INSTITUT FÜR INFORMATIK

DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

An Ontology-Based Repository for Web Services

Christian Bode

Aufgabensteller:	Prof. Dr. Hans Jürgen Ohlbach
Betreuer:	Dipl. Inform. Bernhard Lorenz
	Dipl. Inform. Edgar-Philipp Stoffel
Abgabetermin:	06. August 2006

Zusammenfassung

Diese Diplomarbeit beschreibt die Entwicklung eines Systems, das verschiedene Web Services an einer zentralen Stelle speichert, und diese unter Verwendung von semantischen Informationen auffindbar macht. Um eine sehr flexible und erweiterbare Form der Speicherung zu gewährleisten, wird die Struktur der Speicherung durch eine oder mehrere Ontologien vorgegeben. In einer Ontologie werden Klassen und Attribute des Anwendungsgebietes gespeichert, sowie deren Beziehungen zueinander festgelegt. Ein Web Service wird mit Hilfe der Klassen und Attribute beschrieben und ebenfalls von dem System gespeichert. Die Suche nach den Web Services erfolgt nicht mit Schlüsselworten, sondern anhand eines Konzeptes. Dieses beschreibt die Anforderungen an einen Web Service und zwar unter Verwendung der, durch die Ontologie bereitgestellten, Klassen und Attribute. Das System ist in der Lage, die Anforderungen an einen Web Service in Beziehung zu der gespeicherten Ontologie zu setzen und somit auch zu schlussfolgern, welche Web Services den Anforderungen entsprechen.

Abstract

This thesis describes the development of a system, that stores different Web services in a repository. One or more ontologies determine the structure of the repository to ensure a highly flexible and extensible way of storing. The ontologies determine the domain of the repository by providing classes, properties and attributes as well as the relationships among them. A Web service itself is an instance of the classes, properties and attributes. The different instances are saved in the repository, too. The search engine does not use keywords, but concepts for finding Web services. Such concepts describe the requirements of requested Web services by using the classes, properties and attributes of given ontologies. The system itself can reason about the given concept and the stored information. Thus, the system uses also the implicit information provided by the ontologies to retrieve the requested Web services.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 06. August 2006

Unterschrift des Kandidaten

Contents

Contents i

List of Figures iv

List of Tables v

1 Introduction 1

1.1 Motivation . . . 1
1.2 Outline . . . 2

2 Basic Concepts 3

2.1 Repository . . . 3
2.2 Ontology . . . 3
2.3 Web Service . . . 5
2.4 Resource Description Framework (RDF) . . . 8
2.5 Web Ontology Language (OWL) . . . 11
2.6 Web Service Description Language . . . 12
2.7 OWL-S . . . 14
2.8 SOAP . . . 17

3 Related Work 21

3.1 Universal Description, Discovery and Integration . . . 21
3.2 The RDF and RDF Schema Repository Framework: Sesame . . . 23
3.3 OWL Semantic Repository: OWLIM . . . 24

4 Design of the Repository 27

4.1 Concept . . . 27
4.2 Use Cases . . . 28

4.2.1 Authorize Actor 29
4.2.2 Storing an Ontology 30
4.2.3 Registering a Web service 30
4.2.4 Querying a Web service 31
4.2.5 Subscribing for an Information Service 31
4.2.6 Unsubscribing from the Information Service 32
4.2.7 Retrieving Ontologies 32
4.2.8 Retrieving Classes, Properties or Individuals 32
4.2.9 Retrieving the Log File 32
4.3 Requirements 33
4.3.1 Requirements on the Architecture 33
4.3.2 Functional Requirements 33
4.4 System Architecture 33
4.5 Implementation Aspects 36
4.5.1 The Repository Web Service 36
4.5.2 The Repository Manager 38
4.5.3 User Authorization 40
4.5.4 The File Handler 41
4.5.5 The Object Writer 42
4.5.6 The Personal Collection 42
4.5.7 The Properties File 42
4.6 Sample Application for ORWS 44

5 Technologies 47

5.1 Snobase 47
5.1.1 Architecture and Components 47
5.1.2 Java Ontology Base Connector and Queries 49
5.1.3 Representation of Ontologies in a Database 50
5.2 Apache Axis 51
5.2.1 Instant Deployment 52
5.2.2 Normal Deployment 53
5.2.3 Development of a Client 54

5.3 Log4j 56
5.3.1 Logger Hierarchy 56
5.3.2 Appenders and Layouts 58
5.4 Apache Tomcat 58

6 Using Snobase **61**
6.1 General Remarks 61
6.2 Running Snobase within Tomcat 61
6.3 Defining Queries 62

7 Conclusion and Future Work **65**
7.1 Conclusion 65
7.2 Future Work 65

A Running ORWS **67**
A.1 Running with Eclipse 67
A.2 Deploying ORWS on Apache Tomcat 68

B Source Code **70**
B.1 Deployment Descriptor 70
B.2 Undeployment Descriptor 71
B.3 Pizza Home Service Ontology 71
B.4 Query Concept 77
B.5 Pizzeria Rosso 78
B.6 Pizzeria Bella Napoli 79

Bibliography **81**

List of Figures

2.1 Information Model [4] 4
2.2 Simple Ontology 5
2.3 Web Service 8
2.4 Service-oriented Architecture 9
2.5 RDF Graph Describing Eric Miller [28] 10
2.6 OWL-S [29] 14
2.7 Mapping between OWL-S and WSDL 17
2.8 SOAP Message Path 18
2.9 SOAP Envelope 18

3.1 High-Level Overview of Sesame's Components 24

4.1 Architecture of the Ontology-based Repository 28
4.2 Use Case Diagram 29
4.3 UML Class Diagram 35
4.4 Pizza Home Service Ontology 44
4.5 Instances and Classes 46

5.1 Schematic Overview of Snobase [27] 48
5.2 Apache Axis 52

List of Tables

4.1 Interface Methods 37
4.2 Properties 43

1 Introduction

1.1 Motivation

The World Wide Web (WWW) does not only consist of Web pages, but also of Web services. There are basically two classes of Web services: the so-called business to consumer (B2C) Web services like an API for search engines (see Google API or Yahoo API) and more important the business to business (B2B) Web services. For example, travel agencies offer flights from different air carriers. But how does a travel agency know what flights are offered, when flights are scheduled, and how many free seats are available? All the information are usually provided by Web services. The travel agent uses this information for selling flight tickets. Instead of visiting all the different Web pages he has a single application which provides all information needed and which updates the information in the background. In general, a client first looks in a directory of a Web service meeting its requirements. After inspecting the input and output parameters the client accesses the Web service. Usually, each Web service offers meta information to inform the users about their functions, however this meta information cannot be processed by a computer.

Nowadays, to find a Web service, one searches for the Web service in a directory like Woogle[1]. The user has two possibilities for finding the desired Web service. The first possibility is to check the different categories until the Web service is found. The second possibility is search by keywords. The directory returns only Web services, that are labeled with the entered keywords. It dos not find Web services which are semantically identical but syntactically different.

As a result there is a need for directories, which are enhanced with a semantic component. This semantic component should be able to reason about what kind of Web services the user wants. For that reasoning process it is necessary that the computer agent does understand and process the meta information. An ontology, for example, can be such an information representation. An ontology is a data model, that represents a domain and allows to reason about the objects in that domain. If a Web

[1] `http://haydn.cs.washington.edu:8080/won/wonServlet`

service would be such an object, then the directory would be able to understand the relations between the different keywords. This ability can be used to find more Web services which are matching the users' requirements.

1.2 Outline

A few directories are existing for Web services, but none of them have the ability to reason about the information. Thus the topic of this thesis is to develop a repository for Web services, which is able to reason about their properties.
In chapter 2 repositories, ontologies and Web services are explained. In addition the main languages for describing ontologies and Web services are described. Other directories and repositories for Web services or ontologies are introduced in chapter 3. Chapter 4 covers the type of the repository, which was been developed in this thesis. Use cases, requirements, the system architecture, as well as some implementation details are explained in chapter 4. The technologies used, especially Snobase, are introduced in chapter 5. Chapter 6 addresses problems with Snobase and their solutions.

2 Basic Concepts

This chapter describes the different concepts as well as some languages to define ontologies.

2.1 Repository

“A repository is a shared database of information about engineered artifacts” [4]. Engineered artifacts are, for example, software, applications, documents, or information systems. The main purpose of a repository is to store models and information about these artifacts. For example, during the software development process, a lot of source files are created that need to be saved in different versions, to be updated or sometimes to be deleted. A repository can help developers by storing different versions of source files or additional files like icons or documentation. In the context of this thesis ontologies and individuals are saved in such a repository.

Due to the fact that a repository is a database, the main functionalities like access control[1], concurrency[2] or integrity[3] are also subject of a repository [5]. An important difference between a repository and a database is the *information model*, which covers the structure and semantics of the stored artifacts (see figure 2.1).

2.2 Ontology

The term ontology has its origin in philosophy. The modern analytical approach to ontologies concerns basic categories and terms like ‘things’, ‘attributes’ or ‘events’. An important question is, how these different categories are related to each other, and how they interact.

In recent years ontologies are also used for information systems. An ontology can be understood

[1]Which information can be accessed

[2]access control of shared resources; avoiding deadlocks

[3]information is always valid; control of integrity rules

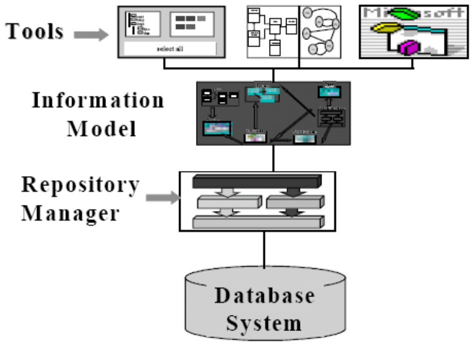

Figure 2.1: Information Model [4]

in this context “as a machine readable vocabulary that is specified with enough precision to allow differing terms to be precisely related” [26]. A more precise definition is stated in [20]:

> “An ontology defines the terms used to describe and represent an area of knowledge. Ontologies are used [...] to share domain information [...]” and “[...] include computer-usable definitions of basic concepts in the domain and the relationships among them [...]. They encode knowledge in a domain and also knowledge that spans domains. In this way, they make that knowledge reusable”

An ontology consists of a formal description of classes (general things), relations between them and properties (of these things). This formal description is normally written in a logic-based language like RDF or OWL, so that “detailed, accurate, sound, and meaningful distinctions can be made among the classes, properties, and relations” [20]. Using ontologies can provide applications in reasoning, searching, decision support and natural speech understanding. They are useful in the domain of knowledge management, intelligent databases, electronic commerce and multi-agent systems.

The main focus of ontologies are the classes, which are describing the concepts in the domain. A knowledge base covers classes and instances of classes [34]. For example, a class of wines represents all wines. Specific wines are instances of this class. The Bordeaux wine you can order in a restaurant is an instance of the class of Bordeaux wines. A class can have subclasses which represent more specific concepts than the superclass. For example, we can divide the class of all wines into red, white, and rosé wines. Alternatively, we can divide a class of all wines into sparkling and non-sparkling wines [34]. Depending on the used logic-based language the ontology author can define very granular differences between the classes. For example, he defines whether classes or properties are disjoint or

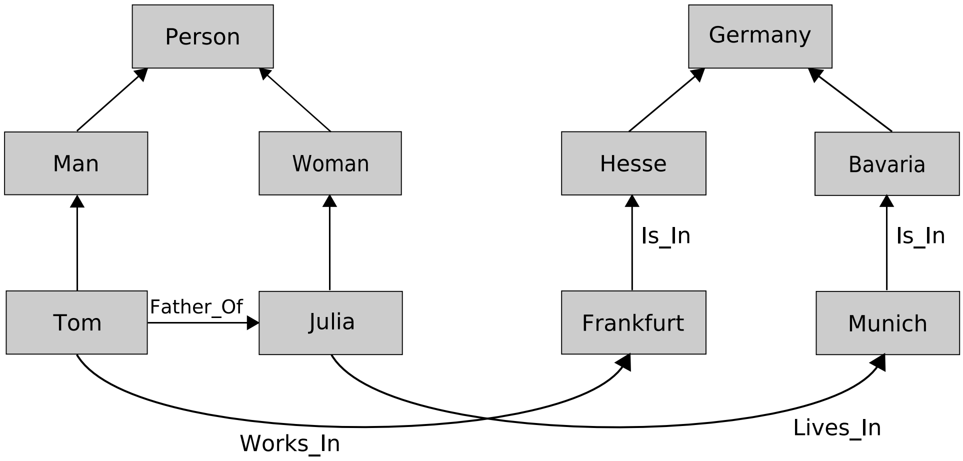

Figure 2.2: Simple Ontology

not.

A very simple ontology as shown in figure 2.2 just describes the relations between man and woman, cities and federal states.

The "Semantic Web" uses the semantic description of Web documents, if available. Usually, Web pages are designed for humans. The semantic description is stored in an ontology, the semantic information can be used by search engines, Web services or web applications.

2.3 Web Service

A Web service is a software system, whose main purpose is the interoperable machine-to-machine interaction over the Web and not the machine-to-human interaction as it would be by a Web page [7]. For example, Google provides a Web service for accessing its search engine. An application developer can use the Web service to implement his own Web search function based on Google. He does not need to parse the result page of the Google Web site, instead he can use the results (usually an XML stream) directly in his application. If a library needs a login system for its Web page, the authorization process can be delegated to Microsoft Passport [4], which is a Web service for authorizing users. A travel agency often needs to poll prices of different vendors like airlines, hotels or rental car companies, which usually do not have the same information and reservation system. The aim of Web services is simplifying this process by providing a standardized way of describing and locating such

[4] `http://www.passport.com`

information systems. Existing information and reservation systems can usually be enhanced by an interface, which provides a formalized access method over a network. This interface can be realized as a Web service. For example, this allows the travel agency to use a single application, which accesses the Web services in the background. This access is transparent for the travel agent.

Web services are built upon several open technologies to avoid security issues and to combine different Web services independent of the vendor. As a consequence the communication, for example, is based on open protocols like HTTP and the message format is based on the XML syntax. The interface definition is provided by the Web service vendor in a machine-processable format (specifically WSDL).

The disadvantages of Web services are, for example, data overhead and implementation problems due to the more complex structure than a simple RPC. The data overhead is a result of the semi-structured data of XML documents, which are exchanged between the client and the Web service. These messages are usually SOAP messages (see section 2.8).

The advantages of using the Web service technology instead of a proprietary technology are the encapsulation of the application from the interface, the usage of open standards, the interoperability between different operating systems and platforms, easy building of integrated services or easy reuse of Web services in different scenarios. Especially the encapsulation is important. If the implementation of the Web service needs to be updated or changed, then there is no need to check any dependencies from other applications, which access the service, as long as the Web service interface is kept untouched.

Technologies for Web services

- *XML* stands for eXtensible Markup Language and is designed and specified by the W3C, especially for exchanging documents in the Web. XML documents are machine-processable and also readable by humans. The document is structured as a tree where the leafs and nodes can be specified by the author. The structure of XML documents can be described with Document Type Definitions (DTD), XML Schema, Relax-NG or other systems. For more information see [9].

- *SOAP* formerly known as Simple Object Access Protocol is a lightweight XML-based messaging protocol for exchanging data between applications. The data is encoded as Web service request and response messages before sending them over a network. Due to the XML syntax

SOAP messages are independent of any operating system or protocol. Section 2.8 explains SOAP more detailed.

- *WSDL* stands for Web Service Description Language; see section 2.6.
- *UDDI* stands for Universal Description, Discovery and Integration. It is a Web-based distributed directory that enables businesses to list themselves on the Internet and discover each other. It is similar to a traditional phone book's yellow and white pages [3]. For a more detailed explanation how the UDDI works see section 3.1.

Due to the modularized concepts of Web services developers can also use only parts of existing Web services. There is no need for implementing the complete Web service interface.
The W3C defines a Web Service as follows [7]:

> "A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine process-able format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards."

The main component of a Web service is the interface description. All operations, which are offered by a Web service, are defined as the combination of incoming and outgoing messages (see figure 2.3). The implementation of the operations is transparent for the user and does not play any role for accessing the Web service. Web services are platform independent and can therefore be seen as an abstraction layer for business applications.
Web services are an implementation of the service-oriented architecture (SOA) [16] (see figure 2.4). The SOA proposes three different roles, each one can be implemented by a Web service again.

Service Provider The Service Provider offers operations, which are usually registered in a global directory service like UDDI.

