Semantic Assistants

Guide for Users and Developers

René Witte
Thomas Gitzinger
Nikolaos Papadakis
Bahar Sateli

Development Release
November 2, 2010

Semantic Software Lab
Concordia University
Montréal, Canada
http://www.semanticsoftware.info

Contents

About this document

This document contains documentation for the *Semantic Assistants* project. You can obtain the latest version from http://www.semanticsoftware.info/semantic-assistants.

Acknowledgments

The following developers contributed to the design and implementation of the Semantic Assistants: René Witte, Nikolaos Papadakis, Tom Gitzinger.

License

[1] AGPL3, http://www.fsf.org/licensing/licenses/agpl-3.0.html

Chapter 1

Introduction to Semantic Assistants

1.1 Overview

The Semantic Assistants project aims to bring natural language processing (NLP) techniques directly to end users by integrating them with common desktop applications, such as word processors, email clients, or Web browsers. To facilitate this integration, a service-oriented architecture (Figure 1.1) has been developed that allows to integrate (desktop) clients with NLP services implemented in the GATE framework.[1]

For the general motivation, design, and background on the Semantic Assistants project, please read the information on the Semantic Assistants Web site[2] and the contained references first (Gitzinger and Witte, 2008; Witte and Gitzinger, 2009). This document only describes the installation and use of the developed system.

1.2 How to read this documentation

To deploy Semantic Assistants, we recommend you first read this overview chapter. Then, consult the following parts of this documentation:

End Users: Please refer to the server installation guide (Chapter 2) and the client guides in Chapter 4.

Language Engineers: If you want find out how to integrate a new NLP service, please refer to Section 3.3 and Section 5.3 for documentation on the OWL service descriptions.

Plug-in Developers: Please first read the background material (Web site, papers). Then refer to the developers' notes in Chapter 5.

1.3 Architectural Overview

This section gives an overview over the current version of the Semantic Assistants architecture, as shown in Figure 1.1:

Tier 1 of the architecture consists of client applications and a *Client-Side Abstraction Layer (CSAL)*. Currently, there are two example clients distributed with the system, a simple command-line client for testing purposes and a plug-in for the OpenOffice.org *Writer* word processor. The client-side abstraction layer consists partly of hand-written Java classes that provide common client-side functionality, partly of automatically generated Java classes.

[1] GATE, http://gate.ac.uk
[2] Semantic Assistants, http://www.semanticsoftware.info/semantic-assistants

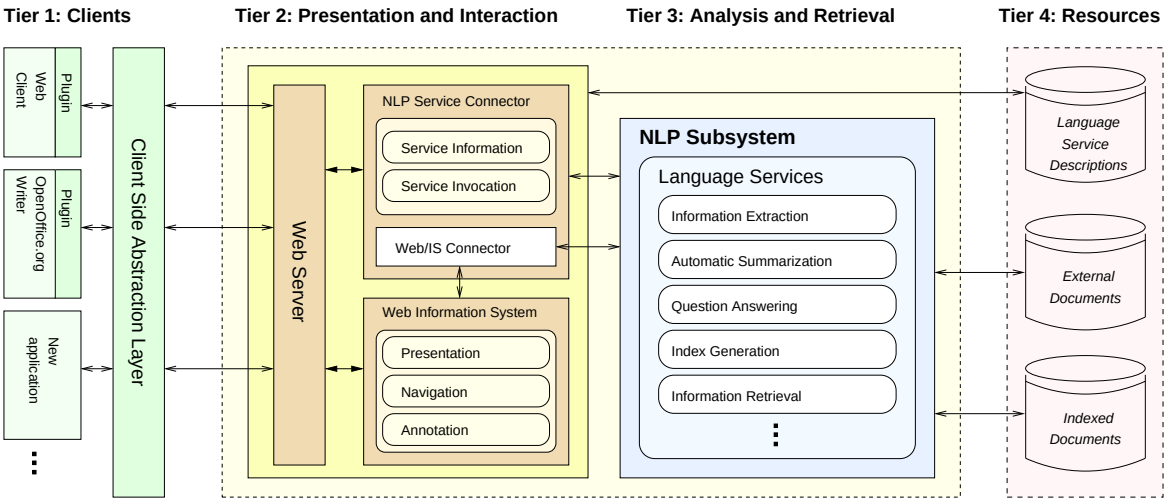


Figure 1.1: The Semantic Assistants architecture

The communication between client and server is implemented by means of W3C Web services.[3]

Tier 2 of the architecture consists of a *Web server* and the *NLP Service Connector*. The Web server used by default in the architecture is the Java 6 embedded Web server. The NLP Service Connector currently integrates the GATE framework for NLP. It is responsible for a number of tasks, including communication with the client, reading and querying the language service descriptions, running requested language services, and generating response messages.

Tier 3 is the NLP subsystem. At present, only the GATE framework is supported (future work might integrate additional frameworks, such as UIMA). It makes use of the GATE API in order to assemble language services, store them in a permanent way, and invoke them when they are requested by a client.

Tier 4 is the resource tier. Here we have the language service descriptions, which are authored in the Web Ontology Language (OWL). Tier 4 further contains external documents, which the NLP subsystem must be able to access. Finally, we possibly have pre-indexed documents as part of the resources tier. For indexing, GATE uses Apache's Lucene indexer[4] as a subsystem, and allows us, through its API, to create and access indices.

1.4 System Components

The implementation of the Semantic Assistants architecture currently comes with the following components:

Server: The server is the core of the architecture. It communicates with the clients through the CSAL on one hand and the NLP framework through the NLP Service Connector on the other. At present, the architecture only contains a connector for GATE. However, it

[3]Web Services Architecture, see http://www.w3.org/TR/ws-arch/
[4]Apache Lucene, see http://lucene.apache.org/

was explicitly designed to allow an easy integration of other frameworks (for example, UIMA). For describing available services, we use ontology-based (OWL) service descriptions. As a service-oriented architecture (SOA), every service is automatically available to all clients connected to the architecture, using standard WSDL[5]) interface descriptions.

Client-Side Abstraction Layer (CSAL): Our top goal was to make it as easy as possible for client (plug-in) developers to integrate NLP functionality. As clients should be able to connect to the architecture entirely by "local" means, we provide an *abstraction layer*, named CSAL, which is located on the client side and performs the actual communication with the server.

Apart from the communication functionality, the CSAL also provides common client-side functionality, i.e., useful data types and methods that are frequently required when integrating NLP into desktop clients.

Clients: Two example clients come with the architecture: a command-line client and a plug-in for the OpenOffice.org Writer word processor.

Example Resources: NLP functionality is provided to clients through Web services. To match clients with suitable services (depending on language, formats, etc.), each NLP service comes with a semantic service description in OWL format. Three example service descriptions are included in the current distribution: an information extraction (IE) service that detects persons and locations (using GATE's ANNIE pipeline), an IR service (using the Yahoo PR) and a compound service, which combines the IR and the IE service. These should help you in defining your own NLP services that you deliver to your end users (e.g., summarization, question-answering, domain-specific NLP services).

Documentation and Online Resources: Apart from this guide, a number of publications on the Semantic Assistants are available online,[6] as well as a discussion forum for support.

[5]Web Services Description Language (WSDL), see http://www.w3.org/TR/wsdl
[6]Semantic Assistants, http://www.semanticsoftware.info/semantic-assistants

Chapter 2

Installation

Note: at present, the installation has been tested under Linux and Mac OS X

2.1 Prerequisites

To deploy the Semantic Assistants architecture, a number of other (open source) components are needed. Note that the current distribution comes with pre-compiled libraries for the Semantic Assistants components, but to run them you will still need to install the common and server-specific prerequisites:

Common throughout the Project:

1. Apache ANT, http://ant.apache.org/
2. Sun JDK 6, http://java.sun.com/javase/downloads/index.jsp
3. GATE v5.0 or newer, http://gate.ac.uk/

For the Semantic Assistants Server:

1. Java API for XML Web Services (JAX-WS) v2.1.x, https://jax-ws.dev.java.net/
2. Protégé-OWL API v3.4, http://protege.stanford.edu/plugins/owl/api/

For the CSAL:

1. Java API for XML Web Services (JAX-WS) v2.1.x, https://jax-ws.dev.java.net/

For the OpenOffice.org Writer Plug-in:

1. OpenOffice 3.2, http://download.openoffice.org/
2. The OpenOffice.org 3.2 SDK, http://download.openoffice.org/sdk/index.html
3. Apache log4j v1.2, http://logging.apache.org/log4j/1.2/index.html
4. Apache Commons Lang v2.4, http://commons.apache.org/lang/

2.2 Download

To download the latest version of the Semantic Assistants and this documentation please refer to http://www.semanticsoftware.info/semantic-assistants-architecture.

2.3 Project Desciption

This section gives an overview of the directory structure and explains the modifications required for every user to be able to proceed with the compilation.

Directory Structure : The implementation of the architecture is located in `SemanticAssistants/`. There are four directories, *Server*, *CSAL*, *Clients* and *Resources* and one file *SemassistProperties.xml*

2.3.1 Server Directory Structure

The following directories and files are found under the *Server* directory.

1. `src`: Contains the java source code.
2. `gate-home`: Contains two gate user configuration files *gate.xml* and `user-gate.xml`.
3. `logs`: It is used for server log files.
4. `nbproject`: Contains configuration and properties files used by the NeatBeans IDE if the *Server* is loaded through Netbeans.
5. `.classpath`: ClassPath file for NetBeans.
6. `.project`: Project Description file for NetBeans.
7. `Makefile`: Makefile that can be used to invoke Ant targets.
8. `build.xml`: Ant build file use to compile the project. Contains dependency information.
9. `nb-build.xml`: Build file used by Netbeans.
10. `protege.properties`:Generated properties file, used by Protege.
11. `build`: Contains the binary java .class files.
12. `dist` Contains a .jar file of the compiled server side.
13. `nbdist` Contains a .jar file of the compiled server. Generated when the server is compiled through Netbeans.

2.3.2 CSAL Directory Structure

The following directories and files are found under the *CSAL* directory.

1. `src`: Contains the java source code of the client side absraction layer.
2. `dist`: Contains the output .jar file result of compilation of the CSAL.
3. `nbproject`: Contains configuration and properties files used by the NeatBeans IDE if the *CSAL* is loaded through Netbeans.
4. `.classpath`: ClassPath file for NetBeans.
5. `.project`: Project Description file for NetBeans.
6. `Makefile`: Makefile that can be used to invoke ant targets.
7. `build.xml`: Ant build file use to compile the project. Contains dependency information.
8. `bin`: Contains the binary java .class files.
9. `nb-build.xml`: Buildfile used by Netbeans.

2.3.3 Clients Directory Structure

The two sample clients we provide, *CommandLine* and *OpenOffice*, have the following directory structure.

Command Line Client Directory Structure

1. `src`: Contains the java source code of the Command Line Client.
2. `nbproject`: Contains configuration and properties files used by the NeatBeans IDE if the *Command Line Client* is loaded through Netbeans.
3. `.classpath`: ClassPath file for NetBeans.
4. `.project`: Project Description file for NetBeans.
5. `build.xml`: Ant build file use to compile the project.Contains dependency information.
6. `bin`: Contains the binary java .class files.
7. `runclient.sh`: Script helps with the class path setting and running the client.
8. `usableCommands`: Example of commands that can be invoked when running the runclient.sh script.

Open Office Writter Client Directory Structure

1. `src`: Contains the java source code.
2. `dist`: Contains the *Addons.xsu*, *Protocolhandler.xsu* and *ProtocolHandlerAddon_java.uno.jar* files.
3. `build.xml`: Ant build file use to complile the project.Contains dependency information.
4. `bin`: Contains the binary java .class files.
5. `Semassist.uno.zip`: Contains the output file result of complilation and packaged in uno.zip plug-in format used by OpenOffice.
6. `nbproject`: Contains configuration and properties files used by the NeatBeans IDE if the *Command Line Client* is loaded through Netbeans.
7. `nb-build.xml`: Buildfile used by Netbeans.

2.4 Path Configuration

The *SemassistProperties.xml* file serves for two purposes. It's included by the ANT *build.xml* files of all projects and contains common directory paths and used to compile the various Semantic Assistants components. Secondly, it used by the server at run-time as a property file.

