Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BSC-2009-03

Bachelorarbeit

im Studiengang "Angewandte Informatik"

Bestimmung relevanter Worte eines Textes und Darstellung unter der Oberfläche TextGrid-Workbench

Ubbo Veentjer

am Lehrstuhl für

Praktische Informatik

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

29. April 2009

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel.	+49 (5 51) 39-1 44 14
Fax	+49 (5 51) 39-1 44 15
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

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

Göttingen, den 29. April 2009

Bachelor Thesis

Determination of Relevant Words within a Text and Representation in the User Interface TextGrid-Workbench

Ubbo Veentjer

April 29, 2009

Supervised by
Prof. Dr. Bernhard Neumair
Dr. Heike Neuroth
Andreas Aschenbrenner

This bachelor thesis deals with the developing process of an Eclipse based plug-in for the TextGrid workbench. To give a better understanding of what a given, presumably unknown text is about, the plug-in supports an easy way to find the most relevant words within the text. Additionally the plug-in extends the TextGrid research facility with a way to look up meanings of search terms entered. Both features are shown in two newly implemented Eclipse perspectives. In each one presented, the user can choose word by word between linked hypernyms and hyponyms. Thus, one is able to effectively navigate through the semantic context. To achieve this the RDF version of WordNet together with a text mining tool are utilized.

Diese Bachelorarbeit beschreibt die Entwicklung eines auf Eclipse basierenden Plug-ins für die TextGrid Workbench. Das Plug-in bietet die Möglichkeit sich die relevantesten Wörter eines Textes anzeigen zu lassen, um einen schnellen Überblick über dessen Inhalt zu erhalten. Zudem erweitert das Plug-in die TextGrid Suchmaske mit der Möglichkeit, sich die Bedeutungen eingegebener Worte anzeigen zu lassen. Beide Erweiterungen werden in zwei neu implementierten Eclipse Perspektiven dargestellt, in denen sich der Nutzer die Hypernyme und Hyponyme der dargestellten Worte anzeigen lassen kann. Das ermöglicht die Navigation durch den semantischen Kontext. Zur Umsetzung werden die RDF Version von WordNet sowie Text Mining Tools genutzt.

Contents

1 Introduction

The TextGrid[1] project is part of the German D-Grid Initiative[2]. It provides tools for collaborative work on texts for scientists and scholars in humanities [3]. Furthermore, it allows to make use of grid resources for storage. The user interface combining the TextGrid tools is called the TextGrid workbench.

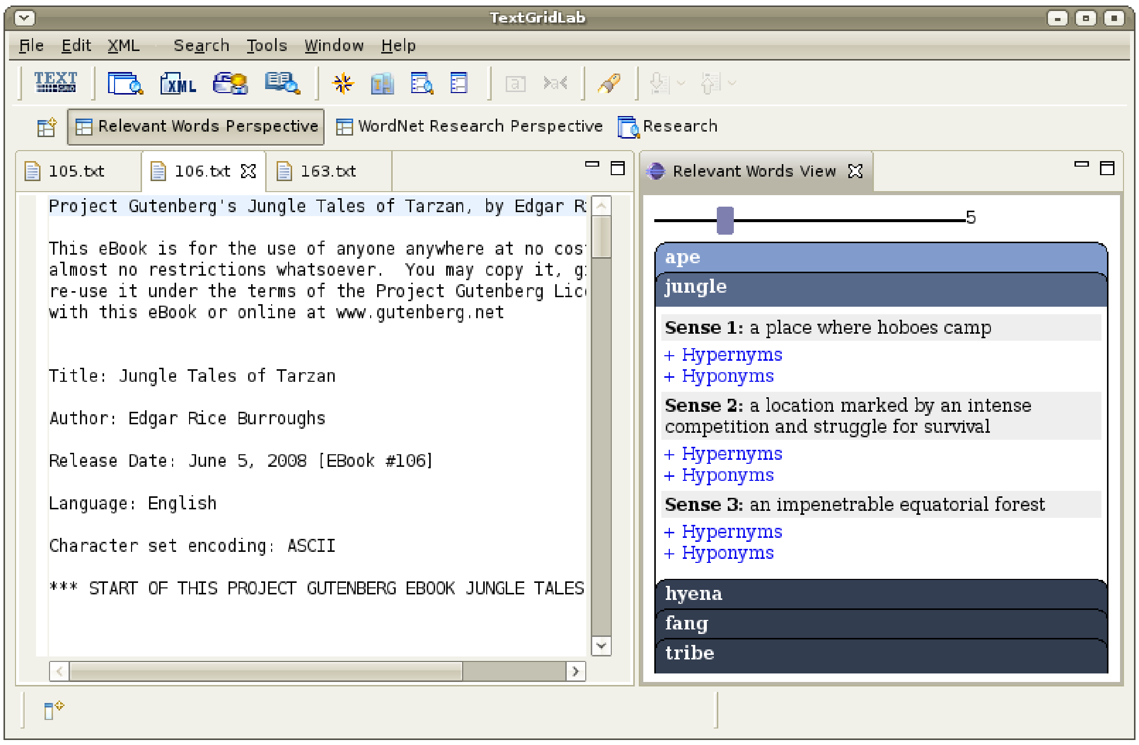
The tools developed in TextGrid till now are mainly focussing philology. Within this thesis research is done on also integrating techniques which come from another discipline of computer science, namely information retrieval. This discipline deals with extracting useful information from large collections of texts. This aims at a different target than the tools integrated in TextGrid till now, which mainly operate on single texts[4].

This thesis describes the development process of two new tools that integrate within the TextGrid-workbench. These are:

1. *Relevant Words View* offers possibilities to view relevant words of a given text and browse their semantic context according to hyponyms and hypernyms[5].
2. *WordNet Research View* enhances the research facility of the TextGrid-workbench. It allows the user to look up meanings as well as hypernyms and hyponyms of the words one enters.

While both tools have a different use-case, they build on the same technology foundation. Both utilize and integrate a semantic network, namely WordNet. They are also based on the same architecture and show the generic applicability of these techniques for the TextGrid project.

[1]`http://www.textgrid.de` (March 2009)
[2]`http://www.d-grid.de` (March 2009)
[3][16], p.1
[4]an exception is tg-search, which allows xquerys over all texts, see 2.5
[5]An example: animal is a hypernym of dog, whereas dog is a hyponym of animal.


TextGridLab
File Edit XML Search Tools Window Help
Relevant Words Perspective
WordNet Research Perspective
Research
105.txt
106.txt
163.txt
Relevant Words View
5

```
Project Gutenberg's Jungle Tales of Tarzan, by Edgar R

This eBook is for the use of anyone anywhere at no cos
almost no restrictions whatsoever.  You may copy it, g
re-use it under the terms of the Project Gutenberg Lic
with this eBook or online at www.gutenberg.net


Title: Jungle Tales of Tarzan

Author: Edgar Rice Burroughs

Release Date: June 5, 2008 [EBook #106]

Language: English

Character set encoding: ASCII

*** START OF THIS PROJECT GUTENBERG EBOOK JUNGLE TALES
```

ape
jungle

Sense 1: a place where hoboes camp
+ Hypernyms
+ Hyponyms

Sense 2: a location marked by an intense competition and struggle for survival
+ Hypernyms
+ Hyponyms

Sense 3: an impenetrable equatorial forest
+ Hypernyms
+ Hyponyms

hyena
fang
tribe

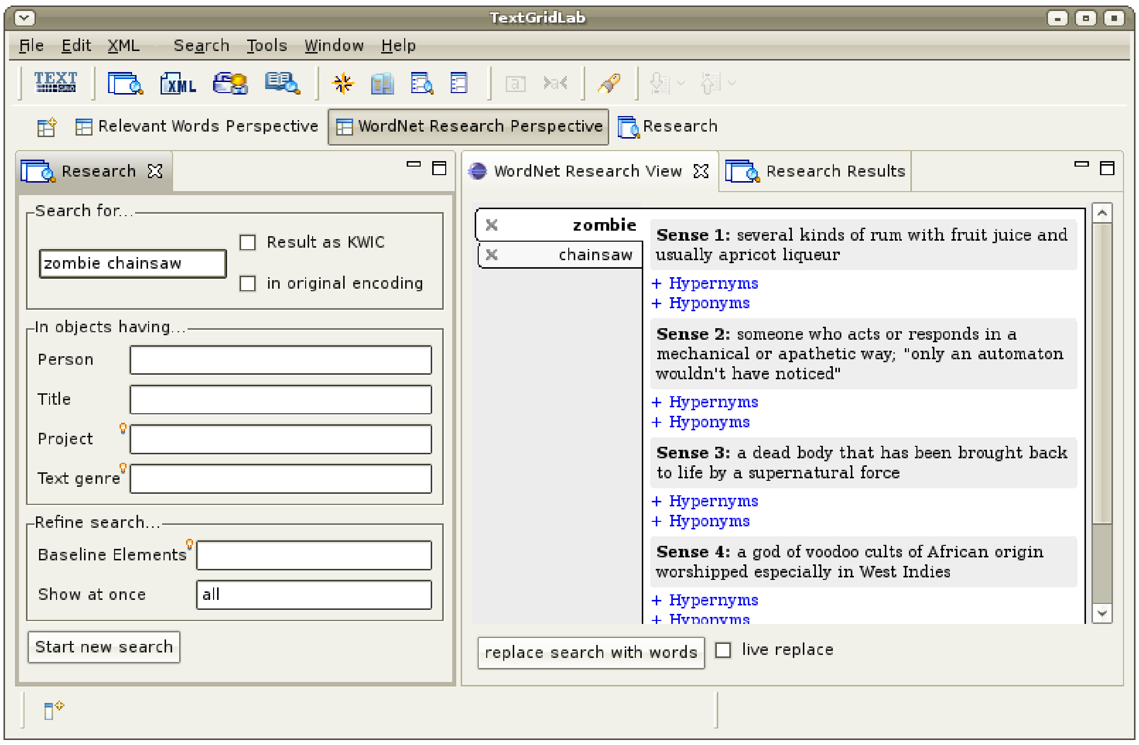
Figure 1.1: Relevant Words Perspective

1.1 Use-Cases

The following two use cases illustrate the benefits of these applications.

1. Alice opens an unknown text in the TextGrid workbench. She wants to get a quick overview of its contents. The sidebar showing the most relevant words of the text gives her a first impression. She clicks on an unknown word and reads its explanation. Browsing the hyponyms and hypernyms helps her to put the words in a semantical context (as shown in 1.1).

2. Bob is looking for a book. His search terms yield too many results. He wants to refine his search. The *WordNet Research View* already shows the words he enters. He is now able to browse through their meanings and add some hyponyms that better describe the document he is searching for (see figure 1.2).


TextGridLab
File Edit XML Search Tools Window Help
Relevant Words Perspective
WordNet Research Perspective
Research
Research
WordNet Research View
Research Results
Start new search
replace search with words

Search for...
zombie chainsaw
☐ Result as KWIC
☐ in original encoding

In objects having...
Person
Title
Project
Text genre

Refine search...
Baseline Elements
Show at once: all

zombie
chainsaw

Sense 1: several kinds of rum with fruit juice and usually apricot liqueur
+ Hypernyms
+ Hyponyms

Sense 2: someone who acts or responds in a mechanical or apathetic way; "only an automaton wouldn't have noticed"
+ Hypernyms
+ Hyponyms

Sense 3: a dead body that has been brought back to life by a supernatural force
+ Hypernyms
+ Hyponyms

Sense 4: a god of voodoo cults of African origin worshipped especially in West Indies
+ Hypernyms
+ Hyponyms

☐ live replace

Figure 1.2: WordNet Research Perspective

1.2 Connection with Roman Hausner's Thesis

Roman Hausner has been working on Web services for text mining at the same time this bachelor thesis was written [12]. In his thesis he accentuates a Web service called *preprocessing service*, which uses different reduction services to be capable of computing information about relevant words of a text in a corpus.

