NeOn: Lifecycle Support for Networked Ontologies

Integrated Project (IST-2005-027595)

Priority: IST-2004-2.4.7 – "Semantic-based knowledge and content systems"

D6.4.2 NeOn core infrastructure services

Deliverable Co-ordinator: **Walter Waterfeld**

Deliverable Co-ordinating Institution: **Software AG (SAG)**

Other Authors: Diana Maynard, University of Sheffield (USFD), Ian Roberts, University of Sheffield (USFD), Michael Gesmann, Software AG (SAG)

Document Identifier:	NEON/2010/D6.4.2/v1.0	Date due:	October 31st, 2009
Class Deliverable:	NEON EU-IST-2005-027595	Submission date:	January 31st , 2010
Project start date:	March 1, 2006	Version:	v1.0
Project duration:	4 years	State:	Final
		Distribution:	Public

2006–2010 © Copyright lies with the respective authors and their institutions.

NeOn Consortium

This document is a part of the NeOn research project funded by the IST Programme of the Commission of the European Communities by the grant number IST-2005-027595. The following partners are involved in the project:

Open University (OU) – Coordinator Knowledge Media Institute – KMi Berrill Building, Walton Hall Milton Keynes, MK7 6AA United Kingdom Contact person: Enrico Motta E-mail address: e.motta@open.ac.uk	**Universität Karlsruhe – TH (UKARL)** Institut für Angewandte Informatik und Formale Beschreibungsverfahren – AIFB Englerstrasse 11 D-76128 Karlsruhe, Germany Contact person: Andreas Harth E-mail address: aha@aifb.uni-karlsruhe.de
Universidad Politécnica de Madrid (UPM) Campus de Montegancedo 28660 Boadilla del Monte Spain Contact person: Asunción Gómez Pérez E-mail address: asun@fi.upm.es	**Software AG (SAG)** Uhlandstrasse 12 64297 Darmstadt Germany Contact person: Walter Waterfeld E-mail address: walter.waterfeld@softwareag.com
Intelligent Software Components S.A. (ISOCO) Calle de Pedro de Valdivia 10 28006 Madrid Spain Contact person: Jesús Contreras E-mail address: jcontreras@isoco.com	**Institut 'Jožef Stefan' (JSI)** Jamova 39 SI-1000 Ljubljana Slovenia Contact person: Marko Grobelnik E-mail address: marko.grobelnik@ijs.si
Institut National de Recherche en Informatique et en Automatique (INRIA) ZIRST – 655 avenue de l'Europe Montbonnot Saint Martin 38334 Saint-Ismier France Contact person: Jérôme Euzenat E-mail address: jerome.euzenat@inrialpes.fr	**University of Sheffield (USFD)** Dept. of Computer Science Regent Court 211 Portobello street S14DP Sheffield United Kingdom Contact person: Hamish Cunningham E-mail address: hamish@dcs.shef.ac.uk
Universität Koblenz-Landau (UKO-LD) Universitätsstrasse 1 56070 Koblenz Germany Contact person: Steffen Staab E-mail address: staab@uni-koblenz.de	**Consiglio Nazionale delle Ricerche (CNR)** Institute of cognitive sciences and technologies Via S. Martino della Battaglia, 44 - 00185 Roma-Lazio, Italy Contact person: Aldo Gangemi E-mail address: aldo.gangemi@istc.cnr.it
Ontoprise GmbH. (ONTO) Amalienbadstr. 36 (Raumfabrik 29) 76227 Karlsruhe Germany Contact person: Jürgen Angele E-mail address: angele@ontoprise.de	**Food and Agriculture Organization of the United Nations (FAO)** Viale delle Terme di Caracalla 1 00100 Rome Italy Contact person: Caterina Caracciolo E-mail address: Caterina.Caracciolo@fao.org
Atos Origin S.A. (ATOS) Calle de Albarracín, 25 28037 Madrid, Spain Contact person: Tomás Pariente Lobo E-mail address: tomas.parientelobo@atosorigin.com	**Laboratorios KIN, S.A. (KIN)** C/Ciudad de Granada, 123 08018 Barcelona, Spain Contact person: Antonio López E-mail address: alopez@kin.es

Work package participants

The following partners have taken an active part in the work leading to the elaboration of this document, even if they might not have directly contributed to writing parts of this document:

Software AG,

University of Sheffield,

University of Koblenz

Change Log

Version	Date	Amended by	Changes
0.9	25/01/10	Diana Maynard	Added another part to Chapter 4 according to Noam's comments
1.0	26/01/10	Walter Waterfeld	Incorporated review feedback

Executive Summary

At the start of the NeOn project the focus of the NeOn Toolkit had been on a development environment for all phases of ontology engineering. Runtime for ontology-based applications was not considered and had to rely on existing runtime services.

Even from the beginning of the project some of the useful functionality for engineering activities within the NeOn Toolkit had been provided as services. These services providing ontology functionality, therefore also called ontology services, offered the desired functionality in independent implementations whereas a loose coupling via services allowed for an easy integration into the Toolkit. However, in many cases those services are not only helpful for engineering tasks, but also valuable for ontology based applications during runtime. Thus there is a need to have the ontology service functionality available both in the NeOn Toolkit as well as within the ontology-based applications.

This deliverable investigates and provides mechanisms to make existing ontology services interoperable in both environments. The usage of existing ontology services within NeOn Toolkit plugins is described. It relies on well-established mechanisms. However, the interoperability in the other direction - i.e. providing engineering plugin functionality as ontology services – is a difficult problem. We describe an elegant solution by leveraging currently not yet used capabilities of the underlying OSGi framework of Eclipse. This enables the development and publication of engineering plugins, without any change, as web services by means of appropriate infrastructure components and a plugin specific wrapping.

2006–2010 © Copyright lies with the respective authors and their institutions.

