

Chapter 15

Customizing Your Interaction with Kali-ma

Alessandro Adamou and Valentina Presutti

Abstract This chapter 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 activities that follow 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 semiautomatic generation of C-ODO Light-based plugin descriptions.

A. Adamou (✉)

Semantic Technology Lab, Institute of Cognitive Sciences and Technologies (National Research Council – CNR), Via Nomentana 56, 00161 Rome, Italy

Department of Computer Science, Alma Mater Studiorum Università di Bologna, Mura Anteo Zamboni 7, 40126 Bologna, Italy

e-mail: alessandro.adamou@istc.cnr.it; adamou@cs.unibo.it

V. Presutti

Semantic Technology Lab, Institute of Cognitive Sciences and Technologies (National Research Council – CNR), Via Nomentana 56, 00161 Rome, Italy

e-mail: valentina.presutti@cnr.it

15.1 Introduction

Being an Eclipse RCP¹-based ontology engineering platform, the NeOn Toolkit has an openly extensible feature set. Third parties may add custom functionalities in the form of software modules called *plugins*. Although the NeOn Toolkit provides its own set of specific extension points for manipulating ontology project hierarchies, most of those commonly provided by the Eclipse platform are supported. As with other Eclipse platforms, NeOn Toolkit plugins are maintained in dedicated repositories, called *update sites*. A pointer to one such update site, which was described in Chap. 13, is hardcoded in the core toolkit, but since the platform is open, anyone may set up their own update sites and use them as sources for additional plugins.

The RCP takes the burden of integrating plugins from the user interface perspective, e.g., by adding menus and toolbar buttons, populating the lists of views and perspectives, or adding new types of items that can be created via wizards. However, it is usually up to the developer to facilitate *conceptual* integration of her plugin by characterizing the goals of its features. The ways for doing so in the Eclipse platforms are little more than giving appropriate names and assigning categories to the UI contributions provided by their plugins. Performing this task can be tricky if the platform is supported by a large community, whose each member develops a plugin not knowing what others are doing. As a result, developers may arbitrarily add whatever categories, items, and labels they see fit for their plugins regardless of the rest. For example, two developers can create multiple categories for views, give them unique identifiers but label them both as *Visualization* independently on one another. As a result, end users will see two *Visualization* categories grouping different UI elements. Again, one developer could name a category after the plugin providing the corresponding UI elements, while another could name it after an arbitrarily named task supported by her plugin. In other words, when a contributor develops a plugin for the NeOn Toolkit, as well as for most plugin-based frameworks, she projects her own interpretation of the implicit metamodel of the user interface. Moreover, the uncontrolled proliferation of features (Damian and Chisan 2006) can clutter the user interface, e.g., if each plugin adds its own menu simply because no common agreement is reached as to which menus should be used for adding entries or submenus.

An instance of the scenarios described above is shown in Fig. 15.1. Here, a NeOn Toolkit user who has installed a large number of plugins from different sources is presented with this two-level tree list upon selecting the *Show View* menu entry. The names of views and the categories grouping them are widely varying in this example: some views, such as *Evolva Main View*, *OntoConto*, and *SearchPoint*, are named after the plugin that provides them; others, such as *Partitioning*, *Relationship*

¹*Eclipse Rich Client Platform*, the software development toolkit originally written for the *Eclipse* integrated development environment (IDE).

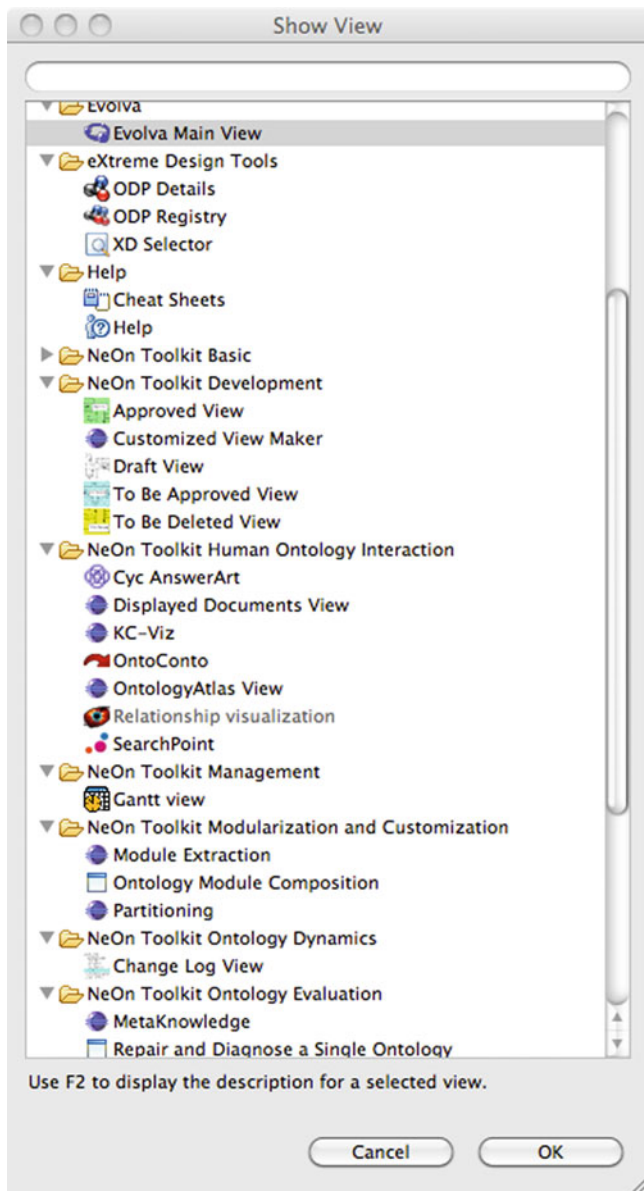


Fig. 15.1 An example of a view selection menu in a very crowded NeOn Toolkit

Visualization, and Repair and Diagnose a Single Ontology, are named after the functionalities they provide; others, such as Gantt, refer to the structure of the view itself. Since there are no set rules about the naming of categories and interface components, there is no right or wrong with any of these rationales. However,