An overview of how the services are interacting and where both theses integrate with each other is available in figure 1.3. More technical details can be found in 3.1.2.

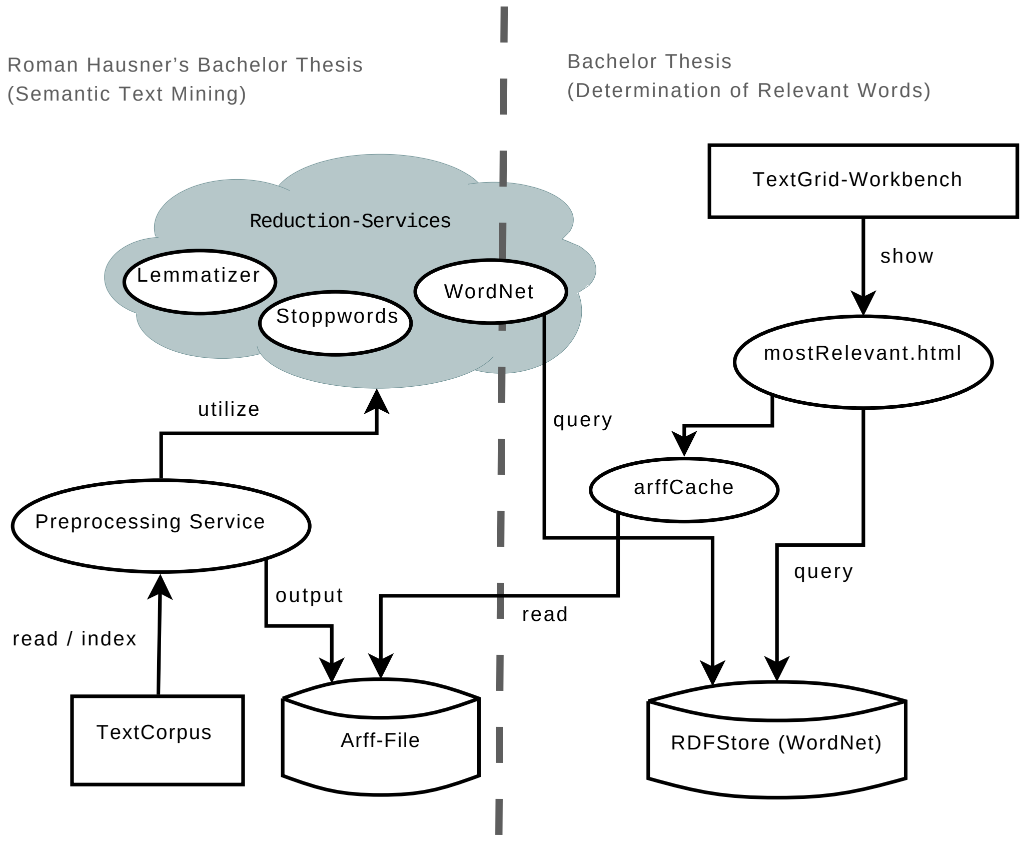


Figure 1.3: Interaction of both theses

1.3 Thesis Organization

The thesis is divided in five parts. Chapter two introduces basic theories and techniques the applications are build upon. A model for the determination of relevant words is presented. After the theoretical background is laid out, chapter three deals with the design of the overall architecture. The requirements are defined and fitting technologies are chosen on this basis. The actual implementation is described in chapter four. This starts with the needed Web services used by the web clients. Those clients are then integrated within the TextGrid workbench. The last chapter explains where to download and how to use the applications. Then a short evaluation is done. Some ideas for further enhancements of the applications are given, finally a conclusion is drawn.

Source code and function names are accentuated like `Java.lang.Object` if they appear in text.

2 Background Knowledge

This chapter imparts background knowledge about concepts and terms further used. Section 2.1 is important for the *Relevant Words View* whereas the other sections give an overview about techniques used in general.

2.1 Determining the Relevance of a Word

To determine the relevance of a word, concepts of information retrieval are used. Information retrieval deals with the issue which index terms (keywords) describe a document best. Different models have been developed in this area, such as boolean model, vector model, and probabilistic model.[1] The common idea is the application of a weight to each index term that should "quantify the importance of the index term for describing the document semantic context."[2] This weight captures the relevance of a word for a given text.

In this thesis the vector model is used because, as stated by Baeza-Yates and Ribeiro-Neto: "A large variety of alternative ranking methods has been compared to the vector model, but the consensus seems to be that, in general, the vector model is either superior or as good as the known alternatives. Furthermore, it is simple and fast. For these reasons, the vector model is a popular retrieval model nowadays."[3]

Vector Model / tf-idf

In the vector model, documents are represented as vectors, where every term is a dimension in a vector space. Important for the determination of the relevance of a word is the method used to obtain index term weights.

An often used weighting method is *tf-idf* (term frequency - inverse document frequency). The *tf-factor* hereby stands for the frequency of a term within a text. It gives an indicator how well the term describes the documents contents.[4] It is calculated with the formula

[1]compare [4] , p. 24.
[2][4], p. 25
[3][4], p. 30
[4][4]., p.29

$$tf_{i,m} = \frac{freq_{i,m}}{\max_l freq_{l,m}}$$

where $freq_{i,m}$ is the number of occurrences of a specific term i in the document d_m and $\max_l freq_{l,m}$ the number of occurrences of the most used term in this document. [5]

"The motivation for usage of an idf factor is that terms which appear in many documents are not very useful for distinguishing a relevant document from a non relevant one." [6]

The inverse document frequency (*idf*) indicates the relevance of a term within the whole corpus. It is defined by the formula

$$idf_i = log\frac{N}{n_i}$$

with N being the number of all documents in the corpus, and n_i the number of documents with an occurance of the term i.

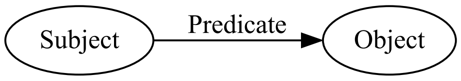
The weight w of of Term i in document d_m , described in *tf-idf* is now:

$$w_{i,m} = tf_{i,m} \cdot idf_i = tf_{i,m} = \frac{freq_{i,m}}{\max_l freq_{l,m}} \cdot idf_i = log\frac{N}{n_i}$$

2.2 WordNet RDF

RDF and the Semantic Web

The Semantic Web[7] provides a set of techniques for interlinking data on the World Wide Web. It is based on RDF, a graph description language working with triples (also known as statements). These are stated in the form of "subject predicate object" (spo). The statements form a directed graph, where the subjects and objects are the nodes, and the predicates the edges. [5]



[5] See [11] , p. 23
[6][4], p. 29
[7]`http://www.w3.org/2001/sw/` (March 2009)

Subjects and predicates are expressed as URIs (Unique Resource Identifier). Objects may be URIs or literals, describing the subject. The unique character of URIs allows the concatenation of statements to larger graphs.

The W3C[8] standardized query language for RDF graphs is SPARQL. It is used to query RDF graphs across diverse data sources. The result can be returned as SPARQL result or in RDF [17].

WordNet

WordNet [7] is a free and publicly available semantic network that consists of a large lexical database of English. "Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations."[18] Some common semantic relations WordNet defines are hyponyms (where A is subordinate of B and A is kind of B) and hypernyms (where A is superordinate of B).

The W3C offers a publicly avavilable free RDF version of WordNet to download.[9]

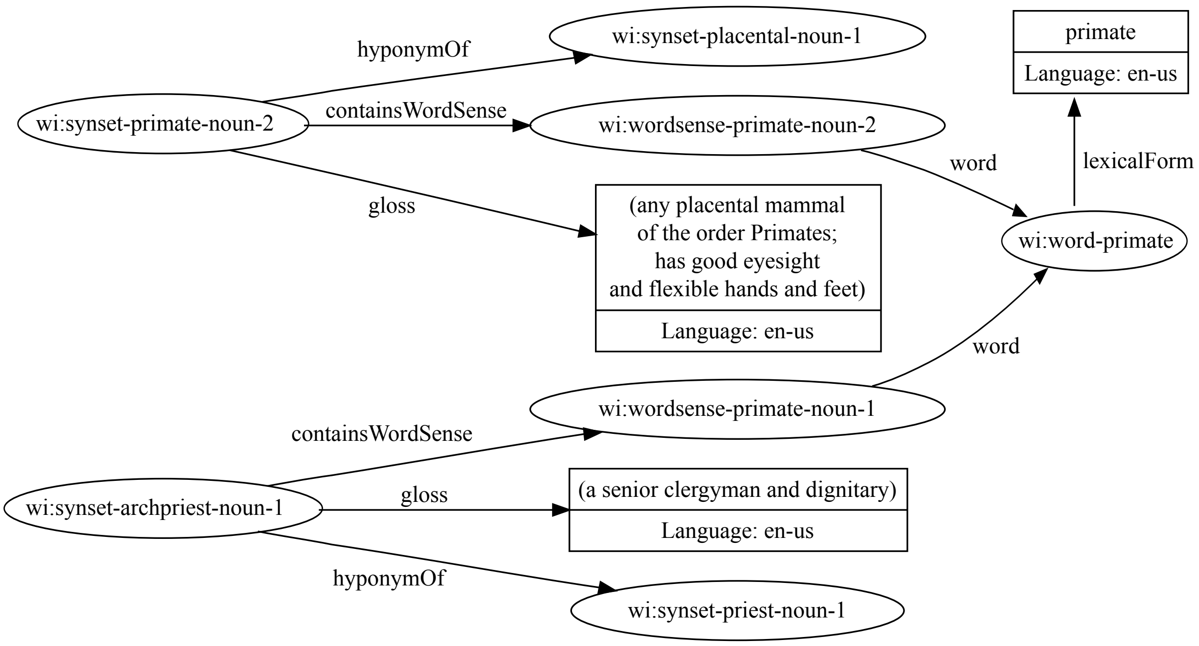

hyponymOf
wi:synset-placental-noun-1
primate
Language: en-us
wi:synset-primate-noun-2
containsWordSense
wi:wordsense-primate-noun-2
word
lexicalForm
gloss
(any placental mammal of the order Primates; has good eyesight and flexible hands and feet)
Language: en-us
wi:word-primate
word
wi:wordsense-primate-noun-1
containsWordSense
wi:synset-archpriest-noun-1
gloss
(a senior clergyman and dignitary)
Language: en-us
hyponymOf
wi:synset-priest-noun-1

Figure 2.1: Graph showing relations for the word *primate@en-US* (WordNet RDF)

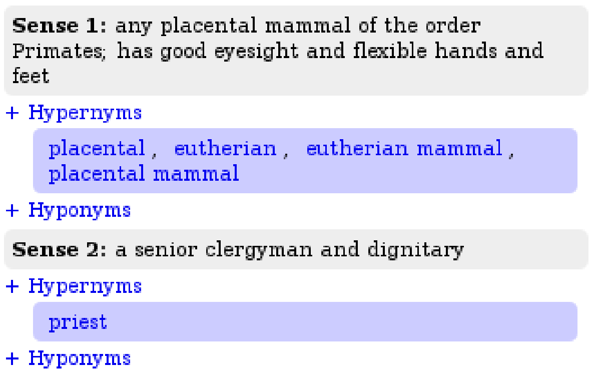
[8] `http://www.w3.org/` (March 2009)

[9] `http://www.w3.org/2006/03/wn/wn20/download/` (March 2009), [8] this version is used throughout the thesis

To demonstrate how words are connected in WordNet (RDF), in figure 2.1 the relations to the literal *primate@en-US* are shown. The word *primate* has two different meanings (synsets):

1. has the hypernym *placental*
2. has the hypernym *priest*

Figure 2.2 shows how the WordNet web clients implemented in this thesis represent the same graph. The wordsenses are shown, but the hypernyms of the synset *primate* seem to be more. That is because the synset *placental* has more than just one wordsense. The web client lists all wordsenses of the synset.



