EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

# Personalized resource recommendation for a semantic-enhanced platform for information resources

## Improving OntoAIMS

Sturm, Martin

*Award date:*
2007

Link to publication

**Disclaimer**
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

**General rights**
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computing Science

# MASTER'S THESIS

## Personalized Resource Recommendation for a Semantic-enhanced Platform for Information Resources

Improving OntoAIMS

by
M. Sturm

Supervisor: Dr. Lora Aroyo
External Adviser: Dr. Vania Dimitrova
Eindhoven, May 2007

# I. Abstract

With the rise of the Semantic Web and freely available web services, the possibilities of searching for information improved. Current information systems, however, often use old-fashioned methods for recommending information resources, based on user models which are based on the behaviour of the user instead of the actual knowledge of the user. An enhanced user model, developed for OntoAIMS, enables improved search methodologies resulting in a personalized resource recommender. This recommender is implemented for the OntoAIMS adaptive information management system which is used as an e-learning environment for testing the results of this recommender. This recommender provides a personalized recommendation system for students, using information in the ontology used for the domain representation and the knowledge of the user stored in an enhanced user model. Furthermore, the recommendation system automatically retrieves resources from the Internet and annotates the resources with available metadata and concepts in the ontology, which is used for improving the recommendation system. Based on an evaluation with real users, the OntoAIMS system is improved with the resource recommender. We present an analysis of this user evaluation and the benefits of a personalized resource recommender. A detailed design and implementation of a personalized resource recommendation system is presented in this thesis, using several components and web services.

# II. Acknowledgements

Special thanks to Lora Aroyo for advice and support during this project. Despite some problems halfway during the project, she keeps providing guidance and advice on the project. Also special thanks to Vania Dimitrova, who made it possible to visit the University of Leeds during this project and provided me with valuable advice on how to handle the project and allowed me to see how universities in other countries work. The work of Ronald Denaux was of much value during this project, especially the user tests he and Vania did during his project, leading to better understanding of the problems with OntoAIMS.

Thanks to all the people who gave advice on various aspects of the project, and were willing to comment on my presentations and work on the meeting groups. Among them, the students of the AROMA group and the students at Leeds.

Finally, I want to thank my gilrfriend Mieke Kleppe for her support and encouraging me to finish this project.

# III. Presentations and Collaborations

Several presentations were given during this project:

- **Visit to Leeds**. A two week visit to Dr. Vania Dimitrova at the School of Computing of the University of Leeds in March 2006 was part of this project. During this visit, to goals of the project was discussed and user tests done during previous projects were evaluated.
- **Presentation for AROMA Group**. On May 2006, a presentation called *Semantic-enhanced, agent-based search for information resources* was given for the AROMA research group, consisting of researches and students doing research projects and internships on ontologies and metadata.
- **Presentation for HERA-S Group**. March 2006, a presentation was given to the HERA-S group consisting of researches and students doing research on web engineering and web information systems.

This project was part of the SWALE project. Relevant web sites for this project are listed here:

| Web site | URL |
|---|---|
| **Demo web site** | http://wwwis.win.tue.nl/~swale/demo.html |
| **Project web site** | http://wwwis.win.tue.nl/~swale |
| **Basic Linux Ontology** | http://wwwis.win.tue.nl/~swale/blo |

# Contents


**1 Introduction** **5**
1.1 Motivation . . . 5
1.1.1 Scenarios . . . 6
1.1.2 Problem definition . . . 9
1.2 Research Questions . . . 10
1.3 Goals and Deliverables . . . 11
1.4 Project Overview . . . 11

**2 Analysis** **13**
2.1 Context . . . 13
2.1.1 OntoAIMS: Ontology-based Adaptive Information Management System . . . 13
2.1.2 User Model: LTCS and STCS . . . 16
2.1.3 OWL-OLM: Interactive Student Modeling . . . 17
2.1.4 OWL ontology language . . . 18
2.2 Approach to Solve Problems . . . 19
2.2.1 Integrate Local and External Search Engines . . . 19
2.3 Example of Recommendation Process . . . 21

**3 Design** **23**
3.1 Agent-based architectures . . . 23
3.2 Overview of Design . . . 25
3.3 Searchable Index of Resources . . . 25
3.3.1 Search Engines . . . 25
3.3.2 Resource Collections on the System . . . 27
3.3.3 Storing documents on the System . . . 28
3.3.4 Parsers . . . 29
3.3.5 Relating Concepts to Resources . . . 30
3.4 Resource Recommender . . . 32
3.4.1 Simple and Detailed Recommendation . . . 32
3.4.2 Simple Recommendation . . . 33
3.4.3 Detailed Recommendation . . . 33
3.4.4 Query Processing . . . 34

**4 Architecture and Implementation** **38**
4.1 Implementation Context . . . 38
4.1.1 OntoAIMS Packages . . . 38
4.2 Indexer Package . . . 40
4.2.1 OntoAIMS and Lucene . . . 41
4.2.2 `LuceneIndexer` . . . 41
4.2.3 `HTTPIndexer` . . . 42
4.2.4 `HTTPIndexerWorker` . . . 45
4.2.5 `IndexerThreadManager` . . . 47
4.2.6 Parsers . . . 48
4.3 Searching for Resources . . . 50
4.4 Search Package . . . 50
4.4.1 `ServerSearchAgent` . . . 50
4.4.2 `SearchWorker` Interface . . . 52

4.4.3 `QueryOptimizer` class . . . . . . . . . . . . . . . . . . . 56
4.5 Implementation on the OntoAIMS Client . . . . . . . . . . . . . . . . . 58
4.5.1 `ServerSearchManager` . . . . . . . . . . . . . . . . . . . . 60
4.5.2 `ConceptPack` and `DocumentPack` . . . . . . . . . . . . . . . . 60

**5 Conclusions and Evaluation** **61**
5.1 Summary . . . . . . . . . . . . . . . . . . . . . . . . . 61
5.2 Evaluation of this Work . . . . . . . . . . . . . . . . . . . . 62
5.2.1 Evaluation of Scenarios . . . . . . . . . . . . . . . . . . . 62
5.3 Evaluation of Research Questions . . . . . . . . . . . . . . . . . . 64
5.4 Future Work . . . . . . . . . . . . . . . . . . . . . . . 66
5.4.1 Improvements to Resource Recommender . . . . . . . . . . . . . 66
5.4.2 Related Future Research . . . . . . . . . . . . . . . . . . . 67

**A Source Code Documentation** **71**

# Chapter 1

# Introduction

## 1.1 Motivation

One of the most important aspects of the Semantic Web [2] is a high level of personalization. People expect that services know what they mean and take their preferences into account. However, most of the time users are not specifying in detail what they want from a service. A well-known example of this problem are search engines. When searching for information on a particular subject using Internet search engines, users typically enter very few query terms [1]. Usually, this leads to very large number of individual results. Search queries on a commonly used search engine like Google [16] often results in more than a million hits.

In order to make the envisioned Semantic Web possible it is necessary to incorporate the preferences and the perspective of the user. Users expect dynamic adaptation of a systems without explicitly specifying how. This desire resulted in the usage of user models in Semantic Web applications based on ontologies [4]. There are also several efforts to incorporate the use of user models to improve the usability of search engines, for example [20], in order to make it easier to find information on the Internet. These efforts are using user models based on the usage of the search engine and do not take the actual knowledge and perspective of the user into account. Until recently, user models were suited to deal with *closed-world assumption* where user models consits of *misconceptions*, based on the discrepancies between the semantics of the user and the expert. User models suited for *open-world assumption* will make it possible to deal with the dynamics of a user's conceptualization required to personalize the Semantic Web [11]. Using available knowledge from an ontology combined with an enhanced user model when searching for information is required to meet the expectation of the user. New approaches for using user models will make it easier to personalize search results and transform it into a recommendation of interesting resources.

Another trend, related to the Semantic Web vision, is the so-called Web 2.0. Web 2.0 refers to the latest developments in technologis for the World Wide Web, such as web services, social tagging (*folksonomies*), social networking and community maintained information (*wikis*) [21]. Another part of the Web 2.0 trend are so-called *mashups*, combinations of public available web service into novel applications. This trend made companies, such as Yahoo!, Google and Microsoft, to create free programming interfaces to their services. Combining these free services with ontologies, user models and other technologies makes it possible to combine the strenghts of the individual services into an application that provides new features or improved usability. New combinations of user models with freely available API's to web services makes it possible to improve the search results of existing Internet search engines.

This research is part of the SAPPHIRE project. This project is part of the SWALE project, which is a joint project between the Eindhoven University of Technology and the University of Leeds that investigates issues of knowledge, content and adaption alignment between students and instructors in the context of the application of Semantic Web technologies. During this project, an *enhanced user model* is created that takes several aspects of the user into account, such as knowledge, interests, preferences and conceptual state. The main goal of the SAPPHIRE project is to provide an *adaptive way for selecting resources*. In order to perform this task, an agent-based web-based application should be created which uses the enhanced user model created during the SWALE project. The adaptive selection of resources results in a personalized resource recommendation for a user with a specific user model. An integrated environment for personalized learning content management, called OntoAIMS, is used for prototyping the ideas developed during the SAPPHIRE project. The OntoAIMS environment consists of several parts, such as a enhanced user model which consists of a conceptual state of the user and deduct the knowledge of the user on specific subjects from this conceptual state. The user model in OntoAIMS also contains the preferences of the user and the usage of the system by the user, such as the topics the user has studied. A diagnostic dialog is also part of the OntoAIMS environment. This diagnostic dialog tries to probe the knowledge of the user on specific topics of a knowledge domain and creates or updates the user model with new information on the knowledge of the user. This is done by discussing concepts in the domain with the user and get an opinion on the user's conceptualization of the concept in the domain. The resulting user model is used throughout the enviroment for adapting parts of the system and the represented domain, concepts and resources in a personalized way. adaptive The recommendation of relevant resources to the user was the main focus during this project. This is done by improving the existing resource recommendation already available in OntoAIMS. User tests with OntoAIMS performed during a previous project [9], revealed several problems both in usability as well as in the functionality of the system. New combinations of existing technologies are used to achieve an improvement in the recommendation of the resources. The data available in the underlying domain and the user's conceptualization of the domain is used to annotate resources in OntoAIMS. This way, the problem of missing semantic annotation of documents available on the Internet is circumvented by annotating the documents dynamically using the available information in the domain and user model. This work demonstrates the following novel aspects:

- combining the enhanced user model with freely available webservices
- provide personalized search engine for various type of resources
- dynamically map concepts to resources from the Internet

### 1.1.1 Scenarios

To make the goals of this project more clear, and get an overview of the context of this project, we created several scenarios. These scenarios are based on the OntoAIMS application, because the eventual solutions will be implemented in this application. However, the scenarios can be applied on other information systems as well. OntoAIMS is designed to be very generic and thus it is possible to use the application for various tasks. The scenarios we have created are based on the situation where OntoAIMS is used as an e-learning environment. In this scenario, the application will support students following a course on Linux, the open source operating system. The system will be used as a support system for self-study by the students.

The scenarios described in this section are based on user tests of a previous project that improved OntoAIMS. These user tests were done with ten first year students of the University of Leeds. The

user tests were recorded on video, and these videos were studied before the scenarios were created. Unfortunately, only five of the ten videos were available. Also, the users answered some questions after the user test in which they had to give opinions on several parts of the system. As a result, many of the problems described in the scenarios were based on the findings in those user tests.

In the descriptions of the scenarios, the *Basic Linux Ontology* is used as the domain for the OntoAIMS application. This ontology was created during a previous project by Ronald Denaux [11]. This ontology defines concepts in the domain about Linux which are needed to understand the basics of the operating system. While we used the Basic Linux Ontology during this project, the results are not restricted to this domain. The scenarios illustrated in this section can be applied to other domains as well. In the description of the scenarios names of concepts are written in a `type writer` font.

#### Scenario 1: Recommendation of Pre-selected Documents.

Andrew is a student who wants to learn about Linux. He has not much experience with the operating system, and has not much knowledge on the concepts of the operating system. Andrew is recommended to use an ontology-based information environment that enables him to learn about the various aspects of the Linux operating system.

This sytems uses a *Dialog Game* to decide the level of experience Andrew has with Linux. The system thinks his knowledge on the concepts of Linux is very limited, and he should start with the first task, about `files`, `filesystems` and `operation on files`. The system presents a visual representation of the relevant concepts and the relations between the individual concepts. This visualization is done using a graph with vertices and edges. Because most of the concepts are not known by Andrew, he would like to quickly get information on the concepts represented by the nodes.

In existing systems, Andrew can search for the names of the concepts using existing search engines, such as Google or Yahoo!. However, the results will be very broad and not specific to the concept represented in the domain. Also the number of documents the search engine will return, can be very large. Other systems allow to search within a predefined set of documents selected by a course creator. These documents are typically relevant to the course and explain the basics of the concepts. However, these documents are presented in an unordered list or ordered by topic. It is not easy to determine which concepts are related to these documents.

Imagine that when Andrew clicks on a concept in the domain, a small list of documents are displayed containing basic information about this concept. This way, Andrew can quickly learn what the various aspects of a concept are. For example, Andrew wants to know how one can `copy files` in Linux. In the ideal situation, he clicks on the concept representing this operation and the system will present basic documentation about the *cp* command and explain how it can be used. The documents about this concept can be selected by a teacher which created the course. The teacher only has to add the documents to the domain, but doesn't have to specify to which concepts the documents are relevant. This elimates the laborous work of reading the entire document and wading through all the concepts in the domain in order to decide which are relevant for the document.

#### Scenario 2: Automatically Search Documents on the Internet

Simon is another student who wants to learn about Linux. He also wants to use an ontology-based information environment which supports him in learning about Linux. Simon has experience with other Unix-based operating systems, and therefore has knowledge about the most important concepts of these operating systems. Because Linux is very similar to other Unix versions, many concepts in the domain of the system are familiar to Simon.

The *Dialog Game*, which probes Simon knowledge on Linux, thinks that he knows a lot on Files and File Operations, but not some details. Simon knows how files are used in this kind of operating systems, but wants to learn about the details of file operations, more specific: he wants to learn about copying files in Linux. Just like when Andrew is browsing the domain, the system will return documents on the basics of a concept when Simon clicks on a concept. When he clicks on copying files, the system recommends basic documentation on the usage of the cp operator. Because Simon knows this operator already, he wants to get more detailed information on the copying process and which flags are available for the *cp* command.

Because the author of the domain did not think of more advanced users when collecting documents on copying files, the system has no available resources on the copying of files. Obviously, on the Internet are many resources describing the inner details of the *cp* command and all the available parameters. Imagine a system that automatically searches for these documents on the Internet, using the available data in the domain combined with the available information the system has on Simon in a user model. The system can automatically create detailed search queries in order to find documents which match the knowledgelevel of Simon. The resulted documents should be presented in a similar way as the documents already available in the system, and automatically *mapped to relevant concepts*.

#### Scenario 3: Accumulating Feedback of Users when Recommending Resources

While Andrew is using the system and browsing through the domain, he reads various documents explaining the concepts in the domain. While he is reading a document about users in Linux, he does not understand it. The documents explains how the list of users are stored in the \etc\passwd file and how the password of a user is stored. This is not the information Andrew was looking for, because he wanted to know how users are created and how he can change his password.

Because the system think the document Andrew was reading, was relevant for the concept user, the next time another user of the system wants information on users, the document will be recommended again. When Andrew has the possibility to mark the document is not interesting for him, the system can use this information when another user wants recommended documents for this concept. However, Andrew has a limited knowledge on the concepts in Linux and wants basic information on the concept of users. Simon, however, wants more in-depth information about the concepts, and may think the document Andrew was reading is relevant to the concept of users. The system has knowledge of the conceptualization of the domain for every user, and can relate this conceptualization when processing the comment of the user on a document. In this scenario, the system can mark the document Andrew was reading not relevant for users with a low level of understanding on users in Linux, but mark it relevant for users with a more advanced notion of this concept.

#### Scenario 4: Reading Documents and Searching For More Information

Assume Andrew is using the system and busy reading documents on the various concepts. Andrew has not only little experience with Linux, but also the Internet is rather new for him. While browsing through the domain, he reads documents. One particular document is difficult to understand. The document is a website on the Internet explaining how e-mail clients can be used in Linux.

Andrew has several problems with the document. The layout is very distracting because there are lots of advertisements on the site and the background is very distracting in his opinion, because the colour of the background is very similar to the colour of the text resulting in a low contrast. Also he reads in the document about *netiquette* and has no idea what this means.

**Layout** The problem with the layout is a fundamental problem with the Internet in general. Every webmaster can layout the documents to his taste, which can result in documents which are very hard to read. In an ideal situation, the user can indicate that the layout of a document is very distracting. In that case, the system could extract the contents of the document and present it in a more readable way.

**Free Text Search** Because Andrew does not know what *netiquette* are, it is understandable that he wants to get more information on this. The Linux domain, used by the information environment Andrew is using, does not contain a concept on *netiquette*. However, other documents in the system may have information on this subject. Imagine that it is possible to search on the word `netiquette` where the system takes the level of knowledge on other subjects of Andrew in consideration. It would be much easier to find relevant documents, compared to when he would search using a standard search engine such as Google or Yahoo.

### 1.1.2 Problem definition

The illustrated scenarios in the previous section, reveals some problems in the current implementation of adaptive information systems such as OntoAIMS on the area of resrouce recommendation. In order to get a clear overview of these problems, we list the problems and a short discription of them. Also the possible implications for the system are mentioned. The problems listed in this section are based on the observations during the user tests of a previous project, which also resulted in many of the scenario's defined in the previous section.

- **Users have different knowledge levels.** Adaptive information systems have detailed user models available with data on the knowledge of the user of concepts in the domain, by relating the user model to the ontology. This information should be used when recommending resources.

- **Deal with manually selecting resources.** In existing adaptive information systems, course authors have to manually select appropriate resources for the domain. In some systems, authors also have to manually add metadata to the resources, such as *title*, *author* of the resource, *summary* of the contents and relevant *keywords*. Techniques are required to automate this task or at least parts of it, such as manually adding metadata.

- **Make resources better available.** The available resources in the system, often added by the creator of a course, should be easily accessible and available for the users of the system. Recommended resources, i.e. resources which are relevant for a certain concept, should be displayed when the user clicks on a concept. This way it is easy for the user to get more information on a concept.

- **Make resources from the Internet available.** Based on the user model, containing the level of understanding of a domain, an adaptive information system has possibilities to make searching for information on the Internet which are relevant for the concepts in the domain, easier. These resources from the Internet should be added when the user thinks the already available resources are insufficient to fulfill its requirement for information on a concept.

- **Recommend additional resources when the available resources are insufficient**. In the current implementation of OntoAIMS, several concepts have no related concepts. The system should add automatically additional related concepts. These concepts can either be a selection of the resources added by the course creator, or from the Internet (see previous two items).

- **Enhance accessibility of resources.** Because the documents in adaptive information systems are often websites on the Internet, maintained by others, the accessibility can be a problem. Distracting layouts or a small font size can decrease the readability. This problem should be solved by automatically enhance the readability by removing distracting and irrelevant elements or changing the format of existing elements.
- **Opinions of users on the relevance of resources should be taken seriously.** The opinion of users on the relevance of a resource to a certain concept can be used in order to improve the recommendation of resources for that concept. This requires methods enabling users tag the relevance of resources, and use these tags in the recommendation process.
- **Merge several resource collections into a single recommendation system.** Because the previous defined problems assume access to resources stored at several locations (i.e. the Internet and the local informatioin system), the system should be easily be adaptable in order to access these resources and present them to the user. This can be done by modularize the system using *agent based* or *service oriented architectures*.

## 1.2 Research Questions

The problem definitions in the previous section results in a list of research questions that need to be answered in order to improve adaptive information systems. In this section we list the research questions we tried to answer during this project.

- **Personalizing resource recommendation.** How can the system recommend resources to the user based on the knowledge of the user and the needs of the user? This requires methods for incorporating the user model for recommending resources.
- **Automatically recommend resources from the Internet.** While there are usually resources available in the system which are added by the domain or course creator, in most cases the number of these resources are limited. How can the sytem acquire additional resources from the Internet (or other information repositories) while ensuring that the resources are relevant for the user?
- **Using ontology information for annotating resources.** In existing adaptive information systems, authors can add information resources to the system, but usually have to annotate these resources with as much metadata as possible. This causes errors in the metadata and often results in incomplete metadata. How can we make this annotation process easier in order to improve the quality of the metadata and as a result improve the resource recommendation?

Additional research questions that are relevant for this project, but will not be the main focus are stated below:

- **Improve the accessibility of information resources.**
  - Is it possible to improve the visual presentation of the information resources using existing technologies?
  - How should the user interface of a adaptive information system behave in order to make it easy to use?

## 1.3 Goals and Deliverables

During this project we tried to answer the research questions stated in the previous section. By improving the existing OntoAIMS adaptive information system with a personalized resource recommender. Combining the open user model based on the ontology used for the domain in the system with existing search technoloies is in our notion the most practical approach to implement a personalized resource recommender.