because they come from different interpretations that each developer had of the user interface model, the overall picture may appear confused and cluttered. It is a goal of *Kali-ma* to try and bring some order into this confusion.

Kali-ma is a NeOn Toolkit plugin that aids developers and end users alike in creating a conceptually harmonized view on other known NeOn Toolkit plugins (and, more in general, tools that support the life cycle of ontologies). *Kali-ma* implements a user interface paradigm alternative to the Eclipse Workbench (and which can be switched with the latter in real time). This interface groups all UI contributions and access methods by the plugins issuing them and, with relatively little development effort, the plugins themselves by categories best representing the goals they are targeted at. It also adds a set of collaboration-oriented functionalities for end users, such as a metadata search feature, a whiteboard for executing dynamic plugin assemblies, and dedicated real-time chat support for ontology projects.

The remainder of this chapter provides an insight on the plugin as a whole, its functionalities, and the rationale behind them. Section 15.2 guides the reader through the plugin features and is structured so that the reader can concentrate on the section for end users (Sect. 15.2.1) or the one for developers (Sect. 15.2.2), depending on the reader's role. Developers are however advised to read both subsections in order to gain an understanding on the effects of their *Kali-ma* extensions on the interaction experience. Section 15.2.3 focuses on the underlying software architecture and how it combines standard components in Java with others in OWL (namely an extended version of the C-ODO Light ontology described in Chap. 4), thus being of interest for software engineers and ontology specialists alike.

15.2 *Kali-ma* Plugin Features

By the end of this section, the reader will have learned about the functionalities exposed by the *Kali-ma* plugin for facilitating interaction with and configuration of software components in the NeOn Toolkit. An insight is also provided, as well as documented, as to which steps the user needs to perform in order to activate and interact with these functionalities.

Although the *Kali-ma* plugin is oriented toward providing alternate modalities for end users to interact with the functionalities provided by the NeOn Toolkit, the rule body and several other aspects by which these modalities are provided are customizable. Some of such features are configurable at runtime by end users, while others are available by applying simple extensions to plugins by their respective developers. By this distinction, the remainder of this section is structured so as to allow a neat separation between functionalities that refer to end users for direct consumption and functionalities that refer to developers for their plugins to

provide alternate interaction paths. In particular, the next section will also focus on what features can be configured by end users prior to launching the Kali-ma plugin on a running NeOn Toolkit platform.

15.2.1 Functionalities for End Users

When the Kali-ma plugin is activated, a desktop-integrated graphical user interface (GUI), called *Dashboard*, replaces the traditional Eclipse Workbench-based NeOn Toolkit interface. The constituents of this user interface, an example of which is shown in Fig. 15.2, are lightweight graphical elements, or *widgets*. A single widget represents either a built-in functionality provided by Kali-ma or a group of functionalities provided by some other NeOn Toolkit plugin.

Kali-ma provides a number of functionalities aimed at end users and aids them in the configuration of, and rapid access to, selected sets of tools apt for completing certain classes of tasks. These are as follows:

- *Tool organization and selection* based on preferred criteria.
- *Quick plugin access* that groups most functionalities of a plugin into a single widget.
- *Profile management* for bookmarking sets of plugins and associating them with ontology projects, thereby managing *profiles*.
- Project-based *real-time chat* that allows remote collaborating parties to share metadata of a common ontology project.
- *Advanced search* for ontology data and metadata.
- *Pipeline assembly*, for broadcasting the output of a plugin to other listening plugins in order to accomplish complex tasks.
- *Assistant*, for obtaining real-time guidance.

15.2.1.1 Preliminary Configuration

As with most NeOn Toolkit plugins, Kali-ma is configurable in several aspects concerning its way to handle interaction with the framework. While it does make sense to customize some of these aspects only once the Kali-ma dashboard has been activated, other features require prior configuration, as they affect the way dashboard elements are constructed. This section discusses the latter set of features and the steps to follow for configuring them.

Kali-ma comes with a “safe” default setup, in that all the plugin functionalities can be activated with no alteration of the default settings, granted an available internet connection. The only exception is the chat functionality, which requires the user to set the hostname of a Jabber/XMPP chat server where she has an account already registered.