Sense 1: any placental mammal of the order Primates; has good eyesight and flexible hands and feet

+ Hypernyms

placental , eutherian , eutherian mammal , placental mammal

+ Hyponyms

Sense 2: a senior clergyman and dignitary

+ Hypernyms

priest

+ Hyponyms

Figure 2.2: Wordsenses for the word *primate* shown in web client

2.3 Web Services, REST, Ajax and JSON

Web services as defined by the W3C "provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. [...] They can be combined in a loosely coupled way in order to achieve complex operations. Programs providing simple services can interact with each other in order to deliver sophisticated added-value services."[10] There are two main architectures for Web services. The so called "Big Web services" based on SOAP/WSDL [13] [6] and the newer RESTful Web services.[11] Big Web services are often found in enterprise environments whereas RESTful Web services are "gaining increased attention not only because of their usage in the Application Programming Interface (API) of many Web 2.0 services"[12]. RESTful Web services can be used with less communication overhead than SOAP based Web services, especially if they use JSON[13] for communication. JSON "is a lightweight data-interchange format. It is easy for humans to read and write."[1]

Also a RESTFul API is better integrated with websites using Ajax. The term Ajax was invented by Jesse James Garrett in an article in 2005 [9], and stands for "Asynchronous JavaScript and XML". In the article he describes techniques to update the content of webpages

[10]`http://www.w3.org/2002/ws/Activity` (March 2009)
[11]compare [15], p.1
[12][15], p.1
[13] `http://www.json.org` (March 2009)

asynchronously. In contrast to traditional webpages, where all content displayed is loaded at once, this allows updating parts of the page on demand (e.g. if the user clicks a link). This is done by JavaScript requesting a server for data to be inserted. The response may be in XML or JSON and is shown on the page afterwards.

2.4 Eclipse / Plug-in-Technology

Eclipse is a framework that allows the development of cross platform applications. It is implemented in Java and offers core-functionality for many aspects of a modern software product[14]. It is based on plug-ins that expand the core.

Plug-ins specify different additions to the workspace, like perspectives, views and actions. A perspective is an arrangement of views. It can define positions and sizes of views.

Widgets in Eclipse are written in a GUI-toolkit named SWT[15].

2.5 The TextGrid Workbench

The TextGrid project "aims to create a community grid for the collaborative editing, annotation, analysis and publication of specialist texts"[2].

Its architecture for achieving this can be described in four layers:

1. the user interface, called the TextGrid workbench, allowing the client side usage of the tools / services distributed within TextGrid
2. the services, which provide core services as authentication and research in archives as well as e.g. tools for linguistic processing of texts
3. the middleware abstracting the grid infrastructure
4. the archive layer which holds the data that is distributed on different servers (the grid).

The TextGrid workbench is used to utilize TextGrid services and tools, as well as accessing texts stored in the grid. It is an Eclipse based Rich Client Platform (RCP), which allows functionality to be integrated in form of Eclipse plug-ins. These plug-ins often act as frontends for Web services. They are also the entry point for third party applications to be added to the workbench.[16]

[14]like a workbench offering perspectives, views, editors, an update manager, a gui-toolkit...

[15]`http://eclipse.org/swt` (March 2009)

[16]A list of tools already distributed with the TextGrid workbench is available on the download page for the beta version: `http://www.textgrid.de/en/beta.html` (March 2009)

The workbench offers a research tool, which allows searching for and within texts in the archives. It provides a full text search as well as fields for specifying e.g. author or title of a text. As the TextGrid archives mainly consist of XML files, it also offers a way to retrieve specific elements of a text with *XQuery*[17].

[17]`http://www.w3.org/TR/xquery/` (March 2009)

3 Design Choices

In this chapter the requirements for the project are determined. The emphasis is to find possible technologies for the implementation and to develop the overall architecture.

3.1 Objective and Requirements

Two objectives are targeted:

1. The user should see the relevant words of a given text and get more information on the words.
2. One should be offered a way to look up search terms while typing in the free text search field of the research perspective.

The solution to be implemented should integrate well into the TextGrid workbench. So three requirements become apparent:

- a thesaurus for looking up words
- a way to determine the relevant words, and match them against the thesaurus
- integration into the TextGrid workbench

Furthermore, because of the nature of the TextGrid architecture, the text is presumably stored in the grid. The solution should take this into account. The information about relevance should be computed server side, and in consequence also be stored there. It makes sense to store the thesaurus online and provide a way to request needed contents from clients. This leads to the next requirement:

- client / server architecture

The main target for TextGrid is working with historical German texts. As words, meanings, and language in general change over the time, thus, the solution should be flexible. It should be easy to change the thesaurus used without the need to rewrite the whole implementation. The application should be able to be extended to other languages. This should be achieved by

- modularity and extensibility.

The user interface should be easy to use, so that the user becomes familiar with it quickly.

Performance is also a factor of usability. Having to wait for results has a negative impact on the user experience. To sum up the solution should provide

- an intuitive GUI
- performance

3.1.1 Further Requirements

Open Source

One important prerequisite for all considered solutions is the availability of all of its components as Open Source, because everything developed should fit into the TextGrid-Project, with the possibility of further use.

Client Side

The only assumption about the client is a running TextGrid workbench, so Java and a sufficient amount of RAM are installed, also network access is given. For the kind of network connection the lowest assumption is to make, as for some work in the lab broadband access is needed, but it is still possible to use the lab e.g. on a train via GPRS-connection, for working on local files.

A main difference between this service and e.g. a file retrieval service in TextGrid is, that the user may understand having to wait a minute when a file is loaded from the grid, but, as already mentioned, not just to look up a word.

Server Side

In TextGrid the assumption may be made that server side infrastructure is on a high level. Solutions that even make use of the Grid for distributed- and/or cluster-computation would be possible, but out of scope for this thesis. Therefore the solution should make realistic use of available technology, which means, RAM and computing-time is still a limiting factor. The needed server-side implementation should be resource efficient.

3.1.2 Meeting the Requirements

WordNet

A thesaurus can be seen as a graph, where the words are encoded as nodes, and their relations as edges. So it makes sense to use a RDF based thesaurus, which can be queried with SPARQL. This adds the flexibility that another RDF based thesaurus can be dropped in. In this case at most the SPARQL queries would need to be modified, while the rest of the architecture could be sustained. If using the standardized SKOS-RDF[1] format for thesauri, even the SPARQL queries can stay the same, only the data set pointed to would change. Another benefit of using a RDF thesaurus is the availability of unique identifiers for words.

The thesaurus of choice is the English WordNet, as it is available in RDF and can be used without license issues. Actually, WordNet is more than a thesaurus; it is a semantical network. The design of WordNet is a kind of archetype for semantical networks in other languages[2], like e.g. GermaNet[3]. GermaNet is released under a far more restrictive license[4]. As this thesis is more the development of a prototype, WordNet is sufficient for testing technologies. Other formats like SKOS or other semantic networks in other languages are embeddable with ease, if available in RDF.

Determining Relevant Words

Roman Hausner's *preprocessing service* mentioned in chapter 1.2 is used to determine the relevant words. It allows to

- upload a corpus of texts bundled as zip file
- use a regular expression to qualify a word
- specify a list of reduction services to use
- choose one or more normalizations for the data to be returned.

The output of the preprocessing service is a zip file containing one or more ARFF[3] files, depending on the chosen normalizations. One normalization option is tf-idf. [5]

Roman Hausner also developed a reduction service template. It can be used for setting up own reduction services by implementing its interface method `reduce`. Every reduction service takes a vector of words as input and returns a new vector of words. Also returned is

[1]`http://www.w3.org/2004/02/skos/` (March 2009), [14]
[2]an overview can be found at `http://www.globalwordnet.org/gwa/wordnet_table.htm` (March 2009)
[3]`http://www.sfs.uni-tuebingen.de/GermaNet/` (March 2009)
[4]`http://www.sfs.uni-tuebingen.de/GermaNet/licence.html` (March 2009)
[5]compare [12], chapter 3.4 and 3.5

some mapping information, holding which word of the old vector resides on which position now.[6]

This reduction service template is used in this thesis to implement a reduction service that utilizes WordNet. The incoming word vector is mapped to WordNet URIs per word, if available. The generated statistics file of the workflow consisting of stopword-, lemmatizer- and wordnet-reduction is used in this thesis to determine the relevant WordNet URIs of a given text.

Text Corpus

A set of plain English text files is downloaded from the Gutenberg-Project[7]. This project offers digital versions of books where the copyright has expired. Digital versions are submitted by volunteers. The downloaded texts are put together in a zip-file. This is the base corpus to feed the web services to gain information about the relevant words.

Eclipse Plug-in

As the TextGrid workbench is Eclipse based and the views to be implemented need to interact with other views, the solution should be implemented as an Eclipse plug-in. As for usability and flexibility it is chosen to implement the representation of the functionality in HTML/JavaScript/CSS. This brings the benefits of not being bound to SWT for the GUI elements. CSS/HTML adds the freedom of quickly prototyping a GUI, which is still easy to redesign and modify. Implementing the logic in JavaScript adds a lot of flexibility. So the representation and the internal functionality (say: another SPARQL Query for another thesaurus) may change, without the user being required to download a new plug-in. The Eclipse part of the plug-in only consists of the glue between the browser and the rest of the lab, e.g. for communication between the views.

3.2 Defining the Architecture

After clarifying the fundamentals of the application to build, the overall architecture can be developed. It is basically split in two parts: server and client side, as visible in figure 3.1. Located on the server are the Web services and the Web service clients, whereas the client consists of a plug-in for the TextGrid workbench. A more detailed explanation of the single modules building the architecture follows.

