D 1.6.1 Portal Requirements and System Design

Coordinator: Ángel López-Cima (UPM)

Asunción Gómez-Pérez (UPM), Miguel Rodríguez Hernández (UPM)

Abstract.
EU-IST Network of Excellence (NoE) IST-2004-507482 Knowledge WebEB
Deliverable D1.6.1 (WP1.6)
The Knowledge Web Portal is a software infrastructure underpinning the integration of the activities of the Knowledge Web partners. It serves as portal for information access and a dissemination point for ontology researchers, engineers, application and content developers in both academic and industrial institutions. It will provide a common medium of presentation where the partners' development work is deployed, publicized and promoted, along with work on technology promotion, research and e-learning. The KW portal is a semantic web portal available at http://knowledgeweb.semanticweb.org

Document Identifier:	Knowledge WebEB/2004/D1.6.1/v1.0
Class Deliverable:	Knowledge WebEB EU-IST-2004-507482
Version:	V1.0
Date:	December 14, 2004
State:	Final
Distribution:	Public

Knowledge Web Consortium

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2004-507482.

University of Innsbruck (UIBK) – Coordinator
Institute of Computer Science,
Technikerstrasse 13
A-6020 Innsbruck
Austria
Contact person: Dieter Fensel
E-mail address: dieter.fensel@uibk.ac.at

École Polythechnique Fédérale de Lausanne (EPFL)
Computer Science Department
Swiss Federal Institute of Technology
IN (Ecublens), CH-1015 Lausanne.
Switzerland
Contact person: Boi Faltings
E-mail address: boi.faltings@epfl.ch

France Telecom (FT)
4 Rue du Clos Courtel
35512 Cesson Sévigné
France. PO Box 91226
Contact person : Alain Leger
E-mail address: alain.leger@rd.francetelecom.com

Freie Universität Berlin (FU Berlin)
Takustrasse, 9
14195 Berlin
Germany
Contact person: Robert Tolksdorf
E-mail address: tolk@inf.fu-berlin.de

Free University of Bozen-Bolzano (FUB)
Piazza Domenicani 3
39100 Bolzano
Italy
Contact person: Enrico Franconi
E-mail address: franconi@inf.unibz.it

Institut National de Recherche en Informatique et en Automatique (INRIA)
ZIRST - 655 avenue de l'Europe - Montbonnot Saint Martin
38334 Saint-Ismier
France
Contact person: Jérôme Euzenat
E-mail address: Jerome.Euzenat@inrialpes.fr

Centre for Research and Technology Hellas / Informatics and Telematics Institute (ITI-CERTH)
1st km Thermi – Panorama road
57001 Thermi-Thessaloniki
Greece. Po Box 361
Contact person: Michael G. Strintzis
E-mail address: strintzi@iti.gr

Learning Lab Lower Saxony (L3S)
Expo Plaza 1
30539 Hannover
Germany
Contact person: Wolfgang Nejdl
E-mail address: nejdl@learninglab.de

National University of Ireland Galway (NUIG)
National University of Ireland
Science and Technology Building
University Road
Galway
Ireland
Contact person: Christoph Bussler
E-mail address: chris.bussler@deri.ie

The Open University (OU)
Knowledge Media Institute
The Open University
Milton Keynes, MK7 6AA
United Kingdom.
Contact person: Enrico Motta
E-mail address: e.motta@open.ac.uk

Universidad Politécnica de Madrid (UPM)
Campus de Montegancedo sn
28660 Boadilla del Monte
Spain
Contact person: Asunción Gómez Pérez
E-mail address: asun@fi.upm.es

University of Karlsruhe (UKARL)
Institut für Angewandte Informatik und Formale Beschreibungsverfahren – AIFB
Universität Karlsruhe
D-76128 Karlsruhe
Germany
Contact person: Rudi Studer
E-mail address: studer@aifb.uni-karlsruhe.de

University of Liverpool (UniLiv)
Chadwick Building, Peach Street
L697ZF Liverpool

University of Manchester (UoM)
Room 2.32. Kilburn Building, Department of Computer Science, University of Manchester,

United Kingdom
Contact person: Michael Wooldridge
E-mail address: M.J.Wooldridge@csc.liv.ac.uk

Oxford Road
Manchester, M13 9PL
United Kingdom
Contact person: Carole Goble
E-mail address: carole@cs.man.ac.uk

University of Sheffield (USFD)
Regent Court, 211 Portobello street
S14DP Sheffield
United Kingdom
Contact person: Hamish Cunningham
E-mail address: hamish@dcs.shef.ac.uk

University of Trento (UniTn)
Via Sommarive 14
38050 Trento
Italy
Contact person: Fausto Giunchiglia
E-mail address: fausto@dit.unitn.it

Vrije Universiteit Amsterdam (VUA)
De Boelelaan 1081a
1081HV. Amsterdam
The Netherlands
Contact person: Frank van Harmelen
E-mail address: Frank.van.Harmelen@cs.vu.nl

Vrije Universiteit Brussel (VUB)
Pleinlaan 2, Building G10
1050 Brussels
Belgium
Contact person: Robert Meersman
E-mail address: robert.meersman@vub.ac.be

Workpackage 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 writing parts of this document:

- Universidad Politécnica de Madrid.
- France Telecom & RD.

Changes

Version	Date	Author	Changes
0.1	01-03-2004	A. López-Cima	Table of Content
0.2	02-06-2004	A. Gómez-Pérez	Software methodology to be used
0.3	01-07-2004	A. López-Cima	Business Model Definition
0.4	15-07-2004	A. López-Cima M. Rodríguez	Section 4.1 - Ontology Repository Functionality
0.5	03-08-2004	A. López-Cima M. Rodríguez	Section 4.2 - Administration Definition
0.6	08-08-2004	A. López-Cima M. Rodríguez	Section 4.3 - Logging Definition
0.7	20-08-2004	A. López-Cima M. Rodríguez	Section 4.4 - Semantic Editing Definition
0.8	10-09-2004	A. López-Cima M. Rodríguez	Section 4.5 - Semantic Browsing
0.9	21-09-2004	A. López-Cima M. Rodríguez	Section 4.6 - Semantic Searching
0.10	30-09-2004	A. López-Cima	Section 5 - Analysis Phase
0.11	02-10-2004	A. López-Cima M. Rodríguez	Section 4.7 - Semantic Content Visualization
0.12	17-10-2004	A. López-Cima M. Rodríguez	Section 4.8 - Interoperability
0.13	25-10-2004	A. López-Cima M. Rodríguez	Section 4.9 - Semantic Navigation Model Management
0.14	27-10-2004	A. López-Cima M. Rodríguez	Section 4.10 - Web Design
0.15	30-10-2004	A. López-Cima	1st review of the document
0.16	15-11-2004	A. Gómez-Pérez	2nd review of the document
1.0	10-12-2004	York Sure	3rd review of the document

Executive Summary

The Knowledge Web Semantic Portal [Annex I] is a software infrastructure underpinning the integration of the activities of the Knowledge Web partners. It serves as portal for information access and a dissemination point for ontology researchers, engineers, application and content developers in both academic and industrial institutions. It will provide a common medium of presentation where the partners' development work is deployed, publicized and promoted, along with work on technology promotion, research and e-learning. Among other things, the portal will be used with the advanced learning platform to deliver semantically indexed learning units.

In this deliverable we present the specification of the systems to be created in the Knowledge Web Semantic Portal web site.

First we review the software methodology (Rational Unified Process) that we will follow to develop it.

Then we will present the results of the business modelling and requirement analysis phases of development of the Knowledge Web Semantic Portal.

The portal is running under the following URL since March 1st, 2004:

http://knowledgeweb.semanticweb.org

This deliverable only contains the software specification and system design. The ontologies used by the portal are presented in detail in the deliverable D.1.6.2 “Portal Ontology”, already delivered by 30/6/2004.

Acknowledgments

Table of Contents

1 Introduction

As stated in the executive summary, this deliverable presents the semantic portal requirements and system design for the Knowledge Web WoE. The KW Semantic Portal is built reusing and improving the technology produced in the Esperonto (IST-2001-37343) project.

For this deliverable we study differents platforms of standard portals and semantic portals:

- OntoWebber [Y. Jin et al., 2003]
- OntoWeaver-S [Y. Lei et al., 2004]
- OntoOrganizer [R. Keller et al., 2004]
- SEAL [J. Hartmann, Y. Sure, 2004]
- OIP-Fs [E. Valle, M. Brioschi, 2004]
- IntelliDimension[1]
- My [F. Bellas et al., 2004]
- Bea WebLogic Portal[2]

ODESeW [Corcho et al., 2003] is an ontology-based application built inside the WebODE ontology engineering workbench, that allows managing knowledge-intensive ontology-based Intranets and Extranets. The first version of ODESeW was developed in the framework of the Esperonto[3] project (IST-2001-34373) for building the Esperonto Semantic Portal.

Given that the KW Semantic Portal is the latest instantiation of ODESeW, seen in *Figure 1-1*, the following sections present the ODESeW Business Model and the ODESeW Requirement Analysis Model, which actually are the same as the Knowledge Web ones.