Table of Contents

1 Introduction ... 6

2 NeOn Ontology Services ... 7

3 Interoperability of engineering plugins and ontology services ... 8

3.1 Analysis of existing Eclipse plugins ... 9
3.2 Application Server ... 9
3.3 Web Service Container ... 9
3.4 Combining an application server with OSGi ... 9

4 Loosely coupled engineering plugin ... 11

4.1 Infrastructure ... 11
4.2 Example: GATE web services ... 11

5 Publish engineering plugins as web services ... 14

5.1 Conceptual Runtime Architecture based on OSGi server ... 14
5.2 Development Activities ... 16
5.2.1 Description of the specific adaptor development ... 17
5.3 Engineering Service Container ... 17
5.4 Example Service: SAIQL ... 21
5.4.1 Publish SAIQL engineering plugin as web service ... 21

6 Conclusion ... 24

7 References ... 25

List of figures

Figure 1 - Example: GATE web services 12
Figure 2 - conceptual Runtime Archtecture 14

2006–2010 © Copyright lies with the respective authors and their institutions.

1 Introduction

The main topic of the NeOn Toolkit is support of all ontology engineering phases in a network environment. When engineering end-user applications the goal is to re-use existing ontologies and benefit from the evolution of these ontologies in real time. Consequently, such existing ontologies are not replicated in a local environment; instead, they will be referenced in their primary location. For applications that eventually have to be deployed into new stages, e.g. into production scenarios, this means they will always use the latest version of such remote ontologies. For the runtime functionality needed by deployed ontology applications NeOn relied more on well-established mechanisms like web services, web applications frameworks and reasoners. However, we observe that the distinction between engineering time and runtime for ontology application is increasingly blurred. The following two simple scenarios illustrate this:

- During runtime of ontology applications a typical functionality is to retrieve individuals of an ontology, for example, via SPARQL queries. However, the same functionality is also useful during the engineering of an ontology in order to check whether a newly developed class is of use.
- On the other hand, ontology engineering tools like the NeOn Toolkit typically contain functionality to check the consistency and completeness of ontologies. However, this functionality can also be very useful during runtime of ontology applications. If, for example, the ontology is populated or updated with individuals the check for consistency is also necessary at runtime.

In the sequel we assume that applications, which either consume or provide ontology functionality, make use of ontology services. Interoperability between ontology functionality in the NeOn Toolkit and ontology services usable at runtime is required. In this deliverable we describe the chosen mechanisms in NeOn to enable this interoperability.

In the deliverable we first describe our definition of ontology services. We then discuss the relationship between NeOn Toolkit engineering plugins and ontology services, from which we derive the need for interoperability in both directions. Finally, we describe the technical solutions for the interoperability in each direction. For each of them we include an example from the NeOn Toolkit plugins.

2 NeOn Ontology Services

In the context of service-oriented architectures (SOA) there are many principles that constitute a service. Although there is no complete agreement on every detail, for the moment many of these principles can be considered as well established. We consider here only completely machine-executable services. The most common technical realisation is in the form of a web service.

When these principles are applied to ontology functionality we can conclude that ontology services have the following characteristics:

- Coarse grained interfaces

  This means an ontology service typically contains a larger portion of functionality. For instance, automatically annotating a document with an ontology is an ontology service, however, annotating a single word in a larger document is typically not a service.

- Self contained realisation

  A service must be realized and deployable in a self contained way. This means a package exists which contains all components necessary for the execution of the ontology service. This package should be easily deployable into a well defined runtime environment.

- No presentation (GUI) functionality

  The ontology service interface must not contain any presentation functionality otherwise a machine to machine communication is not possible. Consequently, and much harder to fulfil, the implementation should not contain any probably unused presentation functionality. This requires a clean separation between software components that provide on one hand presentation functionality and on the other hand application logic.

- Not interactive

  Typically, this demands for service operations following a simple request response pattern. A highly interactive functionality like an ontology document editor is therefore not an ontology service.

As these principles show, not all ontology functionality is suitable as an ontology service. Furthermore, not every realization of a suitable ontology functionality is a suitable service. For the NeOn Toolkit plugins we can say that the NeOn Toolkit engineering plugins meet these characteristics very well. In comparison, the presentation plugins are not as well aligned with these principles.

2006–2010 © Copyright lies with the respective authors and their institutions.

3 Interoperability of engineering plugins and ontology services

As already indicated the functionality provided by the NeOn Toolkit engineering plugins can be easily mapped to ontology services. Assuming that ontology services are provided as web services we can see very different usage patterns:

Ontology service

- Arbitrarily accessible via the Intranet or Internet
- Multiple parallel invocations
- Accessible by any web service client in different infrastructures

Engineering plugin

- Accessible via other NeOn Toolkit plugins, typically from an associated presentation plugin
- No parallel invocation
- Invocation only from same Eclipse workbench of one user

In order to avoid a duplicate implementation of the same functionality, just to meet both usage patterns we require interoperability of ontology services and engineering plugins. It is necessary in both directions:

1. Accessing an ontology service implementation as an engineering plugin: this very important direction we have already analysed and developed under the term 'loosely coupled engineering plugin' [see NeOn D6.9.1]. It usually does not need very specific infrastructures and is relatively easy to realize. Ontology service implementations are usually realized in a way that they can be used by many infrastructures. Eclipse plugins are, in that sense, not a special case.
2. Accessing an engineering plugin as an ontology service: this direction is not very obvious but very useful, especially for the NeOn Toolkit, where a lot of plugins are developed. It requires quite specific infrastructures.

Thus there is not a single solution for the interoperability in both directions. On the contrary, only the first direction (access ontology service from a plugin) has an obvious established solution. The second direction has to be developed completely from scratch and cannot utilize the client access to a web service of the first direction.

Developing engineering plugins accessing ontology services is very similar to the development of standard plugins. These loosely coupled plugins access an ontology service instead of realizing the same ontology functionality directly in the engineering plugin. This requires in the plugin the necessary client infrastructure to access an ontology service. Although the construction of client access to the service consists of a development and runtime component there are several established infrastructures to support the necessary steps.