Users needs to modify the values of the properties in order to correspond at the proper paths of their machine. The options to be adapted, include the path to the service description directory, GATE home, GATE plug-ins, etc. Descriptions of these properties are as the followings:

The first part is an import statement, where it includes the `LocalProperties.xml` file. This file the is the place to store your local paths and customizations e.g. you can add additional paths for your local pipelines in this file.

```
<!-- Importing LocalProperties.xml file for local paths and customizations -->
<import file ="LocalProperties.xml"/>
```

The second part of the `SemassistProperties.xml` is where the machine-specific variables are modified to point to proper paths of the users machine. The variables are as the followings:

```
<!-- Machine dependent properties.Need to be modified -->
<property name="durmtools" value="/usr/local/durmtools" />
<property name="semassist.home" value="${user.home}/Repository/durm/Projects/SemanticAssistants" />
<property name="csal.home" value="${semassist.home}/CSAL" />

<property name="jaxws.home"     value="${durmtools}/jaxws-ri" />
<property name="protege-home"   value="${durmtools}/Protege" />
<property name="gate-home"      value="${durmtools}/GATE/gate" />
<property name="jdk.home"       value="${durmtools}/jdk" />
<property name="log4j.home"     value="${gate-home}/lib/"/>
```

1. `durmtools`: This is one of the most important variables in this properties file and needs to be defined properly. According to the prerequisites mentioned earlier, there are various applications that need to be installed prior to using the Semantic Assistants. Create a folder called `durmtools` and install all the required applications to this folder, or point this variable to where all your applications are installed e.g. *Applications* on Mac or *Programs* in Windows.

2. `semassist.home`: This variable points to the place on your local machine that the Semantic Assistants package has been downloaded and extracted. Put the full path of the folder in here, exempting the user home directory path for it will be automatically replaced by `${user.home}`.

3. `csal.home`: This variable points to the `CSAL` folder inside the Semantic Assistants package. If the structure is untouched, the path should read `"${semassist.home}/CSAL"`.

4. `jaxws.home`: This variable points to the Java API for XML Web Services (JAX-WS) v2.1.x application installed in the path defined in `${durmtools}` variable.

5. `protege-home`: This variable points to the Protégé-OWL API v3.4 application installed in the path defined in `${durmtools}` variable.

6. `gate-home`: This variable points to the GATE v5.0 or newer application installed in the path defined in `${durmtools}` variable.

7. `jdk.home`: This variable points to the Sun JDK 6 installed on your machine.

8. `log4j.home`: This variable points to the GATE application logging component. It is located in the GATE application `lib` folder.

The third part includes all the properties used by the OpenOffice.org Writer plug-in. Therefore, in order to use this plug-in, all these variables should be defined beforehand.

```
<!-- Used by Open Office Writter Plug-In-->
<property name="office.home.dir" value="${durmtools}/OpenOffice"/>
<property name="uno-copy-dest"  value="${user.home}/Documents/uno-components" />
<property name="office.program.dir" value="${office.home.dir}/program"/>
<property name="ooo-classes-basis" value="${office.home.dir}/basis3.2/program/classes/" />
<property name="ooo-classes-common" value="${office.home.dir}/ure/share/java/" />
```

1. `office.home.dir`: This variable points to the OpenOffice 3.2 application installed in the path defined in `${durmtools}` variable explained earlier.

2. `uno-copy-dest`: This variable identifies where the Semantic Assistants plug-in will be stored when it is successfully built.

3. office.program.dir: This variable points to the program folder inside ${office.home.dir} path which contains the Writer program.
4. ooo-classes-basis: This variable points to the folder that contains the bulk, brand-independent OpenOffice functionalities.
5. ooo-classes-common: This variable points to the folder that containes common Java JAR libraries used by OpenOffice.

The next part includes all the properties used by the Eclipse plug-in. Therefore, in order to use this plug-in, all these variables should be defined beforehand.

```
1   <!-- Used by Eclipse Plug-In-->
2   <property name="eclipse.program.dir" value="${durmtools}/eclipse-4.0"/>
3   <property name="eclipse.plugins" value="${eclipse.program.dir}/plugins"/>
```

1. eclipse.program.dir: This variable points to the Eclipse v3.5 or newer application installed in the path defined in ${durmtools} variable explained earlier.
2. eclipse.plugins: This variable points to the plugins folder located inside the Eclipse application installed on your machine.

The last part includes all the properties used by the Semantic Assistants server at runtime.

```
1   <!-- Used by the server at runtime as properties -->
2   <property name="gate.plugin.dir" value="${gate-home}/plugins/"/>
3   <property name="gate.user.file" value="${semassist.home}/Server/gate-home/user-gate.xml" />
4   <property name="service.repository" value="${semassist.home}/Resources/OwlServiceDescriptions/"/>
5   <property name="ontology.repository" value="${semassist.home}/Resources/ont-repository/" />
6
7   <property name="runtime.maxmem" value="1638m" />
8   <property name="runtime.heap.initial" value="128m" />
9   <property name="runtime.heap.max" value="1638m" />
10
11  <!-- Port for which the Server listens for debugers to be attached -->
12  <property name="server.port.debug" value="8885"/>
13
14  <!-- Port used by the Server to communicate with the clients through wsdl-->
15  <property name="server.port.wsdl" value="8879"/>
```

1. gate.plugin.dir: This variable points to the GATE application plugins folder used for service executions.
2. gate.user.file: This variable points to the GATE application user configuration file.
3. service.repository: This variable points to the OWL service description files. New service description files are added to this folder once they're available and will be later detected by the server.
4. ontology.repository: This variable points to the folder containing the SemanticAssistants OWL files.
5. runtime.maxmem: This variable contains the maximum amount of runtime memory used by JDK. Unless your JDK complains about the value, leave this as it is.
6. runtime.heap.initial: This variable contains the initial amount of heap space used by JDK at runtime.
7. runtime.heap.max: This variable contains the maximum amount of heap space used by JDK at runtime.
8. server.port.debug: This variable contains the port number on users machine on which the server listens for debuggers to be attached.
9. server.port.wsdl: This variable contains the port number on which the server communicates with the clients through WSDL.

2.5 Compilation

To compile and start the server and compile the CSAL, follow these steps:

1. cd to *Server*
2. `ant run`. The server is built and should come up, with some debug output on the console.
3. To test if the Server is operating open your favorite browser and paste `http://<serverhost>:<serverport>/SemAssist?wsdl` (Note the `<server host>` has a default value of the local machine name and the `<server port>` is the value of the property `server.port.wsdl` found in `SemanticAssistants/SemassistProperties.xml`)
4. cd to *CSAL*
5. a) `ant dist`. This builds the client-side abstraction layer (CSAL) that all (Java) clients should use to connect to the architecture (i.e., the server). Note that the server must be running for the client to be built, because the code generation step imports the server's WSDL definitions. In addition the WSDL definitions are cached localy.

   b) `ant dist-offline` This builds the client-side abstraction layer (CSAL) with the cached WSDL definitions during a previous on-line compilation. In this case the server does not have to be running for the CSAL to be built.
6. To stop the server, simply change to its console window and hit `Ctrl-C`

We recommend you test if the server is running correctly by accessing it through the command-line client described below.

2.6 Configuration of the Example Services

Three example pipelines (NLP services) come with the architecture. They are located in the `Resources` directory. There are two parts: the semantic service descriptions in OWL format stored in the `OwlServiceDescriptions` directory and corresponding GATE pipelines (`.gapp` files) implementing these services in the `GatePipelines` directory.

Because `.gapp` files contain hard-coded directory paths, you must generate them on the same system where you later run the server. Below are step-by-step instructions on how to generate and save these pipelines.

In a similar matter the two example `.owl` files contain a hardcoded path which point to the corresponding GATE pipelines (`.gapp` files). Search and replace the following path in the `annie.owl` file: `<sa:appFileNamexml:lang="en">~/Repository/durm/Projects/SemanticAssistants/Resources/GatePipelines/Annie.gapp</sa:appFileName>`. In the same fashion search and replace the path in the `yahoo.owl` file.

Person and Location Extractor. This NLP service runs the ANNIE IE system that comes with the standard GATE distribution. It detects a number of named entities, such as persons, locations, organizations, etc.

To configure the ANNIE NLP service in your installation, perform the following two steps:

1. Start GATE and create an ANNIE pipeline "with defaults". ANNIE, together with its components, will load automatically (for details on ANNIE, please refer to GATE's user's guide).

2. Store the created GATE pipeline under `Resources/GatePipelines` using GATE's *Save Application State* menu function and name it (exactly) as `Annie.gapp`.

An OWL service description for this pipeline is already implemented and stored in the directory `Resources/OwlServiceDescriptions/annie.owl`. For details on the OWL-based description format, please refer to Section 5.3.

Yahoo Search. This service performs a Web search for a user and returns the first n documents (where n is a user-configurable parameter). To create the corresponding GATE pipeline, you need to follow these steps:

1. Start GATE and create a new pipeline through *File* → *New application* → *Pipeline*. Note that you must select "Pipeline" and not any of the other options ("Corpus Pipeline" etc.).
2. Start the CREOLE plug-in manager with *File* → *Manage CREOLE plug-ins*, scroll down to "Web_Search_Yahoo", select "Load now" and hit "OK".
3. Create a new Yahoo PR component through *File* → *New processing resource* → *YahooPR*. Note that you will need a Yahoo API key for the "applicationID" (see the GATE manual for details).
4. Insert the newly created Yahoo PR into the Pipeline (double-click on the pipeline, and add the PR from the left to the pipeline on the right). If you have problems with this step, please consult the GATE manual for basic instructions on how to use the GATE GUI.
5. Save the pipeline as above with the name `Yahoo.gapp`.

Web IR Extractor. The third example service shows how to combine two existing services, by first calling the Yahoo IR service and then using the search results as input to the ANNIE IE service. This service is located in `yahooExtractor.owl`.

2.7 Client Installation

The installation and configuration of clients is covered in Chapter 4.

Command-Line Client: For information on how to compile and run the command-line client, please refer to Section 4.1.

The OpenOffice.org Writer Plug-In: For details on how to compile and run the OpenOffice.org Writer plug-in please refer to Section 4.2.2.

Chapter 3

The Semantic Assistants Server

3.1 Starting the Server

Type `ant run` in the `SemanticAssistants/Server` directory to start the server. Please refer to Section 2.5 for more installation and compilation details. The server will automatically load all available OWL service descriptions from the default location `Resources/OwlServiceDescriptions` and publish these to the clients.

3.2 Testing using the Command Line Client

To test if the server is running correctly and can be accessed from the clients, we recommend you run some tests using the command-line client described in Section 4.1.

3.3 Integrating New NLP Services

For the server to know how to handle the different NLP services offered through the architecture, it needs a *description* of each offered service. These are by default located in the `SemanticAssistants/Resources/OwlServiceDescriptions` directory. The GATE pipelines corresponding to these service descriptions are located (by default) in `Resources/GatePipelines`. The language service descriptions are ontologies, building on the *SemanticAssistants.owl* ontology, which, in turn, extends the *ConceptUpper.owl* ontology. Both of these are located in `ont-repository` under `SemanticAssistants/Resources`.

In order to create a new language service description, it is often easier to copy an old one and edit it. Protégé[1] is helpful as an ontology editor. Most important is to define the parameters that can be passed to this language service, as well as the description of the results that should be passed back to the client.

In summary, to integrate a new NLP service, two steps are necessary:

1. Store the GATE pipeline implementing the service under `Resources/GatePipelines` (using GATE's *Save Application State* menu function).

2. Develop an OWL service description for this pipeline. For details on the OWL NLP description format, please refer to Section 5.3.

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

Chapter 4

Semantic Assistants Clients

4.1 Command-Line Client

This is a simple example client to access the server from the command line. It is located under SemanticAssistants/Clients/CommandLine.

1. To compile: *ant compile*
2. To run: *./runclient.sh*

The runclient.sh script helps with the class path setting, but also adds some difficulty with getting quotes right when passing parameters to the program. For example, to list all available services, you can run

```
./runclient.sh listall
```

to query the server for all available NLP services. For the default installation, you should see an output like:

```
Retrieving service info from server...   done
Listing services:
Yahoo Search
IR Information Extractor
Person and Location Extractor
```

Now you can invoke one of the services. For example, to extract all person and location entities from a Wikipedia article, you can run

```
./runclient.sh invoke "\"Person and Location Extractor\"" \

"docs=http://en.wikipedia.org/w/index.php?title=Christiane_Kubrick&printable=yes
"
```

If everything works, you will see the raw service response (in XML format). Note again that the server has to be running and both the CSAL and command-line client must have been compiled successfully.