Service Requester The Service Requester requests from the Service Registry the desired operations. If the desired operations are registered at the directory, the directory discloses the address and binding information of the operation implementing service. After receiving this information from the

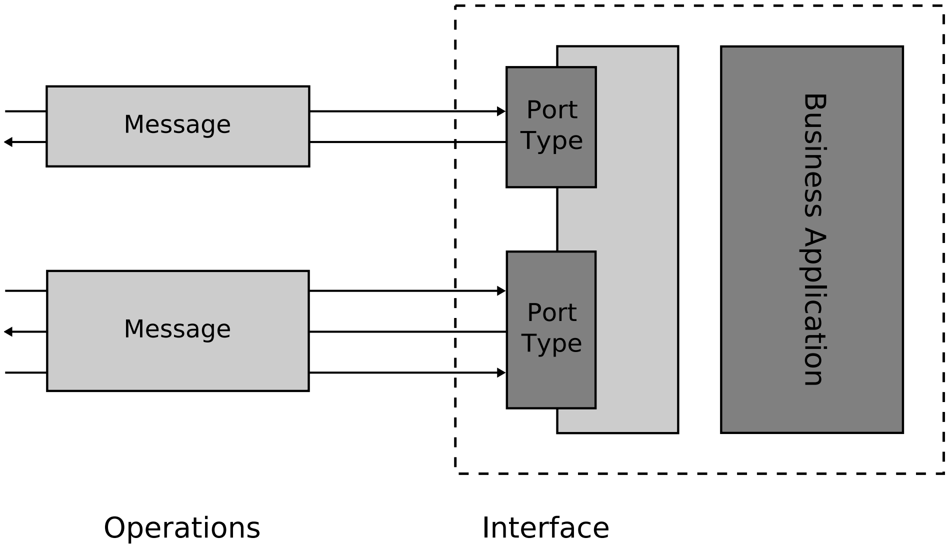

Figure 2.3: Web Service

Service Registry the Service Requester binds himself to the selected service and can call the operations.

Service Registry The Service Registry acts like a directory. The services are registered at the directory by the Service Provider. Later on, the services can be requested by the Service Requesters. So the main purpose of the Service Registry is to link the Service Requesters and the Service Providers together in a shared network.

2.4 Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a language for representing information on the Web. RDF can be used for a broad range of information, in particular for the presentation of meta information about Web documents, as author, title or last date of an update. The meta information can be useful for humans, but the main purpose of meta information is to be processed by applications. Due to the standardized syntax it is even possible to exchange information between different applications. A definition of what exchanging information between different applications means is given in [28]:

> "The ability to exchange information between different applications means that the information may be made available to applications other than those for which it was originally

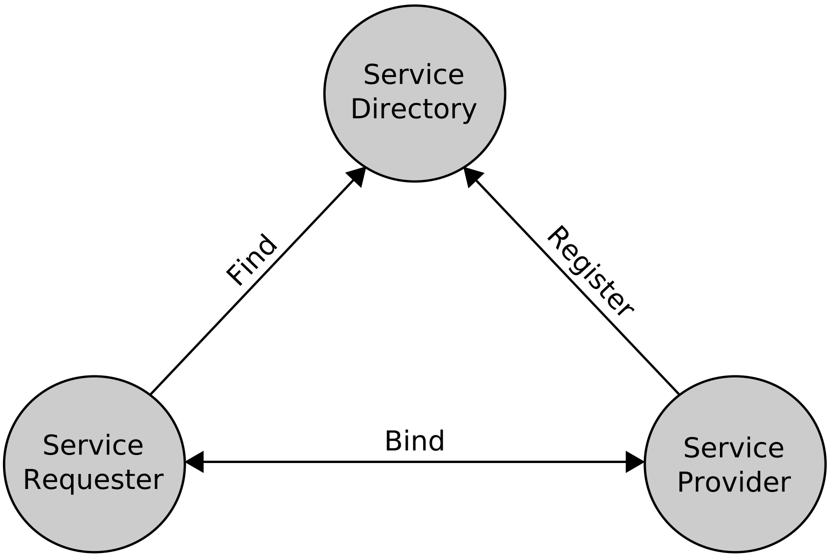

Figure 2.4: Service-oriented Architecture

created."

Basic RDF statements consist of an object-attribute-value triple. That means, an object O has an attribute A with a value V, written O(A,V). Values need not only to be of the basic types like String or Integer. They can also be Objects. Thus, an RDF statement can also be presented as a graph (see figure 2.5). Due to the possibility that an RDF statement can be the object or a value of a triple, graphs can be nested or chained [14].
In the official RDF Primer document [28], an RDF statement consists also of three parts, but the nomenclature is different. The first part is called the subject. The subject describes what the statement is. Secondly, the predicate denotes what the property or characteristic of the subject is. Thirdly, the object denotes the value of that property. For example, a Web page can be described as "http://www.example.org/index.html has a creator whose value is John Smith". Thereby the subject is "http://www.example.org/index.html", the predicate is "creator" and the object is "John Smith" [28]. This example is easy to understand for humans, but a computer agent cannot exactly identify what part of the statement is what. So there is a method needed, that each part can be clearly identified by computer agents. This identifier is the Uniform Resource Identifier (URI). With the help of the URI not only virtual things, but also real things can be described. To be more precise RDF uses *URI references*. The exact definition can be found in [28]:

"A URI reference (or URIref) is a URI, together with an optional fragment identifier at the

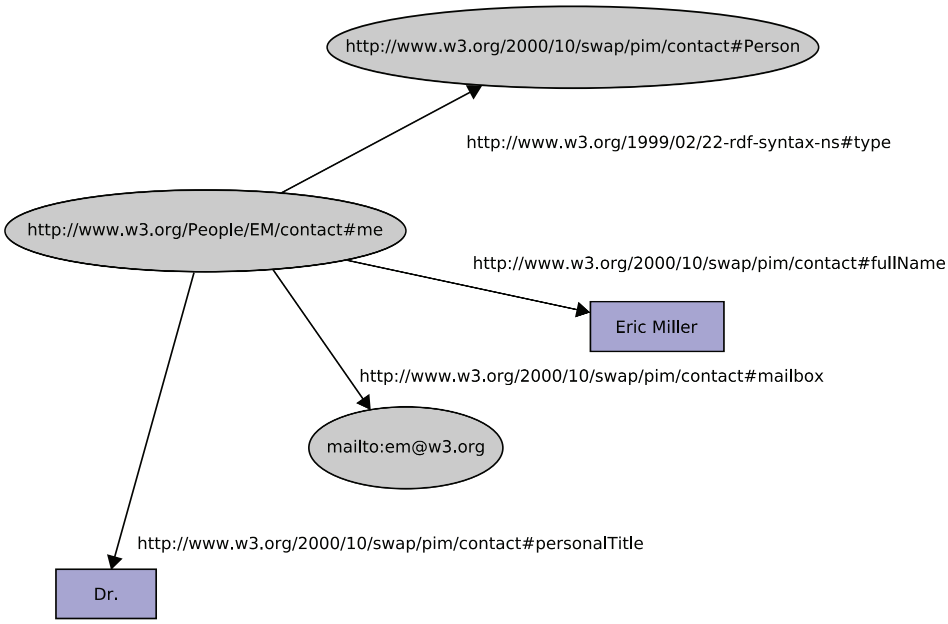

Figure 2.5: RDF Graph Describing Eric Miller [28]

> end. For example, the URI reference `http://www.example.org/index.html#section2` consists of the URI `http://www.example.org/index.html` and (separated by the "#" character) the fragment identifier `Section2`."

The syntax of RDF is based on XML. This ensures a standardized way of exchanging RDF documents between different applications or agents. Agents can easily process RDF documents, because there are many XML parsers which can read and process these documents. A simple RDF document looks like:

```
1 <?xml version="1.0"?>
   <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
               xmlns:exterms="http://www.example.org/terms/">
4  <rdf:Description rdf:about="http://www.example.org/index.html">
         <exterms:creation-date>August 16, 1999</exterms:creation-date>
   </rdf:Description>
7 </rdf:RDF>
```

Line 1 indicates that the document is an XML document based on XML version 1.0.

Line 2 defines the namespaces of *rdf* and *exterms*.

Line 4 provides a simple RDF statement.

Only simple statements about resources can be expressed in RDF. For different types of applications, users need the ability to define their own application-specific notions. For example, a travel agency may want to describe a class `flyaroundtheworld:booking` with properties like `flyaroundtheworld:departure` or `flyaroundtheworld:person`. A bookseller needs, for example, a class `thebooks:book` with properties `author` and `title`. With standard RDF it is not possible to define any application-specific classes or properties [28], but they can be described by an RDF extending vocabulary called RDF Schema.

RDF Schema does not provide any pre-formed classes or properties, but it provides the facilities for *describing* the classes and properties. In analogy to an object-oriented programming language, RDF Schema is the type system of RDF [28].

2.5 Web Ontology Language (OWL)

The Web Ontology Language (OWL) is another language for defining ontologies. It is derived from DAML+OIL [13]. Like RDF or RDF Schema it can be used to express the meaning of terms and the relations among them. Compared to RDF Schema, however, OWL has more facilities for expressing meaning and semantics [30]. OWL extends the basic statements facility of RDF and the possibilities how classes and properties are defined in RDF Schema [21]. OWL can specify classes by using different logical operations like intersection, union or complement of other classes. Properties can also be defined and, if necessary, organized in a hierarchy of subproperties. In addition domain and range restrictions can be declared. Domains are OWL classes. Ranges can be either OWL classes or externally-defined datatypes such as `String` or `Integer` [21]. Instances of classes can also be represented in OWL together with the values of their properties.

The most important extension compared to RDF and RDF Schema is the ability to define restrictions for properties or classes.

For example, in OWL it can be stated that:

- A class `PizzaBase` is disjoint from a class `PizzaTopping`.
- The individual `PizzaFunghi` is different from an individual `PizzaMargherita`.
- `HasTopping` is the inverse property of `IsToppingOf`.

OWL comes in three different versions:

- *OWL Lite* has the lowest expressiveness. It is mainly designed for applications with simple taxanomic hierarchies and simple constraints.
- *OWL DL* has already the maximum expressiveness of OWL, “while retaining computational completeness and decidability” [30]. All language constructs of OWL can be used, but they underlie certain restrictions.
- *OWL Full* is designed “for users who want maximum expressiveness and the syntactic freedom of RDF with no computational guarantees” [30].

2.6 Web Service Description Language

The Web Service Description Language (WSDL) was initially developed by Ariba, IBM and Microsoft. Meanwhile, WSDL 2.0 reached the W3C Candidate Recommendation in March, 27th 2006. WSDL is a standardized independent language for describing the interface of Web services. Due to this standardized language, it does not matter in which language the Web service itself is programmed. Therefore, WSDL can be the interface between different proprietary technologies like J2EE and .NET. Web service definitions written in WSDL are split in two different blocks. This separation ensures that some parts of the definition can be reused by other Web services. Part one is an abstract definition of the provided *operations*. An operation usually consists of several messages, which are exchanged between the client and the service. The sequence of associated messages, the so-called exchange pattern [8] as well as their types are also specified in part one. The concrete bindings and the exact definition of the message formats can be found in part two. To map the whole functionality of a Web service for description, WSDL specifies the following elements:

Types This section defines the different data type definitions, which can be used in the different messages. To ensure a higher interoperability and platform neutrality, WSDL prefers XML Schema (see [17]) as the type system, but other type systems like the one of Java are also possible.

Messages The different messages, which are exchanged during a communication between the service and client are defined. A message can be incoming or outgoing information / data or a failure response of an operation. Thereby, each message can consist of one or more logical parts. Each part is allocated to a data type, which is specified in the “Types” section.

Port Types A set of messages, that are exchanged between a Web service client and the Web service itself, is described by the port type component. These related messages are grouped into operations. An operation can be seen as a set of input and output messages, and a port type is a set of operations.

Bindings For each port type, there exists at least one binding information. So each binding element binds a concrete protocol to the port type. A binding element *must* bind a port type to a concrete protocol, but it *need not* to specify a concrete address.

Ports A port is a single concrete endpoint. It is defined by a port type and an address.

Services The services section describes all operations of the Web service, whereas each service element comprises related ports.

The WSDL specification provides with the binding element a common mechanism for binding messages to protocols. Thereby any transport protocol or message format can be assigned to the messages. Three concrete protocols are already predefined in [11], which can be used in a binding element:

- SOAP 1.1
- HTTP GET/POST
- MIME

The open policy for protocols covers also all other WSDL elements, like types or messages. Therefore, new type systems can be used for specifying the messages without changing WSDL as the description language.

Besides the technical information, the WSDL specification provides a `<document>` element, which can be added to every WSDL element. The contents of a documentation tag is addressed to humans. For example, it can be used to provide further meta information about the Web service.

The `<import>` element of WSDL enables to split up the description of a Web service to several documents. So different documents can be added to one service description. For example, the abstract definitions or concrete definitions can be reused in other contexts or data types, and their schema need only to be defined once. A separation into the WSDL description, which covers the abstract part, and the WSDL definition, which is the concrete part, can be found quite often.

2.7 OWL-S

No software agent can understand and utilize a Web service interface, which is specified by a WSDL document, without human assistance. The meaning of the different operations and attributes can not be stated with any WSDL elements. This missing semantic information must be supplemented by humans. For example, many parts of the messages are Strings, whose meaning is only obvious for humans. The need of such semantic information was one of the reasons for developing the Web Ontology Language for Services (OWL-S), which itself is an OWL ontology describing Web services [29].

OWL-S supports three main interactions with Web services: the location of a Web service with certain characteristics, the interaction with the Web service itself, and how the messages need to be serialized and accordingly how the Web service can be invoked.

A single Web service is described in the OWL-S ontology by a `service` element. It has three

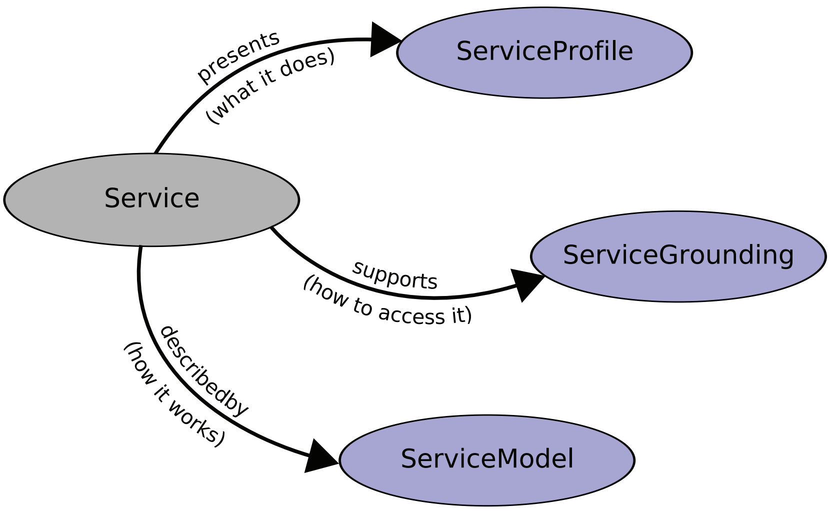

Figure 2.6: OWL-S [29]

subontologies, *Profile*, *Process* and *Grounding* (see figure 2.6). These subontologies represent the essential information about the Web service:

Profile A profile defines, what the Web service does. The `ServiceProfile` is used to advertise and to locate the Web service. It also includes information about the quality of service, limitations of service applicability or other requirements, which a service requester must fulfill [29]. The `ServiceProfile` consists of the following elements:

- textual, human-readable description of the service as well as the contact information of the Web service provider. The corresponding elements are `serviceName`, `textDescription` and `contactInformation`;
- a functional description of the service. It describes the parameters and the results generated by the Web service, together with preconditions and effects.
  For example, a book shop may require from a costumer a valid credit card prior to selling a book. After requesting the credit card number and the expiration date as the input, the output is a receipt. As an effect, the credit card is charged. What a "receipt" or a "credit card number" is, has to be specified in an ontology, which covers the designated domain;
- categorizations can be expressed with the property `serviceCategory`. It refers to common categories provided by UNSPSC[5] or NAICS[6].

Process A process defines how a service requester should interact with the service, in order to achieve the effects described in the profile. It is important to understand, that a process is not executable like an application, but it is a specification of the possible interactions between the client and the service. The process defines exactly, which inputs under which conditions in which order are required to achieve the effects described in the profile.

Atomic processes not covering any subprocesses can be directly invoked. They are executed in a single step. The service requester sends one message to the Web service, which processes it and returns the output message. For each atomic process there must exist a *grounding* element, in which the service requester can find the necessary information to construct the process relating messages. More complex processes - composite processes - can also be specified. A composite process can be

[5] `http://www.unspsc.org/`

[6] `http://www.census.gov/epcd/www/naics.html`

split up in several atomic processes or other composite processes. By using control structures such as `If-Then-Else`, `Sequence` or `Repeat-While` a possibility is given to arrange the messages, that are exchanged while communicating with the Web service.

Grounding The details of how a Web service can be accessed is specified in the grounding. This is basically related to the specification, how the inputs and outputs of an atomic process can be packed into a concrete message, how they are serialized and transported and under which address the service can be found. Generally the grounding can be seen as the mapping of an "abstract to a concrete specification of those service description elements that are required for interacting with the service" [29].
The access description is already provided by WSDL. Hence OWL-S defines how a grounding to WSDL can be realized. Both technologies complement each other, on the one hand the process model and the expressiveness of OWL's class type mechanism related to XSD, on the other hand the already existing and widely-used WSDL/SOAP infrastructure. The grounding concept of OWL-S corresponds basically to the binding concept of WSDL. Only both languages together specify a complete grounding, because both languages do not cover the same conceptual space, but both languages do overlap in the area which WSDL calls "abstract types". These are responsible for the inputs and outputs of Web services. By default, WSDL uses XML Schema for specifying abstract types, whereas in OWL-S OWL classes are used. However, the concrete binding information is missing in OWL-S, similarly WSDL has no elements for expressing the semantics of an OWL class. Figure 2.7 clarifies what is covered by WSDL and what by OWL-S.

