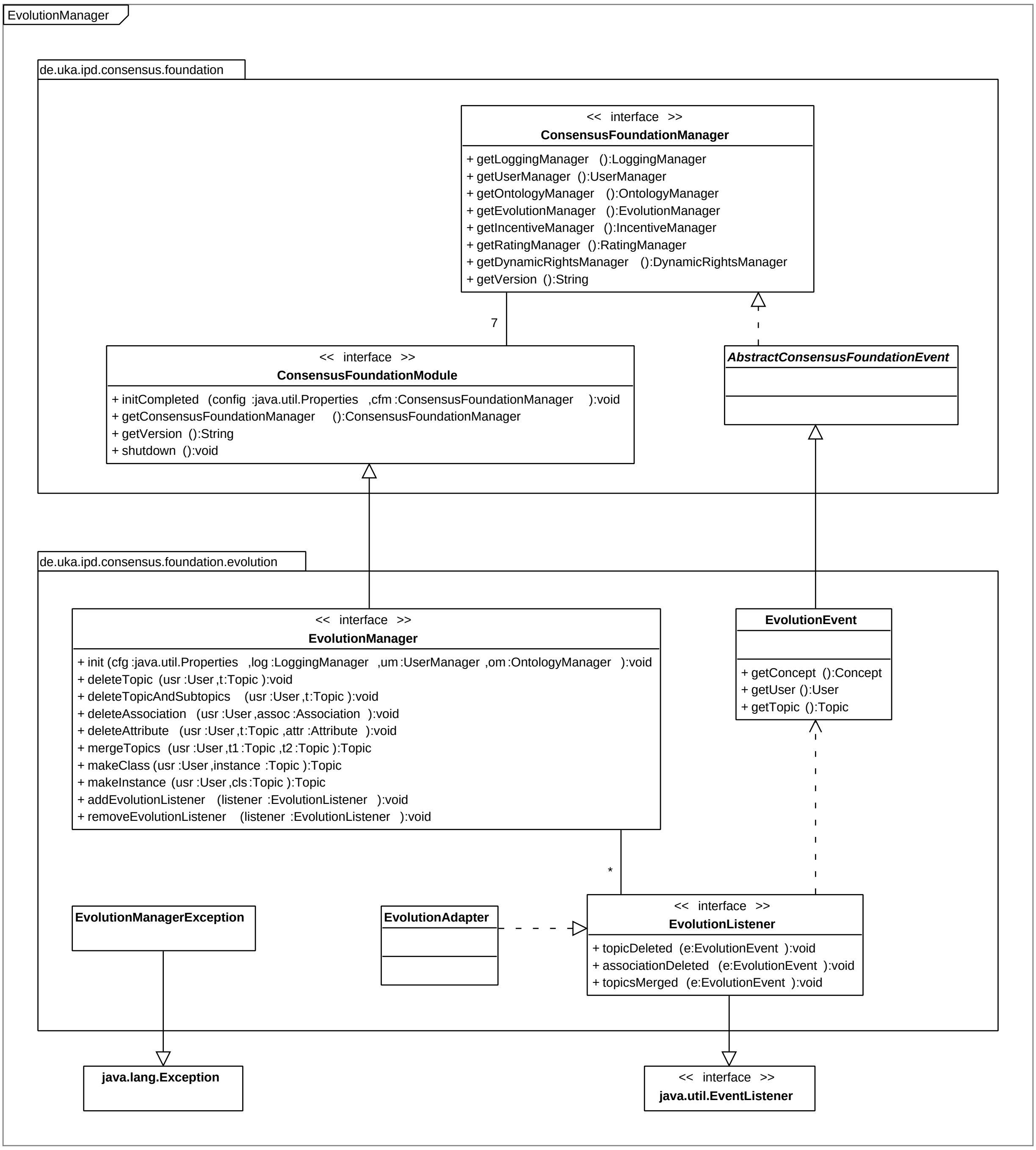


Abbildung 22: Das Paket de.uka.ipd.consensus.foundation.evolution

deleteTopicAndSubtopics() funktioniert prinzipiell wie `deleteTopic()`, aber bei einem Klassen-Topic werden auch die Unterklassen (und jeweils deren Instanzen) gelöscht. Problematisch wird dies in dem Fall, wenn die Klassenhierarchie Zyklen enthält. Die Implementation muss sicherstellen, dass trotz der Zyklen nicht plötzlich große Teile der Klassenhierarchie gelöscht werden. Erlaubt sind solche zyklischen Unterklassen-Beziehungen, weil eventuell erst durch Bewertungen geklärt werden kann, welche Hierarchie-Beziehungen korrekt sind. Die Implementation sollte beim Löschen also die Bewertungen beachten.

deleteAssociation() löscht normalerweise nur die angegebene Assoziation. Es könnte aber Assozationen geben, bei denen auch die Spieler-Topics mitgelöscht werden, wenn sie sonst nirgendwo mehr referenziert werden.

deleteAttribute() ist unkritisch, wenn das zu löschende Attribut zu einem Instanz-Topic gehört, da die Änderung nur lokal dieses Topic betrifft. Bei einem Attribut eines Klassen-Topics muss aber geprüft werden, ob gleichnamige Attribute (bzw. Attribute vom gleichen Typ) in den Instanzen ebenfalls gelöscht werden sollen.

mergeTopics() soll zwei Topics verschmelzen. Die Frage ist, was mit Assoziationen und Attributen passiert, falls es Überschneidungen bei deren Typen gibt, aber die enthaltenen Werte unterschiedlich sind — im Zweifel müssen alle Werte gespeichert und zur Bewertung angeboten werden. Bei Instanz-Topics läuft dies ansonsten darauf hinaus, als Typ der neuen Instanz die Vereinigung der bisherigen Typen zu verwenden. Bei Klassen-Topics ist das Zusammenführen kritischer, da hier gegebenenfalls auch Unterklassen und deren Instanzen betroffen sind, die überprüft und eventuell angepasst werden müssen.

Insgesamt können beim Verschmelzen vielfältige Probleme auftreten, die bereits an anderer Stelle untersucht wurden. So stellt [72] „PROMPT“ vor, einen Algorithmus zur halbautomatischen Fusion von Ontologien. Die dabei gefundenen Schwierigkeiten dürften auch dann relevant sein, wenn die Entscheidung zur Lösung eines Konflikts nicht durch einen Anwender (ob mit oder ohne Expertenwissen) getroffen wird, sondern vollautomatisch durch die Software aufgrund der abgegebenen Bewertungen.

makeClass() wandelt ein Instanz-Topic in ein Klassen-Topic um, beispielsweise weil die Bewertung des Topics ergeben hat, dass es sich eher um einen eigenen Typ als um eine Instanz einer anderen Klasse handelt.

makeInstance() versucht umgekehrt, aus einem Klassen-Topic ein Instanz-Topic zu machen. Dies ist schwieriger, denn es stellt sich die Frage, was mit eventuell vorhandenen Instanzen der Klasse passiert. Werden diese gelöscht oder weiter oben in der Hierarchie (bei einer der oder allen Oberklassen des zu löschenden Topics) eingehängt?

Die Standard-Implementierung `de.uka.ipd.consensus.impl.EvolutionManagerImpl` realisiert die besprochenen Methoden zum Teil nur trivial und lässt einige der oben gestellten Fragen außer Acht. Andere Arbeiten sind derzeit in Vorbereitung, aus denen eine nicht-triviale Implementation des EvolutionManagers hervorgehen soll.[13] Der verwendete EvolutionManager muss in der Konfigurationsdatei mit dem Schlüssel `cf.evolutionmanager` explizit konfiguriert werden (siehe Abschnitt A.3), da es sich um eine optionale Komponente handelt.

Wenn es darum geht, ein Änderungsprotokoll zu erstellen oder eine andere der in [59] beschriebenen Repräsentationen für Ontologie-Änderungen zu verwenden, ist der EvolutionManager dafür nicht der richtige Ort, da hier nur ein Teil aller Ontologie-Änderungen durchgeführt wird. Stattdessen würde man einen spezialisierten LoggingManager (siehe Abschnitt 4.11) einsetzen, der sich als Beobachter sowohl am Ontology- als auch am EvolutionManager anmeldet und dadurch die gemeldeten Änderungen passend protokollieren kann.

[13] Falls eine Implementierung eine bestimmte Methode überhaupt nicht realisieren kann, sollte eine `java.lang.UnsupportedOperationException` geworfen werden. Dies ist eine Unterklasse von `java.lang.RuntimeException` und taucht daher nicht in der Signatur der Methoden auf.

4.7 Bewertungsvergabe — der RatingManager

Das durch die Schnittstelle `de.uka.ipd.consensus.foundation.rating.RatingManager` definierte Bewertungssystem ist für einen technischen Aspekt verantwortlich, die Erzeugung, Manipulation, Speicherung und Abfrage von `Rating`-Objekten, wofür diverse Methoden angeboten werden (siehe Abbildung 23). Diese Komponente behandelt dagegen explizit nicht die Punktevergabe, die nach dem Abgeben einer Bewertung als Belohnung erfolgen kann, oder die Auswertung der Punkte als Grundlage für andere Anreize — dies beides ist Sache des IncentiveManagers (siehe Abschnitt 4.8).

Der RatingManager dient also als Fabrik (Factory) [32] für Bewertungen, die von anderen Klassen des Rahmensystems genutzt wird. Eine konkrete Applikation greift aber in aller Regel nicht direkt auf den RatingManager zu, sondern verwendet die entsprechenden Methoden des Schemas (siehe Abschnitt 4.1). `Topic`, `Association`, `Attribute` und `User` definieren die Schnittstellen für die bewertbaren Elemente des ConsensusFoundation-Rahmenwerks, und die Methoden ihrer Implementierungen müssen die Aufrufe an den RatingManager (das Delegationsobjekt [32]) delegieren, sofern ein solcher überhaupt vorhanden ist (siehe Listing 10). Da es sich um eine optionale Komponente handelt, muss die implementierende Klasse in der Konfigurationsdatei mit dem Schlüssel `cf.ratingmanager` explizit deklariert sein (siehe Abschnitt A.3).

```java
package de.uka.ipd.consensus.impl;
...
public class TopicImpl extends TM4JTopicWrapper implements Topic {
    ...
    public Rating addRating(User rater, double value)
                                    throws RatingManagerException,
                                           DynamicRightsManagerException {
        Rating r = this.getRating( rater );
        if (r != null) {
            r.setValue( value );
        }
        else {
            RatingManager rm =
                this.getOntologyManager().getRatingManager();
            if (rm != null) {
                r = rm.createRating( rater, value );
                rm.addRating( this, r );
            }
        }
        return r;
    }
}
```

Listing 10: Delegation der Topic-Bewertungsmethoden an den RatingManager

Für den Entwickler ist der Vorteil dieses Ansatzes die einfachere Programmierschnittstelle (`TopicImpl.addRating()` fasst die beiden Fälle des Neuanlegens und des Änderns einer Bewertung zusammen) und die intuitivere Zuordnung der Funktionalität (wenn man ein Topic bewerten möchte, muss man die Methode nicht erst bei einer anderen Komponente suchen).

Trotzdem bleibt durch die Delegation die gesamte Funktionalität zur Verwaltung der Bewertungen im RatingManager, der somit eine kohärente Komponente [74, Abschnitt 2.5] mit loser Kopplung darstellt. In der Tat ist der RatingManager vollständig orthogonal zum übrigen System und kann ohne Nebenwirkungen ausgetauscht — oder sogar ganz weggelassen — werden.

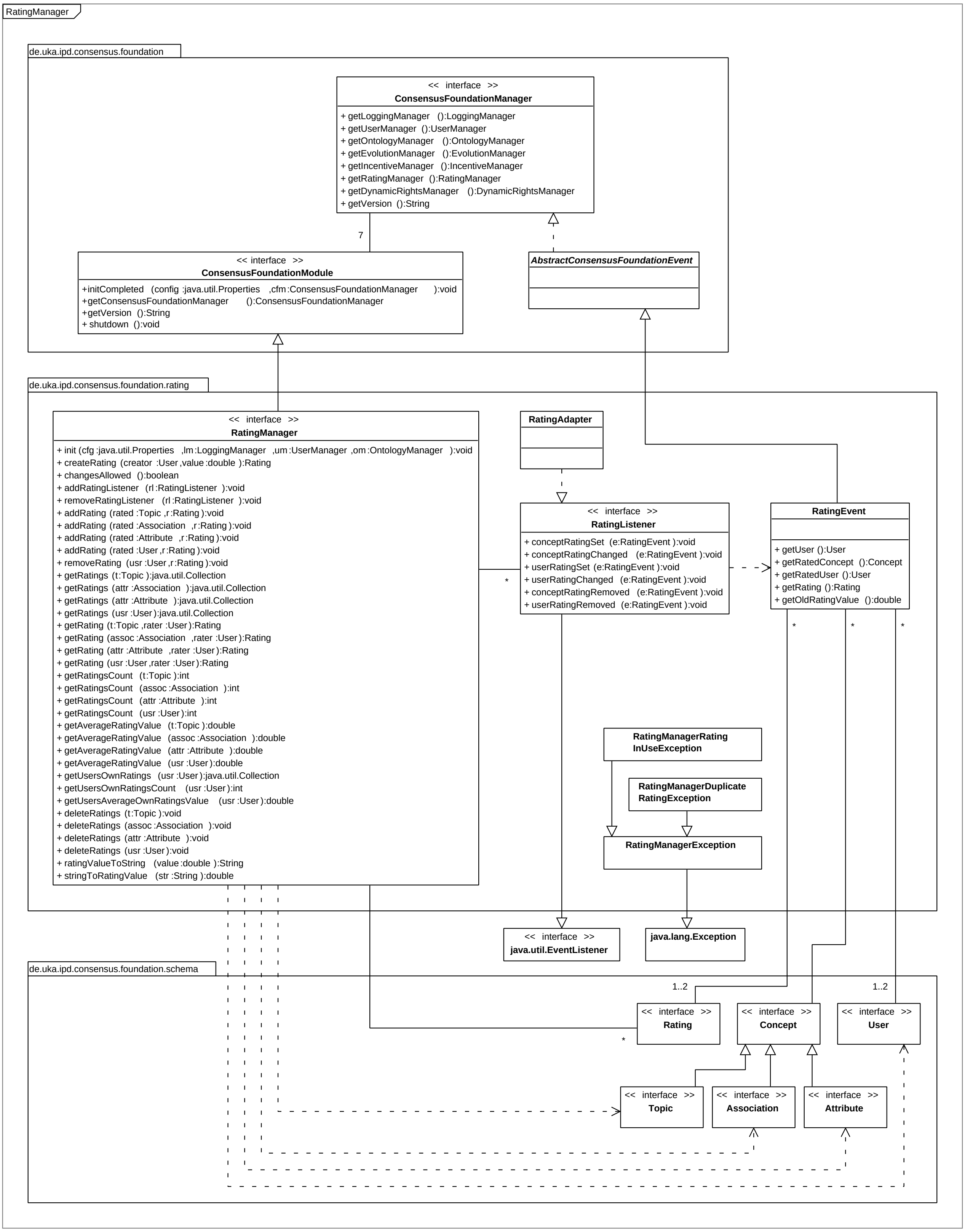


Abbildung 23: Das Paket de.uka.ipd.consensus.foundation.rating

Die Manipulation und Abfrage der eigentlichen Bewertungsdaten (der Wert für Zustimmung bzw. Ablehnung, der bewertende Benutzer, ein Kommentar zur Bewertung, die Bewertungs-Historie) erfolgt über die Schnittstelle `de.uka.ipd.consensus.foundation.schema.Rating`. Auch wenn diese Schnittstelle zum Schema gehört, muss sie vom Bewertungssystem realisiert werden. Die Implementierungen für `Rating` und `RatingManager` müssen also immer zusammen ausgetauscht werden. Dies ist sinnvoll, denn nur der RatingManager als Fabrik der Bewertungen sollte deren innere Struktur kennen, und es ist auch problemlos möglich, denn die Bewertungs-Implementierung ist von den anderen Implementierungen des Schemas vollständig über die Schema-Schnittstellen entkoppelt.

Weiter oben wurden die Schnittstellen `Topic`, `Association`, `Attribute` und `User` genannt, welche die bewertbaren Elemente von ConsensusFoundation darstellen und für die Delegationsmethoden im RatingManager existieren. Die ebenfalls erwähnten Bewertungsmethoden werden nicht von diesen Schnittstellen definiert, sondern von der Schnittstelle `Rateable`, von der die anderen Schnittstellen direkt oder indirekt erben. Es gibt nun aber weitere Schnittstellen, die auch vom Typ `Rateable` sind, namentlich `TypeInstance`, `SuperSubclass` und `Synonym` (siehe Abbildung 17). Dies sind die Schnittstellen für die Assoziationshüllen (Wrapper), mit denen spezielle Assoziationen (beispielsweise für Typ-Instanz-Beziehungen) besser handhabbar werden. Diese Assoziationshüllen werden nicht speziell vom RatingManager behandelt, denn es steht zu erwarten, dass im Laufe der Zeit weitere Hüllen (z.B. für Homonyme) zum Schema hinzukommen. Dennoch soll die Schnittstelle des RatingManagers stabil bleiben und nicht bei jeder neuen Hülle angepasst werden müssen. Daher delegieren die Hüllen-Implementierungen alle Bewertungsaufrufe einfach an die enthaltene (eingehüllte) Assoziation. Diese eingehüllten Assoziationen sind ganz normal bewertbar, auch wenn die eigentliche Assoziation selbst nie in der Anwendung in Erscheinung tritt, sondern immer nur deren Hülle. Die Stabilität der Schnittstelle ist auch der Grund, warum `RatingManager` mehrfach überladene Methoden z.B. für `addRating()` besitzt und nicht nur eine einzige Methode `addRating(Rateable)` — der RatingManager soll alle Typen, deren Bewertungen er verwalten muss, vorab kennen, damit er möglichst unabhängig von Änderungen an anderen Stellen im Rahmenwerk bleibt.

Neben den Methoden zum Abfragen einzelner oder aller Bewertungen definiert `Rateable` noch weitergehende Methoden für alle abgegebenen Bewertungen, und zwar zum Ermitteln ihrer Anzahl und des arithmetischen Mittels ihrer Werte, die im stetigen Intervall $[-1, 1]$ liegen. Genau an dieser Stelle sind Erweiterungen der Schnittstelle denkbar, falls vom RatingManager weitere statistische Auswertungen angeboten werden sollen. Dies kann sinnvoll sein, damit die Berechnungen nicht bei jeder Abfrage erfolgen müssen, sondern nur bei einer Änderung der Berechnungsgrundlage (neue oder geänderte Bewertung). Der RatingManager kann den ermittelten Wert zusammen mit der Bewertung speichern und schnell darauf zurückgreifen. So berechnet auch die Standard-Implementierung den Mittelwert immer bei der Abgabe oder Änderung einer Bewertung neu und liefert bei der Abfrage diesen vorausberechneten Wert. Derartige Erweiterungen von `RatingManager` und `Rateable` sollten aber immer durch die Spezialisierung der Schnittstellen erfolgen und niemals durch eine Änderung der Schnittstellen selbst, damit bestehende Implementierungen von der Ergänzung nicht betroffen sind.

Ohne Erweiterung von Schnittstellen sind solche Auswertungen aber auch an anderer Stelle machbar: In einem Beobachter vom Typ `RatingListener`. Er wird vom RatingManager immer genau dann benachrichtigt, wenn sich eine Bewertung, also die Auswertungsgrundlage, geändert hat. Zwei der Standard-Klassen, der IncentiveManager (siehe Abschnitt 4.8) und der LoggingManager (siehe Abschnitt 4.11), nutzen dies und melden sich als Beobachter am RatingManager an. Als weiteres Anwendungsbeispiel kommt das in [60] beschriebene Bewertungsprotokoll in Frage, das die Kompetenz eines Benutzers (die Wahrscheinlichkeit, dass ein Benutzer ein Konzept korrekt bewertet) berechnen muss. Die *abgeleitete Kompetenz (inferred competence)* errechnet sich aus den offenbar korrekten Bewertungen eines Nutzers und aus der Anzahl aller seiner abgegebenen Bewertungen, und genau diese Berechnung kann in einem RatingListener sinnvoll durchgeführt werden.

Während es die Standard-Implementierung den Nutzern erlaubt, eine einmal abgegebene Bewertung jederzeit wieder zu ändern (die alten Bewertungen werden dabei automatisch in der Bewertungs-Historie abgelegt), ist dies eventuell nicht immer erwünscht. Aus diesem Grund kann man die Methode `changesAllowed()` überschreiben und mit dem Rückgabewert `false` die Änderung von Bewertungen verhindern. Dies wird im Übrigen auch von der Standard-Implementierung beachtet, wenn man sie als Oberklasse nutzt.

Im Laufe der Entwicklung der RatingManager-Implementation trat ein Problem auf. Zunächst sollten die Bewertungen innerhalb der Ontologie als TM4J-Topics gespeichert werden, jedoch wurde die Abfrage einzelner Bewertungen von einem gegebenen Benutzer deutlich langsamer, je mehr Bewertungen zu einem Konzept vorhanden waren. Obwohl sich dies als unkritisch für die Topic-Abfrage erwies (genauer wird dies in Abschnitt 5.3 untersucht), war das Laufzeitverhalten für Bewertungen nicht akzeptabel, da mit sehr vielen Bewertungen pro Topic gerechnet werden muss. Daher erfolgte eine Neuimplementierung in der Klasse `de.uka.ipd.consensus.impl.SQLRatingManagerImpl`, von der die Bewertungen direkt in MySQL persistiert werden und nicht mehr Bestandteil der Ontologie sind (das verwendete Datenbasisschema ist in Abschnitt A.6 dokumentiert). Die alte (unvollständige) Version steht zu Anschauungszwecken noch in der Klasse `TM4JRatingManagerImpl` zur Verfügung, sie sollte aber nicht mehr produktiv eingesetzt werden.

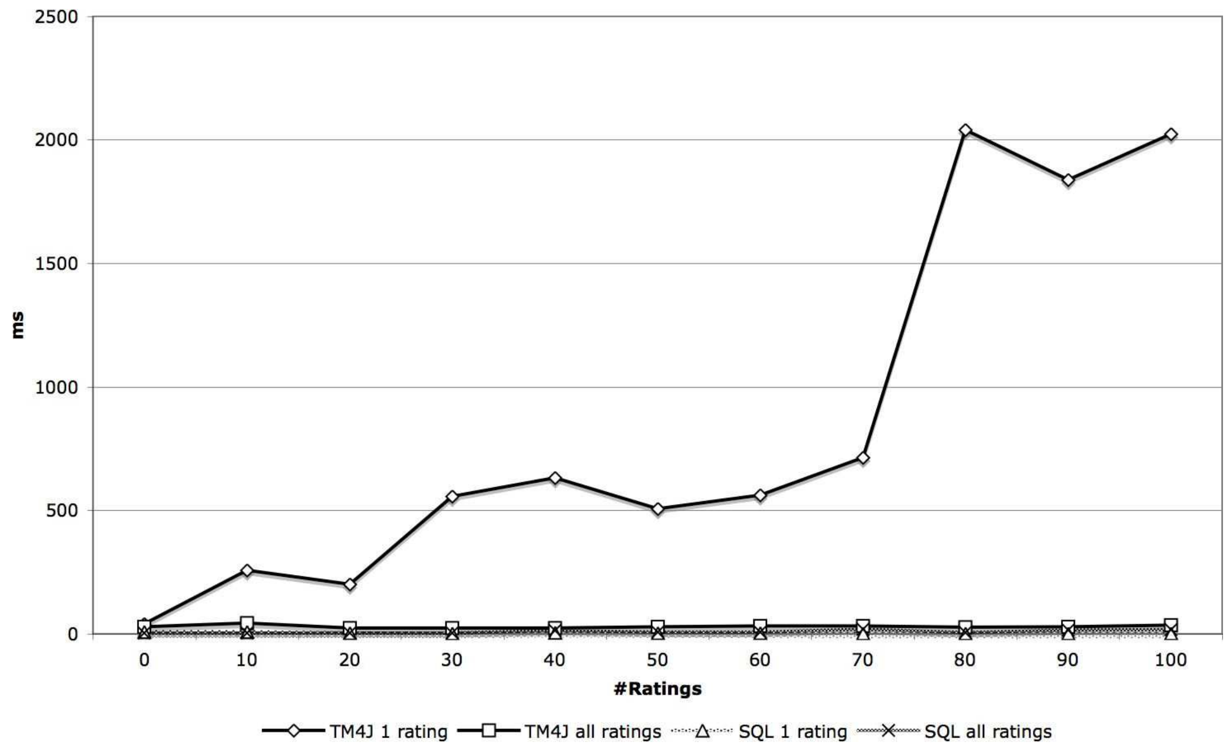


Abbildung 24: Performanzvergleich zwischen TM4J- und SQL-Ratings

Abbildung 24 zeigt einen Vergleich der beiden Implementierungen. Der Test fügt zu einem Topic in zehn Schritten jeweils zehn Bewertungen hinzu, insgesamt also für 100 Anwender. Nach jedem Schritt wird zunächst die eine Bewertung von einem bestimmten Benutzer gelesen, anschließend alle Bewertungen des Topics. Der Test kann mit der Klasse `de.uka.ipd.consensus.test.RatingTestServlet` nachvollzogen werden.[14] Testrechner ist ein iMac G5 2 Ghz mit 2 GB RAM und Mac OS X 10.4.6, kein dedizierter Server, die Software-Versionen sind in Abschnitt A beschrieben.

[14]Sofern das Test-Servlet und der zugehörige Servlet-Filter in `/WEB-INF/web.xml` aktiviert werden.

Die Abfrage aller Bewertungen („TM4J all ratings“) ist bei TM4J ausreichend schnell (im getesteten Bereich unter 50ms), aber bei der gezielten Abfrage einer Bewertung („TM4J 1 rating“) bricht die Performanz ein. Warum? TM4J muss in diesem Fall solange die Bewertungs-Topics abfragen und die Assoziation zum erzeugenden Benutzer nachverfolgen, bis das gewünschte Topic gefunden ist. Leider ist es unerheblich, ob man die Abfragesprache „Tolog“ einsetzt oder die Abfrage von Hand ausprogrammiert, TM4J ist an dieser Stelle für die Bewertungen nicht effizient genug. Die Optimierung von TM4J hätte aber letztendlich deutlich mehr Aufwand bedeutet als das Neuschreiben des SQL-RatingManagers, der im getesteten Bereich bei der Gesamtabfrage („SQL all ratings“) unter 20ms und bei den Einzelabfragen („SQL 1 rating“) unter 10ms bleibt, was auch für viele Bewertungen ausreichend ist.