[6]compare [12], chapter 3.3.2

[7]`http://www.project-gutenberg.org` (March 2009)

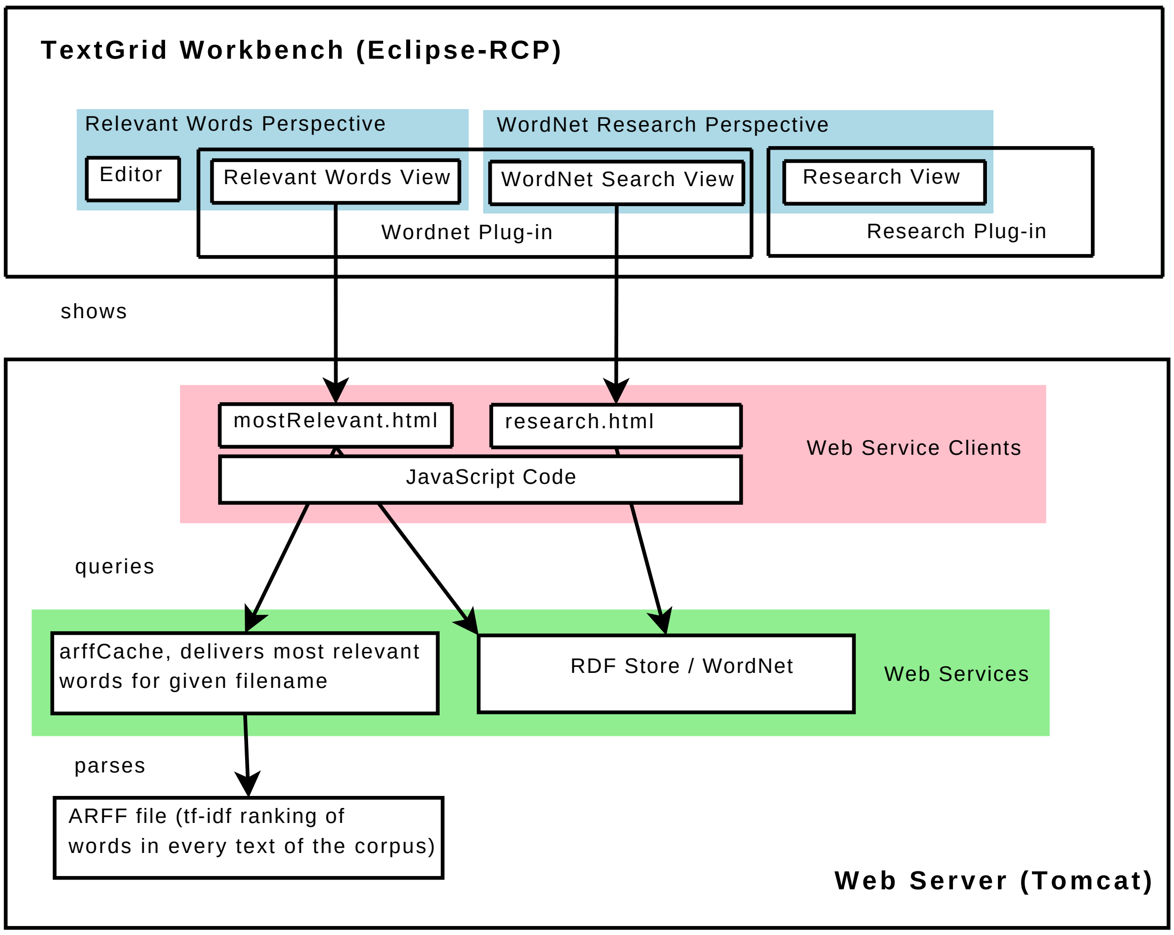


Figure 3.1: Overview of the architecture

3.2.1 Web Services

TextGrid mainly uses SOAP for communication between services and the workbench. For the services in this thesis the implementation of a REST based approach using JSON[8] for communication can be adopted, as they have a different use case. As already mentioned, the "real time user experience" is important, while the data transferred needs neither authentication nor a high level error handling. The data is just needed instantly, when the user clicks a link, even with limited bandwith. JSON has the benefit of being more lightweight than SOAP / XML for data transfer.

[8] `http://www.json.org/` (March 2009)

RDF Store

As it was decided to use the RDF version of WordNet, it should be made queryable with SPARQL. This is typically done by loading the data into a RDF store. There are different options like Sesame[9], Mulgara[10] or a Jena based one. In this thesis a Jena based triplestore is used, which is already included in the TextGrid subversion repository. Jena is a framework for building Semantic Web applications. It can deal with RDF, OWL, SPARQL, and supports reasoning.[11] Jena offers different options for persistance, namely SDB and TDB being optimized for SPARQL queries. SDB uses a relational database as backend. TDB is a newer development that can be used "as a high performance, non-transactional, RDF store on a single machine."[12]

ArffCache

The text corpus is indexed by the preprocessing service mentioned above. The resulting ARFF file containing the word statistics is stored on the server. For using it from a client a Web service is needed to hand out the most relevant words of a given text on demand. Loading and parsing the ARFF file takes time. So this service should keep the contained data in memory for instant access. This means the service acts as cache for the ARFF file, allowing queries for the relevant words of a specific text contained in the corpus.

3.2.2 Web Service Clients

Two Web service clients are also located on the web server. They offer a user interface written in HTML/CSS, with the programming logic written in JavaScript. So they are downloaded from the server, but executed by the client. The web service clients are:

1. *mostRelevant.html*, sending requests to arffCache for showing the most relevant words of a specific text. It also queries WordNet to display different meanings of the relevant words and allows to dig into hyper- and hyponyms.
2. *research.html*, querying the RDF store (WordNet), showing the different meanings of a word within WordNet. It also offers the possibility of looking up their hyponyms and hypernyms.

[9] `http://openrdf.org/` (March 2009)
[10] `http://www.mulgara.org/` (March 2009)
[11] `http://jena.sourceforge.net/` (March 2009)
[12] `http://jena.hpl.hp.com/wiki/TDB` (March 2009)

Both use SPARQL to query the RDF-Store and retrieve responses in JSON.

HTML is used for the representation of the GUI elements, whereas CSS is styling the elements. CSS adds the flexibility of changing the style of the GUI without changing other source code.

Web service requests are done in JavaScript utilizing the XMLHttpRequest (Ajax). Responses are parsed and new elements added to (or removed from) the user interface.

3.2.3 Plug-in for the TextGrid Workbench

As already mentioned the TextGrid workbench additions are distributed in the form of an Eclipse plug-in, called *WordNet Plugin*. It offers two views, each of them having a web browser as the main widget:

1. *Relevant Words View* shows *mostRelevant.html* from the webserver
2. *WordNet Research View* displays *research.html*.

The views are combined with other views in two new perspectives:

1. *Relevant Words Perspective* combining the *Relevant Words View* with an Eclipse editor. Opening a file in the editor triggers the look up of the filename in *mostRelevant.html*, displaying the relevant words for the filename if found.
2. *WordNet Research Perspective* displaying the *Wordnet Search View* and the *Research View* from the *Research Perspective* distributed with the TextGrid workbench. If a search term is entered in the free text search field of the *research view* it is looked up and shown in *research.html*. It allows replacing words from the free text search field with the words shown in the web client.

Both perspectives can be opened in the TextGrid workbench after downloading and installing the *WordNet plug-in*. How to do this is explained in section 5.1.

4 Implementation

After deciding on technologies and the overall architecture, this chapter deals with the actual implementation.

4.1 Preparation

Development Environment

Tomcat[1] (version 6.0.18) is running on the server *textgrid-ws2.gwdg.de*. It is used as a Java web application server. Web services are deployed there. Axis2 is installed in tomcat for hosting Axis2 Web services related to the *preprocessing service*.

Development is done in the Eclipse IDE. The web service clients are tested in Mozilla Firefox[2], with the Firebug extension installed[3]. Firebug allows amongst other things debugging of Ajax requests and DOM manipulation with JavaScript.

Build Files

Maven[4] is used as build tool, which also resolves dependencies on needed jar files. The build configuration is done in pom.xml. Maven offers different project templates. The webapp-template for example sets up the directory structure for a Java web application. This allows packing as a *war* file to be deployed on a Java application server (Tomcat in this case). Maven also has a plug-in to create Eclipse project files, with all library dependencies resolved.

[1]`http://tomcat.apache.org/` (March 2009)
[2]`http://www.mozilla.com/en-US/firefox/firefox.html` (March 2009)
[3]`http://getfirebug.com/` (March 2009)
[4]`http://maven.apache.org/` (March 2009)

4.2 Web Services

As already mentioned in section 3.2.1 two Web services are implemented, the RDF store and arffCache. Before arffCache can be used, a statistics file, the ARFF file, is needed. This section describes setting up the Web services and how statistics about relevant words are obtained[5].

4.2.1 WordNet / RDF Store

The RDF store is checked out from TextGrid subversion[6]. It provides a REST endpoint for SPARQL queries and is able to respond in JSON, if requested. It can be configured to use TDB or SDB as persistance backend. Because of the better performance of TDB, this backend is used within this thesis.

The RDF version of WordNet is downloaded from the W3C[7] and imported in the RDF store. As the RDF store has a TDB backend this can be done with tdbloader [8] from the TDB distribution. For the purpose of this thesis the following subset of the WordNet RDF download is sufficient, and imported:

- wordnet-synset.rdf
- wordnet-wordsensesandwords.rdf
- wordnet-glossary.rdf
- wordnet-hyponym.rdf

The total import has an amount of 1.864.697 triples according to tdbstats. Afterwards statistics for the stats based optimizer [9] are generated with bin/tdbstats from the TDB distribution. This speeds up execution time for SPARQL queries.

4.2.2 Determining the Relevant Words

The *preprocessing service* mentioned in 3.1.2 is used to get the tf-idf ranking of WordNet words in the corpus. So an own *reduction service* which maps the words to WordNet URIs is needed.

[5]compare with figure 1.3
[6]`https://develop.sub.uni-goettingen.de/repos/textgrid/branches/rdfstore/`(March 2009)
[7]`http://www.w3.org/TR/wordnet-rdf/` (March 2009)
[8]`http://jena.hpl.hp.com/wiki/TDB/Commands#tdbloader` (March 2009)
[9]`http://jena.hpl.hp.com/wiki/TDB/Optimizer` (March 2009)

To set up the *reduction service* the *reduce service template* is checked out from subversion. The ant build file is modified[10]. `deploy.serverandport` is set to textgrid-ws2.gwdg.de:9090. `service.name` is set to `WordnetReduce`, the `service.class` to `info.textgrid.wordnet.Reduction`. Now the class `Reduction` can be implemented. It extends the class `hausner.bac.reduce.VectorReduceServiceSkeleton` and therefore needs to offer a method reduce, which returns a string for a given input string. It sets up a HTTP client to execute a SPARQL query to the RDF-Store. The query asks for the WordNet-URI of the given input. The result is returned.

Now ant can be executed to build this service. The resulting service archive gets deployed in Axis2 on *textgrid-ws2.gwdg.de*. The service endpoint is now located at `http://textgrid-ws2.gwdg.de/axis2/services/WordnetReduce` and ready for usage from the preprocessing service.

Now the server side preparations are finished and a client for the *Preprocessing Service* can be implemented. To do so, wsdl2java from the Axis2 distribution is executed with the URL of the preprocessing service WSDL [11] to generate the service client stubs. These are used in a new Java class with a main-method, that initializes the stub and sets the parameters:

- reduceEndpoints
- normalization of the return matrix
- file

reduceEndpoints takes an array of endpoint-addresses for reduction services. Three services are executed in a specified order. First *StopWordReduce* is called to remove irrelevant words [12]. Then *StemmerReduce* returns the elementary form of the given word [13]. Finally WordnetReduce comes into action.

The resulting matrix is set to tf-idf using the normalization parameter. The location of the zip file containing the corpus can be declared using the file parameter. As a last step the query execution is triggered through the PreprocessService method.

Running `main()` returns an ARFF-file, say output.arff, with information about the tf-idf values per word (WordNet URI) of each text found within the zip-file.

[10]according to [12], p. 34, chapter 4.7.2
[11]http://textgrid-ws2.gwdg.de/axis2/services/PreprocessingService?wsdl
[12]e.g. and, or, the,...
[13]e.g. car for cars

4.2.3 arffCache

For implementing arffCache JAX-RS [10] is used. JAX-RS provides a convenient way of describing RESTful Web service endpoints with Java annotations[14]. This leads to better readability and, thus, better maintainability of the source code. Further JSON output is already contained in the specification. The JAX-RS implementation used here is CXF[15].