All the settings of the Kali-ma plugin are grouped under a single *Kali-ma* entry in the *NeOn Toolkit Preferences* category. Remember that the *Preferences* panel

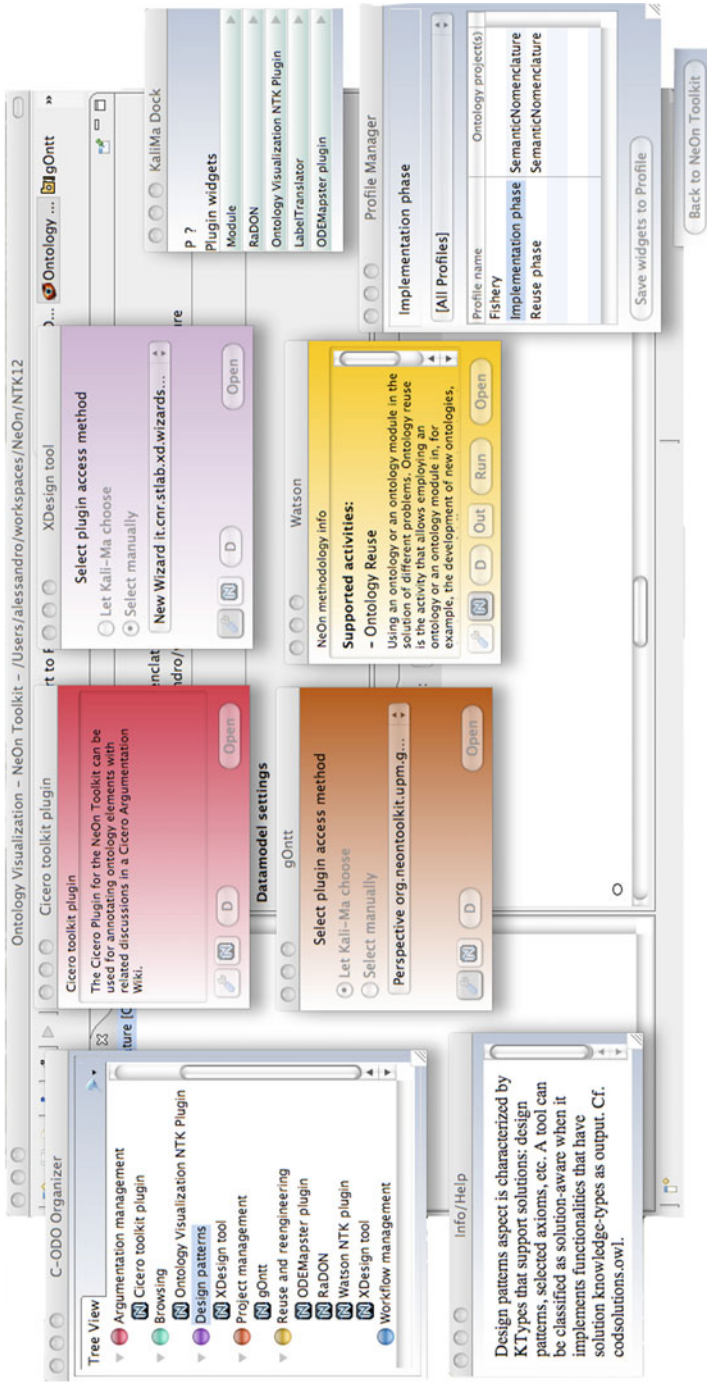


Fig. 15.2 The Kali-ma dashboard of widgets. Sorted by column, *top to bottom*, then *left to right*: the *codot organizer*; the *helper widget*; widgets representing the following plugins: Cicero, gOntt, XDesign Tools and Watson; the *dock widget* with placeholders for five more plugin widgets; the *profile manager*; the *switch widget* for returning to the NeOn Toolkit workbench

can be accessed in different ways, depending on the operating system used. For example, Windows users will find it in the *Window* top menu, while OS X users will find it in the *NeOn Toolkit* top menu.

Due to their intrinsic heterogeneity, the configuration parameters are in turn grouped into four categories:

1. *Appearance* is the category of customizable cosmetic aspects of the Kali-ma user interface.
 - *Open profiles docked* is an optional override for the docking options of each plugin widget in a user profile. When this option is checked, if the user opens a Kali-ma user profile, all of its plugin widgets will be minimized to the Kali-ma dock on startup, even if set otherwise in the profile itself. This option is preferable for users who wish to start with a dashboard as clear as possible.
 - *Widget background policy* determines what background color should be used for each plugin widget. Depending on the setting, the color can be either the one used for a category that classifies the plugin or one set by the user for that specific plugin.
2. *Network* deals with how Kali-ma exploits online resources. Currently, all the settings in this category are related to the built-in XMPP chat service.
 - *XMPP Host* and *Port* locate the resource where the XMPP messaging service is provided, e.g., for GTalk use Host `talk.google.com` and Port `5222`.
 - *XMPP Service name*, the identifier of the XMPP service on the host, if different from the host name, e.g., `jabber.org`.
 - *Multiuser chat service*, the identifier of the Multiuser Chat (MUC) service on the host, e.g., `conference.jabber.org`. Although not all XMPP-based services come with this functionality, this is required for the Kali-ma chat to work.
3. *Reasoning* enables the user to configure the parameters by which Kali-ma should locate and classify ontology design tools. These settings can have a significant impact on startup performance, but their default values are relatively safe on that respect. Note that changes to this configuration will only take effect the next time the Kali-ma dashboard is launched.
 - *Plugin address book location* is the physical URI of the ontology that indicates where the OWL descriptions of each plugin should be fetched from. Its default value is a plugin registry maintained by the Ontology Design Patterns portal² (Presutti et al. 2008).
 - *Criterion for tool classification* selects which property should be used as a criterion for classifying ontology design tools. Currently selectable criteria are *Design aspects*, *Processes and activities*, and *Design functionalities*.