The WSDL/OWL-S grounding is based upon three correspondences between OWL-S and WSDL:

- An atomic process in OWL-S is mapped to an operation in WSDL.
- The set of inputs and the set of outputs of an atomic process in OWL-S corresponds to one input message or one output message of a WSDL operation.
- The types of the inputs and outputs correspond with the extensible concept of abstract types in WSDL. If the WSDL types are defined in XML Schema, a mapping of the XML serialization of the OWL concepts to the relating XML Schema data types can be given in XSLT [7].

[7] `http://www.w3.org/TR/xslt`

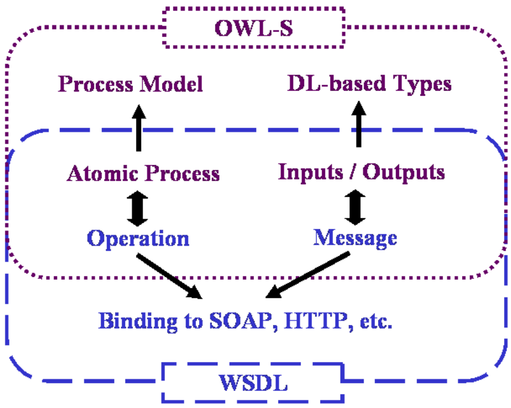

Figure 2.7: Mapping between OWL-S and WSDL

2.8 SOAP

In the following, the parts of SOAP, which are relevant to this thesis will be introduced. More detailed information can be found in the SOAP specification [32]. Until SOAP version 1.2 was released, SOAP was the abbreviation for Simple Object Access Protocol. Since version 1.2 reached the W3C recommendation status, SOAP means no more “Simple Object Access Protocol”, but is now the name of the complete framework.

SOAP was designed to build a framework for exchanging messages in a decentralized, distributed environment. SOAP is stateless and based on an unidirectional communication model: a sender sends a message to a receiver. Of course, this is not sufficient for complex applications or Web services, but these messages can be combined to more complex interacting patterns like request/response.

Due to the XML based message structure different platforms or applications can communicate with each other. Furthermore, the specification of SOAP outlines no concrete protocol. Many protocols like SMTP or POP3 can be used as the transportation protocol, but the most commonly used is HTTP (see figure 2.8).

The structure of a SOAP message is fairly simple. It has one `envelope` element, which contains an optional `header` and a `body` element (see figure 2.9).

SOAP Header The SOAP header contains meta information such as requirements on the quality of service. or information for a middleware. Complex services require sometimes more than just a

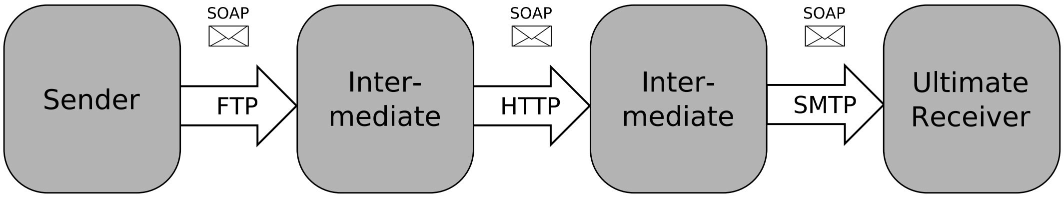

Figure 2.8: SOAP Message Path

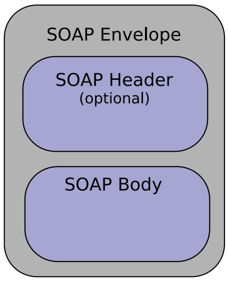

Figure 2.9: SOAP Envelope

simple message or one receiver. Therefore, a whole message path can be defined in the header. Some of the header blocks can be processed by one of the SOAP nodes along the path (see figure 2.8). Header blocks can also be deleted or new blocks can be added along the path [32].

SOAP Body The body element covers the actual data which is to be transmitted. In addition a `fault` element specifies what to do with exceptions. SOAP provides a simple type system, which is a combination of the most-used types in the different programming languages. Complex data types can be constructed out of the simple data types. Binary data can also be transferred.

Take a simple transaction like the payment of a book and assume, that only the credit card number and expiration date are missing to complete the payment. A SOAP message from the client to the server with this information can look as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
```