Maven is used to generate the basic structure of a Web service. The resulting directories and files created are shown below. The file pom.xml is modified to reflect the dependencies of this project, which should be a JAX-RS Web service using *Weka*. Maven can now keep track of the dependencies and download needed jars automatically. Weka.jar is not available in the maven-repositories, so it needs to be downloaded [16] and manually added to the local maven repository.

```
arffcache
  |-- pom.xml
  `-- src
      `-- main
          |-- resources
          `-- webapp
              |-- WEB-INF
              |   `-- web.xml
              `-- index.jsp
```

Maven is able to create Eclipse project templates, setting dependencies and Eclipse configuration automatically. After template generation the resulting project is imported into the Eclipse workspace. The class info.textgrid.arffcache.Rest is added, which is the entry point for the web clients implemented later on.

Rest.java is a JAX-RS annotated web service. The method `getMostRelevant()` is added, providing the following functionality:

- reacting to GET-Requests
- taking the textfile to be queried for and the number of words to be returned as parameters
- returning a JSON encoded String

ArffUtils utilizes Weka, which offers functionality for working with ARFF-files and dealing with the contained data. The method `getRelevantWords` gets the row from the data belonging to the filename queried for. With the help of TiValue, which implements a way to compare

[14] `http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html` (March 2009)
[15] `http://cxf.apache.org/` (March 2009)
[16] `http://www.cs.waikato.ac.nz/~ml/weka/` (March 2009)

tf-idf values, the `numWords` highest tf-idf rankings are extracted. A `JSONObject` is filled with key-value pairs, being the WordNet-URI as key, and the tf-idf-ranking as value. This object is returned.

Finally the Web service is configured with the file web.xml. It is modified to declare how the web-context root ("/") should be handled, more details are defined in beans.xml. The file beans.xml needs to be created, including the information that the class `info.textgrid.arffcache .Rest` should act as a REST endpoint.

```
arffcache
  |-- pom.xml
  `-- src
      `-- main
          |-- java
          |   `-- info
          |       `-- textgrid
          |           `-- arffcache
          |               |-- ArffUtils.java
          |               |-- Rest.java
          |               `-- TiValue.java
          |-- resources
          `-- webapp
              `-- WEB-INF
                  |-- beans.xml
                  `-- web.xml
```

The final directory structure is shown above. Now maven is run again to create the arrf-Cache.war. After deploying this on the server, the endpoint can be tested:

```
http://textgrid-ws2.gwdg.de/arffcache/service/relevantWords/163.txt/2
```

the JSON response is:

```
{
   "http:\/\/www.w3.org\/2006\/03\/wn\/wn20\/instances\/word-fairy" : 0.26900176663175673,
   "http:\/\/www.w3.org\/2006\/03\/wn\/wn20\/instances\/word-elf" : 0.433097815666239
}
```

Read "elf" is most relevant word in 163.txt with an tf-idf value of 0.43 .

4.3 Web Clients

With the necessary Web services in place, the Web service clients can be implemented. These are also located on the web server, but they are mainly written in JavaScript.

A JavaScript framework, namely MooTools[17], is used. JavaScript frameworks in general hide the complexity of cross browser JavaScript development, and provide utility methods for Ajax communication and DOM manipulation. MooTools consists of core functions for Ajax requests, adding and removing elements from the webpage. It also offers methods for creating effects. It also provides some widgets based on the core, like the accordion.

4.3.1 Creating Web Client Project Structure

A new web project is created with maven, called "wordnetWebclients". The resulting directory structure is the same as already shown in section 4.2.3. The *MooTools core* JavaScript library and *MooTools More* including the features Color, Slider and Accordion are downloaded. The retrieved libraries `mootools-1.2.1-core-nc.js` and `mootools-1.2-more.js` are placed in a new directory named js below the webapp directory. Further directories created are `css`, which contains CSS files and `img` for images. The directory structure is now:

```
wordnetWebclients
  |-- pom.xml
  ‘-- src
      ‘-- main
          |-- resources
          ‘-- webapp
              |-- WEB-INF
              |   ‘-- web.xml
              |-- css
              |-- img
              |-- index.jsp
              ‘-- js
                  |-- mootools-1.2-more.js
                  ‘-- mootools-1.2.1-core-nc.js
```

[17] `http://mootools.net/` (March 2009)

4.3.2 mostRelevant.html

A new file `mostRelevant.html` is created and placed in the `webapp` folder. It contains a web client combining the data available from the WordNet RDF store and arffCache. It is a basic XHTML[18] file which defines two boxes in the body section. These are handled by JavaScript later on (see figure 4.1). One contains a MooTools slider[19] for adjusting the number of relevant words shown. The other one is used to display the relevant words. The list of relevant words is handled by a MooTools accordion[20], only showing one word explanation at once, while the others are collapsed (compare with figure 4.1). In the header section of the HTML file the JavaScript libraries (mootools, mostRelevant.js) and a CSS file (mostRelevant.css) are included.

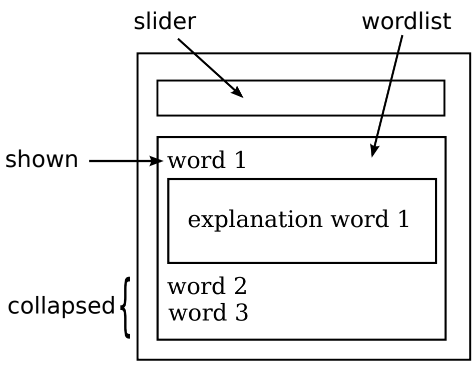


Figure 4.1: Layout of mostRelevant.html

Now a JavaScript file `mostRelevant.js` is created in the directory `js`. It contains the actual program to be run, when the HTML file is loaded. Also it offers functions for communication with the web services and for adding elements to the page.

The first function implemented is `handleRequest`, responsible for parsing the request header[21]. It checks if a filename is requested and extracts its name.

The function `showRelevantWords`, taking the parameter `filename`, sets up a new MooTools accordion object, responsible for managing the word list. This includes methods to add words or their explanations, handling user interaction, showing and hiding explanations and providing a sliding effect when switching the selected word. After creating accordion, the function `showRelevantWords` sends a JSON request to arffCache, asking for the relevant words belonging to the filename. ArffCache responds with corresponding WordNet URIs and the tf-idf ranking. The words are extracted from the URIs, and added to the accordion (see figure 4.2). The tf-idf ranking is used to modify the brightness of the background color: the higher the ranking, the brighter the color. The base background color is a shade of blue.

The accordion object is extended with the functionality to request further explanations for every word, if not yet loaded (compare figure 4.3). These explanations are WordNet glossaries for all meanings of the words. So if a word is clicked, and the explanations were already

[18]`http://www.w3.org/TR/xhtml11/` (March 2009)
[19]`http://mootools.net/docs/Plugins/Slider`
[20]`http://mootools.net/docs/Plugins/Accordion`
[21] e.g. http://.../mostRelevant.html**?filename=163.txt**

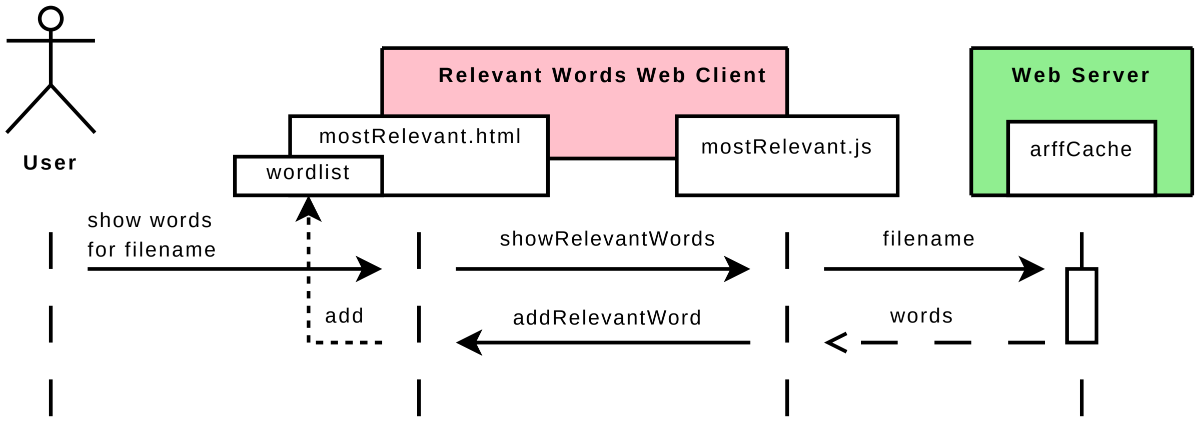


Figure 4.2: Request and show relevant words for a given filename

loaded, this information is displayed. If the box, which should contain the information is not filled with content yet, the function `getGloss` is called.

`GetGloss` sends a SPARQL query via Ajax to the WordNet RDF-Store. The query asks for all wordsenses of the word, and the corresponding glossaries. When the response arrives, new elements are added to the explanation box. These are:

- all wordsenses
- their glossaries
- two links titled hypernyms and hyponyms

The links hypernyms and hyponyms (see figure 4.6) offer further functionality. They get an onclick handler that triggers the function `getRelated`. The WordNet URI of the wordsense and the relation to query for (hypernym or hyponym) are passed as parameters.

`getRelated` queries the RDF store and retrieves a list of hyper-/hyponyms if it exists. They are added as links below the hypernym or hyponym link (compare figure 4.7). Clicking one of them executes the function `addWord`, which adds the word to the top of the accordion. It gets a different background color (orange) so it is easy to determine, that it does not directly belong to the relevant words (as shown in figure 4.8). Being in the accordion, it is handled the same way as the other entries. This means clicking on it also triggers a request and adds the explanation, if the data is not yet present.

If a word from the hyper-/hyponym list is already in the accordion, clicking on it causes highlighting its entry.

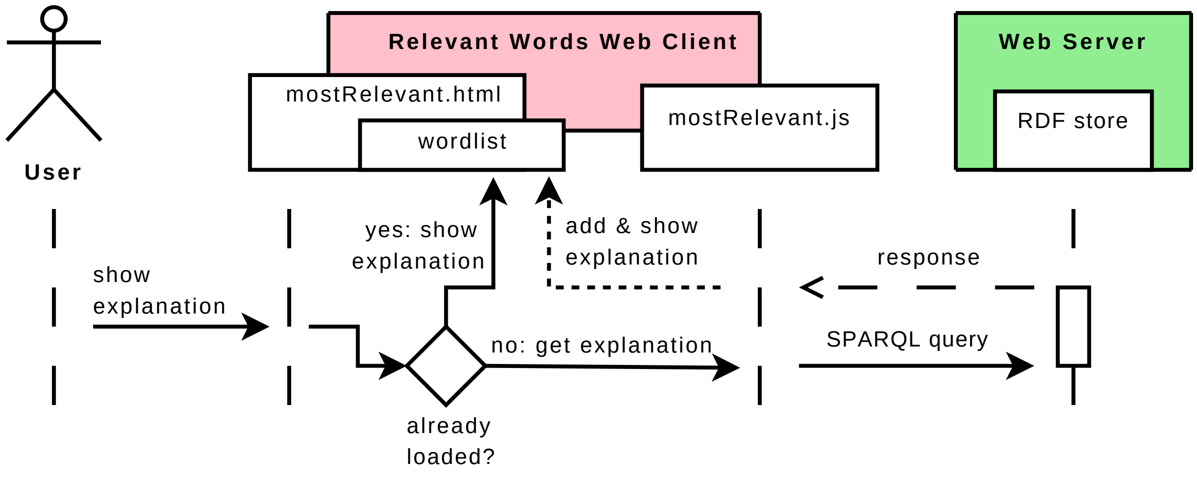


Figure 4.3: Load and/or show word explanation

A MooTools slider is added, that allows choosing a number of relevant words between 1 and 20 to be shown. It just resets the internal variable for `numWords` and then triggers the function `showRelevantWords` again. The accordion is emptied and refilled with `numWord` words.