This project resulted in the following deliverables:

- The design, architecturye and implementation of a personalized resource recommender for OntoAIMS incorporating a system for retrieving information resources from the Internet and annotating these resources automatically using information in the domain used by the application.
- A literature review of existing search technologies that could be implemented in OntoAIMS.
- A prototype of OntoAIMS implementing a personalized resource recommender using an enhanced user model based on ontologies. The prototype also uses recommend resources from the Internet which are annotated using the information available in the ontology used for representing the domain in OntoAIMS.
- A presentation for the AROMA group and HERA-S group on the research part of the project.
- A final presentation on the research part of this porject and the architecture, design and implementation of the prototype.
- A Final thesis report describing the research and implementation part of this project, including the process of this project.

## 1.4 Project Overview

This section will document the several phases of this project. The project started with an **introduction phase**. During this part of the project, an initial research was performed to the existing implementation of OntoAIMS and possible research topics. The various technologies which were used in the existing implementation of OntoAIMS, as well as technologies which were relevant for this project were studied, such as *ontologies*, languages and methods to describe ontologies, technologies for implementing client-server applications and *service-oriented architectures*. On the field of personalization research was done in user models and how they could be used in applications.

The next phase of the project was the **research and analysis phase**. During this part, which was partly done at the University of Leeds, the scenarios and problem definitions were created and researched. Evaluations of previous projects were studied and used while defining the new additions to OntoAIMS. Part of this part of the project was the study and evaluation of user tests part of a previous project on OntoAIMS. The problems identified in these tests were used as a starting point for this project.

During the **design and implementation phase** the personalized resource recommendation in OntoAIMS was designed and implemented. The research and analysis phase were not clearly seperated, because during the design and implementation phase, we descovered that some ideas were not possible and we needed to focus on a different approach.

Finally, the entire project was documented during the **documtation phase**. This report was written and a final presentation was created based on the results of the project. Also the architecture and implementation of the new OntoAIMS prototype was documented during this phase.

The various parts of this project are presented in the various sections of this document. Chapter 2 presents an analysis of the problems and possible solutiosn as described in section 1.1.2. Based on this analysis, the design for a personalized resource recommendation system incorporating Internet resources was created. This design is described in chapter 3. Based on this design, a prototype was implemented using the existing OntoAIMS information system. The architecture of this implementation and the implementation itself is described in chapter 4. Chapter 5 summarize and evaluates the project and relate the work to relevant research and possible future work.

# Chapter 2

# Analysis

In this section we expand on the context for this project and introduce a approach for solving the problems illustrated in chapter 1.1. A design is proposed in this chapter to achieve the goals and solve these problems.

## 2.1 Context

In section 1.1 we already mentioned OntoAIMS as the context for this project. From the beginning of the project, the goal was to improve this application. During several previous projects [11] [23], OntoAIMS was used as a platform for creating prototypes for several research projects. The personalized resource recommender created during this project is implemented in OntoAIMS. This platform was chosen because of the previous experience with OntoAIMS during these projects. Another advantage of using OntoAIMS was that during a previous project [11] several user tests were done, which could be used as a starting point for this project. The OntoAIMS system and the OWL-OLM knowledge acquiring system are presented in this section. The approach for solving the problems defined in section 1.2 is described in this section 2.2.

### 2.1.1 OntoAIMS: Ontology-based Adaptive Information Management System

The research within this project is based on the existing content management system OntoAIMS. This system evolved during several research projects which enhanced and improved the system. Initially OntoAIMS started as *AIMS* which did not had the possibility to use an ontology defined in *OWL*. This last possibility was added during a previous research project [11]. This project created also *Dialog Game* based on the OWL-OLM student modelling system. This sytem is used for creating and updating the user model in OntoAIMS.

OntoAIMS is a system implemented using a client-server architecture. The server part contains the datamodels used by the system and the actual information. Most of the processing of the data is done on the server. The client part is in fact a viewer for the data contained on the server, but consists of several parts for the different users of the system.

Figure 2.1 shows a screenshot of the *Viewer* of OntoAIMS. In the center of the image, the visual representation of a subset of the domain is visable. The right part of the screen contains a list of resources, recommended by the system. The bottom of the screenshot shows the search box, which can be used to enter custom search queries. The empty space below the search box is available to display information on a resource.

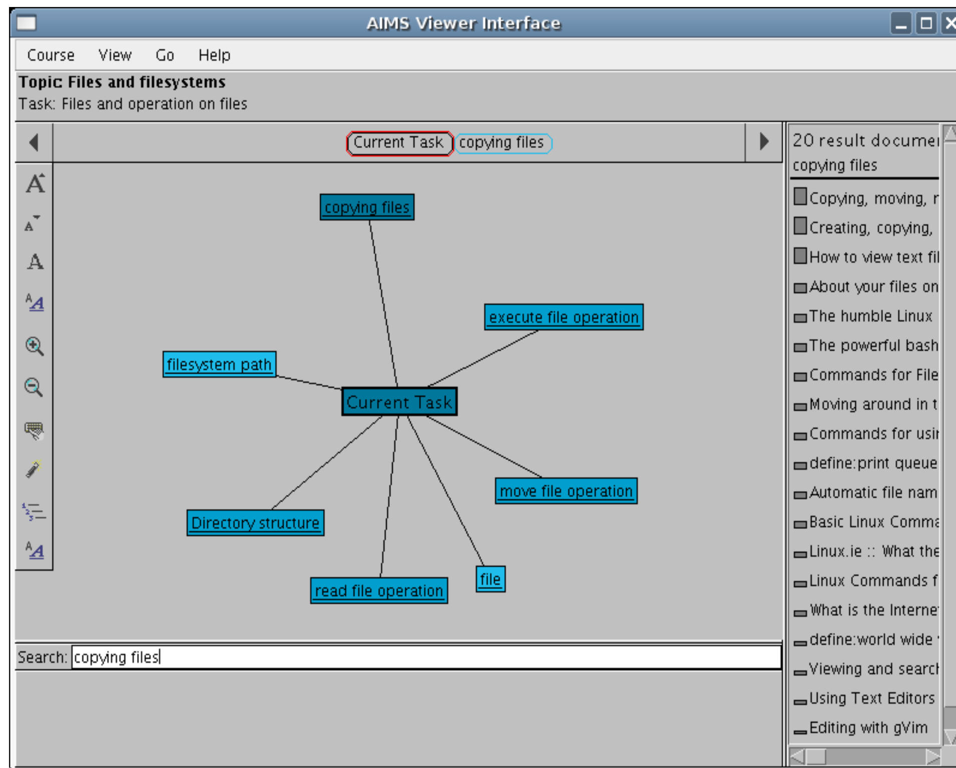


Figure 2.1: AIMS search and browse window

The data models and the actual data are located on the server.

- **The domain model.** The actual ontology in OntoAIMS is represented by the domain. In a large part of the implementation of OntoAIMS, the domain consists of *concepts* and *links*. The links between the concepts are used for indicating relations between them. OntoAIMS supports multiple domains.

  Figure 2.2 shows the architecture of OntoAIMS. The server contains the various data models and retrieves the distributed resources. The OntoAIMS Client is the client for the system which is used by the users of OntoAIMS. It contains the domain browser and resource recommender. The OntoAIMS Authoring Environment is used by the course creators and domain authors - in the figure represented by the *author(s)* - and provides an interface for creating and modifying the various information models.

- **The library.** The library in OntoAIMS contains the resources added by the author of the domain or a course. These resources are not stored themselves, but a *URL* is stored. Resources are related to a concept. Resources can be for example documents, articles, books, images, video's, sound and presentations. A resource in the library can belong to several courses and different domains. In figure 2.2 the resources are represented by the *distributed resources* data collection and used by the Library.

- **Course model.** On top of a domain, courses can be created. A course consists of *Tasks* and *Topics*. Tasks are described in terms of concepts: required knowledge is described using con-

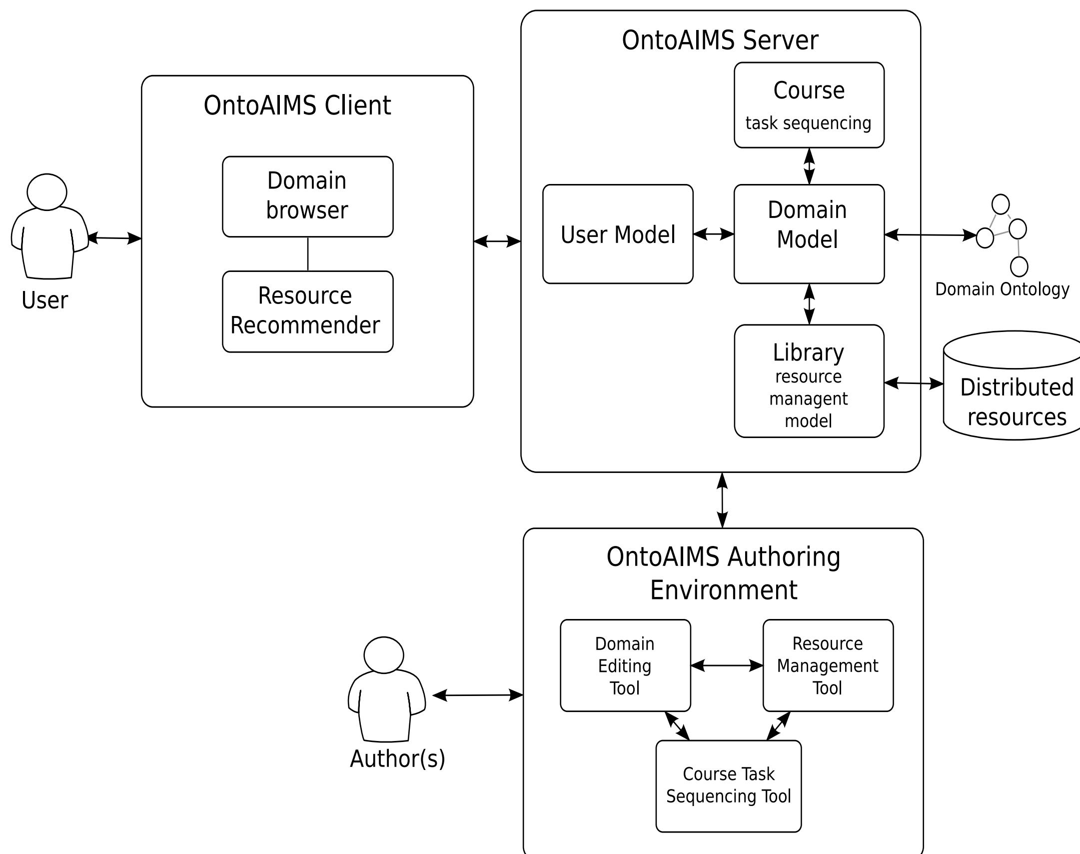


Figure 2.2: OntoAIMS architecture

cepts and the taught material is also defined using concepts. Tasks are hierarchical structured. A topic consists of a collection of tasks that cover a particular subject.

- **User model.** The user model contains information about the user. This model keeps track of the tasks the user has finished and contains an overlay of the domain annotated with the knowledge level of the user of the individual concepts and links.

The data models in OntoAIMS are stored in plain text data files. Individual aspects of the data are stored in individual files. The domain, consisting of *concepts*, *conceptlinks*, *linktypes* and *document references*, are stored in four seperate files. The same holds for the other four data models used by OntoAIMS.

For the client part of OntoAIMS three different parts can be distinguished:

- **Viewer** This part is the most commonly used client and allows a user to browse the domain. In the scenarios used during this project, this part is used by the students.
- **Editor** This part is used for authoring the domain, the course and the resources. This part is typically used by the course creator and the domain author and in fact consists of several subsections: a domain editor, resource editor and course editor. The editor is visible in figure 2.2 as the OntoAIMS Authoring Environment. The course creator and domain authors are called *Author(s)* in the figure.

- **Admin** The admin part of OntoAIMS is only used by the system administrator and makes it possible to create users and user groups. Also this part makes it possible to set the rights of the users and groups. This part of OntoAIMS is not visible in figure 2.2, because it is not relevant for the description of the workings of the OntoAIMS system.

When the support for OWL was added to OntoAIMS, the existing architecture was kept for a large part. The OWL ontology is converted to the old data storage structure, causing loss of information. The notion of classes, subclasses and instances in OWL are reduced to concepts, making it impossible to decide whether a concept is an individual or a subclass of a particular class.

The user model in OntoAIMS is split in two parts. The first part has relatively static information about the the user, such as the user name, the tasks the user has completed and so on. The other part is a OWL based representation of an overlay on the OWL domain. This part is used for keeping track of the knowledge level of the user on the concepts within the domain.

### 2.1.2 User Model: LTCS and STCS

The overlay on the OWL domain in the user model is in fact another ontology. This user model is a reresentation of the *Conceptual State* of the user, which is a representation of the user's understanding of the domain. There are two different conceptual states used in OntoAIMS: a *short term conceptual state* and a *long term conceptual state*. The short term conceptual state, or *STCS*, is a temporary simplified representation of the understanding of the domain by the user. The STCS is updated when the system is interacting with the user, for example when the knowledge of the user is probed by the system using the *Dialog Game*. The STSC registers the actions of the users. The following aspects are stored in the STCS:

- **correct use:** The number of times a user has used a concept or a relation between two concepts correctly.
- **wrong use:** The number of times a user has used a concept or a relation between two in a way that contradicts the definition of the concept or relation in the domain.
- **total use:** The number of times a user has used a concept or relation, either wrong or correct.
- **affirmation:** The user has indicated that he or she knows about a concept or relation.
- **negation:** The user has indicated that he or she does not have knowledge on a concept or a relation.

When the interaction with the user is finished, the data in the STCS is used to update the long term conceptual state, or *LTCS*. The data in the STCS is not stored on the system for use between different sessions. The LTCS represents the knowledge of a user based on a longer period. Therefore, the LTSC is stored on the harddisk making it possible to reuse the user's conceptualization in subsequent sessions. The LTCS is a permanently updated representation of the user's conceptualization of the information in the domain.

The LTCS has only one value for each concept in the domain or for a relation between two concepts in the domain. This value is called *belief value* and represents the level of understanding of the user for a concept or link according to the system. The belief value is an integer value between 0 and 100. This belief value is an easy to use reppresentation of the knowledge of the user on a specific concept in the domain by the system. It is easier to use a single integer in an application for determining the knowledge level of a user on a specific concept than the individual values contained in the STCS.

OntoAIMS suffers from the problems defined in section 1.1.2. It lacks a personalized resource recommender, while a detailed user model is available. The number of available resources for courses in the system are often limited, because resources have to be added manually. The primary goal is to improve OntoAIMS by creating a resource recommender that solves these problems by creating a Internet-aware, personalized resource recommender that uses the available information in the user model.

### 2.1.3 OWL-OLM: Interactive Student Modeling

OntoAIMS contains an integrated interactive student modeling system called OWL-OLM [10]. This system uses a *Dialog Game* to get an opinion on the knowledge of the user. In the proposed scenarios, presented in section 1.1.1, the user is typically a student. The goal of a Dialog Game is to probe the knowledge of the user in a dynamic way. Usually, learner systems use predefined knowledge levels for the users. The problem is that users are not identical and the predefined knowledge levels cannot deal with individual preferences and variation in the knowledge on parts of the domain of a single user. For example, it is possible that a student has a high level of knowledge on a particular part of the domain, but has a low level of knowledge on another part. A predefined set of levels used by a learner system can not deal appropriately with this kind of variation.

Detailed user models are used by dynamic learner systems to providing students with personalized content. The user model is created based on the actions and preferences of the user, but not on the actual student's conceptualization of the domain. OWL-OLM tries to solve this problem by actually probing the knowledge of the student using an interactive system. An additional bonus is that this system also deal with the *cold start problem*. This cold start problem arises when an adaptive information system uses a user model, but the system has not yet created a user model for this user.

The OWL-OLM system is able to detect mismatches in conceptualization between the course designer and the students. OWL-OLM can confirm an already available conceptualization by a student or adapt the user model when needed.

#### Dialog Agent

The main component of the OWL-OLM system is the *Dialog Agent*. The Dialog Agent maintains a user-knowledge acquisition dialog. This dialog serves various purposes. It will create or update a detailed user model. For the student, the dialog with the OWL-OLM system can be used to clarify concepts in the domain or anwer questions. It is also possible that the OWL-OLM system gives the user recommendation for the next learning task. The dialog with the user is an interactive process. The system can ask questions to the user to get an indication on the knowledge of the user on a particular part of the domain. These questions can also exists of statements on which the user can react. The user can also ask questions to the system or make statements in order to confirm his or her conceptualization of the domain.

During the interaction with the user, the Dialog Agent will maintain a *short term conceptual state*. When the dialog is finished, and the system has optionally recommended a learning task for the student, the information in the short term conceptual state is merged with the *long term conceptual state*.

A screenshot of the user interface of the OWL-OLM system, showed in figure 2.3, consists of three parts. The first part, at the top of the window, presents the conversation between the system and the user in a way that is similar to instant messaging programs or IRC clients. The second part, located at the lower part of the window, consists of a graphical respresentation of the current subject

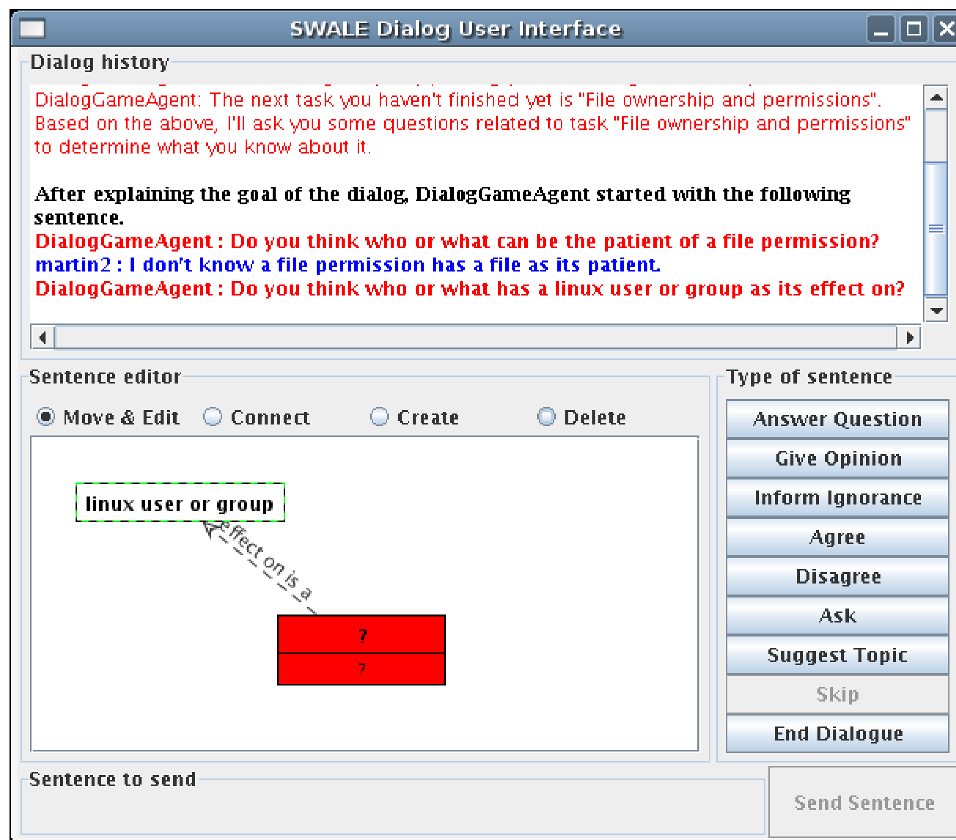


Figure 2.3: OWL-OLM Dialog Game window

the system or user is talking about. This visualization supports the dialog between the user and the system. The third part, located at the bottom and the right edge of the window, is the input part for the user. Using the buttons on the right, the user can create sentences and send these to the system.

OWL-OLM uses an OWL ontology representation of the domain in order to create sentences and process the responses of the user. The user model is also based on the ontology representation of the domain on which the knowledge of the domain is projected. For processing the information stored in the OWL ontologies, OntoAIMS uses the *OWL generic reasoner* of Jena[1]. This reasoner is used for reasoning on the ontologies and updating them with new information.

### 2.1.4 OWL ontology language

OntoAIMS uses the OWL ontology language[2] for representing the domain. OntoAIMS also has a description of the domain using concepts and links, referred to as the *domain model*. The OWL ontology language is used for reasoning on the domain. This is not possible with the internal representation of the domain used by OntoAIMS. The OWL ontology language is based on XML making it easy to transfer and use the domain representation on other systems. The OWL ontology language describes a domain using *classes*, *properties* and *individuals*. Individuals are representations of real-world con-

[1] See http://jena.sourceforge.net/ for more details on Jena
[2] See http://www.w3.org/TR/owl-features/ for more information on OWL

cepts and objects. These individuals can be grouped by classes, which are used to group individuals with common properties. Furthermore, OWL uses properties for describing relations between individuals of particular classes. OWL exisists in three variants which differ in the applied restrictions to the flexibility: OWL Full, OWL DL and OWL Lite. OWL Lite and DL guarantee computational completeness, while OWL full provides maximum expressivenss, but does not guarantee computational guarantees. These restrictions ensure that it is possible to perform reasoning operations on the ontology. In OntoAIMS, the OWL representation of the domain is mainly used by the OWL-OLM part for reasoning purposes. The OWL-OLM system creates questions and statements based on the OWL representation of the domain and checks the statements, answers and questions of the user using the reasoning features of OWL. A detailed description of the integration of OWL in OntoAIMS can be found in [11].