Connecting to any Server

The user is able to specify the Server information (Host and Port) of a local or distant server. To achieve that the params part of the command needs to be used. The only extra info needed is appending the following string to the end of the command:

```
"params=(Host=<target Host>,Port=<target server port>)"
```

For example:

```
"params=(Host=localhost,Port=8080)"
```

This parameter list has to be added at every invocation.

4.2 OpenOffice.org Writer Plug-In

The OpenOffice.org application suite offers a mechanism to add application extensions, or plug-ins. We used this mechanism to integrate OpenOffice.org's word processing application Writer with our architecture, and thus equip the Writer with Semantic Assistants (Gitzinger and Witte, 2008).

Our primary goal for the Writer extension was to be able to perform text analysis on the current document. This text can, for instance, be a large document from which information should be extracted, or a problem statement consisting of a few questions, which serves as input for a question-answering (QA) Semantic Assistant. Especially for the last use case, it must allow a user to highlight part of a document (e.g., a question) and be able to pass only the highlighted part as input to a language service. Furthermore, the extension must offer the possibility to specify parameters that need to be passed to a selected NLP service.

An OpenOffice.org plug-in is basically a zip file with specific contents and certain descriptions of these contents. For a detailed description of the implementation please refer to Section 4.2.3. **Note:** The current version of the plug-in requires at least OpenOffice.org Version 3.1.

4.2.1 Features

Our plug-in creates a new menu entry "Semantic Assistants:"

In this menu, the user can inquire about available services, which are selected based on the client (here *Writer*) and the language capabilities of the deployed NLP services (described in service metadata, see Section 5.3). The dynamically generated list of available services is then presented to the user, together with a brief description, in a separate window, as shown in Figure 4.1. Note that the integration of a new service does not require any changes on the client side—any new NLP service created and deployed by a language engineer is dynamically discovered through its OWL metadata maintained by the architecture and so becomes immediately available to any connected client.

The user can now select an assistant and execute it. In case the service requires additional parameters, such as the length of a summary to be generated, they are detected by our architecture through the OWL-based service description and requested from the user through an additional dialog window. An example, for the *Web Retrieval Summarizer* assistant, is shown in Figure 4.2. After the service is executed, the result is displayed in Writer depending on the type of the server response: either as a new document, as annotations on the existing document, or by opening an external viewer (e.g., a Web browser for HTML documents).

Side-Notes View

The latest release of the OpenOffice.org Suite offer a new feature for text annotation. Depending on the annotation results received from GATE, the Semantic Assistants Writer plug-in presents it in a sidenote manner (see Figure 4.3).

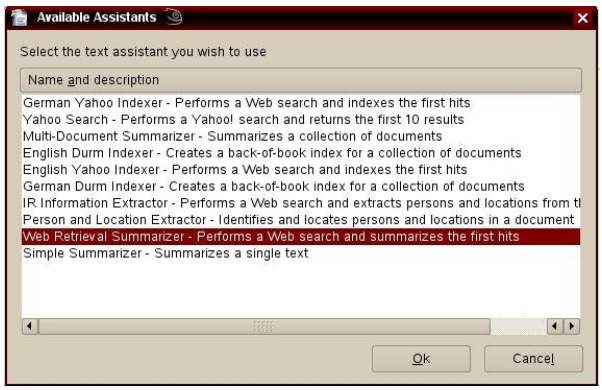


Figure 4.1: List of available semantic assistants

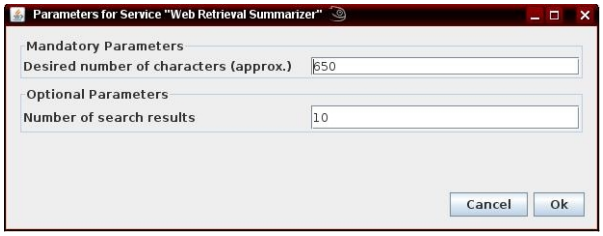


Figure 4.2: The parameters dialog, which appears when a Semantic Assistant requiring further input is invoked

New Document Creation

Creation of a new document comes handy when the output of an NLP service corresponds to a complete document, or the result itself is indivisible. Some examples are summarization or question-answering (see Figure 4.4).

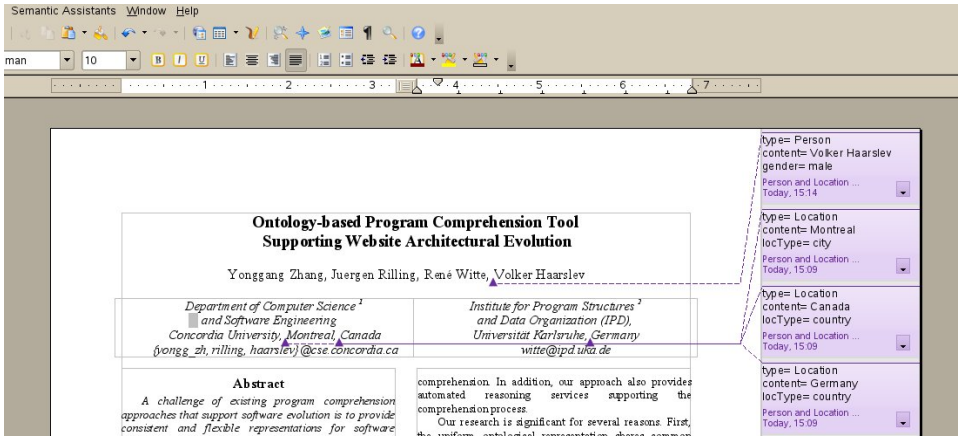


Figure 4.3: Auto-Generated SideNotes Example

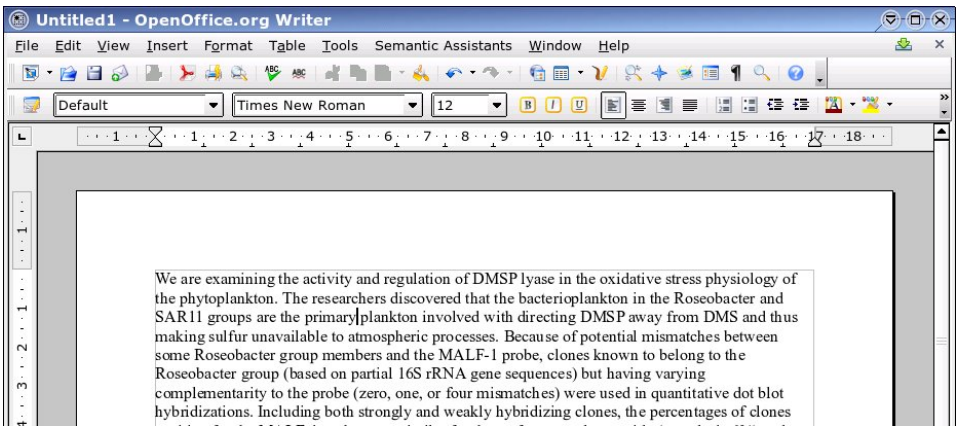


Figure 4.4: NLP services can also create a completely new document as a result (e.g., through summarization)

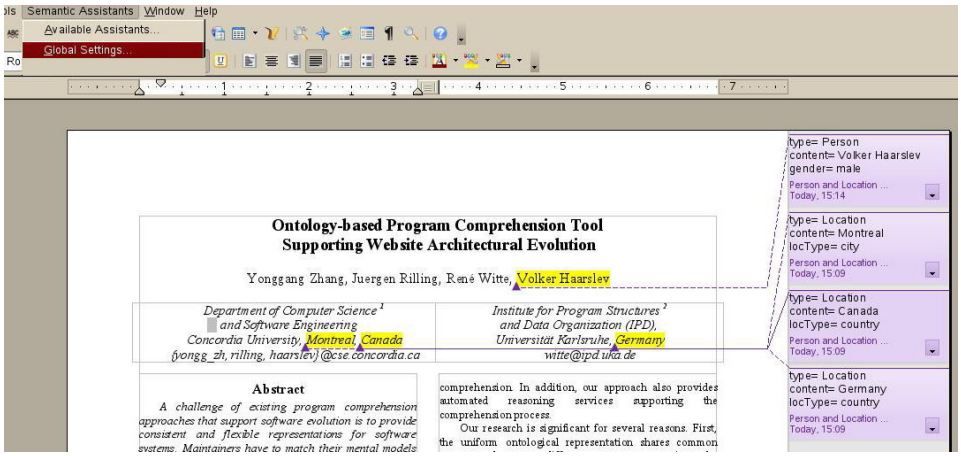


Figure 4.5: Highlighted Annotations Example

Annotation Highlighting

Besides text annotation, we offer the option for enabling/disabling annotation highlighting for text that has been processed by GATE. This option can be found under the Semantic Assistants menu in "Global Settings." See Figure 4.5) for an example.

Server Settings

Another option found under the Semantic Assistants menu in "Global Settings."is *Server Setings*. There the user is able to specify the Server information (Host and Port) of a local or distant server.

4.2.2 Installation

The OpenOffice.org plug-in can be found in `SemanticAssistants/Clients/OpenOffice`. To use it, do the following:

1. Start OpenOffice.org Writer and ensure the right Java VM is used. Go to *Tools* → *Options*. Under *OpenOffice.org* you will find a *Java* item. There, you can specify JREs.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE manifest:manifest PUBLIC
"-//OpenOffice.org//DTD Manifest 1.0//EN" "Manifest.dtd">
<manifest:manifest
 xmlns:manifest="http://openoffice.org/2001/manifest">
  <manifest:file-entry
    manifest:media-type=
      "application/vnd.sun.star.configuration-data"
    manifest:full-path="Addons.xcu"/>
  <manifest:file-entry
    manifest:media-type=
      "application/vnd.sun.star.configuration-data"
    manifest:full-path="ProtocolHandler.xcu"/>
  <manifest:file-entry
    manifest:media-type=
      "application/vnd.sun.star.uno-component;type=Java"
    manifest:full-path=
      "ProtocolHandlerAddon_java.uno.jar"/>
</manifest:manifest>
```

Figure 4.6: The *manifest.xml* file for our plug-in

If it is not already there, add the currently used Java version.

2. In the same dialog window, click on the *"Class path..."* button. You will get a new dialog window. Select *"Add archive..."* and navigate to the `SemanticAssistants/CSAL/dist` directory and add the `CSAL.jar` file. Close the open dialogs with "OK" and exit Writer.

3. cd to the `Clients/OpenOffice` directory.

4. Type `ant run`, or `ant run-gui`. Note that the client-side abstraction layer must have already been built and packaged. Both `ant run` and `ant run-gui` provide an UNO package named `Semassist.uno.zip`. Both targets additionally copy it to `~/Documents/uno-components`. If `ant run-gui` is issued the OpenOffice.org *Extension Manager* will pop up and prompt the user to install the extension. If `ant run` is issued the above process is automated. After the installation, OpenOffice Writer starts with the plug-in installed.

   Note: you can also manually add the plug-in from within OpenOffice (skip this step if you already used the *run* or *run-gui* target): Go to Tools, Extension Manager. Select *My Extensions*, then click *Add...* on the right. Choose the UNO package (available in `~/Documents/uno-components` if you used the *deploy* target for ant).

5. Leave the dialog and open a text document. You should have a menu bar entry labelled *Semantic Assistants*. Now play around. Remember the server must be running to be able to query or execute language services.

4.2.3 Development Notes

In this section, we provide further technical details on our plug-in for developers interesting in enhancing or modifying it.

OpenOffice.org Plug-in Specifics

Every plug-in has to include a *META-INF* directory, which contains a file called *manifest.xml*. This XML file lists the elements that come with this plug-in; The concrete manifest file for our plug-in is listed in Figure 4.6. We can see that it defines three *file-entry* elements specifying the type and location of the following files:

```
1
2  private static XComponent CreateNewDocument( XDesktop xDesktop,
3                                  String sDocumentType )
4  {
5        ...
6  }
7
8  private static void AnnotateField( Annotation annotation )
9  {
10       ...
11       // Use the text document's factory to create an Annotation text field
12       XTextField xAnnotation = (XTextField) UnoRuntime.queryInterface(
13             XTextField.class, mxDocFactory.createInstance(
14             "com.sun.star.text.TextField.Annotation" ) );
15
16       // get the properties of the field
17       XPropertySet xPropertySet = (XPropertySet) UnoRuntime.queryInterface(
18                                    XPropertySet.class, xAnnotation
19 );
20
21       ...
22
23       // Highlight annotated field
24       HighlightField();
25 }
26
27 private static void HighlightField()
28 {
29 ....
30
31 }
```

Figure 4.7: Core methods implementing the OpenOffice Writer plug-in features are part of the `UNOUtils` class

Addons.xcu. This XML file defines how the plug-in should be integrated with OpenOffice.org. In our case, it contains a menu definition, specifying that the menu should only appear in the *Writer* application. For each menu item, we specify which messages should be broadcast throughout the OpenOffice.org runtime system when the menu item is activated.

ProtocolHandler.xcu. This XML file specifies that the messages defined in *Addons.xcu* should be handled by an object of a certain class. This class is provided in the Java archive and must adhere to a certain interface.

ProtocolHandlerAddon_ java.uno.jar. This Java archive contains the actual functionality of the plug-in. It holds classes responsible for receiving the messages generated by the menu items, as well as classes responsible for the interaction with the client-side abstraction layer.

Implementation Details

A useful class called `UNOUtils` found in the `packageinfo.semanticsoftware.semassist.client.openoffice.utils` contains most of the OO-Writer feature implmentations. More specifically, the three methods in Figure 4.7 implement a major part of the above described features (Side-Notes, Highlighting and New Document Creation).

More details on how to compile and debug an OpenOffice plug-in can be found in Section 5.2.3.

4.3 Eclipse Plug-in

Eclipse is not a single monolithic program, but rather a small kernel containing a plug-in loader surrounded by hundreds of plug-ins. The behaviour of each plug-in in this architecture is stored in its code, and its dependencies and services are declared in the plug-in's manifest file. On each Eclipse startup, the plug-in loader scans all the available manifest files in the Eclipse exclusive plug-in folder and then builds a structure containing this information.

We used this characteristic of Eclipse architecture to integrate our Semantic Assistants architecture into the Eclipse environment in form of a plug-in, in order to offer various Natural Language Processing services. The Eclipse Semantic Assistants plug-in is basically a Java archive (JAR) file that ships with its own specific content and a description file to introduce itself to the Eclipse plug-in loader.

4.3.1 Features

Once the Semantic Assistants plug-in is installed, it creates a new menu entry in the Eclipse menu toolbar. A user can inquire about the available services from the "Available Assistants" item and modify the connection settings to the Semantic Assistants server by selecting the second item.

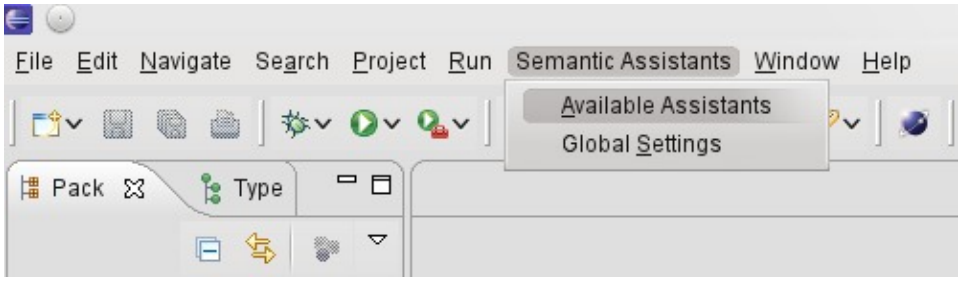


Figure 4.8: Semantic Assistant Menu in Eclipse

Available Assistants

Selecting the "Available Assistants" item from Semantic Assistants menu will open a file selection dialog. The file selection dialog allows the user to select the desired files, folders and even complete projects to send to the server as inputs to an NLP service.

This dialog also lets the user to select an NLP service from a dynamically generated list of services. This list is generated dynamically by selecting the available services based on the client and the language capabilities of the deployed NLP services. Note that the integration of a new service does not require any changes on the client side - any new NLP service created and deployed by a language engineer is dynamically discovered through its OWL metadata maintained by the architecture and so becomes immediately available to any connected client.

Upon selecting the resource files and the desired service, the user can executed the selected service on the checked files in the tree. After the successful execution of the selected NLP service, a set of results with be shown to the user in the "Semantic Assistants" view. If the service execution fails, a description of the occured error will be shown to the user in the "Semantic Assistant Status" view and invocation will be aborted.

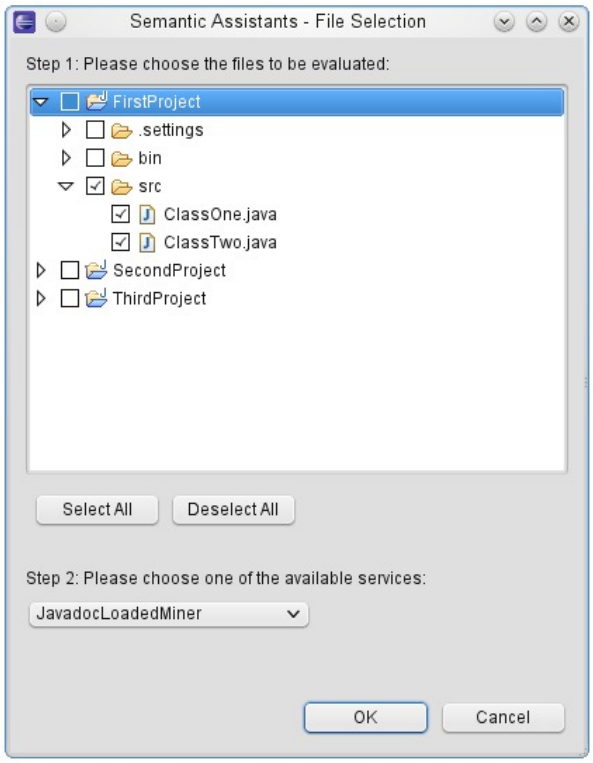


Figure 4.9: File Selection Dialog

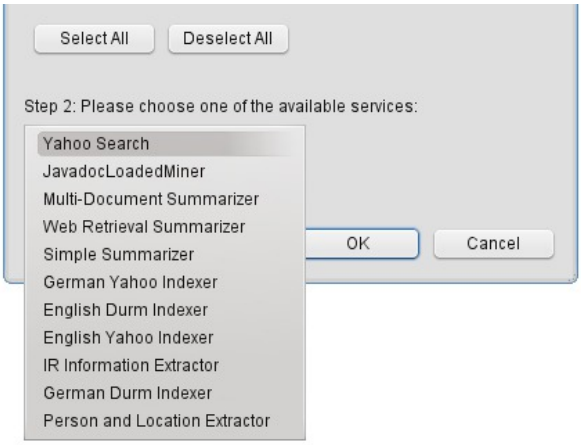
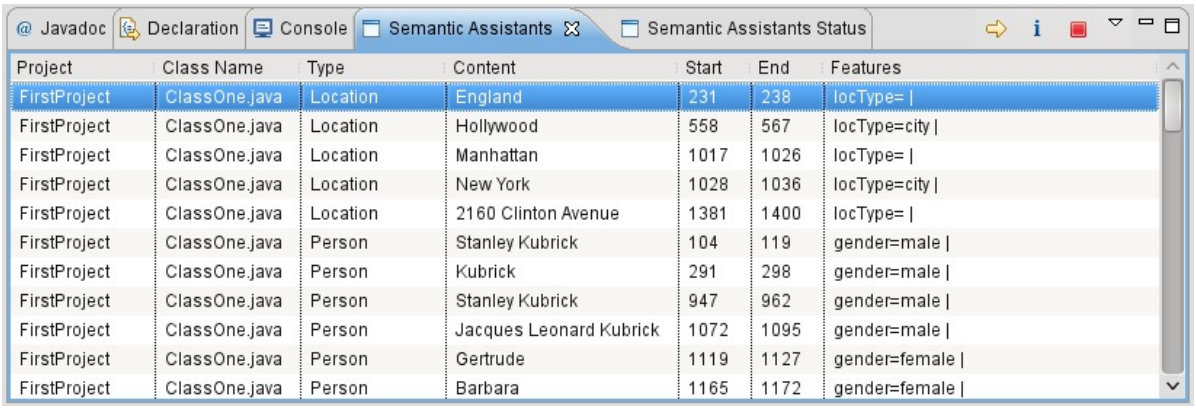


Figure 4.10: A List of Available NLP Services

Annotations

The results of a successful service invocation are being shown to the user in an Eclipse view part called "Semantic Assistants" view. In the mentioned view, a table will be generated dynamically based on the server response that contains all the parsed annotation instances. In Figure 4.11 the result of an execution of the "Person and Location Extractor" service on

two sample classes is shown.



Project	Class Name	Type	Content	Start	End	Features
FirstProject	ClassOne.java	Location	England	231	238	locType= |
FirstProject	ClassOne.java	Location	Hollywood	558	567	locType=city |
FirstProject	ClassOne.java	Location	Manhattan	1017	1026	locType= |
FirstProject	ClassOne.java	Location	New York	1028	1036	locType=city |
FirstProject	ClassOne.java	Location	2160 Clinton Avenue	1381	1400	locType= |
FirstProject	ClassOne.java	Person	Stanley Kubrick	104	119	gender=male |
FirstProject	ClassOne.java	Person	Kubrick	291	298	gender=male |
FirstProject	ClassOne.java	Person	Stanley Kubrick	947	962	gender=male |
FirstProject	ClassOne.java	Person	Jacques Leonard Kubrick	1072	1095	gender=male |
FirstProject	ClassOne.java	Person	Gertrude	1119	1127	gender=female |
FirstProject	ClassOne.java	Person	Barbara	1165	1172	gender=female |

Figure 4.11: Semantic Assistants View

This table can be sorted by different criteria through clicking on each column title. Additionally, by double-clicking on each row in the table, the selected annotation will appear with a graphical representation attached to the corresponding resource. For instance, in Figure 4.12 the JavadocMiner service has been invoked on a Java source code file. Some of the annotations returned by the server bear a *lineNumber* feature, which attaches an annotation instance to a specific line in the java source file. Upon double-clicking on the annotation instance in the Semantic Assistant view, the corresponding resource - here, a .java file - will be opend in an editor and an Eclipse warning marker will appear next to the line defined by the annotation *lineNumber* feature.

Global Settings

The second option found under the Semantic Assistants menu is the "Global Settings" item. By clicking on this menu item, a dialog will be shown to the user that lets him customize the hostname and port number values used to connect to the Semantic Assistants server. The values provided by the user in the textfields will be saved to a properties file in the Eclipse workspace ".metadata" folder and will be used to store useful information between different Eclipse launch profiles. The ".metadata" folder can be found on the root of user's Eclipse workspace folder. Note that this folder is a hidden folder and is not meant to be viewed or modified by the users. The Semantic Assistants plug-in folder can be found on

```
${workspace_loc}
/.metadata/.plugins/info.semanticsoftware.semassist.client.eclipse
```

where the