4.3.3 Implementing research.html

The research web client has many similarities with the relevant words webclient. Its basic structure is contained in the XHTML file `research.html`, the layout in `research.css`, the JavaScript program in `research.js`. The file `research.html` shows two boxes, one for the list of words, one for their explanations (see figure 4.4).

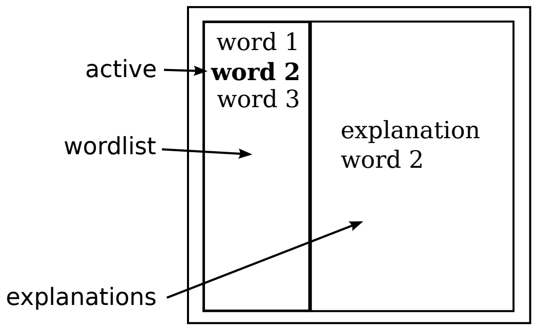


Figure 4.4: Layout of research.html

`reseach.js` also has a function `handleRequest`, which is responsible for checking the request. If the key `show` is set its values are handled. The values may be a list of words, split by the character "|". If a list of words is given, each word is passed to the function `addWord`.

The function `addWord` adds a new element containing the word to the wordlist. Then a SPARQL request to the RDF store is made, querying for the word and its glossaries.

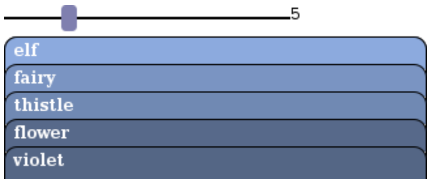


Figure 4.5: Page after loading

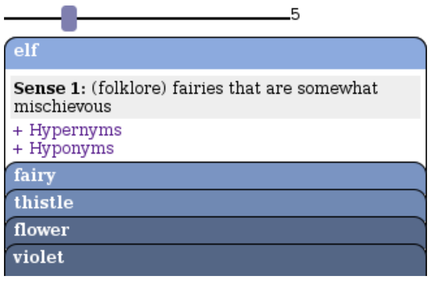


Figure 4.6: *Elf* is clicked

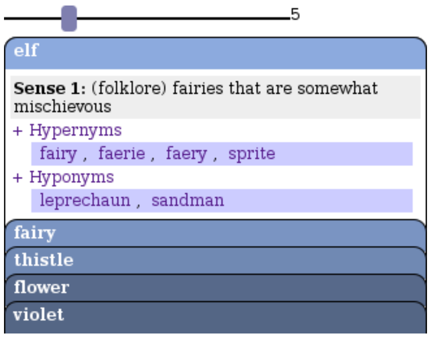

5
elf
Sense 1: (folklore) fairies that are somewhat mischievous
+ Hypernyms
fairy , faerie , faery , sprite
+ Hyponyms
leprechaun , sandman
fairy
thistle
flower
violet

Figure 4.7: *Hypernyms* and *hyponyms* are selected

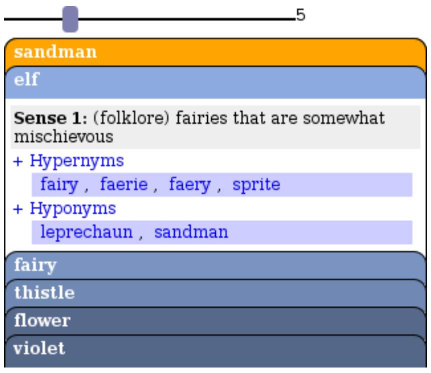

5
sandman
elf
Sense 1: (folklore) fairies that are somewhat mischievous
+ Hypernyms
fairy , faerie , faery , sprite
+ Hyponyms
leprechaun , sandman
fairy
thistle
flower
violet

Figure 4.8: *Sandman* is added

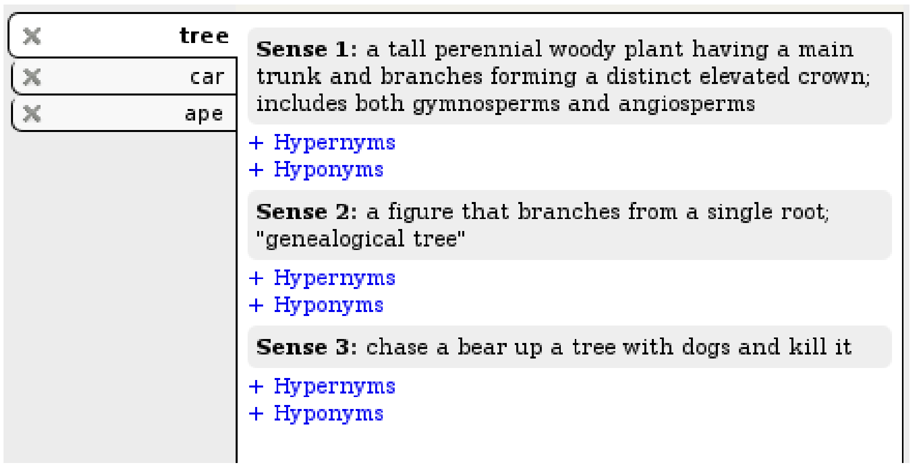
If the entered word is not found in WordNet, the class `notInWordnet` is applied to its wordlist entry. It is shown stroked as defined in `mostRelevant.css`. A corresponding entry is added to the explanations with the text "not in wordnet".

If the entered word is found, the glossaries for its meanings are added to the explanations, each with two links below labeled "hypernyms" and "hyponyms". These are connected with an onclick event, that triggers the function `getRelated` with either the hypernym or hyponym. This function works exactly the same way as the one already described in section 4.3.2. So clicking on a link leads to a comma separated list of words below the link. Clicking on a word from that list triggers the function `addWord`.

Only one explanation is shown at the same time. It is recognizable in the wordlist which one is visible (compare with figure 4.9). The other explanations are hidden. Clicking another word removes the class `active` from the former active while setting the class of the clicked word's element to active. Further the former active words explanation is hidden while the new active explanation is set to be visible. This gives the user the feeling of switching tabs. This impression is supported by the CSS definition of the active class. This defines a white background and lets the element pad over the border of the explanation box (as visible in 4.9). This clarifies that active word and explanation belong together. Further the word contained in the active element is set to bold. Inactive elements are shown with a gray background and normal font weight.

Every "tab" has an icon x (img/x.gif) on the left. Clicking it triggers the function `removeWord`. It removes a given word and its explanation from the page.

Example:
`http://textgrid-ws2.gwdg.de/wordnetWebclients/research.html?show=tree|car|ape`



tree
car
ape

Sense 1: a tall perennial woody plant having a main trunk and branches forming a distinct elevated crown; includes both gymnosperms and angiosperms

+ Hypernyms
+ Hyponyms

Sense 2: a figure that branches from a single root; "genealogical tree"

+ Hypernyms
+ Hyponyms

Sense 3: chase a bear up a tree with dogs and kill it

+ Hypernyms
+ Hyponyms

Figure 4.9: research.html shows the words tree, car and ape

Every change of the wordlist is reflected in the statusbar by the function `words2Statusbar`. It is called every time a word is added or removed from the list. It sets the statusbar to show the entry `wnwords:` and a list of all words, separated by the character "|"[22]. Statusbar messages set by JavaScript are not shown anymore by modern browsers (security issues), but this functionality is needed for the TextGrid workbench integration of `research.html`, described in section 4.5.1.

4.4 Implementing the WordNet Plug-in Structure

As the TextGrid workbench is Eclipse-based, integration of new features is done with Eclipse plug-ins. Writing of Eclipse plug-ins can be done with the version "Eclipse for RCP/Plug-in Developers"[23].

[22]example: `wnwords:car|tree|ape`
[23]`http://www.eclipse.org/downloads/packages/eclipse-rcpplug-developers/ganymedesr2` (March 2009)

The browser

The web clients are shown in the web browser widget of the SWT library. It is able to display HTML and execute JavaScript. Therefore it utilizes an underlaying browser engine. The TextGrid workbench comes already bundled with *XULRunner*[24]. The browser widget `org.eclipse.swt.browser.Browser` has the methods `setUrl()` and `execute()` which are further used.

`setUrl()` loads the webpage given as argument.
`execute()` executes JavaScript given as argument.

Communication between browser widget and Eclipse is very limited, but there are ways to work around this. To get information from JavaScript executed in the browser widget the statusbar can be used, with a listener on `statusbarchanged()`[25].

Creating a plug-in

A new plug-in project can be created by selecting *Plug-in Project* in the *New* dialog of Eclipse. In the wizard a project name "info.textgrid.lab.wordnet" and other parameters can be set. As a basic template *Plug-in with a view* is chosen, which sets up a sample view that can be edited later. Clicking *Finish* creates the new plug-in project, including needed files and directories.

```
info.textgrid.lab.wordnet
  |-- META-INF
  |   `-- MANIFEST.MF
  |-- build.properties
  |-- icons
  |   `-- sample.gif
  |-- plugin.xml
  `-- src
      `-- info
          `-- textgrid
              `-- lab
                  `-- wordnet
                      |-- Activator.java
                      `-- views
                          `-- SampleView.java
```

[24]a Mozilla product, which has the same rendering engine as firefox `https://developer.mozilla.org/en/XULRunner`

[25]As the TextGrid workbench is bundled with XULRunner now, it would also be possible to use the XPCOM interface `http://www.eclipse.org/swt/faq.php#howusejavaxpcom` (March 2009)