## 2.2 Approach to Solve Problems

To tackle the problems with resource recommendation in adaptive information systems defined in section 1.1.2, a design is created of an improved resource recommender. This resource recommender uses the enhanced user model available in OntoAIMS to present the user a personalized list of resources for the concepts in the domain.

The initial approach was to integrate existing web services in the adaptive information system providing a search engine for resources on the Internet. This search engine would be used by the application to search directly for relevant resources. Using the results of the public search engine, the system would make a selection of the resources. The resources would not be processed or investigated by the system, in order to make the system lightweight and keep the design easy. The queries used for the search actions would be refined by the application based on the information available in the enhanced user model. However, after some initial researche, we discovered that this was an unfruitful approach. The available search engines were too restrictive in the flexibility of the search queries, which resulted in relatively short queries, making it impossible to get useful results. Another problem was that the results provided by those search engines consisted of URL's with some basic metadata. This metadata was not sufficient to determine if a resource was relevant to a concept or would be interesting for the user. While this approach would solve one of the problems introduced in section 1.2, because resources from the Internet would be available to the system, it would not make it possible to personalize the recommendations and improve the recommendations of the resources added by the course creator or domain author.

### 2.2.1 Integrate Local and External Search Engines

A new design uses a different approach. The system will maintain a collection of resources relevant to the domain and use the external search engines available as a web service for collecting the resources which are stored in the local collection. The metadata provided by those search engines is used, but the system also downloads every resource which could be interesting to the user and process the content of the resource. The metadata of the resource will be extracted from the resource, possibly using external repositories for the metadata, and stored in the collection of resources used by the system. This tackles one of the problems with the initial approach, namely the lack of metadata provided with the results of search engines available as web service. The local collection of resources maintained by the system is indexed in such a way that it is possible to search the contents and metadata of the resources easily. This approach provides a very rich collection of information on the resources

which can easily be searched, and provides a lot of flexibility in processing the resources. Various improvements of the resource recommendation are possible using this approach:

- **Automatically add additional documents to the system.** In OntoAIMS it is required that every document in the library is added manually by a course or domain author. This is a labourous task resulting in a limited number of documents for a concept. The system should search on the internet when the user wants a recommendation of relevant resources. This search can be done using existing search technologies, such as Google, Yahoo, AltaVista, Excite and Ask. The user model should be used in order to get resources which are understandable by the user. Queries are created by the user, but expanded by the system using the information in the domain combined with the user model.
- **Extract metadata from documents.** Relating information resources to the correct concepts is of high importance. This is the only way the system can decide if a resource is relevant for a concept and also if it is relevant for the user when he or she wants a recommendation for a concept. The metadata of the documents should be extracted and used in order to map the concepts to the documents. Most document formats on the Internet have embedded metadata information, such as the author of the document, a summary, relevant keywords, creation date and last modification date.
- **Enable users to search the entire contents of resources in the library.** In OntoAIMS, only a URI to the document is stored in the system with the available metadata. Therefore it is not possible to search through the entire contents of the documents, which makes it harder to map a document to a particular concept. The system should index the entire contents of the documents. This requires local storage of the documents on the Internet and a full-text index on those documents. The system should create detailed queries in which the information in the domain is combined with the knowledge level of the user on the concepts in the domain, which is stored in the user model.
- **Map concepts in the domain to documents in the library.** The resources stored on the system should be related to concepts in the system. This way, it is possible to easy recommend relevant resources to the user by searching for documents related to a certain concept. The process of mapping resources to the concepts in the domain should be done automatically by the system using *information extraction* techniques.
- **Using UM to order recommended resources.** When the user wants to read information resources on a certain topic or concept, the system recommends relevant resources. When the system has several recommended resources, the resources should be ordered in a way that the most interesting resource for the user should be at the top and marked as most relevant. This should be done based on the preferences, knowledge and behaviour of the user. When the knowledge level of the user on a certain topic is limited, the system should put resources which are written for people with limited knowledge on a subject on top. This requires that the system has an opinion on the public the resource is written for and store this with the resources.
- **Recommend additional resources when needed.** The system should give a basic set of recommended resources in normal situations. This set should contain resources which are very relevant to a topic or concept and are most of the time the same for all users. However, when a user wants more information on a concept or resource, the system should give the possibility to present additional resources to the user. These additional resources are a subset of the resources

stored already on the system by the creator of the course and, when necessary, expanded with resources available on the Internet which are not yet known by the system. These resources should be added dynamically to the system and annotated in the same way as the other resources on the system. These additional resources from the Internet are collected using external web services providing access to repositories of information resources. Examples of such web services can be commonly known web search engines, but also bookmarking sites.

- **Enhance resources by improving layout.** Resources with distracting layout or bad readability, should be processed by the system in order to improve the accessability of the resource. The system should extract the content of the resource and display it in a more friendly way to the user. The actions for this process depends on the type of information resource. Processing of resources with video's, sound and images is difficult. Resources with textual content, such as HTML-documents or Word-documents, can be processed in various ways. The existing document can displayed using a different style, which requires less processing, but also the contents of the document can be extracted using parsers and displayed in a generic way, which requires a bit more processing time.

- **Deal with changing resources.** The system uses a dynamic collection of resources. Most of the resources will be located on the Internet and maintained by others. This will result in changes in the content of the resources or even the dissapearance of resources. In existing adaptive information resources, changed resources or resources that are not available anymore will still be recommended by the system because it is not aware of the changes in the resources. This could be solved by periodically checking if a resource is still available and if the contents are still relevant for the concepts to which it is related.

## 2.3 Example of Recommendation Process

Before we explain the design of the personalized resource recommender system, we will give an short example of a concrete recommendation process to illustrate the working of the recommendation process. This description will introduce relevant terminology and illustrate the working of the created design. The example is based on the scenarios presented in section 1.1.1.

The user of the system is a *student* who is following a beginners course called *"The Linux Operating System"*. He follows an interactive web course on the subject. The system has no user model on the student yet, so the first time the student logs into the system, knowledge acquisition is required in order to create an initial user model. This acquisition process deals with the *cold start problem* and makes it possible to present a dynamic personalized course to the student.

When the user logs into the system, a *Dialog Game* is started automatically. The system discusses the topic *"Files and operations on files"* to get an understanding of the user's conceptualization of this topic and suggest a task the user should do. The dialog asks questions to the user and processes the responses of the user. Because the initial user modelling is not the main subject of this project, we will not go into detail on this process, but a short snippet of the dialog is presented here in textual form to give an idea of this process:

**DialogAgent**: Do you think a `file` has a `unix user` as its `owner`?

**User**: Yes, a `file` has a `unix user` as its `owner`.

**DialogAgent**: Do you think `a file` has a `path` as its `full name`?

**User**: I don't know whether a `file` has a `path` as its `full name`.

After this dialog, the system suggests the user to do a task on the topic of *"Files and operation on files"*, because the system thinks that the user does not understand all concepts in this section to their full extends. The student agrees with this and wants to learn more on `copying files` and `moving files`. The system finishes the dialog with the student and opens the concept browser and resource recommender presenting a relevant part of the domain. The system shows the concepts relevant for the task *"Files and operation on files"*, with concepts such as `file`, `filesystem path`, `copying files` and `move file operation`. No recommended resources are presented yet.

First, the student wants to know a little more about files in general. He clicks on the `file` concept. The system recommends immediately a small number of resources which are very related to this concept and explain the basics of this concept, such as a resource titled *"About your files in Linux"* and *"Automatic file name completion in Linux"*. Also the other concepts which are tightly related to the recommended resources are displayed to give the user a better understanding of the context of those resources.

The student reads these two resources and decides that he knows enough about files in general in Linux. Now, he wants to learn more on copying files in Linux. The user already knows how the `cp` command works in Linux, which is used for copying files. The student navigates through the visual representation of the domain and clicks on the concept `copying files`. The system again suggests a limited number of relevant resources, such as *"Copying, moving, renaming and removing files in Linux"*. However, the user already knows this and thinks this resource does not contains new information. So, the student wants additional resources. He requests more resources.

The system now searches for more information taking the knowledge of the user into consideration. Additional resources are now suggested and the resources which are too basic for the user are omitted. New resources recommended by the system are for example *"The powerful bash shell wildcards in Linux"* and *"The Virtual Filesystem in Linux"*. These resources are targeted at a more advanced audience and gives the student additional information on ths subject which increases his knowledge.

# Chapter 3

# Design

The design of an adaptive information system incorporating a personalized resource recommender is described in this chapter. The personalized resource recommender is implemented in OntoAIMS, an already available adaptive information management system designed for use in learning environments. Crucial parts, required for the design of the personalized resource recommender are already available in this system, saving us a lot of effort.

Because the design is made for OntoAIMS, this system is often referred to in this chapter. The design is described in a generic way, making it possible to implement it in other information systems. However, some parts are specific tailored for OntoAIMS. As a result, the presented design will require some changes in order to use it in other systems. The design for OntoAIMS the supported document types for resources is limited to two: HTML documents and PDF documents. The description of the design sometimes refer to these document types.

In section 3.2, the design of the OntoAIMS system is presented as it is created in previous projects. In section 3.3 we describe the design for the in OntoAIMS integrated searchable index of resources including the mapping of concepts to resources, described in subsection 3.3.5. Section 3.4 describes the design of the actual resource recommender including the processing of query refinement in subsection 3.4.4.

## 3.1 Agent-based architectures

One of the important requirements of an application is a flexible design which is easy to extend. In an information system similar to the system used for this project, it is even more important to create a design which can be easily modified and extended, because of the fact that the system is updated and improved by other researches and students during a long period. Considering this fact, a short study was done on the possibilities to create a design which is prepared for future changes and improvements. For applications which are specificly developed for use as teaching and learning environment (TLA), the IEEE has published a standard for the architecture of these applications [18]. This standard proposes a framework for learning applications in a way which exhibits the characteristics of a *Service Oriented Architecture* [22]. Because our application is not intended as a pure e-learning environment, this standard is not very relevant for this work.

Since the current implementation of OntoAIMS consists of several seperate agents which provide distinct services. For example, the user profile agent provides information on the current conceptual state of the user, while a domain agent provides services which apply to the domoain model of the system. An agent-based architecture is a obvious approach in this case. A popular variant of this

approach are Service Oriented Architectures (SOA), which consists of loosely coupled, highly interoperable application services forming a application when combined. A service is implemented in an agent, which can use an arbitrary language for implementing the services. Service Oriented Architectures often use web services, which are services that use protocols which have their origin on the Internet, such as HTTP and HTTPS. Standard based protocols are used for creating web services and passing messages between the various individual services [3]. Commonly used standards for for communication with web services XML-based protocols such as SOAP, sometimes referred to as *Simple Object Access Protocol* [26] and REST [14]. However, the protocols used for communication with web services are not limited to these two. It is possible to use other protocols as well, which do not use XML, but this will restrict the interoperability. Before a web service can be used, a description of the functionality provided by a web services should be available. A proposed standard for this task is the *Web Services Description Language* (WSDL). This XML-based language is not yet an official standard endorsed by the World Wide Web Consortium, but a draft is available [5].

Not only the communication between web services is important, because in order to use a web services it should be discovered first. The discovery of a web service can be done manually, where a software developer manually enters the location of the web service and write programming code for communicating with the webservces. However, there are also protocols created for automatically discovering web services, called web service discovery (WSD). There is no dominating standard for this task, because it involves some issues. For example, a standard way to describe the features provided by a web service should be available in order to determine which web service is available for a particular task. An ontology-based approach to tackle this problem is researched, resulting in various proposals for semanticly-enhanced descriptions of web services which can are machine processable. The DAML-S [15] is a proposal to describe web services using ontologies, based on the DAML+OIL ontology language. A successor of this language is OWL-S[8], which describes web services using the Web Ontology Language (OWL).

The main idea behind the Service Oriented Architecture - or agent-based architectures in general - is that the implementation of the agents is not of relevance for the user of such an agent. This makes it possible to entirely re-implement an agent in another development language, without the user noting this. The flexibility and imteroperability provided by this approach should encourage the re-use of agents and services which will lead eventually to less development time when creating a new application or modifying an existing one. The main task when creating an application in an ideal world would consist of combing serveral services in such a way that the requirements of the client are fulfilled.

For this project we do not convert the application to an agent-based architecture using individual agens, because it would require a significant amount of development. However, some web services are used for some tasks, such as searching on the Internet. The current design of OntoAIMS is composed of individual parts implemented in agent objects. In theory, this is in accordance to the ideas of agent-based architectures. The agents are loosely coupled to a certain extend, for example every agent - such as the domain agent, resource agent, user model agent and search agent - are accessed using a common interface by the client. Only the contents of a message sent to the server are used to decide which agent is used tha handle a request of the client. This makes it possible to change the implementation of existing clients or add additional clients providing new features without needing to change the client or the base packages of the server. In a way, this makes the system comply to the ideas of a agent-based architecture. The communication between the client and the server - or the agent which form the server - is not using standard based protocols such as SOAP or REST.

## 3.2 Overview of Design

The design for the new resource recommender is based on the OntoAIMS information system. OntoAIMS is an *Adaptive Information Management System* which is extended with ontologies. The ontologies in this system are used to model the domains used by the system. The user model available in OntoAIMS, also uses ontologies to represent the knowledge of the user of the system. Both the user model and the ontology-based domain are used for recommending resources in the personalized resource recommender.

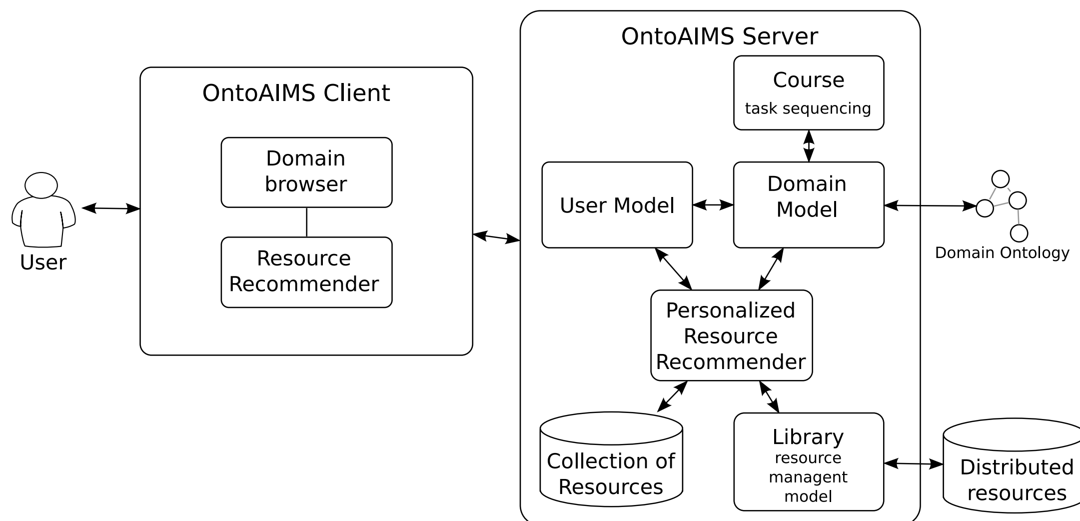


Figure 3.1: Integrated Design Resource Recommender

Figure 3.1 shows the integration in the architecture of OntoAIMS. In this figure, the OntoAIMS Authoring Environment is omitted, because it is not relevant for the resource recommender. As one can see, the personalized resource recommender communicates with several parts in the system. The library in the new design, does not communicate directly with the domain model agent, but is used by the resource recommender. The personalized resource recommender uses a *collection of resources* for recommending resources in a personalized way to the user. The client does not have a notion of the personalized resource recommender. All calculations and processing of the recommendation agent is done on the server.

## 3.3 Searchable Index of Resources

The new resource recommender uses several search technologies. Before choosing a particular search engine, we investigated the available search egines which were potientially suitable for our use.

### 3.3.1 Search Engines

In an initial approach, described in section 2.2, we choose to directly use a search engine provided as a web service. However, this approach turned out to be unsuccesful. As a result, a new approach was to implement a search engine into the system which uses web services when necessary. To get an

overview of the available possibilities with regard to searching, a list of possible search engines and web services providing search engines was created.

### Apache Lucene

The Apache Lucene[1] project consists of several sub-projects which develops full text search engines for various purposes and various programming languages. Currently, implementations of Lucene are available in Java, .Net, C and PHP and other programming languages. The Lucene project started with the Java implementation. Lucene is a generic search engine which, which means that this search engine can be used for a various number of tasks.

Lucene consists of an indexer which has the ability to index plain text files. The indexer filters out irrelevant words. This filtering is language dependent, which means that for every language different filtering rules will be used. Currently filters are availabele for common languages, such as English, German and also Dutch. Lucene is not capable of indexing other document formats, such as HTML, PDF or Word documents. Lucene also does not contain a crawler for indexing documents. Seperate crawlers are available which can be used with Lucene. The omission of a crawler is mainly because Lucene tries to be as generic as possible.

The other part of Lucene is a searcher which can search an index created with the indexer. A complex scoring algorithm makes it easy to order results in a sensible way. The searcher accepts queries in a specific format, however various ways are provided to create these queries. On such way is a parser which parsers strings into queries accepted by the searcher.

The benefits of Lucene is that its implementation is optimized for speed but also provides a decent amount of flexibility. Both indexing and searching documents is fast enough for our purpose.

### Apache Nutch

Nutch[2] is a project based on Lucene. The difference with Lucene is that Nutch is extended with features which enable the program to work better as a search engine for web documents. Features which are missing in a default Lucene-installation, such as a crawler and the possibility to index various kinds of document formats, are included in Nutch. This way Nutch is able to directly function as a search engine for web resources, however, it is necessary to create an index in advance. The creation of this index requires a lot of time, bandwith and storage capacity. These requirements can be limited, when only a small subset of the internet is indexed, but I'm not sure if this is possible for the application we are planning to use the search engine for.

Nutch has the ability to configure the various websites the crawler should index in great detail. The search functionality is identical to the functionality of Lucene.

Since Nutch is based on the Lucene engine, the indexing and searching using Nutch is as fast as Lucene. However, the flexibility is less, since parsers and crawlers are already provided by Nutch, making it more difficult to adapt their behaviour. Furthermore, Nutch is mainly intended as a web based search engine, used via a web page. This does not really fits our requirements.

[1] See http://lucene.apache.org

[2] See http://lucene.apache.org/nutch/

#### Egothor

Another Java-based search engine is Egothor[3]. This engine is developed, like Lucene, as an open-source project. When we had to choose the engine used for the resource recommender, the latest version 2.0 of Egothor, was not yet freely available. Of the older versions, there is limited documentation in comparisation with Lucene. The final version of Egothor 2.0 is currently freely available, however there is still limited documentation available on this project. The search engine is optimized for speed, and according to benchmarks created by the developers of Egothor, the indexing speed is better than Lucene[4]. Egothor contains a search engine and indexer which supports various document formats, such as HTML, PDF and Word documents. A crawler is also provided. Due to the limited documentation Egothor was not selected for the resource recommender.

#### Internet Search Engines as Web Service

Lucene, Nutch and Egothor are search engines which could be integrated in the application and have to create an index on the system before it is possible to search. On the other hand, there are web services which provide programmatic access to an Internet search engine, such as Google or Yahoo. Some of these web services can be used freely, such as Google and Yahoo. Others require payment for their usage, such as Alexa Web Search[5]. Free web services providing access to an Internet search engine limit the usage to a certain number of queries and only return a limited number of results.

Google SOAP Search API is discussed in detail in section 4.4.2 and Yahoo WebSearch API is discussed in section 4.4.2, including the limitations and the implementation details. As most web services, the Internet search engine webservices use standard protocols such as SOAP, REST and WSDL. The advantage of using web services providing search features is that it is not required to create an index before the search can be used, because the index is already available. Another advantage of this, is that these indexes contains much more documents than the indexes created with Lucene, Nutch or Egothor. While it is possible to create very large indexes with these latter search engines, they are not optimized for very large indexes (containing several million documents) and the hardware requirements, such as storage space and processing power, restrict the size of the indexes. These drawbacks do not apply on the search engines such as Yahoo, Google and Alexa. Because of this, we choose to combine a search engine such as Lucene with web services for discovering new resources.

### 3.3.2 Resource Collections on the System

Before the system is able to recommend resources, a collection of resources is required. This collection of resources should be available on the system. In the design of the personalized resource recommender, three collections of resources can be distinguished:

- **Resources available on the Internet:** $\mathcal{R}_I$ Most resources added by the course creator to the system will be located on the Internet. However, considering the fact that there are millions of websites on the Internet, there will be many more resources available. The selection of the resources the course creator has made, will be revelant for the course and of sufficient quality. The other resources on the Internet can be relevant for a course, but this is not for sure. Also the quality of the resources it not guaranteed. This collection is accessible using various methods. Resources can be addressed directly or found using search services.

[3] The Egothor project page is located at http://www.egothor.org
[4] See http://www.egothor.org/ galambos/twiki/bin/view/Egothor/EngineBenchmark for these benchmarks
[5] See http://www.amazon.com/gp/browse.html?node=269962011 for the details on Alexa

- **Resources collected by a course creator:** $\mathcal{RL}$ A course creator creates a course in the adaptive information system. Students doing the course needs resources to study the subjects in the course. These resources, which can be documents, images, video's and sound files, are collected by the author of the course and stored in the system. These resources can also be added to the system by the domain author, but in many cases the course creator and the domain creator will be the same person. This collection of resources is usually a subset of $\mathcal{RI}$. In most cases, this collection contains references to the resources which are part of this collection, instead of the actual resource itself.