Zusätzlich zur Performanz sei noch folgende Überlegung angestellt: Sei T Anzahl der Topics und U Anzahl der Benutzer, jeweils ganzzahlig und größer Null. Wenn nur jeder zweite Benutzer jedes zweite Topic bewertet (eine eher konservative Schätzung für Systeme, bei denen die Anwender motiviert werden sollen, möglichst viele Bewertungen abzugeben), ist die Anzahl der Bewertungen $B = \lfloor \frac{1}{2}T \cdot \frac{1}{2}U \rfloor$. Spätestens ab acht Benutzern[15] übersteigt dann die Anzahl der Bewertungen B die Anzahl der Topics T. Die Ontologie enthielte dann also mehr TM4J-Bewertungs-Topics als eigentliche ConsensusFoundation-Topics. Dies kann unerwünscht sein, denn es stellt sich die Frage, ob Bewertungen überhaupt Teil der Ontologie sind oder als Metadaten separat gespeichert werden sollten. All dies spricht dafür, den SQL-RatingManager auch in Zukunft beizubehalten, zumal die Bewertungsdaten dadurch auch für Auswertungen außerhalb das ConsensusFoundation-Rahmenwerks zur Verfügung stehen.

[15] Um die Ganzzahligkeit zu berücksichtigen, nehmen wir $\frac{1}{4}TU \geq T + 1$ an, woraus sich $U \geq 4 + \frac{4}{T}$ ergibt. Mit der vorausgesetzten Topic-Mindestanzahl 1 ergibt sich die hinreichende Benutzer-Anzahl von 8.

4.8 Das Anreizsystem — der IncentiveManager

Mit `de.uka.ipd.consensus.foundation.scoring.IncentiveManager` (siehe Abbildung 25) wird die Schnittstelle für die zentrale Komponente des Anreizsystems definiert. Die Schnittstelle ist dabei sehr schlank, denn die Aufgabe des IncentiveManagers besteht vor allem darin, punktewürdige Ereignisse (Topic wurde erzeugt, Topic wurde bewertet etc.) im Rahmenwerk zu erkennen, zusammenzufassen und an flexibel austauschbare Module weiterzuleiten, die sich um die Punktevergabe und die sich daraus ergebenden Anreize kümmern. Die eigentliche Punktevergabe erfolgt also — zumindest bei der Standard-Implementierung — nicht im IncentiveManager, sondern in einer Klasse außerhalb des Rahmenwerks. Diese Klasse muss die Schnittstelle `ScoringListener` implementieren und wird als Beobachter (Observer) [32] am IncentiveManager registriert. Wie beim Beobachter-Muster üblich können beliebig viele ScoringListener angemeldet werden, was auch deklarativ möglich ist (s.u.).

Während der Punktestand eines Benutzers schon Anreiz genug sein kann, wenn dieser an geeigneter Stelle in der Anwendung beispielsweise als Rangliste angezeigt wird, kann es auch erwünscht sein, den Punktestand in anders geartete Anreize zu transformieren, beispielsweise in Berechtigungen. Hier kommt der DynamicRightsManager (siehe Abschnitt 4.9) ins Spiel, der zwar kein Bestandteil des Anreizsystems ist, aber Berechtigungen als Anreiz durchsetzen kann. So könnten Anwendern unterhalb eines gewissen Punktestandes bestimmte Topics vorenthalten werden, oder einem Anwender wird ab einem gewissen (hohen) Punktestand automatisch die Administrator-Rolle zugewiesen.

Wie eine sinnvolle Punktevergabe erfolgen sollte und mit welchen Anreizen Nutzer motiviert werden können, ist nicht Bestandteil dieser Arbeit. An anderen Stellen wurde bereits untersucht, wie die Punktevergabe erfolgen kann, um möglichst wahrheitsgemäße Bewertungen zu erhalten [48], oder welche Motivationsmechanismen generell wirksam sind, beispielsweise eine hierarchische Mitgliedschaft mit „Bronze“-, „Silber“- und „Gold“-Status für den Anwender, anhand dessen bestimmte Privilegien bzw. Rechte zuerkannt werden [19]. Entsprechend dient die Klasse `de.uka.ipd.consensus.impl.ScoringListenerImpl` nur zu Demonstrationszwecken und vergibt einfach bei jeder Aktion einen Punkt, unabhängig vom Kontext und von der Qualität der Aktion.

Ein etwas ausführlicheres Beispiel zeigt Listing 11. Wiederum handelt es sich nicht um eine empirisch oder statistisch untermauerte Punktevergabe, es soll nur die Funktionsweise verdeutlicht werden. Der ScoringListener vergibt beim Erzeugen eines Topics oder einer Assoziation zwei Punkte, beim Hinzufügen eines Attributs einen Punkt und beim Bewerten eines Konzepts (sowohl bei der Erstbewertung als auch bei einer Änderung) ebenfalls einen Punkt, sofern der Anwender eine andere Bewertung als „neutral“ (was hier als „keine Meinung“ aufgefasst wird) abgegeben hat.

```
public class MyScoringListenerImpl implements ScoringListener {

    public void score(ScoringEvent event) {
        User   usr    = event.getUser();
        double punkte = usr.getScore();

        switch( event.getId() ) {
            case ScoringEvent.TOPIC_CREATED:
            case ScoringEvent.ASSOCIATION_CREATED:
                punkte += 2.0;
                break;

            case ScoringEvent.ATTRIBUTE_ADDED:
                punkte += 1.0;
                break;
```

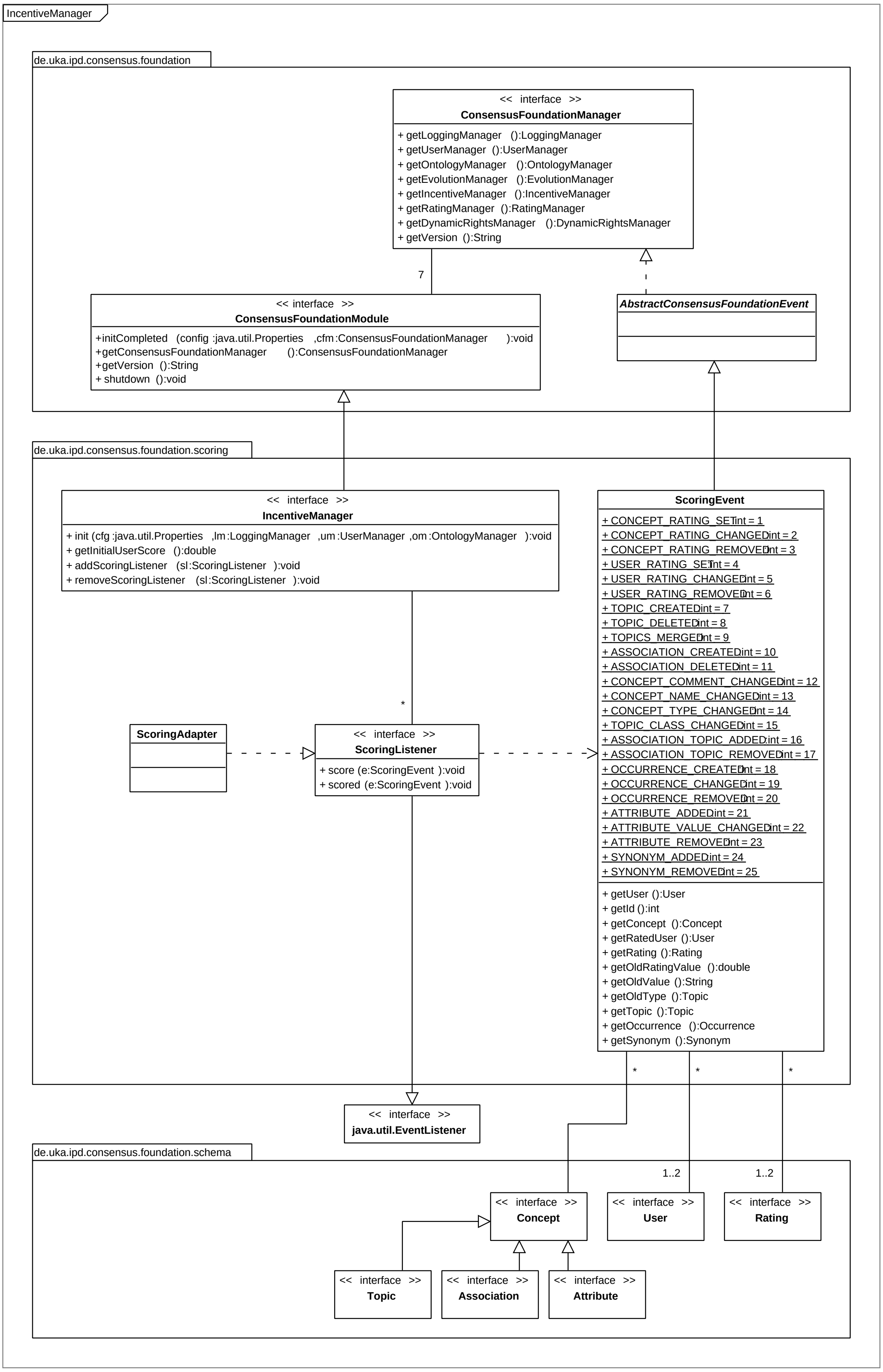


Abbildung 25: Das Paket de.uka.ipd.consensus.foundation.scoring

```

            case ScoringEvent.CONCEPT_RATING_SET:
            case ScoringEvent.CONCEPT_RATING_CHANGED:
                Rating r = event.getRating();
                if (r.getValue() != Rating.NEUTRAL) {
                    punkte += 1.0;
                }
                break;
        }

        usr.setScore( punkte );
    }

    public void scored(ScoringEvent event) { }
}
```

Listing 11: Punktevergabe für das Erzeugen und Bewerten von Konzepten

Die zwei Schritte der Punktevergabe und der Ermittlung der sich daraus ergebenden Anreize ist auch bei der Schnittstelle `ScoringListener` in zwei Methoden aufgeteilt. Zunächst ruft der IncentiveManager in allen registrierten Beobachtern die Methode `score()` auf, wo jeder Listener seine Punkte vergeben kann. Anschließend wird in allen diesen Beobachtern die Methode `scored()` aufgerufen, in der keine weiteren Punkte mehr vergeben werden dürfen. Hier können die Listener stattdessen den aktuellen Punktestand in andersartige Anreize umwandeln. Der Beispiel-Listener nutzt dies nicht und implementiert die Methode daher leer. Listing 14 in Abschnitt 4.9 zeigt aber den Anwendungsfall, dass der DynamicRightsManager als ScoringListener in der Methode `scored()` neue Berechtigungen als Anreiz ermittelt.

[48, Kapitel 6] beschreibt die Umsetzung diverser Modelle zur Punktevergabe im Simulations-Rahmenwerk „Pinocchio“. Innerhalb des Pakets `de.uka.ipd.pinocchio.sim` werden dort verschiedene Algorithmen realisiert, die jeweils mit einer simulierten Nutzergemeinschaft getestet werden. Da die Pinocchio-Klassen zur Punktevergabe auf der erwähnten `ScoringListener`-Schnittstelle des ConsensusFoundation-Rahmenwerks basieren, können die Algorithmen auch problemlos innerhalb von ConsensusFoundation-Anwendungen eingesetzt werden, wodurch die simulierte durch eine reale Nutzergemeinschaft ersetzt werden kann.

Das Registrieren eines ScoringListeners ist entweder programmatisch oder aber deklarativ möglich. Damit letzteres funktioniert, sind alle IncentiveManager-Implementierungen verpflichtet, den Schlüssel `cf.scoringlistener` auszuwerten (siehe Abschnitt A.3), von allen dort aufgeführten Klassen Objekte zu erzeugen und diese bei sich selbst zu registrieren. Mit diesem Ansatz sind Testläufe mit unterschiedlichen Listenern einfach realisierbar, ohne jedesmal die Anwendung neu übersetzen zu müssen.

Jede Implementierung von `IncentiveManager` sollte außerdem alle in der Klasse `ScoringEvent` als Konstanten definierten Ereignistypen an die ScoringListener melden können. Die Standard-Implementierung realisiert dafür die Schnittstellen `RatingListener`, `OntologyListener` und `EvolutionListener` und registriert sich bei den entsprechenden Komponenten als Beobachter. Spezielle Anpassungen für das Anreizsystem sind dadurch in den anderen Komponenten nicht notwendig!

Der Beispiel-Listener hat gezeigt, wie die Punktevergabe in der Methode `score()` umgesetzt wird, was auch der in der Praxis am häufigsten gewählte Weg sein dürfte. Sollte die Fallunterscheidung in der Methode zu groß werden, kann man die einzelnen Fälle beispielsweise an private Methoden der Beobachter-Klasse delegieren. Falls eine solche Fallunterscheidung überhaupt nicht gewünscht ist, weil damit nur Ereignisse differenziert werden, die bereits vorher einzeln erkannt, dann aber vom

IncentiveManager zu einem ScoringEvent zusammengefasst wurden, bietet sich als Alternative die Implementierung eines eigenen IncentiveManagers an. Die einfachste Lösung ist das Ableiten einer Unterklasse von `de.uka.ipd.consensus.impl.IncentiveManagerImpl`. Die Standard-Implementierung kümmert sich bereits um das Aufrufen eventuell registrierter Beobachter und ist zum Erkennen der punktewürdigen Ereignisse selber als Beobachter an diversen Komponenten angemeldet, so dass man nur noch die gewünschten Ereignis-Methoden überschreiben muss, in denen die Punktevergabe erfolgt. Listing 12 zeigt, wie ein spezialisierter IncentiveManager auf die Vergabe bzw. die Änderung von Bewertungen reagieren kann (vgl. Zeilen 17-23 in Listing 11). Der Aufruf der überschriebenen Methode mit `super` ist notwendig, weil dort die registrierten ScoringListener benachrichtigt werden.

```
public class MyIncentiveManagerImpl extends IncentiveManagerImpl {

    public void conceptRatingSet(RatingEvent event) {
        super.conceptRatingSet( event );
        ... spezialisierte Punkteberechnung ...
    }

    public void conceptRatingChanged(RatingEvent event) {
        super.conceptRatingChanged( event );
        ... spezialisierte Punkteberechnung ...
    }
}
```

Listing 12: Spezialisierung des IncentiveManagers

Da sowohl die ScoringListener als auch der IncentiveManager flexibel austauschbar sind, kann man sich für die Variante entscheiden, welche die beste Struktur und Übersichtlichkeit verspricht. Beachten Sie nur, dass der IncentiveManager eine optionale Komponente ist, die mit dem Schlüssel `cf.incentivemanager` in der Konfigurationsdatei explizit deklariert werden muss (siehe Abschnitt A.3).

Eine Unterklasse des Standard-IncentiveManagers muss man auch dann einsetzen, wenn man den anfänglichen Punktestand von neuen Benutzern ändern möchte. Normalerweise wird der Wert mit 0 vorbelegt, was für die meisten Anwendungen der gewünschte Wert sein dürfte. Um dies zu ändern, überschreiben Sie die Methode `getInitialUserScore()` und liefern den neuen Anfangswert zurück. Wenn häufiger mit wechselnden Anfangspunkteständen gearbeitet wird, sollte ein spezialisierter IncentiveManager in Betracht gezogen werden, der den Punktestand beispielsweise aus der Konfigurationsdatei einlesen kann.

4.9 Dynamische Rechtevergabe — der DynamicRightsManager

Der DynamicRightsManager ist als Komponente für die Rechtevergabe dafür zuständig, welche Konzepte der Ontologie ein Anwender sehen und welche Aktionen er ausführen darf. Er kann damit zum einen klassische Berechtigungen durchsetzen, die auf Rollen oder Gruppen basieren und beispielsweise zur Unterscheidung von Gästen, registrierten Anwendern und Administratoren verwendet werden — der in Abschnitt 4.10 vorgestellte UserManager macht Gebrauch davon. Zum anderen kann die Rechtevergabe als Teil des Anreizsystems Berechtigungen dynamisch anhand des Punktestandes eines Nutzers vergeben, was für auf dem ConsensusFoundation-Rahmenwerk aufbauende Applikationen die interessantere Fähigkeit sein dürfte.

Die Schnittstelle `de.uka.ipd.consensus.foundation.rights.DynamicRightsManager` (siehe Abbildung 26) ist im Wesentlichen eine große Sammlung von Anwendungsfällen, die sich aus der Funktionalität der anderen Komponenten ergeben (Konzepte erzeugen, löschen, abfragen etc.). Entsprechend wird der DynamicRightsManager von den anderen Komponenten aufgerufen, wenn dort eine Berechtigung geprüft werden muss — die Rechtevergabe ist eine passive Komponente und wird nicht selber aktiv.

Für jeden einzelnen Anwendungsfall gibt es eine `mayXXX()`-Methode, der ein Benutzer-Objekt übergeben wird, für das die Berechtigung zu prüfen ist. Je nach Fall müssen gegebenenfalls noch weitere Parameter übergeben werden, beispielsweise ein Topic-Typ, falls das Erzeugen eines Topics überprüft werden soll. Rückgabe ist ein boolescher Wert, ob die Berechtigung besteht oder nicht. Die `mayXXX()`-Methoden sind die Stellen, an denen die eigentliche Funktionalität des DynamicRightsManagers ausprogrammiert wird. Zusätzlich gibt es zu jeder `mayXXX()`-Methode eine entsprechende `checkXXX()`-Methode, welche die Rückgabe der `mayXXX()`-Methode auswertet und eine Ausnahme meldet (d.h. eine Exception wirft), sofern keine Berechtigung für die gewünschte Aktion besteht.

Obwohl die Verwendung der `checkXXX()`-Varianten in den Implementierungen der anderen Komponenten sehr bequem ist (die dortigen Methoden werfen die `DynamicRightsManagerException` einfach weiter, sie ist also Teil ihrer Signatur; siehe Listing 13), birgt dies ein Problem: Wenn eine Ausnahme auftritt, kostet deren Behandlung im Vergleich zum Normalfall relativ viel Zeit (siehe [87, Kapitel 6]). Während dies bei einzelnen Prüfungen akzeptabel ist, kann sich dies negativ auf die Performanz auswirken, wenn für eine Aktion viele Prüfungen durchgeführt werden müssen — beispielsweise, weil bei der Abfrage aller Instanzen eines Topic-Typs für jede Instanz entschieden werden muss, ob der Benutzer sie sehen darf. In diesem Fall greift die Standard-Implementierung des Rahmenwerks auf die `mayXXX()`-Methoden zu, mit denen sich viele Prüfungen unabhängig von deren Ergebnis effizient ausführen lassen, weil hier mit einer `boolean`-Rückgabe und nicht mit Ausnahmen gearbeitet wird. Dies ist auch der Grund dafür, warum beide Methoden-Varianten vorgesehen sind, selbst wenn sich dadurch die Größe der Schnittstelle verdoppelt. Listing 14 zeigt die Implementierung der Berechtigung, die Abfragesprache (in der Beispiel-Anwendung „Tolog“) zu nutzen (`mayQuery() / checkQuery()`), was hier nur Administratoren gestattet ist.

```
package de.uka.ipd.consensus.impl;
...
public class OntologyManagerImpl
        extends AbstractConsensusFoundationModule
        implements OntologyManager {
    ...
    public Topic createTopic(User creator, String id,
                             String name, Topic type)
            throws OntologyManagerException,
                   DynamicRightsManagerException {
```

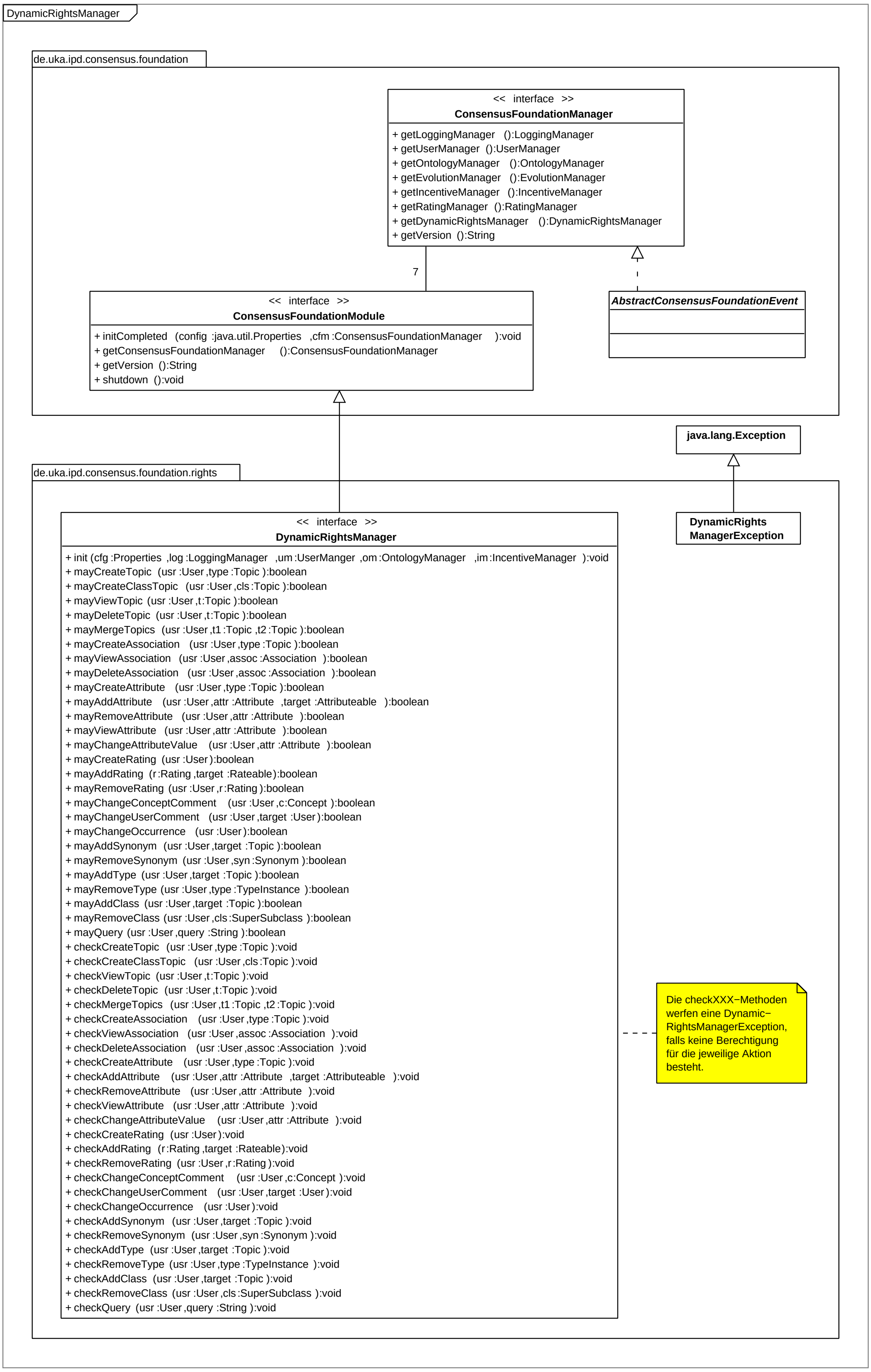


Abbildung 26: Das Paket de.uka.ipd.consensus.foundation.rights

```
        ...
        DynamicRightsManager drm = this.getDynamicRightsManager();
        if (drm != null) {
            drm.checkCreateTopic( creator, type );
        }
        ...
    }
}
```

Listing 13: Abprüfen einer Berechtigung

Die `checkXXX()`-Methoden sind an die Schnittstelle von `java.lang.SecurityManager` angelehnt ([51] und [73, Kapitel 4]). Auf weitergehende Konzepte wie `java.security.AccessContoller` und `Permission` wie bei der Java-Klassenbibliothek wurde aber verzichtet, weil zum einen die für das Rahmenwerk nötigen Anwendungsfälle vorab bekannt sind und somit kein Bedarf für eine flexible Erweiterbarkeit besteht, und weil zum anderen die Berechtigungen normalerweise dynamisch anhand des Punktestands berechnet werden, was für einen programmatischen Ansatz spricht (und gegen den deklarativen mit Policy-Dateien [73, Seite 33 ff.]). Zumindest für statische Berechtigungen (z.B. aufgrund von bestimmten Rollen) ist es aber denkbar, dass eine Implementierung des DynamicRightsManagers zur Konfiguration deklarative Policy-Dateien wie bei den SecurityManagern der Java-Klassenbibliothek einsetzt.

Wenn eine alternative Implementierung des DynamicRightsManagers entwickelt wird, sollte bei den einzelnen Methoden beachtet werden, dass diese zum Teil sehr häufig aufgerufen werden (z.B. `mayViewTopic()`). Die Entscheidung, ob der angegebene Benutzer die jeweilige Berechtigung besitzt, sollte also nicht zu viel Rechenzeit beanspruchen. Daher kann es Sinn machen, die Berechtigungen im Voraus zu berechnen, sobald sich die Entscheidungsgrundlage für die Rechtevergabe — der Punktestand des Nutzers — geändert hat. Zu diesem Zweck implementiert auch `DynamicRightsManagerImpl` die Schnittstelle `ScoringListener` (und registriert sich als Beobachter beim IncentiveManager), in deren Methode `scored()` man den neuen Punktestand des Benutzers auslesen und daraus die neuen Rechte ermitteln kann. Die Standard-Implementierung nutzt dies zwar nicht, zeigt damit aber, wie dies bei anderen Implementierungen realisiert werden kann (siehe auch Listing 14).

```
public class MyDynamicRightsManagerImpl
        extends AbstractConsensusFoundationModule
        implements DynamicRightsManager, ScoringListener {
    ...
    public void initCompleted(Properties config,
                              ConsensusFoundationManager cfm)
                              throws Exception {
        super.initCompleted(config, cfm);
        IncentiveManager im = cfm.getIncentiveManager();
        if (im != null) {
            im.addScoringListener( this );
        }
    }
    ...
    public boolean mayQuery(User user, String query) {
        return (user != null) && user.isAdmin();
    }

    public void checkQuery(User user, String query)
```

```
                           throws DynamicRightsManagerException {
        if (!mayQuery( user, query )) {
            throw new DynamicRightsManagerException( "..." );
        }
    }

    public void scored(ScoringEvent event) {
        User   usr    = event.getUser();
        double punkte = usr.getScore();
        ... Berechnung der neuen Berechtigungen ...
    }
}
```

Listing 14: Rollenbasierte und punkteabhängige Rechtevergabe

Die Standard-Implementierung `de.uka.ipd.consensus.impl.DynamicRightsManagerImpl` erlaubt die meisten Aktionen allen Anwendern und beschränkt nur wenige Aktionen auf Administratoren oder den Erzeuger des jeweiligen Konzepts.[16] Mit dem Konfigurations-Schlüssel `cfimpl.dynrights.typed-only` kann aber auf einfache Weise erreicht werden, dass nur Administratoren typenlose Konzepte anlegen dürfen — alle anderen Anwender sind dann gezwungen, beim Anlegen eines neuen Konzepts immer auch einen passenden Typ zu übergeben.