For accessing engineering plugins as ontology services, we need to publish the engineering plugin implementation as a web service. For general purpose Eclipse plugins this is not intended and hardly possible. However, for NeOn Toolkit engineering plugins the realized ontology functionality fits to that purpose. Thus we have to address the technology issues. We discuss in the following

the realisation for the Neon Toolkit by further analysing Eclipse plugins and discussing the choices of important infrastructure components for services.

3.1 Analysis of existing Eclipse plugins

Plugins are normally quite different from services. The starting point of Eclipse plugins were an extensible mechanism to offer functionality for an integrated development environment. The focus of the Eclipse designers was on mechanisms for component-based development. During its history a lot of refactoring of Eclipse has been done to separate it into orthogonal components, which are all plugins now. The main principle for communication between plugins is the extension point mechanism. Additionally, plugins enforce a clear separation of interfaces and implementations via a strong dependency management.

Although some of these mechanisms are also well suited for service realisations, the plugin concept in general differs from services. For example, a plugin with an extension point often defines an interface and directly implements the usage of this interface. The plugin with the corresponding extension realizes this interface. However, the plugin with the extension point also contains the only usage of the interface. Thus, the interface will only be used by the plugin with the extension point. This is different to services, where the interface definition and its usage is completely separated.

Fortunately, however, a complete redesign of the Eclipse base layers has introduced, with OSGi [OSGi2009], a very flexible and dynamic component model. It can be considered as a good basis for a service framework. All Eclipse plugins are now based on OSGi, however, at the moment Eclipse does not use all its capabilities.

3.2 Application Server

For hosting a service an application server is usually needed. The following application servers are possible choices for java-based services.

- Tomcat: this is the standard open source servlet container. It contains basic infrastructures.
- Full Java EE server: there are several application servers, which follow the Java Enterprise Edition specification. In addition to a servlet container, this comprises a lot of other infrastructures like messaging, persistence, transactions, and registry etc. In addition, they often provide many further extensions. In sum, servers such as these are therefore very heavy-weight and require a lot of administration [Sharma2001].
- Jetty: like Tomcat, it is largely comprised of the servlet functionality and can therefore host any servlet based web applications. However, in contrast to Tomcat, it is a very lightweight server and only requires a minimum level of administration.

3.3 Web Service Container

In order to host a web service in java-based application servers, a web service container is also needed.

The primary choice here is the web service container AXIS. The current version AXIS2 provides all the necessary functionality to publish Java components as web services in a servlet container.

3.4 Combining an application server with OSGi

Given that we have OSGi-based Eclipse plugins and we have to host them in an application server, the question is how we then combine OSGi runtime with the application server. There are two implementation options:

- OSGi runtime within application server

2006–2010 © Copyright lies with the respective authors and their institutions.

This is the traditional approach; to use an application server as the hosting environment for all components. Usually, such application servers have specific requirements for or limits on the components they can host.

We have experimented with that approach by hosting the OSGi runtime Equinox in the application server Tomcat. In general it worked well but required a number of changes and imposed restrictions on the web service container which has to be hosted in the application server.

- Application server within OSGi runtime

This approach means that the resulting server process is an OSGi server which is, for a web service container, quite an unusual configuration. This is not possible for heavy weight Java EE application servers. Their administration and infrastructures require direct access of an operating system process to system resources.

However, it is possible to host very lightweight application servers like Jetty in an OSGi server. Thus, we follow this variant as the infrastructures of the heavyweight Java EE application servers are not usually required for the realisation of most ontology services. This is certainly true for our situation where the aim is to publish existing ontology functionality originally developed for an ontology engineering tool as a web service.

4 Loosely coupled engineering plugin

4.1 Infrastructure

The essence of a loosely-coupled plugin is that, rather than embedding a complex processing application within the NeOn Toolkit and running it locally, we instead host the processing logic on a remote server which exposes a web services interface. The Toolkit plugin is then a simple client for this interface. As described in D6.2.1 this model is particularly suited to:

- non-interactive components
- large grained components
- remotely used components
- components which eventually have their own (additional) repositories.

It is also an appropriate model to use when the processing logic is implemented in a language other than Java and where it would be prohibitive, either in terms of complexity or performance, to make the process run reliably on all the platforms supported by the NeOn Toolkit. The server-side part of a loosely-coupled plugin can be implemented in any language using any underlying tools, but as long as it exposes a web services interface it will be usable from the Toolkit. Additionally, when the complex process is exposed as a web service it is more easily usable by other (non-NTK) clients.

To build an engineering plugin for a loosely-coupled service requires us to generate a Java client template from the service's WSDL description using any of the standard web service client libraries (Java 6 built-in, Apache Axis2, Apache CXF, etc.) and then implement the user interface necessary to collect the required parameters, call the service using the generated client, and make its results available to the Toolkit. No further work is required for an ontology application to make use of the loosely-coupled plugin at runtime; it can simply program to the same generated client API (or generate its own from the WSDL as appropriate).

4.2 Example: GATE web services

The GATE web services plugin consists of a number of web services, each of which performs a different GATE application. These applications perform annotation on text, with respect to ontologies. The client supplies text and the service responds with semantic annotation relative to the specific ontology. The plugin contains 4 different applications:

- SARDINE: ontology population: associating textual metadata with ontology evolution by assimilating textually-derived knowledge from the fisheries domain into a new version of the deployed ontology.
- SPRAT: ontology population: a generic version of the SARDINE tool, which operates on any textual domain and does not require a seed ontology
- TermRaider: a terminology extraction tool
- COAT: collaborative semantic annotation of text with ontologies

The GATE Web Services plugin represents a typical situation where interoperability needs to be provided between two rather different kinds of systems. As described in various other NeOn deliverables (e.g. D1.5.4), NLP (Natural Language Processing) techniques can assist in the development of Semantic Web technology, providing tools such as linguistic annotation of text and ontology generation from textually-derived sources. In D6.2.1, we described various networked collaborative annotation architectures and concluded that GATE was the most appropriate of these for integration with the NeOn Toolkit, since it tracks the ISO standards and is a reference implementation for several of their components, as well as already being interoperable with other NLP Toolkits and architectures such as UIMA, LingPipe and OpenCalais. GATE does provide its

2006–2010 © Copyright lies with the respective authors and their institutions.

own ontology support, but this is quite limited in comparison with semantic web architectures and frameworks such as the NeOn Toolkit and Protege. Providing interoperability between GATE and the NeON Toolkit is therefore very useful, because it enables on the one hand NLP specialists and GATE developers to make use also of the more powerful ontology tools available in the Toolkit, and to apply the results of GATE processing as input to other Toolkit plugins. On the other hand, it enables ontology specialists to benefit from the linguistic processing offered by GATE, as a black box without having to understand anything about it.

