NeOn: Lifecycle Support for Networked Ontologies

Integrated Project (IST-2005-027595)

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

D2.3.4 C-ODO plugin v1.0

Deliverable Co-ordinator: **Alessandro Adamou**

Deliverable Co-ordinating Institution: **CNR**

Other Authors: **Aldo Gangemi (CNR), Valentina Presutti (CNR)**

This deliverable presents the Kali-ma NeOn Toolkit plugin, which exploits the versatility of the C-ODO Light model to assist ontology engineers and project managers in locating, selecting and accessing other plugins through a unified, shared interaction mode. Kali-ma offers reasoning methods for classifying and categorizing ontology design tools with a variety of criteria, including collaborative aspects of ontology engineering and the NeOn methodology. Furthermore, it provides means for storing selections of tools and associating them directly to development projects, so that they can be shared and ported across systems involved in common engineering tasks. In order to boost Kali-ma support for third-party plugins, we are also offering an online service for the semi-automatic generation of C-ODO Light-based plugin descriptions.

Document Identifier:	NEON/2009/D2.3.4/v1.1	Date due:	September 30th, 2009
Class Deliverable:	NEON EU-IST-2005-027595	Submission date:	October 31st, 2009
Project start date	March 1, 2006	Version:	v1.1
Project duration:	4 years	State:	Final
		Distribution:	Public

NeOn Consortium

This document is 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: Martin Dzbor, Enrico Motta E-mail address: {m.dzbor, 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: Peter Haase E-mail address: pha@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.ump.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 SL–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 – 665 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 Kolenz-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. Marino 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 00100 Rome Italy Contact person: Marta Iglesias E-mail address: marta.iglesias@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 the writing of this document or its parts:

- CNR
- FAO
- JSI
- ONTO
- UKARL
- UKO-LD
- UPM
- USFD

Change Log

Version	Date	Amended by	Changes
0.1	07-11-2009	Alessandro Adamou	TOC and early layout
0.2	14-11-2009	Alessandro Adamou	TOC update and chapter 2 additions
0.3	18-11-2009	Alessandro Adamou	added Aldo and Valentina contributions, updated chapter 2
0.4	21-11-2009	Alessandro Adamou	added chapter 4, revised section 2.1, sent draft to reviewer
0.5	23-11-2009	Alessandro Adamou	added chapter 3, TOC update
0.6	24-11-2009	Alessandro Adamou	finalized chapter 3, swapped chapters 3 and 4, added chapter 6, sent second draft to reviewer
0.7	25-11-2009	Alessandro Adamou	updated chapter 1
1.0	27-11-2009	Alessandro Adamou	final draft, sent to reviewer
1.1	04-12-2009	Alessandro Adamou	amended per reviewer feedback
1.2	11-12-2009	Valentina Presutti	introduction rewritten, some editing and revision.

Executive Summary

This deliverable presents the Kali-ma NeOn Toolkit plugin, which exploits the versatility of the C-ODO Light model to assist ontology engineers and project managers in locating, selecting and accessing other plugins through a unified, shared interaction mode. Kali-ma offers reasoning methods for classifying and categorizing ontology design tools with a variety of criteria, including collaborative aspects of ontology engineering and the NeOn methodology. Furthermore, it provides means for storing selections of tools and associating them directly to development projects, so that they can be shared and ported across systems involved in common engineering tasks. In order to boost Kali-ma support for third-party plugins, we are also offering an online service for the semi-automatic generation of C-ODO Light-based plugin descriptions.

Portions of this work, namely Chapter 2, include WP8-related case study contributions based on material provided by ATOS.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The NeOn Toolkit[1] (NTK) is an integrated software application based on the Eclipse platform[2]. As such, it can be easily extended in terms of functionalities, by plugging new components into the system. During the NeOn project life, many plugins have been developed, and many will be developed in the future by the NeOn community of developers. These plugins provide NTK with a wide range of functionalities e.g. documentation, visualization, reengineering, localization, pattern-based design, reasoning, planning, etc., thus supporting ontology engineers through the whole development process of networked ontologies. However, each plugin supports a certain task, and has its own access mode and interface, making difficult and time consuming, for NTK users, the tasks of finding, selecting, and using them: the higher the number of plugins, the higher the degree of software support for ontology life cycle, the higher the complexity of usage of the integrated tool, from the user's point of view.
On one hand, if NTK users want to be aware of all available NTK functionalities, they have to browse the NTK wiki, read the description of the plugins, and install the ones they are interested in. On the other hand, if users have a set of tasks to carry on, and they are looking for software support within NTK for those tasks, there is no way to query NTK about the means it provides for supporting those tasks. Furthermore, if users want to save a list of plugins as their working profile for a certain project, this is not possible. They will have to remember and launch all plugins they need, every time they start a new working session. For example, consider a user that needs support for planning the activities for the ontology project, and for reengineering a legacy database to an ontology. How can (s)he know that NTK provides e.g. gOntt and ODEMapster, that support specifically those tasks? How can (s)he know the way to access them from NTK interface? How can (s)he make NTK remember for that project, that those plugins should be open also next time?
In this deliverable, we present **Kali-ma**, a NTK plugin based on *codolight* that aims at addressing this use case. Kali-ma is a semantic-based tool that allows NTK users to navigate the space of available plugins by classifying them in terms of the tasks they support. Users can also customize the criteria of classification of plugins, Kali-ma will show them available plugins that satisfy those criteria. For example, if a user wants all plugins to be classified based on the NeOn methodology activity they support, Kali-ma will show all NeOn methodology activities, and each of them will be associated with the plugins that provide support for that activity. The classification is dynamic, as it is based on the semantic description of plugins, based on codolight. Kali-ma provides easy access to NTK plugin; it hides the diversity of interaction paths that are supported by plugins, and offers a unique access point for each of them, which is resolved to the appropriate interaction path once the user has communicated her/his will to open that tool. Furthermore, Kali-ma gives the user the possibility to save working profiles and to associate them with specific ontology projects.

The present work is structured as follows. The remainder of this chapter illustrates the rationale behind our proposal for the Kali-ma plugin, based both on previous usability studies conducted in NeOn WP4 and an analysis of the state of the art for semantic user interfaces. Chapter 2 discusses the tool itself from a

[1] http://www.neon-toolkit.org
[2] http://www.eclipse.org

functional standpoint, with examples based on the WP8 case study on Semantic Nomenclature ontologies. Chapter 3 concentrates on support for ontologies describing design tools such as NeOn Toolkit plugins, with particular focus on gathering such ontologies and generating them in a semi-automatic way. Chapter 4 describes the architectural components of the Kali-ma plugin, in both concept and implementation, including the ontological subsystem that serves as a basis for the whole infrastructure. Chapter 5 illustrates the strategy and outcome of a preliminary evaluation session on an alpha version of the tool, as well as the action taken upon it. Finally, Chapter 6 concludes with a synthesis on the updates and improvements currently underway.

1.1 Motivation

The raison d'être of a tool for supporting semantic-driven user interaction based C-ODO Light (or simply *codolight*) takes up from the usability studies carried out by Dzbor et al. within the scope of WP4, between 2006 (cf. [DMGP$^+$06] and [DMB$^+$06]) and 2008 (cf. [DBMGP08]). In the course of this work, observational usability studies were conducted in order to assess effectiveness, efficiency and user satisfaction of existing ontology editing platforms i.e. *Swoop*, *TopBraid*, *Protégé*3 and *Protégé*4, and base NeOn Toolkit interface design thereupon. The resulting qualitative analysis of the findings has led to the conclusion that these tools lacked significant support for hastening repeated tasks and allowing users to customize their interface in order to filter information that either is unwanted or users are not knowledgeable enough to handle. In other words, these user interfaces proved to be monotonic, in that plugins only add new interface elements, but users may not adequately filter them based on their needs. An additional issue emerged from user hardships in acquiring awareness of some available functionalities that remained unnoticed or even ignored.

Although these findings eventually materialized into lessons learnt for designing user interaction with the NeOn Toolkit, it is also true that its versatile extensibility does come at a price. It is not imaginable for us to keep every software contribution and every running instance of the platform under control, nor could we forbear deploying it as an open, extensible environment from the start. Plugin providers are left at complete liberty to contribute to the interaction capabilities of the NeOn Toolkit in whatever way they see fit, with respect to the data to be managed, and that complies with the Eclipse specifications. More importantly, they can, and most times will have to, do so regardless of how other parties will interpret the relationships between user interface and data when providing their own contributions.

Being a relatively younger discipline with respect to software engineering or business process management, ontology engineering seems to suffer particularly from these interaction issues. Two reasons that support this claim are both founded on the lack of a common understanding on which data can be considered to be significant on a generally domain-independent practice as is ontology engineering. The first reason is that focusing on the semantics of data for a field that has to cope with their extreme heterogeneity brings in some hardly surmountable hurdles. Most Semantic Web specialists have little affinity with the discipline of user interaction, therefore they often end up building their own data-driven ad-hoc interfaces which are then hard or impossible to port to other (sub-)domains. This happens because Semantic Web developers often feel that most common widget toolkits assume a data model that is incompatible with the ones describing the heterogeneous data in their possession. The second reason is the lack of an agreement over the most relevant criteria by which the pool of available software is measured. Plugin-based open platforms which support such lifecycles tend to suffer from these setbacks even more, and not even the NeOn Toolkit is exempt from lending itself to such risks. One clear symptom will be detailed in the section describing Kali-ma's tool organization functionality (cf. Section 2.4.1), and deals with the fact that each plugin provider that contributes to the NeOn Toolkit interface and set of functionalities tends to project her own mental model of the software platform as a whole. Consequently, new interface objects are presented into categories that, once aggregated, end up lacking of significance as they incorporate classes based on functionalities along with logical languages, specific tools and interaction methods altogether. Plugin developers may feel further constrained to adopt criteria that ultimately result incoherent, due to the fact that the Eclipse RCP enforces a user interface-related criterion above all others. That is, in order to find a tool that supports certain tasks, the user first needs to know how the tool is presented at a user interface level and, upon figuring that out, traversing the list of such interface items to locate the one of interest. While this surely makes sense in the context of interaction, it may ultimately disrupt the perception of the platform's functional capabilities.

What we have outlined so far are not faults of the NeOn Toolkit per se, and the advantages of the assumed datamodel for the Eclipse RCP are many, not least a model for project-centered resource distribution that bears great affinity with our methodologies for ontology lifecycle management. However, we felt it sensible to provide a means to contain the potentially uncontrolled proliferation of features by an interactive display of available plugins organized by exploiting the semantics of their operational capabilities. Through the rest of the deliverable, we will refer to this functionality as *organization*, or *classification*, or *categorization*, or *pigeonholing* of the tool space.

1.2 Relation to state of the art

The use of semantic guidance to user interaction is a recurring challenge in several research fields of Intelligent User Interfaces (IUI), including the one collectively known as *S*emantic Desktops [BCT07][SBD05]. This field is mainly interpreted as covering the practice of annotating data in a local environment with machine-readable metadata. Leading research efforts in this field have concentrated on both domain-centered user interfaces and common vocabularies for rendering annotated data as shareable objects which would make sense in different environments and applications.

One groundbreaking albeit premature effort in this sense was the *H*aystack project [QHK03], aimed at providing a platform for Semantic Web applications. One significant contribution of this project that still holds to this day was the Haystack client, an Eclipse-based RDF-based personal information manager. Haystack used a declarative approach that employed ontologies through which it is possible to define layout constraints for widgets. Pietriga et al. have argued in [PBKL06], that declarative approaches tend to fall short of portability due to their tight bindings to specific representation paradigms, hence they could be of little use in contexts like the strongly interface-centered model adopted by the NeOn Toolkit.

Since Haystack, several studies have gone in the direction of customizing the interaction experience by filtering the annotated data that are intended to be manipulated by the user. An example of such an approach is the IRIS project, along with its open-source counterpart *O*penIRIS [CPG05], which provides integrated dashboard-like views on data concerning office-related activities. However, it wasn't until the *Fresnel* vocabulary surfaced, that it became feasible to reach common agreements on which types of data retrieved from the Semantic Web should be presented and how [PBKL06]. Fresnel is an RDF presentation vocabulary that is independent on applications and output formats, and allows to specify lenses on RDF properties to show in a given order, and formats indicating how these should be presented. The *Longwell* faceted RDF browser, born within the SIMILE project, is an implementation of this proposal that uses a Fresnel-based rendering engine.

Taking advantage of the intuitions as well as the shortcomings of the works mentioned above (low scalability and steep requirements of Haystack and lack of support for collaboration from most tools), the *NEPOMUK* project delivered an open-source specification of a framework for Social Semantic Desktops [GHM$^+$07]. One significant aspect of this cross-platform and language-independent framework, with respect to the work presented herein, is the presentation layer. This infrastructural layer was designed to provide user interfaces on demand for each service on the NEPOMUK desktop. Additionally, plugins and add-ons are provided seamless bridging of third-party applications and the NEPOMUK desktop. Demonstrators for KDE and Eclipse RCP are reference implementations of parts of the NEPOMUK specification, as is the *G*nowsis Semantic Desktop presented in [SS04].

The studies so far recalled focus mainly on the semantics of data, hence most usability studies and user interface implementations are targeted at annotating, exploring and leveraging metadata-enriched data sets. The approach adopted for Kali-ma takes up from this intuition and moves on to exploiting the semantics of process models, which in turn embrace the semantics of data. In this context, we refer to process models from an engineering perspective, where it denotes a set of activities involving transformations of input resources into products with specific constraints.

While the NeOn task presented in this deliverable is obviously tailored around the process models of ontology lifecycle management, we will give simple insights as to how our approach can be regarded as a proof-of-concept, and the same paradigm can be applied to software engineering, industrial processes and the like. This theory hints at another binding between the Kali-ma approach and existing work i.e. the one with semantic services. On one hand, as detailed in Section 4.4, our approach has an analogy with several state-of-the art standards for the semantic enrichment of services, which rely on the annotation of input and output in a service description [SGR08] as much as we are doing when defining rules for collaborative aspects of ontology design. On the other hand, Kali-ma itself may be seen as a simple semantic stack for services (i.e. functionalities realized by plugins) available on a local platform (i.e. a NTK installation), in accordance with the Software as a Service (SaaS) paradigm [BHL08].

In the context of project lifecycle management guided by semantics, some significant research work has been carried out, which has proven strongly inspiring to our solution design efforts for the Kali-ma plugin. As for the software engineering field, the work presented by Silva Parreiras et al. in [SPSP$^+$08] also proposes the use of ontologies for model-driven software development. Another approach can be found in [SNTM08] that describes *Collaborative Protégé*3, a Protégé3-based tool supporting collaborative ontology design. It uses an ontology that describes collaborative workflows for storing and querying metadata information during collaborative ontology design processes.

Chapter 2

The Kali-ma plugin

This chapter provides a conceptual and user-level overview of our C-ODO Light-based tangible contribution to NeOn technologies. The rationale and features behind the Kali-ma plugin, with particular regard towards what was anticipated in the preceding deliverable [PMP+09] are herein listed and explained in further detail. While the actual user documentation and guides are deferred to a dedicated deliverable for NeOn Toolkit plugin documentation, at the end of this chapter the reader will have learned what functionalities are available in the first release of the Kali-ma plugin and have an insight, corroborated with actual user interface screenshots, of the interaction approach to these functionalities. To further clarify the benefits of the approach provided, a familiar run-through scenario based on the WP8 Semantic Nomenclature case study will provide examples of each functionality in a running instance of the plugin.

2.1 Philosophy

Pigeonholing the entire application support for a production domain that supports engineering methods and procedures is a broad, open problem to which this NeOn task can be seen as offering a domain-specific proof of concept. Every domain where some kind of project lifecycle management occurs can benefit from rich structured semantics describing the tasks, processes, actors and resources involved in it, and the relationships with each other. However, when these semantics are employed for offering a certain view on interaction between designers and software platforms for lifecycles management, choices must be made as to what particular vertical section of the domain should be presented to the users, lest they feel unguided and lost when interacting with the environment. While most previous studies have chosen to focus on the semantics of data (cf. Section 1.2), we have opted in Kali-ma for exploiting the semantics of activities for authoring and manipulating these data.

With the Kali-ma plugin, whose first version is focused on customizing user interaction with the NeOn Toolkit, users can benefit from an interaction-independent view on the system they have before them. The approach taken focuses first on the plugins, then on functional categories that make sense in the world of networked ontology engineering. Interaction-related aspects are shelved to the level of plugins themselves, which can be seen as the atomic component of the Kali-ma approach, as opposed to UI components that drive the flow of interaction with Eclipse-based tools. In an effort to resolve this apparent conceptual contradiction, our approach maps selected atomic entities (plugins) to independent user interface objects, called *widgets*, which operate in a visual space that only has feeble bindings with the originating standard Eclipse interface. Our claim is that a semantic approach can definitely ease user interaction with the functionalities that handle projects within the domain.

Achieving this goal is made easy thanks to the nature of the bottom-level layer of the Kali-ma architecture. A core of networked ontologies, covering different layers of the interaction with the toolkit, is part and parcel of the logic behind our approach, and as a matter of fact, it is treated in the same manner as dynamically linked software libraries. If we were to port the conceptual approach to a completely different domain, these semantic components could be replaced with an updated version or even with a set of ontologies that serve a completely different purpose, provided they are aligned with the basic specifications of an ontological component supported by Kali-ma.

For instance, software engineers may want to use the Kali-ma interface to manage and interact with their Eclipse IDE and its plugins to perform tasks like modeling UML diagrams, performing unit tests or actually writing code. At the time of writing, this can already be made possible with little intervention in the Java code of the plugin and a complete replacement of its ontologies. What would be needed is 1) a model that describes the discipline of software engineering, a fine example being *Seontology* [DCW08]; 2) descriptions of known Eclipse plugins for software engineering, based on this model; 3) a set of rules that provide a taxonomy of defined classes for software engineering aspects.

Conversely, this approach still makes sense when remaining in the context of ontology engineering and migrating to another development environment. The exception is that what is retained this time is the ontological component, or part of it. for example, several entities in the codolight metamodel are aligned with the Protégé workflow ontology [SNTM08], which is used in a plugin for Protégé3 called Collaborative Protégé. These entities include interface object, workflow, design functionality and task, knowledge resource and other entities that are involved in the formalization of design aspects, which hints that the classification rules proposed in this release can be easily backported to work in the Protégé environment. A plugin for this platform can then be developed based on the same grounds as Kali-ma and, provided that ontological descriptions of Protégé plugins exist, a similar interaction alternative can be offered. On these grounds, any other plugin-based ontology design tools, such as TopBraid and Protégé4, can benefit from the Kali-ma approach.

2.2 Relations with the Semantic Nomenclature case study

The remainder of this chapter will be entirely dedicated to displaying the main features of this first version of the Kali-ma plugin, with particular focus on its proposed methods for seamlessly coordinating interaction across plugins. In order to maximize the comprehension of the functionalities that will be shown, and aid the reader in realizing the full potential of this tool for real-world use cases, we will use the pharmaceutical case study conducted in WP8 (cf. deliverables D8.3.1 [GPBC+07] and D8.3.2 [CL08]) as a testbed for our demonstrations.