Damit überhaupt ein DynamicRightsManager, der eine optionale Komponente ist, verwendet wird, muss in der ConsensusFoundation-Konfigurationsdatei zum Schlüssel `cf.dynamicrightsmanager` ein passender Klassenname eingetragen sein (siehe Abschnitt A.3).

[16] Die genauen Berechtigungen sind in der Javadoc-Dokumentation nachzulesen.

4.10 Benutzerverwaltung — der UserManager

Das ConsensusFoundation-Rahmenwerk ist im Gegensatz zu Bibliotheken wie TM4J, deren Fokus auf der programmatischen Verwaltung von Ontologien liegt, von Anfang an darauf ausgelegt, dass die Konzepte der Ontologie nicht automatisiert angelegt oder von anonymen, nicht unterscheidbaren Anwendern eingegeben werden. Es ist vielmehr die Frage interessant, *wer* etwas anlegt oder bewertet, um diesen unterscheidbaren Individuen zusätzliche Anreize für weiteres Anlegen bzw. Bearbeiten von Konzepten und für weitere und wahrheitsgemäße Bewertungen zu geben. Natürlich kann ein Anwender oder eine Gemeinschaft von Anwendern simuliert werden (siehe das Simulations-Rahmenwerk „Pinocchio“ [48]), dies spielt aber für die folgenden Überlegungen keine Rolle. Wichtig ist nur, dass ConsensusFoundation prinzipiell Anwender unterscheiden kann — und dazu muss die entsprechende Verwaltungs-Funktionalität zur Verfügung stehen.

Eine Benutzerverwaltung gibt es in jedem mehrbenutzerfähigen System (beispielsweise Web-Portale oder Content-Management-Systeme) — in Abschnitt 2.2 war dies Grundlage für die meisten der vorgestellten Bewertungssysteme. Welche Daten werden bei solchen Systemen typischerweise pro Benutzer verwaltet?

- Ein Benutzername (Kürzel) zur eindeutigen Identifikation des Anwenders. Dieser wird üblicherweise vom Anwender selber ausgesucht und ihm von der Anwendung entsprechend zugewiesen, sofern nicht schon ein anderer Nutzer diesen Namen verwendet.
- Ein Realname (eventuell explizit in Vor- und Nachname aufgeteilt). Diese Angabe ist oft optional, was dem Anwender mehr Privatsphäre bietet.
- Eine E-Mail-Adresse, über die die Applikation mit dem Benutzer in Kontakt treten kann — beispielsweise, um ihn über Änderungen an von ihm erzeugten Topics zu informieren. Die E-Mail-Adresse sollte aus Datenschutzgründen nur mit Zustimmung des Besitzers anderen Anwendern angezeigt werden.
- Berechtigungen, welche Aktionen der Benutzer innerhalb der Anwendung ausführen darf. Diese Rechte werden entweder automatisiert von der Anwendung vergeben (z.B. abhängig davon, wie lange der Anwender schon beim System registriert ist, oder — speziell bei Bewertungssystemen — als Anreiz für gesammelte Punkte) oder von einem Administrator zugewiesen.
- Ein Punktestand (Score), den der Benutzer beispielsweise durch das Anlegen und Bewerten von Topics oder durch das Bewerten anderer Anwender erworben hat. Dieser Punktestand kann als Entscheidungsgrundlage für das Anreizsystem genutzt werden.
- Für Web-Applikationen kann zur Vereinfachung ein Status gespeichert werden, ob der Benutzer zur Zeit am System angemeldet ist oder nicht. Die Nutzung dieses Zustands sollte aber optional sein und der Anwendungsebene vorbehalten bleiben, das Rahmenwerk selbst sollte diesen Zustand nicht weiter auswerten.

Als Besonderheit kommt bei ConsensusFoundation hinzu, dass ein Anwender (repräsentiert durch die Schnittstelle `de.uka.ipd.consensus.foundation.schema.User`) nicht nur Teil der Benutzerverwaltung ist, sondern vor allem auch als Teil der Ontologie verfügbar sein muss, um Beziehungen zwischen dem Anwender und den Elementen der Ontologie aufbauen zu können (siehe auch das Entwurfs-Schema in Abbildung 17). Dies ist wichtig, falls das Rahmenwerk zusammen mit einer externen

Benutzerverwaltung eingesetzt werden soll: Die Alternativimplementation der ConsensusFoundation-Benutzerverwaltung wäre dann im Wesentlichen nur eine Fassade (Facade) [32] für die externe Verwaltung und würde nur wenig Funktionalität selber realisieren (vor allem die Verwaltung des Punktestands). Es muss also bei den Schnittstellen der Benutzerverwaltung darauf geachtet werden, dass sich das Rahmenwerk in bestehende Systeme integrieren lässt.

Die Vergabe von Berechtigungen an die Benutzer mithilfe von Rollen wurde in Abschnitt 2.3.3 beschrieben. ConsensusFoundation bietet zur Rollenverwaltung die Schnittstelle `UserRole` an.

Mit diesen Rahmenbedingungen und den Anwendungsfällen aus Abbildung 13 ergibt sich für den im Paket `de.uka.ipd.consensus.foundation.user` definierten `UserManager` der in Abbildung 27 ersichtliche Entwurf.

Der UserManager bietet Methoden zum Anlegen, Ändern und Löschen von Benutzerkonten[17], wobei nur der Benutzername (als Identität) und das vom Anwender gewählte Passwort benötigt werden. Alle weiteren Daten (Vor- und Nachname, E-Mail-Adresse) werden hier nicht explizit als Parameter aufgeführt, da sie nicht in allen Systemen Verwendung finden. Solche zusätzlichen Parameter können in einem Schlüssel-Wert-Paar-Speicher (einer Map) übergeben werden, für die jede Implementierung des UserManagers bestimmte Schlüssel festlegen kann. Die Standardimplementierung definiert Schlüssel für den Vornamen, Nachnamen, die E-Mail-Adresse und einen Kommentartext[18]. Dies erzeugt zwar eine Abhängigkeit von der Steuerung (und im Zweifel auch von der Darstellung) zur konkreten Implementierung, aber diese Abhängigkeit würde bei jeder anderen Benutzerverwaltung (und den unterschiedlichen Feldern, die in den Masken bearbeitet werden sollen) auch bestehen. Die UserManager-Schnittstelle ist wenigstens so universell, dass diese Abhängigkeit nicht zwingend bestehen muss.

Die Methoden zum An- und Abmelden[19] kümmern sich vor allem um die Passwort-Prüfung und um die Benachrichtigung von eventuell registrierten Beobachtern vom Typ `UserListener`. So können beliebige Klassen — gerade auch in der Steuerungsschicht — diesen wichtigen Zustandswechsel eines Benutzers mitbekommen. Wichtig ist in diesem Zusammenhang die Frage, was passiert, wenn der Anwender vergisst, sich abzumelden. Den angemeldeten Zustand unbegrenzt weiter beizubehalten, ist unter anderem aus Gründen der Sicherheit nicht wünschenswert (andere Nutzer könnten Zugriff auf fremde Anwenderdaten erhalten). Daher bietet die User-Standardimplementierung hierfür eine einfache, aber wirkungsvolle Unterstützung an: Die Klasse implementiert die Servlet-Schnittstelle `HttpSessionBindingListener`, wodurch sich jedes Benutzer-Objekt selber abmeldet, wenn die vom Web-Server verwaltete Sitzung (Session) abgelaufen ist (siehe Listing 15). Dies setzt natürlich voraus, dass das Rahmenwerk in einem Servlet-Container eingesetzt und dass das angemeldete User-Objekt in der jeweiligen Servlet-Session abgelegt wird, aber genau dies dürfte der Normalfall sein. Und falls die Anwendung außerhalb eines Servlet-Containers verwendet wird, hat diese Abhängigkeit von der Servlet-API keine Nebenwirkungen.

```
package de.uka.ipd.consensus.impl;
...
public class UserImpl
        extends TM4JObjectWrapper
        implements User, javax.servlet.http.HttpSessionBindingListener {
    private transient UserManager manager;
    ...
```

[17] Genutzt werden diese Methoden in den Klassen `de.snailshell.consensus.demo.actions.RegisterAction`, `ChangeAccountAction` und `DeleteAccountAction` der Beispiel-Applikation, wo die Verwendung eingesehen werden kann.

[18] In der Klasse `de.uka.ipd.consensus.impl.UserManagerImpl` finden sich zu diesem Zweck die Konstanten `USERDATA_FIRSTNAME_KEY`, `USERDATA_LASTNAME_KEY`, `USERDATA_EMAIL_KEY` und `USERDATA_COMMENT_KEY`.

[19] Diese Methoden finden in der Beispiel-Applikation in den Klassen `de.snailshell.consensus.demo.actions.LoginAction` und `LogoutAction` Verwendung.

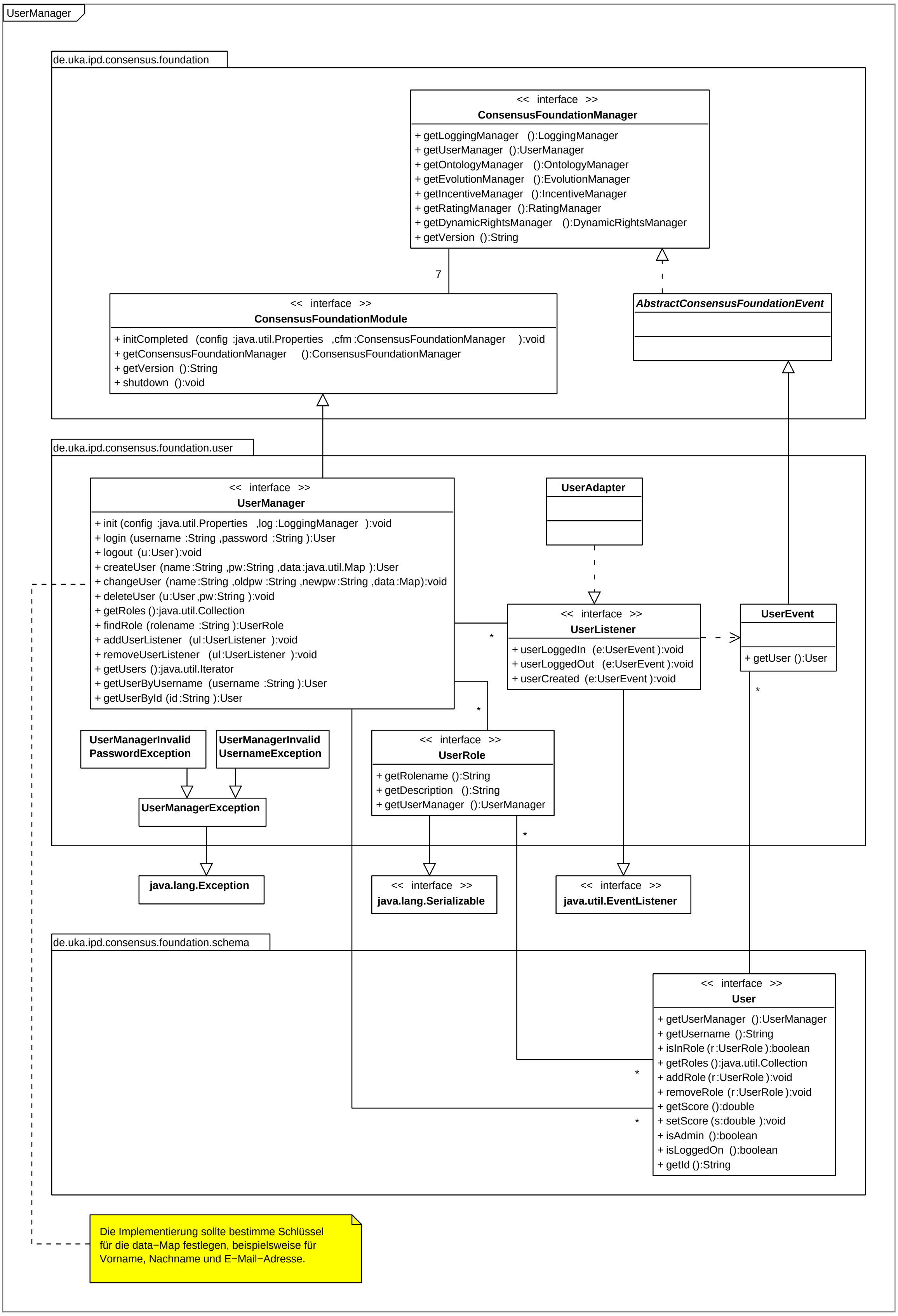


Abbildung 27: Das Paket de.uka.ipd.consensus.foundation.user

```
    public void valueBound(HttpSessionBindingEvent event) { }

    public void valueUnbound(HttpSessionBindingEvent event) {
        try {
            if (this.isLoggedOn()) {
                manager.logout( this );
            }
        }
        catch (UserManagerException e) {
            manager.getLoggingManager().error( e );
        }
    }
}
```

Listing 15: Automatisches Abmelden von Nutzern beim Ablauf einer Sitzung

Wie bereits oben erwähnt „lebt“ ein User in zwei Welten — in der Ontologie und in der Benutzer-Datenbasis. In beiden muss er eindeutig identifizierbar sein. Allerdings ist nicht garantiert, dass ein und dieselbe Kennung in beiden Bereichen eindeutig ist bzw. überhaupt eine gültige Kennung darstellt. Daher verwaltet der UserManager zwei Kennungen, die unterschiedlich sein können, aber denselben Benutzer identifizieren. Der Benutzername ist eindeutig innerhalb der Benutzer-Datenbasis, er kann mit `getUsername()` abgefragt werden. Für die Ontologie wird automatisch eine eindeutige Kennung erzeugt, die dann mit `getId()` ermittelt werden kann. So ist es der Anwendung möglich, auf jeden Fall das entsprechende Benutzerobjekt vom UserManager abzufragen, je nachdem welche Kennung gerade zur Verfügung steht. Und über das ermittelte User-Objekt ist dann auch die Umwandlung der Kennung für den jeweils anderen Bereich machbar.

Die Benutzerdaten (inkl. Punktestand, Rollen und Schatten-Passwort) werden von der Standardimplementierung direkt in MySQL abgelegt (siehe Abschnitt A.6). Sie sind absichtlich nicht Teil der Ontologie, weil dadurch zum einen die privaten Nutzerdaten besser geschützt werden können, und weil zum anderen bei einer externen Benutzerverwaltung sowieso keine Wahl besteht. Wenn Letztere eingesetzt wird, werden die Nutzerdaten extern verwaltet, und in der Ontologie müsste eine — eventuell nur aufwändig synchronisierbare — Kopie der Daten gehalten werden. Dennoch muss der Benutzer selbst innerhalb der Ontologie verfügbar sein, um sinnvoll Assoziationen zu ihm aufbauen zu können (z.B. weil er als Erzeuger eines Topics mit diesem assoziiert wird). Aus diesem Grund legt die Benutzerverwaltung bei der Abfrage eines User-Objekts automatisch ein Stellvertreter-Topic (Proxy) [32] mit passender Kennung, aber ohne die eigentlichen Benutzerdaten in der Ontologie an, sofern ein solches Topic noch nicht vorhanden ist.[20]

Welche Berechtigungen aus den Rollen entstehen, wird vom Rahmenwerk weitestgehend der jeweiligen Anwendung überlassen. Nur zwei Rollen sind von ConsensusFoundation fest vorgesehen und werden innerhalb der Standardimplementierung ausgewertet. Dadurch ist das Rahmenwerk leicht in rollenbasierte Benutzerverwaltungen zu integrieren, ohne dies zwingend vorauszusetzen. Sichergestellt werden muss von der Implementierung nur, dass die beiden Rollen „Administrator“ und „Super-Administrator“ vorhanden sind (Letzterer ist für die Verwaltung der Administrator-Rechte anderer Nutzer zuständig).[21] ConsensusFoundation sieht vor, dass in der Konfigurationsdatei mit

[20] Das Erzeugen des Stellvertreter-Topics erfolgt in der überladenen Methode `de.uka.ipd.consensus.impl.UserImpl.getInstance()`, die gleichzeitig auch einen Schattenspeicher für alle angemeldeten Benutzer verwaltet. Im Gegensatz zum allgemeinen Ontologie-Schattenspeicher (siehe Abschnitt 5.3) geschieht dies mit starken Referenzen, weil die angemeldeten `User`-Objekte sehr häufig benötigt werden und die Anzahl der angemeldeten Nutzer überschaubar ist, den Hauptspeicher also nicht übermäßig belastet.

[21] `UserManagerImpl` bietet dafür die Rollen-Namen-Konstanten `ROLE_ADMIN` und `ROLE_SUPERADMIN` an, für die auch automatisch `UserRole`-Objekte erzeugt werden.

dem Schlüssel `cf.superadmin` der Benutzername angegeben wird, dem beim Anmelden automatisch Super-Administrator-Rechte zugewiesen werden. Dadurch ist unabhängig von der Implementierung der Benutzerverwaltung sichergestellt, dass es überhaupt einen Nutzer gibt, der das System administrieren darf — die Beispiel-Anwendung erlaubt diesem Anwender u.a. den uneingeschränkten Im- und Export der Ontologie.

Die Standard-Implementierung des UserManagers kann ausgetauscht werden, indem in der Consensus-Foundation-Konfigurationsdatei unter dem Schlüssel `cf.usermanager` ein geeigneter Klassenname als Wert eingetragen wird (siehe Abschnitt A.3).

4.11 Protokollierung der Aktionen — der LoggingManager

Das Paket `de.uka.ipd.consensus.foundation.logging` definiert nur eine einzige Schnittstelle, den `LoggingManager` (siehe Abbildung 28). Der LoggingManager ist für zwei Einsatzzwecke konzipiert: Zum einen sollen die üblichen Fehler-, Debug- und Informationsausgaben an einer zentralen Stelle entgegen genommen werden, damit die Protokollierung nicht mehrfach konfiguriert werden muss und an einer einzigen Stelle aktiviert bzw. deaktiviert werden kann. Zum anderen sollen die diversen (Benutzer-)Aktionen im Rahmenwerk erkannt und in einem beliebigen Format (beispielsweise in einer Datenbank zur flexiblen Auswertung) gesichert werden können.

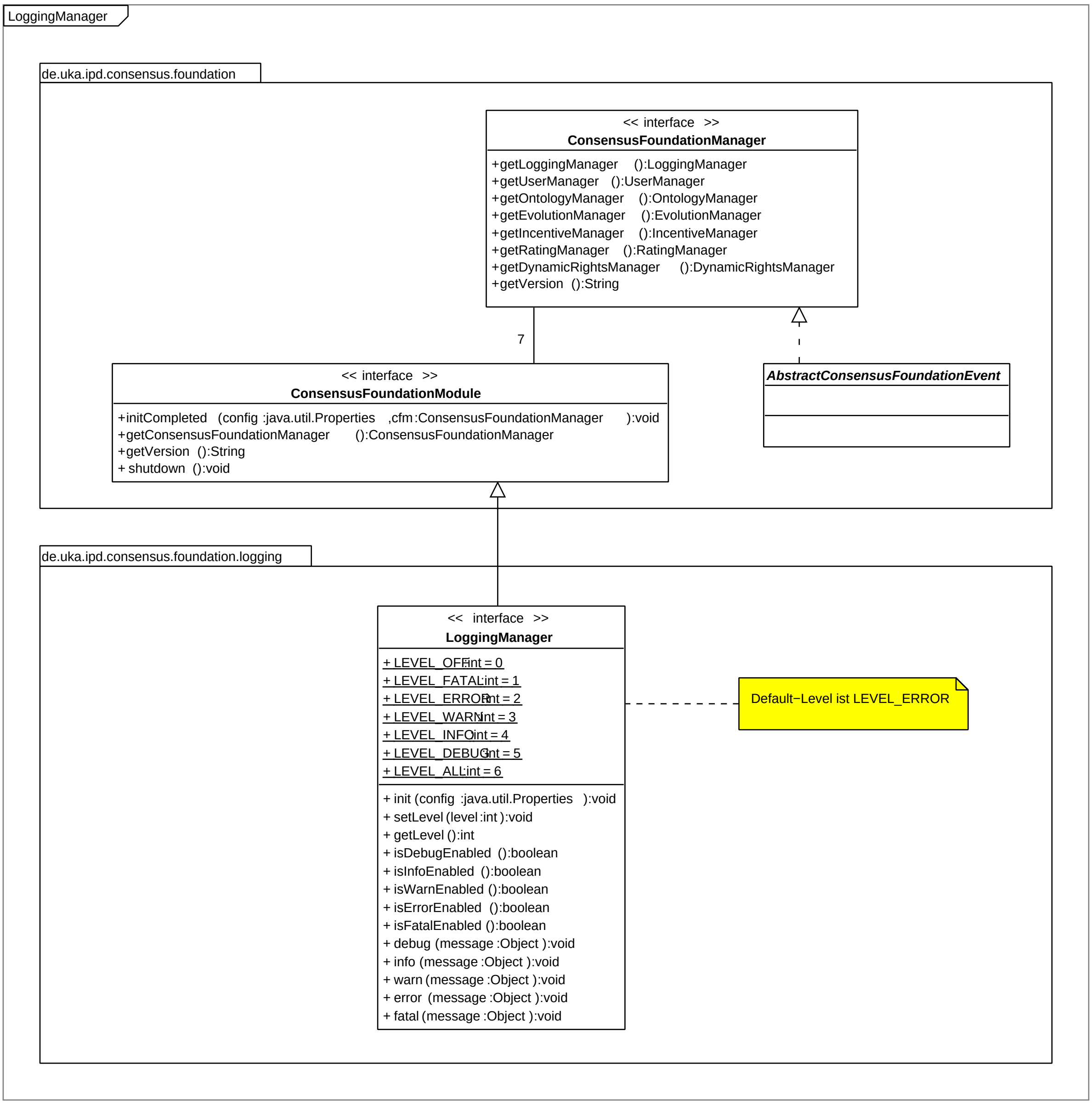


Abbildung 28: Das Paket de.uka.ipd.consensus.foundation.logging

Für den ersten Zweck ist die Schnittstelle sehr ähnlich zu denen der bekannten Java-Logging-Rahmenwerke definiert, damit sich die ConsensusFoundation-Protokollierung gut in bestehende Systeme integrieren lässt. Insbesondere zum häufig eingesetzten *Commons-logging*, das selber nur eine Brücke

(Bridge) [32] zu anderen Implementierungen darstellt, ist die Verwandtschaft deutlich erkennbar (Schnittstelle `org.apache.commons.logging.Log` [20]). Die Delegation der LoggingManager-Funktionalität an *log4j* (Klasse `org.apache.log4j.Logger` [62]) oder die *Java Logging API* (Klasse `java.util.logging.Logger` [54]) ist ebenfalls ohne großen Aufwand möglich.

Die Grundidee aller dieser Lösungen ist, dass der Anwender die Stufe der Protokollierung (Loglevel) festlegt. Die Anwendung gibt Meldungen aller Stufen aus (Fehler, Warnungen etc.), aber im Protokoll erscheinen diese Meldungen nur, wenn mindestens die jeweilige Stufe eingestellt ist. Gibt die Anwendung also eine *Debug*-Meldung aus, während die Standard-*Error*-Stufe konfiguriert ist, wird die Meldung nicht protokolliert. Bei ConsensusFoundation wird die Protokollstufe mit dem Konfigurationsschlüssel `cf.loglevel` festgelegt (siehe Abschnitt A.3).

Wichtig ist, dass die Anwendung durch die Protokollausgaben nicht merklich verlangsamt wird. Wenn die Ermittlung der zu protokollierenden Werte aufwändig ist, sollten Sie diese Berechnung nur dann durchführen, wenn die Meldung auch tatsächlich im Protokoll erscheint (siehe Listing 16).

```
LoggingManager logging = cfm.getLoggingManager();
logging.debug("Einfache, zeitunkritische Debugausgabe");
if (logging.isDebugEnabled()) {
    String debugstr = ... Berechnung, die relativ viel Zeit kostet ...;
    logging.debug( debugstr );
}
```

Listing 16: Protokollierung von aufwändigen Aktionen

Der zweite Einsatzzweck der LoggingManagers ist die automatische Protokollierung von Ereignissen im Rahmenwerk (z.B. das Erzeugen und Löschen von Topics etc.), ohne dass dafür in den anderen Modulen spezielle Protokollaufrufe stattfinden müssen. Ansonsten müsste die `LoggingManager`-Schnittstelle deutlich mehr Methoden enthalten, um die einzelnen Ereignisse (oder passende Gruppierungen davon) unterscheiden zu können. Der Implementierungsaufwand für einen LoggingManager wäre deutlich höher, zumal nicht jede Implementierung alle Ereignisse verarbeiten wird.

Stattdessen registriert sich ein LoggingManager als Beobachter (Listener) an den Modulen, deren Ereignisse ihn interessieren. Sollen beispielsweise nur die Bewertungen protokolliert werden, wird nur der RatingManager beobachtet, wozu der LoggingManager zusätzlich die entsprechende Schnittstelle `RatingListener` implementieren muss.