```
  <ns1:validate
   soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:ns1="urn:CardValidator">
   <number xsi:type="xsd:string">1234 5678 9876 5432</number>
   <valid xsi:type="xsd:string">12/08</valid>
  </ns1:validate>
 </soapenv:Body>
</soapenv:Envelope>
```

and the server answers that the credit card was validated successfully:

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <soapenv:Body>
  <ns1:validateResponse
   soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:ns1="urn:CardValidator">
   <addReturn xsi:type="xsd:boolean">true</addReturn>
  </ns1:validateResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

One disadvantage of SOAP can easily recognized from this simple example. The overhead of sending just a simple information like a credit card number is quite big. Compared to the other methods like a remote procedure call, the generating and parsing of these message documents cost more CPU time and the transfer more network bandwidth.

3 Related Work

In the following a few proposals are made how different Web services of different Web service providers can be managed and how ontologies can be stored.

3.1 Universal Description, Discovery and Integration

The focus of Universal Description, Discovery and Integration (UDDI) is to offer a business registry for Web services. This directory presents a standardized framework "for businesses to build a registry, discover each other and describe how to interact over the Internet" [15]. Three different kinds of information are stored in the UDDI registry:

- address, name and other identifiers of the business - like White Pages;
- industrial categorization according to standard taxonomies - like Yellow Pages;
- documentation of technical information about the services - like Green Pages.

An UDDI directory can be interrogated by SOAP messages and can deliver the WSDL documents, which are describing the protocol bindings and other specifications like the message format or input and output parameters. So Web service providers can advertise their service in a central registry including all the meta information about the service. On the other side, consumers can discover services which match their requirements and retrieve the information for accessing the services.

The primary goal of the UDDI initiative was to develop one public central registry for Web services called "Universal Business Registry" (UBR), but there have not been so many businesses which wanted to register their service in the UBR. Since UDDI has become an important part in the Web service infrastructure, many UDDI directories were developed. Contrary to the initial intention they were not developed as a single directory, but instead as many different directories behind the firewalls of businesses. Meanwhile version 3 of the UDDI specification is published. Mechanisms are foreseen which allow an interaction and linking between the different directories for building one big network

of directories. This kind of distributed directories network is similar to the Domain Name System [33]. The UBR was closed down at the beginning of this year.
Currently in an UDDI registry there is no possibility of storing semantic information. This "creates inefficiencies in exploiting the Web service registries" [15]. As already mentioned in section 1.1, Web services described by semantic information allow automatic Web service discovery, invocation, composition and interoperation [15]. For example, locating a specific service is quite cumbersome. Due to the fact, that the service name need not be related to the topic of a service, a user needs to browse through the different categories of the UDDI registry and check every single service whether it matches his needs. Especially, if the service is very specific, this could be very laborious.
A solution for this problem is introduced in [15]. They "describe a framework for exploiting the semantics of Web services through UDDI registries" [15]. There are a lot of active research activities in the field of Web service description with semantics such as [31] or [12]. An ontology for e-businesses should cover functionalities which, according to [15], are necessary for the discovering process. For example, it should be possible to discover services with a complementary functionality. The upper ontology, which should provide these functionalities, is based on DAML-S [2], which is the predecessor of OWL-S [29]. The top level class *service* has three properties:

- *presents*: The property *presents* contains the so-called *ServiceProfile*. It specifies what the service does for the users and what the service requires from its users. A service-seeking agent gets the necessary information to verify, whether the service matches the requirements or not. All needed input and output parameters are also stated in the properties of the ServiceProfile; other preconditions or postconditions and binding patterns can also be found. Special properties are *serviceParameter*, *serviceType* and *serviceCategory*.

- *describedBy*: The class Service is *describedBy* a *ServiceModel* to specify how it works.

- *supports*: The property *supports* provides a *ServiceGrounding*, which specifies how the service can be used. A service agent uses the information stored in *ServiceGrounding* to access the service. The *ServiceGrounding* overlaps with the WSDL specification (see section 2.7 and Figure 2.7).

3.2 The RDF and RDF Schema Repository Framework: Sesame

Sesame is an open source Java framework for semantic repositories. Originally, it was developed by Aduna[1] as a research prototype for the EU research project On-To-Knowledge[2]. Nowadays it is further developed and maintained by Aduna in cooperation with NLnet Foundation.

Sesame provides storing, querying and reasoning mechanisms for RDF (see section 2.6) and RDF Schema. Sesame can be used in two ways: as a database for RDF or RDF Schema, or as a Java library for applications. Sesame supports an application developer in dealing with RDF files like parsing, interpreting, querying and using relevant information. The application developer can use these functions in his own application or in a separate database or on a remote server [35].

The storage of the RDF or RDF Schema files is separated from a concrete database management system (DBMS) by the *Storage And Inference Layer* (SAIL). SAIL is a single architectural layer of Sesame, which abstracts the storage and inference details. It provides a programming interface, which allows one to implement a repository on top of different DBMS. Therefore, it is possible to use a DBMS, which fulfills best the requirements of an application [10].

The functional modules of Sesame are clients of the SAIL API. Currently, there are three such modules available: The RQL query engine, the RDF admin module and the RDF export module. The implementation of RQL (see [23]) in Sesame follows not exactly the definition in [22]. Instead it "features better compliance to W3C specifications, including support for optional domain- and range restrictions as well as multiple domain- and range restrictions" [10]. The RQL query engine optimizes an RQL query before the query is executed. Therefore the query will be parsed and a query tree is created. The optimization process has mainly to do with rearranging the query sub-clauses. At the end, the query tree is equivalent to the original one, but can be evaluated with a better performance. Furthermore, each basic unit of the model tree is able to evaluate itself by calling some SAIL functions. So the RQL query engine itself processes the main bulk of the query evaluation. This makes Sesame more independent from the different DBMS [10].

The admin module provides two main functions: inserting RDF or RDF Schema data into the repository as well as a function to clear the whole repository. A partial delete function is not implemented yet. The data will be delivered to the admin module as triple statements `(Subject, Predicate,`

[1] `http://aduna-software.com/index.html`
[2] `http://www.ontoknowledge.org/`

`Object)`. The admin module then tries to insert the data into the repository by calling the corresponding SAIL functions.
The RDF export module is just able to export the contents of the repository as a formatted RDF file. It can be chosen, whether only the data, the schema or both should be exported.
Besides these functional modules some protocol handlers are provided. The protocol handlers are intermediaries, which are placed between the different modules and the clients. The protocol handlers provide again a maximum of flexibility, because it is possible to implement and add new protocol handlers.
The different components of Sesame and their dependencies is shown in figure 3.1. The Sesame

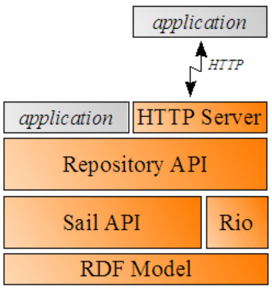

Figure 3.1: High-Level Overview of Sesame's Components

framework is based on the RDF Model. It defines interfaces and provides an implementation for all basic RDF entities. Rio stands for "RDF I/O" and consists of a few parsers and writers for various RDF file formats. Rio can be used as a stand-alone application, too. The Storage And Inference Layer (Sail) API is already explained above. The Repository API supports developers with a large number of methods to handle RDF data. In particular there are functions for uploading data files, querying, extracting and manipulating data. On top of the different layers is the HTTP server. This provides a servlet, which can be used to access the Sesame repositories over HTTP.

3.3 OWL Semantic Repository: OWLIM

OWLIM is an implementation of SAIL in the Sesame framework. It enhances the Sesame framework to allow partial reasoning over OWL DLP, which is also a language for describing knowledge bases (see [36]). For the reasoning process the complete repository is loaded into the main memory. This allows a faster and more efficient way of retrieval and query answering. Nevertheless comprehensive persistence and backup strategies are provided.

The used reasoning engine is *TRREE*[3], which stands for Triple Reasoning and Rule Entailment Engine. The reasoning process is based on forward-chaining of entailment rules over RDF triple patterns with variables [25]. TRREE is configured by a set of rules. For the support of OWL only some OWL constructs are provided. These constructs underlie also some restrictions. For more information, which OWL constructs are provided and which restrictions apply see [25].

OWLIM underlies some limitations, which makes OWLIM not applicable for the repository of this thesis. For example, the reasoning engine cannot process all OWL statements. As a result complex ontologies written in OWL cannot be loaded and reasoned about. The in-memory reasoning strategy is limited by the available free main memory. Another disadvantage is that deletion of statements is very slow, which makes it not very suitable for applications where deleting is a typical transaction (see [24]).

[3] `http://www.ontotext.com/trree/`

4 Design of the Repository

4.1 Concept

The goal of this thesis is to develop an ontology-based repository for Web services. As already explained in section 2.1, a repository can be seen in this scope as a kind of database system (DBS). It stores ontologies and instances of classes, where the instances are the Web services. The domain of the repository is specified by an ontology (see section 2.2). In addition to the usual description of the domain, the ontology must provide classes and properties that define a Web service. It is necessary that there is at least one field that specifies the URL of the Web service. In addition the URL to the WSDL document should also be provided. The ontology axiomatizes the structure of the domain including the classes and properties of the corresponding Web services. The Web service itself is an instance of such a class. Already existing ontologies, which determine the scope of the repository, can be easily exchanged or enhanced to cover other domains. If the data would be stored in a database, the complete data schema must be changed and applied to the new domain. The ontology-based approach allows a higher flexibility for covering different domains.

Storing Web services and ontologies is not the only purpose of the repository. In particular it supports users in finding the desired Web service. The information stated in ontologies can be processed by software agents. Such an agent is able to understand the relations between the different classes and properties. In case of this work an agent is an inference engine. For a user who wants to find a Web service, which is specified by some properties, an inference engine can use these properties and their relations stored in the repository to process the request. In contrast to a normal search engine that only finds Web services, which exactly match the requirements, an inference engine can find more Web services, because it can reason about the statements describing the domain and the Web services to fulfill the request. An inference engine does not only search for the keywords, but exploits semantic relations between them. Hence, the structure of the query is different from a keyword search. The desired Web service must be specified by a concept, which is built upon the properties and classes provided by the stored ontologies. This concept is then processed by the inference engine. Therefore, depending on the quality of the ontology, an ontology-based search should find more Web services than a pure keyword-based search.

An ontology-based repository consists of an ontology, which determines the domain of the Web services and an inference engine for finding Web services. Instances of the classes, which represent Web services, must also be stored within the repository. The client interface to the system is also a Web service, which provides the most flexible usage from developers' point of view. The design of an ontology-based repository is shown in figure 4.1.

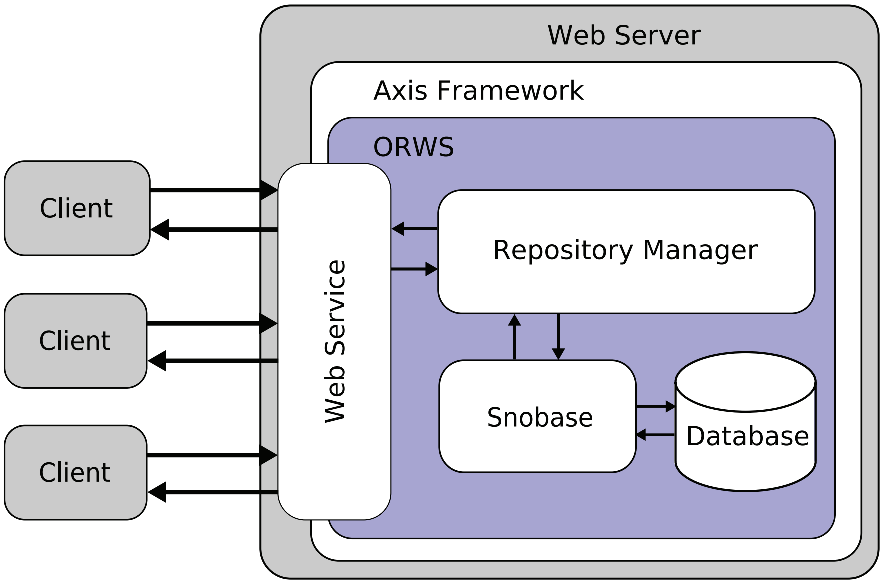

Figure 4.1: Architecture of the Ontology-based Repository

In the following the Ontology-based Repository for Web Services Web service is called *ORWS*. ORWS runs within the Web service framework Axis (see section 5.2), which itself is executed by a servlet engine, the proposed servlet engine is Apache Tomcat (see section 5.4). The core functionalities of the repository are provided by Snobase (see section 5.1).

4.2 Use Cases

Depending on the different roles, how the repository can be used, three actors can be specified: a user, a Web service provider (WSP) and an administrator. The user uses the repository like a search engine.

He wants to find a Web service. The client accesses the repository via the Web service interface. The WSP provides one or more Web services. He registers his Web services in the repository in order to allow users to find them. The administrator's major task is registering new ontologies in the repository to guarantee, that the repository fits all kinds/categories of Web services, which WSPs will register. If WSPs cannot find a class to inherit from, they need to contact the administrator to provide further classes.

Reflecting these three different actors the following use cases can be stated (see figure 4.2).

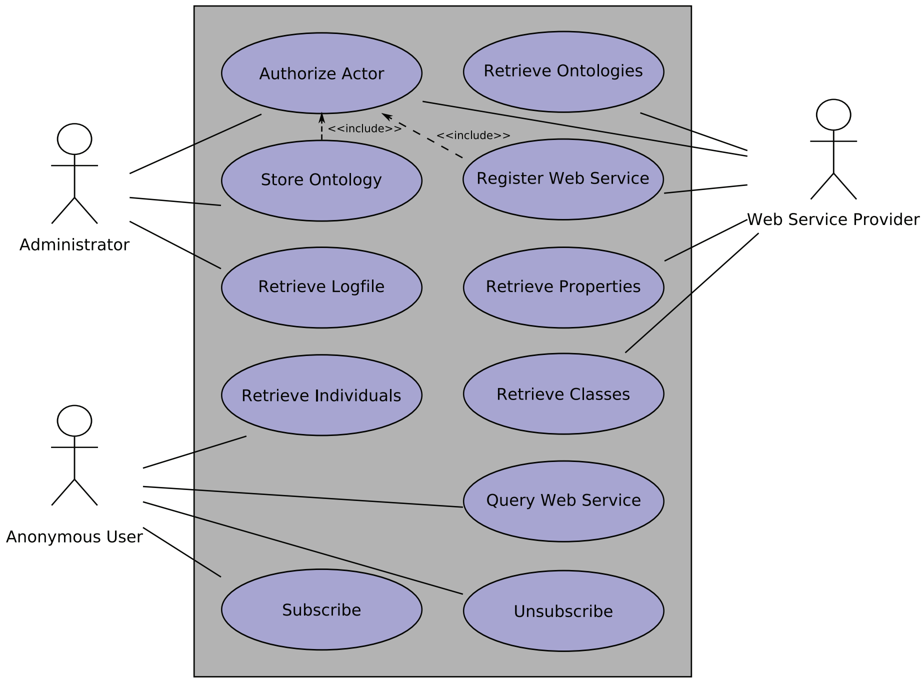

Figure 4.2: Use Case Diagram

4.2.1 Authorize Actor

Some of the functions provided by ORWS will be only processed, if the caller is logged on at ORWS. Therefore ORWS provides a simple authorization mechanism. Before one of the restricted functions such as retrieving the logfile are processed, the caller must send his username and password to ORWS. The given username and password are then compared with possible user accounts. In case of a positive match, the caller is registered at ORWS. Depending on the role, which can be an administrator, a WSP or an anonymous user, the functions are processed now. If the caller is not logged in, he can still call the functions, but without effect.

4.2.2 Storing an Ontology

For enabling the repository to reason about different classes, properties and attributes of the Web services, it has to provide one or more basic ontologies that describe the basic structure of the repository. By exchanging the basic ontologies, the domain of the repository can be changed easily. Consequently, before any Web service can be registered, these ontologies must be stored in the repository first.

The administrator must publish the desired ontology on a place, where it can be accessed by the repository Web service. Before he is allowed to register that ontology, he needs to log in first. Therefore he sends his username and password to ORWS. After a successful login he sends the URL of that document to ORWS, that loads the ontology and stores it in the repository. If the linking process was successful, the administrator gets a positive message, otherwise an error message is returned.

4.2.3 Registering a Web service

The ontologies of the repository can be compared with the data schema of a database. The instances of the classes are like the data in a database. So before any Web services can be found, they must be registered at the repository. There are two scenarios how a Web service can be registered. In both scenarios the WSP must first log in at ORWS (see section 4.2.1)

In the first scenario, the WSP registers a Web service for the first time. Before the WSP knows how to describe the Web service he must get an overview of which domains are already covered by the repository. Therefore, he can retrieve a list with all stored ontologies, which can be opened by calling the corresponding URL. After choosing an ontology or a combination of them, which cover his Web service, he must create an instance of the ontology, that is the semantically-enriched description of his Web service. Once he has written the description, he has to send the URL of the published document to ORWS that loads the document from the given URL and stores the content in the repository. The WSP then gets a status message.

In the second scenario, the WSP already knows the applicable ontology, he just needs to write the description, publish it and send the URL to the repository Web service. If the registering process was successful, the WSP gets a status message.

In both scenarios - a successful registration assumed - the Web service can then be found by the query mechanism.

4.2.4 Querying a Web service

Once the ontologies are stored and some Web services are registered, users can search them. In contrast to a simple search engine, the search function of the repository is not based on keywords - instead, concepts of the service requesters are checked. A concept consists of a collection of triple-based statements (known from RDF; see section 2.4), which specify the desired properties a Web service must have. For example, if you want to search for all map vendors, you need to build the following statement: (?X, isA, MapVendor). Every instance, which *isA MapVendor* whereas *MapVendor* is a class, will be bound to the variable *?X*. After receiving the concept, the repository returns the results. More complex queries must always use the triple-based structure.
Hence, before the user can send a query to ORWS, he needs to write down all statements in one document. After passing the URL of the published file to ORWS or sending the content directly to ORWS. ORWS loads the statements into the runtime environment and executes the whole statement collection. If some instances fulfill the query, a corresponding XML answer is sent back to the client. If there are no instances ORWS returns an empty document.

4.2.5 Subscribing for an Information Service

After launching such a system, ORWS does not cover every domain and does only contain a few Web services. Consequently it can happen that a query does not deliver any results or the results do not satisfy the user's requirements. As a service, ORWS can inform the user, if a request can be fulfilled later on. Therefore the user can send a concept covering his requirements related to the Web service and providing his email address to ORWS. ORWS saves the concept and returns the new filename of the document, that will be needed for unsubscribing from the information service. Each time a new Web service is registered, ORWS loads these concepts again and performs the search. If the query was positive, an email is sent to the user containing the result.

4.2.6 Unsubscribing from the Information Service

When the user does not want to be informed anymore about new Web services matching his requirements, he needs to unsubscribe from the information service. It requires to send the filename, which he got during the subscribing process, to the ORWS. If the file exists ORWS will delete the concept from the server and return a status message to the user.

4.2.7 Retrieving Ontologies

Before a WSP registers his Web service for the first time, he needs to check if his Web service can be described with the already stored ontologies. Some of the ontologies can contain many classes, properties and attributes. Viewing such huge documents as plain text is not very efficient. To offer the WSP an easy way for exploring an ontology, he can open each ontology saved on the server by calling the appropriate URL. A list with the URL can be retrieved from ORWS. The document can then be viewed with any ontology editor (e.g. Protegé[1]).

4.2.8 Retrieving Classes, Properties or Individuals

All data which are stored in the repository can be retrieved by a user, WSP or administrator. But to enhance the usability, four standard queries are predefined in ORWS. There are functions for retrieving all classes, properties, ontologies or individuals of a class which are currently available in the repository. ORWS executes the corresponding functions of the repository and returns the result.

4.2.9 Retrieving the Log File

Only the administrator is allowed to retrieve the log file, which contains information about all interactions between the different users and ORWS. Consequently, he must log in first at the ORWS. In case of a positive login he can retrieve this file by calling the appropriate function. ORWS will then load the log file and send it back to the administrator. Due to the continuous logging, the log file is just a snapshot.

[1] `http://protege.stanford.edu/`

4.3 Requirements

In this section the requirements on the architecture of the repository are explained. On the one hand there are the general requirements on the infrastructure and on the other hand there are the specific requirements on the functions, which the repository must fulfill.

4.3.1 Requirements on the Architecture

The architecture of the repository is mainly determined by Snobase (see section 5.1). Snobase is a Java package that provides the core functionalities of the repository, like the storing and querying mechanisms. Hence it is obvious to implement the other software components also in Java. In order to allow a wide spectrum of possible clients, the repository should be implemented as a Web service.

4.3.2 Functional Requirements

The functional requirements are basically stated by the use cases. The following list will summarize the functionalities the repository should provide:

- Authorization mechanism for users; a role must be assigned to each user.
- Storing of ontologies in a repository in such a way that it is possible to retrieve the information again.
- Inferencing mechanism for reasonin about the information stored in the ontology.
- Functions to register a Web service in the repository as an instance of the stored ontology.
- Information service for new Web services including subscribing and unsubscribing functions for the users.

4.4 System Architecture

In order to run ORWS as a Web service, an application server such as JBOSS, Apache Geronimo or Apache Tomcat is needed. In this thesis the Apache Tomcat server is used (see section 5.4). The main reason is a good documentation and the support for the Axis framework (see section 5.2). Axis is a

SOAP engine, that provides functions to send SOAP messages over a network. The communication between ORWS and the clients is done via SOAP messages. Furthermore Tomcat and Axis are open-source implementations of the underlying technologies, so that no licenses need to be purchased. The support of Tomcat and Axis within several Integrated Development Environments (IDE) such as Eclipse or NetBeans is quite well, too.

Major parts of ORWS The classes of ORWS are divided into the packages `server` and `orws` (see figure 4.3). The classes within the `server` package are mostly generated by the WSDL2Java tool of Axis. They are needed to communicate with the clients. The core functions of ORWS are provided by the classes within the `orws` package.

- The `RepositoryManager` provides the core functions of the repository, which includes operations for storing, retrieving or searching for Web services.
- Functions relating to the user management can be found in the `RSession` class. The instance of the actually logged in user is hold as an `RSession` object.
- The class `OR` defines all functions of the repository which can be accessed by a client via SOAP messages. The class `OR` is the programming language specific definition of the repository's interface.
- The `ORSoapBindingImpl` implements the interface and - aside from some data conversions - delegates the functions to the `RepositoryManager`.

In addition some utility classes are available:

- The class `User` represents a single user.
- The `Serializer` converts complex data types like Vectors or Arrays into an XML structure or a string representation, which is needed to send the data in the SOAP body.
- The `FileHandler` provides some functions for reading, writing or deleting files or directories.
- The `ObjectWriter` reads XML documents and converts them into objects or converts the objects into an XML representation.

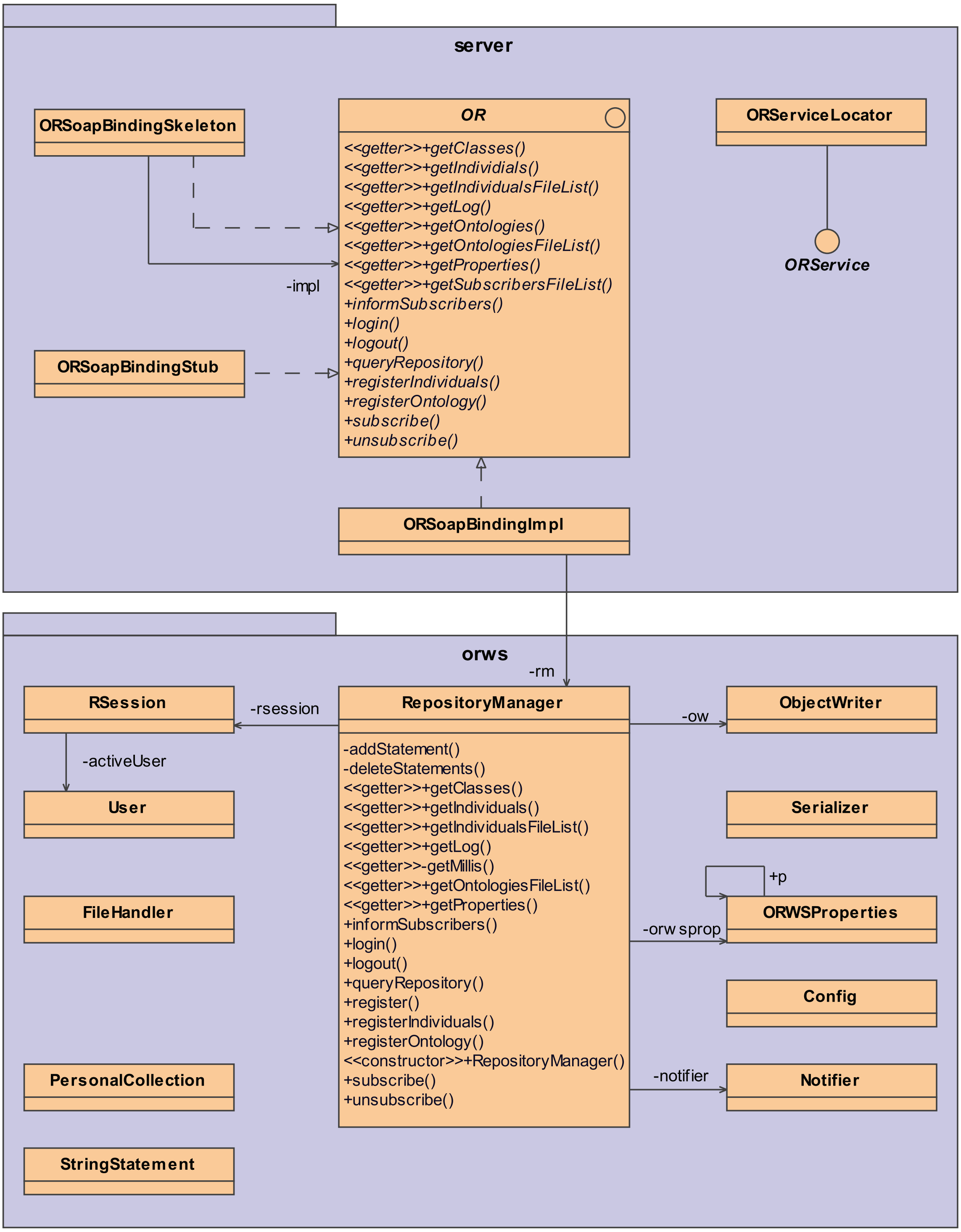

Figure 4.3: UML Class Diagram

- The class `ORWSProperties` provides the properties, which are specified in the file *repository.properties*. This file is read by the `Config` class.

- The `Notifier` provides functions to send emails.

4.5 Implementation Aspects

4.5.1 The Repository Web Service

Before a Web service can be accessed it must be deployed in a Web service runtime environment. The Web service environment is, if Tomcat and Axis are used, a servlet that provides the necessary functions like instantiation of the Web service or delegation of messages. A Web service can have three different scopes: *request*, *session* or *application*. This scope must be defined in the `deploy.wsdd`, which is required to deploy a Web service to the server. Therefore, within the `<service>` section of the desired Web service a new tag `<parameter>` must be added with an attribute `name="scope"` and an attribute `value`. Depending on the scope, the Web service will be instantiated on every request, once in a session or as a singleton. In the latter scope, all clients are accessing the same instance of the Web service, which means, for example, all attributes are shared. If the service was already deployed, the scope must be also defined in the `server-config.wsdd`, which can be found in the `WEB-INF` directory of the server project and is the configuration file of for the Axis SOAP engine. If sessions are used the client must maintain the session. Therefore, the function `setMaintainSession(true)` of the `call` object must be called with the parameter `true`. If a client was generated by the WSDL2Java tool of Axis, the method call must be added in the `<servicename>SoapBindingStub.java` class within the `createCall()` method.
ORWS offers several functions for using the ontology-based repository. These functions are listed in table 4.1. Depending on the authorization level, the call of some functions has no effect. For example, the function for registering new ontologies can be called by every client, but only after a successful login of an administrator the ontology is really saved in the repository. A detailed description how the Web service itself works or which classes are needed can be found in section 5.2.2.

Method	Description
`getClasses()`	returns a list with all classes available in the repository
`getIndividuals(String class)`	returns a list with all individuals of the given class available in the repository
`getIndividualsFileList()`	returns a list with all individuals files stored on the server
`getLog()`	returns the log file
`getOntologies()`	returns a list with all ontologies available in the repository
`getOntologiesFileList()`	returns a list with all ontology files stored on the server
`getProperties()`	returns a list with all properties available in the repository
`getSubscribersFileList()`	returns a list with all submitted query concept files of the information service
`informSubscribers()`	informs users, whose query concepts matches some individuals
`login(String name, String pw)`	registers the client on OWRS
`logout()`	logs the client out
`queryRepository(String doc)`	expects an URL or document containing a query, executes the query and returns the result
`registerIndividuals(String url)`	registers an instance of classes in the repository
`registerOntology(String url)`	registers a new ontology in the repository
`subscribe(String doc)`	expects an URL or document containing a query concept and an email address of the subscriber
`unsubscribe(String identifier)`	removes the given query concept from the information service

Table 4.1: Interface Methods

4.5.2 The Repository Manager

The `RepositoryManager` is the central class of ORWS. Its main task is the interaction with the repository, whose core functionalities are provided by Snobase and the user authorization (see section 4.5.3). In the following the initialization process and some important functions are explained.
The constructor of the `RepositoryManager` object initializes first the logging mechanism log4j (see section 5.3) and then gets a model from the repository by the `DriverManager` of Snobase. The model represents the ontologies, that define the domain of the repository. Internally the model represents what the JDBC connection is for a DBS. It provides all methods for managing the ontologies. Later on, the individuals as well as new ontologies will be stored in the same model. Normally the general `getModel` method should be used to open a connection to the repository. More specific models can also be retrieved by calling the appropriate functions. These specific models can yield ontologies written in the different ontology languages like RDF, OWL or OWL-S. When using the language-specified methods such as `getOWLModel`, the model is restricted in the operations supported by the respective languages. In the scope of this thesis the method `getOWLModel` is used in order to profit from the expressiveness of *OWLOntologies*.

Registering an ontology Due to the fact, that Snobase already provides the functions for storing ontologies, all that must be done by the `RepositoryManager` is to call the appropriate function `link` with the URL, which points to the ontology. The model loads the ontology from the given URL and adds it to the repository. As already mentioned in section 4.2, the ontology document is not only stored in the repository, but also copied to the server. These copied files can be requested individually by the actors. The copying process itself is done by the `FileHandler`.

Running a query After passing the URL of the query document to ORWS, the `RepositoryManager` opens the document and passes the `InputStream` to the `ObjectWriter`, which creates a `PersonalCollection` instance. A `PersonalCollection` represents the XML structure of the query document. The class also provides functions for adding and retrieving of statements as well as some meta information. The method `createPersonalCollection` of the `ObjectWriter` parses the document and fills the attributes with data. Thereby the query document must observe the following structure:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<statementcollection id="Ch_1150993744000">
  <subscriber>
    <name>Christian Bode</name>
    <email>bodec@ifi.lmu.de</email>
  </subscriber>
  <statement>
    <subject>RDFVarible:?X</subject>
    <predicate>
        http://www.w3.org/1999/02/22-rdf-syntax-ns#type
    </predicate>
    <object>http://www.ifi.lmu.de/~bodec/pizza_hs.owl#Pizza</object>
  </statement>
</statementcollection>
```