More specifically, it will be shown how Kali-ma could be employed to assist project managers and engineers alike, through the entire phase of Semantic Nomenclature implementation. This phase aimed at constructing a Nomenclature Networked Ontology by reengineering existing resources from the mainstream classification models in the pharmaceutical sector (such as the Snomed CT ontology, the MeSH thesaurus, the DIGITALIS and BOTPlus databases) to a reference model for all the knowledge concerning pharmaceutical products.

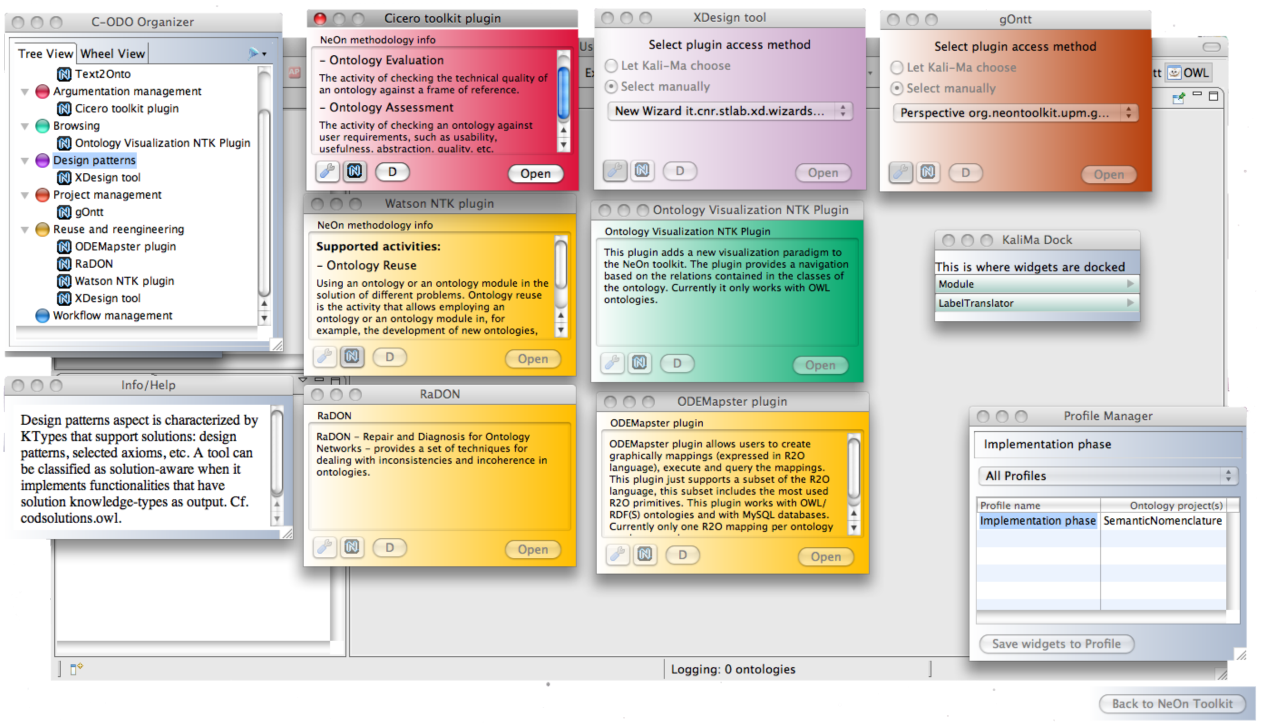
As the Pharmaceutical case study is centered on the application of the NeOn methodology (cf. [SFdCB+08, SFBd+09]), we will assume this to be the driving force of the entire workflow of the implementation phase and classify the used plugins in terms of the supported activities. The implementation phase involves a variety of activities including ontology re-engineering, population, alignment, argumentation, consistency checking, documentation and the scheduling of all said activities. Most of the tasks associated to these activities were indeed performed through the use of NeOn technologies, most of them being NTK plugins. However, the examples provided will occasionally highlight possible relationships between activities in the NeOn methodology and our proposal for a collaborative design aspect-based classification.

The examples related to the WP8 case study will be adequately highlighted through the rest of this chapter, by including them within bordered text boxes bearing the title “Semantic Nomenclature example”.

2.3 The Kali-ma dashboard

As with the vast majority of plugins for the Eclipse Rich Client Platform (RCP), many NeOn Toolkit plugins integrate with the platform by providing additions to the set of interface components that make up the so-called Eclipse Workbench, e.g. views, editors, perspectives, wizards and cheatsheets. However, Kali-ma is not a plugin designed to provide functionalities that allow to perform certain tasks in engineering networked ontologies; its aim is to facilitate the interaction flow across such functionalities provided by other plugins and to provide an alternative central thread to support the advancement of an ontology engineering process, and in doing so, it leverages high-level semantics of these plugins. To this end, the Kali-ma interface components are not based on the standard Eclipse Workbench, but they directly exploit the capabilities of typical widget toolkits offered by windowed operating systems. These components, called *widgets*, are then integrated as a single aggregate Graphical User Interface (GUI) called the *Kali-ma Dashboard*. This is not provided as a standard Eclipse composite panel (Perspective) or a single component of one (View), but is an invisible container for operating system windows on top of the NeOn Toolkit Workbench.

Every widget in the Kali-ma Dashboard identifies a functionality, or set of functionalities that mirror those provided by the NeOn Toolkit core and its plugins. These can be grouped in two major categories: **native widgets** denote built-in interaction-oriented functionalities offered by the Kali-ma plugin itself, and are always available regardless of what tools are installed on the platform; **plugin widgets** are representatives for plugins that are installed on the system and they offer quick access to the functionalities available due to these plugins being installed. Widgets belonging to this latter category are available upon user request when the corresponding plugin is installed on the NeOn Toolkit platform, no matter what the canonical interaction paths to access them.


C-ODO Organizer

Tree View | Wheel View

- Text2Onto
- Argumentation management
  - Cicero toolkit plugin
- Browsing
  - Ontology Visualization NTK Plugin
- Design patterns
  - XDesign tool
- Project management
  - gOntt
- Reuse and reengineering
  - ODEMapster plugin
  - RaDON
  - Watson NTK plugin
  - XDesign tool
- Workflow management

Info/Help

Design patterns aspect is characterized by KTypes that support solutions: design patterns, selected axioms, etc. A tool can be classified as solution-aware when it implements functionalities that have solution knowledge-types as output. Cf. codsolutions.owl.

Cicero toolkit plugin

NeOn methodology info

– Ontology Evaluation

The activity of checking the technical quality of an ontology against a frame of reference.

– Ontology Assessment

The activity of checking an ontology against user requirements, such as usability, usefulness, abstraction, quality, etc.

Watson NTK plugin

NeOn methodology info

Supported activities:

– Ontology Reuse

Using an ontology or an ontology module in the solution of different problems. Ontology reuse is the activity that allows employing an ontology or an ontology module in, for example, the development of new ontologies,

RaDON

RaDON

RaDON – Repair and Diagnosis for Ontology Networks – provides a set of techniques for dealing with inconsistencies and incoherence in ontologies.

XDesign tool

Select plugin access method

○ Let Kali-Ma choose

◉ Select manually

New Wizard it.cnr.stlab.xd.wizards...

Ontology Visualization NTK Plugin

Ontology Visualization NTK Plugin

This plugin adds a new visualization paradigm to the NeOn toolkit. The plugin provides a navigation based on the relations contained in the classes of the ontology. Currently it only works with OWL ontologies.

ODEMapster plugin

ODEMapster plugin

ODEMapster plugin allows users to create graphically mappings (expressed in R2O language), execute and query the mappings. This plugin just supports a subset of the R2O language, this subset includes the most used R2O primitives. This plugin works with OWL/RDF(S) ontologies and with MySQL databases. Currently only one R2O mapping per ontology

gOntt

Select plugin access method

○ Let Kali-Ma choose

◉ Select manually

Perspective org.neontoolkit.upm.g...

KaliMa Dock

This is where widgets are docked

Module

LabelTranslator

Profile Manager

Implementation phase

All Profiles

Profile name	Ontology project(s)
Implementation phase	SemanticNomenclature

Logging: 0 ontologies

Figure 2.1: The Kali-ma dashboard of widgets. Sorted by column, top to bottom then left to right: the *codo organizer*; the *helper widget*; widgets representing the following plugins: Cicero, Watson, RaDON, XDesign Tools, Ontology Visualization, ODEMapster and gOntt; the *dock* widget with placeholders for the Module and Label Translator plugin widgets; the *profile manager*; the *switch* widget for returning to the NeOn Toolkit workbench.

A running example of an open Kali-ma dashboard is displayed in Figure 2.1. The widget on the top-left corner is the *C-ODO organizer*, which exposes all available plugins classified according to a preferred criterion, and allows the user to browse them and select the ones of interest. Below is the *helper* widget, which provides realtime user guidance across the dashboard, by providing information on focused items, such as plugins, classification criteria or native widgets. The seven windows with varied background gradients are plugin widgets, which represent seven plugins installed on the NTK platform and selected by the user through the C-ODO organizer. Each widget provides panels for displaying a description of the tool, the NeOn methodology activities that it supports and the available interaction paths to launch it. As will be more thoroughly explained, the background color choice is not due to sheer aesthetics, but is bound to the categories each plugin falls under. The rightmost column displays the *dock*, which holds placeholders for two more plugin widgets, and the *profile manager*, which allows the user to manage customized configurations of the dashboard, called profiles. Profile operations include saving a certain set of widgets, binding it to a certain project and opening it. Finally, it is possible to switch between the dashboard view and the standard NTK interface view in as little as one click, through the *switch* widget in the bottom-right corner. The inverse operation is provided by clicking once either the Kali-ma icon in the NTK toolbar or the Launch Dashboard menu item from the Kali-ma menu, just as if launching the dashboard for the first time in the session.

Being the parent window to the whole dashboard and its widgets, the NeOn Toolkit workbench is displayed in the background. However, this behavior can be customized through the Kali-ma entry in the NeOn Toolkit preference panel: the user may opt for the Kali-ma dashboard to be independent of the main NeOn Toolkit window, with the sole exception of the close operation, which is propagated to the Kali-ma dashboard and, in fact, causes the toolkit itself to quit. When the Kali-ma window is set to behave independently, the plugin can be set to automatically hide or iconify the main window for as long as the dashboard is open.

The sections that follow will give detailed descriptions of the functionalities provided by each native widget and those that can be made available through plugin widgets. The main points of this interface are: (i) to present users with an overview of all available plugins within their running instance of NTK, together with a description of them, and (ii) allow users to access such plugins without knowing or caring about which is the interaction path to open a certain plugin within the NTK. Kali-ma takes the burden of hiding such interaction path and driving the user straight towards the expected interactive mode for that specific plugin.

2.4 Built-in functionalities

2.4.1 Plugin organization

As previously outlined, the first release of the Kali-ma plugin focuses on enhancing interaction between the user and the space of ontology engineering tools, by overcoming the hurdles that arise from the adoption of a fixed set of modalities that come with a specific integrated platform as is the Eclipse RCP. One of these hurdles is represented by the lack of an aggregated view of available functionalities with harmonized conceptual organization and minimal mental load. Kali-ma aims at bridging this "semantic gap" through a framework for providing such an organized view by means of customizable, design-centered criteria. This framework is synthesized as the *C-ODO organizer*.

The main goal of this widget is to present end users with an overview of plugins that are available in their running instance of the NTK and help them select the one(s) needed for performing certain tasks. The relationship by which plugins are organized may or may not strictly depend on the C-ODO Light framework [GP09]; however, some form of binding with it, given or implied, is constantly present, the least being a definition for the class of ontology design tools. The name of this widget is to reflect such a binding with the design domain.

It is reasonable to assume end users to be ontology engineers with a variable degree of experience and knowledgeability with regard to some ontology design methodology. However, we cannot imply that they possess any prior knowledge of specific plugins and what tasks they offer support for, let alone expect them to be confident with the interaction modes supported by these plugins. In order to cope with this assumption, we provide a choice of three relations in terms of which plugins are classified and shown in the C-ODO organizer tree view. These relations will be outlined in further detail in the next sections.

For a better understanding of the mechanics and the rationale behind Kali-ma, figure 2.2 will be used to draw a comparison between one classification proposed by Kali-ma and two built-in methods of presenting NeOn Toolkit plugins for access. The C-ODO organizer is shown on the top-left corner, while the other two windows are popup dialogs that allow to open specific composite panels, or Perspectives, and single panels, or Views, all of which are provided by either the NTK core or its plugins. Note that the three windows shown in Figure

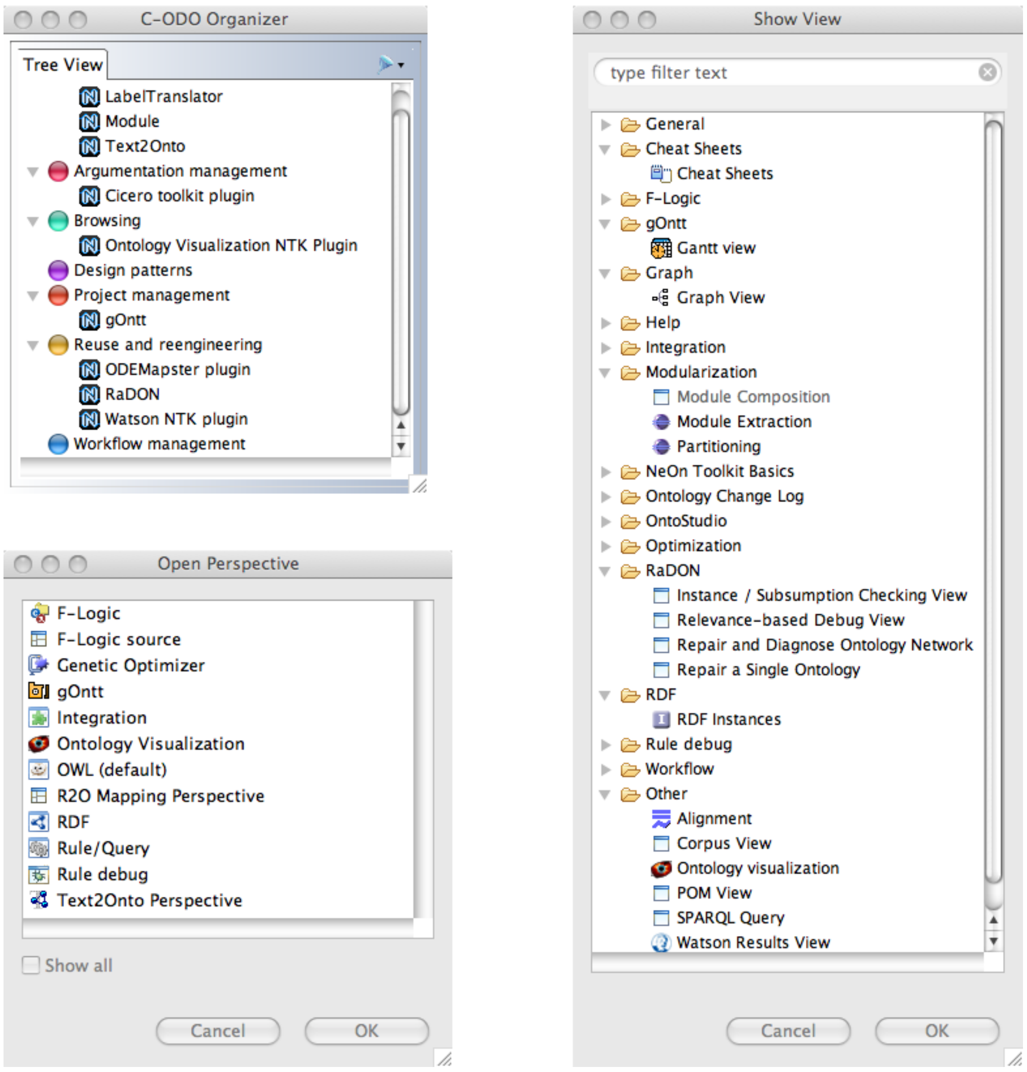


Figure 2.2: Comparison of the C-ODO organizer widget (top-left) against two standard user interface selection dialogs provided by the NeOn Toolkit i.e. the Perspective selector (bottom-left) and the View selector (right). It is shown how the organizer widget adopts a uniform strategy for classifying plugins that, in the other dialogs, appear incongruously categorized, or do not appear at all. These windows all refer to the same running instance of the Toolkit.

2.2 all refer to the very same running instance of the Toolkit.

Let us first analyze the two dialogs proposed by the Eclipse platform. One can easily surmise that the Perspective dialog (bottom-left corner of Figure 2.2) provides an overview of the system that is partial, unstructured and conceptually falling short of significance. Its flat list shows all available perspectives ordered by the name that was assigned to them at development time. This reflects the subjective point of view of each independent contributor: some perspectives are named after the logical formalism or standard supported (F-Logic, OWL, RDF...), some after the plugin that issued them (gOntt, Text2Onto...), and some seem to recall the purposes they serve (Integration, Ontology Visualization, R2O Mapping, Rule Debug...). The dialog for opening views (right side of Figure 2.2) shows the same drawbacks as the one for perspectives, yet in a much larger scale. This is due to the natural abundance of single views, which represent by far the most common way to contribute to the Eclipse RCP user interface, and are therefore extensively employed by the vast majority of plugins. The View dialog does provide support for categories, which in fact allows views to be organized in a shallow taxonomy. However, as view categories are once again defined by plugin contributors, each projecting their own interpretation of the tool space to the taxonomy, the conceptual view becomes incoherent. Categories are named not only after specific plugins (RaDON, gOntt), languages (F-Logic, RDF) and functionalities (Help, Integration, Modularization, Rule debug, Optimization), but also after presentation patterns (Graph, Cheat Sheets) and other generic or undefined criteria. This heterogeneity, compounded with the proliferation of views provided by each plugin, may introduce clutter and confusion and ultimately cause the user to lose orientation in a crowded ontology engineering environment.

Additionally, the user interface elements that users may choose to display can limit their knowledge of the functionalities available on the running NTK platform they have before themselves. Not all NTK plugins contribute to the user interface of the platform by means of perspectives or views; in fact, some of them, like *Cicero* or *Label Translator* provide extensions by means of menu items, hence no entry would appear in either dialog to testify that such tools even exist.

The C-ODO organizer, shown in the top-left corner of Figure 2.2, attempts to make up for such conceptual inarticulateness. In this example, installed plugins are displayed in groups with respect to one particular classification criterion (cf. section 2.4.2), i.e. the aspect(s) of ontology design they are known to cover. Plugins are listed regardless of the interaction path needed to access them, so long as adequate semantic representations of them are fed to Kali-ma. This correspondence is, in general, neither bijective nor total: a plugin that encompasses more than one design aspect will appear as a child of multiple nodes in the taxonomy. Also, plugins for which an ontological description exists, but does not provide a set of facts that conform to classification rules, are listed as “unknown”, as Kali-ma’s reasoning features are unable to determine what categories these plugins fall under.