```
${workspace_loc}
```

refers to the path of the workspace of the Eclipse installed on the user's machine.

If this folder gets accidentally deleted by the user, the Semantic Assistants plug-in will create a new one on the next startup of the plug-in.

In the Semantic Assistants settings dialog, a "Use Defaults" option is also available. If this widget is checked, the default values will be overwritten on the values previosuly provided by the user and the Semantic Assistants plug-in will use the default hostname and port number values to connect to the server and invoke services.

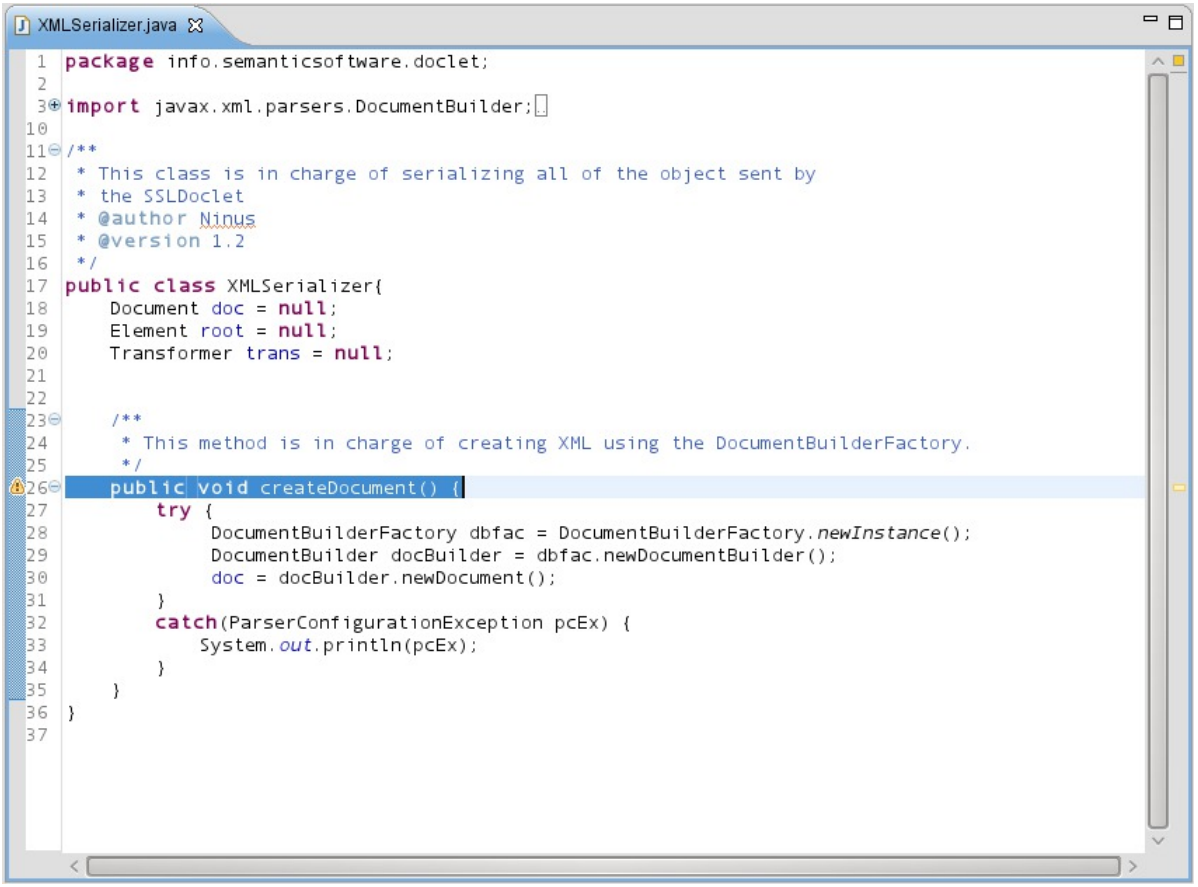


Figure 4.12: A Semantic Annotation

4.3.2 Installation

In order to use the Eclipse plug-in, do the following:

1. Make sure that the client-side abstraction layer is already been built and packaged.

2. cd to the `Clients/EclipsePlugin` directory.

3. Type `ant`. The ant file will compile all the classes and package the plug-in JAR file in `EclipsePlugin/plugins` folder.

4. Browse to your Eclipse installation folder and paste the generated JAR file from previous step into the `plugins` folder.

5. Start the Eclipse application. The Eclipse plug-in loader will automatically install the plug-in for you. If the plug-in is installed successfully, you should be able to see the Semantic Assistants menu added to you toolbar.

   Note: Remember the server must be running to be able to query or execute language services.

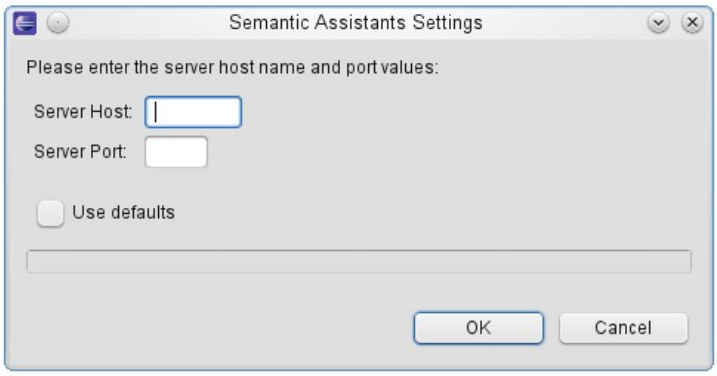


Figure 4.13: Semantic Assistants Server Settings Dialog

4.3.3 Development Notes

In this section, we provide further technical details on our plug-in for developers interesting in enhancing or modifying it.

Plug-in Directory Structure

1. `lib`: This is where all the external plug-in dependencies are stored. This folder contains the Client-Side Abstraction Layer JAR file used in the project to communicate to the server.
2. `META-INF`: This is where the plug-in manifest file is stored. This file is used by the Eclipse plug-in loader to identify the Semantic Assistants plug-in.
3. `plugins`: This is where the Semantic Assistants plug-in JAR file will be stored when successfully built.
4. `src`: This folder contains the Semantic Assistants plug-in source code. The internal structure of this directory is further explained in the next section.

There is also another file called `plugin.xml` in this directory. This file contains general information of the plug-in like its dependencies, runtime classpath and extension points in form of an XML file and is used to introcude the plug-in to Eclipse plug-in loader. By using this file, you can test the plug-in inside an Eclipse runtime application.

Plug-in Source Directory Structure

The Semantic Assistants plug-in uses Model-View-Controller pattern for its implementation. Thus, most of the source code files are divided into different packages related to their responsibilities. When you browse to `src/info/semanticsoftware/semassist/client/eclipse/` folder, you see the following structure:

1. `dialogs`: This folder contains the dialogs that are shown to the user for interactions e.g. file selection. The classes inside this folder are the codes for graphical user interfaces.
2. `handlers`: This folder contains the classes which play the controller role in MVC pattern. Examples of these classes are dialog handlers and service invocation jobs.
3. `model`: This folder contains the classes that provide data for graphical user interfaces. These models are consumed by classes inside `views` package.

4. `utils`: This folder contains utility classes e.g. logging feature.

5. `views`: This folder contains the source codes for Semantic Assistants view parts. These are again graphical user interfaces but packaged differently from dialogs.

There is also another file called `Activator.java` in the source directory. It is the main class of the plug-in that will be loaded initially and control the plug-in's life cycle.

Modifying the Plug-in

If you want to modify the plug-in behavior or enhance it, follow these steps:

1. Open the Eclipse application.

2. Select File → New → Project and under the "Plug-in Development" category select the "Plug-in Project".

3. A new plug-in project wizard will open up. Keep the EclipsePlugin name for the project. Just below the project name there is a checkbox reading "Use Default Location". Uncheck it and browse to the EclipsePlugin folder inside where you've stored the Semantic Assistants folder.

4. Leave the other options untouched and press Finish.

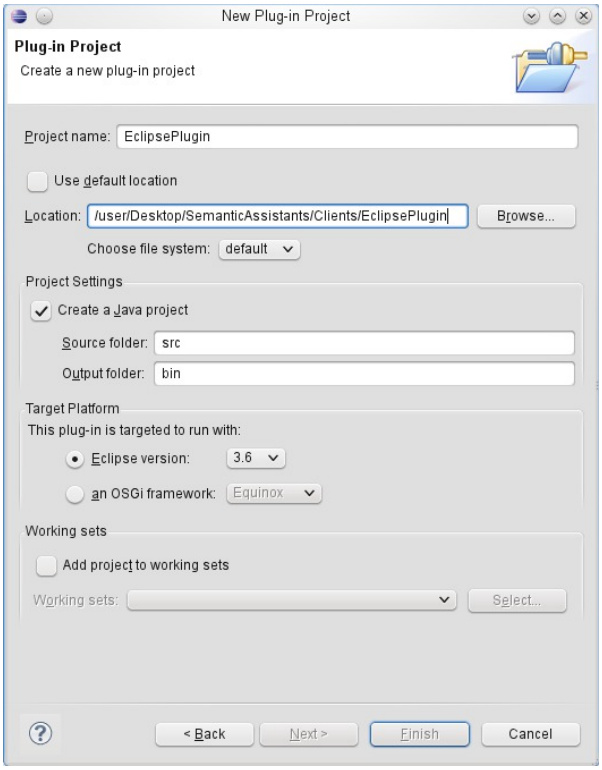


Figure 4.14: Eclipse New Plug-in Project Wizard

Note: Remember you should manually copy the CSAL.jar file into you the project's `lib` folder because it is a part of the project's dependency and is defined in the classpath.

When the project is loaded in your workspace, feel free to play around. Browse the source directory and add your development codes. To run your code, right click on `plugin.xml` file and select Run As → Eclipse Application.

Chapter 5

Developer Notes

In this chapter, we provide additional information for developers.

5.1 Generic Compilation Instructions

All parts of the Semantic Assistants architecture (server, CSAL, clients) come with ant build scripts that allow compilation from the command line (usually `ant compile` or `ant run`).

5.2 NetBeans Development

The project has been developed using the NetBeans IDE.[1] Debugging is made possible for all elements of the project, the steps will be described in this section. **Note:** The following steps have been validated with NetBeans IDE 6.7.

5.2.1 Open the Project Workspace

1. Open the NetBeans IDE.
2. Select *File* from the Menu Bar then *Open Project*.
3. From the Dialog that pops-up navigate to the *SemanticAssistants* Directory and open *Server* and *CSAL*.
4. Repeat the previous step and navigate to the *SemanticAssistants/Clients* Directory and open *OpenOffice* and *CommandLine*

5.2.2 Run a Project through NetBeans

1. Open the *Files* Window (*Ctrl-2*) and the list of the open project will appear.
2. Collapse the files for a given project
3. To *compile/dist/run/clean* or any other action, *R-Click* to the *build.xml*, select *Run target* and select one of the list.

[1] NetBeans IDE, http://netbeans.org/.

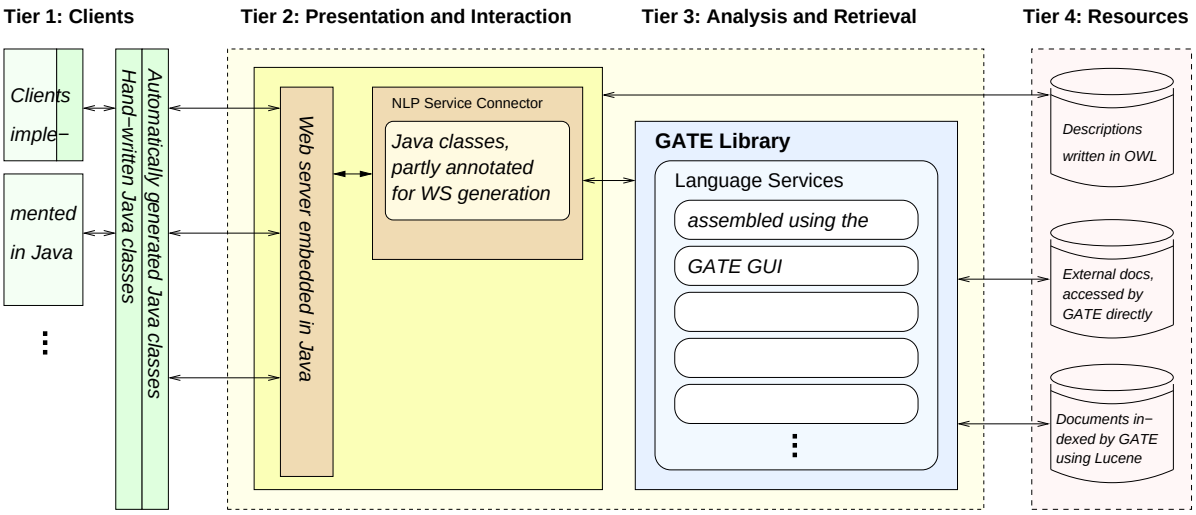


Figure 5.1: The architectural overview with implementation notes

5.2.3 Debug a Project with the NetBeans debugger

Currently the projects: *Server*, *CSAL* and *OpenOffice* are being built with debug info and the developer has to follow the following steps to attach the debugger in order to set breakpoints and pause the execution of any given projects. **Note:** To configure OpenOffice Writer to listen for incoming debugger connections please refer to the subsection *Configure the OpenOffice Writer to Run in Debug Mode* below.

1. To debug a running process in the system, the developer should go to *Debug* from the Menu bar and select *Attach Debugger*. A window will pop up.
2. Set the following arguments to the pop-up window:
   - **Debugger**: Java Debugger (JPDA)
   - **Connector**: Socket Attach (Attaches by socket to other VMs)
   - **Trasport**: dt_socket
   - **Host:** The local machine's IP, e.g. *Host: localhost*
   - **Port:** The port that the process is listening. The value is found and changed at the build.xml. The *Server* is listening at *8888*.
3. Click *OK*. The process that the debugger was attached, now runs in *Debug Mode*. Repeat the above steps every time you run the project.

Configure the OpenOffice Writer to Run in Debug Mode

This Section describes how to configure the JAVA VM in OpenOffice Writer to accept incoming connection for a debugger.

1. Open OpenOffice Writer
2. Go to Tools, Options. Under *OpenOffice.org*, there is a *Java* item. Select it and then Click on the *Parameter* button. There the parameters when running the JAVA VM are set.
3. To run in debug mode *Assign* the following 2 parameters:
   - **-X debug**

- **-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=7081**

4. **Note: The** *address=7081* **should be the consistent with the port set in *Attach Debugger* within NetBeans**
5. Now OO Writer is ready to accept connections from the debugger

5.3 Service Descriptions

NLP services are described with metadata in OWL format. The ontology format provides for expressive service description that captures users, their language capabilities, tools, services, and result formats. This allows connected clients to *recommend* services that are suitable for the current user, task, and output capabilities (e.g., a cell phone has quite different output capabilities from a netbook or desktop system).

5.3.1 Context and Service Representations

Clients can request a list of available language services from the server. Rather than simply returning all available services (which could be a long list), we want to be able to *recommend* possibly useful NLP services to the user, based on her context. In other words, services that of no use are omitted, e.g., because of language reasons or missing capabilities of the currently used client. As a first, minimal representation of the user's context, we enable a client to communicate the languages that its user knows, as well as the language of the document or documents in question. This representation is recorded in Table 5.1.

Field	Meaning	Example
User languages	The languages the user knows	<en,de,fr>
Document language	The language of the current document	en

Table 5.1: Minimal representation of the user context

For the client, the point of requesting a list of available language services is to know which language services exist, and how it can invoke them. Thus, each description sent by the server contains the name of the language service it represents, so that the client can identify it. Secondly, services might contain user-configurable parameters, so the returned service description also contain a list of parameter representations. Finally, for practical reasons, the description holds a short, plain text description to make it clearer what the language service achieves (examples can be seen in Figure 4.1). Together, this gives us the NLP service representations shown in Table 5.2

Field	Meaning	Example
Service name	The name of the language service	*Single-document summarizer*
Service description	Short description of what the service achieves	*Creates a summary of a single document*
Parameter list	List of descriptions illustrated in Table 5.3	See Table 5.3

Table 5.2: Representation of an NLP service

While service name and service description are simple strings, the parameter list needs further clarification. This parameter representation holds the parameter's name, its data

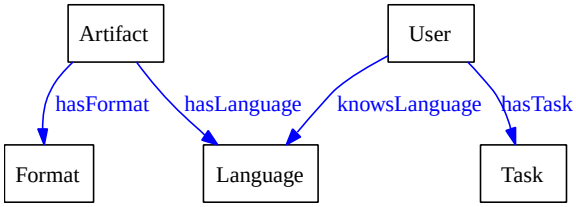


Figure 5.2: The five central concepts of the common upper ontology

type, and the information whether the parameter is mandatory or optional. Table 5.3 shows its data fields in detail. In addition to the mentioned fields, we added a *Parameter value* field to be written by the client, which it can use to announce parameter values for a subsequent NLP service invocation. To increase the usability, we added a *Label* and a *Description* field that allows a system integrator to add some explaining words to the parameter representation that are, e.g., helpful for user interface integration. The same holds for the *Default Value* field, which can be used to give the user an idea of what sensible parameter values are.

Field	Meaning	Example
Parameter name	The (codified) name of the parameter	*outputFormat*
Parameter type	The data type of the parameter	*string*
Optional	Is the parameter optional or mandatory?	*no*
Parameter value	The value the parameter should take for the following invocation. Field to be written by the client.	*mediawiki*
Parameter description	A short description of the meaning of the parameter, or advice on its usage	*Use "mediawiki" or "html"*
Label	A more expressive name suitable for display in user interfaces	*Output format*
Default value	A suitable standard value for the parameter	*html*

Table 5.3: Representation of a parameter for an NLP service

5.3.2 The Semantic Assistants Ontology

In this section, we provide details on the design of the ontology used for NLP service descriptions in the Semantic Assistants architecture.

The Semantic Assistants Upper Ontology

The Semantic Assistants upper ontology contains five core concepts to model the relationships between users, their tasks, the artifacts involved and their format and language. Figure 5.2 shows these five concepts and Table 5.4 provides some descriptions and examples.

Note that, while artifacts must have a format, they are not required to have a language. Format information helps us to differentiate between various kinds of artifacts, e.g., it enables us to tell a GATE annotation from an executable program. We consider this kind of information essential and relatively easy to provide – even if there is no obvious format, or

Concept	Description	Example/sub-concept
Artifact	The artifact is the central parent concept for all kinds of objects like documents, source files, compiled files, programs, NLP services, parameters, and annotations. Each field of application will most likely introduce its own artifacts.	Document, Tool, Parameter
Format	Every artifact must have a format. For our language service infrastructure, it is important to know what formats the input and output of a service have in order to handle them correctly.	PDF, HTML
User	In order to respond to a user's context, we also have to have some knowledge on the user himself. Which languages does he know? What is he working on?	EnglishLearner
Language	Languages are orthogonal to formats. Just as it is important to know if a file is binary or text, it is important to know if it is a Java source file or a manual written in HTML, or an article written in Spanish. NLP services are often language specific.	English, French, Java
Task	Tasks further connect users and services. A task expresses a user's goal. If we know that goal, and we know what each of our offered services are good for, we can match the two pieces of information and, ideally, provide the user with services that are useful to him.	TranslationTask

Table 5.4: The five central concepts of the upper ontology

the format is uncertain, we can most likely still rely on generic formats like "plain text file" or "binary file". On the other hand, it is not always intuitive to give a language for a given artifact. E.g., we would hesitate to say that a GATE annotation specifying a part of speech has a certain language. Likewise, it is not very intuitive to say that a runtime parameter has a language. Due to these difficulties, we leave the language as an optional attribute for artifacts.

More Concrete Artifacts. *Tools*, like an NLP tool, are a subconcept of *artifacts*. A tool possibly processes artifacts as input and can produce artifacts as output. These are modeled with the *consumesInput* and *producesOutput* relations in our ontology. Moreover, we mentioned parameters that a tool can have, which is modeled by the *hasParameter* relation. For our domain, *documents* are of great importance, in particular natural language documents. These documents all have languages (English, French, Java, etc.), modeled by the *hasLanguage* relation. As natural language documents are very common and important, we introduce a sub-concept of *Document* called *NaturalLanguageDocument*, which has a *hasNaturalLanguage* property.

In order to work with documents, they often must somehow be identified and retrieved. In fact, the Semantic Assistants server must be able to pull documents for analysis from the Internet. Thus, when we model such an input document in our ontology, we must have a way to specify its URI (Uniform Resource Identifier) by which we can address it. As not only documents, but also other artifacts like Web services have to be uniquely identified and addressed, we introduce *hasIdentifier* as an optional property for artifacts. We define *IdentifiableArtifact* as a class whose members are artifacts and have this property. In practice, the

identifier can, and often will, be a URI, but it does not have to be. For example, if we have a set of elements with unique names, a simple string can be enough. With *hasIdentifier*, we provide an important property on the highest level, while leaving the exact semantics to the concrete ontologies and the applications that use them.

Output Modelling. We introduced one more concept that is not as obvious as the concepts introduced so far. Suppose there is an NLP system that can, among others, output XML and OWL files. We would model artifacts representing these file types, and create the according *producesOutput* relation pairs. However, which one of the output variants is produced most likely depends on the way the NLP system is invoked. Therefore, we have to somewhere store the information, under which *circumstances* a certain output is produced. We could store it directly with the individual representing the output. However, we feel that this information concerns more the relationship between the system and the output, not the output itself. Thus, we introduced a concept called *IOArtifact* where information on input and output relationships can be stored. We will see a more concrete use of this concept in the following section, when we concretize the upper ontology. By means of an *isActualArtifact* relation, we have *IOArtifact* individuals "point" to *Artifact* individuals. They can be seen as a proxy for artifacts.

We can see the artifact class, its sub-classes, and important relations between these classes in Figure 5.3. A textual overview is given in Table 5.5.

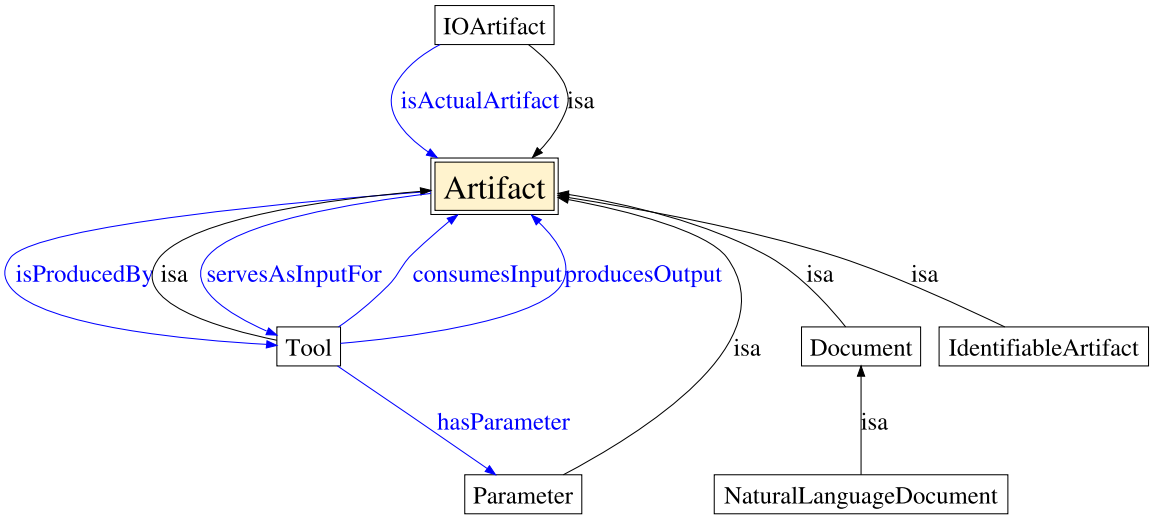


Figure 5.3: The artifact base class with its subclasses

We have not yet mentioned the relations *servesAsInputFor* and *isProducedBy*. They are the inverse relations of *consumesInput* and *producesOutput*, respectively, and have been defined for reasons of completeness.

5.3.3 Specializing the Upper Ontology

The upper ontology that we have just introduced gives us several things we need: artifacts, users, parameters, tools, etc. However, we still lack some important NLP-specific concepts, which is why, in this section, we substantiate the abstract upper ontology and refine it into an ontology for language services. In particular, we include concepts specific for the NLP

Concept	Description	Example
Tool	Parent concept for all processing artifacts. Typically consumes some input and/or produces some output.	Single Document Summarizer, POS tagger
Document	Artifact with a language (mandatory)	Web document, Java source code file
Natural Language Document	Document with a *hasNaturalLanguage* property	English Manual, French Novel
Identifiable Artifact	Artifact that has some identifiable string, e.g., a URI	Web document, GATE processing pipeline
Parameter	Parameter for a tool, typically with some type information	Output format, desired summary length
IOArtifact	Proxy for an input or output artifact, holding additional information on the relationship between the artifact and the consuming/producing tool	Plain text output proxy, HTML output proxy

Table 5.5: Sub-concepts of the *Artifact* concept

subsystem in use by the Semantic Assistants architecture, GATE. Integrating a different NLP subsystem, like UIMA, would require an alternative specialization focusing on UIMA specific concepts and their relations.

New Tools and Artifacts. We can now add a core concept for the Semantic Assistants architecture, NLP services. They are modeled as child concepts to the *Tool* concept, classifying language services in two categories: *IRTool* and *NLPTool*. The semantics that we want to convey by this separation is that an information retrieval tool (*IRTool*) finds documents, but leaves them untouched, while an NLP tool processes documents and typically generates some new artifact(s) from them. For NLP tools, input and output natural languages can be specified. However, they do not have to be specified, as there are also language-independent language services. Therefore, if no input language is specified, we will assume that the language service can handle any input language. Likewise, if no output language is specified, we will assume that it can handle any output language.

Additionally to *IRTool* and *NLPTool*, we have one more child concept named *GATEPipeline*. While the two first concepts imply certain semantics, the latter is defined solely via its *Format*, which has to be a certain GATE pipeline format. The semantics of a *GATEPipeline* are defined by its also being an instance of one of the other two concepts. This means that a *GATEPipeline* can be an *NLPTool* or an *IRTool*.

The *GATEPipeline* concept is obviously GATE specific: A sequence of processing components is called *pipeline* in GATE. In this ontology, we do not model these processing components explicitly, as an end user is not interested in single components of language services, but in language services as a whole. Instances of *GATEPipeline* are the language services we offer the user through our architecture.

Having covered the *Tool* extensions of our concrete ontology, let us have a look at the artifacts they are associated with. As mentioned, we need a semantic description of several GATE data structures. Accordingly, we introduce new concepts as children of the *Artifact* concept. *GATECorpus* represents a collection of documents, which typically serves as input

for a language service. *GATEAnnotation* instances describe the information which language services add to a document. Annotations are organized into annotation sets, hence we provide *GATEAnnotation* with an attribute specifying to which annotation set a given instance belongs to. GATE annotations can have features, which are, as presented in Section 5.4.2, included in our response messages. Therefore, *GATEAnnotation* instances can have an unlimited number of *hasFeature* attributes, which are simple strings. *GATERuntimeParameter* models parameters that control certain aspects of a GATE pipeline, and is introduced as a sub-concept to the already existing *Parameter* concept. Along with these artifact types, we have to introduce according new formats (remember that artifacts are required to have a format). These are defined under the common parent concept of *GATEFormat*, which is a child of the *Format* concept.

Conditional Output. The output of a tool can vary depending on the input or on parameters: Suppose there is a language service that produces an index of its input documents, similar to the ones found at the back of many books. Suppose further that this indexer has a parameter *outputFormat*, which can be set to *plain text* or *html*. Therefore, the NLP service is modelled to have two different output artifacts, a plain text file and an HTML file. However, with the parameter settings available, only one of them is produced at any single invocation of the NLP service, and our server should know which one. The solution to this is provided through the *IOArtifact* concept already introduced, or, more specifically, through a sub-concept thereof named *GATE_OutputArtifact*. Instances of *GATE_OutputArtifact* have a property *necessaryParameterSetting*, thus connecting an output artifact to a certain parameter value. This parameter value is represented through an instance of a newly introduced concept called *ParameterValuePair*. Therefore, in the example of the indexer, the plain text output file would be associated with a *ParameterValuePair* instance holding the right parameter value for plain text output, and the HTML output file would be associated with one holding the value for HTML output. The server would then compare the parameter value sent by the client to these values, and thus know which output can actually be expected.

Concepts and Relationships. The most important additional concepts and relations of the more concrete ontology are shown in Figure 5.4, and also listed in Table 5.14. Concepts and relations that are already present in the upper ontology are drawn more faintly, and some of them have been omitted entirely to keep the diagram clearer.

The relation called *urlGivenByParameter* connecting a GATE output artifact to a GATE runtime parameter remains to be explained. Often, when language services produce a file as output, they offer the user an option to specify where this file should be stored. In our architecture, our server has to take advantage of this so that it can take the file and pass it on to the requesting client. The *urlGivenByParameter* relation informs the server which parameter it can set to specify the desired output destination of an output file.

We have noted that GATE pipelines contain several processing resources, each of which can, in theory, take parameters. Obviously, the parameter values sent by the client must be assigned correctly, i.e., to the correct processing resources. To satisfy this requirement, *GATERuntimeParameter* instances hold an attribute called *prName*, the "pr" standing for "processing resource". Additionally, parameters hold a *paramType* and a *defaultValue* attribute.

Concatenation of Language Services. Language services can also be concatenated. In order to do this, we introduced an attribute to the *GATEPipeline* concept, called *concatenationOfPipelines*. If a language service is actually a concatenation of several GATE pipelines. This attribute holds a comma-separated list of the according pipeline names. We chose this flat data type approach over object relations because it is an easier way of guaranteeing the correct order of the language services. OWL does not support ordering in an obvious way, and

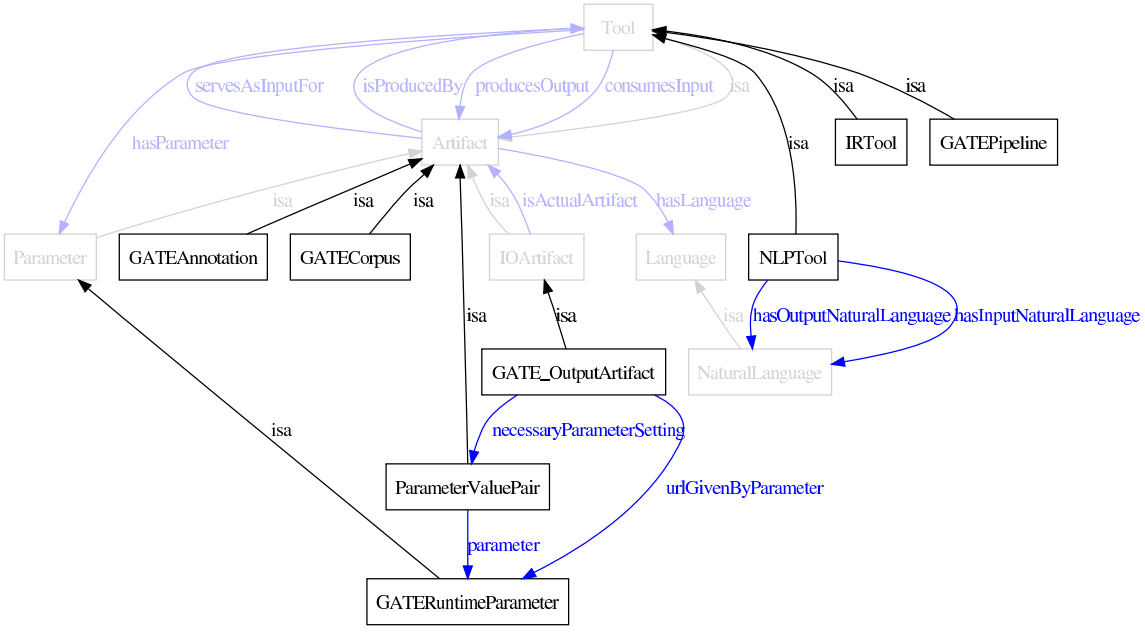


Figure 5.4: Additional classes introduced in the specialized ontology

although a design pattern for modelling sequences exists, we felt that our simple approach was sufficient for the situation at hand.

Querying the Semantic Assistants Ontology

The ontology described so far now contains the information needed to dynamically find, load, parametrize, and execute available language services, based on user's current task and language capabilities. In our implementation, it is queried using Jena's SPARQL[2] interface, using the context information delivered by the client plug-in, in order to recommend applicable Semantic Assistants.

For example, when a recommendation request is received, with a context object saying the user knows English and German, the generated SPARQL query should restrict the available services to those that deliver English or German as output language. A simplified version of such a generated query is shown below:

```
SELECT ?x ?name
WHERE { ?x sa:hasGATEName ?name .
       {?x cu:hasFormat sa:GATECorpusPipeline_Format} . {
          {?x sa:hasOutputNaturalLanguage cu:en} UNION
             {?x sa:hasOutputNaturalLanguage cu:de}}
}
```

Once the SPARQL query has been generated, it is passed to the `OntModel` instance containing the language service descriptions. The results are then retrieved from this object, converted into the corresponding client-side versions, and returned to the client.

[2]SPARQL Query Language for RDF, see http://www.w3.org/TR/rdf-sparql-query/

Concept	Description	Example
IRTool	Instances of this concept compile a collection of documents, usually in response to some query.	*YahooSearch*
NLPTool	Instances of *NLPTool* process or analyze documents, and usually produce new information from them.	*Summarizer, Person Extractor*
GATE Pipeline	The GATE specific implementation of a language service.	*Summarizer, Person Extractor*
GATE Annotation	GATE annotations represent linguistic and semantic information that has been added to a document. They can have *hasFeature* attributes.	*PersonAnnotation, LocationAnnotation*
GATE Corpus	A collection of documents, usually serving as input for a language service.	*StandardGATECorpus*
GATE Runtime Parameter	Many language services accept parameters that influence their behaviour. Type, default value, and other information can be specified for a parameter.	*outputFileParameter, summaryLengthParameter*
GATE Output Artifact	Holds additional information on the relationship between a GATE pipeline and its possible output	*HTMLOutputArtifact, PlainTextOutputArtifact*
Parameter Value Pair	Can be used to specify certain values for a parameter. Usually used in combination with *GATEOutputArtifact* and the *necessaryParameterSetting* relation.	*htmlSetting, plainTextSetting*

Table 5.6: Concepts newly introduced in the concrete ontology for language services

5.4 Service Invocation and Result Passing

In this section, we provide details on the communication between the clients and the server in the Semantic Assistants architecture.

5.4.1 Service Invocation

Clients must be able to pass a single document as well as multiple documents as input to an NLP service. The Semantic Assistants architecture allows to pass documents either literally or via a URI. In order to implement these requirements while maintaining a uniform way of service invocation, a client must pass two lists to the server: The first list contains one URI per document passed to the language service. Thus, the number of URI entries in this list equals the number of documents passed. If a document is passed via URI, then its URI entry in the first list is simply its URI. If the document is to be passed literally, we put a special URI that is not a URL in the first list, which acts as a signal. In that case, the second list comes into play. The second list contains one entry for each document that is passed literally, and each entry is the document text itself. Thus, when the server encounters a URL in the first list, it can use this URL to access the document and when a special URI is given,

Property Name	Domain	Range	Description
consumesInput	Tool	Artifact	Lists the GATERuntimeParamter(s) of a language service
hasFormat	Artifact	Format	Specifies the format of a language service (GATEPipeline_Format)
hasParameter	Tool	Parameter	Lists the GATERuntimeParamter of a language service
producesOutput	Tool	Artifact	Specifies the output of a language service (e.g. GATEOutput_Artifact)

Table 5.7: Object Properties for Artifact

Property Name	Domain	Range	Description
hasLabel	Artifact	xsd:string	Specifies the label to be shown to clients
appFileName	GATEPipeline	xsd:string	The actual filename of the GATE application, without path, e.g. Durm-Indexer
hasGateName	Artifact	unset	Specifies the name of the Artifact.
mergeInputDocuments	Artifact	xsd:boolean	Specifies if input documents are to be merged into one before they are passed to this language service
publishAsNLPService	Tool	xsd:boolean	Specifies if an NLP tool should be published to client(s)

Table 5.8: Datatype Properties for Artifact

the server advances to the next entry in the second list and finds the document contents there. Obviously, order is important when we use this algorithm. The use of the two lists is illustrated in Figure 5.5, where four input documents are passed. The first and the fourth document are passed via URL, while the second and the third are passed literally.

Besides the name and the input for a language service, we might need the client to provide a list of parameters. Therefore, a list of parameters as described in the previous section is passed along. The *Value* field of a parameter holds the user-specified value this parameter should be assigned. Finally, we enable the communication of user context by passing a representation thereof, as introduced in the previous section. Table 5.15 lists all the data elements transmitted on language service invocation.

Property Name	Domain	Range	Description
hasFormat	Artifact	Format	Specifies the format of a language service (Standard_GATEAnnotation_Format)

Table 5.9: Object Properties for GATEAnnotation

Property Name	Domain	Range	Description
hasFeature	GATEAnnotation	xsd:String	Specifies the additional features of an annotation
hasGATEName	Artifact	xsd:String	The name of the Annotation

Table 5.10: Datatype Properties for GATEAnnotation

Property Name	Domain	Range	Description
hasFormat	Artifact	Format	Specifies the format of a language service (Standard_GATEAnnotation_Format)
isActualArtifact	IOArtifact	Artifact	Specifies the actual GATEAnnotaiton Individual
isProducedBy	Artifact	Tool	The Artifact that prodcues the output (e.g. GATE pipeline)

Table 5.11: Object Properties for GATEOutputArtifact

While language services can most likely be invoked without the current user context at hand, knowing some facts can prove useful before or after the invocation. Thus, the invocation could be cancelled if it is detected that the chosen language service does not fit the input document(s), or one could tailor the generated output to the user client's preferences or capabilities.

5.4.2 Language Service Results

Once a language service has finished its work, we want to collect its results and pass them, in an appropriate format, on to the client. The Semantic Assistants architecture allows for different result formats (document annotation, new documents, results files such as ontologies) and passes them in a uniform XML response format back to the client.

The basic structure of our response message is rather simple. No matter what the contents of a message are, they are always be enclosed in a single `saResponse` element, the "sa" standing for "Semantic Assistant". Within this enclosing element, the results produced by the language service are listed one by one. In Figure 5.6, this is illustrated for a language service which produces two different results. Here, they are called *result1* and *result2*. In reality, they might be an annotation and a file, or two different annotations, etc.

Property Name	Domain	Range	Description
hasGATEName	Artifact	xsd:string	The name of the GATEOutputArtifact
isPerDocument	GATE_OutputArtifact	xsd:boolean	Specifies is this artifact produced for each input document, or is it valid for a whole input corpus.

Table 5.12: Datatype Properties for GATEOutputArtifact

Property Name	Domain	Range	Description
hasFormat	Artifact	Format	Specifies the format of a language service (Standard_GATE_RTP_Format)
servesAsInputFor	Artifact	Tool	Specifies the tool (e.g. pipeline) that the parameter serves as input for.

Table 5.13: Object Properties for GATERuntimeParameter

Property Name	Domain	Range	Description
hasLabel	Artifact	xsd:string	Specifies the label to be shown to clients
isOptional	IOArtifact or Parameter	xsd:boolean	Specifies if the runtime parameter for the processing resource is optional.
defaultValue	GATERuntimeParameter	xsd:string	The name of the GATEOutputArtifact
hasGATEName	Artifact	xsd:string	The name of the GATERuntimeParameter
paramType	GATERuntimeParameter	owl:oneOf{"string" "double" "int" "boolean" "url" "list(string)" "list(double)" "list(int)" "list(boolean)" "list(url)"}	Specifies The type of a parameter. Use string, double, int, or boolean for now.
prName	GATERuntimeParameter	xsd:string	Specifies the name of the processing resource the parameter belongs to.

Table 5.14: Datatype Properties for GATERuntimeParameter

Result from GATE Annotations. With the outer structure of a response message in place, we have to properly define how specific results are represented. We take care of annotations first, especially annotations in GATE (see the GATE documentation for a description of the GATE document data model if you do not know about document annotations). Instead of `result1` or `result2` in the example above, we will thus have `annotation` tags. An important

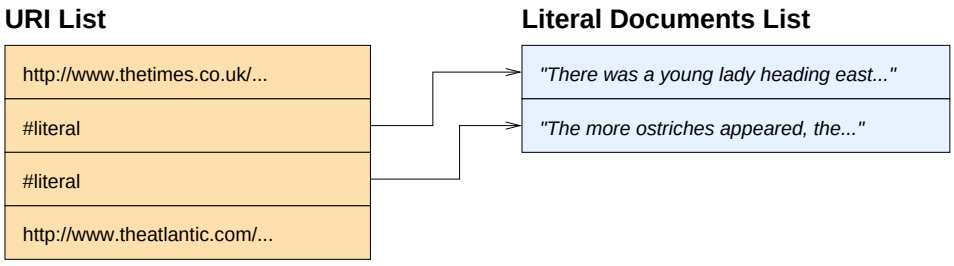


Figure 5.5: The two lists representing the input documents for an NLP service

Field	Meaning	Example
NLP service name	The unique identifier of the desired language service	*English Single Document Summarizer*
URI list	A list of actual input document URIs and special URIs signalling that this document is passed literally. Every input document is represented here.	See Fig. 5.5
Document text list	List of document texts, only of those documents that are passed literally	See Fig. 5.5
Parameter list	Transports the parameter values that the client specifies	See Table 5.3
User context	The user context, as specified in Section 5.3.1	*(userLangs=en,fr docLang=en)*

Table 5.15: Data fields sent from the client when invoking an NLP service