- **Resources available to the system:** $\mathcal{RS}$ This is a collection of resources the system has notion of. This collection typically contains all the resources selected by the course creator $\mathcal{RL}$. It is possible that a subset of the resources on the Internet, collection $\mathcal{RI}$ is part of this collection. The system will have detailed information on these resources and is able to use this information to recommend these resources to the user. This collection is indexed using a search engine, making it possible to search this collection in an easy way.

While it is posible to use the three collections when the actual recommendation of resources is done, this will lead to an unsatisfying result, because the three collections differ in the way the resources are stored and do not have sufficient metadata available for the resources. The goal is to provide a personalized resource recommendation using the experience and knowledge level of the user. This requires as much metadata as possible on the resources, in order to decide if a resource is relevant for the user. Furthermore, one of the research questions is to automatically recommend resources from the Internet, which requires that resources from the Internet are available for the resource recommender.

A more promising approach is only selecting resources which are part of the collection $\mathcal{RS}$, because the system has detailed information on these resources, including metadata and concepts to which these resources are relevant. This information can be used to personalize the recommendation of the resources.

The resources of the first two collections, the resources collected by a course creator $\mathcal{RL}$ and a relevant part of the resources available on the Internet $\mathcal{RI}$, have to become a part of the resources available to the system $\mathcal{RS}$. This merging of various resource collections into a single resource repository is on of the problems defined in section 1.1.2. Considering the fact that the resources collected by the course creator are typically of high quality and are relevant to the concepts in the domain, all the resources in this collection $\mathcal{RL}$ should be added to the resources available to the system $\mathcal{RS}$. This makes it easier to map the ontology to the resources, which in its turn can be used to personalize the recommendation process. As a result, this makes the process of collecting resources by a course creator or domain author easier, because it it nos required to provide a resource with metadata.

The most difficult part is to get a subset of Internet resources $\mathcal{RI}$ and add these to the resources known by the system $\mathcal{RS}$. The number of resources on the Internet is very large, and a decent strategy needs to be developed in order to get a subsection which contain resources that are relevant to a concept in the domain for which the system wants to recommend resources.

$\mathcal{RS}$ is stored on the system in such a way that it is easy to query this collection. Additional available information on a resource should be stored with the resource on which this information applies. Resources in $\mathcal{RS}$ are related to a domain used by the system. Every domain in the system has a subset of resources $\mathcal{RS}$ that are related to concepts in that domain.

### 3.3.3 Storing documents on the System

The process of storing documents on the system in the resources collection $\mathcal{RS}$ consists of various steps. Figure 3.2 gives a visual representation of this process. It is required to store the resources on the system to annotate the resource using the ontology information, as is defined in the third research questions in section 1.2.

First, the resource needs to be extracted from its location. Second, the content and the metadata of the resource should be extracted or collected. The extraction process is done by a *parser*, of which a part is specific for each the document type of the resource. The metadata can be provided by an external resources. Third, the document should be related to concepts in the domain to which this document belongs. Finally, the resource, combined with its metadata and links to related concepts, is stored on the collection $\mathcal{RS}$. The collected metadata and related concepts are used to automatically recommend resources from the Internet.

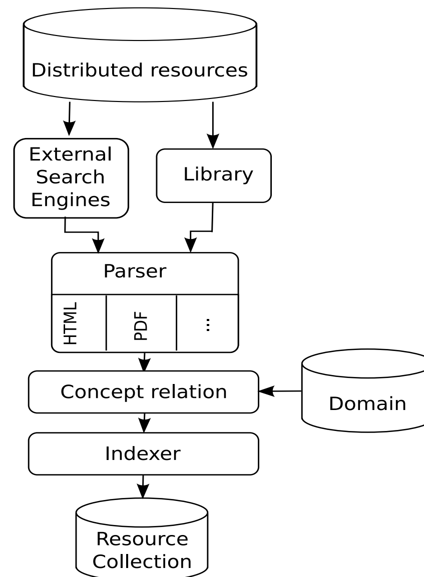


Figure 3.2: Process of storing documents in resource collection.

### 3.3.4 Parsers

The parser processes the documents that are located on the Internet before they are stored in the resource collection $\mathcal{RS}$ on the system. The parser consists of a generic part, that provides functionality that is required for all document types, and a part that is specific for a document type. This latter part of the parser performs the actual extraction of the content of the document and all the available metadata.

The documents acquired from the Internet are often formatted using a markup language, a good example of such a language is HTML. In order to process the document in a sensible way, it is required to strip all the markup tags. The various parts of a document, such as a header containing metadata, the title of the document, links in the document and important sections of the document, have to be seperated in order to ease the processing of these sections. In the resource recommender, this task is also done by the parser.

The parser gives a number of fields as result after processing a document:

- **URL:** The url of the resource.
- **Title:** The title of the resource. This is usually specified in the metadata of a document.
- **Summary:** A summary of the resource.
- **Author:** The name or names of the authors of the document.
- **Creation date:** The date the resource was created.
- **Modification date:** Date specifiying the moment the last change to the resource was made.

- **Keywords:** A list of keywords indicating the subject of the document.
- **Links in document:** A list of links in the resource to other resources. Both the URI of the links and the anchor text are available.
- **Content of document:** A textual representation of the contents of a resource without formatting information.
- **List of important sections:** This section of a resource contains a list of important section, such as headers, text in a bold typeface and emphasized text.
- **List of related concepts:** This contains of a list of concepts that are related to the resource. This list is not created by the parser, but is stored in the same way as the other fields. See section 3.3.5 for a detailed description of this field.

These fields are stored and indexed, making it possible to search the collection. The fields are stored in such a way that it is possible to search only on particular fields in the collection. For example, it is possible to search for documents which have a specific word in the title of the document or to search for documents that were created after a specific date.

### 3.3.5 Relating Concepts to Resources

Resource recommendation is done based on the concepts of the domain. The system should have a notion of the resources in $\mathcal{RS}$ to which concepts in the domain they are relevant. This information can be used in two ways: first, to get relevant concepts for a certain information resource and second, to get relevant resources for a particular concept in the domain.

Because one of our goals is to reduce the effort needed to add resources to the system, the system should automatically find relevant concepts for a given resource. This process, called concept relating, is done when the system adds a resource to the collection of resources $\mathcal{RS}$. The related concepts are also used to automatically recommend resources from the Internet and try to tackle the research problem of automatically annotating resources with ontology information.

This process can be done using *information extraction* technologies. A lot of research is done on this subject and there are libraries available that contain technologies for information extraction, such as GATE [6]. However, these libraries are quite complex and difficult to integrate in an existing application. We choose to design our own method of relating concepts to a document using existing search technologies. Because the OntoAIMS system has already an ontology with concepts available, this ontology can be used to simplify the process of relating concepts to a resource.

#### Concept Extraction

Inspired by technologies used for relating advertisments to documents on the Internet [25], which is a similar process, we designed a fast and easy method to relate concepts to a resource. The contents and metadata of a resource are extracted and seperated into several parts, resulting in *document sections*. For every document section, the system decides if a name of a concept (partially) occurs in that section. Based on the number of occurences in a section and the similarity of these occurrences to the concept name in a document section, the system calculates a score for every concept in the domain. Documents that score above a predefined value are considered relevant. When more than ten resources are relevant, the system only relates the ten most relevant concepts to the document.

The sections, often also called *fields* which are used for relating concepts to a document are the following:

- **Title of document.** The title of the document including a subtitle, if it is available. Most document types have a seperate field defining the title, making extraction of the title a straight forward process.
- **Content of document.** The entire content of the document. This section contains the content of a document visible to the user. No metadata is contained in this section.
- **Keywords of document.** This field contains keywords related to the documents. Many document formats have a *keyword* field in the meta data, specifying relevant keywords for the document. For example, in HTML documents, relevant keywords can be defined using the `meta` tag in the header section. These keywords are also used by web search engines. Keywords can also come from external sources. For example, in documents which are selected by a course creator and are stored in the library, the author specifies keywords for that document.
- **Anchor text of links.** In document formats that are using the hypertext feature of linking to other documents, the anchor text of the links are relevant for determining the concepts the document is related to. Many document formats support links: HTML, PDF, Word documents, OpenDocument, etc.
- **Titles in document and specially formatted words.** Many document formats have special formatting features for important words and sentences. Chapter titles, section titles, emphasized words and bold words are some examples of this kind of words or sentences. These parts of the document are very useful to decide if a concept is related to the document.

**Concept Definition** To describe the process of calculating the score for the relevance of a concept to a certain resource, we first describe how a concept is defined in OntoAIMS. A concept $\mathcal{C}$ is defined as triple:

$$\mathcal{C} = (\mathcal{N}, \mathcal{O}, \mathcal{S})$$

In this definition, $\mathcal{N}$ is the name of the concept consisting of one or more words used to refer to a concept; $\mathcal{O}$ is a set of *otherforms* which are alternatives to the name of the concept and can be considered as alternative names; $\mathcal{S}$ is a set of *synonyms* which are names of other concepts which have the same meaning as this concept.

The name $\mathcal{N}$ of the concepts and all the alternatives in $\mathcal{O}$ are matched against all the fields of a document. If $n$ is a name of the concept $c$, in which case $n \in \mathcal{N}$ or $n \in \mathcal{O}$ is true, a score is calculated for the relevance of the resource for concept $c$ if $n$ is found in a part of a resource. This score depends on several factors:

- The number of occurrences of a the name $n$. If a name $n$ is found several times in a field, the score is increased.
- The length of the field in which the name $n$ is found. A title of a document is usually shorter than the contents of a document. An occurence of a name of a concept in a title is less likely than in the content of a document. Furthermore, a title generally gives a very rough description of the content of a resource. When a name is found in the title, the score is higher than when the concept name is found in the contents of a document.

- If a name $n$ is found in more than one field, the score of every field is added to the total score.
- When elements of of $\mathcal{N}$ and one or more of $\mathcal{O}$, or when multiple names in $\mathcal{O}$ are matched in a field, the score is higher compared to when only $\mathcal{N}$ or one of the alternatives in $\mathcal{O}$ is matched. So, if both $n$ and $m$ occur in the resource, and $n \in \mathcal{N}$ and $m \in \mathcal{O}$, then the score is higher than when $n = m$ and $n \in \mathcal{N}$.

The score for every $n \in \mathcal{N} \cup \mathcal{O}$ for every field of a resource, which is a real value between 0 and 1, is added resulting in a total score for a concept. The ten concepts with the highest score which is at least higher than a predefined value are related to the resource.

## 3.4 Resource Recommender

This section describes the resource recommender in OntoAIMS. The *resource recommender* is in fact a search engine for which queries are created by the system. This process is transparent to the user and implemented on the server. The *Search Agent* on the server of OntoAIMS is the main component of the resource recommender. When the user requests a list of relevant resources for a certain concept or search query, the client sends a request to the server. The *Search Agent* on the server takes this request and creates a list of recommended resources for this request. When the server has created a list of resources, using the information in the request from the client, the concepts in the domain and the knowledge of the user stored in the user model, the result is sent to the client and presented to the user. Figure 3.3 gives an overview of the processing of search requests.

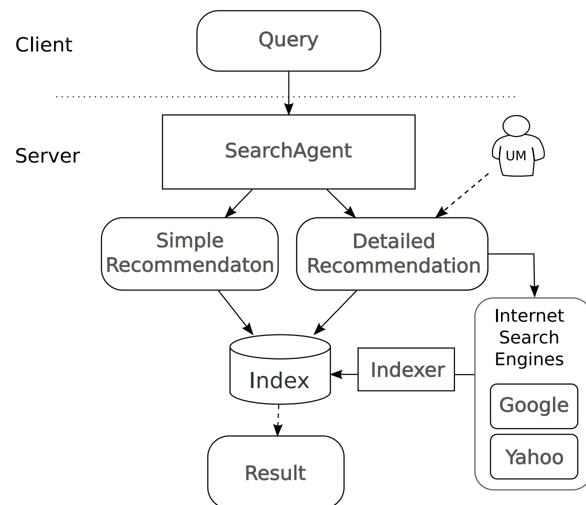


Figure 3.3: Processing of recommendation requests.

The resource recommender implements a way of recommending resources from the Internet and personalize the resource recommender. Additionally, it provides a general interface to all available resource collections available to the system.

### 3.4.1 Simple and Detailed Recommendation

The resource recommender uses two different *recommender requests*. These two request types are:

- **Simple recommendation.** This recommender agent will return a limited number of resources in case of a simple recommendation. Simple recommendation is done when a user does not explicitly request a recommendation. Because of this, the simple recommendation should not let the user wait and therefore has to quickly return results. A simple recommendation does not consider the user model and only return resources that are related to the concept currently studied by the user.
- **Detailed recommendation.** A detailed recommendation can be requested by the user when he or she thinks the resources returned by the simple recommendation are not sufficient. A

detailed recommendation will return a larger number of resources compared to a simple recommendation. The detailed recommendation uses query expansion techniques and uses all the available information stored with the resources collection in the system. The result of a detailed recommendation is a personalized list of recommended resources on the concept the user is studying.

The distinction between thos recommendation types are made because one of the defined problems is to recommend additioal resources when the available resources are insufficient. The detailed recommendation provides more resources when the user has more knowledge on a certain concepts, which is a way to implement personalization in the resource recommender. This tackles the problem of different knowledge levels of the users by adapting the resources to the information available in the user model.

### 3.4.2 Simple Recommendation

A simple recommendation request is handled quickly by the resource recommender. A simple recommendation request supplies the name of the concept the user is studying. The collection of resources stored on the system, defined as $\mathcal{RS}$, is searched. All resources which are related to the concept supplied in the recommendation request, are returned as recommended resources.

### 3.4.3 Detailed Recommendation

Processing *detailed recommendation request* is more complex than a simple recommendation request. A detailed recommendation request consists of the name of the concept the user is studying along with optional additional *query terms* that can be added to the request by the user. In some situations, for example when the user is not actively browsing the domain using the visualization agent, it is possible that there is no concept supplied in the recommendation request, and only query terms given by the user are supplied.

The searchable collection of resources stored on the system is used for recommending resources in a personalized way. As is described in section 3.3, this collection contains both resources from the library and from the Internet. Before the resources are stored in the collection of resources in the system, they need to be found on the Internet. For this task, we use existing search engines. It is also possible to use other web services providing access to collections of links. For example, social bookmarking sites offers a selection of links to Internet sites and provides programming interfaces for accessing these links.

The process of resource recommendation uses the searchable collection of resources, defined as in the system $\mathcal{RS}$ and the documents available on the Internet, defined in section 3.3 as $\mathcal{RI}$. First, the system searches the local collection $\mathcal{RS}$. Depending on the first result, the system optionally searches the Internet. If $\mathcal{RS}$ already returnes enough interesting results for the user, the system will not search $\mathcal{RI}$. The resources found in $\mathcal{RI}$ are stored in the local collection of resources $\mathcal{RS}$, using the method described in section 3.3.3.

The queries used for searching $\mathcal{RS}$ and $\mathcal{RI}$ are refined using the *concepts in the domain* and the *user model*. This section describes the process of refining the queries. Because the queries for the local collection of resources and the Internet search engines are refined in a similar way, the process will be described only once in this section. When necessary, it will be indicated if there is a difference in the refinement of the queries for the Internet or local collection.

**External Search Engines** Many search engines provide an API making it possible to use the search engine in applications. However, in order to get good results, it is necessary to use queries which result in relevant resources and decrease the number of *false positives*, i.e. websites that are part of the result, but are not relevant. Using the domain ontology and the user model, it is possible to expand the queries with additional *search terms* and *operators*.

**Resource Collection on System** The collection of resources in the adaptive information system, described in section 3.3, is searchable using queries. Because this collection is integrated in the system, the queries for this collection can be more detailed than queries for external search engines. However, the basics for expanding queries for this collection are similar to the query refinement for the external search engines.

### 3.4.4 Query Processing

A detailed recommendation request consists of the *name of a concept* and an optional query string. This information is combined with information existing in the domain and the user model. The following inputs are available for the query processing:

#### Inputs for Query Processing

- Inputs from recommendation request.
  - Main concept $c_m$ of type $\mathcal{C}$. This is the concept the user is studying. The concept consists of a conceptname $\mathcal{N}$, alternative names $\mathcal{O}$ and synonyms $\mathcal{S}$. See section 3.3.5 for a detailed description of the definition of concepts.
  - Query string $\mathcal{T}$. This consists of one or more query terms $t$.
- The *Domain model* $\mathcal{DM}$.
- The *User Model* $\mathcal{UM}$.

These three main inputs are used to create queries.

**The Domain Model** The domain model is defined as:

$$\mathcal{DM} = (\mathcal{P}, \mathcal{Q})$$

Where $\mathcal{P}$ is a set of concepts of type $\mathcal{C}$ and $\mathcal{Q}$ is a set of links between those concepts, defined as $\mathcal{L}$:

$$\mathcal{L} = (p_1, t, p_2)$$

Where: $p_1, p_2 \in \mathcal{P}$ and $t$ defines the type of the link. In other words, $\mathcal{L}$ defines a link of type $t$ between concept $p_1$ and concept $p_2$.

**The User Model** As described in section 2.1.2, there are two different conceptual state representations used in OntoAIMS. For the query refinement, the system only uses the *long term conceptual state*. This representation of the user's conceptualization of the domain is an overlay on top of the domain model which annotates every concept and link between two concepts with an *belief value*:

$$\mathcal{UM} = (\mathcal{PA}, \mathcal{QA})$$

Where $\mathcal{PA}$ is a set of annotated concepts of type $\mathcal{CA}$ and $\mathcal{QA}$ is defined as an annotated set of links, defined as $\mathcal{LA}$. Each annotated concept and link is defined as:

$$\mathcal{CA} = (c, v)$$
$$\mathcal{LA} = (l, v)$$

In these definitions, the variable $v$ is the belief value, which is an integer value beween 0 and 100. The variable $c$ is a concept in the domain, defined as $c \in \mathcal{P}$ and $l$ is a link between two concepts, defined as $l \in \mathcal{Q}$.

### Query Refinement

Based on the intputs defined in the previous section, a query for a resource recommendation can be refined. In the description of this process, the variable names defined in the previous section are used.

This refinement process consists of several steps. The result will be a query $Q$.

**Step 1: Define Main Concept** Some detailed recommendation requests do not specify a concept, but only a list of query terms $\mathcal{T}$ The first step required is to try to find a concept which is described by the query $\mathcal{T}$:

1. For all concepts in domain, if there exists a $c \in \mathcal{P}$ for which $c = \mathcal{T}$, set this concept $c$ as the main concept $c_m$.

2. For all concepts in domain, if there exists a $c \in \mathcal{P}$ for which there exists a term $t \in \mathcal{T}$ for which $c = t$ holds, set this as the main concept $c_m$.

3. For all terms $t$ in the query $\mathcal{T}$, check if there exists a concept $c$ in the set of concepts in the domain $\mathcal{P}$, for which holds $t \subset c$. If there is such a concept, then this concept $c$ is the main concept $c_m$.

The name of the main concept $c_m$ is added to the query:

$$\{Let\ n_{cm}\ be\ the\ name\ of\ concept\ c_m\}$$
$$Q' = Q \wedge n_{cm}$$

**Step 2: Expand Query with Alternatives and Synonyms** Most concepts in the domain have alternative names. These names are often used in documents relevant for a particular concept. Therefore, it is necessary to expand the query with the alternative names for the query:

$$\begin{aligned}
&\ldots \\
&Q' = Q \wedge n_{cm} \\
&\{\textit{Expand query with alternative names; let } o_1 \ldots o_n \textit{ the terms in } \mathcal{O} \textit{ for } c_m\} \\
&Q' = Q \wedge (n_{cm} \vee o_1 \vee o_2 \vee \ldots \vee o_n)
\end{aligned}$$

**Step 3: Add Additional Concepts** Depending on the knowledge of the user on the main concept $c_m$ and the knowledge of the user on the concepts that are linked to $c_m$, the query is expanded with additional terms.