Semantic Nomenclature example

A project manager involved with the Semantic Nomenclature case study has setup a development plan for the whole implementation phase. He wishes to make a recommendation of necessary NeOn Toolkit plugins to ontology engineers that will perform the activities involved. However, he does not know which plugins should be most useful from the beginning, nor is he able to determine immediately which of them are available since, for example, Perspective and View selector dialogs do not say anything about the availability of the Cicero plugin for ontology argumentation. He then launches the Kali-ma dashboard after having chosen “supports activity (Activity)” as the preferred classification criterion. The C-ODO organizer displays a tree-structured view on plugins organized by NeOn methodology activities.

As hinted in the previous paragraph, this high-level (i.e. design and functionality-related) knowledge of plugins comes exclusively from OWL descriptions that are authored in accordance with the ontological component of Kali-ma, which in turn is based on the *codolight* networked ontology [GP09] (cf. Section 4.4). These ontologies also include statements describing plugins at a lower abstraction level, e.g. their dependency on other plugins or platforms and their unique identifier in the OSGi framework, which constitutes the basis for the Eclipse RCP plugin mechanism that is used in the NeOn Toolkit. Kali-ma exploits these low-level statements in order to link the complete tool space with the actual real-world environment it is running on, and determine which of the NTK plugins that exist are installed on that environment at runtime.

By either double-clicking one or more entries representing installed plugins, or selecting the “Open widgets” context menu option once these entries have been selected, the corresponding plugin widgets are opened. Multiple widgets may be opened in any order and this does not affect the order in which the user interacts with the actual plugins. From this angle, the dashboard can be seen as a set of bookmarks for favorite or frequently used plugins, so the user is free to access a plugin by clicking the “Open” button from its widget, do some work with that plugin, then switch back to the Kali-ma dashboard if in need for another tool.

Aside from being generated from harmonic collective semantics of plugins, the view on the NTK platform that is provided by the C-ODO organizer can also be filtered. The funnel-shaped toolbar icon in the top-right corner of the widget provides some optional rules for skimming the classified content. The user can setup these filters in order to show not only the whole set of known NeOn Toolkit plugins, no matter whether they are installed on the current system, but also how *all* known ontology design tools are classified.

Semantic Nomenclature example

Having scanned the list of plugin-activity associations, the project manager picks the following plugins by double-clicking the corresponding entries and opening widgets for them: the *XD Tools* and *NTK core* for ontology reengineering and implementation, *ODEMapster* for ontology population, *Cicero* for both ontology argumentation and documentation, *RaDON* for ontology validation, the *Alignment plugin* for ontology alignment, *OWLDoc* for ontology documentation, and *gOntt* for scheduling this plan formally and keeping track of its progress.

We have opted for allowing users the freedom to browse classified tools that are not NeOn Toolkit plugins, for the simple reason that design tools for networked ontologies within the scope of NeOn can be, and are indeed being, offered in different ways and shapes. For instance, some tools (e.g. Cupboard, SerachPoint or Codomatic) are released as Web applications, while others (e.g. an early version of ODEMapster) come as standalone desktop applications. All of these types of tools can be shown or hidden in the organizer view per user preference. In Figure 2.2, both filters mentioned above are applied.

2.4.2 Categorization rules support

We have claimed that the capability of organizing the tool space, according to logical classification schemes that users are either accustomed to or comfortable with, is a key point in Kali-ma. On the other hand, heed must be paid to avoid enforcing one single logical view on the NeOn Toolkit platform, as it would potentially discourage users from adopting such an interaction mode, if only they found such an interpretation to be inappropriate or simply misaligned with their mental model of the Toolkit. Therefore, the aim was to provide a series of recommendations on what lenses to apply in order to view a running NeOn Toolkit instance as a provider of functionalities for the design of networked ontologies. On such a basis, a variety of criteria for the organization of the tool space has been incorporated in the first Kali-ma release. More precisely, three sets of rules, each representing a lens on the tool space, are built-in as part of the ontological component of Kali-ma. These are as follows:

Custom design functionalities denote functionalities arbitrarily defined by individual plugin descriptions.

NeOn methodology refers to the finite set of activities belonging to the NeOn methodology canon as defined in deliverables D5.4.1 [SFdCB$^+$08] and D5.4.2 [SFBd$^+$09].

Common aspects of ontology design are a proposal for the rules defined in D2.3.2 concerning recurring coarse-grained functionalities in the design of networked ontologies in a collaborative environment [PMP$^+$09].

While the first and third criteria can be viewed as the same interpretation of the tool space with different granularities, they can be seen as orthogonal to the NeOn methodology, as not always can a direct correspondence be defined between a design aspect or functionality and one of more activities. For instance, browsing or navigating an ontology does not map to a specific activity in the NeOn methodology, but can be considered as transversal to a number of activities such as Ontology Customization, Alignment, Enrichment and so on.

An advantage of these three sets of categorization rules is that, on the side of the single plugin description, they can be synthesized in a very small set of facts, possibly even a single statement. That is to say, classification by any of the aforementioned criteria can be performed by reasoning on a single property that holds for design tools and varies across criteria. It is sufficient to query a reasoner for all the design tools that are related to some other individual through a specific property, and the whole overview on the tool space becomes available.

It has to be noted that the binding with C-ODO Light is not lost on any of these lenses. Although complete freedom is theoretically given with respect to defining rules for organizing tools, these rules can never be completely independent on the underlying C-ODO Light model. Reasons for such a persistent dependency are to be found in the need for aligning each and every set of rules with a common foundational ontology, which allows the Kali-ma reasoning capabilities to have a common understanding of basic notions. For instance, all the sets of rules that are supported ultimately imply the need for reasoning on entities that are design tools. The definition of design tool is then the *invariant* of Kali-ma's categorization functionality, and it was chosen to be the `codkernel:DesignTool` class as declared in the C-ODO Light core module. Any definition of classes describing NeOn Toolkit plugins or ontology design tools in general is required to provide an alignment with `codkernel:DesignTool`, either by equivalence or by subsumption.

Custom design functionalities

The C-ODO Light core vocabulary[1] provides the set of upper classes covering descriptive, linguistic and social aspects of ontology design. The `codkernel` ontology defines `DesignFunctionality` as the class of tasks to be performed within an ontology project. Just as tasks are assignable to specific roles, design functionalities can be implemented by specific design tools, and this relationship is highlighted by the `[implements, isImplementedIn]` inverse property pair which, along with the class of design tools itself, is asserted in the C-ODO Light module for tool descriptions[2]. With this criterion selected, the DL reasoner used by Kali-ma constructs the tool taxonomy by inferring which design tools are related to which design functionalities through the `implements` property.

Because C-ODO Light does not provide ABoxes for design functionalities, it is generally up to the tool providers to explicitly state what are those implemented by their tools, most likely by instantiating several of such functionalities by themselves in the ontologies describing their own plugin. This can potentially lead to some degree of heterogeneity, as the granularity used for defining functionalities is a highly subjective parameter that greatly depends on the particular software engineering perspective that was adopted when outlining functional requirements for the tool. For example, "ontology argumentation" can be identified as a functionality as much as "annotate object property with issue" can be. However, the alignment with the C-ODO Light model guarantees a minimal degree of harmony in the classification. Additionally, the Codomatic service for the automatic generation of plugin descriptions, to be detailed in section 3.1, allows for reuse of functionalities defined by other plugin providers in their ontologies. For that matter, it is worth pointing out that design aspects described at the end of Section 2.4.2 are themselves design functionalities, therefore will appear in the classification along with provider-defined functionalities.

NeOn methodology

The Kali-ma plugin provides native support for the methodological guidelines for networked ontology engineering that were developed and in the context of WP5 (cf. [SFdCB^{+}08] and [SFBd^{+}09]). These guidelines include a wide range of methods, techniques and processes that encompass the whole lifecycle of networked ontologies, with strong orientation towards reusing existing resources.

The most prominent method by which Kali-ma supports the NeOn methodology is by offering an option to categorize ontology design tools with respect to the activities in the methodology that they support. This was done in order to take into account the maintenance of a harmonic environment for those users who are already familiar with and accustomed to such methodological guidelines. For example, if a lifecycle plan was scheduled by using the gOntt plugin (cf. [GPSFV09]), a user will be interested in executing activities and processes according to this plan. Kali-ma can be used for selecting a set of tools that best suit the activities that must be performed, in which case it is convenient for the C-ODO organizer to categorize ontology design tools using a similar criterion as the one that drove the scheduling of a gOntt plan.

[1]http://www.ontologydesignpatterns.org/cpont/codo/codkernel.owl
[2]http://www.ontologydesignpatterns.org/cpont/codo/codtools.owl

The glossary of NeOn activities is formalized as a simple OWL vocabulary that goes by the name `neonactivities.owl`[3]. At the time of writing, this vocabulary contains no less than 53 instances of the `Activity` class, which is asserted in the vocabulary itself. The `supportsActivity` object property, also asserted in the activity vocabulary, is used as the relation that defines this classification criterion.

Stating in the tool description what activities in the NeOn methodology are supported is straightforward: it is sufficient to import the `neonactivities.owl` ontology and adding a fact for each supported activity, using the `supportsActivity` property and the activities defined in the vocabulary. This procedure is supported by the Codomatic description generator (cf. Section 3.1).

Common aspects of networked ontology design

In [PMP$^+$09] we outlined a proposal for a set of customizable axioms that allows a tool to be classified within a group of five design aspects. These axioms were implemented as the `designaspects.owl` ontology[4], which has since been adapted as to make room for a few additional rules and is now offered by Kali-ma as an optional categorization criterion.

The variant for this criterion is identified as the `DesignAspect` class, which is asserted in the `designaspect.owl` ontology. Design aspects are a specialization of design functionalities at a higher level of abstraction than specific design functionalities provided by individual plugin contributors. Our proposal is now comprised of a fixed set of six design aspects, described as follows:

Reuse and reengineering refers to the process of retrieving existing resources, be they either OWL resources or of any other kind (ontologies, relational databases, text corpora etc.), and either transforming them into new conceptual models or using them in the development of an ontology.

Project management is the ability to author and manipulate metadata related to ontology development projects.

Workflow management refers to the manipulation or automated execution of ontology development workflows, or steps thereof.

Argumentation management is the design aspect of tools that implement functionalities related to ontology design decisions: ideas, positions, arguments, and issues.

Design patterns denotes the support for solutions, including ontology design patterns and selected axioms.

Browsing is typical of design tools that offer special support for visualization of and navigation within ontologies. This aspect also encompasses ontology editing, so long as it is performed within the very same navigation context.

Semantic Nomenclature example

In addition to the plugins previously selected, the Semantic Nomenclature project manager wants to suggest an additional visualization tool that could aid engineers in their tasks by displaying ontologies and entities in a convenient fashion. However, due to its ‘passive’ nature, ontology visualization is not an activity per se, as it is transversal to most activities. The user re-organizes the plugin taxonomy by selecting the “has aspect (Design Aspect)” criterion. The *Browsing* aspect now displays the *Ontology Visualization* plugin as one that suits his needs. He then selects this entry and the corresponding widget is added to the dashboard.

[3]http://www.ontologydesignpatterns.org/cpont/codo/neonactivities.owl
[4]http://www.ontologydesignpatterns.org/cpont/codo/designaspects.owl

There are several ways by which a tool can be stated to support certain design aspects, although only one of these can actually be considered a best practice for describing a tool. Support for a design aspect can be explicitly stated, but it can also be inferred from the knowledge that we have about plugins, i.e. what knowledge types they can consume as input or produce as output; what user categories are allowed to use a certain functionality.

For example, something can be classified as a Project Management tool if any of the following holds:

1. it is explicitly said to have the `ProjectManagement` aspect as a value;
2. it is said to implement a functionality that, in turn, is a specialization of the `ProjectManagement` aspect;
3. it is said to produce some output that is of the Project Description knowledge type.

Statements (1) and (2) are both viable ways of supplying specific knowledge on what design aspect are supported by a tool. However, by best practice, they are not intended for direct usage and are not available when generating a tool description using Codomatic. The reason is that tool descriptions, by default, have a direct dependency on the plain C-ODO Light network, which is explicitly imported (along with the ontology describing the NeOn Toolkit, if the tool is a plugin for it), while `designaspects.owl` is also built on top of C-ODO Light, and is not intended for consumption by plugin providers. In particular, (1) is only provided in order to infer the value, not to be directly attached to the tool description. Of course, there is nothing stopping providers from importing the design aspects ontology and making these statements explicit. However, the golden rule, instantiated in (3), is to remain agnostic to the design aspect rules when producing a tool description, and simply input as many functionality-related facts as are known for that tool. It is Kali-ma's job to infer which design aspects are supported by reasoning on these facts.

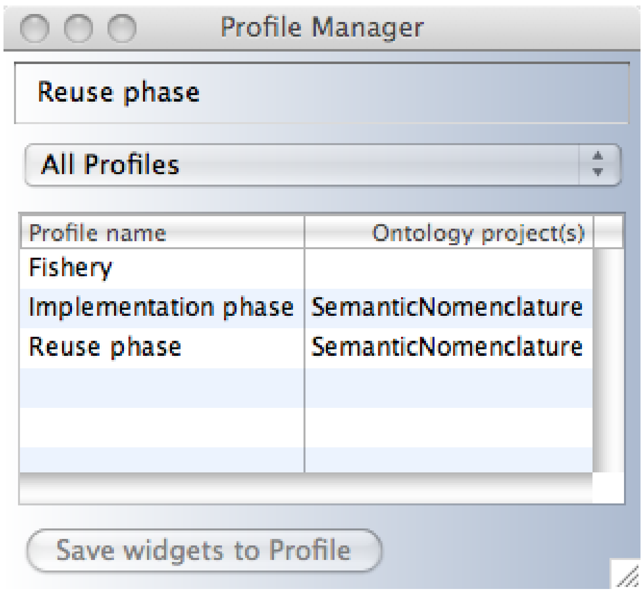
2.4.3 Profile management

A selection of plugins to be displayed as widgets in the Kali-ma dashboard could be of much more use than simply assisting a single user during a single engineering session. If an open dashboard were just a volatile object that had to be manually rebuilt from scratch every time the NeOn Toolkit is restarted, not only would it be awkward to share in a collaborative context (which is assumed to be recurrent in NeOn-compliant ontology engineering); it would also discourage users and project managers from adopting Kali-ma to support medium and long-term phases in an ontology engineering project.

In order to counter these preposterous potential shortcomings, Kali-ma offers a *profile management* functionality, which is concretely available as a native widget by its own right. The *Profile manager* widget, depicted in Figure 2.3, allows users to store, open and manage dashboard profiles.

A dashboard profile is essentially a named sorted set of plugins that can be serialized as an XML element and lives in the scope of both NeOn Toolkit workspaces and single ontology projects. Having performed a selection of plugins, all of which have a corresponding widget open in the Kali-ma dashboard, the user is able to retain this selection of plugins for sharing or future reuse. To do this, it is sufficient to type a name for the new profile in the top area of the widget and click the “Save widgets to Profile” button in the bottom area. This done, the current set of plugins is stored locally in the `kalima_profiles.xml` file in the workspace metadata directory for the Kali-ma plugin. Profiles can be listed, renamed or deleted and one at a time can be set as active and displayed on screen by opening the corresponding set of widgets. These operations are made available through context menu actions on the table occupying the middle portion of the widget.

Although dashboard profiles exist by their own right in a given NeOn Toolkit workspace, it is possible to bind them to one or more ontology development projects. This operation is also available as a context menu action, and its effects are visible on the second column in the table, which displays the names of the ontology projects to which a profile is bound to. Binding a profile to one or more ontology projects results in saving a copy of that profile in another `kalima_profiles.xml` file, this time placed in the project directory. This



Profile name	Ontology project(s)
Fishery	
Implementation phase	SemanticNomenclature
Reuse phase	SemanticNomenclature

Figure 2.3: Profile manager widget. Three profiles have been stored and are displayed in the profile table. Two of them (named *Implementation phase* and *Reuse phase*) are bound to the *SemanticNomenclature* ontology project.

action implies the ability to carry profiles along with a single project when it is exported to another system, as it is a common practice to share entire projects in Eclipse environments.

Semantic Nomenclature example

The project manager wishes to share his selection of tools for the Implementation phase with all the ontology engineers who are set to perform each activity. He uses the profile management widget to save the set of plugins as a profile named “Implementation phase” and associates it to the “SemanticNomenclature” ontology development project in the NeOn Toolkit. Because all participants are synchronized on this project via a Subversion repository, once the change is committed, they will all get a copy of the new profile the next time they check out the project.

2.4.4 Dashboard control and docking

The demand for an additional widget originated from the evaluation of an early prototype of the plugin during the FAO training session (see chapter 5). In the course of such evaluation, numerous test subjects sustained the claim that end-users were not willing to have each and every widget displayed on-screen, even though it belonged to the set of plugins selected by users themselves. On the other hand, relying to standard window-closing methods was deemed an inappropriate solution, as such an action was logically associated to the intent of removing the corresponding plugin from the set of those which users wish to use for the current session, thus releasing all resources associated with that plugin.

Our solution proposal comes as an additional dashboard widget called Kali-ma Dock. As its name suggests, the Dock is conceptually inspired by a consolidated praxis in modern operating systems, which provide a user interface feature for quickly switching between applications. In our interpretation, the Kali-ma Dock provides a compact user interface for holding references to elements of the dashboard that are not of immediate interest yet it still makes sense to hold in the current view of the system. For example, the user may want to remember having selected a certain plugin, but does not need to access it in that particular instant. Every widget that supports docking comes with a toolbar button that, when clicked, instructs the dashboard controller to hide that widget and add a corresponding entry in the Kali-ma Dock. A dock entry is a very simple interface

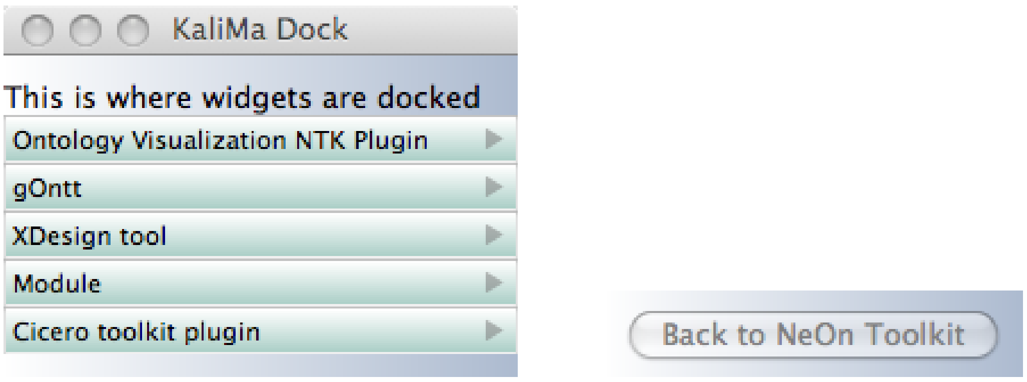


Figure 2.4: *(left)* The Kali-ma Dock with placeholders for five docked plugin widgets; *(right)* the Dashboard Control widget.

element that serves a placeholder for a docked widget. It consists of a label with the plugin identifier and an arrow button for restoring the docked widget to its original position.