The COAT tool is largely web-based, but a simple Toolkit plugin is provided to launch COAT from within the NeOn Toolkit for convenient access and improved visibility with users.

The other GATE web services are provided using the loosely-coupled plugin architecture, where the server side uses GATE and provides a web-services interface using the Apache CXF Toolkit.

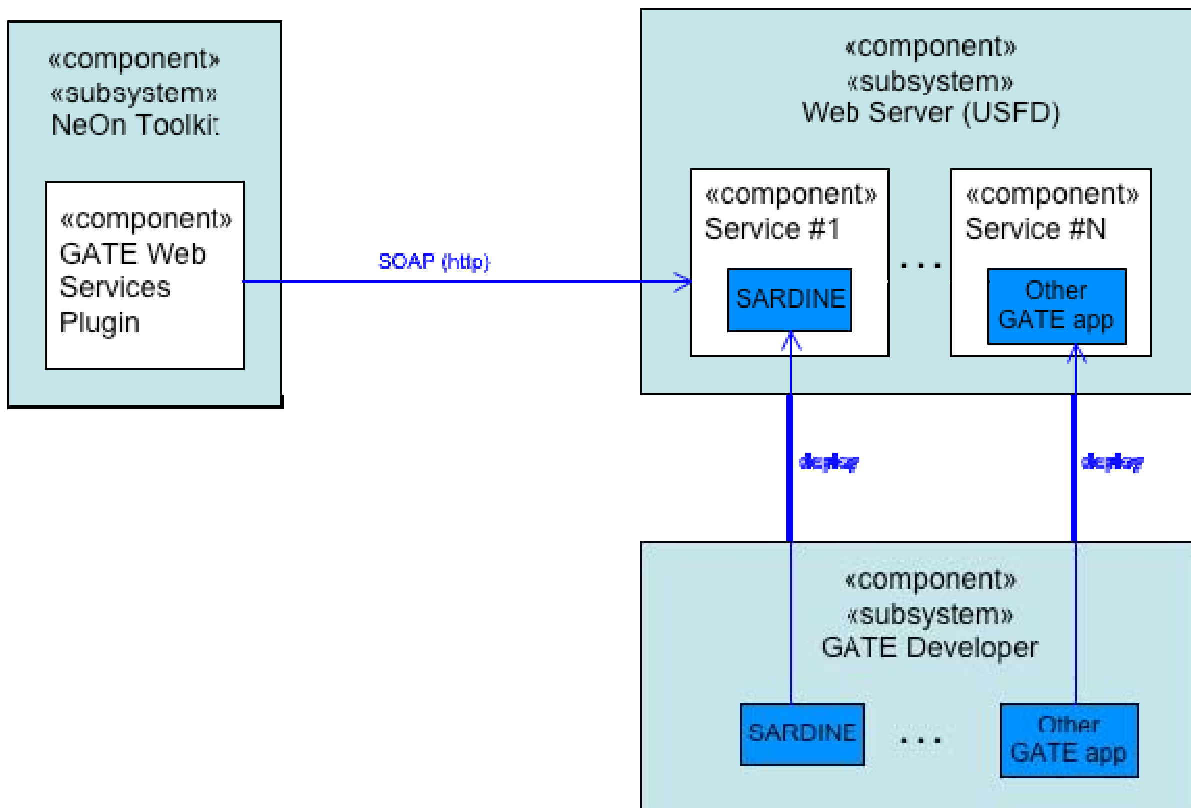

Figure 1 - Example: GATE web services

Each interaction between client and server is a series of web service calls, linked into a single session using HTTP cookies in the standard way. A client is expected to:

- Optionally send some initial data used to seed the ontology that will be populated (SPRAT only)
- Send one or more textual documents to be processed
- Retrieve the ontology produced by the GATE application (new ontology instances in the case of SPRAT or SARDINE, extracted terms for TermRaider, etc.)

The SARDINE, SPRAT, ANNIE and TermRaider applications are deployed on a Tomcat server at Sheffield.

The Toolkit plugin is a simple wrapper around this web service API. It provides a wizard to solicit the required input information from the user (source texts, seed ontology, output location) and then communicates with the web service from a Job (using the core Eclipse Jobs API). For large, complex, or numerous documents the web service processing can take some time, so using the Jobs API allows the Toolkit user to leave the job running in the background while they continue to work with the Toolkit, and they will be notified when the process completes.

The Toolkit plugin communicates with the web service using a standard Java client stub generated from the service's WSDL using Apache CXF (http://cxf.apache.org), and writes the ontology returned by the service into the output file specified by the user. Any errors are reported in the usual way by returning the appropriate IStatus from the Job, which will cause the Toolkit to report the error to the user.

The plugin also includes a preferences page to configure the list of available services that can be called (a default list is supplied by the plugin that includes the Sheffield services described above), as well as common parameters relating to the HTTP communication between client and server.

2006–2010 © Copyright lies with the respective authors and their institutions.

5 Publish engineering plugins as web services

We will leverage the OSGi infrastructure already internally used in Eclipse as the basis for publishing engineering plugins as web services.