Die Standard-Implementierung `LoggingManagerImpl` registriert sich an sämtlichen Modulen, die beobachtbar sind, implementiert die Methoden der Listener-Schnittstellen dann aber nur leer. Dadurch kann man von dieser Klasse eine Unterklasse ableiten[22] und gezielt die Methoden überschreiben, welche die relevanten Ereignisse signalisieren. Listing 17 zeigt dagegen, wie sich eine eigene LoggingManager-Implementierung als Beobachter am OntologyManager registriert[23], bei Erzeugung eines Topics benachrichtigt wird und die Daten von Topic und Erzeuger für die Protokollierung aufbereitet.

```
public class MyLoggingManagerImpl
        extends AbstractConsensusFoundationModule
        implements LoggingManager, OntologyListener {
    ...
    public void initCompleted(Properties config,
```

[22] Die Ausgabe der normalen Protokoll-Meldungen erfolgt bei der Standard-Implementierung auf `System.out`, was von Web-Servern normalerweise passend umgelenkt wird. Um die Ausgabe innerhalb einer `LoggingManagerImpl`-Unterklasse umzuleiten, muss nur die Methode `write()` passend überschrieben werden.

[23] Der Test auf `om!=null` ist strenggenommen nicht notwendig, da `ConsensusFoundation` die Initialisierung abbricht, wenn die drei grundlegenden Module — zu denen auch der OntologyManager gehört — nicht geladen werden können (siehe Abschnitt 4.4). Die Implementierungen sind aber nach Möglichkeit defensiv [50, Kapitel 4] kodiert.

```
                                        ConsensusFoundationManager cfm)
                                        throws Exception {
        super.initCompleted(config, cfm);
        OntologyManager om = cfm.getOntologyManager();
        if (om != null) {
            om.addOntologyListener( this );
        }
    }
    ...
    public void topicCreated(OntologyEvent e) {
        Concept topic     = e.getConcept();
        User    creator   = e.getUser();
        String  topicId   = topic.getId();
        String  topicName = topic.getName();
        String  username  = creator.getUsername();
        ... in DB oder in Textdatei speichern ...
    }
}
```

Listing 17: Protokollierung von Änderungen der Ontologie

Um die Standard-Implementierung durch einen solchen eigenen LoggingManager zu ersetzen, reicht es aus, den gewünschten Klassennamen in der ConsensusFoundation-Konfigurationsdatei unter dem Schlüssel `cf.loggingmanager` einzutragen (siehe Abschnitt A.3).

5 Auswertung

Nach Entwurf und Implementierung soll dieses Kapitel untersuchen, ob das ConsensusFoundation-Rahmenwerk sinnvoll als Grundlage für Applikationen genutzt werden kann. Dazu müssen zunächst alle Ziele des Entwurfs umgesetzt sein, d.h. das Rahmenwerk muss seine Aufgabe effektiv erledigen. Aus Sicht des Programmierers muss mit dem Rahmenwerk eine effizientere Entwicklung möglich sein. Vergleichbare Aufgaben müssen sich mit dem Rahmenwerk in weniger Zeit realisieren lassen als ohne das Rahmenwerk. Trotzdem muss die resultierende Anwendung eine mindestens ebenbürtige Robustheit und Performanz aufweisen. Außerdem — und dies war eine wesentliche Anforderung — müssen sich die Module des Rahmenwerks flexibel austauschen lassen, damit bei leicht veränderten Aufgabenstellungen nicht immer alle Komponenten des Systems angepasst oder sogar neu programmiert werden müssen.

5.1 Vorteile von ConsensusFoundation gegenüber einer reinen TM4J-Lösung

Um die Zeitersparnis gegenüber einer reinen TM4J-Lösung zu testen, sind eigentlich zwei vergleichbare Implementierungen nötig: Einmal auf Basis des ConsensusFoundation-Rahmenwerks und einmal ausschließlich unter Zuhilfenahme der TM4J-Bibliothek. Im gegebenen Zeitrahmen war dies jedoch nicht umsetzbar, weshalb nur die Beispiel-Anwendung *ConsensusFoundation Demo* auf Basis des Rahmenwerks realisiert wurde. Folgende generelle Überlegung verdeutlicht aber, warum die ConsensusFoundation-Schnittstellen einfacher und mit weniger Zeitaufwand zur Umsetzung der in Kapitel 2 identifizierten Anforderungen zu nutzen sind: Während sowohl TM4J als auch ConsensusFoundation Methoden zum Abfragen von Topics bieten, besitzt allein das Rahmenwerk Methoden zum einfachen Abfragen (und Hinzufügen) von Bewertungen (siehe Listing 18). Ebenso einfach lassen sich beispielsweise die Typen und Assoziationen eines Topics abfragen (und dort wiederum Bewertungen hinzufügen und abfragen), wobei ConsensusFoundation automatisch ermittelt, ob der angegebene Benutzer die abgefragten Konzepte überhaupt sehen darf. Bei TM4J müsste die Bewertungsverwaltung und die dynamische Rechteverwaltung für die Sichtbarkeit der Topics komplett von Hand geschrieben werden. Vermutlich würde man die entsprechenden Routinen in eine Bibliothek auslagern, um sie bei der nächsten TM4J-Anwendung wieder nutzen zu können — aber genau das bietet ja bereits ConsensusFoundation.

```
Topic t    = ...; // Topic-Abfrage weggelassen
User  usr = ...; // aktueller Benutzer

Collection alleBewertungen    = t.getRatings();
Rating     bewertungDurchUsr = t.getRating( usr );

Collection typen              = t.getTypes( usr );
Collection assoziationen      = t.getAssociations( usr );
```

Listing 18: Einfach zu nutzende Programmierschnittstellen

Im Folgenden sind daher also noch die Fragen zu klären, wie flexibel und austauschbar die ConsensusFoundation-Module sind, wie gut die Performanz der Implementierung ist und ob sich das Rahmenwerk zur Umsetzung einer vernünftigen Anwendungsarchitektur eignet.

5.2 Austauschbarkeit der Implementierungen

Eine der zentralen Anforderungen an das Rahmenwerk ist die flexible Austauschbarkeit seiner Komponenten. Dazu dürfen nicht zu viele Abhängigkeiten zwischen den Implementierungen bestehen. Abhängigkeiten zu den öffentlichen Schnittstellen hingegen stellen kein Problem dar — die Schnittstellen wurden ja gerade deshalb definiert, um von den Implementierungen zu abstrahieren und deren Abhängigkeiten untereinander zu minimieren.

Idealerweise existieren gar keine Abhängigkeiten zwischen den Implementierungen. Dies lässt sich aber oft nicht sinnvoll umsetzen, weil man dann sehr stark abstrahieren und zusätzliche Indirektionsstufen vorsehen müsste, was leicht zu Performanzeinbußen führen kann. Daher wird man realistischerweise versuchen, zumindest die häufig ausgetauschten Komponenten so unabhängig wie möglich zu gestalten. Welche Teile des Rahmenwerks werden vermutlich häufig ausgetauscht, welche weniger, und warum?

- Die Punktevergabe (ein oder mehrere ScoringListener) wird sehr häufig ausgetauscht werden, ebenso die dynamische Rechteverwaltung (der DynamicRightsManager) als Komponente, die Anreize umsetzen kann. Durch Variation gerade dieser beiden Aspekte soll in Zukunft genauer untersucht werden, wie man u.a. möglichst viele und möglichst wahrheitsgemäße Bewertungen von den Anwendern erhält.

- Das Anreizsystem selbst (der IncentiveManager) wird vermutlich seltener ausgetauscht werden — es sein denn, man will mit ganz anderen Anreizen als Punkten (und Rechten, die sich daraus ergeben) experimentieren.

- Das Bewertungssystem (der RatingManager) wird ebenfalls eher selten ausgetauscht werden, denn er kümmert sich nur um die Speicherung der Bewertungen, nicht aber um deren Interpretation. Letzteres ist Sache der jeweiligen Anwendung. Und mit der direkten SQL-Anbindung liegt eine effiziente Implementierung vor, die für die allermeisten Anwendungsfälle passend sein dürfte. Nur wenn die Bewertungen beispielsweise innerhalb der Ontologie (als eigene Topics) abgelegt werden sollen, müsste dieses Modul angepasst werden.

- Bei der Ontologie- und Schemaverwaltung (der OntologyManager inkl. der Implementierungen der Schema-Schnittstellen: Topic, Association, Attribute etc.) ist es Ziel des Entwurfs, dass sie so gut wie nie gewechselt werden muss, weil dies die grundlegende Funktionalität des ConsensusFoundation-Rahmenwerks darstellt. Diese sollte bezüglich der Schnittstellen und der Standard-Implementierung so universell gehalten sein, dass sie von allen Anwendungen eingesetzt werden kann.

- Wahrscheinlicher ist es, dass als Teil der Ontologie-Verwaltung zusätzliche Im- und Exportmodule benötigt werden, falls das standardmäßig eingesetzte XTM-Format nicht ausreicht.

- Die Benutzerverwaltung (der UserManager) muss angepasst werden, wenn man ConsensusFoundation-Anwendungen in bestehende Benutzerverwaltungen (z.B. mit LDAP) integrieren möchte.

- Der LoggingManager wird immer dann angepasst werden, wenn man die Aktionen des Systems auswerten möchte und dafür bestimmte Ausgabeformate benötigt (beispielsweise spezielle Textformate oder Datenbasisschemata).

- Der EvolutionManager wird sicher, wenn auch nur selten, ausgetauscht werden, weil die Standardimplementierung hier nur grundlegende Operationen bietet, die in Zukunft durch aufwändigere Algorithmen ersetzt werden sollen.

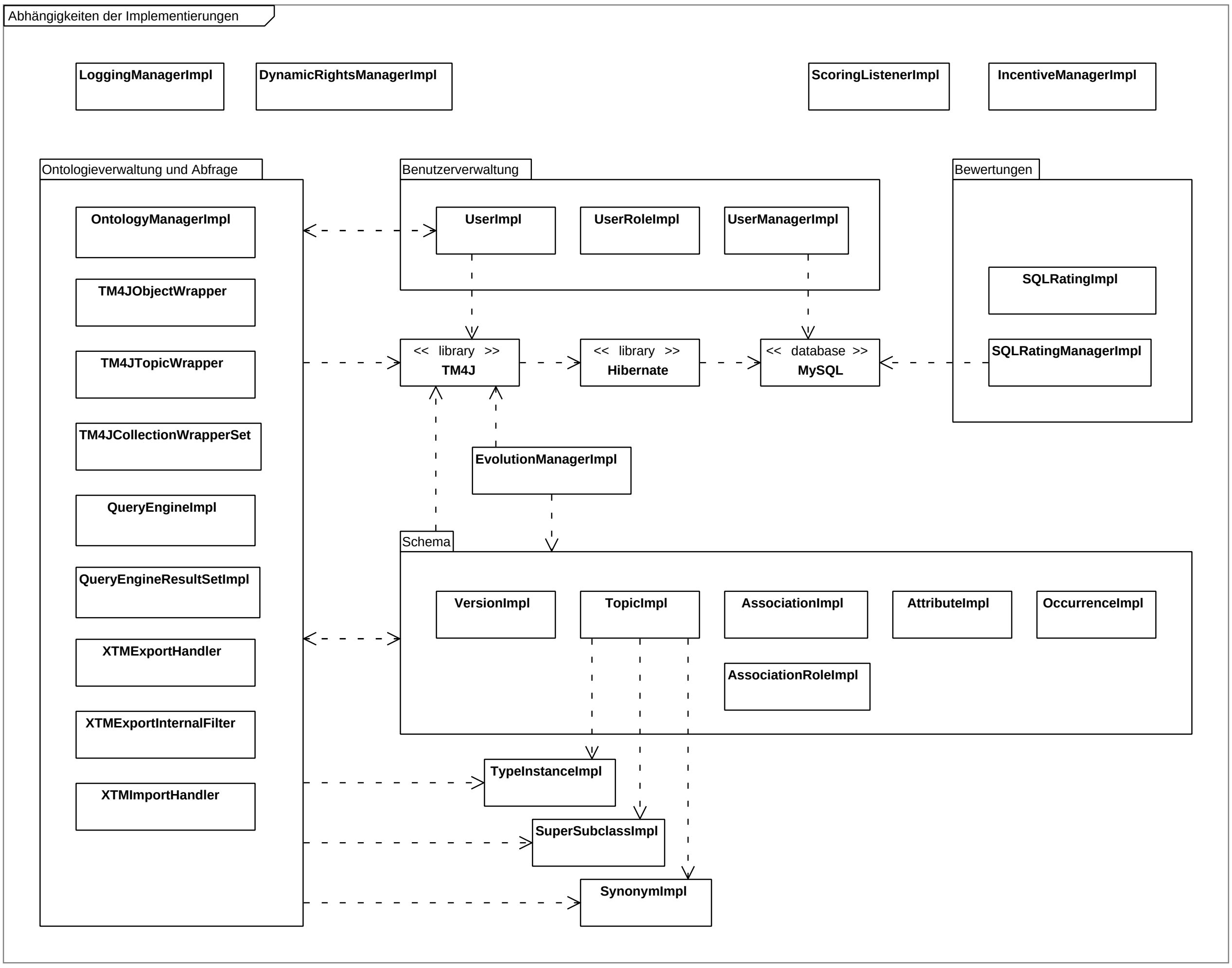


Abbildung 29: Abhängigkeiten der Implementierungen

Abbildung 29 zeigt die Implementierungsabhängigkeiten, die sich aus der ConsensusFoundation-Standardimplementierung ergeben. Innerhalb der gezeigten Gruppen (hier als Pakete dargestellt) sind beliebige Abhängigkeiten möglich. Zu beachten ist, dass bei der vorhandenen Implementierung nicht zwischen allen Klassen eines Pakets Abhängigkeiten bestehen, das Diagramm zeigt hier also eine etwas höhere Kopplung als tatsächlich umgesetzt. Zum einen wird damit die Übersichtlichkeit des Diagramms erhöht, und trotz stärker dargestellter Kopplung wird sich zeigen, dass die Abhängigkeiten kein Problem darstellen. Zum anderen könnten alternative Implementierungen den gezeigten Freiraum durchaus ausnutzen, um durch weitere tatsächliche Abhängigkeiten eine bessere Effizienz zu erreichen. Alle anderen Abhängigkeiten bestehen nur zu öffentlichen Schnittstellen!

Als erstes fällt auf, dass die Protokollierung (`LoggingManagerImpl`[24]), die Rechteverwaltung (`DynamicRightsManagerImpl`), das Anreizsystem (`IncentiveManagerImpl`) und die Punktevergabe (`ScoringListenerImpl`) keinerlei Abhängigkeiten zu anderen Implementierungen besitzen. Das ist insbesondere wichtig, weil diese Komponenten diejenigen sind, die — wie oben beschrieben — vermutlich am häufigsten ausgetauscht werden. Auch die Verwaltung der Bewertungen ist vollkommen unabhängig von der übrigen Implementierung, sie ist einzig von der gewählten Persistenzschicht (hier MySQL) abhängig.

[24]Diese und alle weiteren Klassen können im Paket `de.uka.ipd.consensus.impl` eingesehen werden.

Benutzer- und Ontologieverwaltung besitzen eine direkte Abhängigkeit, da User-Objekte zwar vom UserManager verwaltet werden, User andererseits aber auch Teil des Schemas sind. So werden die Benutzerdaten direkt mit SQL in einem Datenbankschema gespeichert und zusätzlich als TM4J-Topics abgebildet, damit sie (beispielsweise als Erzeuger eines Konzepts) in der Ontologie verfügbar sind. Die Abhängigkeit ist aber minimal und wird durch nur eine einzige Klasse der Benutzerverwaltung realisiert.

Die bewertbaren Assoziationshüllen (siehe Abschnitt 4.1) für Synonyme (`SynonymImpl`), Typ-Instanz-Beziehungen (`TypeInstanceImpl`) und Oberklassen-Unterklassen-Beziehungen (`SuperSubclassImpl`) gehören mit ins Schema und und passen bezüglich der Abhängigkeiten auch dorthin. Hier sind sie separat dargestellt, um zu betonen, dass die Abhängigkeiten nur von den anderen Klassen aus bestehen (zum Erzeugen und Erkennen) — die Hüllen selbst verlassen sich nur auf die öffentlichen Schnittstellen. Entsprechend sind weitere Hüllen (z.B. für Homonyme) später mit geringem Aufwand hinzufügbar.[25]

Die Abhängigkeiten zwischen der Ontologie- und Schemaverwaltung, dem Abfragemodul und der TM4J-Bibliothek ist sehr eng, aus einem einfachen Grund: Die Implementierungsdetails der TM4J-Bibliothek, von denen ConsensusFoundation bewusst abstrahiert, sollen trotzdem effizient verarbeitet werden. Für die praktische Anwendung stellt dies keine große Einschränkung dar, denn diese Teile sollten nur selten ausgetauscht werden. Aus demselben Grund wird auch `EvolutionManagerImpl` als von TM4J abhängig dargestellt. Prinzipiell ist zudem eine ausschließliche Abhängigkeit von den öffentlichen OntologyManager-Methoden denkbar (entsprechende Methoden stehen zur Verfügung), aber wiederum geht es um die Frage, wie effizient interne TM4J-Daten verarbeitet werden können und sollen.

Eine weitere Unabhängigkeit der Implementierungen könnte nur durch mehr Indirektion und Abstraktion erreicht werden. Es ist aber fragwürdig, ob dies sinnvoll ist. Es müssten dann nämlich interne Daten der zugrunde liegenden TM4J-Bibliothek als öffentliche Schnittstellen verfügbar gemacht werden, wodurch das Rahmenwerk zu sehr an diese konkrete Bibliothek gebunden wäre — oder alternative Implementierungen müssten später diese Schnittstellen umständlich nachbilden. Falls man die Abhängigkeiten dennoch weiter verringern möchte, ohne Einbußen bei der Effizienz zu erhalten, könnte eine alternative Implementierung die TM4J-Bibliothek komplett entfernen und das ConsensusFoundation-Schema direkt mit z.B. Hibernate persistieren.

Tabelle 2 fasst die vermutete Austauschhäufigkeit und die Abhängigkeiten innerhalb der Standardimplementierung noch einmal systematisch zusammen.

Komponente bzw. Modul	*Vermutliche Austauschhäufigkeit*	*Abhängig von der Implementierung*
Punktevergabe	++	**nein**
dyn. Rechteverwaltung	++	**nein**
LoggingManager	+	**nein**
Benutzerverwaltung	o	ja
Im- und Exportmodule	o	ja
EvolutionManager	o	ja
Anreizsystem	–	**nein**
Bewertungssystem	–	**nein**
Ontologie- und Schemaverwaltung	– –	ja

Tabelle 2: Vergleich der Austauschhäufig- und Implementierungsabhängigkeiten

[25] Hier wird deutlich, wie wichtig trotz der wenigen tatsächlichen Stellen, an denen Abhängigkeiten zu den Assoziationshüllen bestehen, eine vernünftige Werkzeug-Unterstützung ist. Eclipse beispielsweise findet mit Search→References→Project alle Verwendungen einer Klasse bzw. eines Typs innerhalb des Projekts.

Fazit: Die ConsensusFoundation-Standardimplementierung enthält sinnvolle und vor allem keine zyklischen Abhängigkeiten. Häufig ausgetauschte Komponenten sind komplett unabhängig, selten ausgetauschte Komponenten weisen zwar Abhängigkeiten auf, können dafür aber effizienter arbeiten. Die Umsetzung stellt einen gelungenen Kompromiss zwischen der gewünschten Flexibilität einerseits und einem zu hohen Abstraktionsgrad andererseits dar.

5.3 Laufzeitverhalten

Ein wichtiger Aspekt für den Praxiseinsatz ist das Laufzeitverhalten des Rahmenwerks. Für den Anwender, den die zugrunde liegende Technik nicht interessiert, ist dies neben der sichtbaren Benutzungsoberfläche der spürbare „Kontakt“ mit der Anwendung. Für den Softwarearchitekten, der sich für oder gegen das Rahmenwerk entscheiden muss, ist trotz gut zu benutzender Programmierschnittstellen elementar, dass das Laufzeitverhalten nicht wesentlich schlechter ist als beim direkten Zugriff auf eine andere, wenn auch schlechter anzuwendende Bibliothek.

Im Folgenden wird daher der Ontologie-Zugriff durch ConsensusFoundation mit dem direkten Zugriff durch die intern verwendete TM4J-Bibliothek verglichen. Zwei Anwendungsfälle werden dabei getestet[26]. **Fall 1:** Es wird eine bestimmte Anzahl zufällig ausgewählter Topics gelesen — dies tritt in der Praxis auf, wenn eine Web-Seite mehrere Topics anzeigen muss (z.B. ein Topic als Typ und alle seine Instanzen). **Fall 2:** Ein zufällig ausgewähltes Topic wird wiederholt gelesen. Dies ist relevant, wenn eine Abfolge von Aktionen auf dasselbe Topic zugreift, und tritt auf, wenn mehrere Anfragen (auch unterschiedlicher Nutzer) dasselbe Topic benötigen, beispielsweise einen Topic-Typ aus der Klassenhierarchie. Die gewählte Anzahl von jeweils 100 gelesenen Topics ist ein geschätzter Wert, der sich in der Praxis ergibt, wenn beispielsweise 10 Nutzer gleichzeitig Seiten mit jeweils 10 Topics (z.B. ein Topic mit seinen Ober- und Unterklassen) abrufen.

Folgende Erwartung besteht an den Test: Das ConsensusFoundation-Rahmenwerk sollte nicht deutlich mehr Zeit benötigen als der direkte TM4J-Zugriff, da das Rahmenwerk nur eine relativ dünne Hülle bzw. Fassade (Facade) [32] für TM4J darstellt. Auch wenn die meisten Funktionalitäten des OntologyManagers sich aus mehreren TM4J- und weiteren ConsensusFoundation-internen Aufrufen zusammensetzen (beispielsweise Typprüfung und dynamische Rechteprüfung), sollte dieser erhöhte Aufwand durch einen im Rahmenwerk implementierten Schattenspeicher (Cache) abgemildert werden.

Die ConsensusFoundation-Standardimplementierung realisiert den **Schattenspeicher** in der Klasse `de.uka.ipd.consensus.impl.TM4JObjectWrapper`. Damit der Schattenspeicher Objekte nach einer gewissen Zeit der Nichtbenutzung wieder freigibt, wird dazu eine `java.util.WeakHashMap` eingesetzt, deren Werte in `java.lang.ref.WeakReference`-Objekte eingehüllt sind [51]. Weil die Java-Klassenbibliothek zur Realisierung der `WeakHashMap` eine Streuwerttabelle (Hashtable) nutzt, kann bei wiederholtem Zugriff auf ein Objekt im Schattenspeicher normalerweise eine konstante Zugriffszeit ($O(1)$) erwartet werden. Nur für den schlimmsten Fall — wenn es zu sehr vielen Kollisionen der Streuwertschlüssel kommt — ist mit linearem Aufwand ($\Theta(n)$) zu rechnen [22, Kapitel 12]. Um dies zu verhindern, achtet Java darauf, dass der Belegungsfaktor (Load Factor) der Streuwerttabelle $0,75$ nicht überschreitet, ansonsten wird eine Restrukturierung (Rehashing) der Tabelle durchgeführt ([51] und [22, Seite 224]).

Konkret wird vom Test eine Ontologie in mehreren Schritten mit jeweils 50 Topics aufgefüllt, und nach jedem Schritt werden 100 zufällige Topics zunächst mit ConsensusFoundation, danach mit TM4J gelesen. Anschließend wird ein bestimmtes Topic 100 Mal gelesen. Das Ergebnis eines exemplarischen Testlaufs zeigt Abbildung 30. Testrechner ist ein iMac G5 2 Ghz mit 2 GB RAM und Mac OS X

[26] Das Laufzeitverhalten von Bewertungen wurde bereits in Abschnitt 4.7 untersucht.

10.4.6, kein dedizierter Server, die Software-Versionen sind in Abschnitt A beschrieben. Der Test kann mit der Klasse `de.uka.ipd.consensus.test.PerfTestServlet` nachvollzogen werden.[27]

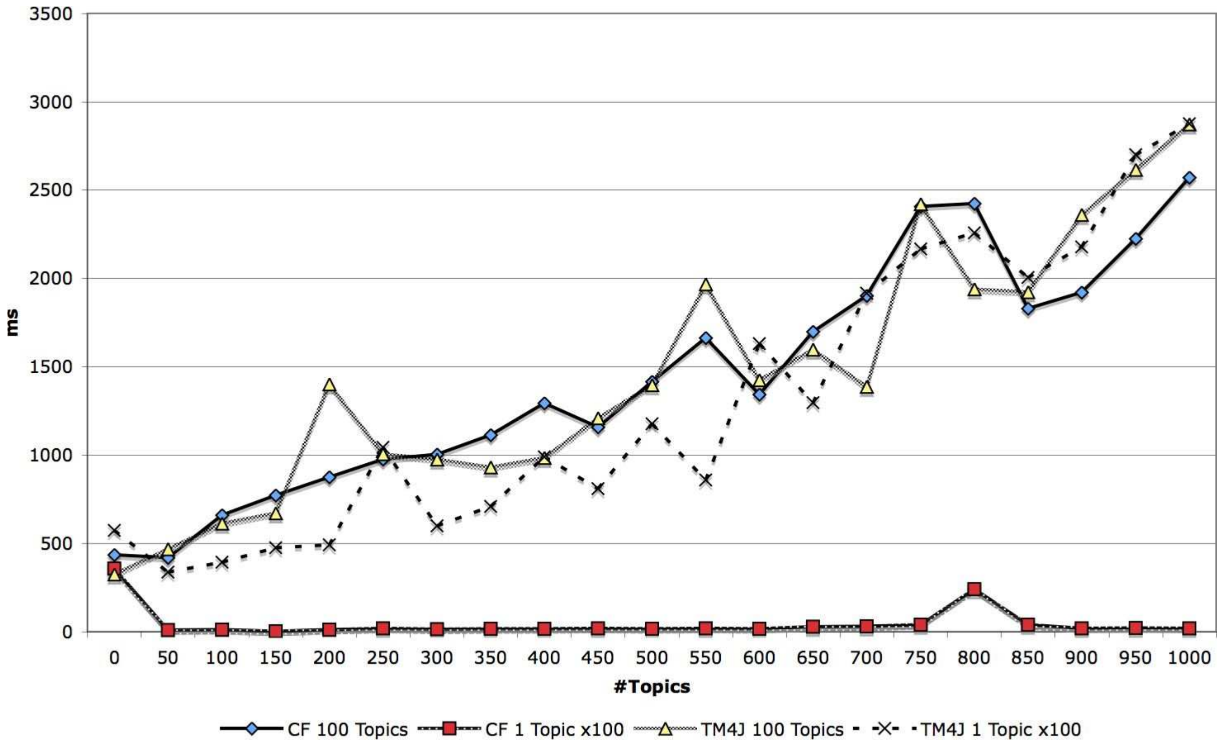


Abbildung 30: Performanzvergleich mit Hibernate-Backend und einer Gesamt-TA

Zwei Dinge fallen sofort ins Auge: Der Zugriff mit dem ConsensusFoundation-Rahmenwerk („CF 100 Topics“) ist wie erwartet nicht wesentlich schlechter als der direkte TM4J-Zugriff („TM4J 100 Topics“), manchmal sogar etwas besser. Beim wiederholten Zugriff („CF 1 Topic x100“) wirkt sich der Schattenspeicher extrem positiv aus. Unabhängig von der Anzahl der Topics hat man wie erwartet nahezu konstante, minimale Zugriffszeit. TM4J führt keine Pufferung durch, wodurch sich hier der wiederholte Zugriff („TM4J 1 Topic x100“) nur unwesentlich vom Lesen verschiedener Topics unterscheidet. Beide Fälle werden zusammen in Abbildung 30 betrachtet, um sowohl die Ebenbürtigkeit von ConsensusFoundation und TM4J bei der Abfrage verschiedener Topics als auch den deutlichen Vorteil des ConsensusFoundation-Schattenspeichers bei wiederholten Abfragen gegenüberzustellen.