The Dock widget itself responds to the same screen overcrowding issue that holds for plugin widgets and other dashboard widgets, therefore it is not visible on-screen at all times. The Dock hides itself every time the last docked widget is restored (i.e. there are no more dock entries) and becomes visible again once a widget is docked (i.e. a dock entry is added). This is due to the fact that, at this stage, the Dock serves the sole purpose of holding references to widgets that are hidden from view. This behavior may vary as further functionalities are added to the Kali-ma Dock in the future.

Semantic Nomenclature example

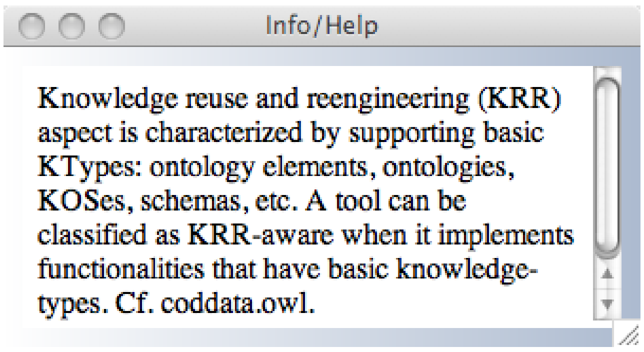
An ontology developer checks out the SemanticNomenclature project and receives the dashboard profile and gOntt plan for the implementation phase. However, he is only concerned at the moment with reusing resources from the BOTPlus database and populating reference ontologies. For this reason, he docks all open plugin widgets but those representing the ODEMapster and Ontology Visualization plugins. This way, he does not lose track of the other plugins like Cicero and OWLDoc, which he is going to need in the future, once the current activity is in an advanced state.

The visibility and operativeness of the Kali-ma dashboard can be easily controlled through an extremely simple widget, i.e. the `Dashboard Controller`. It is essentially a button for switching between the standard NeOn Toolkit workbench and the Kali-ma dashboard in as little as one mouse click. The dashboard is not disposed when the switch button is clicked: it is hidden from view and left in a dormant state, whereby it does not generate or react upon dashboard events, until the “Launch Dashboard” menu entry or toolbar item is selected again. Every time the user switches back to the standard NeOn Toolkit user interface, the Workbench is brought on top and restored, in case it had previously been set to be hidden or iconified.

2.4.5 Realtime help

Based on the feedback received in the early evaluation phase (cf. Chapter 5), where the need for guidance through plugin selection and dashboard features was an issue brought up by several participants, we have provided a realtime help system in Kali-ma. Our main concern was to display appropriate justification of each node appearing in the C-ODO Organizer taxonomy, and in doing so, to take advantage of any metadata present in the ontologies describing tools and classification criteria.

Realtime guidance is provided through the *Helper* widget, displayed in Figure 2.5. The Helper is essentially a lightweight Web browser capable of rendering HTML. However, it also reacts to local events within the dashboard, such as a particular widget being focused or a node being selected in the C-ODO organizer.



Knowledge reuse and reengineering (KRR) aspect is characterized by supporting basic KTypes: ontology elements, ontologies, KOSes, schemas, etc. A tool can be classified as KRR-aware when it implements functionalities that have basic knowledge-types. Cf. coddata.owl.

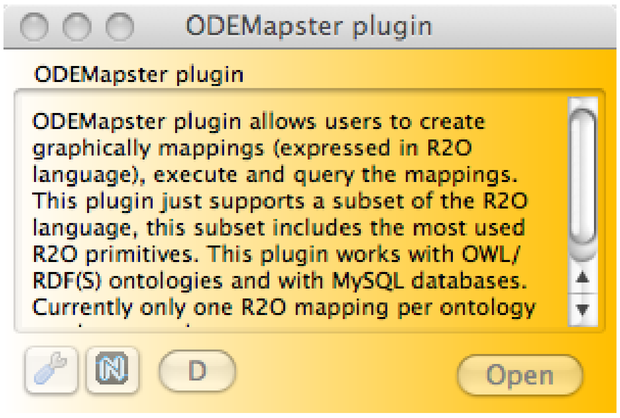
Figure 2.5: The Helper widget, displaying a plain-text definition of the Reuse and Reengineering design aspect after it was selected by the user in the C-ODO organizer.

While help messages related to native functionalities are hardcoded, those deriving from metadata such as OWL annotations derive from elements of the ontological component of Kali-ma, which also include remote tool descriptions. For instance, when a node is selected that represents a design aspect, NeOn methodology activity, functionality or design tool, the Helper widget displays the `rdfs:comment` annotation for the corresponding OWL individual, as is the case of the `ReuseReengineering` design aspect highlighted in figure 2.5.

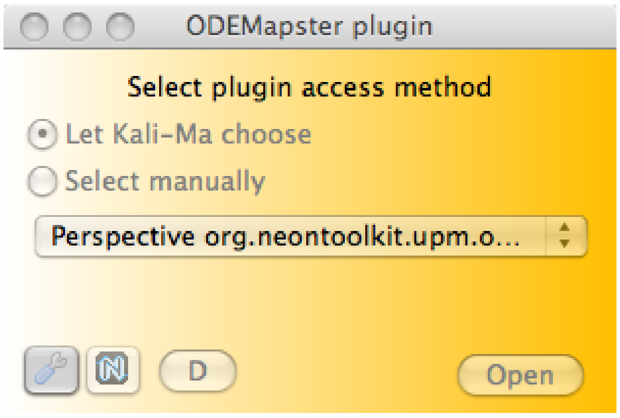
2.5 Plugin-supplied functionalities

As anticipated in Section 2.3, a plugin widget represents an installed NeOn toolkit plugin in the Kali-ma dashboard and allows the user to access the functionalities provided by that plugin by one click. The capability of detecting the installation status of an ontology design tool is subject to both semantic and platform-dependent characteristics. Kali-ma determines whether a tool is installed by matching the low-level semantic layer of its description with the local extension registry of the NTK installation where Kali-ma is running. Details on the procedure and preconditions for this functionality are deferred to Section 4.4.

While built-in Eclipse dialogs like the ones presented in Figure 2.2 are specific for one interaction mode each, such as perspectives, views and wizards for importing, exporting or creating items, Kali-ma defers this choice to the scope of the single plugin. Contributing to the NTK platform by means of a view or a wizard is not a requirement for Kali-ma to show a widget for that plugin, but the widget itself then allows the user to select one among the applicable interaction modes from that plugin.



ODEMapster plugin allows users to create graphically mappings (expressed in R2O language), execute and query the mappings. This plugin just supports a subset of the R2O language, this subset includes the most used R2O primitives. This plugin works with OWL/RDF(S) ontologies and with MySQL databases. Currently only one R2O mapping per ontology



[✓ Let Kali-Ma choose] Select manually

Perspective org.neontoolkit.upm.o...

Figure 2.6: Two views of a Kali-ma sample plugin widget with basic functionalities. Left to right: the default view and the preference panel.

Figure 2.6 shows two distinct states of an example widget for the ODEMapster plugin, which manages mappings from relational databases to ontologies. The default appearance of this widget once it is generated and opened is shown on the left side of the figure. There, the widget body shows a free-form text description of the plugin functionalities. This description, as well as the plugin name shown in the top toolbar, is not hard-coded in the plugin itself, but is retrieved from metadata in its OWL description, namely the `rdfs:comment` and `rdfs:label` annotations on the individual corresponding to the plugin. Also, the amber-colored background for this widget is not due to a completely arbitrary choice: background graphics for each widget are strictly dependent on the category (i.e. design aspect, functionality or activity) that is associated to the corresponding plugin. This mapping to graphical objects can be explicitly asserted in an ontology[5] that is aligned with the standard C-ODO Light module for interface objects[6] and allows to assign specific graphical objects to virtually anything. Consequently, anything that is mapped to a user interface object that allows background graphics can implement the referenced graphical object. In the case of the figure, the ODEMapster plugin has Reuse and Reengineering as a design aspect, which in turn has been associated to a white-to-amber horizontal linear gradient.

The first two buttons in the bottom row allow the user to cycle between the three different states of a plugin widget. The button with a wrench icon switches to the settings panel, shown on the right side of Figure 2.6. Here, the user is given a choice for the “plugin access method”, i.e. the preferred or most appropriate interaction mode among those available for that plugin. The next section will outline the supported interaction modes and heuristics for automatic selection.

The button with the ‘N’ of NeOn switches to an information panel where all the activities supported by that plugin, as per the NeOn methodological guidelines, are listed. This view is always available regardless of the classification criterion that was chosen for categorizing ontology design tools. The ‘D’ button minimizes the widget to an entry in the Dock widget, which is also displayed if no other widgets were previously docked. Finally, the rightmost “Open” button takes the user to the plugin interface through the selected access method. Depending on the nature of this access method, such an action may or may not cause Kali-ma to hide the dashboard and switch back to the NeOn Toolkit workbench (cf. next section).

2.5.1 Access methods

The standard mechanism by which a plugin is integrated with the Eclipse Rich Client Platform is by implementing *extension points*. An extension point allows a plugin to provide a contribution to the hosting platform at both the functional and user interface level. The latter in particular includes a set of standard user interface objects that a plugin can implement to enrich the interactive experience with the platform. Some of them, such as wizards, views or perspectives, can be standalone elements that can be displayed without any need for prior action upon other user interface or content items. For example, a wizard for exporting a given resource in a given format might depend on the user having previously selected the resource to export, but it might also allow the user to select that resource from a browser within the wizard instead. Conversely, other extension points contribute to the user interface by providing items that strictly depend on the interaction context. For example, context menu items will require the user to request a context menu on an item (typically by right-clicking on it). Therefore, running the action associated with a context menu item with no prior selection would make little sense, and would in fact be unlikely to even work.

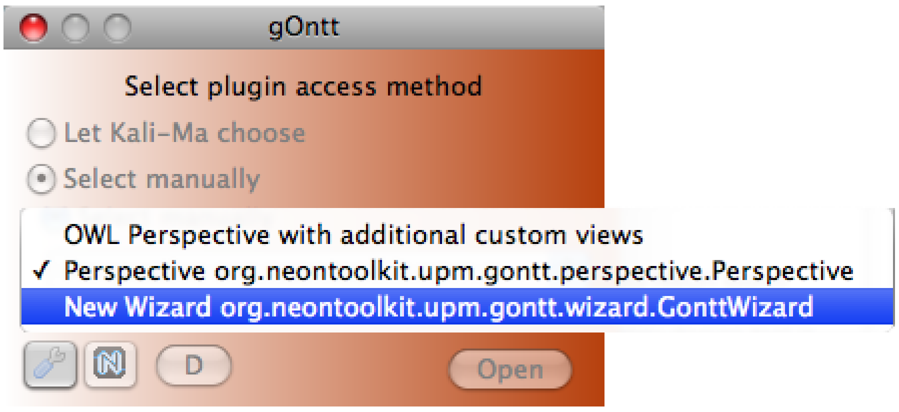
The current version of the Kali-ma plugin allows users to run NeOn Toolkit plugins through the following standalone access methods:

1. *Views* are single panels within the Eclipse workbench that serve as containers for arbitrary user interface controls. Multiple views can be aggregated in container objects, called Folders, which are essentially tabbed panes where each tab allows to display one view at a time within the same folder. Views are usually associated to single use cases, such as displaying the results of a SPARQL query, and can be manually moved across folders.

[5]http://www.ontologydesignpatterns.org/cpont/codo/kalimainterfacecustom.owl
[6]http://www.ontologydesignpatterns.org/cpont/codo/codinterfaces.owl

2. *Perspectives* are named composite panels that combine a group of folders and views in a predefined fashion. View combinations are usually associated to entire functionalities, which can be performed by interacting with the user interface elements in each view. Single views can only be shown within a perspective, and the NeOn toolkit provides a default perspective for authoring OWL ontologies.

3. *New Wizards* are paged dialogs for guided creation operations. The list of available New Wizards in a system can be accessed from the "New" item in the "File" menu. Examples of this access method allow users to create ontology development projects, ontologies and gOntt plans. While we cannot rule out cases where new resources have to be created from existing ones (e.g. ontologies need to be created within an existing project), many New Wizards are associated to standalone use cases for creating new resources from scratch.


gOntt

Select plugin access method

( ) Let Kali-Ma choose
(•) Select manually

OWL Perspective with additional custom views
✓ Perspective org.neontoolkit.upm.gontt.perspective.Perspective
New Wizard org.neontoolkit.upm.gontt.wizard.GonttWizard

Figure 2.7: Two views of a Kali-ma sample plugin widget with basic functionalities. Left to right: the default view and the preference panel.

Figure 2.7 shows an example selection of access methods for the gOntt plugin (whose widget sports a white-to-rust gradient background, as this is the graphical object assigned to the Project Management design aspect). The gOntt plugin contributes to the NeOn Toolkit by means of both a Perspective (named `org.neontoolkit.upm.gontt.perspective.Perspective`) and a New Wizard (named `org.neontoolkit.upm.gontt.wizard.GonttWizard`) for scheduling new plans. A user can select either access method for launching the gOntt plugin once the "Open" button is clicked.

Semantic Nomenclature example

Another engineer wants to reuse Ontology Design Patterns by means of the XD Tools plugin when implementing reference ontologies. He is aware that this plugin provides its own perspective, but having never explored the "New" dialog, he does not know it also comes with a wizard for creating new ontology modules. From the settings panel of the XD plugin widget, he becomes aware of the existence of this wizard, and decides to launch that instead. Once the process is complete, the Kali-ma dashboard becomes available again.

In addition to these two options, the one displayed on the top of the combo box allows the user to open the default NeOn Toolkit OWL perspective along with all the views that are contributed by that plugin (the "Gantt" view in the case of gOntt). This option is always available for all plugins, even when they are not declaring any custom views, in which case the simple OWL perspective is opened. Note that not all views that a plugin creates are explicitly defined in the plugin manifest and can be manually opened e.g. from the View selection dialog. Some are created within the Java code of the plugin without having been previously defined. These are not accessible programmatically and are not supported by Kali-ma.

Additional extension points, such as actions, cheat sheets and export wizards, are planned to be supported in the next version of the Kali-ma plugin (cf. Chapter 6).

Chapter 3

OWL plugin description management

As will be presented in Section 4.4, the Kali-ma infrastructure includes a semantic layer involving components that are invariant in the domain of collaborative ontology engineering, as is the C-ODO Light network, and others that can be customized and adapted to new and refined taxonomies and criteria, such as the rules for categorizing the tool space. Standing amid these two levels are the real-world entities, i.e. the ABoxes where actual ontology design tools are instantiated and facts are provided for them. Kali-ma has no built-in or prior knowledge of which design tools exist, whether C-ODO Light-based ontologies describing them are provided and what physical URIs should be dereferenced for locating these descriptions. It does, however, provide a mechanism for locating such ontologies from a single, configurable source. Coupled with this mechanism, we are offering an online service for semi-automatic construction of C-ODO Light based plugin descriptions. This chapter details the key functional characteristics of both features mentioned above.

3.1 Plugin description generator

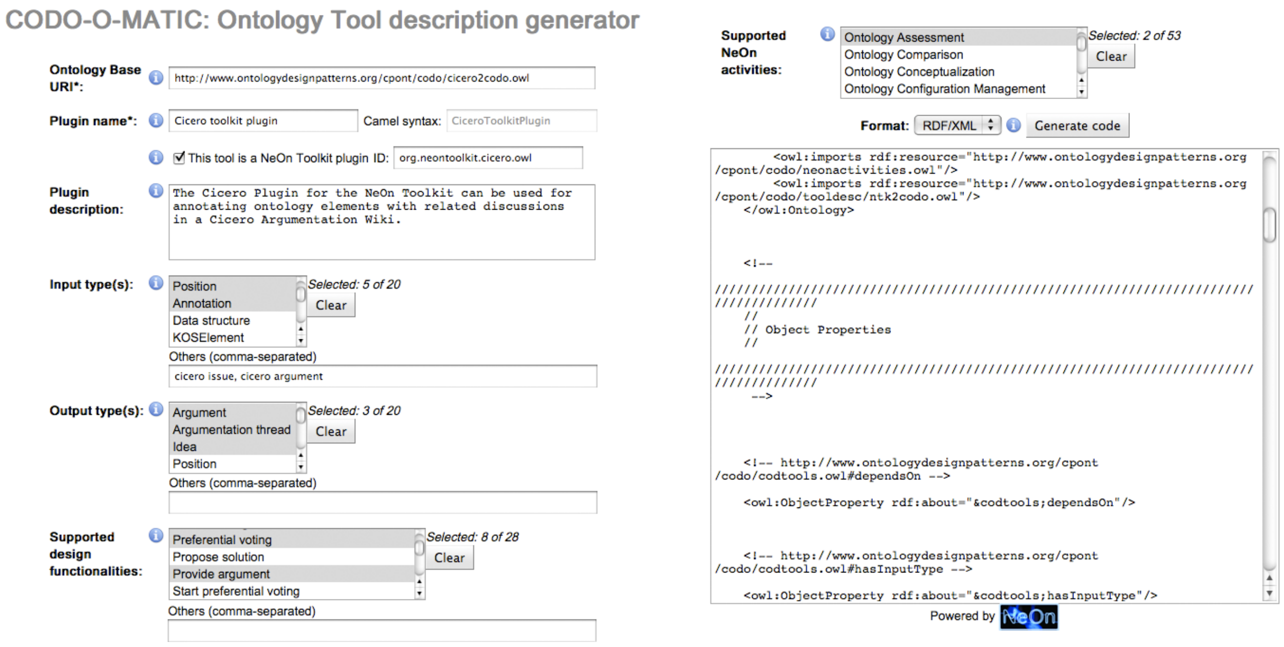
Unlike our previous proposal for a library of ontology design patterns [PGD+08], we do not aim at delegating the whole knowledge of the ontology tool population to a single online repository. It is the plugin provider's call to author pieces of structured knowledge concerning their own products, thus it is reasonable to expect them to remain depositaries of this knowledge, while at the same time sharing it in an open environment such as Linked Data. Concurrently, it was felt convenient to have a system for aggregating references to these ontologies at disposal, rather than crawling the whole Semantic Web.

In an effort to meet both demands, an interactive tool for constructing these OWL tool manifests was devised as a service to be available anytime, anywhere. A working prototype of this service was released as C-ODO-o-matic (simply dubbed *Codomatic* throughout this paper)[1], its name paying homage to an inspiring online form for generating FOAF profiles. Codomatic is a simple, single-page Ajax application for constructing codologht-based OWL manifests of ontology design tools bearing the minimum set of axioms for allowing a DL reasoner to categorize the tool with respect to any of the three supported classification criteria. The Codomatic service features willfully essential styling, so as to keep it open to embedding within Wiki pages or Web frames.