²The Ontology Design Patterns portal, <http://www.ontologydesignpatterns.org>

- *Perform online update* denotes when Kali-ma should check for updates to the online plugin address book. Available options are “Each run,” “Only on next run,” and “Never.” Note that if the address book has not been fetched yet (e.g., on the first run of Kali-ma ever), the update will be performed even if the “Never” option is set.
 - *Cache plugin classification* indicates whether Kali-ma should materialize all inferences about plugins and store them into a local cache ontology. Because inferencing is a lengthy and highly CPU-intensive task, it is recommended to set this option unless major changes in the plugin registry occur. Note that this option only indicates whether the cache should be *built*, not whether it should be *used*: it will always be used if present. To force-rebuild the cache, the user can clear all the local data by clicking the *Clear now* button. This button is grayed out if there are no such local data.
4. *Toolkit integration* manages the way Kali-ma handles the standard NeOn Toolkit user interface along with its own. Users will configure these parameters according to their will to be provided with both interfaces altogether.
- *Stick dashboard to main window*. If this option is set, the Kali-ma UI will appear on top of the standard NeOn Toolkit window, and its behavior will mimic the one of that window. Thus, when the NTK window is minimized, hidden, or maximized, so will be the Kali-ma widgets. Note that the Kali-ma dashboard is not modal; therefore, the NTK UI components in the background can still be interacted with.
 - *Main window behavior* allows the user to set how the main NTK window should appear or disappear when the Kali-ma dashboard is activated or deactivated. The user can opt for the main window to be hidden or minimized or neither. This option is only available when the “Stick dashboard to main window” option is unchecked.

Example 15.1. This and all the examples in this chapter are based on a run-through scenario extracted from the case study described in Chap. 20. The Semantic Nomenclature of pharmaceutical products was carried out using the NeOn Methodology and related software support. Therefore, in order to use Kali-ma to carry out the activities specified in this methodology, an engineer will select *Processes and activities* from the *Reasoning → Criterion for tool classification* configuration panel.

15.2.1.2 Activating the Dashboard

Unlike most other NeOn Toolkit plugins, which support specific tasks in the engineering of networked ontologies and are therefore integrated with the platform, Kali-ma provides a GUI that runs in parallel with the standard one. For this reason, Kali-ma integration is limited to the preferences panel and the commands for

activating its own user interface, called the *dashboard*. These commands are located:

- In the *Launch Dashboard* menu entry in the *Kali-ma* top menu
- In the NeOn Toolkit top bar as the *Launch Dashboard* button (an open perspective is required for displaying the button)

When one of these two actions is performed, the reasoning and plugin discovery tasks for preparing the dashboard are started as a background job. In particular, the following actions are performed:

1. The local tool descriptions and cache ontology are checked. If neither is present, or the online update parameter is set, plugin descriptions are fetched from the locations indicated in the online registry.
2. If variations between the local plugin ontology and the online registry are detected, the user is notified about these changes and prompted to choose whether to apply them or not. If changes are applied, any local cache is invalidated.
3. Plugins are classified by the designated criterion in one of the following ways:
 - If a valid local cache is present, it is queried directly.
 - If no valid cache is present but Kali-ma is configured to build one, it will first do so then query the cache it just built. This task is highly CPU intensive but will not have to be performed again as long as the cache remains valid.
 - If no valid cache is present and Kali-ma is *not* configured to build one, it will use a reasoner to classify plugins. This task is CPU intensive and will have to be run on every dashboard startup unless a cache is built.
4. The Kali-ma dashboard is activated and displayed in its default state. The NeOn Toolkit main window is hidden from view if set to do so.

Example 15.2. The project manager of the Semantic Nomenclature case study creates a new NeOn Toolkit ontology project called “SemanticNomenclature” and shares it with engineers using a version control tool such as CVS or Subversion. When the Dashboard is activated using the Launch Dashboard button, Kali-ma becomes aware of this project and can store profiles and configurations in its directory.

Recall that the dashboard is an aggregate of basic user interface components called *widgets*, whose look-and-feel exploits the capabilities offered by the GUI toolkit of the host operating system. Every widget identifies a functionality, or set of functionalities, in the NeOn Toolkit. Widgets 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.

15.2.1.3 Organizing the Plugin Space

The heart of the Kali-ma approach for organizing the NeOn Toolkit as a functionality provider resides in the classification of its plugins by a unique, design-centered criterion that is nonetheless customizable. Therefore, its core functionality is to present end users with an overview of the plugins that are available in their running instance of the NTK and to help them select the one(s) whose coverage best suits the tasks that need to be performed.

The *C-ODO organizer* is the widget used for presenting this aggregate overview of plugins. This widget is named after C-ODO Light, the design ontology that is the base for all the classification criteria adopted by default in Kali-ma. Recall that an overview of the goal, rationale, and architecture of the C-ODO Light ontology network was given in Chap. 5.

The *C-ODO organizer* is the tool browser provided by Kali-ma. Users are free to choose from time to time, whether they wish to explore the plugin space as a tree or as a graph, by switching between the *Tree View* and the *Wheel View* tabs. The *Tree View* is organized as a simple *Category* \rightarrow *Plugin* two-level tree; i.e., by expanding a category it is possible to view all and only the plugins that fall under that category. This also implies that a plugin that encompasses more than one category will appear as a child of multiple nodes in the taxonomy. The *Wheel View*, so called after the shape adopted by the category set, provides the same information in a graph. Although it takes up more space than the *Tree View*, it displays more useful information altogether. When a category is selected in the *Wheel View*, all and only the plugins under that category are displayed as in the *Tree View*. However, for each shown plugin, an edge appears for *every other* category it falls under.