When the belief value $v_{cm}$ on the concept $c_m$ is less then 50, no additional concepts are added, because probably the user wants to learn more on this concepts and is not (yet) interested in related concepts. If the $v_{cm}$ is larger than 50, additional concepts are added to the query $Q$. For every concept $b$ that is linked to concept $c_m$ by a link in $\mathcal{Q}$, the belief value is read from the user model. If the belief value for concept $b$ is less then 50, the user probably wants to learn more on this concept in relation to concept $c_m$ and this concept $b$ is added to the query. In our example, we consider the belief value for $b$ smaller than 50 and as such included in the query.

$$\begin{aligned}
&\ldots \\
&Q' = Q \wedge (n_{cm} \vee o_1 \vee o_2 \vee \ldots \vee o_n) \\
&\{\textit{Expand query with linked concept names; let } b_1 \ldots b_n \textit{ linked to } c_n \textit{ in } \mathcal{Q}\} \\
&\{\textit{For every belief value } v \textit{ for } b_1 \ldots b_n \textit{ holds } v < 50\} \\
&Q'' = Q' \wedge b_1 \wedge b_2 \ldots \wedge b_n
\end{aligned}$$

**Step 4: Add Alternative Names for Additional Concepts** For the concepts added in 3.4.4, are also alternative names defined in the domain. These alternative names should also be added to the query using the same process used for the main concept $c_m$.

$$\begin{aligned}
&\ldots \\
&Q'' = Q' \wedge b_1 \wedge b_2 \ldots \wedge b_n \\
&\{\textit{For every concept } c_a \textit{ in } b_1 \ldots b_n \textit{ add alternative names in } \mathcal{O}\} \\
&\{\textit{Let } o_{1b1} \ldots o_{2b1} \textit{ the alternative names for } b_1\} \\
&Q'' = Q' \wedge (b_1 \vee o_{1b1} \vee o_{2b1} \ldots) \wedge (b_2 \vee o_{2b1} \vee o_{2b2} \ldots) \ldots
\end{aligned}$$

After the four steps, query $Q''$ will contain, depending on the user model, the query string provided by the user and the name of the concept provided by the user or the name of the concept extracted from the query string provided by the user. Additionally, the alternatives name for the main concept are included and the concepts that are directly linked to this concept in the domain. The alternative names of the linked concepts are included in the query as well.

This query refinement process is used for queries that are used for searching resources in the local resource collection on the system, defined by $\mathcal{RS}$. The same query refinement process is also used for creating queries for Internet search engines in order to collect additional resources. After these resources are collected from the Internet, they are added to the local collection of resources.

The result which is send to the client, is the result of a final query executed on the local collection of resources $\mathcal{RS}$.

# Chapter 4

# Architecture and Implementation

This chapter presents a detailed description of the implementation of the personalized resource recommendation in OntoAIMS. The implementation of the resource recommender is based on the design described in chapter 3. The first section of this chapter describes the implementation context and the consequences for the implementation of the resource recommender. The system is implemented in the OntoAIMS information system, which is the main subject in that section. The subsequent sections after describe the implementation of the resource recommender and the used technologies and external libraries. The source code documentation of the implementation can be found in Appendix A.

## 4.1 Implementation Context

In section 3.2 we already described the design of OntoAIMS. The OntoAIMS system is used by several researchers and students as a platform for testing various technologies and ideas. As a result the OntoAIMS system consists of several parts and uses a wide variety of technologies. In section 3 we presented an improved design of OntoAIMS by adding an improved personalized resource recommender. The implementation of this design and the difficulties encountered when implementing this, are described in this chapter.

As already mentioned in section 3.2, OntoAIMS is implemented using a client-server architecture. The server of the system includes the various data models, including the domain model, user model, task sequence model and resource library, and performs most calculations and data processing. The client is a simple interface to the user which communicates heavily to the server. The user can browse the visual representation of the domain and search information resources available on the system. The server of OntoAIMS, the part on which most of the development occured during this project, is implemented in *Java*[1]. The OntoAIMS server is running on the *Apache Tomcat*[2] application server. The server communicates with the outside world using a *Servlet*. The client is web-based and can be used using an standard internet browser that supports Java Applets. The client of OntoAIMS is also implemented in Java using Java Applets.

### 4.1.1 OntoAIMS Packages

The most important packages of OntoAIMS are showed in figure 4.1. Not all packages of OntoAIMS are visible in this figure, because the diagram would become too complex. The `Base` package contains

[1]See http://java.sun.com for more information.
[2]See http://tomcat.apache.org for more information.

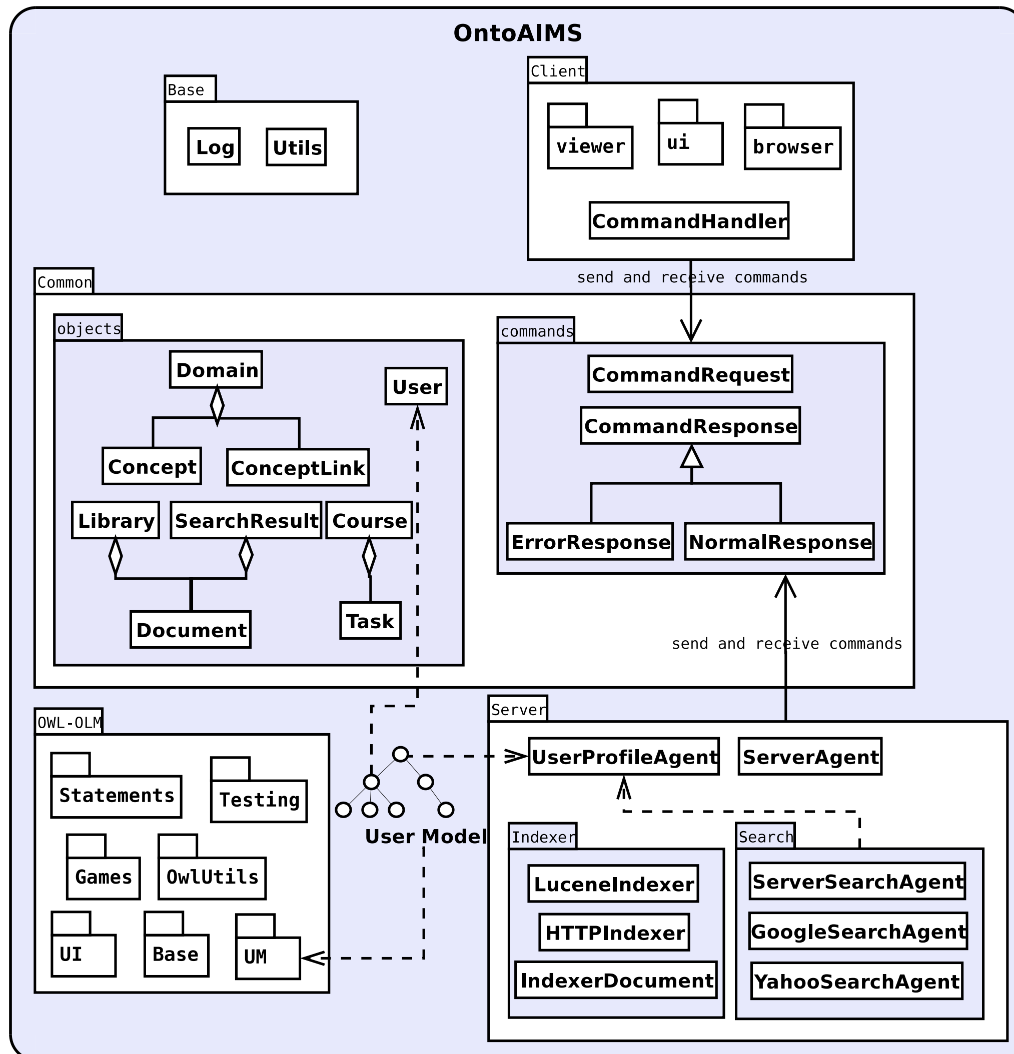


Figure 4.1: Main packages of OntoAIMS

general classes providing basic utility methods for logging, input and output and configuration. The `Common` package contains classes that are used by both the client and the server. The objects defined in this package are primitive structures with only fields containing the relevant information. The `Client` package contains the OntoAIMS client and provides the user interface for the system. The client communicates with the server using the primitive objects in the `Common` package. The `Server` package contains the implementation of the server for OntoAIMS. A Java Servlet provides the web service that can be used by the client to communicat with the server. The server reads and write the various information models, such as the domain model, user model, task sequencing model and library. The `OWL-OLM` package contains the implementation of the OWL-OLM system and the Dialog Agent, described in section 2.1.3. OWL-OLM is implemented in a seperate package, making it easier to use it in other systems.

The personalized resource recommender is integrated into the `Server` package. The implementation uses many aspects of the existing system to refine search requests and extract information of the

information models. Implementing the resource recommender in a standalone package which is not integrated in the server would require major refactoring of the OntoAIMS server. Interfaces to internal parts of the server should have to be created in order to get the required information for the resource recommender, making the implementation a lot more time consuming. The personalized resource recommender is integrated in the server in two subpackages. The `indexer` package is the implementation of the indexer of the resources for storing them in the resource collection on the server. This package provides tools for parsing various document formats and formatting the result in such a way that it can easily be searched. The `Search` package implements the actual search part, which is in fact the resource recommender. This package searches the local collection of resources and the Internet using *Google*[3] and *Yahoo*[4].

## 4.2 Indexer Package

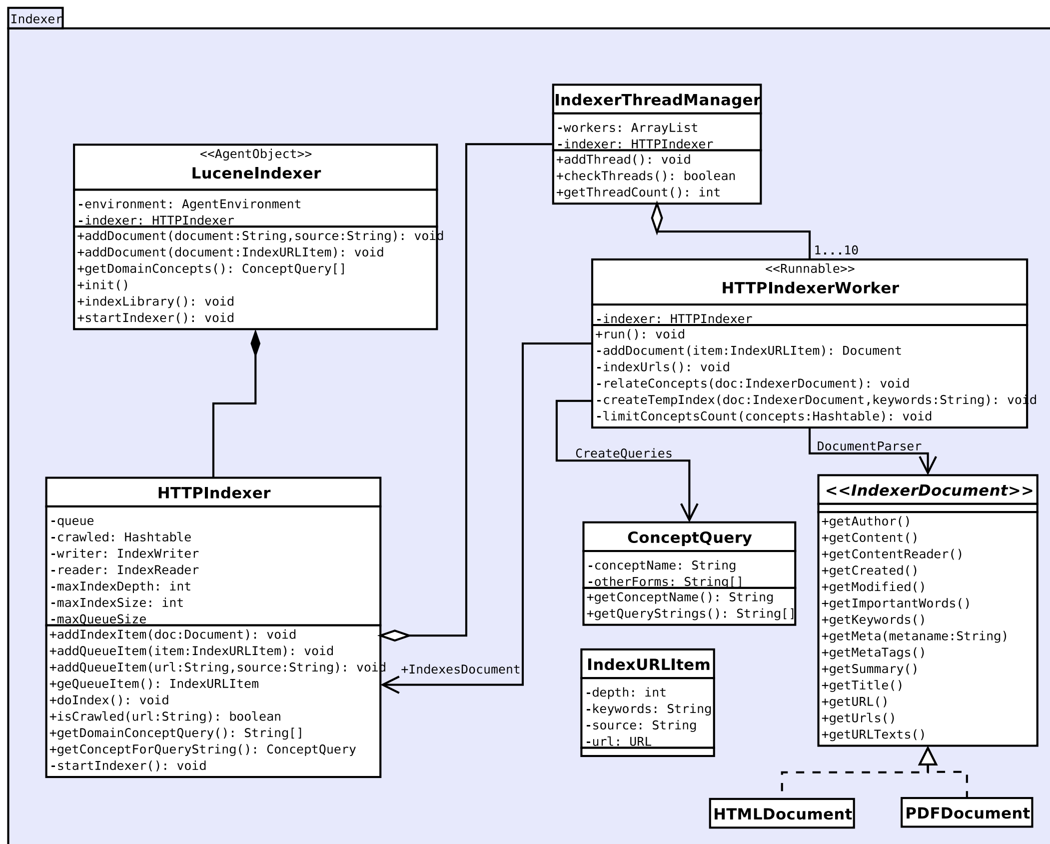


Figure 4.2: OntoAIMS Indexer package.

The `Indexer` packages contains the implementation of the indexer in OntoAIMS. The implementation of the local resource collection, refered to in section 3.3, consists of a searchable index. This index contains the extracted content of the resources and the metadata available for the resources.

[3]See http://www.google.com.
[4]See http://www.yahoo.com.

Figure 4.2 shows the global implementation of the index. The individual classes of this package are described in this section.

We choose to use an existing library for creating the index and to search this index. For this purpose we choose to use the *Apache Lucene*[5] library. The main interface between the indexer and the other parts of OntoAIMS is implemented in the `LuceneIndexer` class.

### 4.2.1 OntoAIMS and Lucene

The `Indexer` package uses the Apache Lucene library for implementing the indexer. The Lucene library is also used for searching this index. Lucene is a text search engine entirely implemented in Java. Lucene provides several methods for creating an index and the way the index is stored. We choose for this project to use the standard indexer which stores the indexing data in plain text files on the filesystem. It is also possible to store the index in an relational database engine or in the internal memory. We used this latter indexer for temporarly indexing a document when extracing the related concepts. See section 3.3.5 for a detailed description of the concept extraction.

Using Lucene, it is possible to index documents using an arbitrary number of *fields*. Each field can contain text of any length or a date. For each individual field, it is possible to indicate if it should be indexed, making it possible to search for documents using the information in this field. Additionaly, it is possble to *tokenize* a field using a so-called analyzer. If a field is tokenized, it is parsed into separate parts, where every part usually consists of one word. For example, the content of the document (the body text) is tokenized and indexed using the standard analyzer provided by Lucene, making it possible to search for individual words or groups of words in the index. This standard analyzer provided by Lucene parses the text assuming the used language is English. On the other hand, the URL of a document is not indexed and not tokenized, because for our usage of the index it is not necessary to search on the URL of a document.

Searching the documents in the index is implemented in Lucene using a combination of the *Vector Space Model* [24] and the *Boolean Model*. The search results can be ranked, where the best result for a particular query is returned first. Lucene supports a wide range of query possibilities, including queries with wildcards, boolean operators or indicating ranges. A standard implemented `Query Parser` implements a parser for text queries. A *query filter* makes it possible to search within search results or limit a search query to only a specific part of the index. This query filter is used in OntoAIMS for limiting the results of a *simple recommendation request* to the resources selected by the author of the course or domain.

### 4.2.2 `LuceneIndexer`

The `LuceneIndexer` is the main interface to the indexer and the only class used for communicating with the implementation of the indexer. The `LuceneIndexer` class implements the `AgentObject` interace, and as such can be used as an agent within OntoAIMS. When OntoAIMS is started, the application creates an instance of the object which implements the servlet called `AgentServlet`. The `AgentServlet` class creates an `AgentServer` object, which controls the flow of the application. The `AgentServer` object handles the incoming messages from the client and delegates the handling of these messages to the appropriate parts of the application. For this purpose another object is created from the `MetaAgent` class. This `MetaAgent` class handles most of the incoming requests on the various information models and the search agent. The `AgentServer` object creates an instance of the `LuceneIndexer` class. On the creation of a `LuceneIndexer` object, the `MetaAgent` calls the

[5] See http://lucene.apache.org

indexLibrary method, which checks if the library created by the author of a course of domain is already indexed. This check opens the library and verifies whether all the resources in the library can be openened. When this is the case, the indexer checks if the resource is contained in the index and when this is not the case, the resource is added to the index.

One instance of the LuceneIndexer class is used for managing the index of resources related to a single domain. The index is stored on disk in a directory specified in the agentServletConfig.txt file which hold configuration data for the OntoAIMS server. The directory which holds the Lucene indexes is specified using the configuration key luceneindexdir. For every domain, a seperate directory is created within this luceneindexdir in order to keep the indexed resources for the various domains seperate.

Furthermore, the LuceneIndexer class provides method for adding documents to the index. These methods, called addDocument() are in fact *delegate methods* and are implemented in objects aggregated by LuceneIndexer. The startIndexer() methods starts the indexing process and should be called when documents are added to the index using the addDocument() methods. Finally, the class provids the method getDomainConcepts() resulting an array of ConceptQuery objects. This method is used for the process of relating concepts to resources added to the index. See section 3.3.5 for a description of this process.

The HTTPIndexer class is the actual implementation of the indexer. This class maintains a *queue* of documents that should be indexed. Indexing documents is a rather expensive tasks, meaning that is time consuming. Therefore, the HTTPIndexer uses worker threads to parallellize the indexing of multiple documents. The worker threads are instances of the HTTPIndexerWorker class. The creation and management of the individual threads is manged by the IndexerThreadManager. The worker threads are implemented the standard threading methods that provided by Java. Every thread created by the IndexerThreadManager fetches resources that have yet to be indexed from the queue implemented in the HTTPIndexer. The resource is downloaded from the internet, parsed, the concepts are extracted and finally added to the index.

### 4.2.3 HTTPIndexer

The HTTPIndexer implements full-featured indexer, with the possibility to recursively index web pages on the Internet. The indexer extracts all links on a page and can index the pages referred to. This way, it is theoretically possible to index a very large number of pages provided that the initial list of web sites is sufficient. Because the usage of the parser in OntoAIMS is limited to index only single pages, it is possible to limit the number of links that should be followed. This is done by setting the maximum *depth* at which links should be indexed using the setMaxIndexDepth() method. By default, this setting is 0, meaning that links on a web page are not followed when indexing the page.

A *first in, first out queue* with resources that have to be indexed is maintained by HTTPIndexer. URL's of resources in the queue are not yet contained in the local collection of resources $\mathcal{RS}$. URL's of documents can be added to the queue using the addQueueItem() methods. Before a document is added to the queue, these methods checks if the document is already in $\mathcal{RS}$ by searching on the URL of the document. In order to improve performance, the indexer maintains a *hashtable* named crawled with documents URL's that are already contained in the collection of local resources or that are not available. When a document is indexed and the HTTPIndexer is set to follow links in the document, the URL's in that document are added to the queue of items that have yet to be indexed. Usually, when indexing multiple web pages located on a single *top level domain*, they have often a set of identical URL's. The previously mentioned *hashtable* crawled improves the speed of adding documents to the queue, because not for every URL the indexer has to check if the URL is contained in $\mathcal{RS}$.

A potential problem when indexing a very large number of web pages including the referenced pages, is that the queue maintained by `HTTPIndexer` can become very large. Especially, if the pages contains a large number of unique links, the queue will quickly be filled. In order to cope with the consequences of this behaviour, it is possible to limit the size of the queue. As a result, not every link will be indexed, but it will solve a potential crash of the application because of unsufficient available memory. In OntoAIMS, the links are usually not indexed, or at most one level deep. For the workings of the indexer in OntoAIMS this potential problem has no effects, but for other usages of the indexer this can become a problem. A solution to this problem is to store the URL's that have to be indexed in a file on the filesystem instead of in a queue.

`HTTPIndexer` has several methods used for configure the exact indexing process, such as the maximum size of the queue, the depth at which URL's should be indexed and the maximum number of threads used during the indexing process. The most important methods in this class, used for adding parsed documents to the index and adding and retrieving items from the queue with resources that are not yet indexed. These functions are listed below:

- `addQueueItem()` adds an item to the internal queue of URL's to resources that are not yet parsed and stored in the local collection of resources $\mathcal{RS}$. This method is mainly used by the worker threads and is therfore *synchronized*. This synchronization ensures that two separate threads cannot call this method at the same time, causing possible concurrency problems. There are two variations of this method, the first accepting an instance of the `IndexURLItem` class and the second accepts a *String* containing the URL and another *String* containing the *source* of the URL.

- `getQueueItem()` returns the top item in the internal queue with resources URL's. This method is also *syncrhonized* to prevent two threads get the same URL for indexing. An instance `IndexURLItem` is returned by this method, containing the source of the URL (either library, Google, Yahoo or something else), available keywords and the *depth* at which the link was collected.

- `addIndexItem()` is a synchronized method that stores a parsed document in the Lucene index. This method is used by the `HTTPIndexerWorker` threads. After a URL is downloaded and parsed, it can be stored in the index using this method. The method accepts a `Document` object. The `Document` class is part of the Lucene library. The parsed document is written to the index using the `writer` object contained by the `HTTPIndexer` class.

- `isCrawled()` indicates if a document is already conteind in the index $\mathcal{RS}$. Returns *true* if the document is already in the index or if the document is already contained in the queue. This method is used to avoid adding duplicate entries to the queue or the index.

- `doIndex()` starts the actual indexing. This method is used by the `LuceneIndexer` class to start the actual indexing process. This method creates a index if it doesn't exists yet or opens and existing index. Furthermore, the private method `startIndexing` is called, which creates an instance of the `IndexerThreadManager`, see section 4.2.5, and manages the indexing threads. When the indexing process is finished and the `startIndexing` method returns, the index is *optimized*, making it faster to search the index. Finally, the index writer `writer` is closed.

### Starting the Indexer

The actual indexing process is started by calling the `doIndex()` method. This method is called by `LuceneIndexer` when the `startIndexer()` method of that class is invoked. The `doIndex()` method of `HTTPIndexer` is described in detail in the previous section, and calls the method `startIndexer()` of the `HTTPIndexer` class. This latter method controls the creation of threads, and uses a instance of `IndexerThreadManager` for the actual creation of the threads.

Many times, the indexer has to index a number of resources at once. For example, when results of a Internet search engine is processed or when the *library* of documents selected by a course author have to be indexed. The `startIndexer()` method creates additional threads when there is enough work. The method also ensures that new threads are recreated when they stop indexing. This sometimes happens when an exception occurs while indexing or when the worker threads are indexing resources faster than the application is adding new resources to the queue of URL's managed by `HTTPIndexer`. Source code listing 4.1 shows the current implementation of the `startIndexer()` method of `HTTPIndexer`.

The `doIndex()` method is in the current implementation a synchonized method which returns when the indexing of all resources in the queue is finished. The process of indexing a number of URL's can take quite some time, up to a minute for approximately 30 documents. As a result, this means the server is not available for other requests. This could be changed by making the indexing entirely asynchronous, for example by creating a seperate thread for the `HTTPIndexer` class. However, this requires a change to the entire server, because checks should be added to ensure that the indexing is finished before other operations are done on the indexer.