A sample of running Codomatic code generator for the Cicero plugin, depicted in Figure 3.1, shows what minimum user input is required and leveraged for the generation of the corresponding RDF code. The *Ontology base URI* field provides the default namespace for any new entities asserted in the ontology to be generated, and is advised to match the physical URI to be dereferenced for locating the ontology itself. The *Plugin name* field, along with its Camel syntax version, denote respectively the `rdfs:label` annotation and the actual URI local name for the OWL individual that identifies the tool itself, while the *Plugin description* field denote the English `rdfs:comment` annotation for that individual. It is possible to state the tool in question to be a NeOn Toolkit plugin, in which case its unique identifier must be supplied.

[1] At the time of writing, the service is hosted at http://150.146.88.63:8080/codomatic

The list boxes that follow allow providers to include functional specifications of their tools: through these interface objects, it is possible to select an arbitrary number of knowledge types that the tool is known to consume as input or produce as output, as well as the design functionalities and NeOn activities that it supports. For all fields but the NeOn activities one, an additional text box is available, where the provider can arbitrarily instantiate new knowledge types and design functionalities, if the existing ones are felt to fall short of accuracy or completeness in describing the tool in question. However, while new design functionalities can immediately be exploited when classifying a set of tools with respect to them, new knowledge types cannot contribute to the rules for inferring supported design aspects, unless the providers include additional defined classes that are restricted on the `hasInputType` or `hasOutputType` properties for their new knowledge types.


CODO-O-MATIC: Ontology Tool description generator

Ontology Base URI*:	http://www.ontologydesignpatterns.org/cpont/codo/cicero2codo.owl
Plugin name*:	Cicero toolkit plugin — Camel syntax: CiceroToolkitPlugin
	☑ This tool is a NeOn Toolkit plugin ID: org.neontoolkit.cicero.owl
Plugin description:	The Cicero Plugin for the NeOn Toolkit can be used for annotating ontology elements with related discussions in a Cicero Argumentation Wiki.
Input type(s):	Position, Annotation, Data structure, KOSElement — Selected: 5 of 20 — Clear — Others (comma-separated): cicero issue, cicero argument
Output type(s):	Argument, Argumentation thread, Idea, Position — Selected: 3 of 20 — Clear — Others (comma-separated):
Supported design functionalities:	Preferential voting, Propose solution, Provide argument, Start preferential voting — Selected: 8 of 28 — Clear — Others (comma-separated):
Supported NeOn activities:	Ontology Assessment, Ontology Comparison, Ontology Conceptualization, Ontology Configuration Management — Selected: 2 of 53 — Clear
Format:	RDF/XML — Generate code

```
        <owl:imports rdf:resource="http://www.ontologydesignpatterns.org
/cpont/codo/neonactivities.owl"/>
        <owl:imports rdf:resource="http://www.ontologydesignpatterns.org
/cpont/codo/tooldesc/ntk2codo.owl"/>
    </owl:Ontology>



    <!--

////////////////////////////////////////////////////////////////////////
//////////////
    //
    // Object Properties
    //

////////////////////////////////////////////////////////////////////////
//////////////
     -->




    <!-- http://www.ontologydesignpatterns.org/cpont
/codo/codtools.owl#dependsOn -->

    <owl:ObjectProperty rdf:about="&codtools;dependsOn"/>



    <!-- http://www.ontologydesignpatterns.org/cpont
/codo/codtools.owl#hasInputType -->

    <owl:ObjectProperty rdf:about="&codtools;hasInputType"/>
```

Powered by NeOn

Figure 3.1: The *Codomatic* tool description generator, after constructing the OWL ontology for the Cicero plugin.

The aforementioned statement supports the claim that by no means is Codomatic intended to serve as a replacement for a full-fledged OWL editor. The service is intended for the creation of minimal OWL manifests based on C-ODO Light, and yet it leaves room for extension and refinement. Providers can use the NeOn Toolkit OWL editor to add annotations for newly declared knowledge types and functionalities, and relate them to existing ones where need be, as well as define additional rules for inferring supported design aspects from knowledge type statements.

The “Generate code” button triggers an asynchronous remote procedure call to a servlet that encapsulates submitted data and uses the same OWL API as Kali-ma’s to output the corresponding ontology, whose source code is posted to the text area below the button. This code includes all the necessary ontology imports and is intended to be copied verbatim to an RDF document, which should then be uploaded to a location of the provider’s choice. Codomatic does not pose restrictions to tool providers as to what physical locations should be used for their newly generated ontologies, nor does it store submitted base URIs or any other information used for generating the OWL code. References to physical locations can be submitted through the corresponding plugin pages on the NeOn Toolkit Wiki, as documented in its plugin development and submission guide[2]. Being a Semantic Media Wiki, it is then possible to export these references in RDF format for Kali-ma to consume.

[2]http://neon-toolkit.org/wiki/Plugin_HowTo

3.2 Online update

The philosophy behind networked ontologies, which is incorporated with Kali-ma, addresses reuse as one of the pillars of ontology design. As per Section 3.1, it is not advisable for a single central repository to hold the entire knowledge of ontology design tools; in general, a distributed approach is encouraged and favored. At the same time, the Kali-ma tool is completely agnostic, with the sole exception of the NeOn Toolkit itself, as to what individuals populate this tool space, thus it needs a mechanism for being fed references to ontologies describing tools and retrieving them.

A proposed solution supported by the Kali-ma version 1.0 comes as an extremely simplistic set of references, which is published itself as RDF. The format used for this sort of 'address book' (which will be called as such in the remainder of this work) is compatible with codolight-compliant tool descriptions, yet does not strictly depend on codolight, nor does it define an ad-hoc vocabulary of its own. An address book for tool descriptions is a standalone (i.e. devoid of imports) ontology consisting of any number of statements of the form

$$x \ \texttt{rdfs:isDefinedBy} \ y$$

where x qualifies as an OWL individual and y is a literal of the `xsd:anyURI` datatype. An individual referenced on the lefthand member must match a `DesignTool` instance in the actual tool description: it does not need to be typed as such within the address book itself (though it is typed as an individual per the OWL metamodel), provided that the corresponding type assertion axiom exists in the tool description, otherwise that individual is discarded in the Kali-ma reasoning process. A URI in the righthand member of an axiom indicates the physical location where an OWL document describing that individual is to be found.

For instance, an address book entry for the Modularization plugin would look like the following (in Turtle syntax):

```
module:Module rdf:type owl:Thing ;
   rdfs:isDefinedBy "http://neon-plugins.googlecode.com/svn/trunk/
     PluginOntology/module.owl"^^xsd:anyURI .
```

where the `module` prefix stands for a namespace that matches the URI on the righthand side of the statement. This assertion roughly reads as "*A resource that defines a thing called Module can be found at http://neon-plugins.googlecode.com/svn/trunk/PluginOntology/module.owl*" (indicating that the C-ODO Light-based description is located in the main branch of a Subversion repository hosted by Google Code), and is interpreted by Kali-ma accordingly.

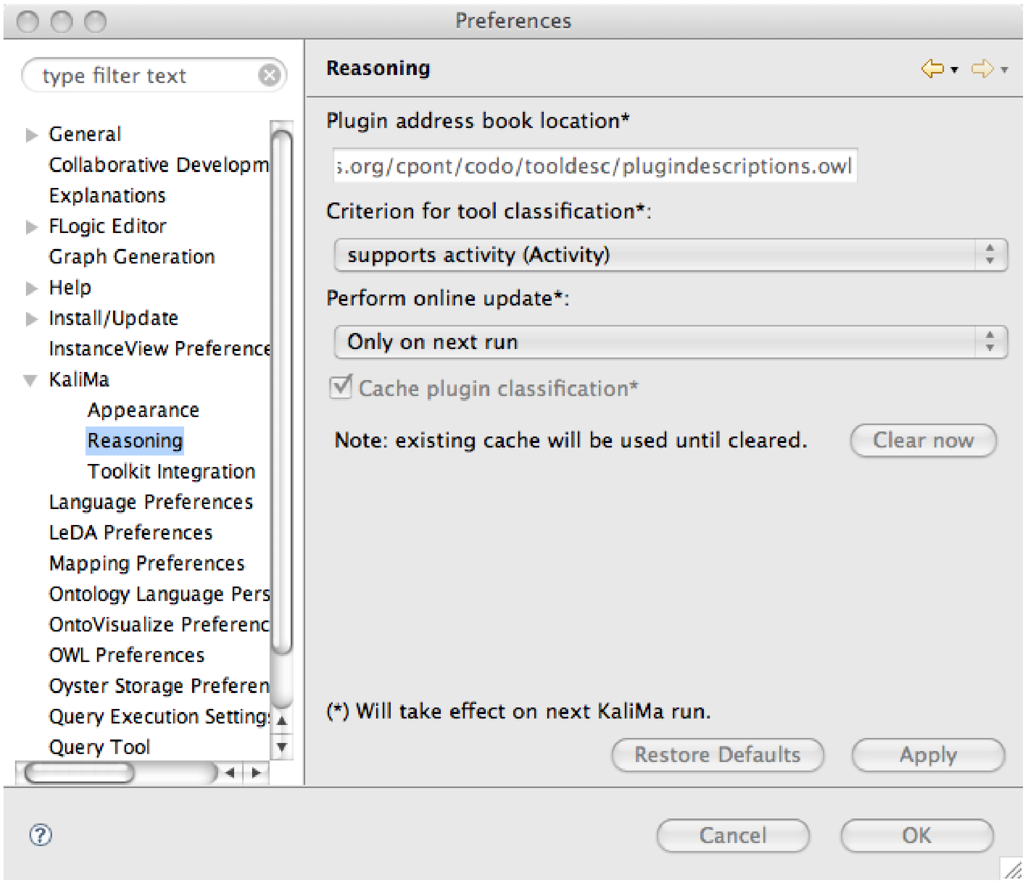
The metamodel used for address books is willfully generic and relies on widely used vocabularies such as RDFS annotations, in order to allow straightforward RDF export. Additionally, it is a preferable approach with respect to the use of import statements, for it does not force large ontology networks to be loaded by OWL managers or editors and defines a direct binding between design tools and their descriptions (which would be lost if using sheer import statements). Dereferencing physical URIs, retrieving OWL manifests and matching individuals with design tools is delegated to ontology management and DL reasoning tasks performed within Kali-ma.

We are providing a default address book specifically for NeOn Toolkit plugins, which is hosted on the ODP portal[3] but includes references to plugin descriptions that are scattered across the Web, including but not limited to the ODP portal itself. However, the extremely simplistic format makes it fairly easy for any content management system, such as Semantic MediaWikis, to export address books based on minimal knowledge on referenced tools and their descriptions.

The source of plugin description references is a customizable parameter that can be set through the Reasoning preference page for Kali-ma (see Figure 3.2), along with the frequency of online updates, which can

[3]http://www.ontologydesignpatterns.org/cpont/codo/tooldesc/plugindescriptions.owl

be performed only once, or at each run or never. Note that the third option is not taken into account the first time Kali-ma is run within the current workspace.



Reasoning

Plugin address book location*

;.org/cpont/codo/tooldesc/plugindescriptions.owl

Criterion for tool classification*:

supports activity (Activity)

Perform online update*:

Only on next run

☑ Cache plugin classification*

Note: existing cache will be used until cleared.

(*) Will take effect on next KaliMa run.

Figure 3.2: The Kali-ma Reasoning preference page. Note the text field for specifying the physical URI of the address book ontology and the combo box for selecting the update frequency.

Tool description address books, as well as the referenced ontologies, are not necessarily fetched remotely at all times. Although Kali-ma is intrinsically a Semantic Web tool, it has been designed to be able to work offline in contexts where collaboration is not an issue and ontology engineering tasks may be entirely performed locally. The only condition that must hold is that Kali-ma has previously been able to gather ontological tool manifests at least once. Each tool manifest that is retrieved from the Web and approved for update is merged with a local copy of the original address book. The merging process excludes all ontology annotations and retains all the other axioms, including import statements but excluding axioms asserted in imported ontologies. Merging is possible due to the simplistic model for address books being aligned with C-ODO Light, hence the definition of an individual design tool can be directly enriched with the set of facts coming from its description ontology, without any need for any transformation whatsoever. The resulting merged ontology is stored locally in the metadata directory for the workspace in use at runtime.

The flowchart in Figure 3.3 depicts how the update process is performed on a functional level. Initially, the system checks for a local ontology with merged tool descriptions. If said ontology is missing or does not describe any design tool, online update is forcibly performed with no further user interaction. Otherwise, the system checks if user preferences as per Figure 3.2 have been set to check for updates (i.e. the “Never” option is not selected). If so, the remote address book is fetched from the URI indicated in the top text field. The statements contained in the address book are checked against those present in the local merged ontology. In particular, the following changes are detected:

1. Previously unknown individuals with associated `rdfs:isDefinedBy` axioms;

2. `rdfs:isDefinedBy` axioms on known individuals with different URIs as their righthand side literals;
3. Orphan individuals, i.e. no longer defined by any `rdfs:isDefinedBy` axioms.

Changes of type (1) usually denote the addition of new design tools to the knowledge base and are the standard means by which Kali-ma enriches its knowledge of the tool space. Type (2) suggest a possible pattern for versioning support in ontology descriptions: by making URIs ‘version-aware’, e.g. by including the ontology version number as part of the URI, it is possible to determine if a description is to be updated, without having to load the new one first. When an update is applied due to a variation of type 2, all previous axioms describing that design tool in the local ontology are removed first. Changes of type (3) are simply for notifying the user, but no pre-existing axioms for orphan individuals will be removed.

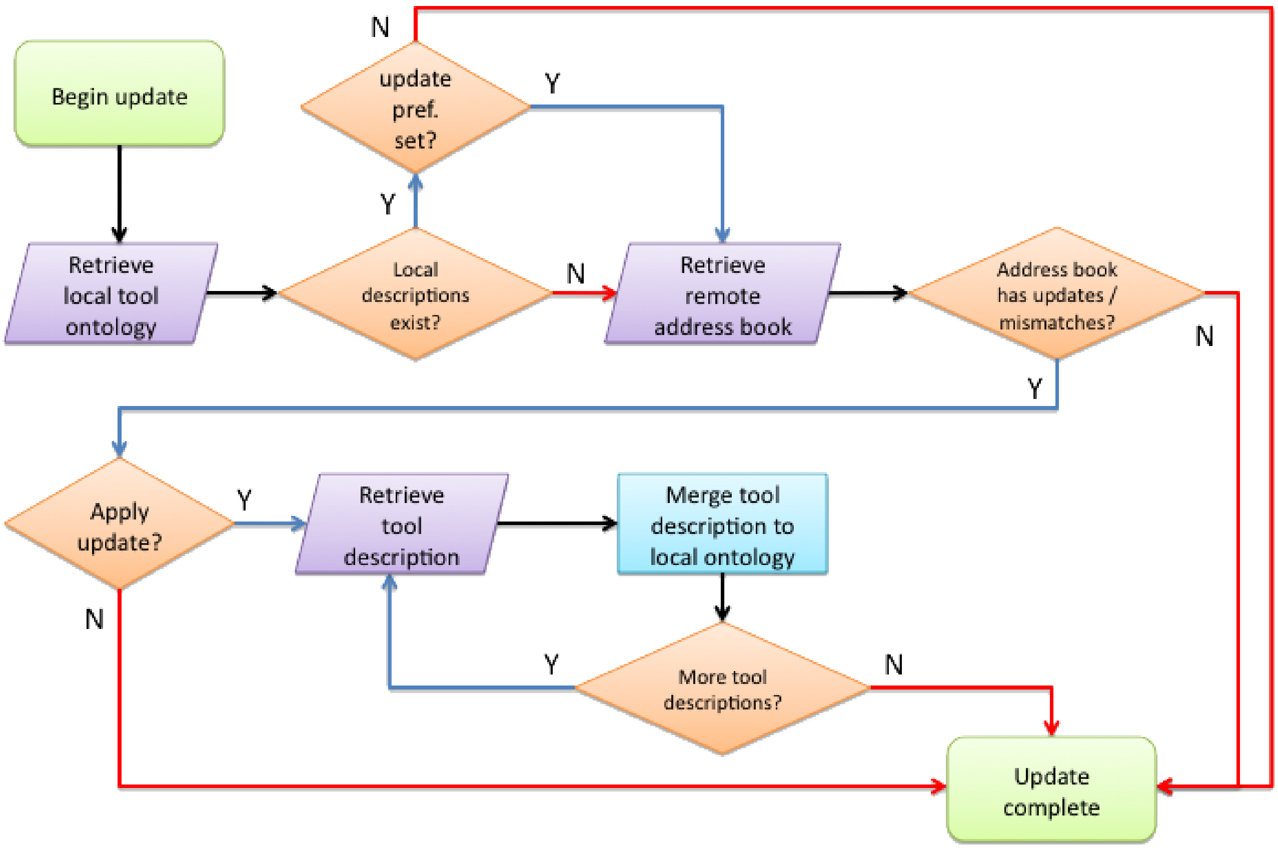

Begin update
Retrieve local tool ontology
Local descriptions exist?
update pref. set?
Retrieve remote address book
Address book has updates / mismatches?
Apply update?
Retrieve tool description
Merge tool description to local ontology
More tool descriptions?
Update complete
Y
N

Figure 3.3: Flowchart describing the online update process for C-ODO Light-base tool descriptions.

If at least one variation of type 1 or 2 is detected, the user will be notified about the amount of changes and prompted to apply the update. If the user chooses to do so, each ontology referenced by new `rdfs:isDefinedBy` statements is loaded and merged with the local one (which is created if not existing). The resulting ontology will be used for reasoning purposes in lieu of the whole original set of tool descriptions.

Chapter 4

Infrastructure

In this chapter, the fundamental elements of the Kali-ma architecture are outlined, with particular focus on its knowledge representation and reasoning capabilities. Not all the runtime components of the Kali-ma plugin consist of Java bytecode and Eclipse-compliant manifest files: as anticipated in Section 2.1, although the entire knowledge needed for managing the tool space is maintained in its original OWL formalism, this is treated in a similar fashion as runtime software libraries. However, in order to avoid confusion with the traditional software engineering doctrine, the term 'software architecture' will be used exclusively for referring to components that are coded in Java, while we will be referring to the whole union of software architecture and knowledge network as simply 'infrastructure'.

The Kali-ma infrastructure was devised in a loosely layered fashion, i.e. where communication between non-adjacent layers is minimized. A number of different reasons corroborate this design choice. Firstly, the heterogeneous representation of infrastructural components, such as Java packages and OWL ontology networks, are a strong motive to such a policy, more so if considering that the OWL component is only partially built-in, as the whole tool population knowledge resides on the Semantic Web. Another reason comes from having isolated infrastructural components that are both reusable and replaceable. As hinted in Section 2.1, ontology engineering and the NeOn Toolkit environment are but one field of application for an approach that can be applied to a myriad of domains and reference architectures, therefore we have designed Kali-ma with cross-context portability in mind. Finally, the layers that make up the infrastructural stack follow a similar approach as a traditional *Model-View-Controller* triad [Ree79].