```
<?xml version="1.0"?>
<saResponse>
   <result1>
      ...
   </result1>
   <result2>
      ...
   </result2>
</saResponse>
```

Figure 5.6: Schematic structure of an example response with two results

aspect of an annotation obviously is its name, or type, e.g., *Protein* or *Person*. Additionally, for GATE annotations, we include the annotation set that the annotation is contained in. Thus, we have an `annotation` element with `type` and `annotationSet` attributes.

There might be several input document to an NLP service, and an annotation is always associated with a specific document. Therefore, the response first lists all input documents within each `annotation` tag, using `document` tags. If the URL of a document is known, it is included in the corresponding `document` tag. However, if a document has been passed literally, we do not have a URL, so we cannot include it in the response message either. This seems to be a problem when several documents are passed literally: How does the client know which result belongs to which document? However, the order of the documents is maintained as it was when the client sent them as input. Therefore, the client is able to correctly assign the results to the input documents.

For every document, we then list the occurrences of the annotation in question, using `annotationInstance` elements. Instances hold a `start` and an `end` attribute, which are character offsets and which enable the client to locate the instance in the input text. Moreover, there is a `content` attribute, which holds the text located between the `start` and `end` text positions. Finally, annotations can hold an arbitrary number of so-called features. Features that the system integrator thinks could be of interest are listed as child elements of the `annotationInstance` elements. We can see an example output with three different annotations (*Date*, *Person*, and *Location*) and one document in Figure 5.7. The *Person* and *Location* annotations each have one feature listed – *gender* in the case of the person, *locType* in the case of the location. The document has been passed via URL, which is why this URL is also

included in the response.