The categories used for classifying plugins have a variable dependency on C-ODO Light, yet they are all based on this ontology for modeling the notion of an ontology design tool. The criterion used for identifying these categories can be selected from the *Reasoning* panel of the Kali-ma preferences (entry “Criterion for tool classification”) prior to launching the plugin. The available criteria are as follows:

1. *Custom design functionalities*. These denote specific tasks and operations involved in the design of networked ontologies. They are arbitrarily defined by plugin developers, so the set of design functionalities can be highly fine grained, depending on the choices of developers. “Create project,” “Cast vote,” or “Delete annotation” are examples of such design functionalities. This criterion is enabled by selecting “*implements (Design Functionality)*” from the *Reasoning* preferences. End users should expect a sparse classification, with many

categories each with a limited number of plugins, yet with high redundancy across multiple categories, roughly one for each functionality implemented in that plugin.

2. *NeOn Methodology* refers to the fixed set of activities that are part of the NeOn Methodology canon as defined in Chap. 2. For this criterion, the categories are established a priori, and whether a plugin supports an activity in the methodology, this reflects the rationale used for selecting such plugins in gOntt (cf. Chap. 14). This criterion is enabled by selecting “*supports activity (Activity)*” from the *Reasoning* preferences.
3. *Ontology design aspects* is a limited, fixed set of generic design functionalities that aggregate the most common aspects of designing networked ontologies in a collaborative environment. The categories are set and very limited in order to provide dense classification of design tools. Also, it is the only case where the categories to which plugins belong are not explicitly defined but are instead obtained by inferencing over other features defined by the developers, namely the types of knowledge their plugins consume and produce. This criterion is enabled by selecting “*has aspect (Design Aspect)*” from the *Reasoning* preferences.

Both the Tree View and the Wheel View in the C-ODO organizer can be filtered by means of the funnel-shaped icon opposite the tabs. The filtering feature is due to the fact that the ABoxes describing ontology design tools, as well as their registries, are not bundled with the actual tools. In fact, they do not reside locally on the host platform in general but are instead exposed on the web. Moreover, they are not necessarily limited to NeOn Toolkit plugins but can span across several frameworks and architectural paradigms, such as plugins for other platforms, stand-alone applications, web applications, and web services.

Thus, three filters are available and can be cascaded: “Show only NeOn Toolkit plugins” will exclude all those design tools that, according to their ontological descriptions, do not qualify as plugins for the NeOn Toolkit. “Show only installed tools” will apply the previous filter and skim all the NeOn Toolkit plugins that are known to exist but are not detected as installed on the host platform. Finally, “Hide empty categories” will remove all the nodes representing categories to which no design tools are known to belong, regardless of the status of the other filters.

Example 15.3. The Semantic Nomenclature project manager has to select plugins for the implementation phase of the use case. The C-ODO organizer Tree View shows all the activities in the NeOn Methodology that come with software support. The *ODEMapster* plugin is selected (by double-clicking) from the “Non-Ontological Resource Reuse” activity, the *OWLDoc* plugin from the “Ontology Documentation” activity, the *Watson* plugin from the “Ontology Reuse” activity, and the *RadON* plugin from the “Ontology Validation” activity. To create a schedule for all the activities to be performed in the phase, the *gOntt* plugin widget is also selected from the “Scheduling” activity. When each plugin is selected, its corresponding widget is displayed.

15.2.1.4 Interaction with Plugins

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, both on the functional level and on the user interface level³. 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 stand-alone 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 stand-alone 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 displaying 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.
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 NewWizards 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 schedules. 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 stand-alone use cases for creating new resources from scratch.

³ http://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points%3F

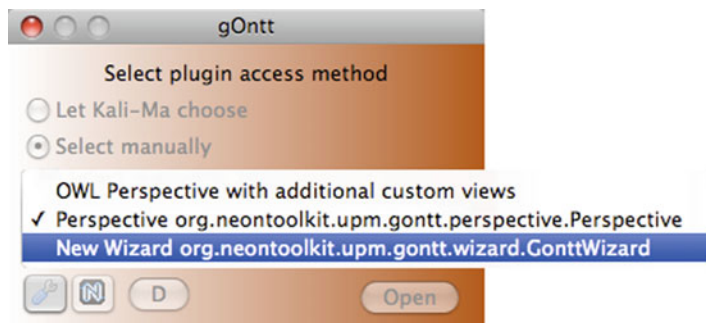


Fig. 15.3 Access method selection for the gOntt plugin

Figure 15.3 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 feature assigned to the Project Management design aspect). The gOntt plugin contributes to the NeOn Toolkit by means of both a Perspective and a New Wizard for creating new schedules. A user can select either access method for launching the gOntt plugin once the “Open” button is clicked.