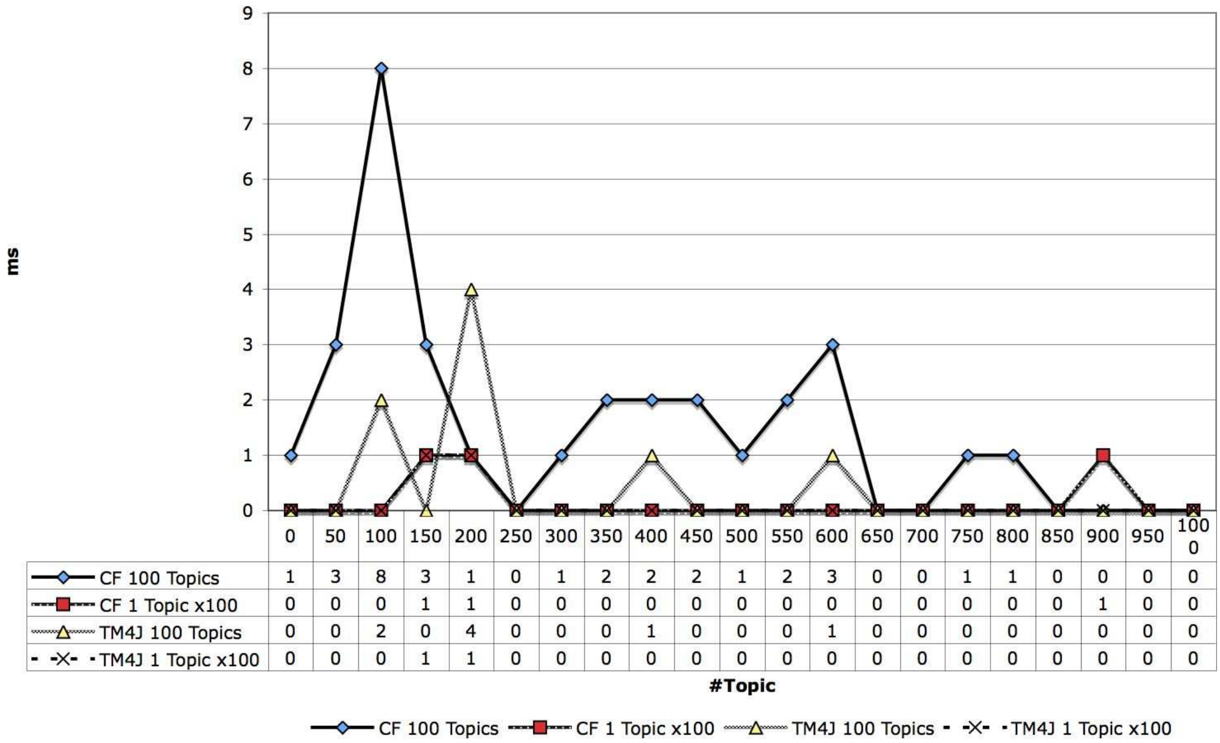
Gerade beim wiederholten Zugriff fallen zwei „Ausreißer“ auf. Am Anfang ist der gepufferte Zugriff nicht besser als die anderen Zugriffsarten, weil in einer leeren Ontologie überhaupt kein Topic gefunden wird, das zwischengespeichert werden könnte. Somit muss auch hier bei jedem Aufruf ein ungepufferter Zugriff erfolgen. Dieser Randeffekt ist aber für den praktischen Einsatz nicht von Bedeutung. Die Spitze bei 800 Topics, die bei unterschiedlichen Testläufen in ähnlicher Höhe, aber bei anderen Topic-Zahlen auftritt, lässt sich mit der automatischen Speicherbereinigung (Garbage Collection, GC) der Java Virtual Machine (JVM) erklären, die von Zeit zu Zeit für eine kurze Dauer aktiv wird [99] und dafür Rechenzeit benötigt, die hier gemessen wurde. Auch in den anderen Messreihen ist dieser Effekt enthalten.

Verglichen mit TM4J spricht also bis hier nichts gegen den Einsatz des ConsensusFoundation-Rahmenwerkes. Zwei Einflussgrößen sollen nun noch untersucht werden: Als Datenhaltungsschicht (das „Backend“) kann TM4J nicht nur Hibernate, sondern auch eine Speicher-Implementierung verwenden, die

[27]Sofern das Test-Servlet in `/WEB-INF/web.xml` aktiviert wird.

zwar nicht persistent ist, dafür aber einen deutlich schnelleren Zugriff bietet. Außerdem ist zu ermitteln, welche Kosten die Transaktionsverwaltung verursacht.

Um als Datenhaltungsschicht die TM4J-Speicher-Implementierung zu aktivieren, reicht es aus, in der ConsensusFoundation-Konfigurationsdatei zum Schlüssel `cfimpl.ontology.backend` den Wert `memory` einzutragen (siehe Abschnitt A.3). Für den praktischen Einsatz kann dieser Fall interessant sein, wenn die Geschwindigkeit des Systems absolut im Vordergrund steht und Abstriche bei der Dauerhaftigkeit der Daten in Kauf genommen werden können. Eine solche im Hauptspeicher gehaltene Ontologie könnte man beispielsweise durch einen automatischen Export in regelmäßigen Abständen persistieren. Das Ergebnis eines Testlaufs mit dem Memory-Backend zeigt Abbildung 31.



	0	50	100	150	200	250	300	350	400	450	500	550	600	650	700	750	800	850	900	950	1000
CF 100 Topics	1	3	8	3	1	0	1	2	2	2	1	2	3	0	0	1	1	0	0	0	0
CF 1 Topic x100	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
TM4J 100 Topics	0	0	2	0	4	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0
TM4J 1 Topic x100	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Abbildung 31: Performanzvergleich mit Memory-Backend

Während beim Hibernate-Backend nur der wiederholte Zugriff auf den Schattenspeicher eine kurze Zugriffszeit von ca. 20 ms lieferte, liegen nun alle Messwerte noch deutlich darunter — wie es beim Vergleich von Haupt- und Hintergrundspeicherzugriffen zu erwarten ist, stellen letztere doch einen herausragenden Leistungsengpass dar [61, Abschnitt 19.2.1]. Die direkten Zugriffe mit TM4J sind nun kaum noch messbar, ConsensusFoundation liefert nur wenig schlechtere Ergebnisse. Die sichtbaren Spitzen, die bei anderen Testläufen an unterschiedlichen Stellen auftreten, lassen sich wieder mit der automatischen Speicherbereinigung der JVM erklären. Denn während TM4J direkt auf seine Topic-Objekte zugreift, die als feste Referenz gehalten werden, muss ConsensusFoundation für jedes Topic, das sich nicht im Schattenspeicher befindet, ein Hüllobjekt (Wrapper) anlegen, auf das nur eine schwache Referenz im Schattenspeicher besteht. Die vielen nicht mehr benötigten Hüllobjekte werden dann von der Speicherbereinigung freigegeben. Dafür spricht auch, dass beim wiederholten Zugriff, bei dem nur jeweils ein Hüllobjekt angelegt werden muss, deutlich kleinere, mit dem direkten TM4J-Zugriff vergleichbare Spitzen auftreten.

Um die Verwendung von Transaktionen zu untersuchen, werden zwei Anwendungsfälle verglichen. Im ersten Fall wird das Lesen von jeweils 100 Topics zu einer Transaktion zusammengefasst. Dies entspricht der in Abschnitt 4.5.3 vorgestellten „Transaktion pro Anfrage (Request)“ und ist typisch für

Web-Applikationen.[28] Das Ergebnis hatten wir bereits in Abbildung 30 betrachtet, dort wurde genau das eben beschriebene Transaktionsverhalten eingesetzt. Im zweiten Fall wird vom Test überhaupt keine Transaktion begonnen, wodurch die aufgerufenen Methoden in TM4J selber dafür sorgen, dass für den jeweiligen Aufruf eine eigene Transaktion verwendet wird. Dieses Ergebnis zeigt Abbildung 32.

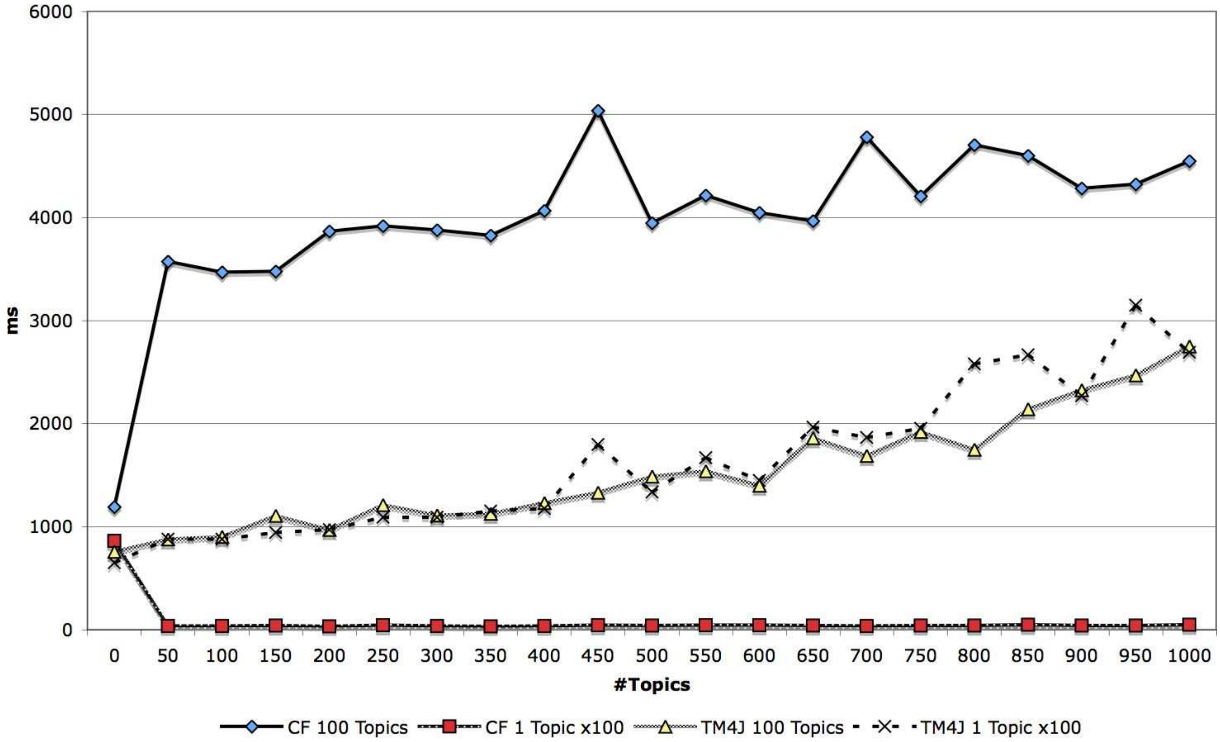


Abbildung 32: Performanzvergleich mit Hibernate-Backend und Einzel-TAs

Die direkten TM4J-Zugriffe sind bei den Einzeltransaktionen nun etwas langsamer als bei der zusammengefassten Transaktion, allerdings nur unwesentlich. Das Verwenden vieler Transaktionen ist also offensichtlich zeitmäßig nicht sehr teuer, zumal wir hier nur lesend zugreifen und beim *Commit* nichts aufwändig festgeschrieben werden muss. Bei ConsensusFoundation sieht es dagegen anders aus: Während der Schattenspeicher-Zugriff wie gewohnt sehr schnell erfolgt, sind die ungepufferten Rahmenwerk-Zugriffe nun deutlich langsamer. Warum?

Sofern keine umspannende Transaktion aktiv ist, verwendet TM4J für Hibernate-Sessions (den Gültigkeitsbereich von Objekt-Identitäten) und -Transaktionen dieselbe Granularität [9, Abschnitt 5.2.2] — den jeweiligen Methodenaufruf. Die Abfrage eines Topics direkt mit TM4J besteht aus einem TM4J-Methodenaufruf und kostet dementsprechend eine Session und eine Transaktion. Die Abfrage eines Topics über ConsensusFoundation setzt sich dagegen aus mehreren TM4J-Aufrufen zusammen, da das Rahmenwerk nach der eigentlichen Abfrage des Topics noch Typprüfungen durchführen muss, weil auch User- und Attribute-Objekte intern auf TM4J-Topics abgebildet werden. Für jede Typprüfung muss nun aber eine neue Session und eine neue Transaktion aufgebaut werden. Und da eine neue Session bedeutet, dass die alte Topic-Objekt-Identität ungültig geworden ist, muss ein neues Topic-Objekt ausgeprägt werden. Dies kommt oft einem erneuten Lesezugriff gleich [9, Abschnitt 5.3.2], was messbar Zeit kostet.

[28] Zur einfachen Umsetzung der Transaktion pro Anfrage setzt die Beispiel-Anwendung den Servlet-Filter [23, Abschnitt SRV.6] `de.snailshell.consensus.demo.ConsensusFoundationFilter` ein, der jede Struts-Action-Anfrage (`*.do`) als Transaktion ausführt, ähnlich wie in [9, Seite 304] vorgeschlagen. Auch wenn dieser konkrete Filter Abhängigkeiten in die Beispielanwendung aufweist, kann er leicht an andere Servlet-basierte Web-Applikationen angepasst werden.

Ist dagegen wie im ersten Test eine umspannende Transaktion aktiv, kann sowohl die Abfrage des Topics als auch die Typprüfung innerhalb derselben Session stattfinden, wodurch die Topic-Objekt-Identität gültig bleibt und kein erneuter Lesezugriff notwendig wird.

ConsensusFoundation verlässt sich an dieser Stelle auf TM4J und steuert Hibernate nicht direkt an, um die Abhängigkeiten zu den verwendeten Bibliotheken nicht zu erhöhen. Natürlich könnte das Rahmenwerk für diesen Fall noch optimiert werden und selber bei Bedarf umspannende TM4J-Transaktionen anlegen. Das würde allerdings die Komplexität der Implementierung unnötig für einen Anwendungsfall erhöhen, der in konkreten Applikationen so gut wie nie vorkommt — hier wird in aller Regel mit Transaktionen auf Anwendungsebene gearbeitet. Und genau dieser Normalfall lässt sich, wie weiter oben gesehen, problemlos mit ConsensusFoundation realisieren und bietet gute Performanz.

Der letzte Test zeigt also vor allem, dass die von TM4J abhängige ConsensusFoundation-Standardimplementierung nicht ohne sinnvolle Transaktionen auf Anwendungsebene eingesetzt werden sollte. Dies stellt aber keine wesentliche Einschränkung dar, und der OntologyManager bietet passende Methoden zur Transaktionssteuerung (siehe Abschnitt 4.5.3).

Unter Praxis-Gesichtspunkten (Hibernate-Backend, von der Anwendung gesteuerte Transaktionen pro Anfrage) spricht also alles für den Einsatz des Rahmenwerks — man erreicht gute Performanz trotz komfortabler Programmierschnittstellen.

5.4 Realisierung der Mehrschicht-Architektur

Ziel des Entwurfs war es, eine solide (strikte, aber nicht zwingend lineare [8, Seite 696-697]) Mehrschicht-Architektur zu schaffen, bei der die höheren Schichten auf die darunter liegenden zugreifen, aber aus den tieferen Schichten keine Abhängigkeiten in die höheren Schichten bestehen. Web-Anwendungen sollten sich nach dem „Model/View/Controller“-Paradigma entwickeln lassen, wobei das ConsensusFoundation-Rahmenwerk ausschließlich in der Model-Schicht angesiedelt sein sollte (siehe Abschnitt 3.6). Die Architektur der ConsensusFoundation-Standardimplementierung und der Beispiel-Anwendung zeigt Abbildung 33.

Der Aufbau des Rahmenwerks und die Abhängigkeiten der Implementierung wurden bereits in Kapitel 4 und in Abschnitt 5.2 besprochen: Die Standardimplementierung nutzt MySQL für die Benutzer- und Bewertungsverwaltung, zur Verwaltung der Ontologie wird auf TM4J zurückgegriffen. Insbesondere bestehen keine Abhängigkeiten in die Steuerungs- oder Darstellungsschicht, das Rahmenwerk weist keine Bindung an bestimmte Bibliotheken (Struts) oder Technologien (JSP) auf. Da ConsensusFoundation auch überhaupt keine Funktionalität für diese Schichten anbietet, ist die Umsetzung des Rahmenwerks ausschließlich für die Model-Schicht demnach gelungen.

Die Frage ist nun aber, ob das Rahmenwerk von den darüber liegenden Schichten, insbesondere von der Steuerungsschicht, praktikabel eingesetzt werden kann. Daher wird im Folgenden kurz untersucht, wie exemplarische Zugriffe von der Steuerungsschicht auf das Rahmenwerk und von der Darstellungs- auf die Steuerungsschicht erfolgen — und was dabei gegebenenfalls zu beachten ist. Dazu wird wieder die auf Struts und JSP basierende Beispiel-Anwendung betrachtet.

Wie sich die Steuerungsschicht Zugriff auf das Rahmenwerk verschafft, hat bereits Listing 7 gezeigt. Ein `ConsensusFoundation`-Exemplar wird initialisiert und eine Referenz darauf im Servlet-Kontext abgelegt, wo sie später wieder von den Klassen der Steuerungsschicht abgefragt werden kann. Wenn nun der Benutzer in der Web-Anwendung eine Aktion auslöst (beispielsweise einem Verweis folgt, um Details zu einem bestimmten Topic abzufragen), wird Struts die `execute()`-Methode in der zugeordneten `Action`-Klasse ausführen. In dieser Methode muss nun der Zugriff auf ConsensusFoundation erfolgen, die Daten müssen abgefragt, gegebenenfalls aufbereitet und an die JSP zur Darstellung weitergeleitet werden (siehe Listing 19).

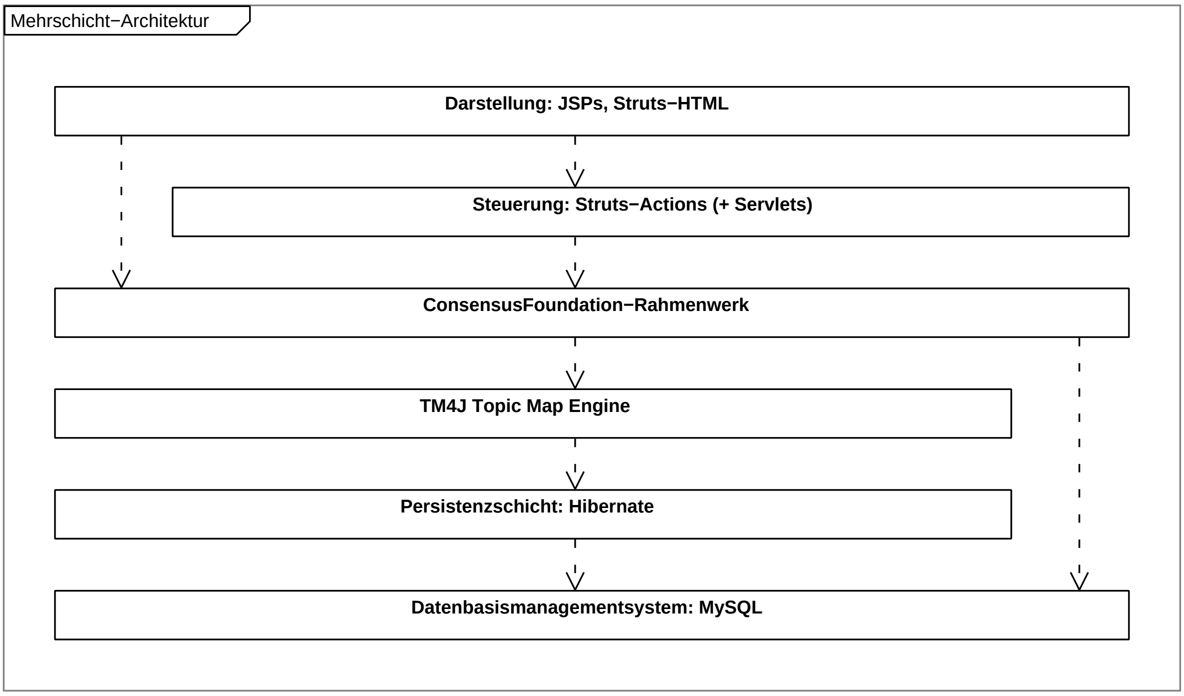


Abbildung 33: Umsetzung der Mehrschicht-Architektur

```
package de.snailshell.consensus.demo.actions;
...
public class TopicViewAction extends BasisAction {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
                HttpServletRequest request, HttpServletResponse response)
                    throws Exception {
        String           id   = request.getParameter( "id" );
        User             user = this.getUser( request );
        OntologyManager om    = this.getOntologyManager();
        ...
        Topic      t     = om.getQueryEngine().getTopicById( user, id );
        Collection types = t.getTypes( user );
        ...
        request.setAttribute( Constants.REQUEST_ATTR_TOPIC, t );
        request.setAttribute( Constants.REQUEST_ATTR_TOPICTYPES, types );
        return mapping.findForward( Constants.MAPPING_SUCCESS );
    }
}
```

Listing 19: Abfrage eines Topics in der Steuerungsschicht

Alle Aktionsklassen der Web-Anwendung erben von einer geeignet implementierten Basisklasse, die einen einfachen Zugriff auf alle ConsensusFoundation-Module sowie auf den angemeldeten Benutzer der aktuellen Sitzung bietet.[29] Man sieht im Listing gut, wie zunächst die Anfrage aus der Darstel-

[29]Zu diesem Zweck erbt `de.snailshell.consensus.demo.actions.BasisAction` nicht nur von `org.apache.struts.`

lungsschicht ausgewertet wird (Zeilen 8 und 9). Mit dem aktuellen Benutzer (User) wird anschließend das gewünschte Topic anhand seiner Kennung (ID) und danach alle Typen dieses Topics vom Rahmenwerk abgefragt (Zeilen 12 und 13)[30]. Zum Schluss werden die ermittelten Daten als HTTP-Request-Attribute gespeichert (Zeilen 15 und 16), wo sie von der Darstellungsschicht gelesen werden können, und Struts wird angewiesen, die Anfrage an die konfigurierte JSP-Seite zur Ausgabe weiterzuleiten (Zeile 17).

Als Darstellungsschicht verwendet die Beispiel-Anwendung *JavaServer Pages* (JSP). Neben dem üblichen HTML-Code finden sich darin auch XML-Elemente („Tags") der *JavaServer Pages Standard Tag Library* (JSTL) und Ausdrücke der zugehörigen *Expression Language* (EL) [10], mit denen die von der Steuerungsschicht erhaltenen Daten für die Anzeige aufbereitet, d.h. in HTML-Code umgewandelt, werden (siehe Listing 20).

```
<%@ page isELIgnored="false" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"  prefix="c" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
...
<h1>Topic "${requestScope.Topic.name}"</h1>
<p>Typen dieses Topics:</p>
<c:choose>
    <c:when test="${not empty requestScope.TopicTypes}">
        <ul>
            <c:forEach items="${requestScope.TopicTypes}" var="i">
                <li>
                    <html:link action="topicview?id=${i.type.id}">
                        ${i.type.name}
                    </html:link>
                    <c:if test="${i.ratingsCount > 0}">
                        <jsp:include page="/WEB-INF/content/rating.jsp">
                            <jsp:param name="wert"
                                       value="${i.averageRatingValue}"/>
                        </jsp:include>
                    </c:if>
                </li>
            </c:forEach>
        </ul>
    </c:when>
    <c:otherwise>
        <p>Dieses Topic ist typenlos.</p>
    </c:otherwise>
</c:choose>
```

Listing 20: Darstellung eines Topics in einer JSP

Die Steuerungsschicht hatte das Topic-Objekt als Request-Attribute gespeichert, daher kann die JSP nun mit dem Ausdruck `${requestScope.Topic}` das Topic aus dem HTTP-Request auslesen. Der Name des Request-Attributs, hier „Topic", muss dazu der JSP bekannt sein. Wenn nun `name` als

`action.Action`, sondern implementiert auch die in Abschnitt 4.4 vorgestellte Schnittstelle `ConsensusFoundationManager`. `BasisAction` kümmert sich dann darum, die `ConsensusFoundation`-Referenz aus dem Servlet-Kontext auszulesen.

[30] Das `User`-Objekt ist hier notwendig, damit die dynamische Rechteverwaltung (siehe Abschnitt 4.9) entscheiden kann, ob der aktuelle Benutzer das Topic überhaupt sehen darf. Die Abfrage, ob ein Topic erfolgreich gelesen werden konnte, wurde in diesem Beispiel der Übersichtlichkeit wegen weggelassen.

Attribut des Topic-Objekts gelesen wird (Zeile 5), wird dieser Zugriff von der Expression Language automatisch in den Aufruf der Topic-Methode `getName()` umgewandelt.