5.1 Conceptual Runtime Architecture based on OSGi server

The conceptual runtime architecture is based on an OSGi server. It contains several types of bundles.

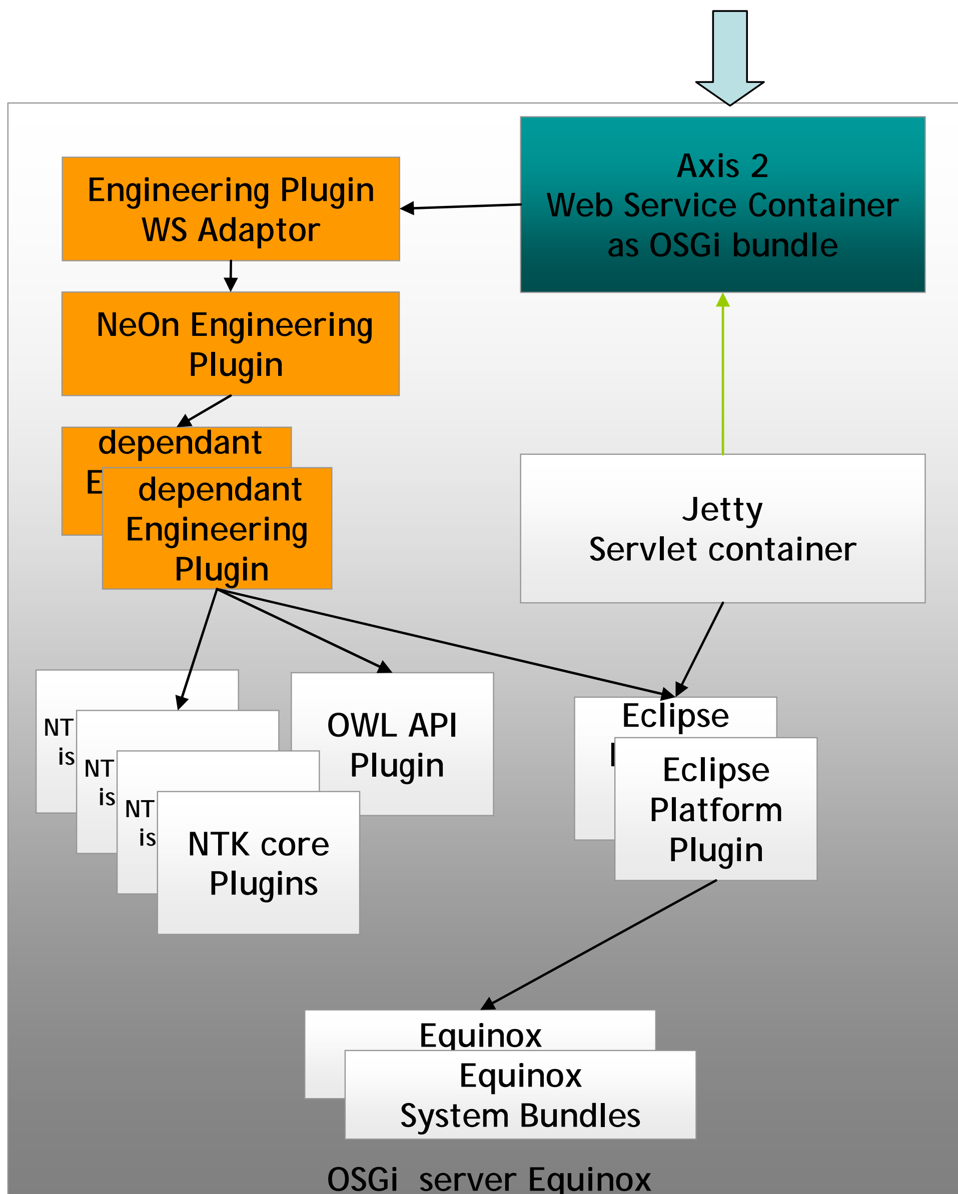

Figure 2 - conceptual Runtime Archtecture

OSGi Server Equinox

We have an OSGi server as the main process for the web service container. The OSGi interfaces are relatively precisely defined. Therefore, in principle it is possible to use any OSGi server besides Equinox like Kopflerfish or Spring. However, Equinox is the platform used within Eclipse and it has specific extensions. The direct usage of Equinox avoids adaption of these special extensions, which may be used by some special engineering plugins. The NeOn engineering plugins we used for our experiments did not use such extensions.

The OSGi server implementation itself is packaged in one of the basic OSGi runtime bundles.

All other components of the runtime archictecture are now OSGi bundles. Either pure OSGi bundles (and not plugins) or Eclipse plugins, which are also OSGi bundles.

Besides the pure OSGi server, the Equinox platform already contains some bundles for general system functionality like logging and web server functionality. Some of them are used in Eclipse also.

Servlet Container Jetty

For java-based web services a servlet container is necessary. In order to run it as an OSGi bundle we have to use a very lightweight servlet container which does not contain a lot of additional functionality requiring direct operating system access. The servlet container Jetty fulfilled these requirements.

Web Service Container Axis2

We have chosen to use the common web service container Axis2 [Tong2009]. Currently the whole Axis2 runtime is packaged as a single bundle. Many additional infrastructures like XML APIs and implementation are required by Axis2 and thus currently contained in Axis2 runtime.

All this functionality is currently packaged as one single large bundle. This avoids consolidation of commonly used technologies like XML, logging or protocol handling across all bundles of the server.

Eclipse basic bundles

The core NeOn Toolkit contains most of the default eclipse bundles. Engineering plugins can in principle use all of their exported non-GUI functionality. Thus these bundles are also necessary in the OSGi server. Example of those Eclipse functionalities are: database access, file I/O. But there are also other core infrastructure bundles contained like Internationalization (ICU), logging and unit testing.