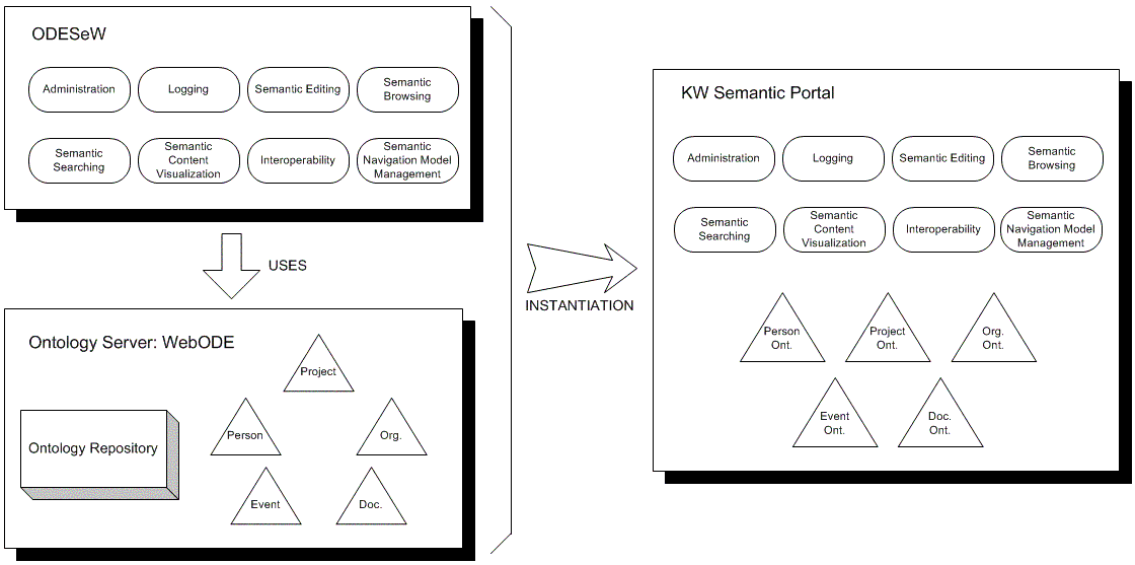

Figure 1-1 *KW Portal as the latest instantiation of ODESeW*

As we said before, the portal is running under the following URL since March 1st, 2004:

http://knowledgeweb.semanticweb.org

[1] http://www.intellidimension.com
[2] http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/portal
[3] http://www.esperonto.net

The deliverable is structured as follows:

- Section 2 explains briefly the Rational Unified Process (RUP), which is the software engineering methodology followed to develop the KW Semantic Portal.

- Section 3 presents the results of the business model phase. In this section, we provide a global overview of the problems addressed by the KW Semantic Portal. This overview contains descriptions of the actors, business use cases and high-level business objects.

- Section 4 contains the results of the requirement analysis model, organized according to the main subsystems identified in the KW Semantic Portal: Administration, Logging, Semantic Editing, Semantic Browsing, Semantic Searching, Semantic Content Visualization, Interoperability and Semantic Navigation Model Management.

  Besides, we include the requirement analysis model of the external systems represented by the web designing and the ontology server.

- Section 5 contains part of the analysis of the KW Semantic Portal, with the Integration environment, which includes a description of the target integration platform, and the software architecture.

- Section 6 includes the bibliographic references used through the deliverable.

2 Software methodology for building the KW Semantic Portal: RUP

This section describes the main aspects of the methodology adopted to develop the KW Portal.

2.1 Rational Unified Process

To build the KW Semantic Portal we have chosen the Rational Unified Process (RUP) [Kruchten, 99] methodology to aid in their development activities. RUP assumes an iterative development approach that includes requirement management, the use of component-based architectures, visual software modelling, its verification and the management and control of changes to it. It is based on six sets of sequential activities enumerated below and explained further in the following sections:

1. Business modelling.
2. Requirement analysis.
3. Analysis.
4. Design.
5. Implementation and testing.
6. Deployment and testing.

2.1.1 Business modelling

In this phase of the methodology, a clear understanding of the environment (in which the KW Semantic Portal is going to be used) is obtained. This phase consists of the following activities:

- **High-level domain modelling**. The main goal of this activity is to define the most important concepts related to the domain of interest.
- **Identification and refinement of business processes**. In this activity, the main use cases of the system are identified and defined.

The results obtained in this phase are the following ones:

- Domain model document for the high-level domain modelling activity.
- UML use case diagrams. The uses cases needed to model the system functionalities from the user's point of view.
- Business object model. A UML object diagram that identifies and models the entities needed for the actual realisation of the previously identified use cases.

2.1.2 Requirement analysis

The main goal of this activity is to obtain the system functional specification. This phase is composed of the following activities:

- **Vision development** which includes high-level description of what the system is going to do.
- **Use cases refinement** which includes a detailed decomposition of the use cases described in the business modelling activity, which are going to be taken into account by the system to be developed.
- **Requirement Specification** which includes a detailed description of what the system should provide in each use case.
- **User-Interface modelling and prototyping**, which includes a first approach to what the users of the system should perceive when using it.

The results obtained in this phase are the following ones:

- Vision document with the high-level description of the system's functionality.
- UML use case diagrams and documents describing them. These UML use case diagrams are obtained from the refinement of the use case diagrams from the previous phase.
- Requirements specification, use case association and requirement characterisation. These documents include who is responsible for their fulfilment, who defined those requirements, etc.
- Sketches of how the user-interface should look like.
- UML interfaces and sequence diagrams. The UML interfaces characterise the interactions of the user-interface with the rest of the system. The sequence diagrams show the dynamic aspects of the interactions of the user-interface with the rest of the system.

2.1.3 Analysis

This activity is intended to obtain a high-level system architecture. The main functional blocks and their interactions are identified on this phase. This phase is comprised of the following activities:

- **Software Architecture**, for describing the main elements that comprise the software architecture system.
- **UML class and sequence diagrams**, for detailing the sequence diagrams which includes the identification of classes and their methods.

The results obtained in this phase are the following ones:

- Software architecture document. A document explaining the system decomposition into subsystems and definition of the main functional blocks.
- Analysis model which includes UML class diagram for the identified functional blocks.
- Use case realization, which includes UML sequence diagrams with the elements of the analysis model and showing how they interact with each other in the use cases identified in the requirement analysis phase.

2.1.4 Design

This activity aims to obtain a detailed system architecture, which is a refinement of the one resulting from the analysis phase. This phase is composed of the following activities:

- **Refinement of the software architecture**. The software architecture obtained in the analysis phases is refined.
- **Refinement of the UML diagrams**. The UML diagrams specified are refined by means of adding, removing or modifying classes or methods.

The results obtained in this phase are the following ones:

- Software architecture document, which includes a refinement of the document developed during the analysis activity.
- Design model, which provides UML diagrams showing the whole system architecture and the design of its components.

2.1.5 Implementation and testing

In this activity the system is implemented, integrated and tested. This phase is composed of the following activities:

- **Integration planning**. It describes how the different components are going to be incrementally integrated and tested.
- **Component implementation and testing**. Each component identified in the previous phases is implemented. Unitary tests are done in order to ensure the correct operation of each component.
- **Subsystem integration and testing**. All components are integrated making integration test in order to guarantee the proper operation of these elements.
- **System integration and testing**. All subsystems are integrated making system tests to verify the correct operations of whole system.

The results obtained in this phase are the following ones:

- An integration plan which provides how the different components are going to be integrated.
- A change request report, for indicating the failures identified during all kind of tests.

2.1.6 Deployment and testing

This activity is devoted to the preparation of the developed system software in order to be delivered to its users for testing. This phase is composed of the following activities:

- **Generation of user documentation and installation support**.
- **Beta testing.** Testing of the beta version of the system in order to verify its correct and proper operation.

The results obtained in this phase are the following ones:

- User's manual. The user's manual is necessary for the correct use by future users.
- Ready-to-install software package which provides the software package with all the components necessary to make the installation and distribution of the system.
- A change request report (indicating the failures identified in beta tests). Requests to modify the detected problems.

In this document we only present the activities 2.1.1 (Section 3), 2.1.2 (Section 4) and part of the 2.1.3 (Section 5).

3 Business Model

In this section we will provide a global view of the domain of the KW Semantic Portal. This overview will include a general description of the objectives that we want to achieve with its development, and a description of its business actors and business use cases. We will also provide a business object model, describing the main subsystems that will be created to give support to the business use cases. We will not go into too much detail on the interactions among subsystems, since this will be covered in the analysis phase.

3.1 Global view of the KW Semantic Portal domain

The Knowledge Web Semantic Portal [Annex I] is a software infrastructure underpinning the integration of the activities of the Knowledge Web partners. It serves as portal for information access and a dissemination point for ontology researchers, engineers, application and content developers in both academic and industrial institutions. It will provide a common medium of presentation where the partners' development work is deployed, publicized and promoted, along with work on technology promotion, research and e-learning. Among other things, the portal will be used with the advanced learning platform to deliver semantically indexed learning units.