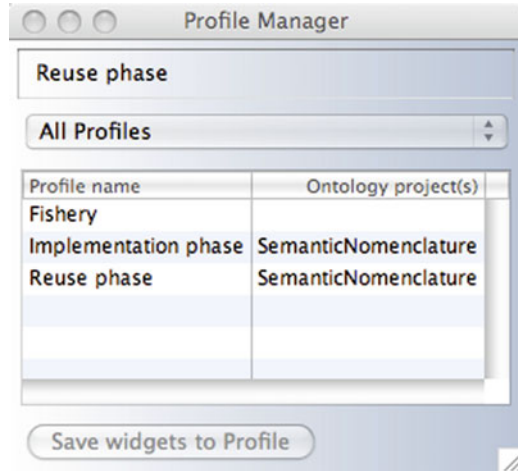
15.2.1.5 Profile Management

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 Fig. 15.4, 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 so, 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

Fig. 15.4 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 *Semantic Nomenclature* ontology project



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 of the table in the center of the widget, 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 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.

Example 15.4. The Semantic Nomenclature project manager wishes to share the tools for the Implementation phase selected earlier with all the ontology engineers who are set to perform each activity. A profile named “Implementation phase” is created and bound to the “SemanticNomenclature” ontology development project in the NeOn Toolkit. Because all participants are synchronized on this project, they will all get a copy of the new profile the next time they update their working copy of the project.

15.2.1.6 Dashboard Control and Docking

To counter the risks of ending up with a screen overcrowded by widgets, Kali-ma comes with an additional interface element called the *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 element that serves a placeholder for a docked widget. Each entry 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.

15.2.1.7 Project-Based Real-Time Chat

Several phases of the articulated ontology life cycle management process are conceived with user collaboration in mind, and as such should they be carried out (Holsapple and Joshi 2002). Activities such as the collective argumentation of ontologies, or portions thereof, can be performed asynchronously, i.e., no different than by posting comments on message boards and the like. There may be cases, however, where multiple users collaborating on the same ontology project may require to coordinate their efforts in real-time, in order not to bottleneck one another. One such circumstance may involve two ontology engineers developing separate modules of an ontology network, whose entities need to be related via equivalence statements nonetheless. In such a situation, the user who needs to perform the alignment will need to know the name of the alignment target as soon as possible, and this can be significantly sped up by synchronous communication.

Kali-ma includes a lightweight real-time chat system to support synchronous communication in an environment where users can instantly share references to resources in a common ontology project. Through the *Chat widget*, a single user can join one or more dedicated virtual chat rooms, each named after an ontology project she has in common with other users. Additionally, for each project, it is possible to send the identifiers of any OWL entity loaded within that project with just a few keystrokes.

Example 15.5. The project manager and engineers that share the “SemanticNomenclature” project and have the same XMPP Chat configuration in the Kali-ma

preferences will all be presented with an option to join the “SemanticNomenclature” chat room and discuss their engineering activities there.

As with other optional widgets, the Kali-ma chat interface can be activated by means of the Dock widget by simply clicking the balloon-shaped icon on its toolbar. In the default panel of this widget, it is sufficient for a user to type in her credentials (set by the chat server administrator), freely choose an alternate label, or *alias*, and log into the chat server. With this done, a combo box will display the list of available chat rooms, each named after an ontology project in her NeOn Toolkit workspace. Multiple chat rooms, one per project, can be joined at once, and a chat room will be seamlessly created on the fly if it has not yet been configured by another user. A user may send any free text message by simply typing it in a chat room window. However, if a reference to an OWL class, property, or individual needs to be broadcast to other users sharing the same project, it is sufficient to start typing in part of its name (not necessarily a prefix) and invoke the autocompletion key combination (usually `Ctrl + Space`) to select from a list of matching entities that exist *within that project*. Multiple OWL entity references can be broadcast in a single message by invoking autocompletion.

Any party is free to host a chat server compatible with Kali-ma. The plugin uses the open standard instant messaging protocol *XMPP* (Extensible Messaging and Presence Protocol)⁴, which sports numerous compatible instant messaging clients as well as communication services (Google Talk⁵ and Jabber⁶ being two of them). Anyone can set up an off-the-shelf XMPP server on a host and create accounts for users, who can quickly configure Kali-ma on their clients (see Sect. 15.2.1.1) to instantly use it for relaying their messages.

15.2.1.8 Obtaining Help

Kali-ma provides its own real-time help system, aimed at displaying 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.

Real-time guidance is provided through the *Helper widget*. 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. 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

⁴ XMPP, <http://xmpp.org>

⁵ Google Talk, <http://www.google.com/talk>

⁶ Jabber, originator of the initial XMPP design and implementation, <http://www.jabber.org>

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.

15.2.2 Functionalities for Plugin Developers

One goal of Kali-ma is to reorganize the plugin space under a single, shared criterion that can apply to the majority of plugins. To that end, it provides a set of functionalities to aid developers in describing the features of their plugins so that Kali-ma can elaborate on them and construct a single, harmonic view. These functionalities belong to the following categories:

- *Plugin description management* guides users throughout the creation of the ontology that describes how a plugin contributes to the life cycle management of ontologies.
- The *interoperability API* allows developers to launch and customize a Dashboard programmatically from the code of any plugin.

15.2.2.1 Plugin Description Management

As will be presented in Sect. 15.2.3.2, 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 semiautomatic construction of C-ODO Light-based plugin descriptions. The next section details the key functional characteristics of both features mentioned above.