4.4.1 The Relevant Words Perspective

The Relevant Words View

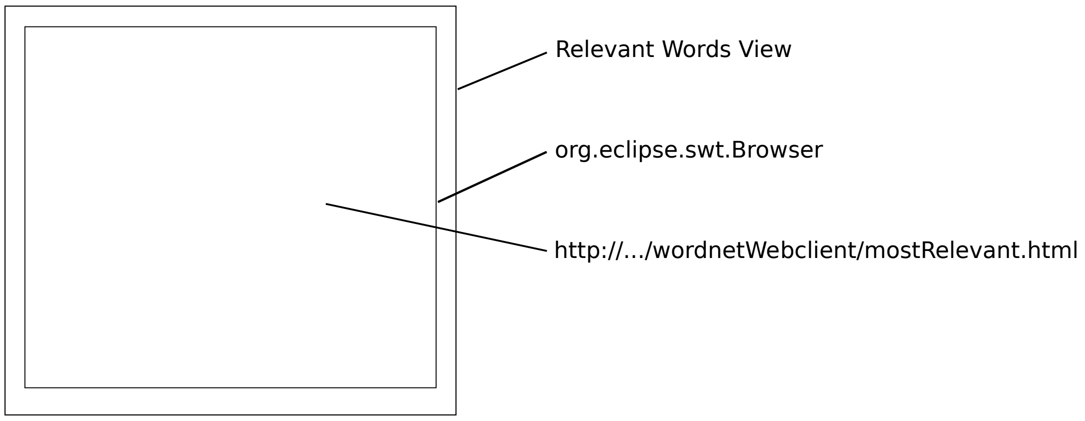


Figure 4.10: Schema: Relevant Words View

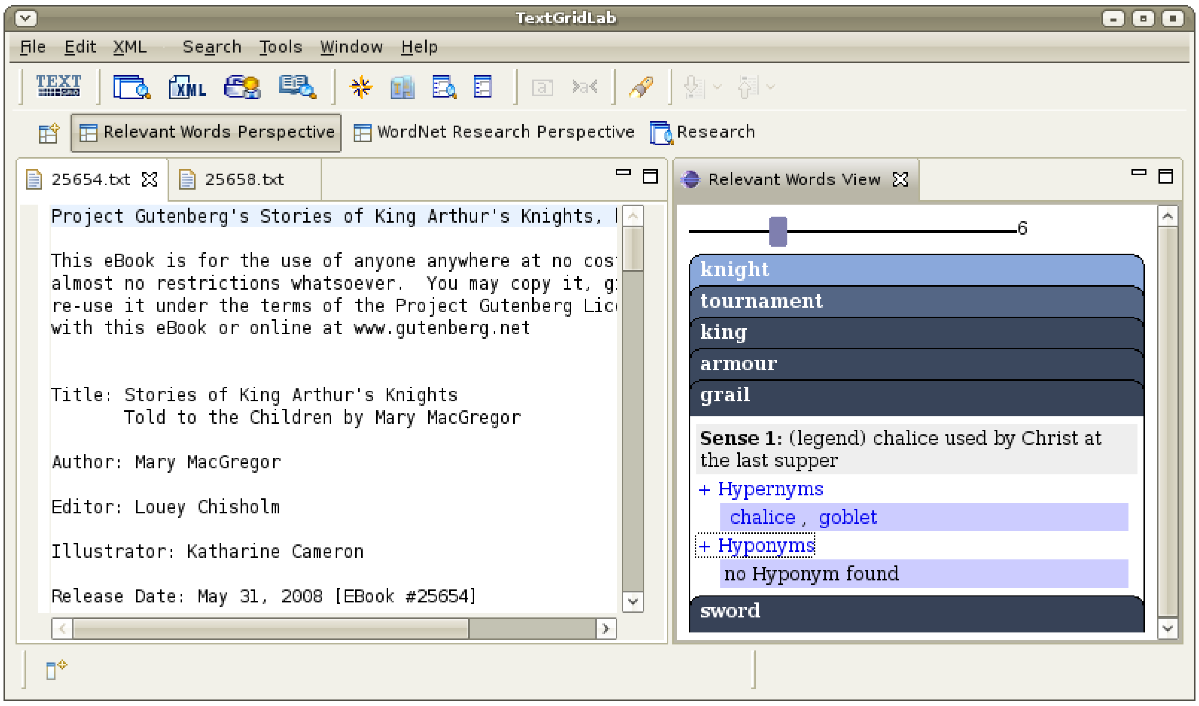
The class `SampleView` is replaced by `MostRelevantView`, which extends `ViewPart` and therefore has the mandatory functions `createPartControl` and `setFocus`. `createPartControl` is important, as it defines the view layout. A private member `wnBrowser` of the type `org.eclipse.swt.browser.Browser` is added to the class. Initializing `wnBrowser` is done in `createPartControl`. The url is set to `mostRelevant.html`[26].

The view needs a way to find out if an editor is shown in the workbench and which textfile is opened in the editor. This is done with the help of an `IPartListener2` which gets notified every time the user switches a view in the workbench. If the now active part of the workbench is of the type `org.eclipse.ui.DefaultTextEditor`, the `execute` method of the `wnBrowser` object is called. It executes methods that are embedded in the JavaScript of the website shown in the browser. The name of the opened file is passed to the method `showRelevantWords`. This triggers functionality of `mostRelevant.js`, already described in section 4.3.2, resulting in the possibility of seeing the relevant words for the opened text, if its name is found within arffCache.

Adding the Perspective

As the view is available now, a perspective is created combining editor and *Relevant Words View* in one layout for better usability. This is done by adding a new package `info.textgrid.lab.wordnet.perspectives` and a class `WordnetEditorPerspective` to the package. `WordNetEditorPerspective`

[26]`http://textgrid-ws2.gwdg.de/wordnetWebclients/mostRelevant.html`


TextGridLab
File Edit XML Search Tools Window Help
Relevant Words Perspective
WordNet Research Perspective
Research
25654.txt
25658.txt
Relevant Words View
6

```
Project Gutenberg's Stories of King Arthur's Knights, I

This eBook is for the use of anyone anywhere at no cos
almost no restrictions whatsoever.  You may copy it, g
re-use it under the terms of the Project Gutenberg Lic
with this eBook or online at www.gutenberg.net


Title: Stories of King Arthur's Knights
       Told to the Children by Mary MacGregor

Author: Mary MacGregor

Editor: Louey Chisholm

Illustrator: Katharine Cameron

Release Date: May 31, 2008 [EBook #25654]
```

knight
tournament
king
armour
grail

Sense 1: (legend) chalice used by Christ at the last supper

+ Hypernyms
chalice , goblet
+ Hyponyms
no Hyponym found

sword

Figure 4.11: Relevant Words Perspective

implements `IPerspectiveFactory`. The method `createInitialLayout` is responsible for arranging the views. Here it is stated, that the `MostRelevantView` should be at the top right of the editor area.

Now `plugin.xml` is edited to tell Eclipse that a new perspective exists. Therefore a new extension with the point `org.eclipse.ui.perspectives` is created. A perspective with the name *Relevant Words Perspective*, the class and an id is defined. The result can be seen in figure 4.11.

4.5 The WordNet Research Perspective

The *WordNet Research View* is added to the *Research Perspective* of the TextGrid workbench. It is shown on the right side of the *Search View* and reacts to user input. Entered words are displayed in research.html. For this functionality *WordNet Research View* needs to be notified of text changes in *Search View* and get hold of its content. The entered text is analyzed to display new entered words.

Modification of SearchView

Search View needs to be able to notify other plug-ins when the user enters text in the free text search field. This is typically done using listeners[27]. *Search View* obtains the possibility to attach listeners to it. This is done by adding the publicly accessible methods `addSearchTextChangedListener` and `removeSearchTextChangedListener` to the class `SearchView`. Other plug-ins can register own listeners with these functions and implement a functionality to be executed in case the listener is called.

SearchView has a private member, that holds listeners added to the view [28]. `searchText` has a `ModifyListener` attached, which calls the method `notifySearchTextChangedListeners`. This method is implemented to send an event to all listeners from the `ListenerList`. This means all listeners are notified when `searchText` changes.

If another plug-in receives this event, it may want to know the exact text entered. Also it may want to change the entered text. To make this possible the methods `setSearchText` and `getSearchText` are added to the view, which are simple get/set methods for the private class member `searchText`.

4.5.1 WordNet Research View

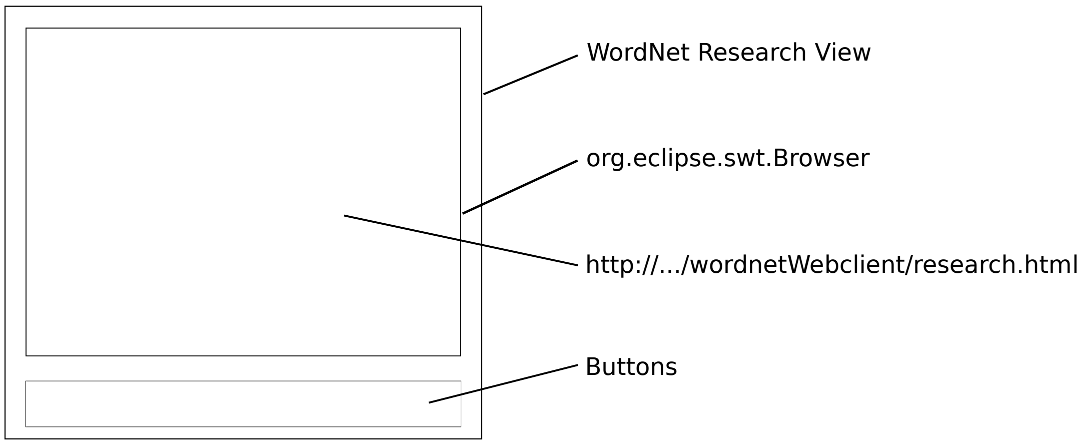


Figure 4.12: Schema: WordNet Research View

`WordnetSearchView` consists of a browser in which *research.html* is displayed. Also shown are some buttons which offer further functionality (see figure 4.12). The class `WordnetSearchView` is

[27]Listeners in eclipse implement the interface IListener, Eclipse takes care of the handling. The interaction is following the *observer pattern*.

[28]as in http://blog.eclipse-tips.com/2008/12/listenerlist-better-way-to-manage-your.html

created in the package `info.textgrid.lab.wordnet.views`. It extends `ViewPart`, and has a private member `browser`. In the method `createPartControl` the browser is set up with the url set to *research.html*.

A listener is registered at *Search View* it to be notified if the event `SearchTextChanged` occurs. In this case the search text entry is read with the `getSearchText` method of *Search View*. If the text entered ends with a space, the new word is passed to the `addWord` function of `research.html` in the browser. Now the functionality is in place to show every word entered (and finished with a space after it) in `research.html`. This process is also illustrated in figure 4.13.

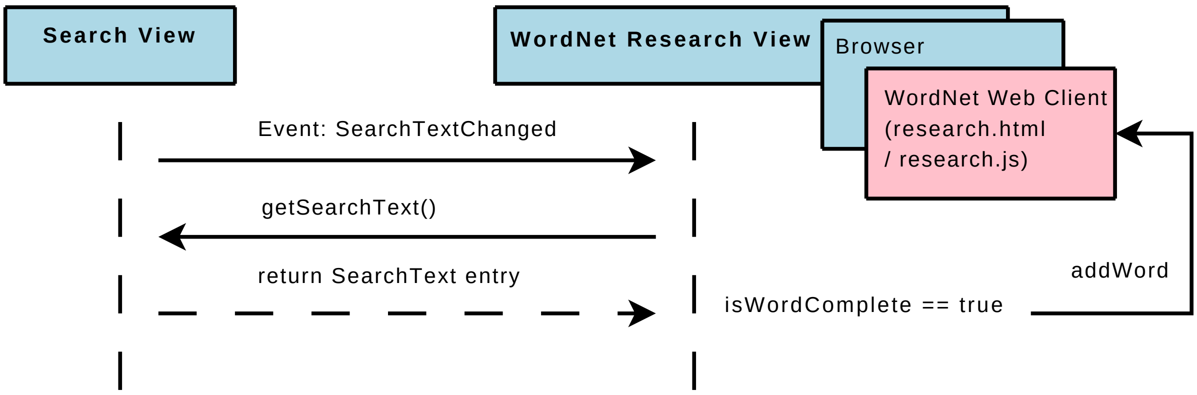


Figure 4.13: Interaction between *Search View* and *WordNet Research View*

Setting words in Search View

Another desirable feature is the possibility to replace the search text entry from *Search View* with the words shown in *research.html*. This includes the need for finding out which words are actually shown in *research.html*. As already mentioned in section 4.3.3, *research.html* appends the list of words shown to the statusbar. So a `StatusTextListener` is added to the browser, which is called every time the statusbar changes. In this case the statusbar string is read. If it starts with `wnwords:` the words listed are used to replace the array `wnWords` of the WordnetSearchView-Object. Thus it is ensured that the same words shown in `research.html` are always also listed in the array `wnWords` (compare with figure 4.14).