The KW Semantic Portal will be:

- **Semantic driven**. The portal uses the five ontologies already presented in deliverable D1.6.2. These ontologies are: Project, Person, Organization, Documentation and Event. The ontologies have been developed using METHONTOLOGY [Gómez-Pérez et al., 2003] and WebODE [Arpirez et al., 2003], and all of them are highly reusable and publicly available at the portal web site in RDF(S) and OWL..

- **User oriented**. We distinguish between KW and External users. KW users, which are also content providers, access public and restricted contents inserted by themselves or by other members in the portal. Finally, External users, who scarcely include new content, mainly access public contents.

- **Permission-based**. Different users will have different permissions either for inserting content on the KW portal or for browsing the collected assets.

- **Interoperable** with semantic web based systems. The KW Portal will be capable of exporting internal content and importing resources from an external semantic information source.

- **Synchronization with the ontologies**. There is an automatic synchronization between the contents of the KW portal and the ontologies in which it is based. So, if an ontology conceptualization is modified with the WebODE ontology editor, the changes will be automatically seen in the KW portal.

3.2 Actors

Many types of users have been identified in this phase. We will group them into five main categories, depending on which part of the system they will work on. These five groups are: *Portal Administrator*, *Ontology Manager*, *Web Designer*, *External User* and *KW User*. *Figure 3-1* presents the specification relationships among all these actors.

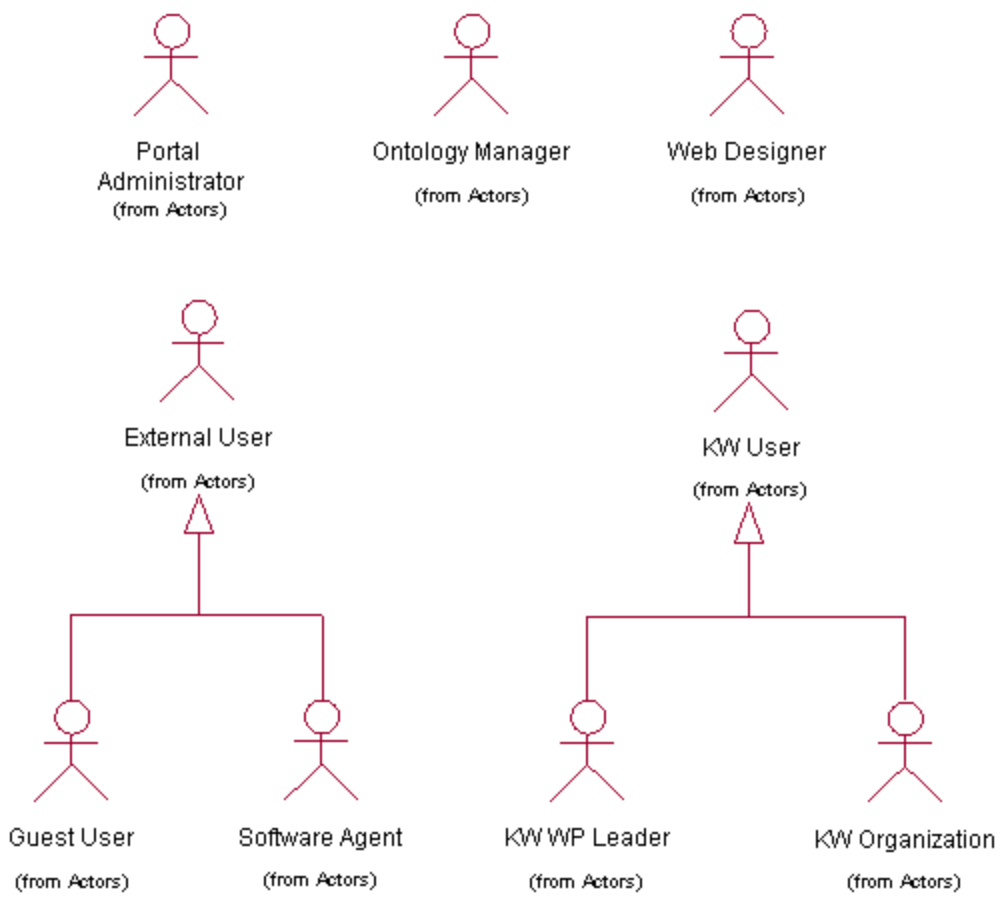

Figure 3-1 *ODESeW business actors.*

- **Portal Administrator.** It will be in charge of administrating the Knowledge Web portal. This involves several tasks, such us user, instance and permission management.

- **External User.** It can be either a *Guest User* or a *Software Agent*. In both cases, it will be an external user that has not logged in, and therefore, has a restricted access to the portal contents.

    - **Guest User.** It represents any user in Internet. It will be able to request KW project information, navigate through hyper-linked information and search for public KW information, but not to introduce, update it nor access KW restricted information.

    - **Software Agent.** It represents a software agent that uses the KW portal content.

- **KW User.** It can be either a *KW WP Leader* or a *KW Organization*. In any case, it will be a corporate user that has logged in the portal. It will be in charge of entering / editing new information, and requesting restricted information to KW members.

    - **KW WP Leader.** This user represents a *KW WP Leader*. It will be in charge of inserting, editing and updating instances from the *Documentation*, *Task* and *Event* ontologies. *KW WP Leaders* should also populate the WP they are leading, their main tasks and deliverables produced.

    - **KW Organization.** Each organization participating in KW will have a user of this type. These users will have permissions for inserting, editing and updating instances from the *Organization*, *Person* and *Event* ontologies.

- **Ontology Manager.** It will be in charge of building the ontologies and managing their evolution.

- **Web Designer.** It will responsible for designing the web pages for accessing the portal contents.

3.3 Business Use Cases

The KW semantic portal is built on top of ODESeW [Corcho et al., 2003] and WebODE platform as ontology server. *Figure 3-2* shows the ten business use cases involved in the KW semantic portal: ontology development and management, administration, logging, semantic editing, semantic browsing, semantic searching, semantic content visualization, interoperability, semantic navigation model management and web designing.

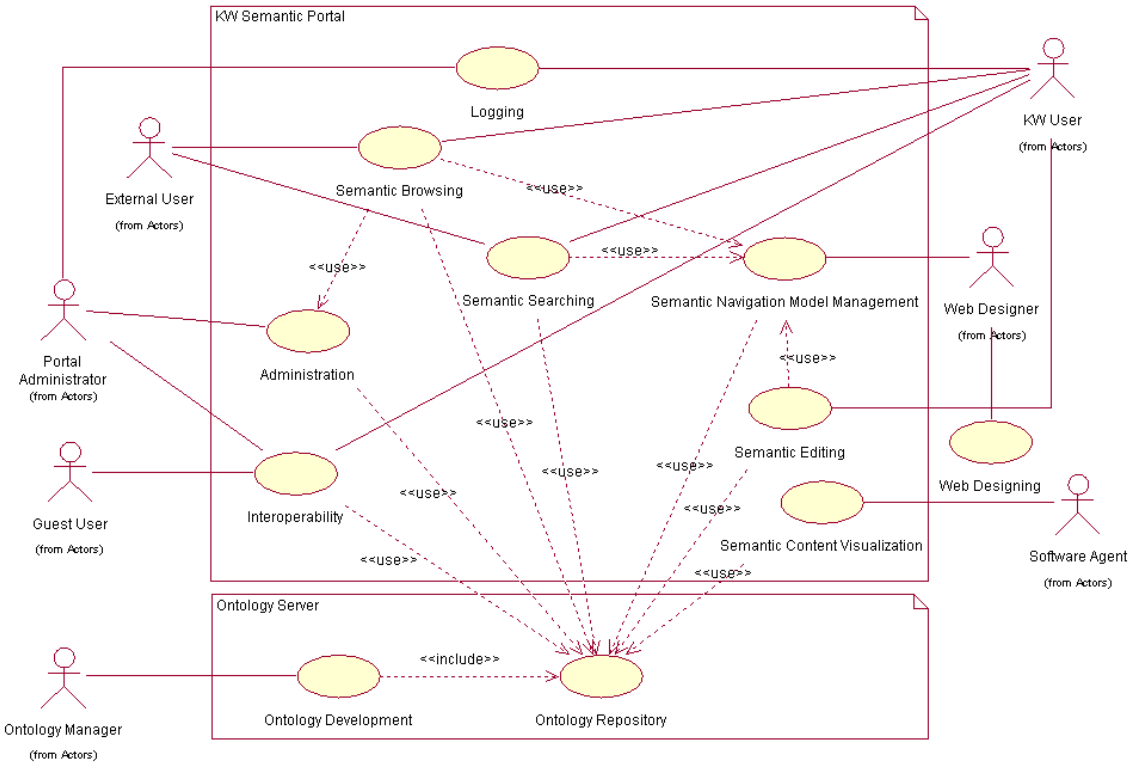

Figure 3-2 *KW semantic portal business use case diagram.*

Although it will be deeply explained below, here is a quick review of the business use cases depicted in the diagram.

On the top of the figure we can see the **Logging** use case. It provides the function of logging in the KW semantic portal, just for the *KW Users* and the *Portal Administrators*.