Listing 4.1: `startIndexer()` method implementation

```
/**
 * Start the actual indexing.
 * This method manages the creation of threads
 * using {@link IndexerThreadManager}
 */
private void startIndexing() {
        log("[HTTPIndexer] Setting up worker threads", DEBUG_DEBUG);

        IndexerThreadManager manager = new IndexerThreadManager(this);
        manager.addThread();
        log("[startIndexing] Worker " + 0 + " started", DEBUG_DEBUG);

        while (manager.checkThreads()) {
                if (queue.size() > 0 &&
                manager.getThreadCount() < HTTPIndexer.NUM_WORKERS) {
                        manager.addThread();
                }

                log("[startIndexing] " + writer.docCount() +
                " documents indexed, " + queue.size() +
                " remaining (" + manager.getThreadCount() +
                " Threads )", DEBUG_DEBUG);

                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
```

```
                        e.printStackTrace();
                }
        }
}
```

### 4.2.4 HTTPIndexerWorker

Most of the real work in the process of retrieving resources from the internet, parsing the resources and relating concepts to a resource is implemented in the `HTTPIndexerWorker` class. As is explained in section 4.2.2, at most ten seperate threads are indexing resources in OntoAIMS. These threads are instances of the `HTTPIndexerWorker` class and managed by the `IndexerThreadManager` which is described in section 4.2.5.

When a `HTTPIndexerWorker` class is running in a thread, it will fetch one URL at a time from the queue maintained by `HTTPIndexer`, download the resource, parse the content, relate concepts to the resource and store the resource in the index.

#### Methods

The main methods defined by `HTTPIndexer` are:

- `indexUrls()` This is the method that fetches resources from the queue in `HTTPIndexer`. The `IndexURLItem` that is returned by the `getQueueItem()` method is passed on to the `addDocument()` method. This method is invoked by the `run()` method of this class, which is invoked when the thread is started.
- `addDocument()` Performs the actual parsing of the document and adds the result to the index by means of the `addIndexItem()` method of `HTTPIndexer`. This function creates a `IndexerDocument` object, realised by either the `HTMLDocument` or `PDFDocument` class, which parses the document. When the document is parsed, this method creates a `Document` object which can be added to the Lucene index.
- `relateConcepts()` Relates concepts to a resource. A detailed description of this method is given in section 4.2.4.
- `createTempIndex()` This method is used by the `relateConcepts()` method and creates a temporarly index in memory for extracting concepts from a resource. the

#### Relating Concepts to Resource

This section describes the implementation of relating concepts to a resource. The design of this part is described in section 3.3.5. Extracting concepts from resources or relating concepts to resources, which is actually a better description of the process, is done in one method in the `HTTPIndexerWorker` class. `relateConcepts()` does the actual work of relating concepts to a resource. Relating concepts to the resource is done using a process consisting of three steps:

1. Create an in-memory temporary searchable index of the resource.
2. Get all *concepts* and *other forms* of concepts in one array.

3. Search the document with the items in the array created in the previous step.

Obviously, the process is a little more complicated than described using the three steps, but it helps getting an overview of the process. The following description of the concept relation process is described using these three steps as a guide.

**Create an Index in main memory** In the collection of contributed tools for the Lucene library, an indexer is included that makes it easy to create a single-document index which is located in the internal memory of a computer system. The internal memory, or main memory, of a computer can be accessed very fast and provides high data transfer rates. The index in main memory can be searched very fast compared to the regular indexes created by Lucene on the harddisk of computer systems. The class providing this feature is named `MemoryIndex`. This class contains methods that covers the entire process of creating the index, adding a document to the index and searching the document in the index. According to the *JavaDoc* documentation of the class[6], the library is very fast and can handle up to $500,000$ queries per second on a currently average computer system.

In the OntoAIMS implementation of the indexer, the memory index is created by the `createTempIndex()` method of the `HTTPIndexerWorker`. This method first creates an Lucene index in the RAM memory of the computer. The resource which is indexed by this method is given to this method by using a parameter. `createTempIndex()` creates five fields for a resource which are used for relating the concepts to a resource. These five fields are:

- **keywords** Keywords specified in the metadata of the document.
- **title** The title of the document.
- **content** The content of the document - the body text.
- **importantWords** Important words in the document - specially formatted sections, such as headings.
- **urltext** The anchor texts of links in the document.

**Get all Concepts of the Domain** Relating concepts to a resource requires at least a list of all the concepts in the domain. For this task, a method in `HTTPIndexer` can be used, called `getDomainConceptQueryStrings()`. This method, not discussed in the section on `HTTPIndexer`, is only used in the relating concepts process. The list of concepts is created during the initialization process of the `HTTPIndexer` class. The `LuceneIndexer` class provides a list of concepts, because this class has easy access to the `ServerDomainAgent` class, which manages the domain in OntoAIMS. `HTTPIndexer` has access to an array containing all the concepts in the current domain. While OntoAIMS supports multiple domains, an instance of a `LuceneIndexer` is specific to a single domain. As a result of this, `HTTPIndexer` objects are also specific for one domain. The items in the array of domain concepts are of the type `ConceptQuery`. This class implements some convenient methods making it easier to get the *otherforms* and creating query strings for the various parts of the concept.

The `ConceptQuery` class has two public methods and one private method. For every concept in the domain, a `ConceptQuery` object is created. The main purpose of the `ConceptQuery` class is to create query strings which can be used by the concept relating process in `HTTPIndexerWorker`. The three methods of the `ConceptQuery` class are listed below:

[6] See http://lucene.zones.apache.org:8080/hudson/job/Lucene-Nightly/javadoc/ for the JavaDoc of `MemoryIndex`

- getConceptName() A public method that returns the name of the concept.
- getQueryStrings() A public method that returns an String array with for all the possible names of the concept. For example, when a concept copying files has three *otherforms*, namely copying files, copy file and copying file, this method will return an array with three items. The first alternative name copying files will be omitted, because this alternative name is the same as the concept name.
- formatQuery() A private function used by getQueryStrings(). This method formats the strings returned by getQueryStrings(). Formatted queries are directly usable by the QueryParser class of Lucene. Non-word characters, such as ˜ and / are escaped to not break the query.

In the HTTPIndexer class, the method getDomainConceptsQueryStrings() method gets all query strings from the ConceptQuery objects and puts them in one large array. This array is returned to the HTTPIndexerWorker class in order to use the queries in this array to relate concepts to a resource.

**Searching a Resource for Concepts** When the temporary memory index is created in step one, and an array with all concepts and *otherforms* is obtained in step two, the final step is to actually get the most related concepts for a resource. This is the done in the relateConcepts() method of HTTPIndexerWorker. This method loops through the array with concepts and searches the five fields created in the temporary index. The result of this search is a float value between 0 and 1 indicating the relevance of the query to the field. These values are added for a single concepts. Because of the nature of Lucene, keywords (concept names) that are found in the title are more important than keywords found in the body text. The title is generally shorter, resulting in a higher value because of the nature of the scoring algorithm used by Lucene [19].

After the calculation of the scores for all concepts, the algorithm filters out concepts with a score lower than *MIN_RELEVANCE*, which is currently defined as 20. The method limitConceptCount() reduces the number of related concepts to at most *MAX_RELATED_CONCEPTS*, which is defined at 10.

Some more research should be done to improve this algorithm, because in the current implementation the results are not always as expected. Possible improvements is to considere links between concepts in the process. For example, a concept with the name more is part of the *Basic Linux Ontology*. This concept is by this algorithm related to a large number of resources, while it only should be related to resources which discuss the *more command* in Linux. This could be fixed when the related concepts of the more concept are taken in consideration when relating concepts to a resource.

### 4.2.5 IndexerThreadManager

The IndexerThreadManager manages the threads used by the HTTPIndexer for adding documents to the index. This class only provides methods for creating new threads, getting the number of currently running threads and checking if there are running threads. This class does not implement a strategy for creating new threads but only for administrating the thread objects. The performance of the indexer is influenced by the maximum number of concurrent threads and the speed at which they are created. In OntoAIMS the strategy for creating new threads is implemented by HTTPIndexer. The most important methods of this class are listed below:

- addThread() creates a new thread using the HTTPIndexerWorker class.

- `checkThreads()` returns *true* if there are any `HTTPIndexerWorker` threads running. Usually, at least one thread will be running if there are URL's that have to be indexed. If all threads are finished, it is very likely that everything that could be indexed, is actually indexed.

- `getThreadCount()` will return an integer indicating the number of threads that are currently running. This value can be used in combination with the number of items in the queue maintained by `HTTPIndexer` to decide if thare should be additional threads created in order to speed up the indexing process.

We choose to use threads, because testing showed that during the indexing of a number of resources, most of the time was used by downloading the resources. Parsing of the documents was only a fraction of the total time needed to indexing one document. Downloading the web pages does not require any resources from the system. The performance can be improved by using multiple threads downloading and indexing documents, causing a more efficient usage of system resources.

### 4.2.6 Parsers

Parsing of the resource is one of the steps when a resource is indexed by OntoAIMS. Section 3.3.4 explains the design of the parsers. In this section we will discuss the implementation of the parsers in OntoAIMS. The `HTTPIndexerWorker` class parses the documents in the `addDocument()` method. The implementation of the parsers is done in seperate classes which implements the `IndexerDocument` interface. This interface defines a set of methods that must be implemented by the classes that implements this interface. Among those methods are `getTitle()`, `getContent()`, `getAuthor()` and `getModified()`. These methods should return `String` objects containing the part of the document indicated by the method name. For example, the name of a document can be retrieved using the `getTitle()` method.

The main purpose of the actual parsing is to transform documents of various types into a general format that can be processed by OntoAIMS. Mainly, the content of the document - the text visible to the user without formatting information - should be available to the application. Lucene only supports plain text while indexing and does not provide methods to process document formats such as HTML, PDF or Word documents. Several implementations of Lucene are available that contain parsers for particular document formats, but these parsers are very generic and does not enable the application to extract specific parts of a document. Therefore, we decided to implement custom parsers using existing libraries.

Currently, OntoAIMS supports two document types that can be indexed by the application: HTML documents, implemented in the `HTMLDocument` class, and *Portable Document Format* documents, implemented in the `PDFDocument` class.

#### HTMLDocument

The `HTMLDocument` class implements a parser for HTML documents. The parser supports documents formatted using the HTML markup language, specified in a standard of the World Wide Web Consortium[7]. The class does not explicitely support a particular version of the HTML specification. On the Internet, most HTML documents are not conform the specifications. `HTMLDocument` tries to parse documents formatted using HTML or XHTML.

[7] See http://www.w3.org/TR/html401/ for the HTML 4.01 specification

An existing library, called *HTMLParser*[8] is used for the actual parsing. This parser is open source and written in Java. The parser is designed to be very flexible, meaning that it will not stop when an error is encountered in the HTML document. Authors of web sites on the Internet tend to create HTML which will work in most commonly used browsers, but is not valid HTML. To let OntoAIMS process most documents on the Internet, the parser should allow invalid HTML and process it in a way the author intended. The `HTMLParser` provides features to strip an HTML document from all its markup tags, resulting in a plain text document. The library also makes it easy to extract particular parts of a document, for example only the title or metadata. HTMLParser is widely used by various popular Java projects and well-tested. The project uses rigid testing strategies, mainly based on unit tests and regression tests.

During testing of the parser for HTML Documents, we discovered that the parser could not process documents with a changing character set. Investigation of the code and HTML specifications lead to the conclusion that this was a bug in the HTMLParser library. To fix this issue, a patch was created and submitted to the maintainer of the library. The upcoming version[9] of HTMLParser will contain this patch. In the current implementation of OntoAIMS we used a patched version of the current version[10] of the HTMLParser library.

HTMLParser can parse web pages directly from the Internet, however this will lead to problems in certain cases. To solve these problems, the documents are downloaded by the `HTMLDocument` class, before they are parsed using HTMLParser. This will solve problems related to web sites that are currently not available. HTMLParser would try to download these documents infinitely, `HTMLDocument` will time out after a certain period. `HTMLDocument` will define a *User Agent* when downloading web pages from the Internet. Some web sites require this, such as http://www.google.com.

Metadata of a HTML document is extracted using the HTMLParser library. In HTML documents, metadata is defined using the `meta` tag in the `head` section of the document. The HTML specifications [7] does not define which metadata should be defined in a HTML document, making it difficult to extract all the available metadata. The specifications only define how a `meta` tag should be defined and which atributes can be used within this tag. Some metadata, such as the title, is defined in seperate tags, making the extraction of metadata from an HTML document even more difficult. There are two variants of the `meta` tag, seperated by the attribute used for defining the *key* of the metadata. The first variant uses the *name* attribute to define the name of the `meta` tag. The second variant uses the *http-equiv* attribute to define the name of the `meta` tag. This latter variant of the `meta` tag is used by the web server to define additional *http headers* for the document based on these metadata tags. In the current implementation of `HTMLDocument`, only the *author* and *keywords* are extracted from the document. However, additional meta data can be added easily using the `getMeta()` method.

#### PDFDocument

A parser for *Portable Document Format* documents is implemented in the `PDFDocument` class. Just like the `HTMLDocument` class, is the `PDFDocument` class an implementation of the `IndexerDocument` interface. The functionality of the `PDFDocument` class is identical to the functionality of the `HTMLDocument` class. In contrast to HTML documents, PDF documents are binary files, making it impossible to read them using a standard text editor. The document format is also more complex. On the other hand, the PDF specifications define more details on the definition of metadata, making it easier to extract the metadata from a resource. Most of the details described for the HTML documents in the previous

[8] See http://htmlparser.sourceforge.net for more information on HTMLParser

[9] The upcoming version of HTMLParser will probably be version 2.0 and is scheduled for May 2007.

[10] During this project, version 1.6 of HTMLParser is used

section, also holds for PDF documents. The `PDFDocument` class also uses an external library for the actual parsing of PDF documents.

*PDFBox* is used for extracting the contents from PDF resources. The website of PDFBox[11] has a short description of the project: "PDFBox is an open source Java PDF library for working with PDF documents. This project allows creation of new PDF documents, manipulation of existing documents and the ability to extract content from documents. PDFBox also includes several command line utilities". We use the library to get a text representation of the content of the document and the available metadata. `PDFDocument` extracts most of the common available metadata from PDF documents: *title*, *author*, *creation date*, *modification date*, *subject*, *producer*, *creator* and *keywords*. The current implementation of `PDFDocument` does not extract the *important sections* of the document. As a result the method `getImportantWords()` does always return an empty string.

## 4.3 Searching for Resources

Section 4.2 describes the implementation of the indexer in OntoAIMS. But only indexing documents does not result in a improved resource recommendation. Therefore, another package is created which searches the index and processes the results. Additional resources are also added to the index by this package, for which external search engines are used. This section will describe the implementation of this part of OntoAIMS.

## 4.4 Search Package

The `Search` package brings all components of the resource recommender together and contains the implementation of the search process that forms the actual resource recommender. An overview of this package is presented in figure 4.3 which shows the main classes of the package along with the relevant methods and fields within those classes. This package uses several external libraries, including the *Lucene library*, as described in 4.2.1. The current implementation of the `Search` package uses web services from *Google* and *Yahoo* for searching the Internet.

The main class of the `Search` package, used by the other parts of OntoAIMS, is the `ServerSearchAgent`. This class us mainly used by `MetaAgent` of the server part of OntoAIMS. This class decides which classes of the package are used for the various tasks and can process the `CommandRequests` received by the OntoAIMS server.

### 4.4.1 `ServerSearchAgent`

The `ServerSearchAgent` class is the part of the `Search` package that is used by the other parts of OntoAIMS. This class implements the `AgentObject` interface. Classes implementing this interface can be used as an agent within the OntoAIMS server. The server part of OntoAIMS uses several agenst, such as the *DomainAgent*, the *UserProfileAgent* and *CourseAgent* which provides access to particular parts of the data located on the server. The common `AgentObject` interface provides a generic way of how the agents handle requests from the OntoAIMS client. The `MetaAgent` class decides which agent processes which commands, represented by a `CommandRequest`. For example, the `MetaAgent` sends commands which are related to the domain to the `DomainAgent`. The search commands are processed by the `ServerSearchAgent` and therefore, the `MetaAgent` sends search commands to an instance of this object.

[11] See http://www.pdfbox.org/

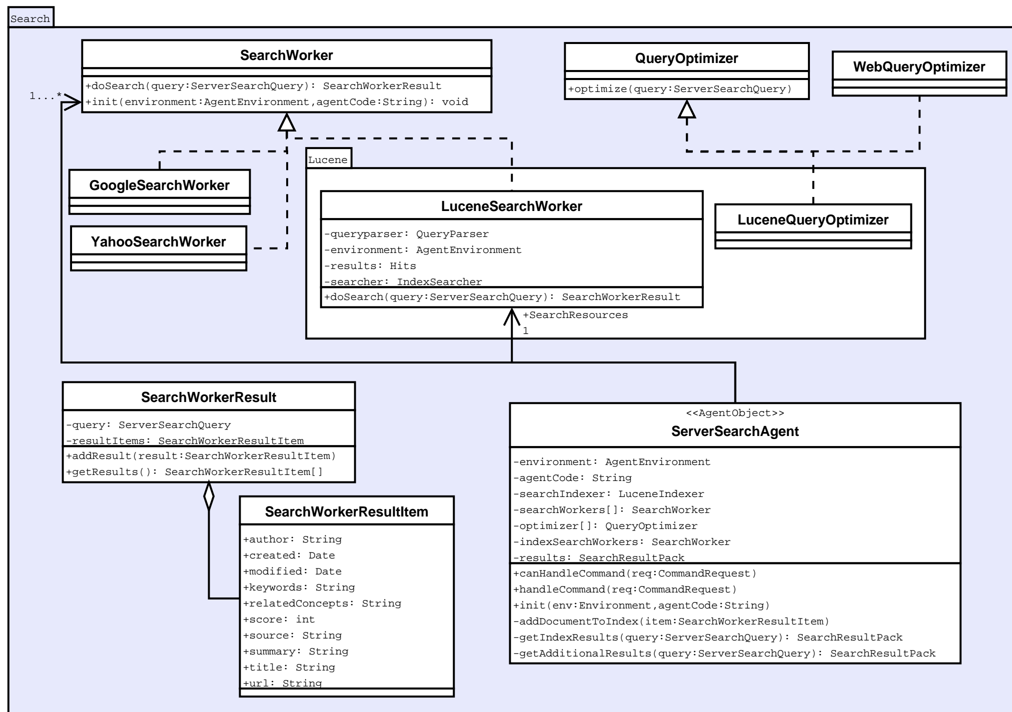


Figure 4.3: Search package.

Three methods are defined by the `AgentObject` interface: `init()`, `canHandleCommand()` and `handleCommand()`. The `init()` method takes two parameters: an `AgentEnvironment` and a string representing the *agentCode*. This latter parameter is used to identify an instance of a `AgentObject`, the first is an interface providing access to other parts of the server, such as configuration data and the other available agents. The previously mentioned `MetaAgent` implements the `AgentEnvironment` interface and is as such used by most OntoAIMS agents as the `AgentEnvironment`.

The `canHandleCommand()` and `handleCommand()` are the two methods used for processing commands by the agents in OntoAIMS. Both methods accepts one parameter shich is of the type `CommandRequest`. The `canHandleCommand()` returns if the agent is capable of processinig a command of the type given as tha argument. This is used by the `MetaAgent` to check which agent should process a particular command received from the client. If the agent can process the command, the method `handleCommand()` is used for delegating the processing of a commend to the agent. The agent will process the command and return an object which implements the *abstract class* `CommandResponse`. These objects can directly sent to the server by the `MetaAgent`.

The `ServerSearchAgent` can handle two type of commands:

- `COMMAND_SEARCH_COMMAND` is send by the client when the user requestis a *simple recommendation*.
- `COMMAND_SEARCH_GETMORERESULTS` is send by the client when the user requests a *detailed recommendation*.

The first, `COMMAND_SEARCH_COMMAND` is a simple resource request and will return a small list of resources. The second, `COMMAND_SEARCH_GETMORERESULTS`, will return a larger list of resources in

which the personal preferences and knowledge of the user is used for refining the list.

Furthermore, the class provides methods used internally for various tasks. ServerSearchAgent has several instances of other classes available which are used for delegating commands to. SearchWorker objects are used for searching specific type of resource collections, such as the LuceneSearchWorker which searches the Lucene index, described in the previous section, GoogleSearchWorker which is used to search the Internet using Google and YahooSearchWorker for searching using Yahoo. Instances of these classes are created during the initialization phase of ServerSearchAgent by calling the method initWorkers(). Two methods, getIndexResults() and getAdditionalResults() are used for actually searching the various resource collections and get relevant resources.

Also an instance of the indexer agent LuceneIndexer is created during initialization of ServerSearchAgent. This object is created by calling the method initIndexer(). The reason for the creation of a LuceneIndexer object, is because ServerSearchAgent indexes resources retrieved from the Internet. For adding a resource to the Lucene index the class provides the method addDocumentToIndex().

Before queries are send to the relevant *search worker*, the queries have to be optimized in order to incorporate the preferences and existing knowledge of the user. This is done by classes which implement the QueryOptimizer interface. ServerSearchAgent uses instances of these classes for optimizing queries.