The RepositoryManager parses the document and creates a new statement collection, which collects all statements of the query. Every single statement of the PersonalCollection, which needs first to be converted into an RDFStatement, is then added to the statement collection. An RDFStatement represents a (subject, predicate, object) triple. If all statements are added to the statement collection, the query is executed by calling the select method of the model. After completing the query the model returns the results as an RDFResultSet object. The RDFResultSet is structured as a table whose data can be explored via a cursor, that is initially positioned before the first row. By calling the next() method in a loop, the cursor is moved from one row to the next. The RDFResultSet provides also the system with some meta data, like the number of columns, which can be retrieved by the getMetaData() method. So the different results can be retrieved independently of the number of columns by calling the getString(int columnindex) method. The RDFResultSet cannot be serialized. The Serializable interface is not implemented, but the serialization is necessary if the result is to be sent via a SOAP message. In order to allow that, the result is converted into an XML document. In case of several variables, which correspond to the columns of the data table, they are explicitly listed in the <variable> element. The attribute id indicates the name of the variable. This document is then sent to the client.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<query>
  <result>
    <row index="1">
```

```
      <variable id="?X">
        http://www.ifi.lmu.de/~bodec/pizza_hs.owl#pizza_funghi
      </variable>
      <variable id="?Y">
        http://www.ifi.lmu.de/~bodec/pizza_hs.owl#pecorino
      </variable>
    </row>
    <row index="2">
      <variable id="?X">
        http://www.ifi.lmu.de/~bodec/pizza_hs.owl#pizza_margherita
      </variable>
      <variable id="?Y">
        http://www.ifi.lmu.de/~bodec/pizza_hs.owl#mozzarella
      </variable>
    </row>
  </result>
</query>
```

Standard Queries As already mentioned in section 4.2.8, the `RepositoryManager` provides the actors with four standard queries: `getClasses()`, `getProperties()`, `getOntologies()` and `getIndividuals(String owlclass)`. All queries return a sorted list with the appropriate results. The first three queries operate on all ontologies which are available to the `RepositoryManager`. The latter only returns the individuals of the specified class. Internally the methods are based on functions provided by the model. For example:

```
String strClass = "http://www.ifi.lmu.de/~bodec/pizza_hs.owl#pizza";
OWLOntology model = DriverManager.getOWLModel(jobc:orws));
OWLClass owlclass = model.createOWLClass(strClass);
RDFResultSet resultSet = model.selectOWLIndividual(owlclass);
```

4.5.3 User Authorization

When a client accesses ORWS for the first time, `RSession` loads a file containing all users, that are allowed to execute the restricted functions. The exact filename is specified in the properties file (see section 4.5.5). For each user defined in the file a `User` object is created and saved within the `RSession` object. The different users can be specified as follows:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<users>
 <user>
  <name>admin</name>
  <password>admin</password>
  <role>1</role>
 </user>
 <user>
  <name>wsp</name>
  <password>wsp</password>
  <role>2</role>
 </user>
 <!--  The following user "anonymous" must always exist! -->
 <user>
  <name>anonymous</name>
  <password>an</password>
  <role>3</role>
 </user>
</users>
```

The element <role> is needed to assign each user its role. Actually only three different roles are implemented, which correspond to the three actors. Actually, role 1 stands for the administrator, role 2 for the WSP and role 3 for all other users. By default the anonymous user will be logged in automatically if the Web service is the first time accessed by a client. Therefore the users file must specify this anonymous user as shown in the listing.

4.5.4 The File Handler

The `FileHandler` class provides the `RepositoryManager` with some functions for handling files and streams. The method `saveFileFromURL(URL url, File dest, int bufSize, boolean force)` copies a file from a remote server to the local server. This method is used to copy the origin ontology documents as well as the concepts of the information service to the server. A query can be sent either as an URL pointing to the document or directly as a `String` to the `RepositoryManager`. In order to avoid redundancy the `String` will be converted into an `InputStream` by the `getStream(String doc)` method. A direct conversion of characters into bytes can cause some encoding errors. In previous versions of Java a method `StringBufferInputStream` allowed to create an `InputStream` out of a string, but there could occur some encoding problems. Since version JDK 1.1 of Java this method is declared as *deprecated*. Therefore the method implemented in the `FileHandler` reads first the document, writes the content into a new temporary file and opens the file again as an `InputStream`.

4.5.5 The Object Writer

The `ObjectWriter` provides functions to handle the different XML documents, which are exchanged by the `RepositoryManager` and the clients. The main functions are `createPersonalCollection(InputStream in)` and `writeResult(RDFResultSet result)`. The first one reads in the query concept presented as an XML document and creates a `PersonalCollection` object. The latter converts the result of a query into an XML document.

4.5.6 The Personal Collection

The class `PersonalCollection` is the presentation of the concepts files. It stores the different statements as well as the name and email address needed for the information service. Functions to retrieve this information are also provided.

4.5.7 The Properties File

The settings of ORWS are defined in the file “orws.properties”, which can be found in the `orws` package. The properties file is loaded by the class `Config`. However, to access the properties the class `ORWSProperties` contains all properties as fields. The properties starting with “SMTP” are needed for the information service. The table 4.2 specifies, which properties are defined and what they are for. Depending on how ORWS is executed (within Eclipse or directly on a server), the relative paths starting at different points. In both cases the root directory corresponds with the root directory of Eclipse or the server. For example, if Eclipse is used, the root directory is `c:\programme\eclipse`.

Property	Description
DATE_FORMAT	date format, which is used for logging; for example: “dd.MM.yyyy HH:mm”
INFORM_SUBSCRIBERS	true, if the information service is activated
LIST_SEPARATOR	used separator for the lists returned by ORWS; for example “;”
LOG_FILE	path and filename, where the logfile should be saved
LOGGING_LEVEL	level of the log4j loggers
MODELNAME	name of the used model, for example “jobc:onto”
PATH_INDIVIDUALS	path, where the original individual files should be saved
PATH_ONTOLOGIES	path, where the original ontologies should be saved
PATH_SEPARATOR	separator of directories
PATH_SUBSCRIBERS	path, where the concepts of the subscribers should be saved
PATH_USERS	path and filename, where the users are specified
PATTERNLAYOUT	specifies the layout of the logger; for example: “%d [%t] %x %5p %c -> %m%n”
SMTP_FROM	email address of the information service
SMTP_HOST	address of the smtp host
SMTP_PWD	password for the smtp host
SMTP_SUBJECT	subject text of the information service
SMTP_USER	username for the smtp host

Table 4.2: Properties

4.6 Sample Application for ORWS

In the following a sample application is introduced, which shows step by step what needs to be done for using ORWS. The domain of pizza home services (PHS) is described by an ontology, which abstract structure is shown in figure 4.4. The complete ontology can be found in appendix B.3. Each

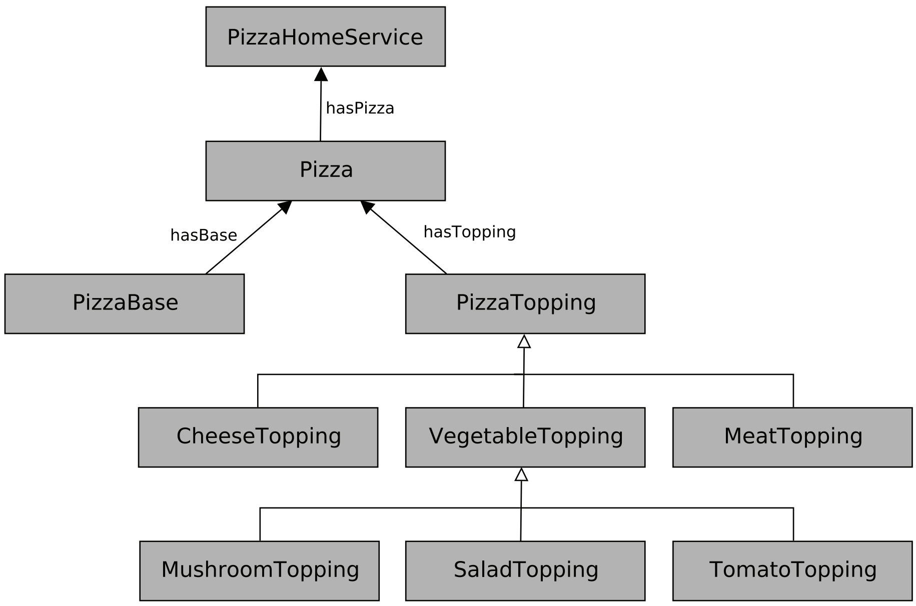

Figure 4.4: Pizza Home Service Ontology

PHS offers some pizzas. A pizza in this ontology exists of one base and has at least one topping on it. The toppings are divided into *CheeseTopping*, *VegetableTopping* and *MeatTopping*, whereas *VegetableTopping* has some subclasses as well. Furthermore, the ontology defines the properties *hasPizza*, *hasIngredient*, *hasTopping*, *hasBase* and the inverse properties of them. The properties *hasTopping*, *hasBase* are sub properties of *hasIngredient*. In addition some restrictions also apply, so is, for example, a “MeatTopping” disjoint from a “VegetableTopping”. Some instances of pizzas, toppings, and bases are already defined in the ontology (appendix B.3, starting at line 158).

Before the administrator can register any ontologies, he needs to login. After the login he registers the described ontology. The ontology is loaded and saved in the repository by ORWS. A PHS must define its service according to the ontology in ORWS. To get an overview, which classes, properties, and individuals are already defined, he can call some of the functions explained in table 4.1. Calling the function `getIndividuals()` with the parameter “http://www.ifi.lmu.de/ bodec/pizza_hs.owl#Pizza”, for example, returns all “pizzas”, which are available. Based on this list a PHS named “pizzeria_rosso” wants to register his service at the ORWS. Therefore he writes down the name of his service and the pizzas he offers in an OWL file:

```
1 <?xml version="1.0"?>
2 <rdf:RDF
3     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4     xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
5     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
6     xmlns:owl="http://www.w3.org/2002/07/owl#"
7     xmlns="http://www.ifi.lmu.de/~bodec/pizza_hs.owl#"
8   xml:base="http://www.ifi.lmu.de/~bodec/pizza_hs.owl">
9 <PizzaHomeService rdf:ID="pizzeria_rosso">
10  <hasPizza rdf:resource="#pizza_con_rucola_and_parma"/>
11  <hasPizza rdf:resource="#pizza_pepperoni"/>
12  <hasPizza rdf:resource="#pizza_margherita"/>
13 </PizzaHomeService>
14 </rdf:RDF>
```

The name of the PHS is specified in the attribute *rdf:ID* within the `PizzaHomeService` tag (line 9) and acts as an identifier. Such identifiers are used to add the offered pizzas. For example, the identifier *#pizza_margherita* points to the instance “pizza_margherita” of the class `Pizza` (see appendix B.3 line 164). A second PHS named “pizzaservice_bella_napoli” registers its service and pizzas as well. The “pizzeria_bella_napoli” offers just the pizzas “pizza_funghi” and “pizza_margherita” (see appendix B.6).

Now the user can search for PHSs, which are offering the desired pizza. As already mentioned in section 4.2.4, he needs to write a concept instead of entering a few keywords because of the different query processing of ORWS. For example, a concept returning all PHSs, which are offering pizzas with some vegetables on it, can be found in the appendix B.4. If the concept was passed to ORWS and processed by ORWS, ORWS returns both PHS, because on the “pizza_con_rucola_and_parma” is rucola salad and on a “pizza_funghi” are mushrooms. The inference engine reasoned about the facts stored in the ontology. For example, mushrooms are an instance of the class “MushroomTopping”, which is a subclass of “VegetableTopping”. Furthermore, “pizza_funghi” has a “MushroomTopping”, and “pizzeria_bella_napoli” has a “pizza_funghi” (see figure 4.5).

However, in case of a keyword based search engine the user might have entered “pizza home service vegetable". As a result the search engine returns only those PHS, which has stated explicitly in the directory, that its "pizza_funghi" has some "vegetables" on it. The advantage of ORWS can already be seen in this sample application. ORWS derived the information, what a "VegetableTopping" means, automatically out of the ontology. This information exists only implicitly.

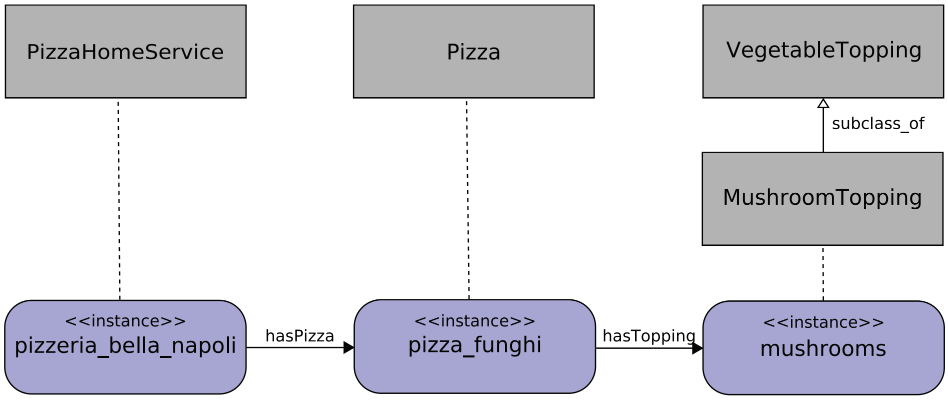

Figure 4.5: Instances and Classes

5 Technologies

In the following chapter, the technologies used are explained in more detail, starting with the Snobase package of IBM.

5.1 Snobase

Snobase is an ontology management system developed by IBM. They describe Snobase as follows [27]:

> “The system programmatically supports the ontology needs of applications in a similar way a database management system supports the data needs of applications.”

Snobase supports application programmers in dealing with ontologies on an accurate level of abstraction. They neither need to implement his own solutions for storing the different ontology languages in databases or file systems nor need to worry sbout how the information can be retrieved again. Snobase provides the application programmer with all functions concerning ontologies in a similar way as a database management system does for normal data.