Under the *Logging* use case, we can see the **Semantic Browsing** use case. It consists of showing the list of ontology concepts and their related instances, presenting the details of ontology instances and their relations with other instances, and allowing the semantic navigation through these relations and between the different ontologies published in the portal. This business use case uses functions from other use cases:

- *Administration*, for retrieving/setting/updating the order of attributes, the instance definitions, the permissions, and the published ontologies.

- *Ontology Repository*, for importing RDF(S) and OWL ontologies, and for retrieving the ontologies conceptualization and instances.
- *Navigation Model Management*, for managing the navigation models.

Next to the *Semantic Browsing* use case, there is the **Semantic Searching** use case. It implements the semantic search engine that allows querying for information in one or in all the ontologies of the portal. As we can see in the figure, it can be accessed by an *External User* or a *KW User*, and is related to the *Ontology Repository* (to get the conceptualization and the instances) and to the *Semantic Navigation Model Management* (to get the search interfaces).

On the other side, there is the **Semantic Navigation Model Management** use case, which allows the *Web Designer* to manage the navigation models implemented in the KW semantic portal. It is only related with the *Ontology Repository* to retrieve the conceptualization and the instances in order to create those navigation models.

We also have the **Administration** use case, which refers to the management functions needed to maintain the KW semantic portal. They are: *User Management*, *Permission Management*, *Ontology Publish Management*, *Instance Description* and *Attribute Ordering*. It is only connected with the *Ontology Repository* to retrieve the ontologies needed in each case.

On the other side, we can see the **Semantic Editing** business use case, which consists of providing content to the KW semantic portal by allowing the *KW Users* to edit concept instances and the values of their attributes, and to connect such instances by means of relations, even if they belong to different ontologies. This business use case uses functions from the *Semantic Navigation Model Management* (to get the form interfaces) and the *Ontology Repository* (to create, edit and remove instances and relations).

On the bottom-right corner of our system, we have the **Semantic Content Visualization** use case, only accessed by *Software Agents*. This use case allows the agent to obtain the semantic visualization of a concept or instance in a certain semantic web language (OWL or RDF), for what it is connected with the *Ontology Repository*.

On the bottom-left corner, we have the **Interoperability** use case, which provides functions for exporting the portal content and importing resources into the portal (i.e, Ontoweb deliverables, FOAF, etc…). It is only connected with the *Ontology Repository* to update content while exporting or to retrieve resources while importing. Both the *KW User* and the *Guest User* are allowed to export content but only the *Portal Administrator* has permission for importing resources to the portal.

The *Web Designer* also communicates with the **Web Designing** use case. It allows him to manage the views of the portal. This is an external business use case, and therefore, it is not implemented in the ODESeW technology.

Finally, we have the *Ontology Server* system. This external system will provide functions to maintain and manage the ontologies needed by the KW semantic portal. The *Ontology Manager* will access the **Ontology Development** use case in order to manage the **Ontology Repository**.

3.4 Business Object Model

A Business Object Model is an object model that describes the realization of the project business use cases. It includes business use-case realizations, which show how the business use cases are "performed" in terms of interacting business workers and business entities.

Part of the business objects used by ODESeW proceeds from the *Ontology Repository*.

Here we will first describe the *Ontology Repository* business objects (concepts, attributes, instance attributes, instance value sets, instances, relatins and formulas) and then, those from the ODESeW portal (users profiles, permissions, navigation models, views and links).

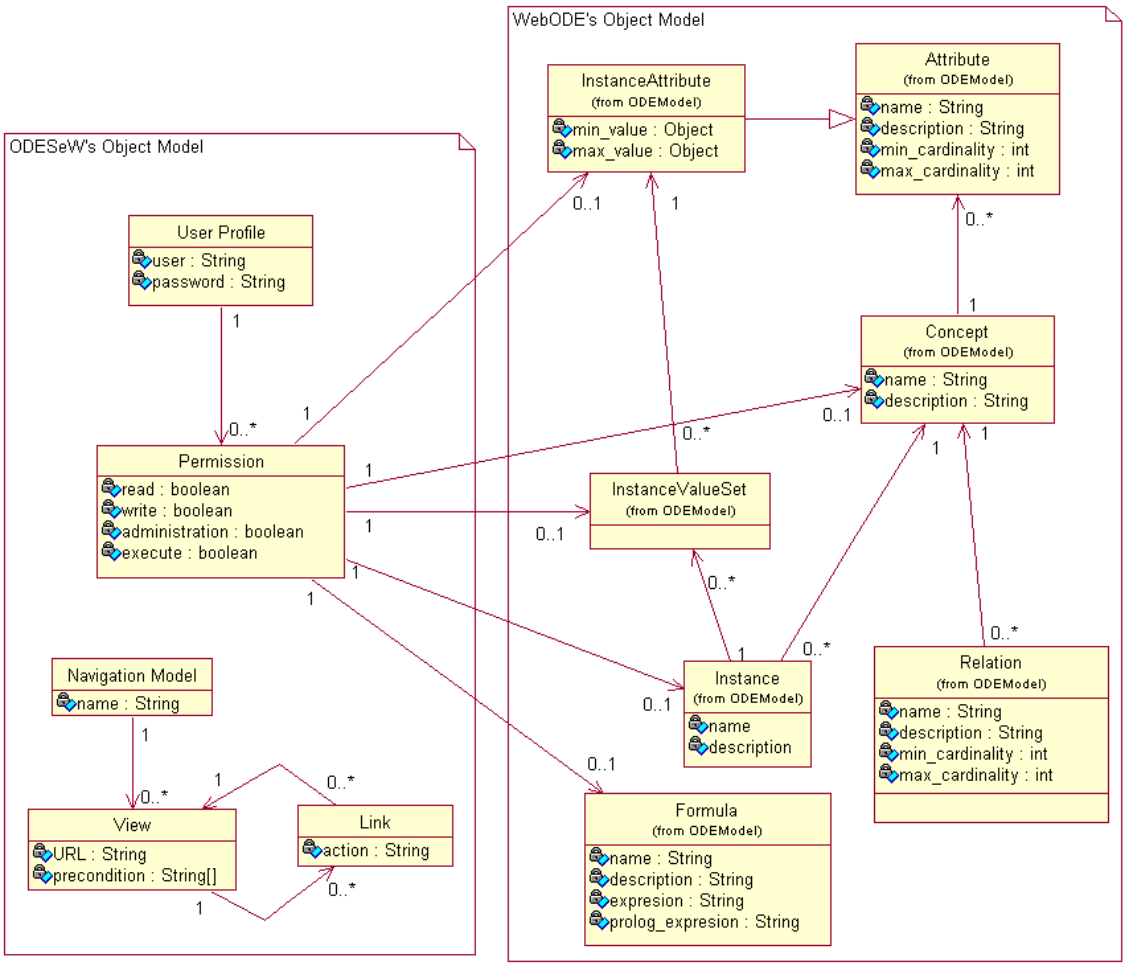

Figure 3-3 *KW Semantic Portal Business Object Model*

3.4.1 Ontology Repository Business Object Model

The *Ontology Repository* extern system is represented by the WebODE ontology development platform, so its business object model will be WebODE's.

WebODE's knowledge model [Arpirez et al., 2001] is extracted from the set of intermediate representations of METHONTOLOGY. AS we can see in *Figure 3-3*, it allows the representation of concepts and their attributes (both class and instance attributes), taxonomies of concepts, disjoint and exhaustive class partitions, ad-hoc binary relations between concepts, properties of relations, constants, axioms and instances. It also allows the inclusion of bibliographic references for any of them and the importation of terms from other ontologies.

Here is the description of all the objects depicted in this diagram:

- **Concept**. In short, a concept (also known as a class) can be anything about which something is said, and, therefore, can also be the description of a task, function, action, strategy, reasoning process, etc.

  Concepts are identified by their **name**. A natural language (NL) **description** can be also included. The same applies to **formulas**, which will be described later in this section.

- **Relation**. ODESeW allows just binary ad-hoc relations to be created between concepts. The creation of relations of higher arity must be made by reification (creating a concept for the relation itself and *n* binary relations between the concepts that appear in the relation and the concept that is used for representing the relation).

  Ad-hoc relations are characterized by their **name** and its **cardinality**, which establishes the number of facts (instances of the relation) that can hold between the origin and the destination term. Their cardinality can be restricted to 1 (only one fact) or N (any number of facts). Additionally, there is some optional information that can be provided for an ad-hoc relation, such as its **description**.

- **Instance attributes** are attributes whose value may be different for each instance of the concept. They have the same properties than class attributes and two additional properties, **minimum value** and **maximum value**, which are used in attributes with numeric value types.

- **Attribute.** This object represents a concept attribute whose value must be the same for all instances of the concept. They are not components themselves in ODESeW's knowledge model, as they are always attached to a concept (and to its subclasses, because of the inheritance mechanism).

  The information stored for an attribute is the following: its **name** (which must be different from the rest of attribute names of the same concept) and its **minimum and maximum cardinality**, which constrains the number of values that the class attribute may have.

- **Instance**. This object represent an element of a given concept. They have their own **name** and an optional **description**.