15.2.2.2 Plugin Description Generator

Knowledge of the ontology tool population is not delegated 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 the remainder of the chapter)⁷, its name paying homage to an inspiring online form for generating FOAF profiles. Codomatic is a simple, single-page Ajax application for constructing *C-ODO Light*-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 explained in Sect. 15.2.1.3. 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 argumentation plugin (Dellschaft et al. 2008), depicted in Fig. 15.5, 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 to be used 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`

CODO-O-MATIC: Ontology Tool description generator

Ontology Base URI:

Plugin name*: Camel syntax:

This tool is a NeOn Toolkit plugin ID:

Plugin description:

Input type(s): Selected: 5 of 20
 Position
 Annotation
 Data structure
 KOSElement
 Others (comma-separated)

Output type(s): Selected: 3 of 20
 Argument
 Argumentation thread
 Idea
 Position
 Others (comma-separated)

Supported design functionalities: Selected: 8 of 28
 Preferential voting
 Propose solution
 Provide argument
 Start preferential voting
 Others (comma-separated)

Supported NeOn activities: Selected: 2 of 53
 Ontology Assessment
 Ontology Comparison
 Ontology Conceptualization
 Ontology Configuration Management

Format:

```

<owl:imports rdf:resource="http://www.ontologydesignpatterns.org/spot/codo/neonactivities.owl"/>
<owl:imports rdf:resource="http://www.ontologydesignpatterns.org/spot/codo/soo/leac/stk2codo.owl"/>
<owl:Ontology>

<!--
#####
// Object Properties
#####
-->

<!-- http://www.ontologydesignpatterns.org/spot/codo/codotoolkit.owl#dependsOn -->
<owl:ObjectProperty rdf:about="<code>codotoolkit:dependsOn</code>" />

<!-- http://www.ontologydesignpatterns.org/spot/codo/codotoolkit.owl#hasInputType -->
<owl:ObjectProperty rdf:about="<code>codotoolkit:hasInputType</code>" />

```


Powered by 

Fig. 15.5 The *Codomatic* tool description generator, after constructing the RDF code for an argumentation management plugin called Cicero

⁷ At the time of writing, the service is hosted at <http://wit.istc.cnr.it:8080/codomatic>

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.

The list boxes that follow this field in the figure 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 processes and activities that it supports. For all fields but the NeOn processes and 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.

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 guide⁸. Being a Semantic Media Wiki, it is then possible to export these references in RDF format for Kali-ma to consume.

⁸ http://neon-toolkit.org/wiki/Plugin_HowTo

15.2.2.3 Interoperability API

In addition to supporting collaboration and interaction between end users, Kali-ma as a plugin comes with additional developer features that allow other ontology plugins to interoperate either with each other or with Kali-ma. There are two distinct methods of allowing *programmatically interoperability*, which is achieved through simple direct intervention on the plugin code. These two methods cover separate interoperability aspects and can be implemented independently. They are:

1. Construction of *Pipeline assemblies* within widgets, for executing plugin functionalities without switching to the plugin user interface for that plugin
2. External *Dashboard control*, for manipulating the contents of the Dashboard

Interoperability between plugins is achieved by construction of *pipeline assemblies*, which are dynamic software structures where the output of one component can be concatenated to one or more other components in the assembly in order to execute complex computational tasks. For instance, a design pattern selection service exposed by the *eXtreme Design* plugin (described in Chap. 3) could reuse the output axioms of a search issued using the *Watson* plugin (described in Chap. 7) in order to perform query expansion for broader pattern selection. Because the process has no strict coupling at build time, such a scenario can be realized without either plugin knowing a priori which other plugin it should expect its input from, or which one should accept its output.

The other supported interoperability aspect is *Dashboard control*, i.e., the programmatic manipulation of the Kali-ma user interface. This allows other developers to construct custom dashboard configurations specific for the engineering activity supported by another tool. Among NeOn Toolkit plugins, the *gOntt* tool for project scheduling supports Kali-ma dashboard interoperability, as it is possible from within a *gOntt* schedule to launch a Kali-ma dashboard containing widgets for all registered NeOn Toolkit plugins that support a given process, activity, or phase in that schedule. This support is among the features showcased by the *gOntt* plugin description in Chap. 14.

To reach either level of interoperability, a plugin must implement a simple Java API exposed by Kali-ma itself. A developer who wishes a plugin functionality to be directly called via its dashboard widget will simply have to implement an Eclipse *extension point*, which is mapped to a simple Java interface, both provided by Kali-ma. The developer will simply have to wrap a call to a plugin functionality into a Java class that implements this interface and annotate the single public method with the types of the parameters expected to be consumed and produced by that functionality.

The Dashboard control API is also simple and straightforward. It is enough for a developer to invoke any static method of the `DashboardLauncher` class exposed by the Kali-ma API, and a dashboard will be launched, containing widgets for all the available plugins whose identifiers were passed as parameters. This implementation may occur in a separate plugin, without any intervention on the original plugin code.

15.2.3 Architectural Design

The software architecture of the Kali-ma plugin, used for performing semantic reorganization of the tool space, incorporates both procedural and logical components. That is, although the plugin is essentially a Java program (or, to be more precise, a set of OSGi bundles) like most other plugins, some functionalities are not entirely encoded as procedures in the plugin code but instead rely on formal semantics that describe their behavior. 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. Ontologies that describe the domain model, plugin space, and classification criteria are dynamically aggregated and linked at runtime.

The sections that follow provide an insight on the software architecture of Kali-ma. After a quick overview on the next section, Sect. 15.2.3.2 describes the actual ontology network used by the tool. Section 15.2.3.3 describes the software modules that handle and reason upon the ontology network in order to classify NeOn Toolkit plugins. Finally, Sect. 15.2.3.4 provides a quick insight as to how the result is presented to the user.