5.1.1 Architecture and Components

Snobase provides a Java API, the so-called Java Ontology Base Connector (JOBC). JOBC can be compared to the Java Data Base Connector (JDBC). The query language of Snobase is similar to OWL-QL and has a similar role as SQL has for JDBC. In future versions of Snobase other query languages such as RDQL will be provided [27]. Figure 5.1 shows a schematic overview of Snobase: Applications access Snobase via the JOBC API, which allows a high-level access to the ontology resources and the ontology engine. The connection to the resources itself is provided by an ontology base driver, which is in this case Snobase. The driver consists of a number of components: a local

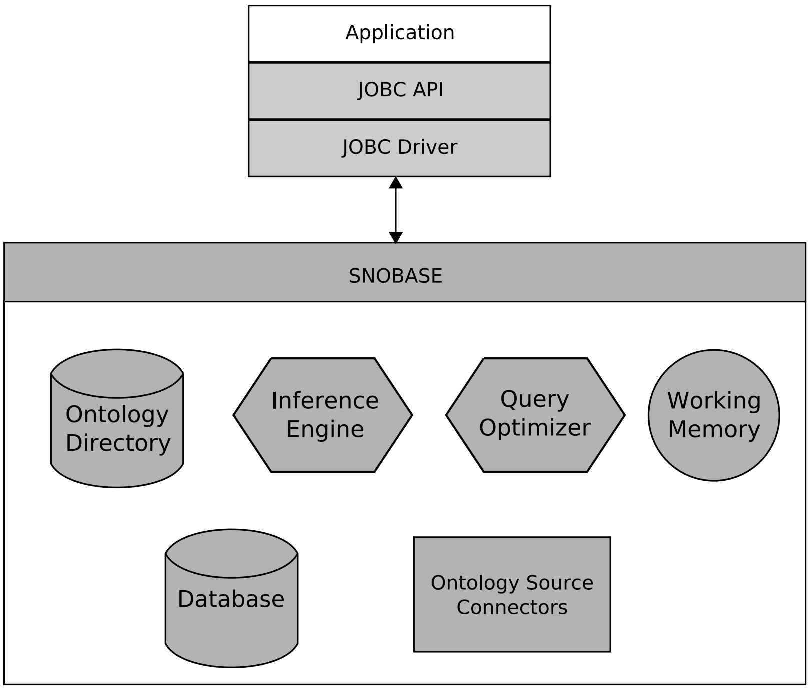

Figure 5.1: Schematic Overview of Snobase [27]

ontology directory, an inference engine, a working memory, a query optimizer and a set of connectors. In the following the various components are explained:

Snobase Driver This component is a driver, developed by IBM, for the JOBC interface. It is the equivalent of an DB2 driver for JDBC. The implementation of the JOBC interface is provided by the Snobase driver, that is implemented in Java. As can be seen in figure 5.1 these Java classes provide a number of components.

Ontology Directory The ontology directory provides the JOBC driver with the meta-level information about the stored ontologies. For example, the URI of every ontology source is saved in the ontology directory. References to the top-level definitions of OWL, RDF, RDF Schema, XML Schema need to be stored in the ontology directory [27] in order to support the different ontology languages. Additional meta information, which may be needed for query optimization, can also be stored in the ontology directory.

Inference Engine The inference engine tries to derive answers from a knowledge base. This knowledge base is represented by a set of language specific rules of the ontology languages. These rules are then used to answer the queries, when the answer is not directly available. With the help of rules the answer can be inferred from available facts. For example, if the application requests the *childrenOf* an individual, but the working memory only contains *parentOf* relations, the inference engine can use the inverse property statements about *childrenOf* and *parentOf* to identify the correct response [27]. Snobase supports JESS[1] and ABLE[2] as an inference engine. By default the ABLE engine is used.

Query Optimizer If applications need large databases or ontologies, it is not very practical to load the whole available information into the working memory. Hence, the driver only loads the appropriate information of the ontology source, that is really necessary to answer a query. Not only the retrieval of information is optimized, however the coordination of queries that span over several sources is also in the scope of the query optimizer.

Ontology Source Connectors The mechanism for reading, querying, and writing ontology information to persistent storage is provided by the ontology source connectors. The file connector stores the information, which is stated in the ontologies, to the local file system. Other connectors with functions like storing the information on remote servers are provided as well. Besides the storage mechanism, the connectors can be also used to implement caching mechanisms for remote information. This improves the overall performance and reliability [27].

5.1.2 Java Ontology Base Connector and Queries

As already mentioned, JOBC was designed as an equivalent to JDBC. Therefore the usage is very similar. The following code example shows how a connection to an ontology source can be opened via the JOBC driver [27]:

```
/* We connect to an ontology resource. */
Connection connection = DriverManager.getConnection();
RDFResource john = connection.createRDFResource("john");
```

[1] http://www.jessrules.com
[2] http://www.alphaworks.ibm.com/tech/able

```
4 RDFProperty isA = connection.createRDFProperty("isA");
RDFClass teacher = connection.createRDFClass("teacher");
/* We assert a statement in Working Memory: john isA teacher. */
7 Statement statement = connection.createStatement(john, isA, teacher);
connection.assert(statement);
/* We create a simple query. */
10 StatementCollection query = connection.createStatementCollection();
query.addStatement(statement);
ResultSet resultSet = connection.select(query);
```

Before any queries can be executed a connection to the repository must be established. Therefore the `DriverManager` returns a reference to the connection object, which is then used to create a statement (line 7), according to the (subject, predicate, object) structure. After asserting the statement into the working memory, a simple query is created and executed. Variables can be also used. At the time of query execution, the variable is bound to the corresponding results. An example shows the following code fragment:

```
Variable X = connection.createVariable("?X");
Statement queryStatement1 = connection.createStatement(X, isA, teacher);
```

For a more detailed and precise description how queries can be build see section 6.3.

5.1.3 Representation of Ontologies in a Database

Snobase uses a relational database system to store the ontological information. It stores all facts into a single table called *FactTable*. Besides of this table, some triggers are defined, which are fired, when a statement is inserted, updated or deleted from that table. The triggers can add facts, which are called *derived facts*. Facts originating from the ontology are called *asserted facts*. Facts, which are added through triggers, are stored together with a so-called *justification* key. The key points to that statement, which causes the asserted fact [27]. If an asserted fact is deleted from the fact table, all derived facts are also deleted regarding this justification key. The process of deletion itself is done by triggers as well. The database schema of the *FactTable* looks as follows:

```
CREATE TABLE FACT_TABLE (
        FACT_ID BIGINT DEFAULT AUTOINCREMENT INITIAL 1 INCREMENT 1 NOT NULL,
        SUBJECT BIGINT NOT NULL,
```

```
        VERB BIGINT NOT NULL,
        OBJECT BIGINT NOT NULL,
        MODEL VARCHAR(256) NOT NULL,
        JUSTIFICATION_1 BIGINT,
        JUSTIFICATION_2 BIGINT,
        JUSTIFICATION_3 BIGINT
)
```

The fields `SUBJECT`, `VERB` and `OBJECT` contain only foreign keys. These keys point to ancillary tables which yield the strings. Examples of how the triggers are defined, can be found in [27]. Queries, which are defined as triples in the context of ontologies, will be converted by the query processor into an SQL statement, which is then run against the fact table to retrieve the right answers.

5.2 Apache Axis

Apache Axis is a SOAP implementation in Java. It can be used to develop and run Web services. Axis itself is a Web application, which is deployed as a Web archive to a servlet container like Apache Tomcat[3] (for more information on Apache Tomcat see section 5.4). The Web archive contains three servlets one for the runtime, one for the administration and a monitor for the SOAP messages as can be seen in figure 5.2. Some additional tools, which support developers in building Web services, and functions are also provided by Axis. The most important tools are:

- Java2WSDL: Generates a WSDL document, which describes the Web service, from a Java interface definition;
- WSDL2Java: Generates some classes from a WSDL document. These classes can then be used to implement the Web service on the server side (skeletons) or they can be used to access the Web service from a client (stubs).

All incoming */services/** and **.jws* HTTP requests are forwarded to the Axis servlet. This servlet then processes the request. It parses the SOAP message, finds the designated class, compiles the class if necessary, calls the implementation and returns the result.
Axis provides two different ways of deploying Web services. Web services can directly be deployed by using a “Java Web Service” *.jws* file (see section 5.2.1), if the Axis servlet runs within a Tomcat

[3] `http://tomcat.apache.org/`

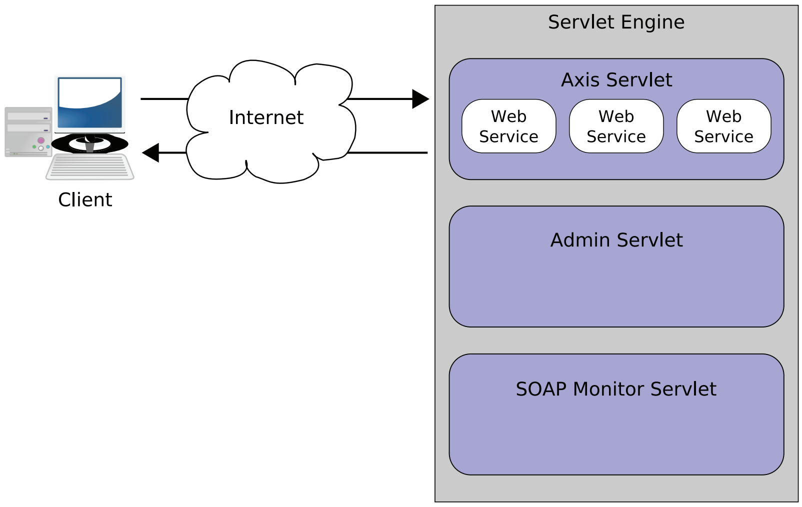

Figure 5.2: Apache Axis

container (otherwise a *WAR* archive must be created). The second way can be used if the developer wants a more fine-granular control. The Web service must then be described by a WSDL document by hand (see section 2.6) and for the deployment a Web Service Deployment Descriptor is needed.

5.2.1 Instant Deployment

Apache Axis provides an "instant deployment" mechanism, which allows one to create and deploy a Web service very simple and fast. All what must be done is to write a Java class with methods which can be accessed later-on as a Web service. This source file must be copied into the Axis path and renamed to a "Java Web Service" file (*.jws*). If this Web service is requested by a HTTP call, the file is compiled and deployed automatically. This is done only on the first call of the Web service. At any of the following requests the already compiled file is used. Furthermore a WSDL document is also generated, which can be accessed by calling the Web service with the parameter *?WSDL* (for example "StockQuotes.jws?WSDL"). The disadvantage is, that there is no possibility to configure or customize the Web service.

5.2.2 Normal Deployment

The instant deployment mechanism is sufficient for simple Web services or for fast testing some functions. Usually, however, a Web service is deployed in a more detailed way. There are six steps which need to be performed to create and deploy the Web service:

1. Firstly an interface needs to be programmed, that defines the public methods of the Web service. These functions define the input and output parameters of the Web service. Furthermore it is useful to implement the interface already at this point.

2. Now the tool Java2WSDL is needed. It creates from the compiled interface definition a WSDL document.

3. In the next step the tool WSDL2Java is called for creating the code needed to deploy the Web service together with some stubs. When WSDL2Java is run, many different files are generated. In the scope of this thesis, the following files were created (see [1]):

   - `ORSoapBindingImpl.java`: in this file the implementation of the Web service must be added or at least the different methods can be delegated to an already existing implementation. In case of delegating the method calls the `ORSoapBindingImpl.java` can be seen as a facade implementation.

   - `OR.java`: this is a remote interface to ORWS.

   - `ORService.java`: this is a simple service interface of the Web service. The ServiceLocator implements this interface.

   - `ORServiceLocator.java`: helper factory for retrieving a handle to the service.

   - `ORSoapBindingSkeleton.java`: server-side skeleton code. It just delegates the request.

   - `ORSoapBindingStub.java`: client-side stub code that encapsulates client access.

   - `deploy.wsdd`: the deployment descriptor contains important information, which is necessary for the right processing of a request. The information includes the name of the class implementing the service and the name of the methods, which can be called. The deployment descriptor must be passed to the Axis system to deploy the Web service. Within the deploy descriptor the scope of the Web service can be defined. For more details see section 4.5.1.

- `undeploy.wsdd`: deployment descriptor that will undeploy the Web services from the Axis system.

4. In the just created file `ORSoapBindingImpl.java` the implementation must be added. It is advisable not to hard-code the implementation directly in that file. Instead, the implementation should only be tied to that file. The object of the implementation should be instantiated by that file and then the Web service functions should be delegated to the appropriate functions of the implementation. In this thesis the functions are delegated to the `RepositoryManager.java` (see section 4.5.2).

5. After finishing all the implementation aspects, the source files must be compiled and packed into an archive. This archive must then be copied to the Axis' classpath.

6. Now, with the help of the Axis tool "AdminClient" the Web service can be deployed and is then accessible by the clients. A Web service can be deployed by the "AdminClient" on a shell with the command `java org.apache.axis.client.AdminClient deploy.wsdd` assumed that the Axis package can be found within the classpath of Java.

5.2.3 Development of a Client

A client for the Web service can be easily developed by using some of the already created files. These files are:

- `OR.java`: This is the service interface. It describes all functions, which are provided by the Web service. The client itself uses this interface to access the service.

- `ORService.java`: Another interface, which is implemented by the `ORServiceLocator.java`. This interface is only used internally.

- `ORServiceLocator.java`: The Locator is a factory class, which returns the service object to the client. With one of the two functions `public server.ORService getORInterface()` or `public server.OR getOR(java.net.URL portAddress)` the service object will be returned.

- `ORSoapBindingStub.java`: The program logic, which interacts with the SOAP messages, can be found in the Stub. Again, this class is only used internally and therefore does not appear in the client.

One can get access to the service by calling the *ServiceLocator*. The methods can be called by using the remote handle that is available from the Web service. The following example shows, how a client for a Web service returning a Fibonacci number can be programmed (see [1]):

```java
package fibonacci;

public class FibonacciTester {
  public static void main(String [] args) throws Exception {
    /* Make a service */
    fibonacci.ws.FibonacciService service =
         new fibonacci.ws.FibonacciServiceLocator();

    /* Now use the service to get a stub to the service */
    fibonacci.ws.Fibonacci fib = service.getFibonacci();

    /* Make the actual call */
    System.out.println("Fibonacci(10) = " +
                         fib.calculateFibonacci(10));
    }
}
```

After the client got the stub, the Web service can be accessed like local method calls. This is similar to the Remote Procedure Call (RPC) [6]. The communication between the user and the server is transparent. The user just communicates with the client stub. This client stub establishes the connection to the server stub, which communicates with the server. The client and the client stub run on the users side or a client machine, whereas the server stub and the server application run on a server. In case of our Web service, the communication between the different components is realized with SOAP messages, but the access of the Web service follows the RPC design architecture.

5.3 Log4j

The administrator can monitor, what is happening with the repository, what ontologies have been loaded or which Web services have been registered by means of the `RepositoryManager`. It uses the log4j library, which provides an extensive framework for logging. In our scenario a debugger would not be sufficient, because a debugger allows just to view a snapshot of the running system, whereas a logging mechanism allows to inspect what happened since the server was started.
Log4j has three main components: loggers, appenders and layouts. These three components work together and support developers to integrate log messages. How these messages are formatted and what kind of messages should be logged can be controlled at runtime.

5.3.1 Logger Hierarchy

Unlike to the normal logging with `System.out.println` a logging framework allows one to disable certain log statements, while others are still logged. This function works only, if the developer has categorized his logging statements. Since version 1.2 of log4j these categories are renamed to loggers [19]. The loggers underlie the following definition:

Named Hierarchy A logger is said to be an ancestor of another logger if its name followed by a dot is a prefix of the descendant logger name. A logger is said to be a parent of a child logger if there are no ancestors between itself and the descendant logger.

With this mechanism a hierarchy of loggers can be built. On the top of the hierarchy is the *root* logger. It always exists and one can get by the static method *Logger.getRootLogger*. All other loggers can be instantiated by the static method *Logger.getLogger(String name)*. Each logger can be assigned a logging level. Five core levels are already defined by log4j. These are *DEBUG*, *INFO*, *WARN*, *ERROR* and *FATAL*. Additional levels can be used by introducing a class, which extends the level class. If a logger was not assigned a level, it inherits the level of its closet ancestor [19].

Level Inheritance The inherited level for a given logger C is equal to the first non-null level in the logger hierarchy, starting at C and proceeding upwards in the hierarchy towards the root logger.

The root logger has always a level, otherwise a logger with no assigned level could exist and this is not allowed. The logging itself is done by the methods ‘debug’, ‘info’, ‘warn’, ‘error’, ‘fatal’ and ‘log’,

which can be invoked by a logger instance. The first five methods define already the logging level, whereas the method *log(Level l, Object message)* is generic. If the c is a logger instance and there is a logging request of level INFO, the statement c.info("..") fulfills this task.
A logging statement is logged, if the logging requests level is equal or higher than the level of its logger. Otherwise the logging is disabled.

Basic Selection Rule A log request of level p in a logger with (either assigned or inherited, whichever is appropriate) level q, is enabled if p >= q.

Log4j uses basically this rule. It follows the assumption that the different levels are ordered. The standard levels are ordered as follows:

$$DEBUG < INFO < WARN < ERROR < FATAL$$

The following example (see [19]) explains, how the different levels are used:

```
/* get a logger instance named "com.foo" */
Logger  logger = Logger.getLogger("com.foo");

/* Now set its level. Normally you do not need to set the
  level of a logger programmatically. This is usually done
  in configuration files. */
logger.setLevel(Level.INFO);

Logger barlogger = Logger.getLogger("com.foo.Bar");

/* This request is enabled, because WARN >= INFO. */
logger.warn("Low fuel level.");

/* This request is disabled, because DEBUG < INFO. */
logger.debug("Starting search for nearest gas station.");

/* The logger instance barlogger, named "com.foo.Bar",
   will inherit its level from the logger named
   "com.foo" Thus, the following request is enabled
   because INFO >= INFO. */
barlogger.info("Located nearest gas station.");

/* This request is disabled, because DEBUG < INFO. */
```

```
barlogger.debug("Exiting gas station search");
```

Different instances of loggers can be accessed everywhere in the code without passing the references around. Even if a child logger is defined before the parent logger, the parent logger finds his children. Each logger can be identified by the name defined at the instantiation. The names of loggers can be freely defined by the developers, but the common way is naming the logger with the fully qualified name of the class [19]. The logger itself knows his name. This makes it easy to identify the origin of a log message.