NeOn Toolkit Core bundles

NeOn engineering plugins often need the non-GUI plugins of the core NeOn Toolkit. The OWL API bundle in particular is normally used in engineering plugins. Thus all the non-GUI bundles of the core NeOn Toolkit are included in the engineering server.

Engineering Plugin-specific bundles

Up to now we have described only infrastructure bundles, which are necessary for any engineering plugin. There are also, however, a number of bundles specific to the engineering plugin to be published.

2006–2010 © Copyright lies with the respective authors and their institutions.

1. Bundles of engineering plugin features: The engineering plugin is precisely an Eclipse feature, which consists of a set of plugins. All the bundles of this feature have to be deployed to the OSGi server. Note that those bundles can be directly deployed into the OSGi server without any change.
2. Bundles of dependant engineering plugin features: Additionally, the engineering plugin feature may depend directly or indirectly on other NeOn engineering plugin features. All their bundles have to be deployed too. A plugin has been developed to semi-automatically discover all indirectly dependant engineering plugin features. This allows for an easy deployment of all the necessary bundles.
3. Bundle of Web service adaptors for engineering plugins: For the interface of each to be published engineering plugin, a web service adaptor had to be specifically developed. Most of the required adaptors can be generated by the web service development tool. This bundle contains the mapping between the SOAP protocol of the web service and the interface method invocation of the java class.

Dirty Engineering bundles

Engineering bundles should not contain GUI functionality; they should only contain pure ontology functionality. However, as it is difficult to enforce this objective, several plugins have not strictly adhered to the rules regarding placing GUI and pure ontology functionality into different plugins.

If engineering plugins contain only code with GUI functionality, which is not invoked during the execution of ontology functions, a special modus can be enabled to run those plugins. In this “dirty engineering plugin” modus, all Eclipse and NeOn Toolkit plugins are deployed in the OSGi-based server. Thus it is possible to deploy dirty engineering plugins. However, if GUI functionality is invoked at runtime the function will be aborted.

5.2 Development Activities

In order to publish an existing engineering plugin as an ontology service, the following steps are necessary:

- Identify the interface of the engineering plugin
- Develop a web service adaptor for the interface of the engineering plugin. The necessary components have to be provided via one of the following approaches:
  - automatic generation at runtime via a generic mechanism in the web service container
  - automatic generation at design time, which can be modified
  - manual creation at design time

  The web service adaptor consists of the following components:
  - Web Service Description Language (WSDL) document: It is possible to automatically generate a web service interface in the form of a WSDL from a Java interface.
  - Web Service Adaptor Class: The class which contains the interface method. In simple cases the interface class of the engineering plugin can be used. Otherwise, an adaptor class has to be developed to map between the SOAP interface and the Java interface method. In the latter case a template can be automatically generated based on an existing WSDL.

- Identify all NeOn Toolkit engineering plugins that are directly or indirectly required by the to be published plugin
- Deploy all those engineering plugins, including the generated web service adaptor bundle, as bundles into the engineering service container

5.2.1 Description of the specific adaptor development

Building an Axis2 service bundle is a simple matter of implementing a class that calls the required functionality contained of the UI plugins. This functionality can be provided in a class, command or 3rd party plugin. Discovery and manipulation of projects is possible. The class should have one or more public methods that will be translated by axis to XSDs and soap actions in the WSDL. The arguments and result types of these methods are optional. One should be cautious with complex arguments/results, because these may produce overcomplicated XSDs. If one wishes to use attachments, the easiest way is to define a parameter/result of the type DataHandler or byte[]. Also, one should set the enableMTOM property in the axis2.conf file to true. Thus Axis will take care of encoding and wrapping the attachments for you.

5.3 Engineering Service Container

Based on the above discussion of the OSGi-based runtime architecture we describe here the chosen bundles for building the engineering service container.

Bundles for Equinox System

OSGI_platform contains the basic OSGi platform Equinox. The basic bundles contain the framework and the class to start it. A sample Equinox configuration contains a conf file that describes which bundles are pre-installed and which ones should be started. In addition, there is a bat file that runs the system by calling the main class in the main bundle. The conf file is passed as a parameter to the framework, which loads all other bundles and initializes the system. All Equinox OSGI bundles are also used in Eclipse, which means we can install Eclipse bundles provided that we respect their dependencies. Here, there are additional util and equinox bundles mentioned that are needed in later steps. These may vary depending on the used http and web service container.

OSGI Platform

- org.eclipse.core.runtime
- org.eclipse.osgi

Additional util bundles

- org.apache.log4j
- com.ibm.icu
- org.hamcrest.core
- org.junit4
- slf4j-api
- slf4j-log4j12

Additional eclipse bundles

- org.eclipse.equinox.cm
- org.eclipse.equinox.launcher.win32.win32.x86
- org.eclipse.equinox.launcher
- org.eclipse.equinox.p2.artifact.repository
- org.eclipse.equinox.p2.core
- org.eclipse.equinox.p2.engine
- org.eclipse.equinox.p2.jarprocessor

2006–2010 © Copyright lies with the respective authors and their institutions.

- org.eclipse.equinox.p2.metadata.repositor
- org.eclipse.equinox.p2.metadata
- org.eclipse.equinox.p2.repository
- org.eclipse.equinox.preferences
- org.eclipse.equinox.registry
- org.eclipse.equinox.security.win32.x86
- org.eclipse.equinox.security

Bundles for Jetty Servlet Container

Together with several important packages, Jetty forms the http layer in the OSGI platform. It is also a servlet container, which is useful for building an http UI for the installed plugins.

- org.eclipse.equinox.http
- org.eclipse.equinox.http.jetty
- org.eclipse.equinox.http.servlet
- org.mortbay.jetty.server
- org.mortbay.jetty.util
- org.apache.commons.logging
- javax.servlet.jsp
- javax.servlet

Bundles for Axis2 Web Service Container

Axis2 requires a lot of infrastructure software. It is packed as a bundle and requires an external folder structure. It should contain the axis2 config file, modules and services. The usual way of publishing services would be to place the .aar files in the services folder. However, this had to be modified in order to directly invoke bundles containing the web services. Thus there is an additional OSGi bundle wrapper of Axis2, which allows for building the web service as bundles instead of .aar files. The Axis2 bundle searches for a service descriptor (services.xml) in all plugins and identifies the web service bundles; it then publishes the service according to the descriptor. In most cases it simply needs a class with a public method, which is automatically translated into an XSD schema and a WSDL.