- **Instance Value Set**. This object represents a set of instance values.

Additionally, WebODE improves the reusability of ontologies defining sets of instances, which allow the instantiation of the same conceptual model for different scenarios it may be used for.

3.4.2 ODESeW Business Object Model

In this section we will see the business objects that compose the ODESeW Business Object Model. *Figure 3-3* depicts the diagram that describes them, including those objects provided by WebODE:

Here is the description of all the objects depicted in this diagram:

- **Permission**. Represents a permission of a certain user. There are four types of permissions. Therefore, this object has four different attributes: read, write, administration and execution. All of them are Booleans, so a *true* value will mean that the user has the permission and a *false* value will mean that the user does not have it.

- **User Profile**. Represents a user of the KW semantic portal. Each user has a name and a password to log in the portal, so this object will have these two attributes. This object is related to *Permission* because every user has 0 to 4 permissions.

- **Formula**. There are three types of formulas that can be created in ODESeW: axioms, rules and procedures. All of them are represented by their **name**, an optional **NL description,** a **formal expression** in first order logic, using a syntax provided by ODESeW, and finally, a **prolog expression**..

- **View**. This object represents a view of the portal, and it is contained in a navigation model. A view has two attributes: the **URL** associated with the view and the **precondition** needed to display it.

- **Navigation Model**. This object represents a navigation model of the ODESeW portal. Each navigation model is identified by a **name**, and this will be its only attribute. It is related with the *View* object because a navigation model is composed of a set of views.

- **Link**. This object represents the action that should execute in order to pass from a view to another. It has only one attribute, a string that identifies this **action**.

4 Requirement Analysis Model

In this section we will provide the detailed requirement specification of the ODESeW technology, according to the main groups of functions identified before. As we will see, each one of the business use cases identified in section 3.3 will be decomposed in several use cases.

Instead of providing separated sections for the system's functionality vision, we will integrate in this section the UML use case diagrams, the requirement specification, and the UML interfaces and sequence diagrams. For each main function (that is, for each subsystem) we will show a general use case diagram and its description. Each use case appearing in that general use case diagram will be described with more detail, decomposed in more use cases if proceeds, including its flow of events, its architectural implications and its contracts.

These are the sections that will describe the ODESeW business use cases: Ontology Repository (section 4.1), Administration (section 4.2), Logging (section 4.3), Semantic Editing (section 4.4), Semantic Browsing (section 4.5), Semantic Searching (section 4.6), Semantic Content Visualization (section 4.7), Interoperability (section 4.8), Semantic Navigation Model Management (section 4.9) and Web Designing (section 4.10). Although the Ontology Repository and Web Designing use cases do not belong to the KW semantic portal, we present here their structure in order to understand the rational for using some of their functionalities.

Here we present the structure followed in this section to describe all these use cases:

- **Ontology Repository**
- **Administration**
  - User Management
    - *Insert User*
    - *Remove User*
    - *Modify User*
  - Permission Management
    - *Reading Permission Management*
      - *Modify Instance Reading Permission*
      - *Modify Concept Reading Permission*
    - *Modify Instance Writing Permission*
  - Ontology Publish Management
    - *Add Ontology Publication*
    - *Remove Ontology Publication*
  - Atribute Ordering
    - *Set Order Of Attributes*
  - Instance Description
    - *Set Instance Description*
- **Logging**
- **Semantic Editing**
  - Instance Creation
  - Instance Editing
  - Instance Removal
- **Semantic Browsing**
  - Semantic Navigation
  - Semantic Visualization
- **Semantic Searching**
  - Search In Term Names
  - Search In Instance Values
- **Semantic Content Visualization**
  - Content Generation in Semantic Web Languages
    - OWL Translation
      - *OWL Ontology Translation*
      - *OWL Instance Translation*
    - RDF(S) Translation
      - *RDFS Ontology Translation*
      - *RDF Instance Translation*
- **Interoperability**
  - Import Resource
  - Export Content
- **Semantic Navigation Model Management**
- **Web Designing**

For each one of the use cases contained in a business use case, we will structure as follows:

- Description of the use case, its functionalities and the actors involved in it.
- Flow of events of the use case and Sequence Diagram.
- Architectural Implications of the use case.
- Use case contracts. There will be one contract for each operation of the sequence diagram.

4.1 Ontology Repository

Description
The *Ontology Repository* represents the external system that will maintain all the ontologies managed by ODESeW and all its instantiations. *Figure 4-1* shows the decomposition of the *Ontology Repository* business use case.

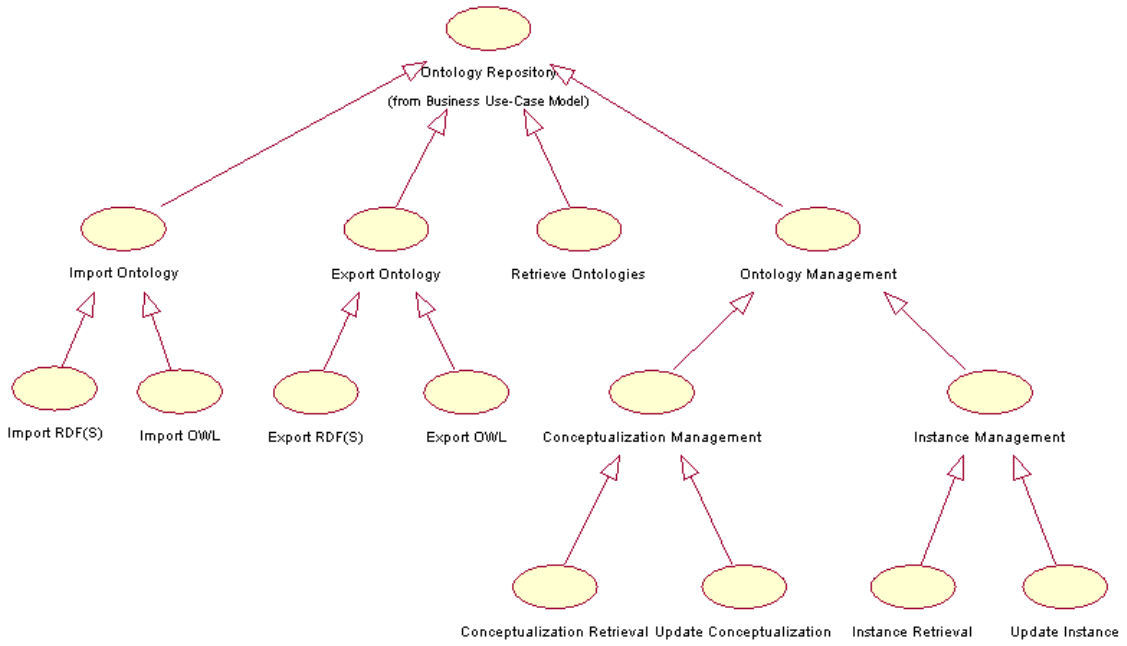

Figure 4-1 *Use case diagram of the Ontology Repository use case.*

We can see that the *Ontology Repository* provides four different functions. First, there is the *Import Ontology* use case, which breaks down in two: *Import RDF(S)* and *Import OWL*. In a similar way, we have the *Export Ontology* use case, which breaks down in another two: *Export RDF(S)* and *Export OWL*.

Then, we have the *Retrieve Ontologies* use case, which allows our system to get the ontologies implemented in the *Ontology Repository*.

Finally, *Ontology Management*, which breaks down in two more use cases: *Conceptualization Management* and *Instance Management*. The first one is divided into: *Conceptualization Retrieval* and *Update Conceptualization*. The *Instance Management*, at the same time, breaks down into two more use cases: *Instance Retrieval* and *Update Instance*.

4.2 Administration

Description
The *Administration* use cases refer to the management functions needed to maintain a semantic portal built with ODESeW. *Figure 4-2* describes all the actors interacting with the *Administration* business use case, as well as the use cases involved in it. The only actor is the *Portal Administrator*. The six functions related to the *Administration* system are: *User Management*, *Permission Management*, *Ontology Publish Management*, *Attribute Ordering* and *Instance Description*. They are explained in sections 4.2.1 to 4.2.5, respectively.