5.3.2 Appenders and Layouts

The possibility to disable or enable the log messages is just one of the functions of log4j. Another useful function is printing the log messages to several destinations. These can even be done in parallel by adding more appenders to one logger. The several output destinations are called *appenders*. For example, the following appenders are available: file, console, GUI components or remote socket servers.

The output format itself can also be specified by the developer. It is possible to have the messages formatted very simple, following a pattern layout, output as a HTML table or as a XML document.

5.4 Apache Tomcat

Apache Tomcat is the reference implementation for the Java servlets and Java Server Pages technologies. The actual version 5.5 of Tomcat implements the Servlet 2.4 and Java Server Pages 2.0 specifications. A servlet is a Java class which implements the `javax.servlet.Servlet` interface. This class is compiled and instantiated by a servlet engine like Apache Tomcat. Within Apache Tomcat, which also provides a simple Web server, a servlet can process HTTP requests dynamically. For example, a servlet can establish a database connection, query some data from the database, process the data and send the data back to the user as a HTML page. In the scope of this thesis the Apache Axis framework is such an servlet. It is addressed via the Web server and then calls ORWS. All requests addressed to the Web service are delegated from the Apache Tomcat to the Axis servlet and finally to the Web service itself. Outgoing messages follow the same way.

Java Server Pages are a technology for developing dynamic HTML pages. Within such HTML pages

Java code can be directly used inside special tags. Furthermore, some Java Server Pages directives can also be used. For example, the test client of ORWS uses some Java Server Pages to send the queries and to perform the answers. Within Java Server Pages the dynamic logic is inserted into the static HTML source code, whereas in Java servlets the content must be created by calling special functions. Thus, content and design is better separated in Java Server Pages than in servlets.

6 Using Snobase

As already mentioned in section 5.1 Snobase provides developers with an API for ontologies. Snobase was developed by IBM in the context of IBM Alphaworks[1]. Therefore Snobase is not very well documented. All documentation that is provided with Snobase are the Javadoc documents of the provided classes as well as a paper introducing the Snobase system [27]. A more precise documentation about how Snobase can be used is missing. In the following some hints are given about the use of Snobase and some problems and their solutions are addressed.

6.1 General Remarks

For running a stand-alone application with Snobase, all what must be done is to add the class files to the classpath of the application. For easier handling of the different class files it is advisable to pack the class files into a Jar file. A properties file comes along with Snobase. It should enable the user to configure Snobase, but it is not documented, where the file need to be placed. Therefore the file is not read and Snobase uses his default configuration. Copying the file into the directory of Snobase as well as adding the file to the classpath did not work.

6.2 Running Snobase within Tomcat

If Snobase is to be used within a Web service application or a servlet, some files need to be copied to another place than running Snobase as a stand-alone application. Snobase needs for the inference engine some `.arl` files, which define the reasoning rules. Depending on the used model the corresponding rule file needs to be loaded. These `.arl` files can be found in the `com/ibm/jobc/ob2/inferencing/ABLE` directory. If the Snobase package is just added to the classpath the following `FileNotFoundException` is thrown:

[1] `http://www.alphaworks.ibm.com/`

```
com.ibm.jobc.owl.ob2.OWLOntologyImpl::RDFModelImpl()::
        Exception Occurred: com/ibm/jobc/ob2/inferencing/ABLE/OWL.arl
java.io.FileNotFoundException:
        com/ibm/jobc/ob2/inferencing/ABLE/OWL.arl
at com.ibm.jobc.ob2.inferencing.ABLE.
        JOBCAbleInferencer.initialize(JOBCAbleInferencer.java:226)
at com.ibm.jobc.ob2.inferencing.ABLE.
        JOBCAbleInferencer.<init>(JOBCAbleInferencer.java:55)
at com.ibm.jobc.ob2.inferencing.ABLE.
        OWLAbleInferencer.<init>(OWLAbleInferencer.java:18)
```

As already mentioned this exception occurs if Snobase is used within the Tomcat environment. After trying to reproduce this exception with a simpler application, the solution is as follows:
The file is most likely loaded by a `ClassLoader`[2]. In the context of Tomcat, the `ClassLoader` must be different from the one of an application. Therefore the path, where the files are searched, must be a different one. As the error message shows the `JOBCAbleInferencer` of Snobase tries to load the file `com/ibm/jobc/ob2/inferencing/ABLE/OWL.arl`. Note the complete path. This suggests, that the `JOBCAbleInferencer` is looking for a file `OWL.arl` placed in a directory `com/ibm/jobc/ob2/inferencing/ABLE/`. If a different `ClassLoader` is used, the file must be copied to the directory `com/ibm/jobc/ob2/inferencing/ABLE/`, but the root directory is the one, where the `JOBCAbleInferencer.class` is in. Accordingly, the `.arl` files must be copied to `com/ibm/jobc/ob2/inferencing/ABLE/com/ibm/jobc/ob2/inferencing/ABLE/`.
The problem with the properties file could not be solved in a similar way, because it could not be figured out, where Snobase tries to load the file from.

6.3 Defining Queries

The examples provided in [27] give only a short introduction how queries can be defined. Especially the exact syntax of more complex queries is missing. Accordingly to [27] the query language of Snobase is similar to OWL-QL (see [18]). A complex query consists of several simple statements following the subject, predicate, object pattern of OWL-QL. These simple statements must be added

[2] `http://java.sun.com/j2se/1.5.0/docs/api/java/lang/ClassLoader.html`

to a `StatementCollection` in order to run the query with Snobase. In the following, two sample queries will be given. Both are for querying an ontology describing pizzas. The ontology itself can be found in the appendix B.3. The first query returns all pizzas that have cheese on it.

```
/* Define namespace */
String PIZZA_NS = "http://www.ifi.lmu.de/~bodec/pizza_hs.owl#";
/* Create collection */
RDFStatementCollection query = model.createRDFStatementCollection();

/* Create first statement */
RDFVariable X = model.createRDFVariable("?X");
RDFProperty isA = model.createRDFProperty(RDFModel.RDF_TYPE);
RDFSResource pizza = model.createRDFSResource(PIZZA_NS + "Pizza");
RDFStatement statement1 = model.createRDFStatement(X, isA, pizza);

/* Add statement to collection */
query.addStatement(statement1);

/* Create second statement */
RDFVariable Y = model.createRDFVariable("?Y");
RDFSResource cheeseTopping =
   model.createRDFSResource(PIZZA_NS + "CheeseTopping");
RDFStatement statement2 =
   model.createRDFStatement(Y, isA, cheeseTopping);
/* Add statement to collection */
query.addStatement(statement2);

/* Create third statement */
RDFProperty hasTopping =
        model.createRDFProperty(PIZZA_NS + "hasTopping");
RDFStatement statement3 = model.createRDFStatement(X, hasTopping, Y);
query.addStatement(statement3);

/* Execute query */
RDFResultSet resultSet = model.select(query);
```

In order to formulate this rather simple query, three statements are needed. The first statement binds all elements which are of type `pizza` to the variable `?X`. The second statement binds all elements which are of the type CheeseTopping to the variable ?Y. Finally, the two variables which are bound to the different types Pizza respectively CheeseTopping must be related to each other. The property which describes this relation is hasTopping. The last statement links all cheese toppings by the property hasTopping to the corresponding pizzas.

The next example query returns all toppings, which are disjoint from the MushroomTopping:

```
/* Create statement */
RDFVariable R = model.createRDFVariable("?R");
OWLProperty OWLDisjointWith =
        model.createOWLProperty(OWLOntology.OWL_DISJOINTWITH);
OWLClass cMushroom =
        model.createOWLClass(PIZZA_NS+"MushroomTopping");
RDFStatement queryStatement =
        model.createRDFStatement(R, OWLDisjointWith,cMushroom );
query.addStatement(queryStatement);
```

In each example a constant for the property is used. There are many more constants available, which are stated in the constant-values.html file within the Javadoc directory of the Snobase package.

7 Conclusion and Future Work

7.1 Conclusion

The goal of this thesis was to develop an ontology-based repository for Web services. Such a repository was implemented based on Snobase. One of the challenges during this thesis was to understand, how Snobase works and how it can be used for desired functions. Figuring out, how queries need to be built and running Snobase within the Tomcat environment took the most time. In the end it can be concluded, that the core functions are implemented. The repository can store ontologies as well as Web services and the inference engine can reason about them. To keep the interface of ORWS simple, not all functions, which are provided by Snobase were used. For example, Snobase provides functions to create ontologies or instances of classes within the application. Some other functions which would be useful, for example, deleting an entire ontology or a Web service, are not supported by Snobase. Only the complete model can be deleted.

7.2 Future Work

Meanwhile Snobase has been merged into the IBM Integrated Ontology Development Toolkit (IODT). The corresponding component is now Minerva. Minerva provides the developers with much more functions to handle ontologies and a repository. Functions for deleting ontologies are available now. In the future ORWS should be based upon Minerva instead of Snobase. This should solve the issues mentioned in chapter 6.

Furthermore in case of a repository that has many different ontologies and Web services stored, a filter mechanism should be implemented. This filter might allow users to define which contents of the repository should be visible. The already implemented session management allows to separate the different users, that is necessary for such a filter mechanism.

Apart of the server side implementations a more useful client could be developed. At least a client should support the users in creating the query concepts as well as the Web service providers in browsing the repository and creating of Web service descriptions. Right now just a simple test client based on HTML was implemented.

A Running ORWS

This chapter explains, how the ORWS project can be opened within Eclipse and which plugins are needed for running ORWS within Eclipse. How ORWS can be deployed to an application server directly is explained in section A.2. ORWS was developed and tested under Java SDK 1.5, Apache Tomcat 5.5.16, Axis 1.4, JavaMail 1.4, XML Security 1.3 and JAF 1.1. Java and Apache Tomcat must be installed before the next steps can be executed. All files needed for running ORWS can be found on the provided CD.

A.1 Running with Eclipse

Eclipse is an open-source integrated development environment (IDE) mainly designed for developing Java applications. The used version for developing ORWS was Eclipse 3.2. The functions of Eclipse can be extended by different plugins. For the development of ORWS the plugin “Web Tools Platform” (WTP) is needed. WTP extends Eclipse with tools for developing J2EE Web applications. Tools for editing HTML, XSD or XML files, graphical editors for WSDL documents, a Web service wizard and some others are included. The plugin also provides functions to start and stop the Apache Tomcat server as well as deploying of ORWS to that server.

After the installation of Eclipse the WTP plugin must be installed. The recommended way is using the “Find and Install” function of Eclipse, which can be found under “Help”, “Software Updates”. On the appearing dialogue the function “Search for new features to install” must be chosen, the “Callisto Discovery Site” marked and the button “Finish” pressed. Eclipse searches then for supported plugins and shows them in a list. Under the topic “Callisto Discovery Site”, “Web and J2EE Development” the plugin “Web Standard Tools” must be marked. The WTP requires some other plugins, which can be simply added by pressing the button “Select Required”. These required plugins are, for example, “Eclipse Modeling Framework” and “Visual Editor”. On the project page of WTP[1] a complete pack-

[1] `http://www.eclipse.org/webtools/main.html`

age containing Eclipse 3.2 SDK, Eclipse Modeling Framework (EMF), Graphical Editing Framework (GEF) and JEM[2] can be found, which allows an easy installation from the scratch.
Once, Eclipse and the required plugins are installed, the server, on which the Web service should be deployed, must be specified. Open the "Preferences" dialog and go to the "Server" settings. Under the point "Installed Runtimes", the Apache Tomcat runtime must be added.
Following this configuration the ORWS project can be imported into the workspace by calling the "Import" function of Eclipse. The import source is "Existing Projects into Workspace" and the checkbox "Copy projects to workspace" should be activated. Once the files are copied, the function "Run on server" deploys the project to the specified server and starts the server. The test client can be imported analogous.

A.2 Deploying ORWS on Apache Tomcat

For the direct deployment of ORWS to Apache Tomcat there are several steps necessary. It is assumed, that the Apache Tomcat is already installed and configured. First Apache Axis[3] (see section 5.2) must be installed. Therefore extract the Axis source files to a temporary directory and copy the complete `axis` directory in the folder `webapps` to `<tomcat-root>\webapps`. Additional, three more files are needed. From the *JavaBeans Activation Framework*[4] the file `activation.jar`, from *JavaMail*[5] the file `mail.jar` and from *XML Security*[6] the file `xmlsec-1.3.0.jar`. These three files must be copied into the directory `<tomcat-root>\webapps\axis\WEB-INF\lib`. To test the installation of Axis, start Apache Tomcat and open the page `http://localhost:8080/axis/happyaxis.jsp` in a browser. If all components are found in the section "Needed Components", the installation was successful.
After the installation of Apache Axis, copy the following libraries needed by ORWS to the directory `<tomcat-root>\webapps\axis\WEB-INF\lib`: `able.jar`, `antlr.jar`, `common.jar`, `ecore.jar`, `jdom.jar`, `JLog.jar`, `jodm.jar`, `SIRPAC.W3C.latest.jar` and `snobase.jar`. The class files of ORWS itself must be also copied to that directory. It is

[2] part of the Visual Editor
[3] `http://ws.apache.org/axis/`
[4] `http://java.sun.com/products/javabeans/glasgow/jaf.html`
[5] `http://java.sun.com/products/javamail/`
[6] `http://xml.apache.org/security/`

recommended to generate from the compiled source-files a *jar* package. If all files are copied, restart Apache Tomcat.

Before ORWS can be deployed the deployment descriptors deploy.wsdd and undeploy.wsdd (see appendix B.1 and B.2) must be copied to the directory <tomcat-root>\webapps\axis\WEB-INF. Then execute the following instructions:

- Open a console
- Change to the directory <tomcat-root>\webapps\axis\WEB-INF
- Enter SET CLASSPATH=".;lib/activation.jar; lib/axis.jar; lib/axis-ant.jar; lib/commons-discovery.jar; lib/commons-logging.jar; lib/jaxrpc.jar;lib/log4j.jar; lib/mail.jar; lib/saaj.jar; lib/wsdl4j.jar"
- Enter java org.apache.axis.client.AdminClient deploy.wsdd

If the deployment was successful, ORWS can be tested with the following requests:

- http://localhost:8080/axis/services/OR
- http://localhost:8080/axis/services/OR?WSDL

Some methods of ORWS can be called directly within the browser. For example, the login method can be called with the request http://localhost:8080/axis/services/OR?method=login\&in0=admin\&in1=admin. If the login was successful, the browser displays the SOAP envelope.

B Source Code

B.1 Deployment Descriptor

```
<!-- Use this file to deploy some handlers/chains and services      -->
<!-- Two ways to do this:                                           -->
<!--   java org.apache.axis.client.AdminClient deploy.wsdd          -->
<!--      after the axis server is running                          -->
<!-- or                                                             -->
<!--   java org.apache.axis.utils.Admin clientserver deploy.wsdd   -->
<!--      from the same directory that the Axis engine runs         -->

<deployment
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <!-- Services from ORService WSDL service -->

  <service name="OR" provider="java:RPC" style="wrapped" use="literal">
      <parameter name="wsdlTargetNamespace" value="http://server"/>
      <parameter name="wsdlServiceElement" value="ORService"/>
      <parameter name="schemaQualified" value="http://server"/>
      <parameter name="wsdlServicePort" value="OR"/>
      <parameter name="className" value="server.ORSoapBindingSkeleton"/>
      <parameter name="wsdlPortType" value="OR"/>
      <parameter name="typeMappingVersion" value="1.2"/>
      <parameter name="allowedMethods" value="*"/>
      <parameter name="scope" value="session" />

  </service>
</deployment>
```

B.2 Undeployment Descriptor

```
<!-- Use this file to undeploy some handlers/chains and services    -->
<!-- Two ways to do this:                                           -->
<!--   java org.apache.axis.client.AdminClient undeploy.wsdd        -->
<!--      after the axis server is running                          -->
<!-- or                                                             -->
<!--   java org.apache.axis.utils.Admin clientserver undeploy.wsdd -->
<!--      from the same directory that the Axis engine runs         -->

<undeployment
    xmlns="http://xml.apache.org/axis/wsdd/">

  <!-- Services from ORService WSDL service -->

  <service name="OR"/>
</undeployment>
```

B.3 Pizza Home Service Ontology

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns="http://www.ifi.lmu.de/~bodec/pizza_hs.owl#"
  xml:base="http://www.ifi.lmu.de/~bodec/pizza_hs.owl">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="MushroomTopping">
    <owl:disjointWith>
      <owl:Class rdf:ID="SaladTopping"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:ID="VegetableTopping"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:ID="TomatoTopping"/>
```

```
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:ID="Pizza">
    <owl:disjointWith>
      <owl:Class rdf:ID="PizzaBase"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:someValuesFrom>
          <owl:Class rdf:about="#PizzaBase"/>
        </owl:someValuesFrom>
        <owl:onProperty>
          <owl:FunctionalProperty rdf:ID="hasBase"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    <owl:disjointWith>
      <owl:Class rdf:ID="PizzaTopping"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:ID="CheeseTopping">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#PizzaTopping"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:ID="MeatTopping"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#VegetableTopping"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:ID="PizzaHomeService"/>
  <owl:Class rdf:about="#VegetableTopping">
    <owl:disjointWith>
      <owl:Class rdf:about="#MeatTopping"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#CheeseTopping"/>
```