15.2.3.1 Basic Software Architecture

The heterogeneous representation of the Kali-ma components, as well as the openness to possibly reusing the procedural components in engineering fields other than ontologies, imply a layered infrastructure of the tool. This infrastructure can be seen as split into three major components as depicted in Fig. 15.6: 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; and lastly, the *presentation component* generates the widgets and handles communication between Kali-ma, NTK plugins, and the NTK core.

15.2.3.2 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 only the essential logical infrastructure is hardwired, whereas expert ontology engineers can 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.

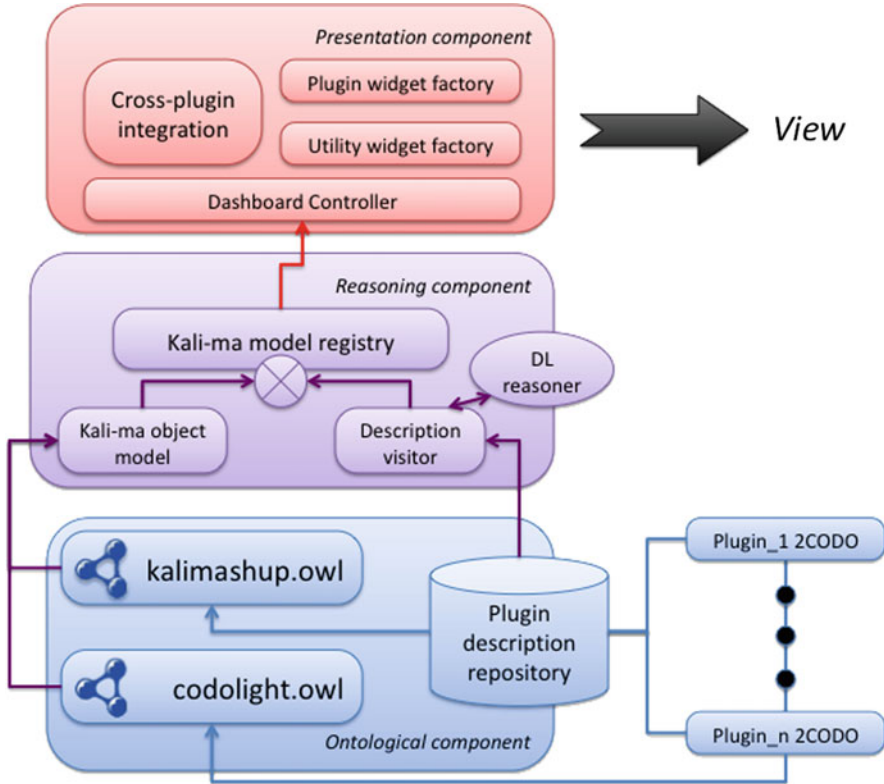


Fig. 15.6 The Kali-ma infrastructure and its main components

The layers of the Kali-ma ontological component include:

- A *foundational/domain layer*, which is essentially the *codolight* model for ontology design (described in Chap. 5). Its modularity and distinctive support for collaborative life cycle 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.
- Sets of *categorization rules*, where the classification criteria used for providing a taxonomy of tools in the Organizer widget are formalized. Recall that the default criteria are implementation of *design functionalities*, support for NeOn Methodology *processes and 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.
- The *tool space population* model which includes C-ODO Light-based OWL descriptions of NeOn Toolkit plugins, each describing what types of task a certain plugin can help accomplish, what types of knowledge representation it

can handle, and so on. These are not hardwired in the built-in portion of the ontological component and can be located anywhere on the web. Recall from Sect. 15.2.2.2 that an online service is available for the automatic generation of these tool descriptions.

15.2.3.3 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 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.

The *reasoning component* comprises the following modules:

- 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 and querying them. This system can be configured to manage a cache; thus, it does not necessarily query the DL reasoner on 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.

15.2.3.4 Presentation Component

The top-level component of the Kali-ma infrastructure, called *presentation component*, implements both the user interface and its controller in the *Model-View-Controller* (MVC) paradigm (Reenskaug 1979). This element is responsible for leveraging the underlying C-ODO Light-based object model and presenting the outcome of reasoning tasks performed thereupon. Widget factories, dashboard management, and event handling support all belong to this component. Once generated, widgets are deployed on the target view (typically, the operating system desktop) and integrated among other operating system windows.

References

- Damian D, Chisan J (2006) An empirical study of the complex relationships between requirements engineering processes and other processes that lead to payoffs in productivity, quality, and risk management. *IEEE Trans Softw Eng* 32(7):433–453
- Dellschaft K, Engelbrecht H, Barreto JM, Rutenbeck S, Staab S (2008) Cicero: tracking design rationale in collaborative ontology engineering. In: Bechhofer S, Hauswirth M, Hoffmann J, Koubarakis M (eds) *ESWC, Lecture notes in computer science*, vol 5021. Springer, Berlin/Heidelberg/New York, pp 782–786
- Holsapple CW, Joshi KD (2002) A collaborative approach to ontology design. *Commun ACM* 45:42–47
- Presutti V, Gangemi A, David S, de Cea GA, Suárez-Figueroa MC, Montiel-Ponsoda E, Poveda M (2008) A library of ontology design patterns: reusable solutions for collaborative design of networked ontologies. Deliverable D2.5.1, NeOn project
- Reenskaug T (1979) Models – views – controllers. Technical report, Technical note, Xerox Parc