Two buttons are added to *Wordnet ResearchView*. One push button labeled "replace search with words" and a radio buttons labeled "live replace".

The "replace search with words" button triggers the method `words2SearchEntry` if clicked. This method replaces the search text entry from *Search View* with the words listed in `wnWords` (as shown in figure 4.14).

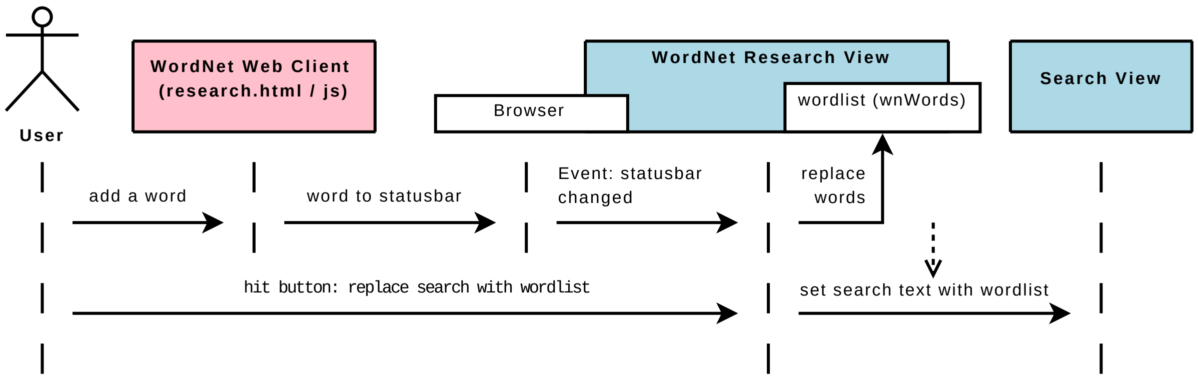


Figure 4.14: Achieving consistency of words in *research.html* and *Wordnet Research View*. Setting words in *Search View*

The check button "live replace" sets the class-variable `liveReplace` to true, and disables the other button. If `liveReplace` is true, the `statusbarChangedListener` calls `words2SearchEntry` on every statusbar change. This results in the effect that with every change of the wordlist in *research.html*, the search term of `SearchView` is also changed. Thus, they are both always in the same state.

4.6 Integration into TextGrid-Workbench

The `plugin.xml`, delivered with every Eclipse plug-in defines the additions the plug-in contributes to an Eclipse product. This is done via so called "extension points", containing the information which parts of the workbench should be extended, and how new functionality should integrate.

The WordNet plug-in extends the Workbench with the contribution of

- two views, *WordNet Research View* and *Relevant Words View*
- two perspectives , *WordNet Research Perspective* and *Relevant Words Perspective*

These extensions are defined in `plugin.xml` by pointing to the Java classes which implement them.

5 Evaluation and Conclusion

5.1 Usage

To use the finished plug-in an installation of the TextGridLab beta is needed. This can be downloaded from `http://www.textgrid.de/beta.html` . The file `info.textgrid.lab.wordnet_1.0.0.jar` can be retrieved from `http://lubl.de/tglab/`. It needs to be placed in the folder named `plugins` from the unzipped TextGridLab. After starting textgridlab executable the welcome screen can be closed. Login is not required for using the WordNet plug-in. Now a click on the *open perspective* button (see figure 5.1) offers two new perspectives, called *Wordnet Research Perspective* and *Relevant Words Perspective*.

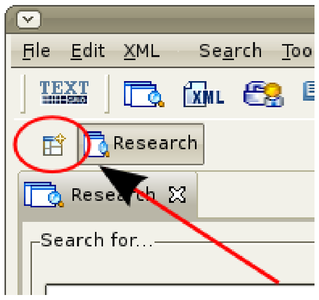


Figure 5.1: *Open Perspective* button

1. Relevant Words Perspective

For testing this perspective the corpus.zip file from `http://lubl.de/tglab` needs to be downloaded and unzipped. Now the text files contained can be opened with the menu entry "File->Open File" in the *Relevant Words Perspective*. The most relevant words are displayed in the view on the right. They are clickable for further research. The slider at the top allows adjusting the number of words shown in a range of one to twenty.

2. Wordnet Research Perspective

Entering words in the field "Search for..." triggers displaying the word in *WordNet Research View*, after the space key is hit finishing the word. Words are displayed in tabs and can be looked up by clicking them. The search text entry can be replaced either by clicking "replace search with words" or by selecting "live replace".

5.2 Evaluation

User Interface

The user interface is intuitive. Only a few GUI elements (like tabs, accordion) are shown. These are introduced in a reasonable way. The applied effects emphasize the sense of the application without distraction. The choice for a HTML/CSS/JavaScript based interface offered design flexibility. The performance of the application is good. The asynchronous loading gives the impression of instant data availability. It is well integrated in the TextGrid workbench.

Combining determination of relevant words with WordNet

The usage of WordNet in the process of corpus indexing changes the output. For example in the book “Jungle Tales of Tarzan”[1] the word list changes as shown in the following tabular.

	WordNet	**w/o WordNet**
1	ape	tarzan
2	jungle	ape
3	hyena	teeka
4	fang	taug
5	tribe	numa
6	bull	bukawai
7	warrior	balu
8	witch	tibo
9	panther	mbonga
10	lion	jungle
11	spoor	momaya
12	spear	sheeta

Noticeable is that with the usage of WordNet (a thesaurus) names lose relevance. This may be a wanted or unwanted effect. For the use case developed in this thesis it is a wanted effect. The pre-condition is that the text is unknown to the user, so names would not help to get a quick overview.

[1] 106.txt in corpus zip, or `http://www.gutenberg.org/etext/106` (March 2009)

5.3 Possible Further Work

Combining RESTful web services and interlinking data

Especially the Ajax based web client architecture may help to use different data sources or (RESTful) services in a so called "Web 2.0 mashup" style[2]. SPARQL and RDF play a special role here, as they are already standardized for querying data across different domains.

As WordNet URIs are also used in other contexts, a further possibility is interconnecting different databases to further enrich the data shown. The *DBpedia* [3] database for example can already be queried with SPARQL for WordNet URIs and is also interlinked with further data sets[4]. This data could also be shown on demand.[5]

Expanding WordNet usage

WordNet offers more relations between words than just hypernyms and hyponyms. Relations like meronyms or antonyms could also be made browsable by the user. The associated data sets would need to be imported in the RDF store and required SPARQL queries added to the WordNet web clients.

SKOS

An interesting option would be the integration of SKOS based thesauri. This could enable easier extension to other languages and domains, as it is likely that with the finalisation of the SKOS recommendation more SKOS based thesauri will show up, which could easily be imported.

Extending Relevant Words View

To really use the *Relevant Words View* the user should be able to index an own corpus. For example by clicking on a button "index corpus" in the navigator of the TextGrid workbench. Further arffCache should handle more than just one corpus index. These should not be kept in memory any more, but could go into a relational database.

[2]an introduction to mashups: `http://www.soamag.com/I18/0508-1.asp` (April 2009)
[3]`http://dbpedia.org/` (April 2009)
[4]`http://wiki.dbpedia.org/Interlinking` (April 2009)
[5]more about interlinking data at `http://esw.w3.org/topic/SweoIG/TaskForces/CommunityProjects/LinkingOpenData` (April 2009)

5.4 Conclusion

This thesis shows how TextGrid could adopt different techniques for further enhancements. One has to differentiate between the technical and theoretical part of this work. Theoritically shown are ideas for integrating methods from information retrieval and a way to integrate an RDF based semantic network in the workbench. Technically shown is an architecture which combines RESTful web services within an Ajax interface, making use of unified identifiers for resources, combining so called "Web 2.0" with semantic web technologies. The modularity of the taken approach allows extension of the components and further recombination in new services.

A Abbreviations

API Application Programming Interface

ARFF Attribute-relation file format

CSS Cascading Style Sheet

DOM Document Object Model

GUI Graphical User Interface

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

OWL Web Ontology Language

RAM Random Acess Memory

REST Representional State Transfer

RDF Resource Description Framework

SOAP Simple Object Protocol

SPARQL SPARQL Protocol and RDF Query Language

XML Extensible Markup Language

WSDL Web Service Description Language

B List of Figures

C Bibliography

[1] JSON homepage. http://www.json.org (March 2009).

[2] TextGrid Homepage. http://www.textgrid.de/ (March 2009).

[3] Attribute-Relation File Format (ARFF). http://www.cs.waikato.ac.nz/ ml/weka/arff.html (March 2009).

[4] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, 1999.

[5] Jeremy J. Carroll and Graham Klyne. Resource description framework (RDF): Concepts and abstract syntax. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/ (March 2009).

[6] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (WSDL) 1.1. W3C note, W3C, March 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315 (March 2009).

[7] Christiane Fellbaum, editor. *WordNet. An Electronic Lexical Database*. The MIT Press, 1998.

[8] Aldo Gangemi, Guus Schreiber, and Mark van Assem. RDF/OWL representation of WordNet. W3C working draft, W3C, June 2006. http://www.w3.org/TR/2006/WD-wordnet-rdf-20060619/ (March 2009).

[9] Jesse James Garrett. Ajax: A new approach to web applications, February 2005. http://www.adaptivepath.com/ideas/essays/archives/000385.php (March 2009).

[10] Marc Hadley and Paul Sandoz. JAX-RS: The Java API for RESTful Web Services. Java Specification Request (JSR) 311, October 2008. http://jcp.org/en/jsr/detail?id=311 (March 2009).

[11] Karin Haenelt. Information retrieval modelle. vektormodell. kursfolien. October 2006. http://kontext.fraunhofer.de/haenelt/kurs/folien/Haenelt_IR_Modelle_Vektor.pdf (March 2009).

[12] Roman Hausner. Semantic Text Mining - linguistische Tools im Preprocessing von Text Mining Methoden. April 2009.

[13] Yves Lafon and Nilo Mitra. SOAP version 1.2 part 0: Primer (second edition). W3C recommendation, W3C, April 2007. http://www.w3.org/TR/2007/REC-soap12-part0-20070427/ (March 2009).

[14] Alistair Miles and Sean Bechhofer. SKOS simple knowledge organization system reference. W3C candidate recommendation, W3C, March 2009. http://www.w3.org/TR/2009/CR-skos-reference-20090317/ (March 2009).

[15] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. RESTful Web Services vs. Big Web Services: Making the Right Architectural Decision. In *17th International World Wide Web Conference (WWW2008)*, Beijing, China, April 2008. http://www.jopera.org/docs/publications/2008/restws (March 2009).

[16] Stefan Büdenbender Fotis Jannidis Marc W. Küster Christoph Ludwig Wolfgang Pempe Thorsten Vitt Werner Wegstein Andrea Zielinski Peter Gietz, Andreas Aschenbrenner. Textgrid and ehumanities. *Science*, 2005. http://www.textgrid.de/fileadmin/TextGrid/veroeffentlichungen/TextGrid-Amsterdam-2006-final.pdf (March 2009).

[17] Andy Seaborne and Eric Prud'hommeaux. SPARQL query language for RDF. W3C recommendation, W3C, January 2008. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/ (March 2009).

[18] Wordnet - a lexical database for the english language. http://wordnet.princeton.edu/ (March 2009).