```
    <rdfs:subClassOf>
      <owl:Class rdf:about="#PizzaTopping"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#TomatoTopping">
    <owl:disjointWith>
      <owl:Class rdf:about="#SaladTopping"/>
    </owl:disjointWith>
    <rdfs:subClassOf rdf:resource="#VegetableTopping"/>
    <owl:disjointWith rdf:resource="#MushroomTopping"/>
  </owl:Class>
  <owl:Class rdf:about="#SaladTopping">
    <owl:disjointWith rdf:resource="#MushroomTopping"/>
    <owl:disjointWith rdf:resource="#TomatoTopping"/>
    <rdfs:subClassOf rdf:resource="#VegetableTopping"/>
  </owl:Class>
  <owl:Class rdf:ID="HamTopping">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#MeatTopping"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:ID="PepperoniTopping"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:about="#PizzaTopping">
    <owl:disjointWith>
      <owl:Class rdf:about="#PizzaBase"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Pizza"/>
  </owl:Class>
  <owl:Class rdf:about="#PepperoniTopping">
    <owl:disjointWith rdf:resource="#HamTopping"/>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#MeatTopping"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#MeatTopping">
    <owl:disjointWith rdf:resource="#VegetableTopping"/>
```

```
    <rdfs:subClassOf rdf:resource="#PizzaTopping"/>
    <owl:disjointWith rdf:resource="#CheeseTopping"/>
  </owl:Class>
  <owl:Class rdf:about="#PizzaBase">
    <owl:disjointWith rdf:resource="#Pizza"/>
    <owl:disjointWith rdf:resource="#PizzaTopping"/>
  </owl:Class>
  <owl:ObjectProperty rdf:ID="hasPizza">
    <rdfs:domain rdf:resource="#PizzaHomeService"/>
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="isPizzaOf"/>
    </owl:inverseOf>
    <rdfs:range rdf:resource="#Pizza"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:ID="hasTopping">
    <rdfs:domain rdf:resource="#Pizza"/>
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="isToppingOf"/>
    </owl:inverseOf>
    <rdfs:range rdf:resource="#PizzaTopping"/>
    <rdfs:subPropertyOf>
      <owl:ObjectProperty rdf:ID="hasIngredient"/>
    </rdfs:subPropertyOf>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:about="#isToppingOf">
    <owl:inverseOf rdf:resource="#hasTopping"/>
    <rdfs:domain rdf:resource="#PizzaTopping"/>
    <rdfs:subPropertyOf>
      <owl:TransitiveProperty rdf:ID="isIngredientOf"/>
    </rdfs:subPropertyOf>
    <rdfs:range rdf:resource="#Pizza"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:about="#isPizzaOf">
    <owl:inverseOf rdf:resource="#hasPizza"/>
    <rdfs:domain rdf:resource="#Pizza"/>
    <rdfs:range rdf:resource="#PizzaHomeService"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:about="#hasIngredient">
```

```
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#TransitiveProperty"/>
  <owl:inverseOf>
    <owl:TransitiveProperty rdf:about="#isIngredientOf"/>
  </owl:inverseOf>
</owl:ObjectProperty>
<owl:TransitiveProperty rdf:about="#isIngredientOf">
  <owl:inverseOf rdf:resource="#hasIngredient"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
</owl:TransitiveProperty>
<owl:FunctionalProperty rdf:about="#hasBase">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
  <rdfs:domain rdf:resource="#Pizza"/>
  <rdfs:range rdf:resource="#PizzaBase"/>
  <owl:inverseOf>
    <owl:InverseFunctionalProperty rdf:ID="isBaseOf"/>
  </owl:inverseOf>
  <rdfs:subPropertyOf rdf:resource="#hasIngredient"/>
</owl:FunctionalProperty>
<owl:InverseFunctionalProperty rdf:about="#isBaseOf">
  <rdfs:range rdf:resource="#Pizza"/>
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
  <owl:inverseOf rdf:resource="#hasBase"/>
  <rdfs:domain rdf:resource="#PizzaBase"/>
  <rdfs:subPropertyOf rdf:resource="#isIngredientOf"/>
</owl:InverseFunctionalProperty><PepperoniTopping rdf:ID="pepperoni">
  <isToppingOf>
    <Pizza rdf:ID="pizza_pepperoni">
      <hasTopping>
        <CheeseTopping rdf:ID="bufalo_mozzarella">
          <isToppingOf rdf:resource="#pizza_pepperoni"/>
          <isToppingOf>
            <Pizza rdf:ID="pizza_margherita">
              <hasTopping>
                <TomatoTopping rdf:ID="tomato_purree">
                  <isToppingOf>
                    <Pizza rdf:ID="pizza_funghi">
                      <hasBase>
                        <PizzaBase rdf:ID="italian_style">
```

```
        <isBaseOf>
          <Pizza rdf:ID="pizza_con_rucola_and_parma">
            <hasTopping>
              <CheeseTopping rdf:ID="emmenthaler">
                <isToppingOf rdf:resource="#
                    pizza_con_rucola_and_parma"/>
              </CheeseTopping>
            </hasTopping>
            <hasTopping>
              <HamTopping rdf:ID="prosciutto_di_parma">
                <isToppingOf rdf:resource="#
                    pizza_con_rucola_and_parma"/>
              </HamTopping>
            </hasTopping>
            <hasBase rdf:resource="#italian_style"/>
            <hasTopping rdf:resource="#tomato_purree"/>
            <hasTopping>
              <SaladTopping rdf:ID="rucola">
                <isToppingOf rdf:resource="#
                    pizza_con_rucola_and_parma"/>
              </SaladTopping>
            </hasTopping>
          </Pizza>
        </isBaseOf>
        <isBaseOf rdf:resource="#pizza_pepperoni"/>
        <isBaseOf rdf:resource="#pizza_margherita"/>
        <isBaseOf rdf:resource="#pizza_funghi"/>
      </PizzaBase>
    </hasBase>
    <hasTopping rdf:resource="#tomato_purree"/>
    <hasTopping>
      <MushroomTopping rdf:ID="mushrooms">
        <isToppingOf rdf:resource="#pizza_funghi"/>
      </MushroomTopping>
    </hasTopping>
  </Pizza>
</isToppingOf>
<isToppingOf rdf:resource="#pizza_margherita"/>
```

```
                    <isToppingOf rdf:resource="#pizza_con_rucola_and_parma"/>
                    <isToppingOf rdf:resource="#pizza_pepperoni"/>
                  </TomatoTopping>
                </hasTopping>
                <hasBase rdf:resource="#italian_style"/>
                <hasTopping rdf:resource="#bufalo_mozzarella"/>
                <hasTopping>
                  <TomatoTopping rdf:ID="cherry_tomato">
                    <isToppingOf rdf:resource="#pizza_margherita"/>
                  </TomatoTopping>
                </hasTopping>
              </Pizza>
            </isToppingOf>
          </CheeseTopping>
        </hasTopping>
        <hasBase rdf:resource="#italian_style"/>
        <hasTopping rdf:resource="#pepperoni"/>
        <hasTopping rdf:resource="#tomato_purree"/>
      </Pizza>
    </isToppingOf>
  </PepperoniTopping>
  <VegetableTopping rdf:ID="garlic"/>
  <TomatoTopping rdf:ID="sweet_tomato"/>
  <PizzaBase rdf:ID="american_style"/>
  <HamTopping rdf:ID="smoked_ham"/>
</rdf:RDF>
```

B.4 Query Concept

The following concept returns all pizza home services, which are offering pizzas with vegetables on it.

```
<?xml version="1.0" encoding="UTF-8"?>
<statementcollection>
  <statement>
    <subject>RDFVarible:?X</subject>
    <predicate>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</predicate>
    <object>http://www.ifi.lmu.de/˜bodec/pizza_hs.owl#Pizza</object>
  </statement>
```

```
  <statement>
    <subject>RDFVarible:?Y</subject>
    <predicate>http://www.w3.org/1999/02/22-rdf-syntax-ns#type</predicate>
    <object>http://www.ifi.lmu.de/˜bodec/pizza_hs.owl#VegetableTopping</object>
  </statement>
   <statement>
    <subject>RDFVarible:?Z</subject>
    <predicate>http://www.ifi.lmu.de/˜bodec/pizza_hs.owl#type</predicate>
    <object>http://www.ifi.lmu.de/˜bodec/pizza_hs.owl#PizzaHomeService</object>
  </statement>
 <statement>
    <subject>RDFVarible:?X</subject>
    <predicate>http://www.ifi.lmu.de/˜bodec/pizza_hs.owl#hasTopping</predicate>
    <object>RDFVarible:?Y</object>
  </statement>
 <statement>
    <subject>RDFVarible:?Z</subject>
    <predicate>http://www.ifi.lmu.de/˜bodec/pizza_hs.owl#hasPizza</predicate>
    <object>RDFVarible:?X</object>
  </statement>
</statementcollection>
```

B.5 Pizzeria Rosso

```
<?xml version="1.0"?>
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:owl="http://www.w3.org/2002/07/owl#"
        xmlns="http://www.ifi.lmu.de/˜bodec/pizza_hs.owl#"
        xml:base="http://www.ifi.lmu.de/˜bodec/pizza_hs.owl">
        <PizzaHomeService rdf:ID="pizzeria_rosso">
                <hasPizza rdf:resource="#pizza_con_rucola_and_parma"/>
                <hasPizza rdf:resource="#pizza_pepperoni"/>
                <hasPizza rdf:resource="#pizza_margherita"/>
        </PizzaHomeService>
```

```
</rdf:RDF>
```

B.6 Pizzeria Bella Napoli

```
<?xml version="1.0"?>
<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:owl="http://www.w3.org/2002/07/owl#"
        xmlns="http://www.ifi.lmu.de/˜bodec/pizza_hs.owl#"
        xml:base="http://www.ifi.lmu.de/˜bodec/pizza_hs.owl">
        <PizzaHomeService rdf:ID="pizzeria_bella_napoli">
                <hasPizza rdf:resource="#pizza_margherita"/>
                <hasPizza rdf:resource="#pizza_funghi"/>
        </PizzaHomeService>
</rdf:RDF>
```

Bibliography

[1] Dion Almaer: Creating Web Services with Apache Axis, 2002. `http://www.onjava.com/pub/a/onjava/2002/06/05/axis.html`.

[2] Anupriya Ankolekar: DAML-S: Semantic Markup for Web Services. `http://www.citeseer.ist.psu.edu/448875.html`.

[3] Vangie Beal: Understanding Web Services. `http://www.webopedia.com/DidYouKnow/Computer_Science/2005/web_services.asp`.

[4] Philip A. Bernstein: Repositories and Object Oriented Databases. SIGMOD Record, 27(1):88–96, 1998.

[5] Philip A. Bernstein and Umeshwar Dayal: An overview of repository technology. In VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases, pages 705–713, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc., ISBN 1-55860-153-8.

[6] Andrew D. Birrell and Bruce J. Nelson: Implementing remote procedure calls. ACM Trans. Comput. Syst., 2(1):39–59, 1984, ISSN 0734-2071.

[7] David Booth, Hugo Haas, and Francis McCabe: Web Services Architecture. `http://www.w3.org/TR/ws-arch/`.

[8] David Booth and Canyang K. Liu: Web Services Description Language (WSDL) Version 2.0 Part 0: Primer. `http://www.w3.org/TR/wsdl20-primer`.

[9] Tim Bray *et al.*: Extensible Markup Language (XML) 1.1. `http://www.w3.org/TR/xml11/`.

[10] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen: Sesame: A generic architecture for storing and querying rdf and rdf schema, 2002. `http://www.citeseer.ist.psu.edu/broekstra02sesame.html`.

[11] Erik Christensen *et al.*: WEB SERVICES DESCRIPTION LANGUAGE (WSDL) 1.1. `http://www.w3.org/TR/wsdl`.

[12] The DAML Services Coalition: RDF VOCABULARY DESCRIPTION LANGUAGE 1.0: RDF SCHEMA. `http://www.w3.org/TR/rdf-schema/`.

[13] Dan Connolly *et al.*: DAML+OIL (MARCH 2001) REFERENCE DESCRIPTION. `http://www.w3.org/TR/daml+oil-reference`.

[14] Stefan Decker *et al.*: THE SEMANTIC WEB: THE ROLES OF XML AND RDF. IEEE Internet Computing, 15(3):63–74, October 2000.

[15] Asuman Dogac *et al.*: IMPROVING THE FUNCTIONALITY OF UDDI REGISTRIES THROUGH WEB SERVICE SEMANTICS. In TES '02: PROCEEDINGS OF THE THIRD INTERNATIONAL WORKSHOP ON TECHNOLOGIES FOR E-SERVICES, pages 9–18, London, UK, 2002. Springer-Verlag, ISBN 3-540-44110-7.

[16] Wolfgang Dostal and Mario Jeckle: SEMANTIK, ODEM EINER SERVICE-ORIENTIERTEN ARCHITEKTUR. Java Spektrum, 1:53–56, 03 2005.

[17] David C. Fallside and Priscilla Walmsley: XML SCHEMA PART 0: PRIMER SECOND EDITION. `http://www.w3.org/TR/xmlschema-0/`.

[18] Richard Fikes, Pat Hayes, and Ian Horrocks: OWL-QL: A LANGUAGE FOR DEDUCTIVE QUERY ANSWERING ON THE SEMANTIC WEB. Technical Report KSL 03-14, Stanford University, Stanford, CA, 2003.

[19] Ceki Gülcü: SHORT INTRODUCTION TO LOG4J, 2002. `http://logging.apache.org/log4j/docs/manual.html`.

[20] Jeff Heflin: OWL WEB ONTOLOGY LANGUAGE USE CASES AND REQUIREMENTS. `http://www.w3.org/TR/webont-req/`.

[21] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen: FROM SHIQ AND RDF TO OWL: THE MAKING OF A WEB ONTOLOGY LANGUAGE. Journal of Web Semantics, 1(1):7–26, 2003. `http://www.citeseer.ist.psu.edu/horrocks03from.html`.

[22] Greg Karvounarakis: RQL V2.1 DOCUMENTATION, 2003. `http://139.91.183.30:9090/RDF/RQL/Design.html`.

[23] Gregory Karvounarakis *et al.*: RQL: A DECLARATIVE QUERY LANGUAGE FOR RDF. In WWW '02: PROCEEDINGS OF THE 11TH INTERNATIONAL CONFERENCE ON WORLD WIDE WEB, pages 592–603, New York, NY, USA, 2002. ACM Press, ISBN 1-58113-449-5.

[24] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov: OWLIM - A PRAGMATIC SEMANTIC REPOSITORY FOR OWL. In WISE WORKSHOPS, pages 182–192, 2005.

[25] Ontotext Lab: OWL SEMANTIC REPOSITORY, 2006. `http://www.ontotext.com/owlim/OWLIMSysDoc.pdf`.

[26] Juhnyoung Lee: FREQUENTLY ASKED QUSTIONS ON ONTOLOGY TECHNOLOGY. Technical report, IBM T.J. Watson Research Center, 2004.

[27] Juhnyoung Lee *et al.*: TOWARDS ENTERPRISE-SCALE ONTOLOGY MANAGEMENT. Technical report, IBM T.J. Watson Research Center.

[28] Frank Manola and W3C Eric Miller: RDF PRIMER. `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`.

[29] David Martin: OWL-S: SEMANTIC MARKUP FOR WEB SERVICES. `http://www.daml.org/services/owl-s/1.1/overview/`.

[30] Deborah L. McGuinness and Frank van Harmelen: OWL WEB ONTOLOGY LANGUAGE OVERVIEW. `http://www.w3.org/TR/owl-features/`.

[31] Sheila A. McIlraith, Tran C. Son, and Honglei Zeng: SEMANTIC WEB SERVICES, 2001. `http://www.citeseer.ifi.unizh.ch/mcilraith01semantic.html`.

[32] Nilo Mitra: SOAP VERSION 1.2 PART 0: PRIMER. `http://www.w3.org/TR/2003/REC-soap12-part0-20030624/`.

[33] Paul V. Mockapetris and Kevin J. Dunlap: DEVELOPMENT OF THE DOMAIN NAME SYSTEM. In SIGCOMM, pages 123–133, 1988. `http://www.citeseer.ist.psu.edu/mockapetris88development.html`.

[34] Natalya F. Noy and Deborah L. McGuinness: ONTOLOGY DEVELOPMENT 101: A GUIDE TO CREATING YOUR FIRST ONTOLOGY. Technical report, Stanford Medical Informatics, Palo Alto, 2001. `http://www.cs.toronto.edu/~nernst/papers/noy-101-SMI-2001-0880.pdf`.

[35] The Sesame and OMM development teams: USER GUIDE FOR SESAME. `http://www.openrdf.org/doc/sesame/users/userguide.html`.

[36] Denny Vrandecic *et al.*: DLP - AN INTRODUCTION. `http://www.citeseer.ist.psu.edu/728245.html`.