```
<?xml version="1.0"?>
<saResponse>
  <annotation type="Date" annotationSet="">
    <document url="http://www.thetimes.co.uk/...">
      <annotationInstance content="yesterday" start="76" end="85">
      </annotationInstance>
      <annotationInstance content="today" start="1056" end="1061">
      </annotationInstance>
      <annotationInstance content="past ten years" start="6477" end="6491">
      </annotationInstance>
    </document>
  </annotation>
  <annotation type="Person" annotationSet="">
    <document url="http://www.thetimes.co.uk/...">
      <annotationInstance content="Tony Blair" start="65" end="75">
        <feature name="gender" value="male" />
      </annotationInstance>
      <annotationInstance content="Rupert Murdoch" start="2357" end="2371">
        <feature name="gender" value="male" />
      </annotationInstance>
      <annotationInstance content="Mr Blair" start="3133" end="3141">
        <feature name="gender" value="male" />
      </annotationInstance>
    </document>
  </annotation>
  <annotation type="Location" annotationSet="" >
    <document url="http://www.thetimes.co.uk/...">
      <annotationInstance content="Iraq" start="1510" end="1514">
        <feature name="locType" value="country" />
      </annotationInstance>
      <annotationInstance content="Downing Street" start="4576" end="4590">
        <feature name="locType" value="" />
      </annotationInstance>
    </document>
  </annotation>
</saResponse>
```

Figure 5.7: Result example with three annotations and their detected instances

Result Files. NLP services can also generate new files as a result – these can be of any format, depending on the concrete components deployed in the pipeline. Instead of embedding the file itself (which can be quite large) directly into the response, we only pass an identifier for the file, along with some format information. The client can evaluate this information, and, if it decides to do so, request the result file itself by using the identifier sent to it. The server then sends the actual file. Again, we have an example response in Figure 5.8. The `outputFile` element holds format information, both in form of a MIME type string and a human readable one. The identifier for the result file is given through the `url` attribute which refers to the server's file system.

Formal Response Message Format Definition. The complete response format definition DTD can be found in Figure 5.9. It defines documents of the type `saResponse`. The `saResponse` element can contain arbitrarily many `annotation` elements and `outputFile` elements. Annotations, annotation instances, features, and output files are defined accordingly.

```
<?xml version="1.0"?>
<saResponse>
  <outputFile url="file:/tmp/serviceResult-..."
          mimeType="text/html" format="HTML Document" />
</saResponse>
```

Figure 5.8: Result example with one resulting HTML file

```
<!DOCTYPE saResponse [

  <!ELEMENT saResponse ( annotation*, outputFile* ) >

  <!ELEMENT annotation ( document+ ) >
  <!ATTLIST annotation annotationSet CDATA #IMPLIED >
  <!ATTLIST annotation type NMTOKENS #REQUIRED >

  <!ELEMENT annotationInstance ( feature* ) >
  <!ATTLIST annotationInstance content CDATA #REQUIRED >
  <!ATTLIST annotationInstance end NMTOKEN #REQUIRED >
  <!ATTLIST annotationInstance start NMTOKEN #REQUIRED >

  <!ELEMENT document ( annotationInstance* ) >
  <!ATTLIST document url CDATA #IMPLIED >

  <!ELEMENT feature EMPTY >
  <!ATTLIST feature name NMTOKEN #REQUIRED >
  <!ATTLIST feature value CDATA #REQUIRED >

  <!ELEMENT outputFile EMPTY >
  <!ATTLIST outputFile format CDATA #REQUIRED >
  <!ATTLIST outputFile mimeType CDATA #REQUIRED >
  <!ATTLIST outputFile url CDATA #REQUIRED >

]>
```

Figure 5.9: The document type definition (DTD) for our response messages

We have now discussed the most important data exchange processes. In the next section, we want to ask the question where this data on NLP services comes from, and how it is organized.

5.5 Developing a New Client for the Semantic Assistants Architecture

We have now covered the most important and interesting implementation issues, and are ready, with the knowledge we have, to give step-by-step instructions to connect a client application to our architecture, and thus to the functionality of NLP services. These instructions depend a bit on the client implementation language. As we described earlier, our implementation of the client-side abstraction layer consists of Java classes packaged in a single Java archive (`.jar`) file. Hence, if the client application is able to make use of Java archives, connecting to our architecture is done as follows:

1. Have your application import the Java archive containing the implementation of the client-side abstraction layer (CSAL).

2. If necessary, tell the CSAL the address of the Web service endpoint. The CSAL classes that need to know the address have a default value for it.

3. Create a `SemanticServiceBrokerService` object, which serves as a factory for proxy objects.

4. Create such a proxy object. This is your “remote control” to the Web service. You can call all methods that have been published through the Web service on this object.

After these four steps, a Java-enabled client application can get lists of available language services as well as invoke these language services. A code example, where an application obtains such a proxy object and invokes the `getAvailableServices` method on it to find available language analysis services, is shown below:

```
// Create a factory object
SemanticServiceBrokerService service = new SemanticServiceBrokerService();
// Get a proxy object, which locally represents the service endpoint (= port)
SemanticServiceBroker broker = service.getSemanticServiceBrokerPort();
// Proxy object is ready to use. Get a list of available language services.
ServiceInfoForClientArray sia = broker.getAvailableServices();
```

A client application developer who cannot use the Java archive that implements the CSAL still has access to the WSDL description of our Web services. If there are automatic client code generation tools available for the programming language of his choice, the developer can use these to create CSAL-like code, which can then be integrated into or imported by his application. In this case, the integration steps would be similar to the steps enumerated above, with the additional step of generating client-side code from the WSDL description before the first step.

If there are no such code generation tools, the client application has to implement the communication with the server itself. To do this, it must create the SOAP messages that represent method invocations, and send them to the Web service endpoint. Also, it must receive the response messages from the server, and extract the relevant information in them. Usually, however, there are standard libraries that facilitate these tasks.

Bibliography

Thomas Gitzinger and René Witte. Enhancing the OpenOffice.org Word Processor with Natural Language Processing Capabilities. In *Natural Language Processing resources, algorithms and tools for authoring aids*, Marrakech, Morocco, June 1 2008. URL http://rene-witte.net/enhancing-openoffice-writer-with-NLP.

René Witte and Thomas Gitzinger. Semantic Assistants – User-Centric Natural Language Processing Services for Desktop Clients. In *3rd Asian Semantic Web Conference (ASWC 2008)*, volume 5367 of *LNCS*, pages 360–374, Bangkok, Thailand, February 2–5 2009. Springer. URL http://rene-witte.net/semantic-assistants-aswc08.