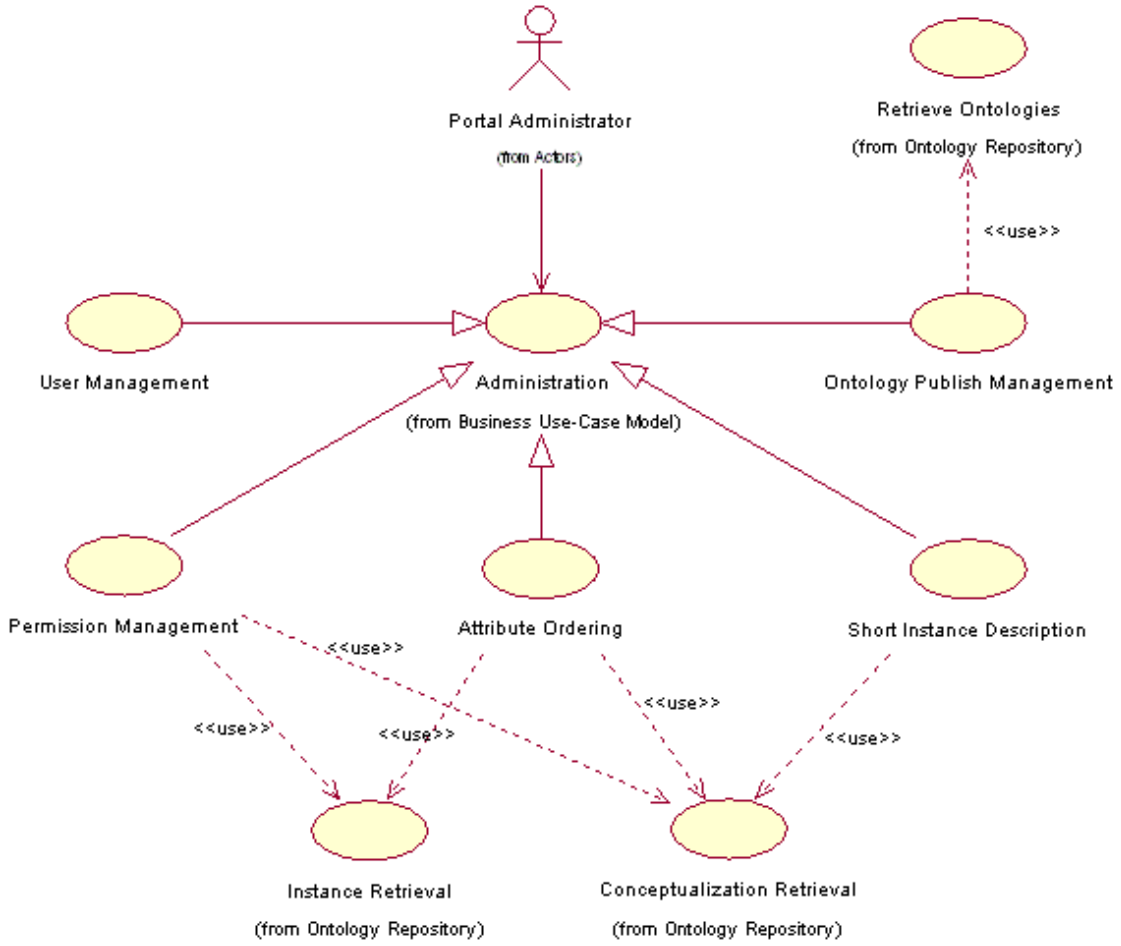

Figure 4-2 *Use case diagram of the Administration use case.*

4.2.1 User Management

Description

With the *User Management* use case, the *Administrator* can insert, remove or modify different types of users. *Figure 4-3* shows its decomposition. As we can see, there are three operations related with this use case: *Insert User*, *Remove User* and *Modify User*.

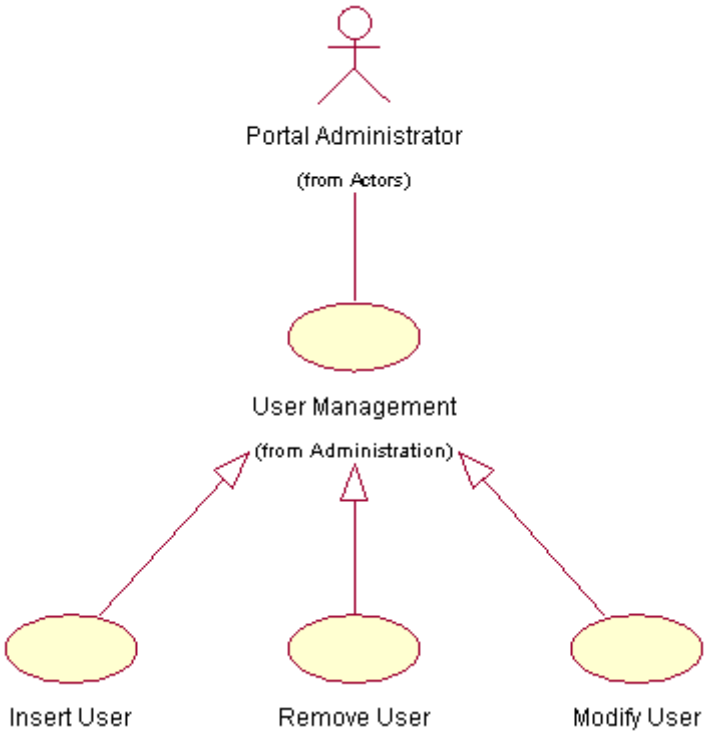

Figure 4-3 *Use case diagram of the User Management use case.*

4.2.1.1 Insert User

Description

With this use case, the *Portal Administrator* can insert a new user in the KW semantic portal using ODESeW technology.

Flow of events

The flow of events of this use case is shown in *Figure 4-4*. The *Portal Administrator* requests to insert a new user and the KW portal shows him the corresponding form. Now, what (s)he has to do is fill it with the data of the new user.

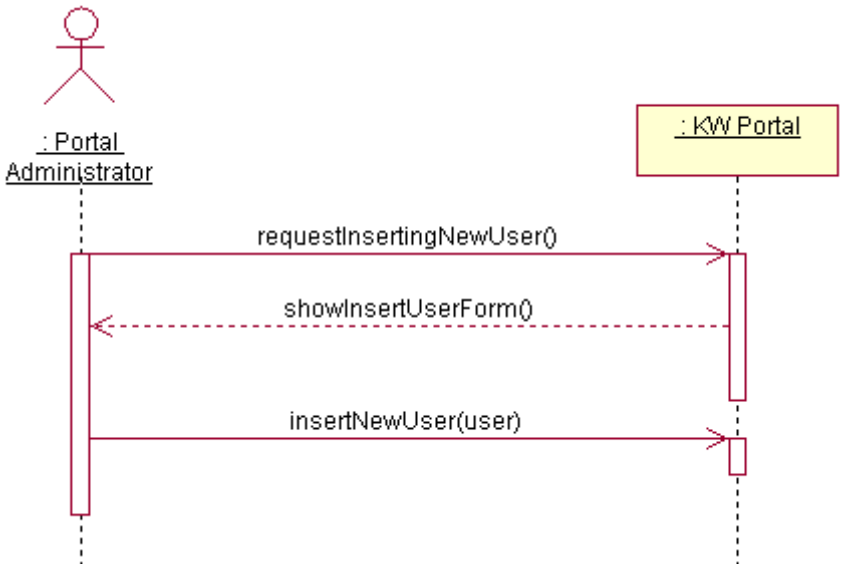

Figure 4-4 *Sequence diagram of the Insert User use case.*

Architectural Implications: None

Contracts

Name:	requestInsertingNewUser()
Responsibilities:	Initiates the use case *Insert User*.
Crossed References:	Use case *Insert User*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The use case *Insert User* has been initiated.

Name:	showInsertUserForm()
Responsibilities:	Shows the form in blank to be filled with the information of a new user.
Crossed References:	Use case *Insert User*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The form in blank to be filled with the information of a new user has been shown to the user.

Name:	insertNewUser(user)
Responsibilities:	Adds a new user of the KW portal with the information given.
Crossed References:	Use case *Insert User*.
Notes:	
Exceptions:	If there is already a user with that name, show a message explaining the error.
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	There is a new user in the KW portal with the information contained in *user*.

4.2.1.2 Remove User

Description
This use case allows the *Portal Administrator* to remove a user from the KW semantic portal.
Flow of events
The flow of events of this use case is quite simple, as we can see in *Figure 4-5*. After requesting the removal of a user, the KW portal shows the *Administrator* the list of users, and (s)he can now select the one to be removed in the KW portal.

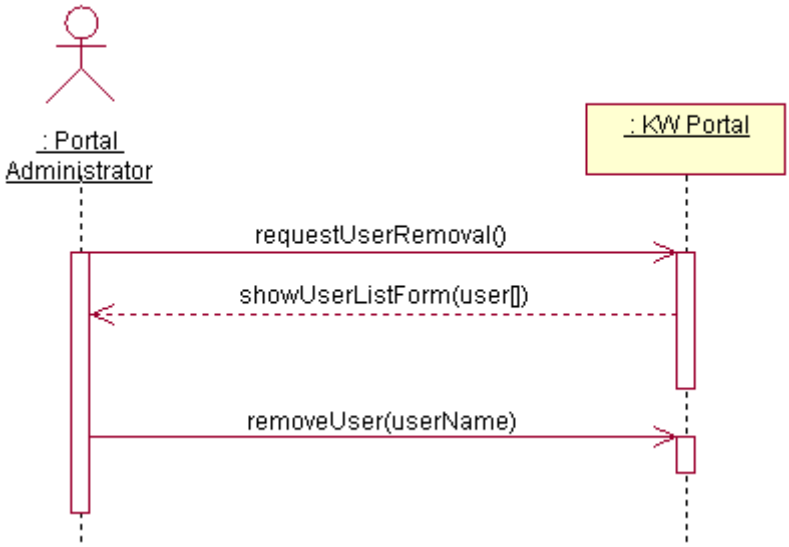

Figure 4-5 *Sequence diagram of the Remove User use case.*

Architectural Implications: None

Contracts

Name:	requestUserRemoval()
Responsibilities:	Initiates the use case *Remove User*.
Crossed References:	Use case *Remove User*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The use case *Remove User* has been initiated.