The layered architecture of Kali-ma can be split into three major components as depicted in Figure 4.1: the *ontological component* is responsible for providing Kali-ma with the necessary knowledge about existing NTK plugins and the rules by which to classify them; the *reasoning component* manages the extraction of such knowledge from the ontological component, as well as the aggregation and classification of plugins; finally, the *presentation component* generates the widgets and handles communication between Kali-ma, NTK plugins and the NTK core.

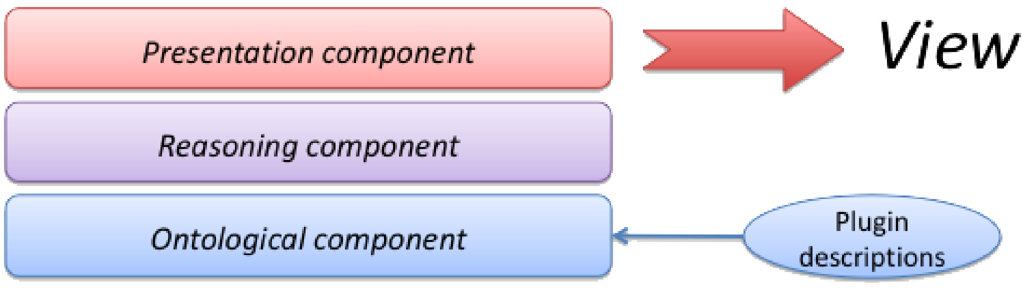


Figure 4.1: The Kali-ma infrastructure. From bottom to top: the ontological component, including external OWL descriptions of NTK plugins; the reasoning component; the presentation component, whose output is presented to the user interface view.

Before getting into the details of all three components top-down in sections 4.2, 4.3 and 4.4, the next section provides and overview of the most significant implementation choices with respect to integrating the Kali-ma tool with the Eclipse RCP platform and, more specifically, with the NeOn Toolkit.

4.1 Basic software architecture

By sheer technical terms, referring to Kali-ma as an Eclipse plugin, as is being done for simplicity through the whole paper sans this very section, is actually an incorrect practice. In an Eclipse environment, which in turn is based on the OSGi service infrastructure, *plugins* are fine-grained units contributing to the host platform in some way. As they are, plugins are unmanageable objects that do seamlessly integrate with the Eclipse platform, but do not allow for a number of configuration functionalities like enabling, disabling, branding, installing and uninstalling the contribution as well as managing its prerequisites. What in common parlance is called a plugin (meaning an installable and manageable set of software contributions) is actually what the Eclipse technical language refers to as a *feature*. Features package one or more plugins into a single, manageable unit and represent the form by which they are serviced as installable bundles. Most partners in the NeOn consortium provide their contributions to the NeOn Toolkit by means of Eclipse features.

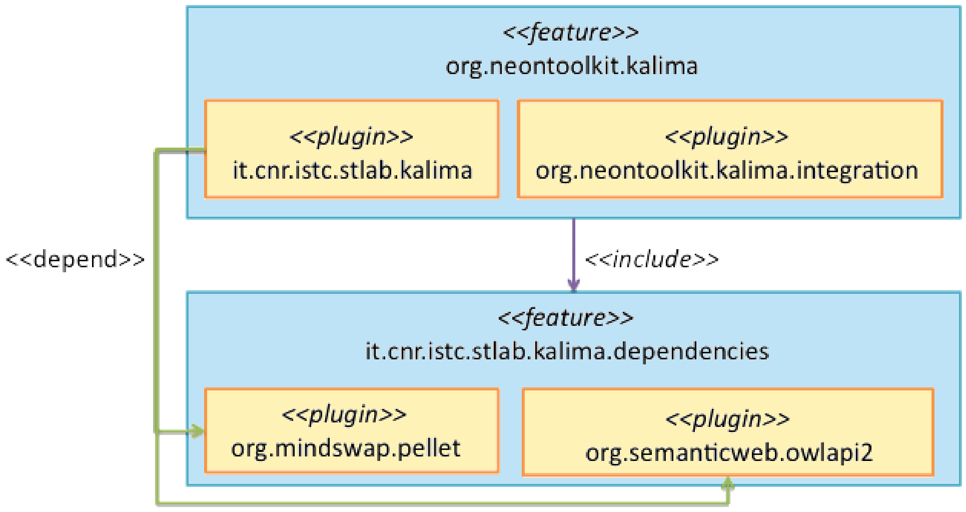


Figure 4.2: OSGi package layout for the Kali-ma tool. Two features make up Kali-ma: org.neontoolkit.kalima includes the core plugin and the NeOn Toolkit integration plugin, while it.cnr.istc.stlab.dependencies includes two plugins for the OWL API 2 and the Pellet reasoner respectively, which the core plugin depends on.

This premise was made necessary by the fact that, in this respect, Kali-ma is articulated in a number of such units. How these are packaged together is shown in the diagram of Figure 4.2. The tool is essentially comprised of two features. The main one is named `org.neontoolkit.kalima` and packages two plugins: the core component `it.cnr.istc.stlab.kalima` and a dedicated plugin that manages integration with the NeOn Toolkit (such as querying the ontology project space for storing profiles (cf. Section 2.4.3), named `org.neontoolkit.kalima.integration`. Additional third-party libraries are collectively packaged as the `it.cnr.istc.stlab.dependencies` feature. These include:

the OWL API version 2.2[1] [HBN07];

the Pellet DL reasoner, version 2.0rc5[2] [SPG$^+$07];

Reasons for packaging these libraries separately are both practical and legal. Development of Kali-ma

[1]http://owlapi.sourceforge.net
[2]http://clarkparsia.com/pellet

started early in the transition period where the OWL support based on the KAON2 API[3] was being shelved in favor of the OWL API. Therefore, it appeared appropriate to anticipate this migration by internally employing the OWL API for managing the C-ODO Light-based datamodel, at least until the same API would be exposed by a mature NeOn Toolkit update. However, the need for an individual reasoner (which will be justified in section 4.3) and the delay of a stable one to surface in compliance with the as yet unstable third version of the OWL API, brought us to adopt a conservative strategy. Therefore, a compatible pair of such libraries was packaged as a standalone feature. It was neither technically sensible nor legally possible to bundle these libraries with Kali-ma, as license restrictions on the Pellet reasoner and API (which, in its open source form, is licensed under the Affero GNU Public License v3[4]) prevent it from being bundled with both the NeOn Toolkit and Kali-ma, lest they be subject to that same license or the General Public License[5]. Additionally, the tool had to be ready to be ported to use the NTK-provided datamodel with as little effort as possible, once a stable platform is released for the upcoming deliverable D7.6.3.

4.1.1 NeOn Toolkit core integration

On the dependency side, the Kali-ma core plugin `it.cnr.istc.stlab.kalima` is a standard plugin for the Eclipse RCP version 3.3, though it can be compiled to run on versions up to 3.5 as well. As described in the previous section, specific functional integration with the NeOn Toolkit is provided as a separate plugin identified as `org.neontoolkit.kalima.integration`. Not only was this design choice aimed at keeping a core module that can be ported to other Eclipse environments, but also it was necessary for keeping up with the deep structural changes in the NeOn Toolkit datamodel, upon migration of the platform from Eclipse RCP version 3.3 to version 3.5. Additionally, it was regarded as sensible to keep track of NeOn Toolkit integration features separately as the bindings with this platform are tightened in future updates.

The following core integration features are available in the 1.0 version of Kali-ma:

1. Java interface `DataModelIntegration` for querying the ontology project space. Implementations of this interface are provided based on both the old KAON2-based and the new OWL API-based datamodels.
2. User interface-level integration for managing the behaviour of the NeOn Toolkit Workbench (i.e. its main window) when the Kali-ma dashboard is running.

Item (1) is needed for profile management functionalities to work (cf. Section 2.4.3). Without a specific module, the profile manager would only be able to store profiles as workspace metadata, while it would not be able to bind them to specific ontology projects. Future usage of the datamodel integration interface includes direct access to OWL resources within an ontology development project, e.g. for presenting them to the Dashboard system to bootstrap a plugin pipeline (cf. Chapter 6).

Item (2) addresses the problem of diminishing user bewilderment in dealing with two discordant user interface paradigms, i.e. the Eclipse Workbench and the Kali-ma dashboard. It is left to the user's liking to have the system treat the Kali-ma dashboard as a direct child of the Eclipse workbench, thus implying that both interfaces can be shown, iconified or disposed together. However, it is possible to have the Kali-ma dashboard behave independently on the originating main window, which can be automatically hidden or iconified when the dashboard is showing. This option can be set from the "Toolkit integration" preference panel.

4.2 Presentation component

The top-level component of the Kali-ma infrastructure, called *presentation component*, implements both the user interface and its controller in the aforementioned *Model-View-Controller* paradigm. This element is

[3]http://kaon2.semanticweb.org
[4]http://www.opensource.org/licenses/agpl-v3.html
[5]http://www.opensource.org/licenses/gpl-3.0.html

responsible for leveraging the underlying C-ODO Light-based object model and presenting the outcome of reasoning tasks performed thereupon. Widget generation, management and event handling support all belong to this subsystem.

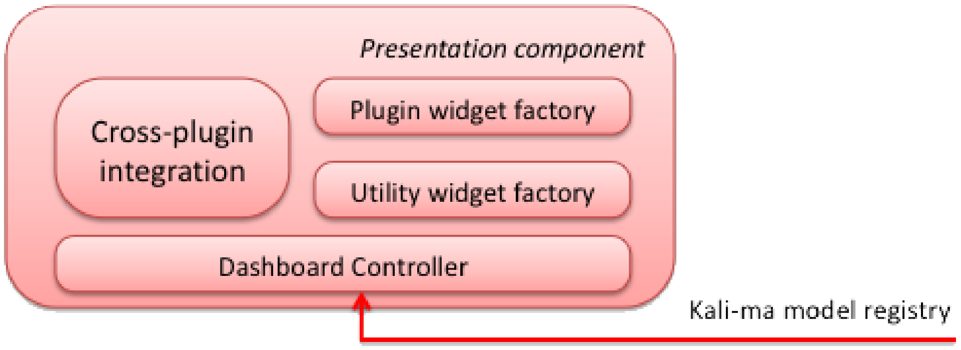


Figure 4.3: The presentation component of Kali-ma.

Below is an outline of the constituent elements of the presentation component, whose structure is sketched in Figure 4.3:

The *dashboard controller* is an OWL-agnostic module responsible for maintaining the loose coupling between data and user interface, as well as keeping them unaware of each other. In its abstract form, this module includes a Java interface with methods for triggering the generation, dispatching and disposal of widgets, and passing data from the underlying object model registry (cf. next section) over to factories. We are providing an implementation of this interface for the Standard Widget Toolkit (SWT), but more implementations could be made available for Swing or other windowing toolkits. The controller manages a *dashboard* object, which contains references to all native and plugin widgets and keeps track of its own state (including its visibility, activity and the docking state of each widget that supports docking) autonomously.

Widget factories are instantiated and invoked for the generation of constituent elements of the Kali-ma dashboard. A *utility widget factory* generates native widgets such as C-ODO organizers, profile managers, helpers, docks and dashboard controllers, and is invoked once per dashboard instance; multiple *plugin widget factories* can generate representatives for installed plugins on demand. Depending on the nature of each widgets, factories may or may not require to be fed parts of the underlying object model. For instance, the organizer widget's content provider will require the model for design tools and categories for displaying the plugin taxonomy. Again, implementations in SWT are provided for both factories.

A *cross-plugin integration* module is responsible for bridging the gap between the Kali-ma dashboard and the host environment. Its main functionality is to collect extensions contributed by each plugin and execute them on demand. Cross-plugin integration is also responsible for capturing user input and relinquishing control to other plugins when they are activated. This module has no ad-hoc NeOn Toolkit support, as it is able to manage extensions provided by NeOn Toolkit plugins or other Eclipse plugins just the same.

4.3 Reasoning component

In avoidance of the unwise practice of allowing the presentation component to handle the knowledge base straight away, Kali-ma implements a dedicated subsystem for extracting relevant knowledge. The ontological component, described in the next section, provides such knowledge that the reasoning component wraps into a Java object model, which can then be accessed from the Dashboard controller in the presentation subsystem. This lower-level component in the Kali-ma software architecture, and the intermediate layer in the whole infrastructure, provides a software counterpart to the ontological component .

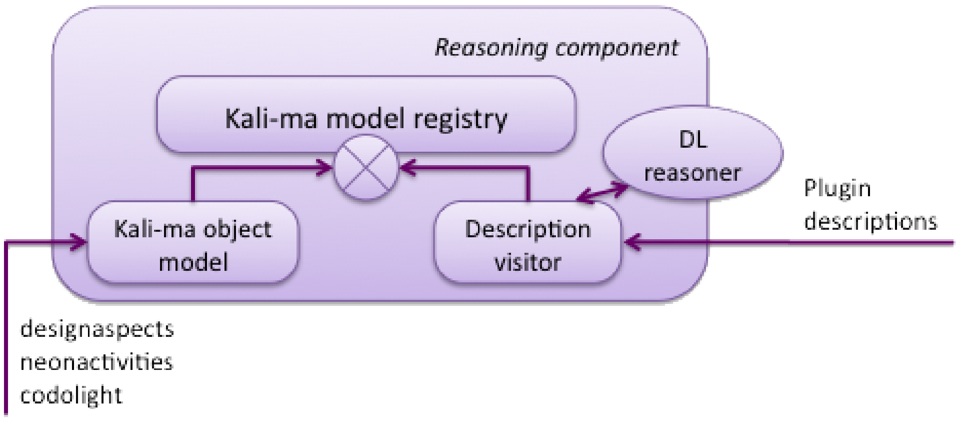


Figure 4.4: The reasoning component of Kali-ma.

Figure 4.4 illustrates the modules the *reasoning component* is comprised of. These are described as follows:

The *Kali-ma object model* represents parts of the C-ODO Light network, along with attached ontologies with additional categorization rules, in the form of Java types. This model includes interfaces for design tool, knowledge type, NeOn activity and design aspect OWL classes, and for generic annotated entities, whose RDFS label and comment annotations are deemed significant in the context of Kali-ma (i.e. they are presented to end users).

The *description visitor* is responsible for instantiating the object model mentioned above from the ABoxes supplied by C-ODO Light-based plugin descriptions and classification rule ontologies. This module includes monitorable operations for initializing OWL managers and DL reasoners (both supplied by external packages), loading them with fixed and user-defined ontologies (for details on the plugin description update mechanism, see Section 3.2) and querying them. As the following subsection will detail, this system does not necessarily query the DL reasoner at each Kali-ma run.

A *model registry* is where the instantiated object model is stored and kept track of. It stores wrapped OWL individuals and relationships between them, and allows changes to the model to be monitored through its own event system. The model registry is ephemeral and does not need to be serialized, as it can be completely rebuilt at runtime from the ontological component in reasonable time.

Through the components of this subsystem, Kali-ma becomes aware of what NeOn Toolkit plugins are known and/or installed in the running system, what are the relevant relations in ontology design and which of them are supported by collected plugins. The Kali-ma application logic has no prior knowledge of such relationships.

4.3.1 Tool description reasoning

There is no assurance that all the relevant facts concerning classification rules appear as assertions in plugin descriptions. As explained in section 2.4.2, the support for certain activities by the NeOn methodological guidelines and the implementation of certain design functionalities is usually coded as explicit RDF statements. This is not the case of common aspects in collaborative ontology design, as support for them often needs to be inferred from other asserted knowledge for a given tool, such as the knowledge types consumed and produced by it. In this case we need to apply description-logic reasoning procedures to the individuals of interest, in order to classify each tool by any criterion.

Querying a DL reasoner for class instances and property relations is a computationally intensive task, which would be long and tedious to perform on each Kali-ma run, especially for multithreaded reasoners running on top of hardware architectures with no Symmetric Multiprocessing (SMP). After this shortcoming emerged during the preliminary evaluation of the Kali-ma alpha (cf. 5), we implemented *inference caching* as a possible way to address the issue. The caching process exploits a capability of the OWL API to attach inferred axiom generators to a DL reasoner. An inferred axiom generator can be set to realize several types of axioms (e.g. type assertions, property relations, disjointness and equivalence) and save them to a new ontology. This ontology, which makes up the actual cache, can then be queried with much greater responsiveness through an ontology manager. By default, the Kali-ma description visitor will extract asserted axioms from the cache when it is available, and fall back to directly querying the reasoner only when the cache is missing or invalid and the user has set her preference *not* to cache inferences.

The caching process can take significantly longer than querying the reasoner in realtime, since it is a brute force approach that requires to classify the whole set of categorization rules, plugin descriptions and C-ODO Light modules, and realize several thousands of axioms: it has clocked up to 50 seconds on a 2.6GHz Intel Core 2 Duo running Mac OS 10.5. Even so, it bears the undeniable advantage of having to be performed only once until the cache is manually cleared or invalidated (e.g. by applying an update to plugin descriptions).

The classification algorithm is very simple and does not vary upon direct usage of a DL reasoner instead of an OWL Manager. The querying mechanism is made unambiguous through a single interface, called `CODOAccessor`, with methods for obtaining instances of a class, object property relations and datatype property values for an individual. Two separate implementations of this feature are used for querying either an ontology manager or a reasoner.

The sequence diagram in Figure 4.7 describes how the classification algorithm works. A pseudocode explanation is provided below.

```
1 CODOModelRegistry registry;
2 CODOAccessor accessor;
3 OWLClass cDesignTool = getDesignToolClass();
4 OWLObjectProperty p = getCriterionProperty();
5 foreach (DesignTool t :  accessor.getInstances(cDesignTool)) {
6     registry.addTool(t);
7     foreach (Category c :  accessor.getPropertyRelations(t, p))
8         registry.addcategory(c);
9         registry.addSupport(t, c);
10 }
```

This pseudocode synthesis was overly simplified for the sake of comprehensibility: in the actual implementation, there are no such functions as `getCriterionProperty()` or `getDesignToolClass()` as in lines 3 and 4, but it is useful to show that, unlike the variables in lines 1 and 2, these are not instantiated from scratch but are obtained from existing sources such as user preferences or the Kali-ma datamodel. In lines 5 through 10, we iterate on the whole set of design tools obtained using a `CODOAccessor` (whether it queries a reasoner or the cache ontology is irrelevant for understanding the algorithm). Each design tool is registered with the Kali-ma model registry (line 6). The `CODOAccessor` is then reused for obtaining the set of individuals that are related to each tool by the property that was chosen as a criterion (line 7). Each individual defines a category, which is added to the Kali-ma registry (line 8) and registered as being satisfied by that tool (line 9).

The sequence diagram shows that more routine operations, such as firing events on each addition, retrieving labels and comments for each individual, assessing NeOn Toolkit dependency for each tool and if so retrieving its unique identifier, are actually performed in the process.