### 4.4.2 SearchWorker Interface

Searching the various resource collections, such as the Lucene index and resource located on the Internet, is implemented in classes that inherit the SearchWorker interface. This interface defines only two methods which have to be implemented by classes that implement this interface. The two methods are init(), which is called before the object is used for searching. This method should initialize the object and should do tasks such as initializing the library which is usd for searching and setting up connections to external document repositories. The other method defined by this interface is the doSearch() method. Calling this method will result in the actual search process and is in most implementations of the SearchWorker interface the most complex method. This method accepts one parameter query which has to be a ServerSearchQuery object.

The implementation of the doSearch() method in the SearchWorker classes, should return a SearchWorkerResult object. This object contains global information on the search results and contains an array of SearchWorkerResultItem objects representing the individual results or the resources. A SearchWorkerResultItem contains the following fields:

- *author*
- *title*
- *url*
- *summary*
- *keywords*
- *relatedConcepts*, containing a comma seperated list of related concepts.
- *score*, containing a score indicating the relevance of the result.
- *source*, containing the source of the result, for example *Google* or *library*.

- *modified*, containing the date of last modification of the resulted resource.
- *created*, containing the creationd date of the resulted resource.

### Google SOAP Search API

Google provides a web service enabling to use the search engine of this company in applications. Google SOAP Search API can be used for searching the Internet using Google's search technology. However, since december 2006 this service is depricated, meaning that the service still remains available for the time being. However, it is not possible to download the software development kit for the service and there will be no new registration keys be issued, meaning that only existing keys can be used.

The Google SOAP Search API uses the *Simple Object Access Protocol*, abbreviated to SOAP, for communicating with the client. This protocol is based on XML. Full documentation for the protocol used by the Google SOAP Search API can be fount on the Google website[12]. In the past, Google also provided a software development kit for the service, containing libraries for many commonly used programming languages. OntoAIMS uses the Java library provided by Google for accessing the service. The webservice accepts a search request containing a search query and various parameters. These parameters could be set using various methods of the main Google class `GoogleSearch`:

- `setKey()` is used to define a key used to identify the user. Keys for the service could be obtained from Google before the service was depricated.
- `doSpellingSuggestions()` sets if the service should provide suggestions for a better spelling of the query words.
- `setFilter()` indicates if similar results in the set of results should be filtered out.
- `setLanguageRestricts()` can be used to restrict the language of the results, for example to Dutch or English.
- `setMaxResults()` is used to limit the number of results returned for a query, the maximum number is limited to 10.
- `setQueryString()` is used to define the query string for a search.
- `setRestricts()` sets if the search should be restrict to a particular subject. This method can be used to restrict the search to *Linux*, *BSD*, *Microsoft Windows* and several other subjects.
- `setSafeSearch()` can be set to *true* to filter out web pages containing certain material, such as pornogrpahy.
- `setSoapServiceURL()` can be used to define a different URL for the service.
- `setStartResult()` indicates the index of the first result.

There are some other methods provided by this class, but these are not relevant in this context. By adding keywords to the query string set using the `setQueryString()` method, it is possible to set some additional aspects of the query. For example, it is possible to restrict the query to specific document types (PDF, Word documents, etc) or limit the search to a particular domain. The actual

[12]See http://code.google.com/apis/soapsearch/

search action is started using the `doSearch()` method, returning a `GoogleSearchResult` object, which contains a number of `GoogleSearchResultElements` for representing the individual result items.

A `GoogleSearchResults` objects contains some meta information on the search action, such as the number of results, the index of the first element and optional suggestions for improved spelling of the query. A `GoogleSearchResultElement` has a number of fields to represent an individual search result item. The most relevant items are listed here:

- The *title* of the document, available using `getTitle()`.
- The *URL* of the document, available using `getURL()`.
- A *summary* can be obtained using the `getSummary()` method, however not for all document is a summary available.
- A *snippet* highlighting the relevant part of a result based on the query, available through `getSnippet()`.
- Some Google results are categorized in directories. If this is the case for a result, the directory title can be obtained using the `getDirectoryTitle()` method and the `getDirectoryCategory()` gives the category of the directory as a result.
- For some results related resources are available. If this is the case, the method `getRelatedInformationPresent()` will return *true*.

Some limitations apply to the Google SOAP Search API. First, it is required to specify an *API key* with every search request. Since Google SOAP Search API is not actively supported anymore, it is not possible to obtain new API keys. It is officially not allowed to share a single API key, since these keys are linked to a specific user name. The API key is used to track usage of the service. Every key has a limit of 1,000 free queries a day and a single query returns at most 10 results. Using subsequent queries it is possible to get at most 1,000 results for a single search query. There are also some limitations on the query string, but these limitations also apply to the normal Google Search. Every query can contain at most ten individual words and the length of the query may not exceed 2048 bytes.

The `GoogleSearchWorker` used by OntoAIMS uses the Google SOAP Search API for implementing the search functionality. The class format the query in a way that can be used directly by Google SOAP Search API and processes the results and put them in a `SearchWorkerResult` object. The API key issued by Google should be put directly in the source code of the class in the variable `key`.

#### Yahoo WebSearch API

The `YahooSearchWorker` implements a search worker that searches the Internet using the web service provided by Yahoo. Like Google, Yahoo provides several web services that can be used to implement search features in applications. The API Yahoo provides for searching the Internet is called *Yahoo WebSearch API*[13]. The Yahoo WebSearch API delivers similar functionality as the Google SOAP Search API, described in the previous section, but there are some differences.

The Yahoo WebSearch API is implemented using *Representational State Transfer (REST)*[14]. Yahoo provides a software development kit which contains libraries for several languages for the

[13] See http://developer.yahoo.com/search/ for more information on the Yahoo WebSearch API

WebSearch API. Among those libraries is a Java implementation which is used in OntoAIMS. The main class in this API is the `SearchClient` class, containing several methods for using the various services provided by Yahoo. To use the web search service, one has to use the `webSearch()` method of this class. The sole parameter accepted by this method should be a `WebSearchRequest` object. This `WebSearchRequest` class consists of methods to configure various settings for the search request. The most important settings are listed here with the relevant method:

- `setQuery()` is used to set the search query string.
- `setResult()` can be used to limit the maximum number of results.
- `setStart()` is used to set the index of the first result.
- `setLanguage()` can be set to only get results in a specific language.
- `addLicense()` can be set to limit the results to documents licenced under a specific license.
- `setSimilarOk()` is used to filter similar documents from the result.
- `setFormat()` can be used to limit the search on documents of a specific format, for example PDF or HTML.

Some other settings are available as well, but these are not relevant or used by the OntoAIMS implementation of the `YahooSearchWorker` class.

The `webSearch()` method of the `SearchClient` class of the Yahoo library gives a `WebSearchResults` object that specifies the number of results and the number of returned results. This object also contains an array of `WebSearchResult` objects, which represents the actual results. A `WebSearchResult` object contains the following items:

- `getClickURL()` returns the URL of the document which should be used to download the resource according to Yahoo. This URL downloads the resource via Yahoo, giving Yahoo the opportunity to track the downloads of a resource and as such to improve the search results, according to the provided documentation.
- `getURL()` returns the direct URL of the web page. `YahooSearchWorker` only uses this method in the current implementation.
- `getTitle()` returns the title of the resource.
- `getSummary()` returns a summary of the resource.
- `getModificationDate()` returns the last modification date for the resource.
- `getMimeType()` returns the mimetype for the resource.

The usage of the Yahoo WebSearch API is limited per ip-address. Every ip-address has a limit of 5,000 queries a day. A query can at most return 100 results, but by specifying the index of the first result using the `setStart()` method, it is possible to get additional results for a particular query. Before an application can use the Yahoo WebSearch API, a key is required called *AppID*. This key is used to identify the application and track its usage of the Yahoo WebSearch API. Currently, the Yahoo WebSearch API has more possibilities than the Google SOAP Search API, both in terms of flexibility in the used queries, but also on the field of usage limitations.

#### Lucene search

A third implementation of the SearchWorker interface is the LuceneSearchWorker. This class is part of the Search.Lucene package, which contains Lucene specific parts of the Search package. The implementation of the LuceneSearchWorker is a bit more complex than the YahooSearchWorker and GoogleSearchWorker, because the results contain more information on the results. The extra information can be used by the ServerSearchAgent to personalize the results from this SearchWorker.

In the current implementation of the OntoAIMS resource recommender, only resources delivered by LuceneSearchWorker objects are returned to the client. Resources retrieved using Yahoo and Google are first added to the Lucene index using the LuceneIndexer and retrieved again using the LuceneSearchWorker. At first sight, this might sound complicated, but adding the resources to the indexer adds additional metadata to the resource which is not available when the resource is provided by the YahooSearchWorker or GoogleSearchWorker. Adding the resources from these two SearchWorkers to the index takes some time, but this is not a real problem because only detailed resource recommendations are processed this way.

Queries accepted by the LuceneSearchWorker's doSearch() method are similar to the queries accepted by the GoogleSearchWorker and YahooSearchWorker. However, the limitations that apply to these two search workers in terms of query length, do not apply to the the LuceneSearchWorker. A parser, implemented in the QueryParser class, part of the Lucene package, is used for parsing the queries and transform them into Query objects which can be directly used to query the Lucene index. An instance of the Lucene IndexSearcher class is used to search the index.

Depending on the query, which is provided as an ServerSearchQuery object, the LuceneSearchWorker searches the index using different fields. A simple resource recommendation causes only a search on the related concepts specified for the resources in the index. A detailed resource recommendation will result in a search on most fields available in the index.

### 4.4.3 QueryOptimizer class

Before a query is used for searching resources, the ServerSearchAgent optimizes the query using an object inheriting the QueryOptimizer abstract class. The current implementation of the resource recommender in OntoAIMS contains two child classes inheriting this class: LuceneQueryOptimizer and WebQueryOptimizer. These two classes are very similar and only differ in detail. The LuceneQueryOptimizer is used to optimize queries used to search the local collection of resrouces managed by Lucene and accessed using the LuceneSearchWorker. Because the LuceneQueryOptimizer class is specific to the Lucene library, this class is contained in the Search.Lucene package. The WebQueryOptimizer is used to optimize queries for the YahooSearchWorker and GoogleSearchWorker. Both Google and Yahoo uses a very similar syntax for defining queries, making it possible to implement the optimizer in a single class.

The two classes that inherit the QueryOptimizer implement the processing of queries described in section 3.4.4. The class defines one important method optimize() which takes a ServerSearchQuery object and returns an object of the same type, but describing a optimized version of the original ServerSearchQuery object. The optimize() method should be overriden by the classes which inherit the QueryOptimizer class and implement an optimization process returning a query string in the specific syntax used by a certain search engine. Code listing 4.2 displays the implementation of the optimize() method implemented in the LuceneQueryOptimizer class. The query optimizing process implemented in this method is described in section 3.4.4.

Listing 4.2: `optimize()` method implementation in `LuceneQueryOptimizer`

```
public ServerSearchQuery optimize(ServerSearchQuery query) {
    queryString = new StringBuffer();
    ConceptStruct mainConcept = extractConcept(query);
    query.currentConcept = mainConcept;

    // append otherforms
    queryString.append("(");
    queryString.append(mainConcept.name);
    queryString.append(getOtherFormsQuery(mainConcept));
    queryString.append(")");

    ConceptStruct[] linkedconcepts = this.getLinkedConcepts(mainConcept);
    if (linkedconcepts.length > 0 && (getBeliefValue(mainConcept) > 50)) {
        for (int i = 0; i < linkedconcepts.length; i++) {
            queryString.append(" AND ");
            queryString.append("(");
            queryString.append(linkedconcepts[i].name);
            queryString.append(getOtherFormsQuery(linkedconcepts[i]));
            queryString.append(")");
        }
    }
    query.query = queryString.toString();
    System.out.println(queryString.toString());
    return query;
}
```

The `optimize()` method in the classes inheriting `QueryOptimizer` use various helper functions provided by the `QueryOptimizer` class. These methods provide access to some aspects of the domain and the user profile and make it easier to extract relevant information of these information models. These methods use instances of relevant objects. The two most important objects used by the `QueryOptimizer` is the `ltcs` object and an instance of the `ServerDomainAgent` class.

The `ltcs` object is an instance of the `ConceptualState` class, provided by the *OWL-OLM* package. This class provides access to the enhanced user model which is used for optimizing the queries. The `ConceptualState` class provides two methods for obtaining the *belief value* for a certain concept, called `getKnowledgeScore()`. The difference between the two methods is that one accepts a string containing the URI to the concept in the OWL ontology as a parameter, while the other accepts a *Vector* object containing a list of resource URI's as parameter. The first method returns the belief value for the concept, while the second returns a value between 0 and 100 indicating how many concepts in the *Vector* object are found in the user's conceptual state and are annotated with a belief value.

The `domainAgent` variable is an object of the type `ServerDomainAgent` providing access to the *domain model*. This object is used to retrieve the domain concepts from the model and relate a concept to other concepts by following the links to other concepts. The `getPack()` method of the `ServerDomainAgent` class returns a `DomainPack` object containing all the concepts, links and document references in the domain. This `DomainPack` is used to get `ConceptStruct` objects for the various concepts in the domain. This last type of object contains information on a concept in the domain, such as `otherForms` and `synonyms`. Furthermore, the `DomainPack` has a `conceptLinkDefs` variable containing `ConceptLinkStruct` objects. These objects defines links between two concepts, and is are used by `QueryOptimizer` to get concepts which are somehow related to another concept. Because

the `ConceptualState` class only allows to get the *belief value* for URI's referring to the concept in the OWL ontology, the `ConceptStruct` also contains a field `basedOnResURI` containing the URI of a concept in the OWL ontology, if available. This value is accessible using the `getBasedOnResURI()` method of the `ConceptStruct` class. The `QueryOptimizer` class provides a delegate method accessing this method, called `getConceptURI()` which returns the URI in the domain of a concept when the name of the concept is provided to the method as a parameter.

The main methods of the `QueryOptimizer` are listed here:

- `extractConcept()` This method tries to extract a main concept for the query. In many cases, a concept is already provided, but some queries do not contain a concept, which is necessary to add additional concepts to the query and refine the query. In section 3.4.4 the variable $c_m$ refers to the main concept. This method tries to extract this main concept $c_m$ from the query.
- `getConceptURI()` Returns the URI in the OWL ontology for a particular concept.
- `getBeliefValue()` Returns the *belief value* $v_c$ for the user for a specific concept $c$. This belief value is extracted from the user model. There are two implementatioins of this method, the first accepting the name of the concept as a string, the second accepts a `ConceptStruct`.
- `getOtherForms()` Returns the alternative names for a concept. In section 3.4.4 these alternative names are refered to by the variable name $o$ in the description of the query refinement process.
- `getLinkedConcepts()` Gives the linked concepts $b_{1...n}$ for concept $c$. This method extracts the concepts which are linkd from concept $c$ or linked to from other concepts to concept $c$. The `ServerDomainAgent` is used for extracting these additional concepts from the domain model.

`ServerSearchQuery`

The `ServerSearchQuery` class is used for managing a query on the server. Objects of this type are used to transfer the query between the various classes within the `Search` package. The `QueryOptimizer` class uses instances of the `ServerSearchQuery` while receiving and returning an optimized query. The `ServerSearchQuery` class extends the `SearchQueryStruct` which is used for transferring queries between the client and the server. The `ServerSearchQuery` class has the following fields, which are accessible using getter and setter methods: `courseCode`, `currentConcept`, `query`, `source`, `userCode`. The `query` field contains the textual representation of the query, `source` is an optional field which restricts the query to resources originating from a particular source, such as *Google*, *Yahoo* or *Library*. The `currentConcept` field contains the most relevant concept for this query, which either can be defined by the client or extracted from the query by the `QueryOptimizer` classes. The `courseCode` contains the current course code of the course the user is currently following and the `userCode` contains the code of the current user, which is required to personalized the query.

## 4.5 Implementation on the OntoAIMS Client

In order to use the new resource recommendation implementation on the server, some changes where required on the client of OntoAIMS. This project was restricted to the *viewer* and as such does not contain improvements to the *admin* and *editor* environment of OntoAIMS. It is beyond the scope of this project to present the existing implementation of the OntoAIMS *viewer* client. In this section,

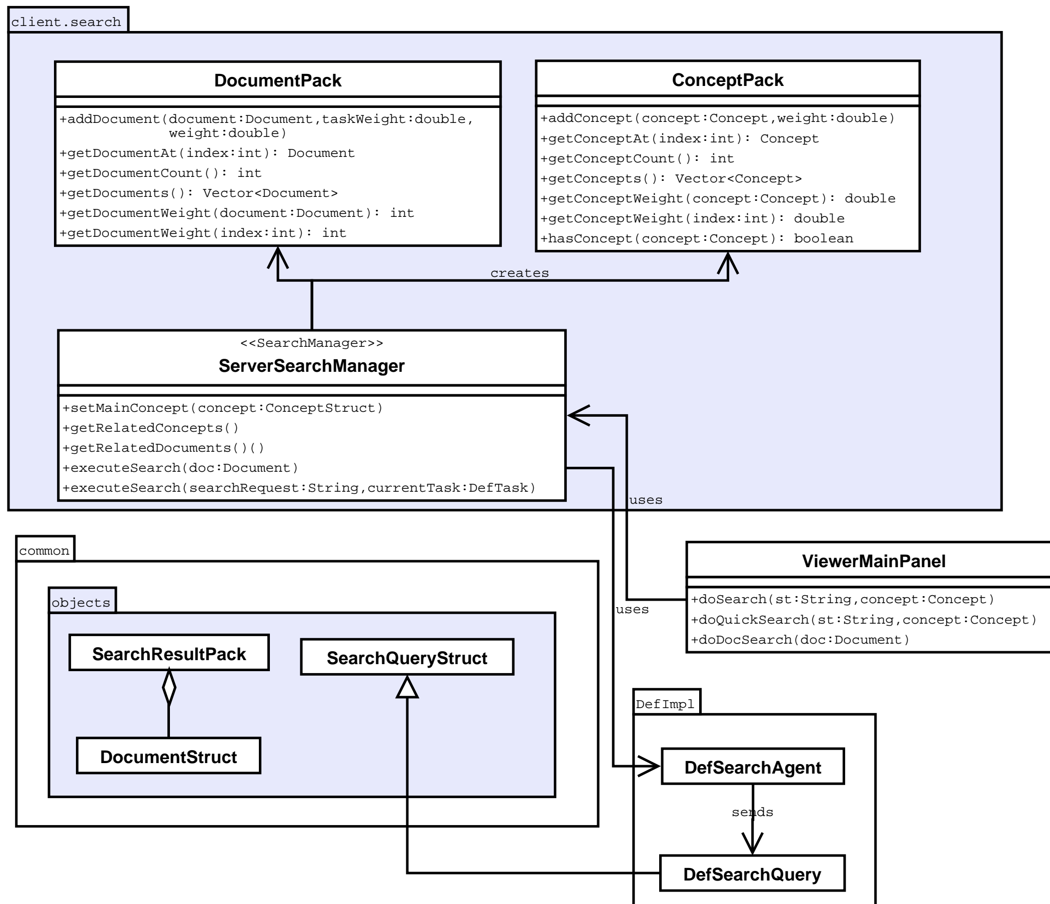


Figure 4.4: Relevant client packages for resource recommender.

the relevant parts of the client for the resource recommender will be described. Furthermore, a short description of the communication between the client and the server will be given.

While some problems where identified with the OntoAIMS user interface during this project, no significant improvements are realised on this area. The work on the client during this project mainly focussed on integrating the OntoAIMS resource recommender in the current implementation.

The OntoAIMS *viewer* is one of the three clients that are available in OntoAIMS. Figure 2.1 shows the user interface of the client. The user interface is implemented in a seperate package. Communication with the server is entirely done by *agents objects* which are in fact client side implementations of the agents available on the server. This implemenatation is commonly known as the *Proxy Design Pattern*[13]. The various agents are contained in the `client.defImpl` package. This package also contains client side implementations of classes that inherit the classes that are used for communication bewteen the client and the server.

The communication between the `ServerSearchAgent` and the client is managed by the `DefSearchAgent` class. This class contains a method `doSearch()` which sends a search request to the server and receives the result. The query is sent from the client to the server as a `SearchQueryStruct` object and the results are returned in a `SearchResultPack` object. An overview of the relevant packages on the client for the resource recommendation are visible in figure 4.4.

### 4.5.1 `ServerSearchManager`

The user interface of the OntoAIMS *viewer* client is implemented in the `ViewerMainPanel`. This large class contains the application logic which handles mouse clicks and keyboard input of the user. A detailed discussion of the implementation of this class is not relevant for this project, and as such omitted here. This class contains three relevant methods: `doDocSearch()`, `doSearch()` and `doQuickSearch`. The first method is invoked when the user wants to search for relaed documents for a particular resource, the second is used for searching on a concept or a free-text query, and the latter is invoked when the user clicks on a concept name in the domain browser and a *simple recommendation* is requested. The implementation of these three methods is very similar, because the actual logic is implemented in the `ServerSearchManager` class, which is part of the `client.viewer.search` package. In the past, OntoAIMS used the `LocalSearchManager` which implements the search logic on the client. The `ServerSearchManager` is developed during this project and transfer search requests to the server.