Name:	showUserListForm(user[])
Responsibilities:	Shows the list of all the KW portal users.
Crossed References:	Use case *Modify User*. Use case *Remove User*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The list containing all the users of the KW portal has been shown to the user. If the list is empty, a message will be shown explaining that there are no users registered in the KW portal.

Name:	removeUser(userName)
Responsibilities:	Removes a user from the KW portal.
Crossed References:	Use case *Remove User*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The user called *userName* is no longer in the KW portal.

4.2.1.3 Modify User

Description
With this use case the user is able to modify the information of any user registered in the KW semantic portal.
Flow of events
The flow of events of this use case is quite simple, as we see in *Figure 4-6*. After requesting the removal of a user, the KW portal shows the *Portal Administrator* the list of users, and (s)he can now select the one to be removed.

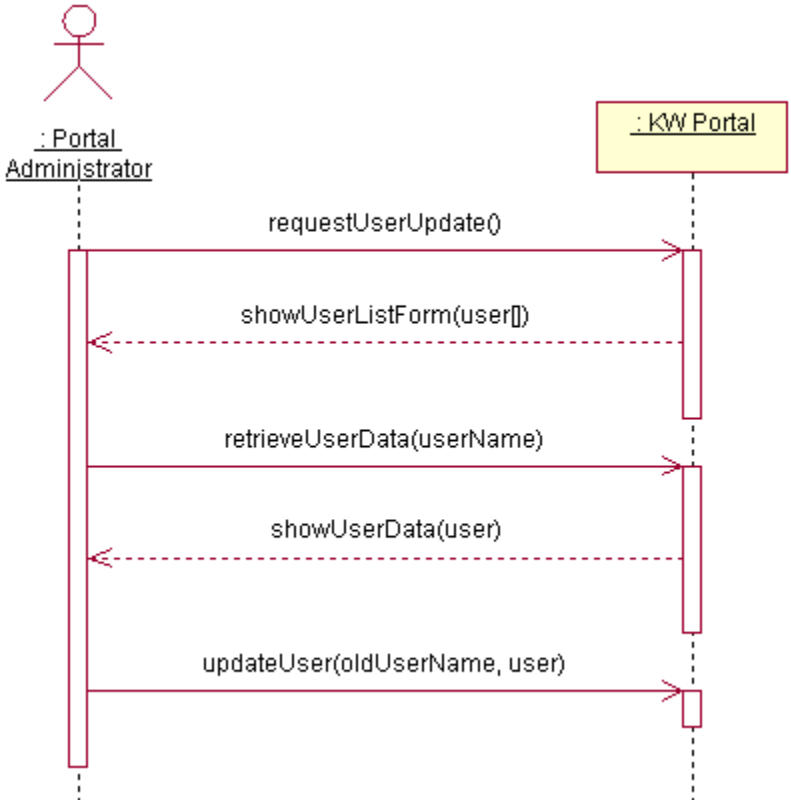

Figure 4-6 *Sequence diagram of the Modify User use case.*

Architectural Implications: None

Contracts

Name:	requestUserUpdate()
Responsibilities:	Initiates the use case *Modify User*.
Crossed References:	Use case *Modify User*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The use case *Modify User* has been initiated.

Name:	showUserListForm(user[])
Responsibilities:	Shows the list of all the KW portal users.
Crossed References:	Use case *Modify User*. Use case *Remove User*.
Notes:	
Exceptions:	If the list is empty, show a message explaining that there are no users registered in the KW portal.
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The list containing all the users of the KW portal has been shown to the user.

Name:	retrieveUserData(userName)
Responsibilities:	Requests to the KW portal the data of a certain user.
Crossed References:	Use case *Modify User*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The data of the user called *userName* has been requested to the KW portal.

Name:	showUserData(user)
Responsibilities:	Shows the information of a certain user of the KW portal.
Crossed References:	Use case *Modify User*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The information of a certain user has been shown to the user.

Name:	updateUser(oldUserName, user)
Responsibilities:	Updates the information of a user, according to the data given. The first argument represents the current name of the user to be modified and the second contains its new values, which could contain a new name.
Crossed References:	Use case *Modify User*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The information of the user called *oldUserName* has been modified in the KW portal according to the values contained in *user*.

4.2.2 Permission Management

Description

With the *Permission Management* use case the *Portal Administrator* can manage the read and write permissions for each user, including the *Guest Users*. *Figure 4-7* shows the decomposition of the *Permission Management* use case.

First of all, we have the *Retrieve Permissions* use case, which is included in the two other use cases. Then, there is the *Reading Permission Management*, which breaks down in two: *Modify Instance Reading Permission* and *Modify Concept Reading Permission*. The first one allows the *Portal Administrator* to decide whether a user can visualize an instance or some attributes of the instance. The second one allows the user to decide whether a user can visualize a concept (and its instances) or some attributes of the concept.

Finally, we have the *Modify Instance Writing Permission*, with which the *Portal Administrator* decides the users allowed to insert, modify and remove instances of a concept.

As we can see in *Figure 4-2*, the *Permission Management* use case retrieves the instances and concepts by means of two use cases from the *Ontology Repository*. These are *Instance Retrieval* and *Conceptualization Retrieval*.

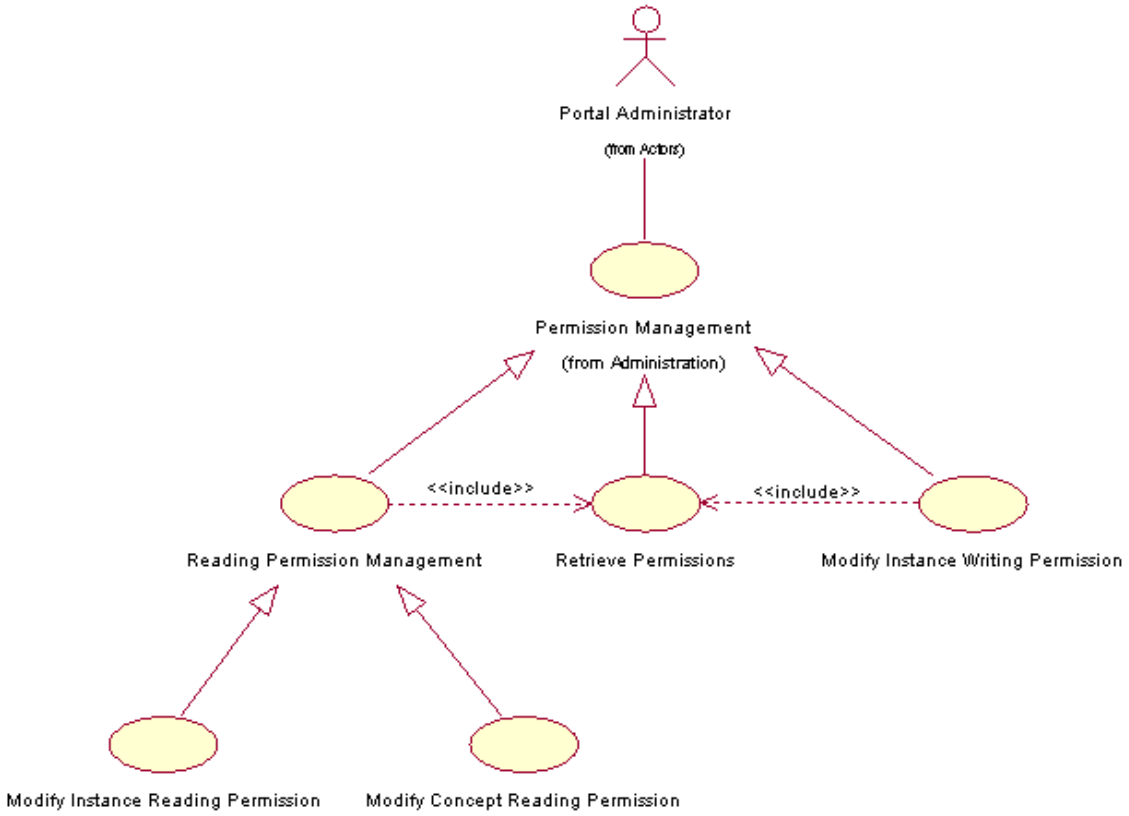

Figure 4-7 *Use case diagram of the Permission Management use case.*

4.2.2.1 Reading Permission Management

4.2.2.1.1 Modify Instance Reading Permission

Description

With this use case the *Portal Administrator* can modify the reading permissions of an instance.

Flow of events

The flow of events of this use case is shown in *Figure 4-8*. When the *Portal Administrator* requests to modify the reading permissions of an instance, the KW portal accesses the *Ontology Server*, gets the list of instances and shows it to the user. Then, the administrator selects an instance from the list, and the KW portal gets its attributes from the *Ontology Server* and shows them to the administrator. (S)he can now update the reading permissions of that instance on his own.