In dem Ausdruck `${requestScope.Topic.name}` findet also sowohl ein Zugriff auf Daten der Steuerungsschicht als auch ein direkter Aufruf des Rahmenwerks in der Model-Schicht statt. Auch bei den Bewertungen der Topic-Typen, die in den Zeilen 10-22 in einer JSTL-Schleife ausgegeben werden, erfolgt mit `ratingsCount` und `averageRatingValue` ein Aufruf der entsprechenden Methoden im Model-Objekt. Dies ist hier möglich, weil die Steuerungsschicht zum einen die aus dem Modell abgefragten Daten direkt an die Darstellungsschicht weiterreicht, und weil zum anderen die hier benutzten Methoden keine Parameter erwarten. Die Methoden entsprechen somit der *JavaBeans*-Spezifikation für Schreib- (Setter) und Lesemethoden (Getter) [40, Abschnitt 7.1], was die Voraussetzung dafür ist, dass sie in der Expression Language genutzt werden können.

Nun kann es aber einerseits unerwünscht sein, dass die Darstellungsschicht direkt Daten und Methoden der Model-Schicht verwendet. Beispielsweise kann der Aufruf des Modells relativ kostspielig sein, und wenn zur Darstellung einer Seite ein Objekt mehrfach abgefragt werden muss, würden die Kosten durch diese Aufrufe vervielfacht. Andererseits ist es oft technisch gar nicht möglich, Methoden des Modells aufzurufen, weil sie Parameter benötigen und dies eventuell — wie hier durch die JSTL-EL — nicht unterstützt wird.[31] In beiden Fällen wird man Datentransferobjekte (DTO) einsetzen, auch Wertobjekte (Value Objects) genannt ([88, Abschnitt 5.7.3], siehe auch das J2EE-Entwurfsmuster *Transfer Object* [21]). Solche Datentransferobjekte werden in der Steuerungsschicht erzeugt, mit den Daten des Modells gefüllt und anstelle der Originaldaten an die Darstellungsschicht weitergegeben. Dort können ihre Daten dann mit gewöhnlichen Gettern und Settern abgefragt werden. Das Ergebnis ist in beiden Fällen eine striktere Trennung der beiden oberen Schichten.

Listing 21 zeigt ein DTO, das in der Beispiel-Anwendung eingesetzt wird, um zu einer Assoziation alle enthaltenen Rollen samt Spielern anzuzeigen. In einer JSP können mit der JSTL die Spieler einer Rolle nicht abgefragt werden, da ein `User`-Objekt als Parameter übergeben werden muss. Nachdem aber die Struts-Aktion[32] stellvertretende `RoleBean`-Objekte erzeugt hat, kann die JSP problemlos auf deren Attribute `role` und `topics` (die Spieler der Rolle) zugreifen.

```
package de.snailshell.consensus.demo;
...
public class RoleBean {
    private AssociationRole role;
    private Collection      topics;

    public RoleBean(User user, AssociationRole role) {
        this.role   = role;
        this.topics = role.getTopics( user );
    }

    public AssociationRole getRole() {
        return this.role;
    }

    public Collection getTopics() {
```

[31] Man kann in JSPs sogenannte „Scriptlets“ (kurze Java-Codeblöcke) integrieren, die dann auch einen Methodenaufruf mit Parametern ermöglichen. Dadurch werden aber die oberen beiden Schichten vermischt, wovon aus Gründen der Wartbarkeit dringend abzuraten ist — Scriptlets sind daher schon lange nicht mehr Stand der Technik (siehe auch [88, Abschnitt 4.2.6.8]).

[32] In diesem Fall die Klasse `de.snailshell.consensus.demo.actions.AssociationViewAction`

```
        return this.topics;
    }
}
```

Listing 21: DTO zur Entkopplung von Model- und Darstellungsschicht

Genau genommen dürfte die `RoleBean` die Topics nicht, wie hier geschehen, direkt speichern, sondern müsste deren Werte in eigene Datentransferobjekte umkopieren, damit Darstellungs- und Model-Schicht vollständig entkoppelt sind. In der Praxis wird eine solche nicht strikte Trennung allerdings oft in Kauf genommen, wenn dadurch keine Nebenwirkungen zu erwarten sind (z.B. zu hohe Kosten bei entfernten Zugriffen oder Preisgabe von Daten, die nicht für die Darstellungsschicht bestimmt sind) — man spart sich dadurch eine Indirektionsstufe und das Erzeugen vieler Datentransferobjekte.

Alles in allem ist eine saubere Trennung der Schichten realisierbar, und die Steuerungsschicht kann einfach auf das ConsensusFoundation-Rahmenwerk zugreifen. Das Rahmenwerk ist für die darüber liegenden Schichten an keine bestimmte Bibliothek oder Technologie gebunden, wodurch sich ConsensusFoundation leicht zusammen mit anderen Rahmenwerken (beispielsweise *Spring* [57] oder *JavaServer Faces* (JSF) [12]) einsetzen lässt.

5.5 Fazit

Die Schnittstellen des ConsensusFoundation-Rahmenwerks und deren Standardimplementierung erfüllen die Anforderungen der Analyse und die Ziele des Entwurfs. Die angebotenen Schnittstellen wurden bereits in Kapitel 4 vorgestellt und decken die gestellten Anforderungen ab, wobei auf eine einfache Benutzbarkeit geachtet wurde (beispielsweise zur Abfrage der Bewertungen). Die technischen Rahmenbedinungen bezüglich der eingesetzten Fremdsoftware und der Versionen sind ebenfalls erfüllt.

Zur Austauschbarkeit der Module trägt bei, dass Abhängigkeiten vor allem zu den öffentlichen Schnittstellen bestehen, nicht zur konkreten Implementierung. Komponenten, die vermutlich häufig ausgetauscht werden, sind vollkommen unabhängig von den anderen Implementierungen (Punktevergabe, Anreizsystem, dynamische Rechteverwaltung, Logging). Selten ausgetauschte Komponenten weisen zwar Abhängigkeiten auf, können dadurch aber effizienter arbeiten (Ontologie-Verwaltung). Die Umsetzung stellt einen gelungenen Kompromiss zwischen der gewünschten Flexibilität einerseits und einem zu hohen Abstraktionsgrad andererseits dar.

Die Performanz des Rahmenwerks ist trotz zusätzlicher Funktionalität und einer einfacher zu nutzenden Programmierschnittstelle nicht schlechter als die direkte Nutzung der zugrunde liegenden TM4J-Bibliothek — ein wichtiges Resultat bei der Entscheidungsfindung, ob ConsensusFoundation als Grundlage eines Projektes eingesetzt werden soll. Einzige Voraussetzung dafür ist, dass die Anwendung Transaktionen mit den vom Rahmenwerk zur Verfügung gestellten Methoden mit einer sinnvollen Granularität verwendet (beispielsweise auf Anfrage-Ebene), was aber in jeder praxisrelevanten Anwendung gegeben sein sollte.

Eine Mehrschicht-Architektur mit sauberer Trennung der Schichten ist problemlos realisierbar. ConsensusFoundation ist so implementiert, dass es auf die Model-Schicht beschränkt bleiben und die Steuerungsschicht einfach auf die Methoden des Rahmenwerks zugreifen kann.

Die Beispiel-Anwendung *ConsensusFoundation Demo* zeigt die Einsetzbarkeit des Rahmenwerks in einer mehrbenutzerfähigen Web-Applikation. Allerdings ist diese Anwendung darauf ausgelegt, die technischen Möglichkeiten des Rahmenwerks zu demonstrieren. Praktische Anwendungen werden sicherlich nicht alle Funktionalitäten verwenden oder diese zumindest teilweise vor dem Nutzer verbergen, um eine zum jeweiligen Anwendungsbereich passende Benutzungsoberfläche anzubieten. Die Beispiel-Anwendung kann dann aber immer noch als Vorlage für solche Eigenentwicklungen auf Basis des Rahmenwerks genutzt werden.

6 Zusammenfassung und Ausblick

In dieser Diplomarbeit wurde die Notwendigkeit eines wartbaren und flexibel erweiterbaren Java-Rahmenwerks zur kooperativen und anreizbasierten Erstellung von Ontologien mithilfe von Bewertungen aufgezeigt. Die dafür notwendigen Anforderungen wurden analysiert, der Entwurf und die Implementierung diskutiert. Die Auswertung hat gezeigt, dass das Ergebnis *ConsensusFoundation* die Anforderungen erfüllt.

In Kapitel 2 wurde zunächst beschrieben, was Wissen ist und wie es mit Ontologien repräsentiert werden kann. Hierzu wurden die drei Repräsentationsformen RDF(S), OWL und Topic Maps vorgestellt. Für ConsensusFoundation fiel die Entscheidung zugunsten der Topic Maps, weil diese Form besser für die menschliche Verarbeitung geeignet ist als RDF(S) und OWL. Anschließend wurden bestehende Bewertungssysteme analysiert und damit die diesbezüglichen Anforderungen für ConsensusFoundation festgelegt. Hierbei fiel auf, dass die vorhandenen Systeme zur Ontologie-Verwaltung generell keine Bewertungen unterstützen bzw. dass ein bestehender Prototyp zwar Bewertungen ermöglicht, aber nicht sinnvoll wartbar, wiederverwendbar und erweiterbar ist, woraus sich der Wunsch nach einer softwaretechnisch besseren Lösung ergibt.

Kapitel 3 hat daher die Kriterien beschrieben, wann Entwurf und Implementierung aus Sicht der Software-Technik und -Architektur qualitativ gut sind. Ziel muss der Entwurf geeigneter Schnittstellen sein, um die Module des Systems unabhängig von den anderen Implementierungen zu machen und somit besser zu entkoppeln. Neben der Abstraktion und der Kapselung ist vor allem die Modularität des Systems wichtig. Man möchte eine möglichst lose Kopplung und hohe Kohäsion erreichen, was letztendlich in einer guten Wiederverwendbarkeit, Wartbarkeit und Erweiterbarkeit resultiert. Um die Begriffe zu klären, wurden anschließend die grundlegenden Ideen hinter und Eigenschaften von Entwurfsmustern, Komponenten, Klassenbibliotheken und Rahmenwerken vorgestellt. Die Überlegungen zu ConsensusFoundation sprachen danach für eine Mischform aus objektorientiertem und komponentenbasierten Rahmenwerk, wie es in der Praxis häufig anzutreffen ist. Schließlich wurde noch die Mehrschichtarchitektur behandelt. Als Ziel wurde festgelegt, dass ConsensusFoundation nur in der Fachkonzeptschicht bzw. im Model (beim MVC-Muster) angesiedelt sein soll und keine Abhängigkeiten in höhere Schichten aufweisen darf.

Der Entwurf des ConsensusFoundation-Rahmenwerks wurde in Kapitel 4 zusammen mit ausgewählten Eigenschaften der Standard-Implementierung besprochen. Dazu wurde zuerst das Entwurfs-Schema, also das Datenmodell für ConsensusFoundation-Ontologien, entwickelt und die Möglichkeiten zur Klassifikation der Topics aufgezeigt. Nach den technischen Rahmenbedingungen — aus denen sich auch ergab, sinnvollerweise die TM4J-Bibliothek zur Verwaltung der Topic Map einzusetzen — wurden die einzelnen Komponenten des Rahmenwerks beschrieben. Die zentrale Verwaltungs-Komponente initialisiert die Anwendung, d.h. auch die übrigen sieben Komponenten, und bietet zur Laufzeit einen einfachen Zugriff auf alle Module. Die restlichen Komponenten, vom Ontology- bis zum LoggingManager, wurden anhand ihrer UML-Diagramme vorgestellt. Die Gründe für den jeweiligen Entwurf wurden dargelegt, ohne allzu sehr auf die Details der Programmierung einzugehen — hierfür kann auf die Javadoc-Dokumentation zurückgegriffen werden. Wenn es sich anbot, wurde aber mit kurzen Quelltextbeispielen die Nutzung des Rahmenwerks demonstriert, beispielsweise Initialisierung und Ende einer Anwendung, Bewertungs-, Punkte- und dynamische Rechtevergabe sowie die gezielte Protokollierung bestimmter Aktionen.

Die Auswertung des Rahmenwerks und seiner Standard-Implementierung erfolgte in Kapitel 5. Demnach sind die Schnittstellen gut und einfach nutzbar, die Implementierungen der Module sind einfach austauschbar, der Konfigurationsaufwand für den Austausch ist gering. Es gibt wenige, sinnvolle Implementierungsabhängigkeiten, ansonsten bestehen Abhängigkeiten nur zu den Schnittstellen — die

Module des Rahmenwerks sind erfreulich lose gekoppelt. Das Laufzeitverhalten ist beim In-Memory-Backend wie zu erwarten hervorragend und beim Hibernate-Backend absolut akzeptabel. ConsensusFoundation erreicht dies durch geeignete Pufferstrategien, um nicht langsamer (und teilweise sogar schneller) als die zugrunde liegende Bibliothek zu sein. Eine sauber getrennte Mehrschichtarchitektur ist mit ConsensusFoundation problemlos realisierbar, das Rahmenwerk als Teil des Fachkonzepts besitzt keine Abhängigkeiten zu höheren Schichten. Der Zugriff von dort auf das Rahmenwerk ist aber denkbar einfach, Web-Applikationen lassen sich also mit ConsensusFoundation problemlos realisieren.

Die Beispiel-Anwendung *ConsensusFoundation Demo* zeigt die generelle Einsetzbarkeit des Rahmenwerks in einer Web-Applikation, konzentriert sich dabei aber auf die eher technische Demonstration aller Möglichkeiten. Derzeit entsteht aufbauend auf ConsensusFoundation bereits eine weitere Anwendung, die den Fokus verstärkt auf eine Benutzungsoberfläche legen wird, die für normale Anwender geeignet ist.

Ob das Rahmenwerk alle Erwartungen erfüllen kann, wird sich erst bei seiner Verwendung zeigen. Wie [83, Seite 395] schreibt, entstehen Rahmenwerke (Frameworks) *nicht von heute auf morgen und sind das Ergebnis eines längeren Entwicklungsprozesses.* Auch Rahmenwerke müssen gewartet und weiter entwickelt werden, wenn Erfahrungen mit dem Einsatz des Rahmenwerkes gewonnen wurden und Änderungen sinnvoll erscheinen. Die Anpassungen erfolgen dabei iterativ und inkrementell [83, Seite 404]. Beispielsweise könnten die ConsensusFoundation-Schnittstellen mittelfristig auf generische Datentypen umgestellt werden, wenn sich Java EE 5 auf den Enterprise-Systemen etabliert haben wird. Außerdem könnte man die Abhängigkeit von Fremdbibliotheken wie TM4J verringern und das Datenmodell von ConsensusFoundation direkt mit Hibernate persistieren. Erweiterungen wird es vermutlich bei der Evolution von Ontologien geben, wenn bei der Nutzung weitere Anwendungsfälle als die derzeit vorgesehenen auftauchen.

Bis dahin wird das bestehende Rahmenwerk samt Implementierung als solide erste Version genutzt werden können. Es zeigt sich gut geeignet für den Einsatz in heutigen Java-Enterprise-Systemen und erfüllt die gestellten Anforderungen.

A Das Eclipse-Projekt und die Beispiel-Anwendung

Dieses Kapitel gibt zum Schluss nun noch einige kurze technische Hinweise, wie man das Consensus-Foundation-Rahmenwerk und die Beispiel-Anwendung auf seinem eigenen Rechner nutzen und gegebenenfalls anpassen kann. Beides wurde in der Entwicklungsumgebung „Eclipse“ realisiert, daher wird im folgenden Abschnitt A.1 zunächst die Projektstruktur vorgestellt und dann stichpunktartig eine Arbeitsanweisung gegeben, wie man die Beispiel-Anwendung zum Laufen bekommt. Getestet wurde dies unter Mac OS X 10.4, Windows XP SP2 und SUSE Linux 10.0.

In den weiteren Abschnitten werden die Konfigurationsdateien und die verwendeten Bibliotheken bzw. die genutzte Fremdsoftware beschrieben. Der Fokus liegt dabei darauf, welche Versionen eingesetzt werden. Verweise zum Herunterladen dieser Versionen sowie die ConsensusFoundation-Quelltexte finden sich auf der Webseite zu dieser Diplomarbeit [69], die Quelltexte auch auf der beiliegenden CD (siehe Anhang B).

A.1 Das Eclipse-Projekt

Als Entwicklungsumgebung kam Eclipse [28] in der Version 3.1.1 zum Einsatz. Mit neueren Versionen sollte sich das Projekt problemlos verwenden lassen, ältere Versionen können eventuell nicht alle Einstellungen korrekt übernehmen.

Das Projekt ist in folgende Gruppen untergliedert (siehe Abbildung 34):

src enthält alle Quelltexte des Rahmenwerks, der Standard-Implementierung und der Beispielanwendung. Aufgeteilt sind sie in die Pakete `de.uka.ipd.consensus.foundation`, `de.uka.ipd.consensus.impl` bzw. `de.snailshell.consensus.demo` (sowie in Unterpakete davon). Zusätzlich stehen im Paket `de.uka.ipd.consensus.test` noch einige Testfälle zur Verfügung.

cfdemo beinhaltet alle Dateien, die zusätzlich für die Beispiel-Web-Applikation *ConsensusFoundation Demo* benötigt werden, darunter JSPs, CSS-Definitionen, Bilder sowie die Konfigurationsdateien `web.xml` und `struts-config.xml`. Der Aufbau dieser Gruppe hält sich dabei an die Vorgaben für Web-Applikationen [23, Abschnitt SRV.9].

rsc fasst die anwendungsabhängigen Konfigurationsdateien des Rahmenwerks und der Beispiel-Anwendung zusammen, die automatisch an die passende Stelle in der Web-Applikation kopiert werden. Dazu gehören die Dateien `ConsensusFoundation.properties`, `hibernate.properties` und `context.xml`. Zudem liegt hier `MessageResources.properties`, was zur Lokalisierung der Benutzungsoberfläche genutzt wird.

lib enthält Bibliotheken, die zwar zum Übersetzen des Projekts nötig sind, die aber nicht Teil der Web-Applikation sein dürfen, da entsprechende Bibliotheken bereits vom Servlet-Container zur Verfügung gestellt werden. Außerdem liegt hier der JDBC-Treiber, der von der Ant-Builddatei (siehe Abschnitt A.2) an eine spezielle Stelle der Tomcat-Infrastruktur (außerhalb der Web-Applikation) kopiert werden muss.

build: In diese Gruppe platziert die Ant-Builddatei nach vollständigem und erfolgreichem Lauf das JAR-Archiv `ConsensusFoundation.jar`, das die Schnittstellen des Rahmenwerks und die Standard-Implementierung enthält, sowie das WAR-Archiv [23, Abschnitt SRV.9.6] `cfdemo.war` für die Beispiel-Web-Applikation.

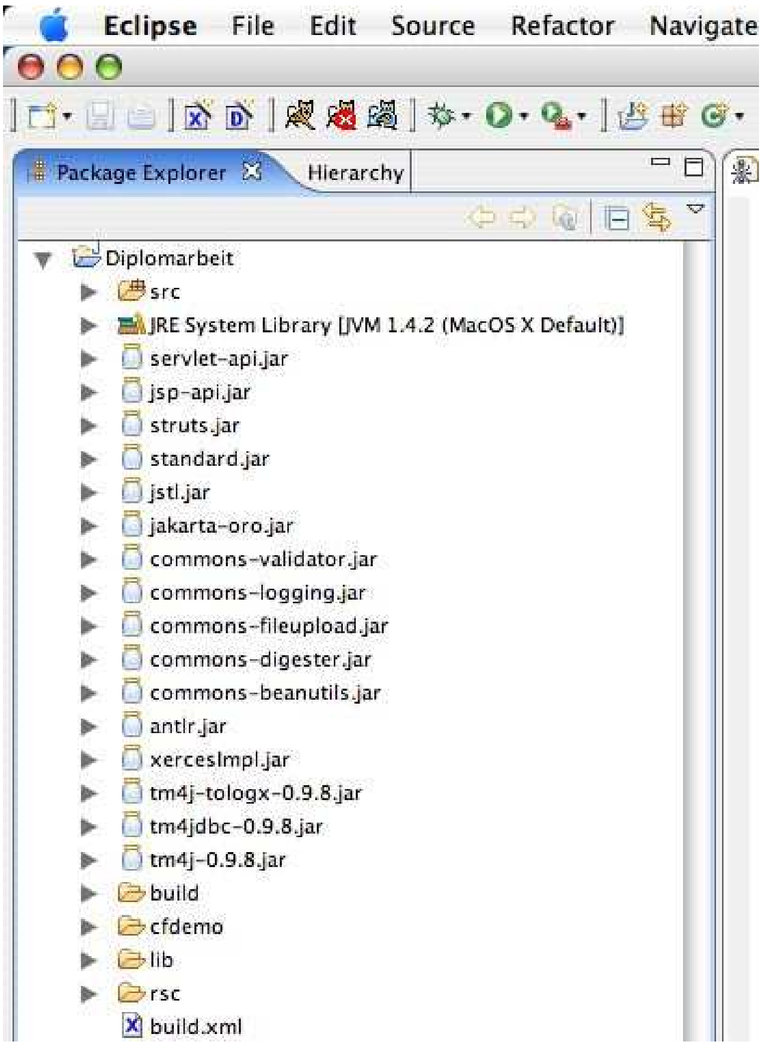


Abbildung 34: Struktur des Eclipse-Projekts

Das Übersetzen des Projekts erfolgt nicht mit der Eclipse-Projektverwaltung, sondern mit der Ant-Builddatei `build.xml` (siehe Abschnitt A.2).

Mit den folgenden Schritten können Sie das Projekt in Ihrer Eclipse-Installation nutzen. Voraussetzung ist dafür ein installiertes Java JDK ab Version 1.4 sowie Tomcat (siehe Abschnitt A.5) und MySQL (siehe Abschnitt A.6):

Installation

- Kopieren Sie das Projekt in Ihren Eclipse-Workspace-Ordner. Dort sollte sich nun das Unterverzeichnis „Diplomarbeit“ befinden.
- In Eclipse fügen Sie das Projekt mit *Import... / Existing Projects into Workspace* dem „Package Explorer“ hinzu. Bei Bedarf können Sie nun den Projektnamen mit *Refactor / Rename...* umbenennen.
- Kopieren Sie den JDBC-Treiber („mysql-connector.....jar“) aus dem `lib`-Verzeichnis des Projekts in das `common/lib`-Verzeichnis von Tomcat. Dies geschieht zwar auch automatisch beim Deployment (s.u.), ist dann aber erst nach dem nächsten Server-Start wirksam.

- Fügen Sie `build.xml` zur „Ant“-View von Eclipse hinzu.

Konfiguration

build.xml

- Setzen Sie `tomcat.home` auf Ihr Tomcat-Verzeichnis.
- Passen Sie `tomcat.username` und `tomcat.password` an.

rsc/ConsensusFoundation.properties

- Setzen Sie `cf.superadmin` auf den Benutzernamen des Anwenders, der die Administratoren-Rechte verwalten soll. Dieser Benutzer muss noch nicht existieren! Wenn sich später ein Nutzer mit diesem Namen registriert, wird ihm automatisch beim Login die Rolle des Super-Administrators zugewiesen.
- Aktivieren Sie die Zeile `cfimpl.ontology.backend=hibernate`, wenn Hibernate verwendet werden soll. Ansonsten wird das In-Memory-Backend genutzt.
- Optional können Sie `cfimpl.ontology.file` aktivieren und als Wert eine XTM-Datei eintragen, die beim Start der Anwendung geladen wird. Diese Datei ersetzt eine eventuell vorhandene Ontologie, beim Hibernate-Backend sollte diese Zeile also nur beim ersten Deployment aktiv sein!
- Achten Sie darauf, dass die Werte von `cfimpl.userdatasource` und `cfimpl.ratingdatasource` mit den Datenquellennamen in `context.xml` übereinstimmen.

rsc/context.xml

- Für die Benutzer-Datenbasis müssen die Parameter „url“, „username“ und „password“ angepasst werden.
- Das bei „url“ angegebene Datenbasis-Schema muss zuvor in MySQL angelegt worden sein.
- Standardmäßig wird das Schema „test“ verwendet, in dem beim Start der Anwendung automatisch die Tabelle „CFUSERS“ erzeugt wird, falls sie noch nicht exisitert.

rsc/hibernate.properties

- Sofern Hibernate verwendet wird, müssen hier die `hibernate.connection`-Parameter „url“, „username“ und „password“ angepasst werden.
- Das eingesetzte Datenbasis-Schema (standardmäßig „tm4j“) muss zuvor in MySQL angelegt worden sein und sollte für nichts anderes — auch nicht für die Benutzer-Datenbasis — genutzt werden.

Einsatz (Deployment)

- Starten Sie MySQL.
- Legen Sie die Datenbasis-Schemata für die Benutzer-Datenbasis und für Hibernate an (s.o.), z.B. mit dem MySQL-Administrator.
- Rufen Sie in Eclipse das Ant-Target „init-hibernate“ auf.
- Starten Sie Tomcat.

- Rufen Sie in Eclipse das Ant-Target „all" auf, mit dem die Beispiel-Anwendung übersetzt und in Tomcat als Web-Applikation eingerichtet wird. Das Target wurde erfolgreich ausgeführt, wenn in der Eclipse-Konsole folgende Zeile auftaucht:
„INFO: Tiles definition factory loaded for module ''."
- Nun können Sie die Anwendung aufrufen und nutzen (siehe Abschnitt A.7).

A.2 Ant-Builddatei

Man sollte sich bei Projekten niemals auf das Build-System einer bestimmten Entwicklungsumgebung verlassen, sondern immer ein vollständig automatisiertes Build-Skript einsetzen, um unabhängig von den zahlreichen Entwicklungsumgebungen zu sein [84, Seite 29-35]. Aus diesem Grund verwendet ConsensusFoundation das bei Java-Projekten übliche Build-Werkzeug *Ant* [3]. In der zugehörigen Builddatei `build.xml` wurden dafür folgende Targets definiert:

clean leert alle Ausgabeverzeichnisse. Die generierten WAR- und JAR-Archive, die Klassendateien und die Javadoc-Dokumentation werden dadurch gelöscht.

compile übersetzt alle Klassen und erzeugt das JAR- und WAR-Archiv für das Rahmenwerk bzw. die Beispiel-Anwendung. Dies ist das Standard-Target.

javadoc generiert aus den Quelltexten die Javadoc-Dokumentation.

deploy führt die vorhergehenden Targets aus und setzt das WAR-Archiv in Tomcat (der dafür bereits laufen muss) als Web-Applikation neu ein.

all führt alle obigen Targets aus.

init-hibernate sollte nur ein einziges Mal für jede Anwendung aufgerufen werden. Hiermit werden die Tabellen des Datenbasis-Schemas neu angelegt (d.h. bestehende zuvor gelöscht), in denen TM4J die Daten der Ontologie speichert.