The `ServerSearchManger` implements the `SearchManager` interface. The main methods of this class are the following:

- `executeSearch()` is available in two variants, the first accepts a `Document` object and searches for documents which are relevant to the same concepts as the provided document. The second variant accepts a query string and a `DefTask` object which represents the current task. This latter version delegates the search to the server.
- `getRelatedConcepts()` returns the concepts which are the result of the last search action executed using the `executeSearch()` method.
- `getRelatedDocuments()` returns the documents resulted from the last search action.

### 4.5.2 `ConceptPack` and `DocumentPack`

The search results on the client are represented in two classes: `ConceptPack` and `DocumentPack`. The first class contains the concepts that are relevant for the found resources and used by the client to display concepts in the graphical representation of the domain. The `DocumentPack` contains the resources that are recommended by the resource recommender. The `ViewerMainPanel` retrieves these objects when a search action is performed. Both the `DocumentPack` and `ConceptPack` contain a weight for every item. This weight indicates the relevance of the resource or concept with relation to the current search query.

# Chapter 5

# Conclusions and Evaluation

This chapter gives a short summary of the work presented in this document and evaluate the project and the achieved results. Section 5.2.1 discusses the work in relation to the scenarios presented in 1.1.1. Section 5.3 discusses the result of this research project for the research questions proposed in section 1.2. Finally, section 5.4 proposes possible future improvements to the resource recommender and possible research questions which emerged from this project.

## 5.1 Summary

A proposal for a personalized resource recommender using ontology based information models and dynamically retrieving documents from the Internet is presented in this document. The project started by defining problems based on user tests performed during a previous project involving OntoAIMS. By extensively use of metadata available in documents combined with information extraction technologies, the system is able to recommend resources which will match the interests of the user of an adaptive information system for educational purposes. The system achieves this by relating concepts to resources using traditional search technologies. This will solve existing problems with traditional content management systems, such as limited available resources collected by the course creator, errors in the metadata of the resources and a mismatch between the knowledge level of the user and the level of the recommended resources. The system automatically retrieves resources from the Internet using web services, coping with the problem of a limited collection of resources created by the domain author or course creator. On the other hand, collection resources by a course creator is simplified by removing the requirement of manually annotating resources with metadata and mapping resources to the various concepts in the domain, since this is automatically done by the system. When recommending resources to a user, the system will automatically modify the number of resources to the knowledge level of the user.

In section 1.1.2, problems with existing resource recommender systems are listed. In chapter 2 an analysis is presented of these problems in relation to the existing OntoAIMS system along with an approach to solve these problems. Chapter 3 presents the design based on the approach in chapter 2 by integrating search technologies in OntoAIMS and use the available information models in combination with the user model to create sophiticated queries for these search engines. In this chapter the integration of several resource collections is described along with a query refinement process using the user model and the OWL ontology.

An implementation of this design in the OntoAIMS system is presented in chapter 4. Here, the integration of the Lucene search engine, Yahoo WebSearch API and Google SOAP Search API in the

existing OntoAIMS adpative information management system is described. The process of indexing documents and the implementation of a fast process for extracting concepts from resources is described in detail. Furthermore, this chapter explains how the query refinement process is implemented and how the improved resource recommender is integrated in the OntoAIMS client.

## 5.2 Evaluation of this Work

This project's main goal was to investigate the possibilities of personalized resource recommendation in an e-learning environment. Different approaches are possible for personalization of learner resources recommendation. For example, [12] describes a system for retrieving learning resources from distributed e-learning environments using query refinement technologies which incorporates user profile information. However, this system only uses existing metadata formats such as Dublin Core does not retrieve normal web pages from the Internet.

The proposed solution for resource recommendation in OntoAIMS recommends resources by using personalized search. Queries are refined using information available in the user model and extended with related concepts extracted from the domain model available in the system. This way, ambiguous usage of terms in documents are filtered out and the resources recommended to the user are in accordance to the knowledge level of the user. Additionaly, the personalized resource recommender uses multiple resource repositories. By combining the different repositories, the recommendation quality improves. A main repository of information resources is maintained by the system, which collects resources from the other available repositories and include all available metadata with the resources.

Usage of web services for searching information resources on the Internet, opens the adaptive information environment to the Internet and make the large number of resources on the Internet available to the applicaton. This requires careful selection of resources in order to prevent an overload of information resources to the user. This is achieved by creating detailed queries limiting the number of clutter in the search results.

Resources retrieved from the Internet are stored in a local repository on the system which can be queried very fast. Before resources are indexed in this repository, information extraction strategies are used to extract concepts from resources and use these concepts to relate a resource to concepts in the domain.

The implementation in an existing information system requires a deep understanding of the system in order to integrate the various parts in a reliable and reusable way. The OntoAIMS system, which was used as a basis for this project, uses a client-server architecture with various agents providing specific functionality. The resource recommender created during this project was implemented in two seperate agents. One agent implements the local resource repository and the indexing process required for adding additional resources to this repository. The other agent implements the actual recommendation part, consisting of several search engines.

### 5.2.1 Evaluation of Scenarios

In this section we discusses the scenarios presented in 1.1.1 and evaluate the implications for these scenarios after the implementation of the improved resource recommender.

### Scenario 1: Recommendation of Pre-selected Documents

This scenario describes the problem of adding resources to a course or domain by the course creator or domain author. This paragraph contains the most important part of the description of this scenario:

*In existing systems, Andrew can search for the names of the concepts using existing search engines, such as Google or Yahoo!. However, the results will be very broad and not specific to the concept represented in the domain. Also the number of documents the search engine will return, can be very large. Other systems allow to search within a predefined set of documents selected by a course creator. These documents are typically relevant to the course and explain the basics of the concepts. However, these documents are presented in an unordered list or ordered by topic. It is not easy to determine which concepts are related to these documents.*

The local collection of resources maintained by the system annotates the resources with the available metadata. When a resource is added to the local repository, for example by the author of the domain or the course creator, the system will extract concepts from the resource and add these concepts to the metedata of the resource. This way, a resource can be easily related to the concepts in the domain. Furthermore, the indexer used for adding the resources to the local collection of resources, extracts all available metadata from the resource, such as the title, author, a summary and relevant keywords. Because of this, it is not required that the course creator adds the metadata of a resource manually, reducing the time to create a course and the possibility for errors.

### Scenario 2: Automatically Search Documents on the Internet

This scenario describes the problems with searching for resources on the Internet. The main problem is presented in this paragraph:

*Because the author of the domain did not think of more advanced users when collecting documents on copying files, the system has no available resources on the copying of files. Obviously, on the Internet are many resources describing the inner details of the* cp *command and all the available parameters. Imagine a system that automatically searches for these documents on the internet, using the available data in the domain combined with the available information the system has on Simon in a user model. The system can automatically create detailed search queries in order to find documents which match the knowledgelevel of Simon. The resulted documents should be presented in a similar way as the documents already available in the system, and automatically mapped to relevant concepts.*

In the improved resource recommender, the system will automatically search existing Internet search engines when a user wants additional resources. The system uses the user model to refine the query defined by the user. Also, the system is not limited to the resources selected by the course author. In the latest version of OntoAIMS, the user can click on a concept and gets a small selection of relevant resources for a concept. These resources are typically selected by the course creator or the domain creator. To search for additional resources, including resources located on the Internet, the user can give a search command by right-clicking on a concept name in the domain browser and click *search*. The system will search the Internet using for example Yahoo and Google web services, and additionally searches the local repository of documents. Relevant resources are recommended to the user, where the current knowledge of the user and the current activity of the user is taken into consideration.

### Scenario 3: Accumulating Feedback of Users when Recommending Resources

This scenario suggests the possibility of taking feedback of users into consideration when resources are recommended by the system:

*Because the system think the document Andrew was reading, was relevant for the concept* `user`*, the next time another user of the system wants information on* `users`*, the document will be recommended again. When Andrew has the possibility to mark the document is not interesting for him, the system can use this information when another user wants recommended documents for this concept. However, Andrew has a limited knowledge on the concepts in Linux and wants basic information on the concept of* `users`*. Simon, however, wants more in-depth information about the concepts, and may think the document Andrew was reading is relevant to the concept of* `users`*. The system has knowledge of the conceptualization of the domain for every user, and can relate this conceptualization when processing the comment of the user on a document. In this scenario, the system can mark the document Andrew was reading not relevant for users with a low level of understanding on users in Linux, but mark it relevant for* `users` *with a more advanced notion of this concept.*

This scenario is not yet implemented in the OntoAIMS resource recommender, mainly because time constraints. This scenario remains intersting, because it has the potential to improve the resource recommendation feature and makes it possible to personalize the recommendation in a more extensive way.

#### Scenario 4: Reading Documents and Searching For More Information

This scenario consists of two parts, the first is an improvement of the presentation of the resources, the second is on the possibility to use custom queries while searching for relevant resources:

*Because Andrew does not know what* netiquette *are, it is understandable that he wants to get more information on this. The Linux domain, used by the information environment Andrew is using, does not contain a concept on* netiquette*. However, other documents in the system may have information on this subject. Imagine that it is possible to search on the word* `netiquette` *where the system takes the level of knowledge on other subjects of Andrew in consideration. It would be much easier to find relevant documents, than he would search using a standard search engine such as Google or Yahoo.*

Use of custom queries that can be typed by the user was already possible in the implementation of OntoAIMS available before the start of this project. However, that search possibility was very limited allowing only to search on concept names. The improvements made to OntoAIMS during this project allows the user to use free-text queries to search for specific information. These queries are used to search in the local collection of resources, but also searches the Internet for relevant resources.

## 5.3 Evaluation of Research Questions

This section shortly discusses the results of this projec related to the research questions proposed in section 1.2. First, we take a look at the main research questions:

- **Personalizing resource recommendation.** *How can the system recommend resources to the user based on the knowledge of the user and the needs of the user? This requires methods for incorporating the user model for recommending resources.*

  In chapter 3 and 4 the design and implement is described of a personalized resource recommender. Using external search web services providing search possibilities combined with a local search engine, using Lucene, makes it possible to use detailed search queries to retrieve resources. Combining the search possibilities with the available enhanced user model available in for example OntoAIMS, allows a system to recommend resources that are in conformance

with the knowledge level and preferences of the user. Incorporating an enhanced user model using the OWL ontology language makes it possible to get detailed information on the user's conceptualization of the domain. This information is used when refining queries using the concepts in the domain model.

- **Automatically recommend resources from the Internet.** *While there are usually resources available in the system which are added by the domain or course creator, in most cases the number of these resources are limited. How can the sytem acquire additional resources from the Internet (or other information repositories) while ensuring that the resources are relevant for the user?*

  Using external web services based on existing standards gives an application access to external resources from the Internet. Using the query refinement process described in section 3.4.4, the system retrieves information resources from the Internet. These resources are added to the local repository of resources using the indexing process described in section 3.3.3 which annotates the resources with related concepts and extracts the metadata from the concept to make it possible to use this information in the search process. Then, the local repository of resources is used to recommend the resources retrieved from the Internet using Google or Yahoo. This way, the system can provide a high level of personalization in the recommendation process and ensure that a resource is relevant for the task the user is doing and the current concept the user wants to understand.

- **Using ontology information for annotating resources.** *In exsiting adaptive information systems, authors can add information resources to the system, but usually have to annotate these resources with as much metadata as possible. This causes errors in the metadata and often results in incomplete metadata. How can we make this annotation process easier in order to improve the quality of the metadata and as a result improve the resource recommendation?*

  The process of relating concepts to resources, described in section 3.3.5, removes the requirement to annotate resources manually with relevant concepts in the domain model. This approach tries to extract relevant concepts from a resource using search technology combined with aspects of information extraction technologies. This approach processes the document in such a way that it is easy to search a document. The document is searched for every concept in the domain and the alternative names for the concepts. This way, the system can decide if a concept is relevant for a resource. Section 4.2.4 describes the implementation of this idea, using the Lucene `MemoryIndex` class. While there are possibilities for improvements in this concept extraction process, the results are very good. The implementation is quite fast considering the complex nature of the process.

Further, we discuss the other research questions which are not considered main research questions for this project, but are related to this work:

- **Improve the accessibility of information resources.**

  In OntoAIMS, resources are displayed in the default viewer specified for a resource type. Web pages are displayed in a standard web browser, PDF documents in the default PDF viewer and other resource formats in a viewer configured for that specific type of content. This research question focusses mainly on resources which represent the information in a textual way, such as web sites and PDF documents. Especially web pages can be designed in such a way that it is difficult to read them easily, especially for people suffering from color blindness.

– *Is it possible to improve the visual presentation of the information resources using existing technologies?*

  Research is done on extraction of contents from HTML documents, for example by using the *Document Object Model* for eliminating non-content information from the document [17]. Implementing this technology in a system such as OntoAIMS could improve the accessibility of resources. This solution can only be applied to HTML documents, but does not improve the accessibility of other document formats. In he current version of OntoAIMS, no work is done yet on this area. Additional research is needed to define the main problems with the accessibility of resources.

– *How should the user interface of a adaptive information system behave in order to make it easy to use?*

The OntoAIMS implementation has problems with the user interface of the client. Initially, the idea was to improve the user interface as well during this project, but in the end this turned out to be rather complex and as a result no real work is done on this area.

## 5.4 Future Work

### 5.4.1 Improvements to Resource Recommender

This section contains future possible improvements to the OntoAIMS resource recommender and also to system in general. Some of these problems came up during the research and analysis phase of the project, but fell outside the scope of this project. Other issues came up during the development of the resource recommender.

A number of possible improvements in the resource recommender came up during this project, but could not be implemented due to time constraints:

- **Improve concept extraction:** The current implementation of the concept extraction process relates concepts in the domain to resources. However, during this process, the relations between the various concepts in the domain are not used for improving this process. For example, a concept with the name `more` is part of the *Linux Basic Ontology*. This concept is related to many resources which contains the word *more* many times. However, the concept in the domain refers to the `more` file operation, so this concept is related to resources which do not cover this concept. By taking the concepts that are linked to `more`, for example the OWL *class* that is the parent of `more`, this problem could be solved. Additional research to the best approach for this part is required.

- **Filter recommendation results:** The user model is currently only used for refining the queries. However, the information in the user model could also be used to filter the results after the recommendation process, eliminating resources that are to difficult to understand by the user.

- **Improve user interface:** The interface of the OntoAIMS *viewer* is not optimal. User tests during previous projects shows several usability problems with the user interface.

- **Add additional resource repositories:** The current implementation of the resource recommender of OntoAIMS only uses the web search services provided by Google and Yahoo. Additional resource repositories could be used to get additional concepts. For example, social

bookmarking sites or domain specific resource collections which provide an API for accessing the resources could be added to the system to get additional resources with detailed metadata.

- **Consider current task in recommendation:** The system already uses much information available in the system when recommending resources. However, the course structure provides some additional information which could be incorporated in the recommendation process. The *current task* describes the task the user is performing and contains a list of concepts which are relevant for this task. These concepts could be used when refining the query and while indexing resources.
- **Use metadata standards:** The indexer of the resource recommender extracts metadata from documents using rather primitive semantics. Standards such as Dublin Core are not supported.
- **Improve code base of OntoAIMS:** The current implementation of the OntoAIMS system contains a lot of problematic parts. Hardcoded paths on the filesystem and exceptions that are catched in a way making it very hard to debug the system are some of the problems which makes developing on the system a challenging task.

### 5.4.2 Related Future Research

During this project we came up with ideas which could be researched in the future. This is a non-exhaustive list of suggestions with topics related to this project.

- **Transform OntoAIMS in an agent-based architecture:** The current implementation of OntoAIMS is based on an client-server architecture, with the server implemented as a monolithic sysem. Considering the fact that the system is build from seperate components, such as the three information models, a user profile agent, the OWL-OLM system and now the resource recommender, it would make sense to transform the architecture of the application into independent agents with a well-defined interface. Each agent should be provide its own interface with the outside world, possibily using standard based protocols, such as SOAP. To realize this, it is required to decouple the various components of the server and define the functionality of the individual agents.
- **Visualization of domain:** This topic is related to the improvements on the user interface, suggested in the previous section. The *viewer* client of OntoAIMS implements the domain graph which visualizes the concepts within the domain and links between the concepts. This representation does not support all features that are available in the OWL-based domain model. For example, the visualization of the domain does not make distinction between classes and indivuals. Also, in some cases a very large number of concepts is related to a certain concept resulting in a cluttered graph. A complete overhaul of the visualization part should address these issues and could possibly include an improved interface to the resource recommendation part of the client with a more clear relation between the resources and the concepts in the domain.
- **Add social tagging feature to resource recommender:** Tagging recommended resources by the users can improve the recommendation process. During this project, we considered this approach for improving the recommendation process. Due to time constraints, it was decided to not research and implement this feature. Using feedback of the users could be used to improve the recommendation process. However, this would require some major changes to the system. A user interface should be designed for providing a user the possibility to give feedback on a resource. Also, methodologies should be researched to match the various opinions of the users.

- **Improve personalization of resource recommendation:** The user model currently used for the personalized resource recommendation is not updated during usage of the system. The enhanced user model is created by the OWL-OLM system during the *Dialog Game* which probes the knowledge of the user. Updating the user model during the usage of the resource recommendation system and while browsing the domain could improve the personalization of the resource recommendation. Furthermore, there are many possibilities to use the user model more extensively during the recommendation of resources. As suggested in the previous section, the user model could be used to filter the list of recommende resources after the system has searched the various resource repositories. Another possible improvement is to use the user model to add additional terms to search queries, which are not concept names or operators, but define other properties of requested resources, such as if it is a tutorial or reference document.

- **Implement resource recommender in domain and course editor:** The authoring environment of OntoAIMS, consisting of the domain, course and resource editor, does not use the resource recommender. Implementing the recommender system into these environmentts can make the process of selecting resources less difficult. Research needs to be done on this subject. Some work in this direction is already done by another Master of Science project [23].

# Bibliography

[1] Major B. J. Jansen Amanda Spink, Dietmar Wolfram and Tefko Saracevic. Searching the web: The public and their queries. *Journal of the American Society for Information Science and Technology*, 52(3):266–234, 2000.

[2] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 2001.

[3] David Booth, Hugo Haas, Francis McCabe, et al. Web services architecture. Technical report, World Wide Web Consortium, 2004.

[4] P. De Bra, L. Aroyo, and V. Chepegin. The next big thing: Adaptive web-based systems. *Journal of Digital Information, 5(1)*, 2004.

[5] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (wsdl) version 2.0 part 1: Core language. Technical report, W3C, 2007.

[6] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*, 2002.

[7] Ian Jacobs Dave Raggett, Arnaud Le Hors. Html 4.01 specification. Technical report, W3C, 1999.

[8] Jerry Hobbs David Martin, Mark Burstein. Owl-s: Semantic markup for web services. Technical report, 2004.

[9] R. Denaux, L. Aroyo, and V. Dimitrova. Integrating open user modeling and learning content management for the semantic web. Technical report, SWALE Project (Submitted to the UM 2005 conference), 2004.

[10] R. Denaux, V. Dimitrova, and L. Aroyo. Interactive ontology-based user modeling for personalized learning content management. In *AH 2004: Workshop Proceedings Part II*, pages 338–347, 2004.

[11] R.O. Denaux. Ontology-based interactive user modeling for adaptive web information systems. Master's thesis, University of Technology Eindhoven, 2005.

[12] Peter Dolog, Nicola Henze, Wolfgang Nejdl, and Michael Sintek. Personalization in distributed e-learning environments. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 170–179, New York, NY, USA, 2004. ACM Press.

[13] Ralph Johnson Erich Gamma, Richard Helm and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*, chapter Structural Pattern. Addison-Wesley Longman, Inc., 1995.

[14] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000.

[15] Nicholas Gibbins, Stephen Harris, and Nigel Shadbolt. Agent-based semantic web services. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 710–717, New York, NY, USA, 2003. ACM Press.

[16] Google web search.

[17] Suhit Gupta, Gail Kaiser, David Neistadt, and Peter Grimm. Dom-based content extraction of html documents. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 207–214, New York, NY, USA, 2003. ACM Press.

[18] IEEE. Ieee standard for learning technology-learning technology systems architecture (ltsa). Technical report, IEEE, 2003.

[19] Grant Ingersoll. Apache lucene - scoring. Technical report, Apache Software Foundation, 2007.

[20] Susan T. Dumais Jaime Teevan and Eric Horvitz. Personalizing search via automated analysis of interests and activities. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 449–456, 2005.

[21] Tim O'Reilly. What is web 2.0. Technical report, O'Reilly, 2005.

[22] Claus Pahl and Ronan Barrett. A web services architecture for learning object discovery and assembly. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 446–447, New York, NY, USA, 2004. ACM Press.

[23] J.T.J. Rijsdijk. Authoring resources in a concept-based educational system. Master's thesis, University of Technology Eindhoven, 2006.

[24] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.

[25] Wen tau Yih, Joshua Goodman, and Vitor R. Carvalho. Finding advertising keywords on web pages. In *WWW '06: Proceedings of the 15th international conference on World Wide Web*, pages 213–222, New York, NY, USA, 2006. ACM Press.

[26] W3C. Soap version 1.2 part 0: Primer. Technical report, W3C, 2007.

# Appendix A

# Source Code Documentation

All the source code developed during this project has been documented using JavaDoc. The documentation can be found at http://wwwis.win.tue.nl/~swale/javadoc/index.html