4.4 Ontological component

The ontological component, encoded in its entirety in OWL, is at the lowest level of the stack. It is itself a layered subsystem, as the dependencies between its modules are acyclic. The component as a whole can be seen as a large networked ontology, although it is generally not sufficient to load one single root module in order to have the entire network at one's disposal. This limitation originates from the impossibility to determine where the ontologies declaring the facts, or ABoxes, for plugins are located. This is actually a positive feature, as it allows experts in ontology engineering to define categorization rules without an exhaustive knowledge of the tool space, while leaving plugin contributors the liberty to author descriptions for their tools and host them wherever they see fit. They do, however, need to update a RDF reference document to point to their ontologies, so that Kali-ma can locate them.

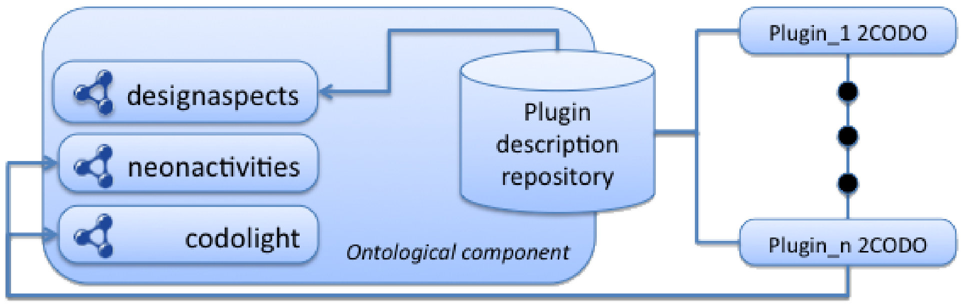


Figure 4.5: The ontological component of Kali-ma. Note that plugin descriptions are not known a priori, therefore they are displayed as external resources on the right side of the figure.

The diagram in Figure 4.5 shows how built-in knowledge (i.e. domain ontologies and classification rules) is separated from plugin-specific knowledge, which remains uncharted until it is aggregated at runtime in a local repository, which, as Section 3.2 has shown, is itself an ontology. Although theoretically the whole ontological component is replaceable, references to specific domain knowledge for collaborative ontology design has been hardcoded in this version of Kali-ma for the purposes of this NeOn task.

The invariable, built-in elements include three ontology networks which provide both the domain model (C-ODO Light, or shortly *codolight*[6]) and classification rules (*designaspects*[7] and *neonactivities*[8]). The codolight core module also provides supports for design functionalities, a third classification criterion for plugins.

Given the nature of this task and its focus on networked ontologies, the following subsections will provide an overview on each of the three fundamental elements that together form the ontological component.

4.4.1 Foundational/domain layer

We have chosen *codolight* as a model for ontology design. Its modularity and distinctive support for collaborative lifecycle management make it easily extensible with specialized classes, additional primitives and rules, without any need for tainting the whole model. What's more, it is aligned to several widely used ontologies describing the Semantic Web for computational and social interoperability. These include the Ontology Metadata Vocabulary (OMV), Description Of A Project (DOAP), the Software Ontology Model, Sweet Tools and the Protégé workflow ontology.

The rounded rectangles in Figure 4.6 represent all the modules that make up the codolight network, sans specific ontology design patterns. While the reader can refer to deliverable 2.1.2 [GP09] for details on this model, it is useful to briefly recall the rationale behind each module that is linked to the core ontology *cod-kernel*:

[6]http://www.ontologydesignpatterns.org/cpont/codo/codolight.owl
[7]http://www.ontologydesignpatterns.org/cpont/codo/designaspects.owl
[8]http://www.ontologydesignpatterns.org/cpont/codo/neonactivities.owl

- *coddata* represents the data, or knowledge resources that are typically involved in ontology projects: ontologies and their entities, mappings, modules, non-ontological resources, etc.
- *codprojects* represents the classes of design project-related entities and their relations.
- *codworkflows* represents workflows from within ontology projects: collaborative workflows, accountable agents, etc.
- *codarg* represents the classes of argumentation entities and their relations.
- *codsolutions* represents the classes of design solution-related entities and their relations.
- *codtools* represents the classes of ontology design tools, their capabilities, input and output types, etc.
- *codinteraction* contains some sample classes and properties to talk about interface objects, with exemplar instances.
- *codinterfaces* covers interaction patterns, with the most common example instances.

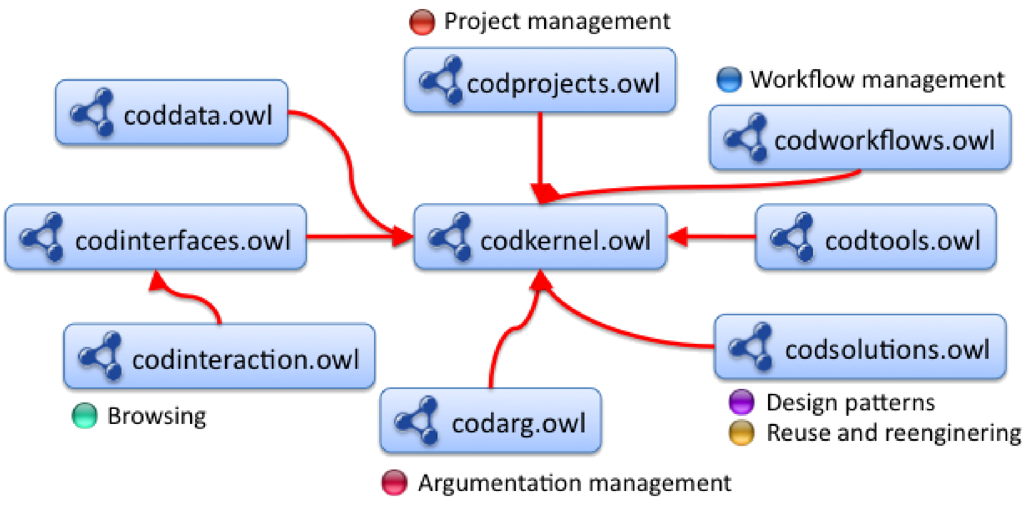


Figure 4.6: The *codolight* network architecture, collectively named *corolla*. Placeholders for collaborative design aspects are placed next to the specific modules they map to.

We will now show how our proposal for a set of classification rules maps to the codolight network.

4.4.2 Categorization rules

In Section 2.4.2, three classification criteria were described as being supported by Kali-ma when providing a taxonomy of tools in the Organizer widget: implementation of *design functionalities*, support for NeOn methodology *activities* and coverage of collaborative *design aspects*. While applications of the first two criteria can be usually extracted without resorting to reasoning tasks, inference will be required in order to determine which design aspects are covered by a plugin.

designaspects is the ontology that defines classes for these common facets of ontology design, as well as the axioms for determining how the rules defined in codolight can be pulled in order to match a tool with the design aspects it is able to cover. This document proposes a fixed, finite set of six design aspects whose strong point lies in the fact that all of them encompass functionalities and tasks which can be shared and performed in collaboration. For example, design patterns can be used for authoring ontology networks as lightweight modules that can be divided among participants; ontology development projects can be scheduled collaboratively and shared among partners, while argumentation spans across inherently collaborative tasks.

Design Aspects		
Label	**Associated DesignTool subclass**	**C-ODO Light module**
Project management	`ProjectManagementTool`	`codprojects`
Workflow management	`WorkflowManagementTool`	`codworkflows`
Argumentation management	`DiscussionEvaluationTool`	`codarg`
Design patterns	`SolutionFindingTool`	`codsolutions`
Reuse and reengineering	`ReuseReengineeringTool`	`codsolutions`
Browsing	`BrowsingEditingTool`	`codinteraction`

Table 4.1: Aspects of collaborative ontology design and their mappings to defined classes and codolight modules.

Even user interaction, e.g. with a shared whiteboard, can be interpreted from a collaborative viewpoint, hence the browsing and navigation aspect. Each aspect is associated with a defined class that is a subclass of DesignTool and has equivalent restrictions on rules involving the corresponding design aspects and other codolight entities. Table 4.1 summarizes all six design aspects, along with the corresponding defined class and the codolight module that each aspect is tightly coupled with.

Design aspects allow to classify design tools into categories that are defined by the type of knowledge that is consumed in input, or produced in output. For example, the class of tools that allow to perform some reuse or reengineering of existing knowledge resources are defined as follows (in Manchester OWL Syntax):

```
Class:  ReuseReengineeringTool
  EquivalentTo:
    (hasAspect value ReuseReengineering),
    (codtools:implements some
        (specialization:specializes value ReuseReengineering)),
    (codtools:hasInputType some ({coddata:DataStructureKType ,
        coddata:KOSElementKType , coddata:KOSKType ,
        coddata:LinguisticKType , coddata:OntologyAxiomKType ,
        coddata:NetworkOfOntologiesKType ,
        coddata:NetworkedOntologyKType , coddata:OntologyKType ,
        coddata:OntologyElementKType , coddata:OntologyModuleKType
        coddata:OntologyMappingKType
    and
    (codtools:hasOutputType some
        ({coddata:NetworkOfOntologiesKType ,
        coddata:NetworkedOntologyKType , coddata:OntologyKType ,
        coddata:OntologyAxiomKType , coddata:OntologyElementKType ,
        coddata:OntologyMappingKType , coddata:OntologyModuleKType}))
```

By the formula shown above, ‘something’ can be classified as a tool for reuse or reengineering *iff* at least one of the following holds:

1. it is explicitly said to have the `ReuseReengineering` aspect as a value;
2. it implements a functionality that on its turn is a specialization of the `ReuseReengineering` aspect (which, to mention once more, is also a design functionality);
3. it has one or more values from a given list (data structures, ontologies, axioms, knowledge organization

systems such as thesauri and classification schemes, etc.) as input knowledge types, and one or more values from another given list (networked ontologies, entities, axioms, modules, mappings etc.) as output knowledge types.

The most appropriate and conceptually significant definition is the third one: definition (1) is there to associate the aspect with the tool in order to infer it as a value, and is not intended to be directly attached to the tool description; definition (2) is provided in case the aspect is conveyed by a functionality that has been already associated with `ReuseReengineering` (which is unlikely unless the *designaspects* ontology is explicitly imported in the plugin description). By far, the most orthodox means to convey knowledge about the design aspects supported by a plugin is to supply as much codolight-based functional information as possible and let a reasoner infer if given information is sufficient for determining how that plugin is to be classified. For this reason, the *codomatic* service only exposes input methods for providing definitions of type (3).

4.4.3 Tool space population

So far we have concentrated on the invariant, built-in components of the ontological subsystem. These are under complete control of the main contributor (the Kali-ma provider in the case of this task) and are known a priori, but there are no such guarantees in the kingdom of ABoxes. Facts concerning each NeOn Toolkit plugin in the tool space are for plugin providers to state in their ontologies, though they may reference other tools when including statements such as depending on another tool, or implementing the functionalities of a Web Service within the NeOn Toolkit.

OWL descriptions of NTK plugins are plugin-specific modules, each describing what types of task a certain plugin can help accomplish, what types of knowledge representation it can handle, and so on. They are based on codolight and are used by the Kali-ma reasoning component for classifying each plugin with respect to any of the three available categorization criteria.

A dedicated codolight-based description for the NeOn Toolkit itself is available and can be automatically imported when creating plugin descriptions with the *Codomatic* generator. This ontology exposes the NeOn Toolkit as a design tool, which other plugins can depend upon, and if so, the same ontology provides a mechanism for declaring the unique identifier for that plugin in the NTK environment, by means of the `hasPluginId` datatype property. Kali-ma exploits this relation in order to determine which of the known NTK plugins are actually installed on a running instance of the platform.

For completeness, ontology design tools need not be NTK plugins in order to benefit from the reasoning capabilities of Kali-ma. Although by default these non-NTK tools are filtered out, they can be set to be classified along with NTK plugins themselves.

Although ontology engineering experts are free to include more dependencies, e.g. on *designaspects*, other tool descriptions or ontology design patterns [PGD$^+$08], in order to refine and enrich their descriptions after generating them, the following ontology imports can be added by default:

1. *codolight.owl*, the transitive closure of all the C-ODO Light modules;
2. *ntk2codo.owl*, a description for the NeOn Toolkit itself[9], if the described tool is a NeOn Toolkit plugin;
3. *neonactivities.owl*, if the tool is said to support at least one activity in the NeOn methodology;
4. any other tool description that declares design functionalities that are either extended or directly implemented by the tool that is being described.

To clarify with an example, the following text box shows the Manchester OWL Syntax representation of an individual representing the Watson plugin for the NeOn Toolkit [dSM08]. For simplicity, all namespace bindings and entity declarations were trimmed and are left to the reader's intuition and the guided interpretation that follows later in this section.

[9]http://www.ontologydesignpatterns.org/cpont/codo/tooldesc/ntk2codo.owl

```
Individual:  WatsonPlugin

Annotations:
rdfs:label "Watson NTK plugin"@en
rdfs:comment "A plugin for discovering, integrating and reusing
knowledge from online ontologies when building an ontology with the
NeOn Toolkit."@en

Types:
codkernel:DesignTool

Facts:
codtools:hasOutputType coddata:OntologyAxiomKType,
codtools:hasOutputType coddata:OntologyElementKType,
codtools:hasInputType coddata:OntologyElementKType,
neonactivities:supportsActivity neonactivities:OntologyEnrichment,
neonactivities:supportsActivity neonactivities:OntologyReuse,
codtools:implements watson2codo:OntologyElementDiscovery,
codtools:implements watson2codo:OntologyStatementReuse,
codtools:dependsOn ntk2codo:NeOnToolkit,
ntk2codo:hasPluginId "org.neontoolkit.watsonplugin" xsd:string
```

In the example above, the `DesignTool` type specification is necessary for identifying the Watson plugin as an ontology design tool. Both RDFS annotations `rdfs:label` and `rdfs:comment` are used for providing respectively a more user-friendly naming in the Organizer widget and a synopsis of the tool in the Helper and any plugin widget that is generated.

The facts stated on properties `dependsOn` and `hasPluginId` identify this tool as a NTK plugin. They are not strictly necessary for classifying the tool per se, since it would appear in the taxonomy anyhow. However, without these relations it would not be possible to generate a widget for Watson and launch the plugin from it.

The remaining facts about the `WatsonPlugin` individual allow Kali-ma to classify the tool by all available criteria. The `supportsActivity` statements make sure that Watson will be available for two categories in a NeOn methodology taxonomy. The `implements` statements provide two new design functionalities asserted in the plugin description itself (note that `OntologyStatementReuse` was added as a functionality because it was not yet available as a NeOn activity by the methodology glossary the OWL vocabulary is based on). Finally, the `hasInputType` and `hasOutputType` statements allow Watson to be classified as a Reuse and Reengineering tool. This happens because `WatsonPlugin` has a value: `OntologyElementKType` that is in the list expected for the property `hasInputType` in the definition of `ReuseReengineeringTool`, as well as two values: `OntologyAxiomKType` and `OntologyElementKType`, that are in the list expected for `hasOutputType`. At least one value for each property is expected by that definition. Because all design aspects are specialized design functionalities, the `ReuseReengineering` design aspect will also appear in the taxonomy for custom functionalities.

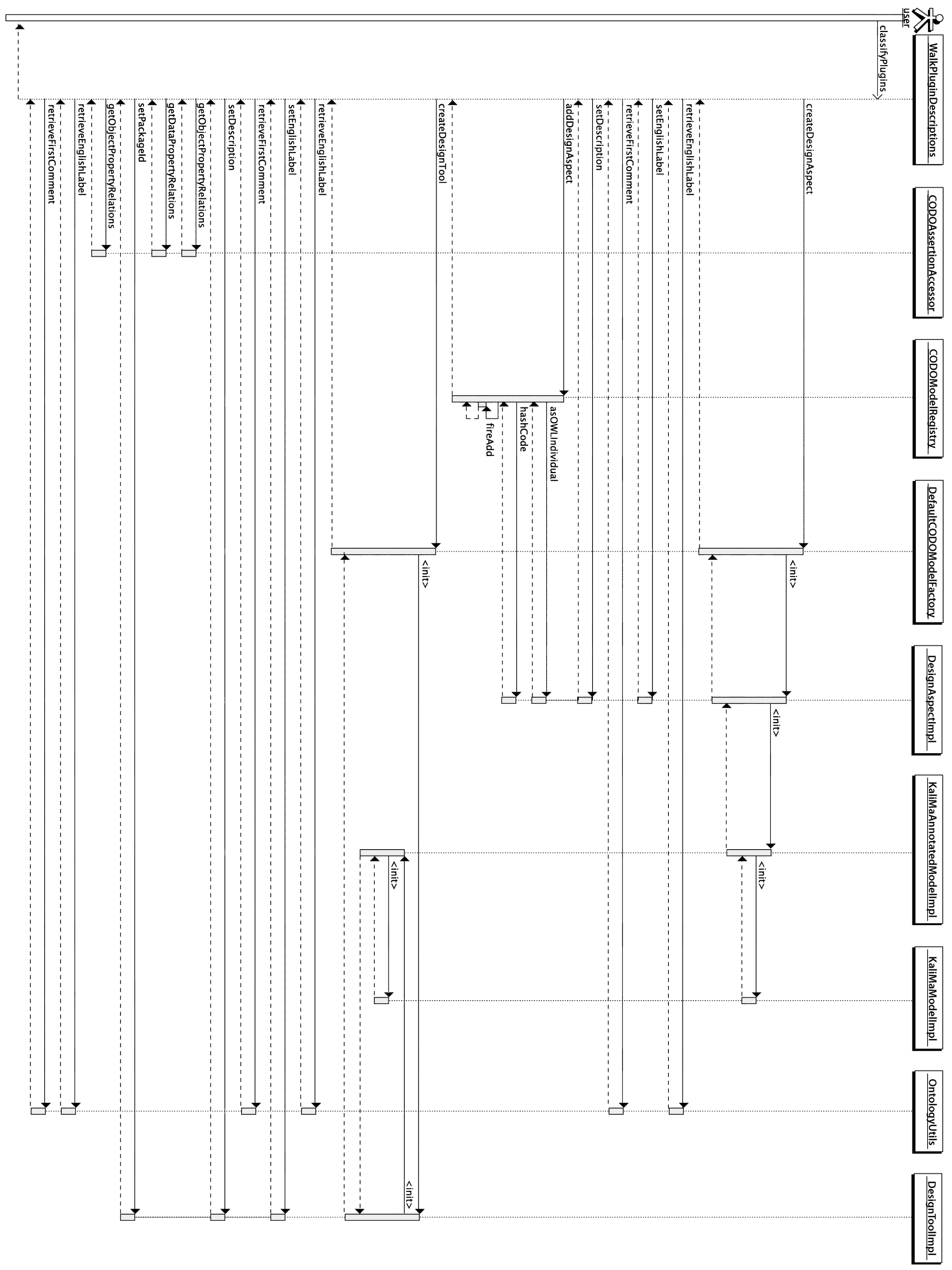


Figure 4.7: Sequence diagram of the tool classification process.