Bundles for basic Eclipse

Eclipse_basic_plugins contains basic non-GUI plugins required by NeOn Toolkit. Some of the Eclipse bundles contain GUI plugins, which are not needed by pure engineering plugins. However, there are dirty engineering bundles, which require these GUI Eclipse plugins during deployment although the GUI functionality itself will not be invoked during runtime.
For the deployment of dirty engineering plugin also, the GUI eclipse plugins that are required by the NTK are contained in the server. They are installed but not started as they are needed only as dependencies.

Eclipse UI plugins

- jcl-over-slf4j

- org.apache.ant
- org.eclipse.ui.intro.universal
- org.apache.commons.el
- org.apache.felix.fileinstall
- org.apache.jasper
- org.apache.lucene.analysis
- org.apache.lucene
- org.eclipse.ant.core
- org.eclipse.compare.core
- org.eclipse.compare
- org.eclipse.core.commands
- org.eclipse.core.contenttype
- org.eclipse.core.databinding.observable
- org.eclipse.core.databinding.property
- org.eclipse.core.databinding
- org.eclipse.core.expressions
- org.eclipse.core.filebuffers
- org.eclipse.core.filesystem.win32.x86
- org.eclipse.core.filesystem
- org.eclipse.core.jobs
- org.eclipse.core.net.win32.x86
- org.eclipse.core.net
- org.eclipse.core.resources.compatibility
- org.eclipse.core.resources.win32.x86
- org.eclipse.core.resources
- org.eclipse.core.runtime.compatibility.auth
- org.eclipse.core.runtime.compatibility.registry
- org.eclipse.core.runtime.compatibility
- org.eclipse.core.variables
- org.eclipse.ecf.filetransfer
- org.eclipse.ecf.identity
- org.eclipse.ecf.provider.filetransfer.ssl
- org.eclipse.ecf.provider.filetransfer
- org.eclipse.ecf.ssl
- org.eclipse.ecf
- org.eclipse.equinox.app
- org.eclipse.equinox.common
- org.eclipse.equinox.concurrent
- org.eclipse.equinox.http.jetty
- org.eclipse.equinox.http.servlet
- org.eclipse.help.base
- org.eclipse.help.ui
- org.eclipse.help
- org.eclipse.jface.databinding
- org.eclipse.jface.text
- org.eclipse.jface
- org.eclipse.ltk.core.refactoring
- org.eclipse.ltk.ui.refactoring
- org.eclipse.osgi.services
- org.eclipse.search
- org.eclipse.swt.win32.win32.x86

2006–2010 © Copyright lies with the respective authors and their institutions.

- org.eclipse.swt
- org.eclipse.team.core
- org.eclipse.team.ui
- org.eclipse.text
- org.eclipse.ui.cheatsheets
- org.eclipse.ui.console
- org.eclipse.ui.editors
- org.eclipse.ui.forms
- org.eclipse.ui.ide.application
- org.eclipse.ui.ide
- org.eclipse.ui.intro
- org.eclipse.ui.navigator.resources
- org.eclipse.ui.navigator
- org.eclipse.ui.views.properties.tabbed
- org.eclipse.ui.views
- org.eclipse.ui.win32
- org.eclipse.ui.workbench.compatibility
- org.eclipse.ui.workbench.texteditor
- org.eclipse.ui.workbench
- org.eclipse.ui
- org.eclipse.update.configurator
- org.eclipse.update.core.win32
- org.eclipse.update.core
- org.eclipse.update.scheduler
- org.eclipse.update.ui

Bundles for core Neon Toolkit

Neon_core__plugins contains the NTK core plugins required by non-UI engineering plugins; that is, mainly OWL2 API implementation:

- org.semanticweb.owl

The following NeOn Toolkit bundles also contain GUI functionality, which is not invoked. They are included as dirty engineering plugins:

- com.ontoprise.dependencies
- com.ontoprise.ontostudio.dependencies
- com.ontoprise.ontostudio.owl.gui
- com.ontoprise.ontostudio.owl.model
- org.neonToolkit.core
- org.neonToolkit.gui
- org.neonToolkit.io
- org.neonToolkit.plugin
- org.neonToolkit.refactor
- org.neonToolkit.search
- org.neonToolkit.swt

All these bundles contain a config file that incrementally adds the names of the bundles that have been installed in the particular step. The OSGi mechanism guarantees that only bundles which are invoked will be started; thus they do not occupy memory and do not impede performance. The

service bundles should be started so that Axis2 can publish their web services. They will load classes from their dependencies and only the necessary UI plugins will be activated.

5.4 Example Service: SAIQL

We use the SAIQL engineering plugin as an example to publish it as a web service. The main functionality of this plugin is to evaluate a SAIQL query on a given ontology. The result is, again, an ontology. This is a typical ontology service functionality as it is quite coarse grain and self-contained.

5.4.1 Publish SAIQL engineering plugin as web service

For the construction of a SAIQL ontology web service, we need to install the SAIQL plugin in the OSGI environment and publish some of its functionality as a web service. For this purpose we need the bundles contained in the following two packages:

- Saiql_Hermit has two bundles of the hermit reasoner and the saiql plugin. They will be used by the SAIQL web service in the next step. In this example, they placed them in the bundles folder. As these contain UI elements and depend on the Eclipse UI plugins, the whole Eclipse environment including the GUI plugins must be installed as explained above.
- Saiql_Service contains one bundle that defines a service published by Axis2. It contains an adapter class and a service.xml file.