Zur Konfiguration des Build-Prozesses können am Anfang von `build.xml` die in Tabelle 3 aufgeführten Eigenschaften (Properties) angepasst werden:

Eigenschaft (Property)	*Standardwert und Beschreibung*
`webappname`	`cfdemo` Der Name der Web-Applikation, der den Namen des WAR-Archivs und des Kontext-Pfades [23, Abschnitt SRV.3] bestimmt.
`libname`	`ConsensusFoundation` Der Name des JAR-Archivs, in dem die Schnittstellen des Rahmenwerks und die Klassen der Standard-Implementierung zusammengefasst werden.
`src`	`src` Das Verzeichnis (relativ zum Projektverzeichnis), in dem sich die Quelltexte befinden.
`properties`	`rsc` Das Verzeichnis (relativ zum Projektverzeichnis), das die Konfigurationsdateien enthält.

Fortsetzung auf nächster Seite

Fortsetzung	
`lib`	`lib` In diesem Verzeichnis liegen Bibliotheken, die nur zum Übersetzen benötigt werden, aber nicht Bestandteil der Web-Applikation sind.
`docs`	`$webappname/doc` In diesem Verzeichnis wird die generierte Javadoc-Dokumentation gespeichert. Standardmäßig erfolgt dies also im Unterverzeichnis `doc` der Web-Applikation.
`build`	`build` Die erzeugten JAR- und WAR-Archive werden vom Buildfile in diesem Verzeichnis abgelegt.
`build.version`	`1.4` Standardmäßig wird das Projekt für Java 1.4 übersetzt. Um eine Übersetzung für Java 5.0 zu erreichen, gibt man hier den Wert `1.5` an.
`tomcat.home`	`/Applications/jakarta-tomcat-5.0.28` Das absolute Verzeichnis der lokalen Tomcat-Installation. Beim hier angegebenen Pfad handelt es sich um einen Mac OS X-Pfad (siehe auch [68, Abschnitt 8.1]), für Windows müsste beispielsweise `F:\Apache Group\Tomcat 5.0` eingetragen werden.
`tomcat.username`	`admin` Der Benutzername der Tomcat-Manager-Anwendung, der die nötigen Administrator-Rechte zum Installieren von Web-Applikationen besitzt.
`tomcat.password`	`admin` Das Passwort zum voranstehenden Benutzernamen.
`jdbc.driver`	`lib/mysql-connector-java-3.1.12-bin.jar` Der Pfad zum JDBC-Treiber, der vom „deploy“-Target in das Tomcat-Verzeichnis `common/lib` kopiert wird.
`jdbc.datasource`	`rsc/context.xml` Der Pfad der Datei `context.xml`, in der die JNDI-Datenquellen definiert werden. Die Datei wird vom „compile“-Target in das `META-INF`-Verzeichnis der Web-Applikation kopiert.

Tabelle 3: Konfigurierbare Eigenschaften in build.xml

Die vorliegende Ant-Builddatei kann die Installation der Web-Applikation nur bei einem lokal laufenden Tomcat durchführen. Zur Installation auf einem entfernten Server reicht es aber aus, der WAR-Archiv `cfdemo.war` in das passende Verzeichnis des Servers zu kopieren — bei Tomcat ist dies das Verzeichnis `webapps`.

A.3 Konfigurationsdateien

A.3.1 ConsensusFoundation.properties

In der Datei `ConsensusFoundation.properties` werden die Komponenten des Rahmenwerks konfiguriert, d.h. welche konkreten Implementierungen verwendet werden sollen. Außerdem können noch gewissen Eigenschaften einer Komponente bestimmt werden, falls die jeweilige Komponente dies vorsieht. Die Komponenten wurden in Kapitel 4 ausführlich beschrieben.

Tabelle 4 führt sowohl die Standard-Schlüssel („cf...“), die jede Implementation unterstützen muss, als auch die implementationsabhängigen Schlüssel („cfimpl...“) auf. Wenn man die Schlüssel programmatisch verarbeiten möchte, findet man zumindest für die Standard-Schlüssel entsprechende Konstanten in der Klasse `de.uka.ipd.consensus.foundation.ConsensusFoundation`.

Properties-Schlüssel	*Standardwert und Beschreibung*
	Module
`cf.loggingmanager`	`de.uka.ipd.consensus.impl.LoggingManagerImpl`
`cf.usermanager`	`de.uka.ipd.consensus.impl.UserManagerImpl`
`cf.ontologymanager`	`de.uka.ipd.consensus.impl.OntologyManagerImpl`
`cf.evolutionmanager`	Keiner, optionales Modul. Als Implementierung steht `de.uka.ipd.consensus.impl.EvolutionManagerImpl` zur Verfügung.
`cf.ratingmanager`	Keiner, optionales Modul. Als Implementierung steht `de.uka.ipd.consensus.impl.SQLRatingManagerImpl` zur Verfügung.
`cf.incentivemanager`	Keiner, optionales Modul. Als Implementierung steht `de.uka.ipd.consensus.impl.IncentiveManagerImpl` zur Verfügung.
`cf.dynamicrightsmanager`	Keiner, optionales Modul. Als Implementierung steht `de.uka.ipd.consensus.impl.DynamicRightsManagerImpl` zur Verfügung.
	Schema
`cfimpl.schema.useclasses`	`false`, d.h. die Topic-Hierarchie besteht nur aus Instanzen. Wird hier `true` eingetragen, kann eine Klassenhierarchie angelegt werden, wodurch Typen und Instanzen unterscheidbar sind (siehe Abschnitt 4.2).
`cfimpl.dynrights.typedonly`	`false`. Wird hier `true` eingetragen, können Topics, Occurrences, Assoziationen und Attribute nur typisiert angelegt werden, d.h. sie dürfen dann nicht typenlos sein (siehe Abschnitt 4.9).
	Ontologieverwaltung
`cf.queryengine`	`de.uka.ipd.consensus.impl.QueryEngineImpl`. Wird normalerweise nicht oder nur zusammen mit `cf.ontologymanager` gesetzt.
`cf.exporthandler`	`de.uka.ipd.consensus.impl.XTMExportHandler`. Zusätzlich können hier weitere Export-Module (durch Komma getrennt) angegeben werden.
`cf.importhandler`	`de.uka.ipd.consensus.impl.XTMImportHandler`. Zusätzlich können hier weitere Import-Module (durch Komma getrennt) angegeben werden.
`cfimpl.ontology.file`	Keiner. Wenn hier eine Datei angegeben ist, für die ein passendes Import-Modul vorhanden ist, wird die Datei beim Start der Anwendung als Ontologie geladen und ersetzt eine eventuell bisher vorhandene Ontologie. Die Dateien liegen normalerweise im XTM-Format vor.
	Fortsetzung auf nächster Seite

Fortsetzung	
`cfimpl.ontology.baseuri`	`http://uka.de/ipd/dbis/consensus/default.xtm`. Dies ist die URI, die TM4J intern als Basis-URI benötigt, um die Elemente der Ontologie eindeutig identifizieren zu können.
`cfimpl.ontology.backend`	Keiner, d.h. es wird das In-Memory-Backend verwendet. Um das Hibernate-Backend zu nutzen, muss hier `hibernate` eingetragen werden.
	Bewertungs- und Anreizsystem
`cf.scoringlistener`	Keiner. Hier kann eine Liste (durch Komma getrennt) von Klassen angegeben werden, die dann alle als ScoringListener registriert werden. Zum Testen steht die Implementierung `de.uka.ipd.consensus.impl.ScoringListenerImpl` zur Verfügung.
`cfimpl.ratingdatasource`	`jdbc/datenquelle`. Dies ist der Name der in `context.xml` definierten JNDI-Datenquelle, in der die Bewertungen verwaltet werden.
	Benutzerverwaltung
`cf.superadmin`	Keiner. Hier kann der Benutzername eines Anwenders angegeben werden, der dann die Rolle des Super-Administrators erhält, durch die er die Administratoren-Rechte anderer Nutzer verwalten darf.
`cfimpl.initdatabase`	`false`. Wenn hier `true` eingetragen ist, werden die für die Benutzerverwaltung nötigen Tabellen automatisch beim Programmstart angelegt, sofern sie noch nicht vorhanden sind.
`cfimpl.userdatasource`	`jdbc/datenquelle`. Dies ist der Name der in `context.xml` definierten JNDI-Datenquelle für die Benutzerdaten.
	Protokollierung
`cf.loglevel`	`error` (andere mögliche Werte: `off`, `fatal`, `warn`, `info`, `debug` und `all`)

Tabelle 4: Einträge in der ConsensusFoundation-Konfigurationsdatei

A.3.2 context.xml

Die Datei `context.xml` definiert die JNDI-Datenquellen (`javax.sql.DataSource`) für die Web-Applikation. Wenn eine solche Datei im Verzeichnis `META-INF` einer WAR-Datei gespeichert ist (so, wie es die Ant-Builddatei macht), werden die darin definierten Datenquellen automatisch vom Servlet-Container lokal für die jeweilige Web-Applikation angelegt und müssen nicht global für alle Web-Applikationen in den Server-Konfigurationsdateien eingetragen werden.

Die Beispiel-Anwendung legt nur eine einzige Datenquelle mit dem Namen `jdbc/datenquelle` an, über die sowohl die Benutzer- als auch die Bewertungsdaten verwaltet werden. Andere Anwendungen können aber durchaus mehrere Datenquellen nutzen. Sie müssen nur beachten, dass alle verwendeten Datenquellen korrekt in `web.xml` referenziert werden.

A.3.3 web.xml

`WEB-INF/web.xml` ist der *Deployment Descriptor* [23, Abschnitt SRV.13] einer Web-Applikationen, in dem die grundlegende Konfiguration festgelegt wird. Da die Beispiel-Anwendung das Struts-Rahmenwerk einsetzt (siehe Abschnitt A.4), wird hier das Struts-`ActionServlet` konfiguriert. Unabhängig vom verwendeten Rahmenwerk für die Darstellungs- bzw. Steuerungsschicht definiert diese Datei allgemeine Eigenschaften der Web-Applikation wie die Startseite, die Fehlerseite oder den Session-Timeout.

Sofern Sie den JNDI-Namen der Datenquelle ändern oder weitere Datenquellen hinzufügen, müssen Sie die Referenzierung der Datenquelle am Ende von `web.xml` entsprechend anpassen (siehe Listing 22). Bei mehreren Datenquellen führen Sie das `<resource-ref>`-Element mehrfach auf.

```
<resource-ref>
    <res-ref-name>jdbc/datenquelle</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

Listing 22: Referenz auf eine JNDI-Datenquelle in web.xml

A.3.4 hibernate.properties

Sofern in `ConsensusFoundation.properties` das Hibernate-Backend aktiviert wurde, wird dieses in der Datei `hibernate.properties` konfiguriert. Sie müssen hier den Parameter `hibernate.connection.url` anpassen und ein existierendes Datenbasis-Schema angeben, das TM4J zum Persistieren seiner Daten verwenden kann (siehe Abschnitt A.1). Benutzername und Passwort für dieses Schema müssen natürlich ebenfalls auf die korrekten Werte gesetzt werden.

Die anderen Werte für den Hibernate-Dialekt — bei der Standard-Implementierung von ConsensusFoundation ist dies „MySQL“ — sowie für den Connection-Pool „C3P0“ bleiben normalerweise unverändert.

A.4 Struts

Das Rahmenwerk *Struts* [4] wird in der Version 1.2.7 eingesetzt. Wie bei Struts üblich wird die Struts-Anwendung mit der Datei `WEB-INF/struts-config.xml` innerhalb der Web-Applikation konfiguriert. Hier findet auch die Verknüpfung mit der Klasse `de.snailshell.consensus.demo.ConsensusFoundationPlugin` statt, die sich als Struts-Plugin um die Initialisierung von ConsensusFoundation kümmert (siehe Listing 23 und Abschnitt 4.4).