Chapter 5

User evaluation

Because the first public version of Kali-ma comes with special focus on user interaction issues, we felt inappropriate to defer the whole usability evaluation of the tool to a beta version whose functional requirements were already implemented and consolidated. In order to assess how positively our proposed approach would be perceived, with respect to user needs emerged in [DMB$^+$06, DMGP$^+$06, DBMGP08], the need for early user feedback was paramount.

A preliminary evaluation on an alpha version of the tool provided useful insights as to the directions to be followed in improving the implementation choices. The scenario for this user-based evaluation was the FAO networked ontology training course held in Rome during June 2009. The user base involved in this session consisted of 12 representatives of a community of practitioners, with variable experience and knowledgeability in the fields of knowledge engineering. All the practitioners involved were employed in organizations that shared a common problem of managing the lifecycle of ontologies, as all of these organizations were approaching the use of Semantic Web technologies.

The ontology training course lasted a total of three days, where the principles and practices of the NeOn methodology for networked ontology engineering were illustrated and applied hands-on by using the NeOn Toolkit and its plugins. Day two included a hands-on session with a preliminary alpha version of the Kali-ma plugin. There, users were required to locate and switch across plugins based on the engineering tasks they were asked to perform. At the end of the Kali-ma hands-on sessions, each user was given a five-level Likert-scaled questionnaire[1] [Ren32] consisting of 47 items spanning from user background to subjective judgement on the user-friendliness of the tool. Out of these 47 items, 12 aimed at evaluating usability and friendliness of Kali-ma interaction, while the others aimed at understanding the users' background, their familiarity and personal preferences with regard to user interface types, and at collecting feedback and suggestions for improvement.

On the basis of their answers to the user background-related questions, we were able to classify these 12 practitioners into three categories: software developers (2), software developers with some knowledge on web ontologies (5), and information management officers (5). As for the analysis of usability and user-firiendliness of the Kali-ma tool, our evaluation proceeded as follows.

The 12 items related to usability and friendliness of Kali-ma interaction were willfully polarized in order to coerce them to focus and maintain attention on the statements provided and the feedback they were giving. That is, agreement with some statements would be interpreted as positive feedback on usability, while agreement with some others indicated negative feedback. For this reason, we normalized the polarity of items in order to make responses comparable with each other. Then we mapped the five Likert levels to three scores: *negative* (strongly disagree and disagree), *neutral* (undecided), and *positive* (agree and strongly disagree), in order to have a brief but comprehensive indication of the results. For each of these score and each questionnaire item, we computed the average number of 'checks', whereby a check indicates that the user has selected a checkbox of a Likert level that maps to a certain score. Finally, we computed the average value

[1]The format of a typical five-level Likert item as the one used for this evaluation is: 1. Strongly disagree, 2. Disagree, 3. Neither agree nor disagree, 4. Agree, 5. Strongly agree.

of these checks in the case the distribution of scores was homogeneous. From such measurements we observed that the actual distribution deviates from the homogeneous distribution by 14,8% towards the *positive* score.

The most relevant interaction-related issues that emerged from user responses, as well as specific comments arbitrarily added by practitioners, were the following:

1. *Lengthy tool classification.* Users observed that the operation for bootstrapping the Kali-ma dashboard, though displaying adequate feedback on the steps being performed and their progress, proved to be annoyingly long on test hardware. This symptom was noticed when the Kali-ma dashboard was launched for the first time on each new run of the NeOn Toolkit, while switching back and forth between the dashboard and the NeOn Toolkit workbench proved to be almost instantaneous.

2. *Dashboard overcrowding.* Users openly claimed that, even when they might have wanted to pick a selection of plugin widgets for covering an entire phase of an ontology development plan, they did not necessarily want to have all of these widgets simultaneously displayed on screen. At the same time, they still intended to keep track of their original selection, i.e. they did not want to be forced to close a plugin widget to solve the issue, as if that widget had never been opened.

3. *Lack of user guidance.* Some users complained about not knowing how to proceed after the Kali-ma dashboard was launched, and not finding any interface objects that could provide useful hints to guide them through the process.

The development phase that followed this session concentrated mainly on taking action upon the three items mentioned above. Issue (1) was a consequence of the high computational cost of individual reasoning operations on each design tool, which are multithreaded and scale dramatically on multi-core CPUs. To address this issue, the reasoning subsystem was reengineered in order to support caching of all the inferred axioms needed in order to classify a tool (cf. Section 4.3). This feature has brought a major cutdown on performance bottlenecks when the system is running at top revs, at the cost of a longer initial caching operation which does not need to be repeated until the cache is cleared, workspaces are switched, or plugin descriptions are updated. Issue (2) was addressed with the inclusion of the Dock as a native widget (cf. Section 2.4.4), which was not present at all in the version under test. Docking one or more widgets allows users to clean up their dashboard with no significant impact on the profiles being stored for reuse. Finally, issue (3) was addressed with another new native widget providing realtime help (cf. Section 2.4.5).

From a more in-depth analysis of the scores and suggestions made by taking into account the single groups, we can read such results in an even more positive way than hinted by the figures resulting from the quantitative analysis mentioned a few paragraphs above. As a matter of fact, the most negative scores concentrated on the possible improvement of the Kali-ma interaction compared to the basic toolkit. However, it has to be observed that negative scores came mainly from software engineers, neutral scores were mainly from the second group, while the group of information management officers gave mainly positive scores. Our interpretation of this further observation is grounded on the apparently conflicting interaction modes provided by the Kali-ma dashboard on one side, and by the Eclipse workbench on the other side. It was therefore predictable that software engineers, who are typically more familiar with the regular interaction methods proposed by the Eclipse platform, would be more deeply affected by a shift in the interaction paradigm. Finally, most suggestions were related to improving interaction by adding tooltips and some nicer graphic solutions to the codo organizer, aspects that we are including in the release to follow.

Chapter 6

Ongoing work

The 1.0 version of the Kali-ma plugin was mainly focused on enhancing user interaction with the NeOn Toolkit platform and providing an interaction mode with support for an alternative point of view on the platform, whereby it can be regarded as a functionality provider, as opposed to interface-centered Eclipse workbench features.

Most of the ongoing development directions for the near future are all directed towards more extensive coverage of C-ODO Light features, thus bringing in broader domain model support in general. A consistent codebase already exists and is being improved in order to furnish the Kali-ma platform with assets that will purvey stronger support for collaborative management of ontology development projects, along with continuous enhancement of the interaction experience.

The closest item in the development timeline is the release of a Java API that will allow other plugin providers to contribute to dashboard management programmatically, beyond the means offered by the ontological descriptions of their tools. The Kali-ma API will provide methods for launching a dashboard from scratch, with a set of plugin widgets already opened and an optional profile name associated to it. Such a functional requirement was initially recognized as a trade-off to allow the Kali-ma and gOntt plugins to interoperate, without having to establish dependencies on each other or implement ad-hoc integration that would make no sense for other plugins to take advantage of. In the case of gOntt, it will be possible to pick a phase, process or activity scheduled in an open plan, and trigger a Kali-ma dashboard with a selection of plugins that support these activities. In order to avoid disrupting the user's view on the tool space, Kali-ma will organize known tools according to the NeOn methodology criterion that the user is assumed to be already accustomed to. The role of Kali-ma is then to serve the execution of a gOntt plan at a given point or interval in time.

The ways in which the Kali-ma API will aid plugin interoperability will be diverse. A prominent feature, for which proof-of-concept coding has already been done, is to allow plugins to interoperate with one another and with the NeOn Toolkit core, by orchestrating and executing combined operations and define *pipelines* between widgets. The API will expose a set of extension points that plugins can extend in order to offer some of the functionalities they have already been designed for as services within the Kali-ma dashboard. In this way, it is possible for some plugin operations to be performed directly from within the plugin widgets themselves. Simple operations include displaying the results of a Watson search, or the axioms that are making an ontology inconsistent in RaDON. Combined operations may be configured by matching widgets based on compatibility between output and input knowledge types handled by plugins. This knowledge is explicitly declared in semantic plugin descriptions, and it will be backed by low-level support by means of Java annotations in the implemented extensions of the Kali-ma API. An example of a combined operation is to rank Watson results in realtime as the user interacts with the SearchPoint plugin. To further support pipelines, Kali-ma will provide an additional native widget for searching ontologies and entities within one or more ontology projects, and present them to the system for third-party plugins to consume.

As for the enhancement and maintenance of the features presented in this version, our top priority is the support for additional access methods provided by plugins:

Export wizards are paged dialogs that guide the user through a process of converting and/or storing a given resource. An example of export wizard in NeOn technologies is the *OWLDoc* plugin for the automatic generation of ontology documentation files.

Actions correspond to procedures that can be triggered in various ways, such as toolbar items or application menus.

Cheat sheets are quick reference help pages, usually appearing within an Eclipse view, with optional support for interactivity. Cheat sheets are especially useful for those plugins that do not come with a wizard or a perspective.

Not every implementation of these extension points is context-free: for example, export wizards usually require the user to select the resources to export. This can either be achieved via the wizard itself or by selecting the resources of interest within a view, then launching the export command from there. Kali-ma cannot indefinitely support all export wizards of the latter category, simply because it is not possible to foresee selection mechanisms for every possible resource type without mimicking their original presentation modes. However, it will take advantage of special methods provided for selecting ontologies or OWL entities for supporting plugin pipelines.

On the interaction side, development of an alternative view on the C-ODO organizer is also underway. In addition to the classical tree view presented in version 1.0, users will have an option for having the same plugin classification presented in a convenient graphical form. This will be based on additional graphical editing libraries available for the extended version of the NeOn Toolkit, and optionally installable on its basic version. Other improvements will leverage the potential offered by specific parts of the C-ODO Light network, such as the user interface module for managing plugin widget appearance, and the project module for allowing users to aggregate and manage a history of the metadata generated by the NeOn Toolkit core and its plugins.

Chapter 7

Conclusion

We have presented a new piece of NeOn technology that can be put to good use in coordinating a multitude of other software contributions released within and beyond the scope of the project. Kali-ma, the ‘C-ODO plugin’, is a semantic tool itself, in that it leverages a set of reasonably simple assertions on the space of NeOn Toolkit plugins, in order to arrange them in a homogeneous fashion, according to a principles of lifecycle management of networked ontologies. Therefore, this approach does not favor plugins that offer several interaction modes, which is praxis in the Eclipse platform, but only those that serve a great number of purposes in ontology engineering. In describing the plugin itself, we have provided examples based on the Pharmaceutical case study for this project, which have shown how project managers and ontology engineers alike may benefit from our proposed approach in collaborative contexts.

This tool also yielded a byproduct that proved a beneficial side effect of the NeOn Toolkit platform, i.e. that the interaction methods implemented in the Eclipse workbench are not restrictive for developers and user interface specialists. Kali-ma shows how the NeOn Toolkit has a virtually limitless potential for implementing a variety of interaction paradigms.

Initial end-user experimentation has shown how Semantic Web specialists, even more than traditional software engineers, are regarding our interactive approach with favor. Following up on these experiments, users have offered valuable advice, which was taken into account in the latest development phase, for giving our contribution the potential for a solid alternative to the standard interaction mechanisms provided by the Toolkit.

Bibliography

[BCT07] Karin Koogan Breitman, Marco Antonio Casanova, and Walter Truszkowski. *Semantic Desktop*, pages 229–239. Springer London, 2007.

[BHL08] Peter Buxmann, Thomas Hess, and Sonja Lehmann. Software as a service. *Wirtschaftsinformatik*, 50(6):500–503, 2008.

[CL08] German Herrero Carcel and Tomas Pariente Lobo. Revision of ontologies for Semantic Nomenclature: pharmaceutical networked ontologies. Deliverable D8.3.2, NeOn project, 2008.

[CPG05] Adam Cheyer, Jack Park, and Richard Giuli. IRIS: Integrate. Relate. Infer. Share. In Stefan Decker, Jack Park, Dennis Quan, and Leo Sauermann, editors, *Proc. of Semantic Desktop Workshop at the ISWC, Galway, Ireland, November 6*, volume 175, November 2005.

[DBMGP08] Martin Dzbor, Carlos Buil Aranda, Enrico Motta, and Jose Manuel Gomez-Perez. Analysis of user needs, behaviours and requirements on ontology engineering tools. Deliverable D4.1.2, NeOn project, 2008.

[DCW08] Tharam S. Dillon, Elizabeth Chang, and Pornpit Wongthongtham. Ontology-based software engineering - software engineering 2.0. In *Australian Software Engineering Conference*, pages 13–23. IEEE Computer Society, 2008.

[DMB$^+$06] Martin Dzbor, Enrico Motta, Carlos Buil, Jose M. Gomez, Olaf Görlitz, and Holger Lewen. Developing ontologies in OWL: an observational study. In Bernardo Cuenca Grau, Pascal Hitzler, Conor Shankey, and Evan Wallace, editors, *OWLED*, volume 216 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[DMGP$^+$06] Martin Dzbor, Enrico Motta, Jose Manuel Gomez-Perez, Carlos Buil Aranda, Klaas Dellschaft, Olaf Goerlitz, and Holger Lewen. Analysis of user needs, behaviours and requirements with respect to user interfaces for ontology engineering. Deliverable D4.1.1, NeOn project, 2006.

[dSM08] M. d'Aquin, M. Sabou, and E. Motta. Reusing knowledge from the semantic web with the watson plugin. 2008.

[GHM$^+$07] Tudor Groza, Siegfried Handschuh, Knud Moeller, Gunnar Grimnes, Leo Sauermann, Enrico Minack, Cedric Mesnage, Mehdi Jazayeri, Gerald Reif, and Rosa Gudjonsdottir. The NEPOMUK Project - on the way to the Social Semantic Desktop. In Tassilo Pellegrini and Sebastian Schaffert, editors, *Proceedings of I-Semantics' 07*, pages pp. 201–211. JUCS, 2007.

[GP09] Aldo Gangemi and Valentina Presutti. The collaborative ontology design ontology (v2). Deliverable D2.1.2, NeOn project, 2009.

[GPBC$^+$07] Jose Manuel Gomez-Perez, Carlos Buil, German Herrero Carcel, Tomas Pariente Lobo, Angel Baena, Joan Candini, and Juan Carlos Dalmacio. Ontologies for the pharmaceutical case studies. Deliverable D8.3.1, NeOn project, 2007.

[GPSFV09] Asuncion Gomez-Perez, Mari Carmen Suarez-Figueroa, and Martin Vigo. gOntt: a tool for scheduling ontology development projects. In *8th International Semantic Web Conference (ISWC2009)*, October 2009.

[HBN07] Matthew Horridge, Sean Bechhofer, and Olaf Noppens. Igniting the OWL 1.1 touch paper: The OWL API. In *OWL: Experiences and Directions Third International Workshop (OWLED 07)*, Innsbruck, Austria, June 2007.

[PBKL06] Emmanuel Pietriga, Christian Bizer, David Karger, and Ryan Lee. Fresnel: A Browser-Independent presentation vocabulary for RDF. In *The Semantic Web - ISWC 2006*, volume 4273 of *Lecture Notes in Computer Science*, pages 158–171. Springer-Verlag, 2006.

[PGD$^+$08] Valentina Presutti, Aldo Gangemi, Stefano David, Guadalupe Aguado de Cea, Mari Carmen Suarez-Figueroa, Elena Montiel-Ponsoda, and Maria Poveda. A library of ontology design patterns: reusable solutions for collaborative design of networked ontologies. Deliverable D2.5.1, NeOn project, 2008.

[PMP$^+$09] Valentina Presutti, Dunja Mladenic, Raul Palma, Klaas Dellschaft, Alessandro Adamou, Enrico Daga, Holger Lewen, Michael Erdmann, and Anne Becker. Practical methods to support collaborative ontology design. Deliverable D2.3.2, NeOn project, 2009.

[QHK03] Dennis Quan, David Huynh, and David Karger. Haystack: A platform for authoring end user semantic web applications. In *The SemanticWeb - ISWC 2003*, pages 738–753. Springer Berlin / Heidelberg, 2003.

[Ree79] Trygve Reenskaug. Models - Views - Controllers. Technical report, Technical Note, Xerox Parc, 1979.

[Ren32] Rensis Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 140:1Ð55, 1932.

[SBD05] Leo Sauermann, Ansgar Bernardi, and Andreas Dengel. Overview and Outlook on the Semantic Desktop. In Stefan Decker, Jack Park, Dennis Quan, and Leo Sauermann, editors, *Proceedings of the 1st Workshop on The Semantic Desktop at the ISWC 2005 Conference*, volume 175 of *CEUR Workshop Proceedings*, pages 1–19. CEUR-WS, November 2005.

[SFBd$^+$09] Mari Carmen Suarez-Figueroa, Eva Blomqvist, Mathieu d'Aquin, Mauricio Espinoza, Asuncion Gomez-Perez, Holger Lewen, Igor Mozetic, Raul Palma, Maria Poveda, Margherita Sini, Boris Villazon-Terrazas, Fouad Zablith, and Martin Dzbor. Revision and extension of the NeOn Methodology for building contextualized ontology networks. Deliverable D5.4.2, NeOn project, 2009.

[SFdCB$^+$08] Mari Carmen Suarez-Figueroa, Guadalupe Aguado de Cea, Carlos Buil, Klaas Dellschaft, Mariano Fernandez-Lapez, Andres Garcia, Asuncion Gomez-Perez, German Herrero, Elena Montiel-Ponsoda, Marta Sabou, Boris Villazon-Terrazas, and Zheng Yufei. Neon methodology for building contextualized ontology networks. Deliverable D5.4.1, NeOn project, 2008.

[SGR08] Amit P. Sheth, Karthik Gomadam, and Ajith Ranabahu. Semantics enhanced services: METEOR-S, SAWSDL and SA-REST. *IEEE Data Eng. Bull.*, 31(3):8–12, 2008.

[SNTM08] Abraham Sebastian, Natalya Fridman Noy, Tania Tudorache, and Mark A. Musen. A generic ontology for collaborative ontology-development workflows. In Aldo Gangemi and Jérôme Euzenat, editors, *EKAW*, volume 5268 of *Lecture Notes in Computer Science*, pages 318–328. Springer, 2008.

[SPG$^+$07] Evren Sirin, Bijan Parsia, Bernardo Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.

[SPSP$^+$08] Fernando Silva Parreiras, Steffen Staab, Jeff. Z. Pan, Krzysztof Miksa, Harald K§hn, Srdjan Zivkovic, Stefano Tinella, Uwe Assmann, and Jakob Henriksson. Semantics for software modeling. In Phillip Sheu, Heather Yu, C. V. Ramamoorthy, Arvind K. Joshi, and Lotfi A. Zadeh, editors, *Semantic Computing*, page 25. IEEE Press/Wiley, 2008.

[SS04] Leo Sauermann and Sven Schwarz. Introducing the Gnowsis Semantic Desktop. In *Poster at the International Semantic Web Conference ISWC 2004*, 2004.