When these bundles are installed, the Axis2 web service environment detects a new bundle containing “service.xml” and publishes its class as a web service. When it is invoked, the SOAP message is parsed by a standard message receiver and the respective method in the adapter class is called. It initializes the functionality of the Saiql plugin, passes the arguments and returns the resulting ontology.

SAIQL Example

The engineering plugin server has been packaged together with the SAIQL service example in one deployment package. It is configured to listen to port 8080. After starting it and waiting for all bundles to load, you can test it by writing “ss” in the console to see all bundles and to check if the *org.neonToolkit.saiql.osgi_ws* bundle is started. Secondly, you can open the address: http://localhost:8080/asix2/services/SaiqlServer?wsdl to load the WSDL of the saiql service. A sample SOAP request would look like this. 311305672103 is the id of the attached file and http://www.NewOnto1.org/ontology1 is ontology ID.

```
<soapenv:Envelope              xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ser="http://service.saiql.neonToolkit.org">
   <soapenv:Header/>
   <soapenv:Body>
      <ser:evaluateQuery>
         <ser:ontology>cid:311305672103</ser:ontology>

         <ser:query><![CDATA[
      CONSTRUCT SubClassOf( ?X ?Z ); Individual( ?i Type( ?Z ))
      FROM <http://www.NewOnto1.org/ontology1>[<http://www.NewOnto1.org/ontology1>]
```

2006–2010 © Copyright lies with the respective authors and their institutions.

```
        LET IndividualName ?i; ClassName ?X; ClassDescription ?Z
        WHERE SubClassOf( ?X <http://www.NewOnto1.org/ontology1#asdasdasd>)
        AND Individual( ?i Type( ?Z ))  AND SubClassOf( ?X ?Z )

        ]]></ser:query>

           <ser:resultIRI>http://test.org</ser:resultIRI>

       </ser:evaluateQuery>
   </soapenv:Body>
</soapenv:Envelope>
```

Development of SAIQL service Adaptor

As the functionality of SAIQL is not focused in one method, we need to build an adapter class. It will not only initialize the environment, translate the arguments and execute the query, but will present a more Axis2-friendly interface. Since we need an ontology and a query to use Saiql, the method in this adapter class will receive a DataHandler (or byte array) and a String as arguments. Axis2 will recognize this DataHandler automatically as a SOAP attachment. When the method is invoked, we receive the attached ontology as an input stream, which we need to write in a temporary file for SAIQL to read.

The SAIQL engineering plugin is currently not completely free of references to GUI plugins of Eclipse. As Saiql is built for the UI environment of Eclipse, it searches for a selected project and accesses an OntologyManager via the selected project. Thus, we need to create a new project or find a preselected one. This could be done through the readily available commands or through the ProjectManager class. Although the UI plugins do not work in the server side OSGI, the workspace bundles are started and allow handling resources and projects.

When we have the name of an available project, we pass it to the SAIQL query class, initialize it with the attached ontology and run the provided query. The result is another ontology, which we can return in a similar fashion, write it to a temporary file and wrap it in a DataHandler. It will be translated by Axis2 into an attachment and returned to the caller.

Runtime Bundles for SAIQL engineering service

We need the following SAIQL specific bundles:

- Dependant engineering plugin: org.neonToolkit.hermitReasoner
- Existing engineering plugin: org.neonToolkit.saiql
- Web service adaptor: org.neonToolkit.saiql.osgi_ws

For the SAIQL service example, we have used those versions of the plugins that were contained in the NTK 2.3 build 164, Hermit 1.3.0 and SAIQL 1.0.0.

The workspace projects of both service bundles can be found in the workspace folder. In "_examples", you can find sample ontologies and soap requests as well as ready-made projects for soapUI.

The currently available plugins are described in deliverable D6.10.3 [NeOn D6.10.3].

The resulting SAIQL web service in the engineering service container has been successfully deployed at the University Koblenz Site. It can be accessed as web service via the following URL:

http://saiql.west.uni-koblenz.de:8080/axis2/services/SaiqlService?wsdl

2006–2010 © Copyright lies with the respective authors and their institutions.

6 Conclusion

We have developed mechanisms and infrastructures to realize the interoperability between NeOn Toolkit engineering plugins and ontology services in both directions. Especially for ambitious direction to publish finished engineering plugins as ontology web services we have developed an elegant solution based on the OSGi framework. It consists of an engineering service container and a description of a development process.

The process requires manual development steps like analyzing the dependency graph of the used engineering plugins and generating a template for the service adaptor plugin. These indicate the need for development tool support for the NeOn plugin developer. This would be the first step towards a new Toolkit not for the ontology engineer but for the NeOn plugin developer.

Another interesting development is Server-side Eclipse, which shows potential to even avoid the currently duplicated deployment of the engineering plugins in the NeOn Toolkit and in the OSGi-based server.

7 References

NeOn D6.9.1 — Walter Waterfeld, Michael Erdmann, Thomas Schweitzer, Peter Haase: Deliverable D6.9.1: Specification of NeOn reference architecture and NeOn APIs V2. NeOn Project Deliverable, 2008.

NeOn D6.6.3 — NeOn Toolkit documentation v3. NeOn Project Deliverable 2009 (*forthcoming*)

NeOn D6.7.3 — Michael Erdmann: D6.7.3 Update of the core NeOn Toolkit. NeOn Project Deliverable 2010 (forthcoming)

NeOn D6.10.3 — Updated NeOn Toolkit plugins v3. NeOn Project Deliverable 2009 (*forthcoming*)

OSGi2009 — The OSGi Alliance, OSGi Service Platform V4.0, Core Specification, 2009

Sharma2001 — Rahul Sharma and Beth Stearns and Tony Ng: J2EE ™ Connector Architecture and Enterprise Application Integration. The Java Series., 2002, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA

Tong2009 — K. Tong, Developing Web Services with Apache CXF and Axis2, TipTecDevelopment, 2005, USA

2006–2010 © Copyright lies with the respective authors and their institutions.