```
<plug-in
    className="de.snailshell.consensus.demo.ConsensusFoundationPlugin">
    <set-property property="config"
                  value="ConsensusFoundation.properties" />
</plug-in>
```

Listing 23: Einbindung von ConsensusFoundation in struts-config.xml

Ebenfalls in `struts-config.xml` wird das `TilesPlugin` von Struts eingebunden, mit dem Schablonen (Templates) für den Aufbau der Web-Seiten angelegt werden können. Die Beispiel-Anwendung *ConsensusFoundation Demo* definiert in der Datei `WEB-INF/tiles-defs.xml` drei solcher Schablonen: Eine für externe Seiten, die der Nutzer ohne Anmeldung aufrufen kann, eine für interne Seiten, die nur nach erfolgreicher Anmeldung sichtbar sind, und eine dritte Schablone für Seiten, die allen Nutzern zur Verfügung stehen, aber angemeldeten Benutzern zusätzliche Optionen bieten.

Auf oberster Ebene der Web-Applikation befinden sich die JSP-Dateien, die diese Schablonen nutzen, beispielsweise `home.jsp`. In diesen Dateien wird dann mit der jeweils passenden Tiles-Schablone eine weitere JSP-Datei eingebunden, welche die eigentlichen Daten der Web-Seite, also das angezeigte

Dokument, enthält (hier: `WEB-INF/content/home-inhalt.jsp`). Weil diese eingebundenen JSPs unterhalb von `WEB-INF` liegen, können sie von außen nicht direkt angesprochen werden, was den Schutz der Seiten erleichtert, die nur angemeldeten Nutzern zur Verfügung stehen.

A.5 Tomcat

Als Servlet-Container [23, Abschnitt SRV.1.2] wird von der Beispiel-Anwendung *Tomcat* [5] genutzt, der für die Verarbeitung der Servlets und JSPs der Web-Applikation verantwortlich ist. Zum Einsatz kommt die stabile Version 5.0.28, die für Java 1.4-Anwendungen geeignet ist. Wenn Sie ConsensusFoundation als Java 5.0-Anwendung betreiben wollen, bedenken Sie bitte, dass Sie dann eine aktuelle Tomcat 5.5-Version verwenden müssen.

A.6 MySQL

Das Datenbanksystem *MySQL* wird von ConsensusFoundation zur Speicherung der Benutzer- und Bewertungsdaten eingesetzt, und intern nutzt es die TM4J-Bibliothek zur Persistierung der Ontologie-Daten mittels Hibernate [47]. Zum Einsatz kommt die MySQL-Version 5.0.19 [71], als JDBC-Treiber wird *Connector/J* [70] in der Version 3.1.12 genutzt.

Der Zugriff auf MySQL erfolgt über JNDI-Datenquellen, die in der Datei `context.xml` (siehe Abschnitt A.3.2) für die Web-Applikation definiert werden. Diese Datenquellen müssen dann auch in `web.xml` (siehe Abschnitt A.3.3) referenziert und in `ConsensusFoundation.properties` (siehe Abschnitt A.3.1) angegeben werden.

Die von der ConsensusFoundation genutzten Tabellen können von der Standard-Implementierung selber angelegt werden. Die Methode `de.uka.ipd.consensus.impl.UserManagerImpl.initDatabase()` erzeugt Tabelle 5 für die Benutzerdaten, `SQLRatingManagerImpl.initDatabase()` die Tabellen 6, 7 und 8 für die Bewertungen, die Bewertungshistorie und Statistikwerte.

Feldname	*Datentyp*	*Beschreibung*
<u>`BENUTZERNAME`</u>	`VARCHAR(50)`	Der eindeutige Benutzername (Username) jedes Anwenders.
`PASSWORT`	`INT`	Der `hashCode()` der Passwort-Zeichenkette.
`VORNAME`	`VARCHAR(50)`	Der Vorname des Benutzers.
`NACHNAME`	`VARCHAR(50)`	Der Nachname des Benutzers.
`EMAIL`	`VARCHAR(50)`	Die E-Mail-Adresse des Benutzers.
`SCORE`	`DOUBLE`	Der Punktestand des Benutzers.
`COMMENT`	`VARCHAR(4096)`	Ein Kommentar vom Benutzer über sich selbst.
`ROLES`	`VARCHAR(255)`	Die Rollen, die dem Benutzer vom System zugewiesen wurden, durch Komma getrennt. Die Standard-Implementierung sieht nur die Rollen „admin“ und „superadmin“ vor.

Tabelle 5: Tabelle CFUSERS für die Benutzerdaten

Feldname	*Datentyp*	*Beschreibung*
`ID`	`INT`	Der Schlüssel für jede Bewertung.
`USERNAME`	`VARCHAR(255)`	Der Name des Benutzers, der diese Bewertung abgegeben hat (Fremdschlüssel BENUTZERNAME aus CFUSERS).
`RATEABLEID`	`VARCHAR(255)`	Die Kennung des bewerteten Elements, d.h. die Konzept-ID oder der Benutzername.
`RATEABLETYPE`	`INT`	Der Typ des bewerteten Elements: 67 für ein Konzept (Zeichenwert von 'C'), 85 für einen Benutzer (Zeichenwert von 'U').
`VALUE`	`DOUBLE`	Der Wert der Bewertung von -1 bis 1 (inklusive).
`COMMENT`	`VARCHAR(255)`	Ein optionaler Kommentar zur Bewertung
`RATINGDATE`	`BIGINT`	Der Zeitstempel der Bewertung als Millisekunden seit dem 1.1.1970 00:00 GMT.
`RATEDDATE`	`BIGINT`	Der Zeitstempel der letzten Ändernung am bewerteten Element als Millisekunden seit dem 1.1.1970 00:00 GMT.

Tabelle 6: Tabelle CFRATINGS für die Bewertungen

Feldname	*Datentyp*	*Beschreibung*
`ID`	`INT`	Der Schlüssel innerhalb der Historie.
`RATINGID`	`INT`	Die Kennung der Bewertung (Fremdschlüssel ID aus CFRATINGS).
`VALUE`	`DOUBLE`	Der Wert der alten Bewertung.
`COMMENT`	`VARCHAR(255)`	Der Kommentar zur alten Bewertung.
`RATINGDATE`	`BIGINT`	Der Zeitstempel der alten Bewertung.
`RATEDDATE`	`BIGINT`	Der Zeitstempel des bewerteten Elements zum Zeitpunkt der alten Bewertung.

Tabelle 7: Tabelle CFRATINGHIST für die Bewertungshistorie

Feldname	*Datentyp*	*Beschreibung*
`RATEABLEID`	`VARCHAR(255)`	Der Schlüssel, für welches bewertete Element Durchschnitt und Anzahl gespeichert werden.
`AVGVALUE`	`DOUBLE`	Das arithmetische Mittel aller Bewertungen zu diesem Element.
`RATINGCOUNT`	`INT`	Die Anzahl der Bewertungen zu diesem Element (nur die aktuellen Elemente, die Historie wird dabei nicht berücksichtigt).

Tabelle 8: Tabelle CFRATINGSTAT für Durchschnitt und Anzahl der Bewertungen

A.7 ConsensusFoundation Demo

Wenn die Beispiel-Web-Applikation *ConsensusFoundation Demo* mit der Arbeitsanweisung in Abschnitt A.1 konfiguriert und installiert wurde, kann sie in einem Web-Browser über die URL `http://localhost:8080/cfdemo/` aufgerufen werden. Wenn der Einsatz nicht auf einem lokalen Web-Server erfolgt oder die Web-Applikation umbenannt wurde, weicht die Adresse entsprechend ab.

An dieser Stelle erfolgt keine Beschreibung der Beispiel-Anwendung. Es soll einfach nur ein kurzer visueller Eindruck der Benutzungsoberfläche gegeben werden. Für weitergehende Informationen sollte direkt die Beispiel-Anwendung genutzt werden, die sich auf der beiliegenden CD befindet (siehe Anhang B).

Abbildung 35 zeigt die Topic-Hierarchie, die hier keinen Unterschied zwischen Klassen (Typen) und Instanzen macht. Sehr schön sind die Bewertungen der Hierarchie-Beziehungen zu sehen, sofern Bewertungen abgegeben wurden. Die farbliche Interpretation der Werte ist Sache der Anwendung und nicht fest vorgegeben.

Wenn man ein einzelnes Topic anzeigen lässt, erscheint die in Abbildung 36 ersichtliche Bildschirmmaske. Sofern für das Topic oder seine Beziehungen Bewertungen vorliegen, wird auch dies entsprechend dargestellt. Interessant ist bei diesem Beispiel, dass der Anwender eine ältere Version des Topics bewertet hat, d.h. nach Abgabe der Bewertung wurde das Topic (z.B. seine Beschreibung) geändert. Die Anwendung weist den Nutzer auf diesen Umstand hin, damit er seine Bewertung gegebenenfalls korrigieren kann.

Die freie Ontologie-Abfrage innerhalb der Beispiel-Anwendung mit der Abfragesprache „Tolog“ wurde bereits in Abbildung 21 gezeigt.

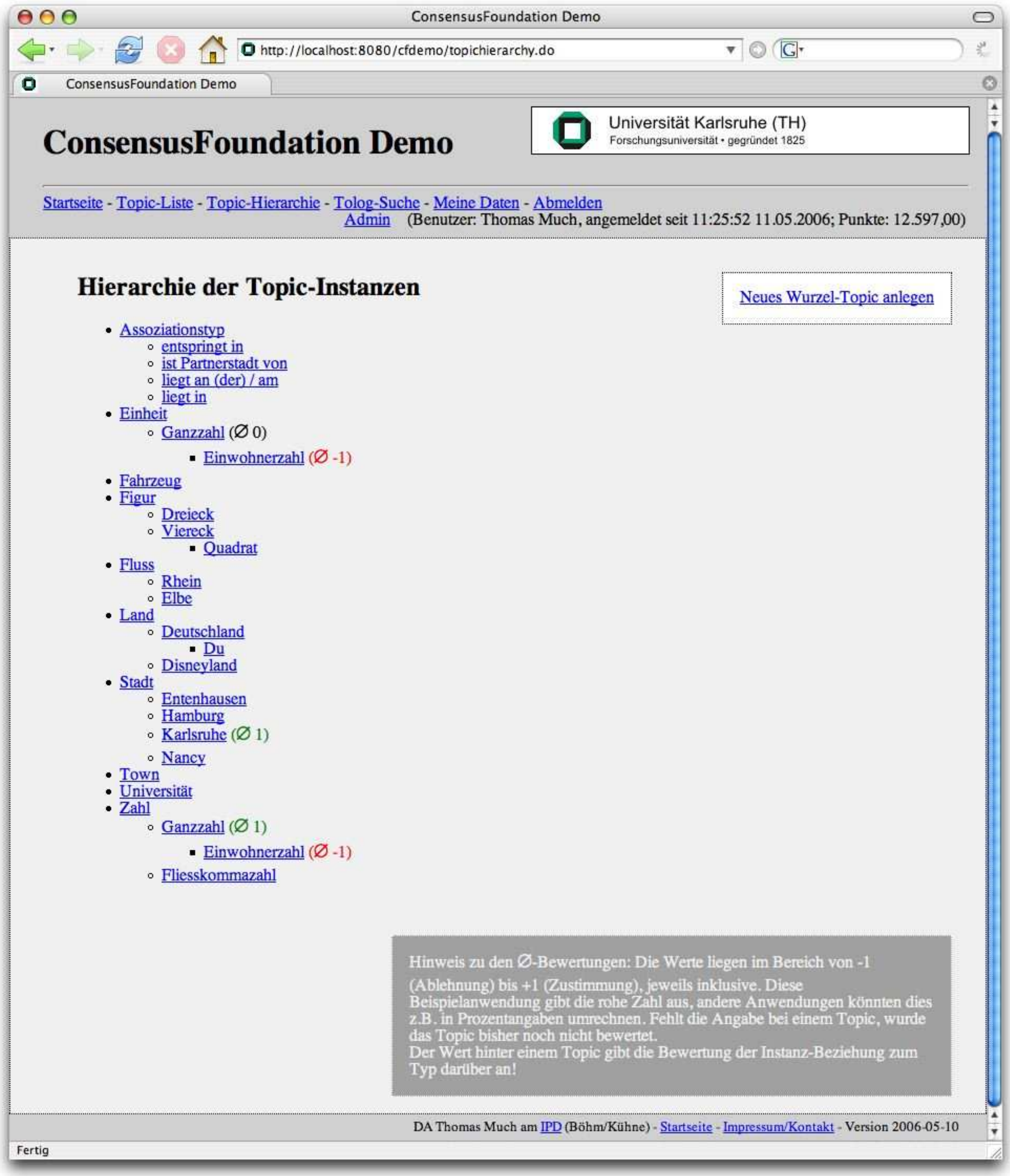


Abbildung 35: Topic-Hierarchie in der Beispiel-Anwendung

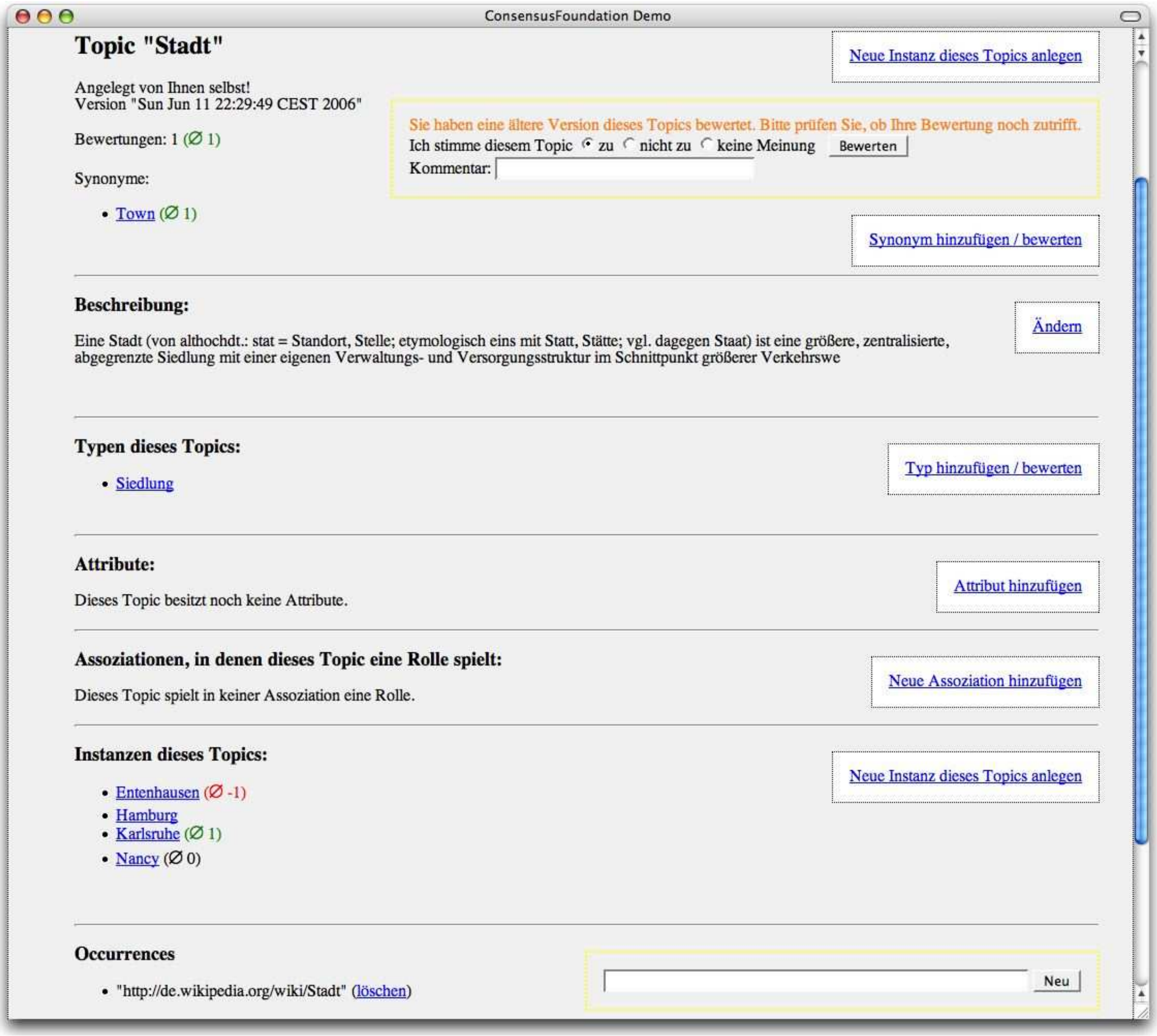


Abbildung 36: Anzeige eines einzelnen Topics in der Beispiel-Anwendung

B Inhalt der CD

Auf der beiliegenden CD-ROM befinden sich die folgenden Daten:

- Das *ConsensusFoundation*-Rahmenwerk als JAR-Archiv.
- Tomcat 5.0 mit der installierten Web-Applikation *ConsensusFoundation Demo*.
- Eine Beispiel-Ontologie im XTM-Format, die in die Beispiel-Anwendung importiert werden kann.
- Das Eclipse 3.1-Projekt mit allen Quelltexten für das Rahmenwerk, die Standard-Implementierung und die Beispiel-Anwendung (siehe Abschnitt A.1). Erzeugt werden die Produkte nicht von der Eclipse-Projektverwaltung, sondern von der enthaltenen Ant-Builddatei (siehe Abschnitt A.2).
- Die Javadoc-Dokumentation der Schnittstellen des Rahmenwerks und der Standard-Implementierung.
- Die UML-Diagramme im Poseidon- und XMI-Format.
- Dieses Dokument als PDF-Datei sowie
- der BibTeX-Eintrag für dieses Dokument.

Die jeweiligen Dateien können auch von der Webseite zu dieser Diplomarbeit [69] heruntergeladen werden.

C Danksagung

Neben Professor Böhm und meinem Betreuer Conny Kühne gilt mein Dank auch all jenen, die mich in den vergangenen Monaten zwar nicht fachlich oder technisch, dafür aber umso mehr mental unterstützt haben. Ich danke

- meinen Eltern Christa und Horst Much für die Geduld, die sie aufbringen mussten, bis ihr ältester Sohn sich doch noch auf den Weg zu einem berufsqualifizierenden Abschluss machte,
- meinen beiden jüngeren Brüdern für das Vormachen eines solchen berufsqualifizierenden Abschlusses,
- den Schwarzwäldern für den Kontakt zu Studium und Universität trotz Berufstätigkeit,
- der GFU für das (weitestgehende) Freistellen in den Monaten der Diplomarbeit,
- dem Galileo-Verlag, dass er mit einem neuen, spannenden Projekt wartet, bis ich wieder zur Verfügung stehe
- und — last but not least — Gesine für ihre liebe Unterstützung, Motivation und jede Menge Sonnenschein :-)

Literatur

[1] *Amaya Editor/Browser.* `http://www.w3.org/Amaya/`, Abruf: 20. Apr. 2006

[2] *Amazon.de.* `http://www.amazon.de/`, Abruf: 6. Jun. 2006

[3] *Apache Ant.* `http://ant.apache.org/`, Abruf: 24. Mai 2006

[4] *Apache Struts.* `http://struts.apache.org/`, Abruf: 24. Mai 2006

[5] *Apache Tomcat.* `http://tomcat.apache.org/`, Abruf: 10. Jun. 2006

[6] Backschat, M. ; Edlich, S.: *J2EE-Entwicklung mit Open-Source-Tools.* Spektrum Akademischer Verlag, Heidelberg, 2004. – ISBN 3–8274–1446–6

[7] Balzert, H.: *Lehrbuch der Objektmodellierung: Analyse und Entwurf.* Spektrum Akademischer Verlag, Heidelberg, 1999. – ISBN 3–8274–0285–9

[8] Balzert, H.: *Lehrbuch der Softwaretechnik, Band 1.* 2. Auflage. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2001. – ISBN 3–8274–0480–0

[9] Bauer, C. ; King, G.: *Hibernate in Action.* Manning Publications Co., 2005. – ISBN 1932394–15–X

[10] Bayern, S.: *JSTL in Action.* Manning Publications Co., 2002. – ISBN 1930110–52–9

[11] Becket, D. (Hrsg.): *RDF/XML Syntax Specification (Revised).* `http://www.w3.org/TR/rdf-syntax-grammar/`. Version: 10. Februar 2004, Abruf: 5. Jun. 2006

[12] Bergsten, H.: *JavaServer Faces.* O'Reilly Media, Inc., 2004. – ISBN 0–596–00539–3

[13] Berners-Lee, T.: *HTTP: A protocol for networked information.* `http://www.w3.org/Protocols/HTTP/HTTP2.html`. Version: 1992, Abruf: 20. Apr. 2006

[14] Berners-Lee, T. ; Fischetti, M.: *Weaving the Web: the original design of the World Wide Web by its inventor.* HarperCollins, 2000. – ISBN 0–06–251587–X

[15] Biezunski, M. (Hrsg.) ; Bryan, M. (Hrsg.) ; Newcomb, S. (Hrsg.): *ISO/IEC 13250, Topic Maps (Second Edition).* `http://www.y12.doe.gov/sgml/sc34/document/0322_files/iso13250-2nd-ed-v2.pdf`. Version: 22. Mai 2002, Abruf: 2. Jun. 2006

[16] Bloch, J.: *Effective Java™Programming Language Guide.* Addison-Wesley, 2002. – ISBN 0201310058

[17] Brickley, D. (Hrsg.) ; Guha, R. V. (Hrsg.): *RDF Vocabulary Description Language 1.0: RDF Schema.* `http://www.w3.org/TR/rdf-schema/`. Version: 10. Februar 2004, Abruf: 5. Jun. 2006

[18] Cheng, R. ; Vassileva, J.: Adaptive Reward Mechanism for Sustainable Online Learning Community. In: *AI in Education AIED'2005.* Amsterdam : IOS Press, 18.-22. Juli 2005, 152-159

[19] Cheng, R. ; Vassileva, J.: User Motivation and Persuasion Strategy for Peer-to-peer Communities. In: *Proceedings HICSS'2005 (Mini-track on Online Communities in the Digital Economy/Emerging Technologies).* Hawaii, 3.-6. Januar 2005

[20] *Commons-logging.* `http://jakarta.apache.org/commons/logging/`, Abruf: 26. Apr. 2006

[21] *Core J2EE Patterns - Transfer Object.* `http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html`, Abruf: 7. Mai 2006

[22] Cormen, T. H. ; Leiserson, C. E. ; Rivest, R. L.: *Introduction to Algorithms.* MIT Press, 1990. – ISBN 0–262–03141–8

[23] Coward, D. ; Yoshida, Y.: *Java™Servlet Specification Version 2.4.* `http://jcp.org/aboutJava/communityprocess/final/jsr154/`. Version: 24. November 2003, Abruf: 24. Apr. 2006

[24] Cregan, A.: *An OWL DL construction for the ISO Topic Map Data Model.* `http://www.cse.unsw.edu.au/~annec/Building_TMs_in_OWL_Draft_Proposal_16_May_2005.PDF`. Version: 16. Mai 2005, Abruf: 2. Jun. 2006

[25] *Der Brockhaus in fünfzehn Bänden.* F.A. Brockhaus GmbH, Leipzig, Mannheim, 1998

[26] *Dublin Core Metadata Initiative (DCMI).* `http://dublincore.org/`, Abruf: 5. Jun. 2006

[27] *eBay Deutschland – Der weltweite Online-Marktplatz.* `http://www.ebay.de/`, Abruf: 6. Jun. 2006

[28] *Eclipse.org home.* `http://www.eclipse.org/`, Abruf: 9. Jun. 2006

[29] Ehrig, M. ; Bruijn, J. de ; Martín-Recuerda, F. ; Polleres, A. ; Predoiu, L.: Ontology Mediation Management V1 / DERI Innsbruck. Version: 8. Februar 2005. `http://www.aifb.uni-karlsruhe.de/WBS/meh/publications/sekt-D4.4.1mediation-management.pdf`, Abruf: 1. Jun. 2006. 2005 (4.4.1). – Forschungsbericht

[30] *Extensible Markup Language (XML) 1.0 (Third Edition).* `http://www.w3.org/TR/REC-xml/`, Abruf: 1. Jun. 2006

[31] Frick, A. ; Zimmer, W. ; Zimmermann, W.: Konstruktion robuster und flexibler Klassenbibliotheken. In: *Informatik, Forschung und Entwicklung* 11 (1996), Nr. 4, S. 168–178. – ISSN 0178–3564

[32] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, 1996. – ISBN 3–89319–950–0

[33] Garshol, L. M.: *tolog 1.0.* `http://www.ontopia.net/topicmaps/materials/tolog-spec.html`. Version: 2003, Abruf: 22. Mai 2006

[34] Garshol, L. M.: *The Linear Topic Map Notation.* `http://www.ontopia.net/download/ltm.html`. Version: 7. Januar 2006, Abruf: 2. Jun. 2006

[35] Garshol, L. M. (Hrsg.) ; Moore, G. (Hrsg.): *Topic Maps – Data Model.* `http://www.isotopicmaps.org/sam/sam-model/`. Version: 16. Dezember 2005, Abruf: 5. Jun. 2006

[36] Goos, G.: *Vorlesungen über Informatik.* Bd. 1. 3. Auflage. Springer Verlag, Berlin, Heidelberg, New York, 2000. – ISBN 3–540–67270–2

[37] Gosling, J. ; Joy, B. ; Steele, G. ; Bracha, G.: *The Java™Language Specification.* 3. Auflage. Addison-Wesley, 2005. – ISBN 0–321–24678–0

[38] Grassmuck, V.: *Freie Software – Zwischen Privat- und Gemeineigentum.* Bundeszentrale für politische Bildung (bpb), Bonn, 2002. – ISBN 3–89331–432–6

[39] Gruber, T. R.: *A Translation Approach to Portable Ontology Specifications.* http://ksl-web.stanford.edu/KSL_Abstracts/KSL-92-71.html. Version: 1993, Abruf: 31. Mai 2006

[40] Hamilton, G. (Hrsg.): *JavaBeans™1.01 Specification.* http://java.sun.com/products/javabeans/docs/spec.html, Abruf: 6. Mai 2006

[41] Härder, T. ; Rahm, E.: *Datenbanksysteme: Konzepte und Techniken der Implementierung.* 2. Auflage. Springer-Verlag, Berlin, Heidelberg, New York, 2001. – ISBN 3–540–42133–5

[42] Heflin, J. (Hrsg.): *OWL Web Ontology Language Use Cases and Requirements.* http://www.w3.org/TR/webont-req/. Version: 10. Februar 2004, Abruf: 6. Jun. 2006

[43] *heise online – Meldung Nr. 67007 vom 06.12.2005.* http://www.heise.de/newsticker/meldung/67007, Abruf: 30. Mai 2006

[44] *heise online – Meldung Nr. 73579 vom 27.05.2006.* http://www.heise.de/newsticker/meldung/73579, Abruf: 30. Mai 2006

[45] *heise online – 7-Tage-News.* http://www.heise.de/newsticker/, Abruf: 6. Jun. 2006

[46] Hendler, J. (Hrsg.): *Frequently Asked Questions on W3C's Web Ontology Language (OWL).* http://www.w3.org/2003/08/owlfaq. Version: 10. Februar 2004, Abruf: 5. Jun. 2006

[47] *Hibernate.* http://www.hibernate.org/, Abruf: 24. Mai 2006

[48] Hilt, N.: *Analyse und Realisierung von spieltheoretischen Modellen zum Vermeiden von wahrheitswidrigen Bewertungen in einer Online-Community*, Universität Karlsruhe (TH), Diplomarbeit, 16. April 2006

[49] *HTML 4.01 Specification.* http://www.w3.org/TR/REC-html40/, Abruf: 1. Jun. 2006

[50] Hunt, A. ; Thomas, D.: *Der Pragmatische Programmierer.* Carl Hanser Verlang, München, Wien, 2003. – ISBN 3–446–22309–6

[51] *Java™2 Platform Standard Edition 5.0 API Specification.* http://java.sun.com/j2se/1.5.0/docs/api/, Abruf: 3. Mai 2006

[52] *Java 2 Platform Standard Edition 5.0 Compatibility with Previous Releases.* http://java.sun.com/j2se/1.5.0/compatibility.html, Abruf: 12. Jun. 2006

[53] *Java EE At a Glance.* http://java.sun.com/javaee/, Abruf: 24. Mai 2006

[54] *Java™Logging APIs.* http://java.sun.com/j2se/1.4.2/docs/guide/util/logging/, Abruf: 26. Apr. 2006

[55] Johnson, R.: *J2EE™Design and Development.* Wiley Publishing, Inc., 2003. – ISBN 0–7645–4385–7

[56] Johnson, R.: *J2EE™Development without EJB™.* Wiley Publishing, Inc., 2004. – ISBN 0–7645–5831–5

[57] Johnson, R. ; Hoeller, J. ; Arendsen, A. ; Risberg, T. ; Sampaleanu, C.: *Professional Java™Development with the Spring Framework.* Wiley Publishing, Inc., 2005. – ISBN 0–7645–7483–3

[58] *Klassifikation.* http://de.wikipedia.org/wiki/Klassifikation, Abruf: 24. Mai 2006

[59] Klein, M. ; Noy, N. F.: *A Component-Based Framework for Ontology Evolution*. http://smi-web.stanford.edu/pubs/SMI_Abstracts/SMI-2003-0960.html. Version: 2003, Abruf: 18. Apr. 2006

[60] Kühne, C. ; Böhm, K.: Classification Schemes for Contributions in Peer-Rating Online Communities – a Quantitative Assessment. In: *Forthcoming in Group Decision and Negotiation 2006 (GDN 2006), Joint conference of the INFORMS section on Group Decision and Negotiation and the EURO Working Group on Decision Support Systems*. Karlsruhe, Deutschland, 25.-28. Juni 2006

[61] Lang, S. M. ; Lockemann, P. C.: *Datenbankeinsatz*. Springer-Verlag, Berlin, Heidelberg, 1995. – ISBN 3–540–58558–3

[62] *log4j*. http://logging.apache.org/log4j/docs/, Abruf: 26. Apr. 2006

[63] Luckhardt, H.-D.: *Wissensrepräsentation und Wissensorganisation*. http://is.uni-sb.de/studium/handbuch/wissrepr/. Version: 18. Juni 2004, Abruf: 31. Mai 2006

[64] Manola, F. (Hrsg.) ; Miller, E. (Hrsg.): *RDF Primer*. http://www.w3.org/TR/rdf-primer/. Version: 10. Februar 2004, Abruf: 3. Jun. 2006

[65] McGuinness, D. L. (Hrsg.) ; Harmelen, F. van (Hrsg.): *OWL Web Ontology Language Overview*. http://www.w3.org/TR/owl-features/. Version: 10. Februar 2004, Abruf: 6. Jun. 2006

[66] Meyer, B.: *Objektorientierte Softwareentwicklung*. Carl Hanser Verlag, München, Wien, 1990. – ISBN 3–446–15773–5

[67] Meyer, B.: *Eiffel: the language*. Prentice Hall Int. Ltd, 1992. – ISBN 0–13–247925–7

[68] Much, T.: *Java für Mac OS X*. Galileo Press GmbH, Bonn, 2005. – ISBN 3–89842–447–2

[69] Much, T.: *Webseite zu dieser Diplomarbeit*. http://www.snailshell.de/uni/dipl/. Version: 2006, Abruf: 09. Jun. 2006

[70] *MySQL AB: Download Connector/J 3.1*. http://dev.mysql.com/downloads/connector/j/3.1.html, Abruf: 10. Jun. 2006

[71] *MySQL AB: MySQL 5.0 Downloads*. http://dev.mysql.com/downloads/mysql/5.0.html, Abruf: 10. Jun. 2006

[72] Noy, N. F. ; Musen, M. A.: *PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment*. http://smi-web.stanford.edu/pubs/SMI_Abstracts/SMI-2000-0831.html. Version: 2000, Abruf: 20. Mai 2006

[73] Oaks, S.: *Java™ Security*. 2. Auflage. O'Reilly & Associates, Inc., 2001. – ISBN 0–596–00157–6

[74] Oestereich, B.: *Objektorientierte Softwareentwicklung – Analyse und Design mit der UML*. 6. Auflage. Oldenbourg Wissenschaftsverlag Gmbh, München, 2004. – ISBN 3–486–27266–7

[75] *Ontologie (Informatik)*. http://de.wikipedia.org/wiki/Ontologie_(Informatik), Abruf: 1. Jun. 2006

[76] *Ontopia Solutions – Ontopoly*. http://www.ontopia.net/solutions/ontopoly.html, Abruf: 6. Jun. 2006

[77] Pepper, S. (Hrsg.) ; Moore, G. (Hrsg.): *XML Topic Maps (XTM) 1.0 TopicMaps.Org Specification.* `http://www.topicmaps.org/xtm/1.0/`. Version: 2001, Abruf: 21. Apr. 2006

[78] Pepper, S.: *The TAO of Topic Maps.* `http://www.ontopia.net/topicmaps/materials/tao.html`. Version: 2002, Abruf: 21. Apr. 2006

[79] Pepper, S. (Hrsg.) ; Vitali, F. (Hrsg.) ; Garshol, L. M. (Hrsg.) ; Gessa, N. (Hrsg.) ; Presutti, V. (Hrsg.): *A Survey of RDF/Topic Maps Interoperability Proposals.* `http://www.w3.org/TR/rdftm-survey/`. Version: 10. Februar 2006, Abruf: 2. Jun. 2006

[80] Pérez, A. G. (Hrsg.). Departamento de Inteligencia Artificial, Madrid: OntoWeb. A survey on ontology tools / Departamento de Inteligencia Artificial, Madrid. Version: 31. Mai 2002. `http://www.aifb.uni-karlsruhe.de/WBS/ysu/publications/OntoWeb_Del_1-3.pdf`, Abruf: 12. Jun. 2006. 2002 (D1.3). – Forschungsbericht

[81] Posch, T. ; Birken, K. ; Gerdom, M.: *Basiswissen Softwarearchitektur: Verstehen, entwerfen, bewerten und dokumentieren.* dpunkt.verlag GmbH, Heidelberg, 2004. – ISBN 3–89864–270–4

[82] Probst, G. ; Raub, S. ; Romhardt, K.: *Wissen managen: wie Unternehmen ihre wertvollste Ressource optimal nutzen.* 3. Auflage. Gabler, Wiesbaden, 1999. – ISBN 3–409–39317–X

[83] Reussner, R. (Hrsg.) ; Hasselbring, W. (Hrsg.): *Handbuch der Software-Architektur.* dpunkt.verlag GmbH, 2006. – ISBN 3–89864–372–7

[84] Richardson, J. R. ; Gwaltney, W. A.: *Ship it!* Carl Hanser Verlang, München, Wien, 2006. – ISBN 3–446–40425–2

[85] Schönebaum, J.: *Kooperatives und anreizbasiertes Erstellen von strukturierten Datenbeständen*, Otto-von-Guericke-Universität Magdeburg, Diplomarbeit, 30. September 2004

[86] *Semantic MediaWiki.* `http://meta.wikimedia.org/wiki/Semantic_MediaWiki`, Abruf: 6. Jun. 2006

[87] Shirazi, J.: *Java™Performance Tuning.* 2. Auflage. O'Reilly & Associates, Inc., 2003. – ISBN 0–596–00377–3

[88] Singh, I. ; Stearns, B. ; Johnson, M. u. a.: *Designing Enterprise Applications with the J2EE™Platform.* 2. Auflage. Addison-Wesley, 2002. – ISBN 0–201–78790–3

[89] *Slashdot: News for nerds, stuff that matters.* `http://slashdot.org/`, Abruf: 6. Jun. 2006

[90] Smith, M. K. (Hrsg.) ; Welty, C. (Hrsg.) ; McGuinness, D. L. (Hrsg.): *OWL Web Ontology Language Guide.* `http://www.w3.org/TR/owl-guide/`. Version: 10. Februar 2004, Abruf: 6. Jun. 2006

[91] Staab, S.: Wissensmanagement mit Ontologien und Metadaten. In: *Informatik-Spektrum* 25 (2002), Nr. 3, S. 194–209. – ISSN 0170–6012

[92] *The Apache Tomcat 5.5 Servlet/JSP Container - Realm Configuration HOW-TO.* `http://tomcat.apache.org/tomcat-5.5-doc/realm-howto.html`, Abruf: 11. Mai 2006

[93] *the friend of a friend (foaf) project.* `http://www.foaf-project.org/`, Abruf: 2. Jun. 2006

[94] *The Protégé Ontology Editor and Knowledge Acquisition System.* `http://protege.stanford.edu/`, Abruf: 6. Jun. 2006

[95] *TM4J – Topic Maps For Java*. http://www.tm4j.org/, Abruf: 24. Mai 2006

[96] *TMAPI – Common Topic Map Application Programming Interface*. http://www.tmapi.org/, Abruf: 29. Mai 2006

[97] *tolog – Language tutorial*. http://www.ontopia.net/omnigator/docs/query/tutorial.html. Version: 12. Januar 2006, Abruf: 13. Mai 2006

[98] *Topic Map*. http://de.wikipedia.org/wiki/Topic_Map, Abruf: 1. Jun. 2006

[99] *Tuning Garbage Collection with the 1.4.2 Java™ Virtual Machine*. http://java.sun.com/docs/hotspot/gc1.4.2/, Abruf: 5. Mai 2006

[100] *W3C Semantic Web*. http://www.w3.org/2001/sw/, Abruf: 29. Mai 2006

[101] *Wikipedia - Die freie Enzyklopädie*. http://de.wikipedia.org/, Abruf: 24. Mai 2006

[102] *Wikipedia:Qualitätssicherung*. http://de.wikipedia.org/wiki/Wikipedia:Qualit%C3%A4tssicherung, Abruf: 6. Jun. 2006

[103] *XHTML™ 1.0: The Extensible HyperText Markup Language (Second Edition)*. http://www.w3.org/TR/xhtml1/, Abruf: 1. Jun. 2006

Abkürzungsverzeichnis

ACID .... Atomicity, Consistency, Isolation, Durability (Eigenschaften einer Transaktion)
API .... Application Programming Interface, Programmierschnittstelle
BOT .... Begin of Transaction
CERN .... Organisation Européenne pour la Recherche Nucléaire (früher Conseil Européen pour la Recherche Nucléaire), `http://www.cern.ch/`
CMT .... Container-Managed Transaction
CSS .... Cascading Style Sheets
DBS .... Datenbanksystem
DL .... Description Logics, Beschreibungslogik
DTO .... Datentransferobjekt, von der Steuerungsschicht verwaltetes Objekt zum Übertragen von Daten aus dem Modell in die Darstellungsschicht
EIS .... Enterprise Information Systems
EJB .... Enterprise JavaBeans
EL .... Expression Language zur Auswertung von Ausdrücken mit der JSTL
FOAF .... Friend of a Friend
GC .... Garbage Collection, automatische Speicherbereinigung
HTML .... HyperText Markup Language
HTTP .... HyperText Transfer Protocol
ID .... Identität (Kennung)
IFS .... Identität, Facetten, Scope (Topic Map-Konzepte)
IoC .... Inversion of Control, Umkehrung des Kontrollflusses
ISBN .... Internationale Standardbuchnummer
ISO .... International Organization for Standardization
J2EE .... Java 2 Platform, Enterprise Edition
JAR .... Java Archive
JDBC .... Java Database Connectivity
JDK .... Java Development Kit
JNDI .... Java Naming and Directory Interface
JSF .... JavaServer Faces
JSP .... JavaServer Page
JSTL .... JavaServer Pages Standard Tag Library
JTA .... Java Transaction API
JVM .... Java Virtual Machine
LDAP .... Lightweight Directory Access Protocol
LTM .... Linear Topic Map Notation
MVC .... Model View Controller (Entwurfsmuster)
OKS .... Ontopia Knowledge Suite
OO .... Objektorientierung
OOD .... Objektorientierter Entwurf (Design)
OOP .... Objektorientierte Programmierung
OWL .... Web Ontology Language
PDF .... Portable Document Format
PSI .... Published Subject Indicator
RDF .... Resource Description Framework
RDFS .... RDF-Schema
SoC .... Separation of Concerns, Trennung von Zuständigkeiten
SQL .... Structured Query Language

TA	Transaktion
TAO	Topics, Assoziationen, Occurrences (Topic Map-Konzepte)
TM4J	Topic Maps for Java
TMAPI	Common Topic Map API, standardisierte Topic Maps API
UML	Unified Modeling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium, `http://www.w3.org/`
WAR	Web Application Archive
WWW	World Wide Web
XHTML	eXtensible HyperText Markup Language
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XTM	XML Topic Maps

Index