It is important to mention here that we have followed a certain nomenclature in all the sequence diagrams of the document. If a contract does not have any nested calls to other systems, the answer is implicit. This way, for instance, when the portal requests the list of concepts to the *Ontology Server*, the variable *concept[]* represents the data returned by the *Ontology Server* to the *KW Portal*.

Figure 4-8 *Sequence diagram of the Modify Instance Reading Permission use case.*

Architectural Implications: None

Contracts

Name:	modifyInstanceReadingPermission(ontology)
Responsibilities:	Initiates the use case *Modify Instance Visualization Permission*.
Crossed References:	Use case *Modify Instance Reading Permission*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The use case *Modify Instance Visualization Permission* has been initiated.

Name:	concept[] := getConceptList(ontology)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the list containing the names of the concepts of all the ontologies implemented in the *Ontology Server*.
Crossed References:	Use case *Set Order Of Attributes*. Use case *Set Instance Description*. Use case *Modify Concept Reading Permission*. Use case *Modify Instance Reading Permission*. Use case *Modify Instance Writing Permission*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The variable *concept[]* contains the list of the names of the concepts of all the ontologies of the KW portal.

Name:	showConceptListForm(concept[])
Responsibilities:	Shows to the user the form containing the list of the concepts appearing in a list.
Crossed References:	Use case *Set Order Of Attributes*. Use case *Set Instance Description*. Use case *Modify Concept Reading Permission*. Use case *Modify Instance Reading Permission.* Use case *Modify Instance Writing Permission.*
Notes:	
Exceptions:	If the list is empty, show a message explaining that there are no concepts in the ontology.
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The list of the concepts contained in *the variable* has been shown to the user.

Name:	retrieveConceptInstances(concept)
Responsibilities:	Requests to the KW portal the instances of a concept.
Crossed References:	Use case *Modify Instance Reading Permission.*
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The instances of *concept* have been requested to the KW portal.

Name:	instance[] := getConceptInstances(concept)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the list of the instances of a concept.
Crossed References:	Use case *Modify Instance Reading Permission.*
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The variable returned contains all the instances of the concept *concept*.

Name:	showConceptInstances(instance[])
Responsibilities:	Shows the list of the instances of a concept.
Crossed References:	Use case *Modify Instance Reading Permission.*
Notes:	
Exceptions:	If the list is empty, show a message explaining that the concept has no instances.
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The list of the instances contained in the variable has been shown to the user.

Name:	retrieveInstanceAttributes(instance)
Responsibilities:	Requests to the KW portal the attributes of a certain instance.
Crossed References:	Use case *Modify Instance Reading Permission.*
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The attributes of *instance* have been requested to the KW semantic portal.

Name:	instanceAtt[] := getInstanceAttributes(concept)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the list of attributes of any instance of a certain concept.
Crossed References:	Use case *Modify Instance Reading Permission*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The variable *instanceAtt[]* contains the list of attributes of the concept *concept*.

Name:	showInstanceReadingPermissionForm(instanceAtt[], permission[])
Responsibilities:	Shows the reading permissions of an instance, as well as the list of its attributes, which can be individually restricted by the user.
Crossed References:	Use case *Modify Instance Reading Permission.*
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The reading permissions of an instance (contained in *permission[]*) and the list of its attributes (contained in *instanceAtt[]*) have been shown to the user.

Name:	updateInstanceReadingPermission(instance, permission[])
Responsibilities:	Updates in the KW portal the reading permissions of a certain instance.
Crossed References:	Use case *Modify Instance Reading Permission.*
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The permissions of the instance has been updated according to the variable *permission[]*.

4.2.2.1.2 Modify Concept Reading Permission

Description

With this use case the *Portal Administrator* can modify the reading permissions of a concept.

Flow of events

The flow of events of this use case is shown in *Figure 4-9*. When the *Portal Administrator* requests to modify the reading permissions of a concept, the KW portal accesses the *Ontology Server*, gets the list of concepts and shows it to the user. When the *Portal Administrator* selects a concept from the list, the KW portal shows its reading permissions to the user. Now, (s)he can update the reading permissions of that concepts on his own.

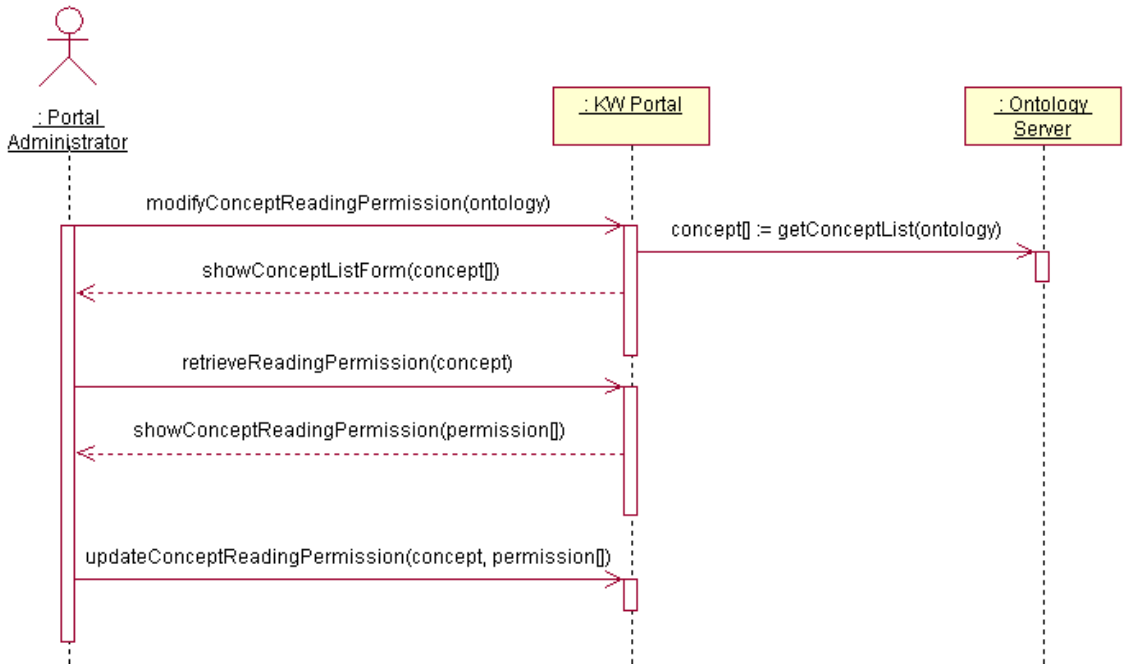

Figure 4-9 *Sequence diagram of the Modify Concept Reading Permission use case.*

Architectural Implications: None

Contracts

Name:	modifyConceptReadingPermission(ontology)
Responsibilities:	Initiates the use case *Modify Concept Reading Permission.*
Crossed References:	Use case *Modify Concept Reading Permission.*
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The use case *Modify Concept Reading Permission* has been initiated.

Name:	concept[] := getConceptList(ontology)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the list containing the names of the concepts of a certain ontology of the *Ontology Server*.
Crossed References:	Use case *Set Order Of Attributes.* Use case *Set Instance Description*. Use case *Modify Concept Reading Permission.* Use case *Modify Instance Reading Permission.* Use case *Modify Instance Writing Permission.*
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The variable *concept[]* contains the list of the names of the concepts of all the ontologies of the KW portal.

Name:	showConceptListForm(concept[])
Responsibilities:	Shows to the user the form containing the list of the concepts appearing in a list.
Crossed References:	Use case *Set Order Of Attributes.* Use case *Set Instance Description*.

	Use case *Modify Concept Reading Permission*. Use case *Modify Instance Reading Permission*. Use case *Modify Instance Writing Permission*.
Notes:	
Exceptions:	If the list is empty, show a message explaining that there are no concepts in the ontology.
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The list of the concepts contained in the variable has been shown to the user.

Name:	retrieveReadingPermission(concept)
Responsibilities:	Requests to the KW portal the reading permissions of a concept.
Crossed References:	Use case *Modify Concept Reading Permission*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The reading permissions of concept have been requested to the KW portal.

Name:	showConceptReadingPermission(permission[])
Responsibilities:	Shows the current reading permissions of a concept.
Crossed References:	Use case *Modify Concept Reading Permission*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The reading permissions of a concept, contained in the variable *permission[]*, have been shown to the user.

Name:	updateConceptReadingPermission(concept, permission[])
Responsibilities:	Updates in the KW portal the reading permissions of a concept.
Crossed References:	Use case *Modify Concept Reading Permission*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The reading permissions of *concept* have been updated according to the variable *permission[]*.

4.2.2.2 Modify Instance Writing Permission

Description
With this use case the *Portal Administrator* can modify the writing permissions of an instance.
Flow of events
We can see the flow of events of this use case in *Figure 4-10*. First, the *Portal Administrator* requests to modify the writing permissions of the instances of a concept. Then, the KW portal gets the list of concepts from the *Ontology Server*, and shows it to the *Portal Administrator*, from which (s)he can now select one of them and update its writing permissions on his own.

Figure 4-10 *Sequence diagram of the Modify Instance Writing Permission use case.*

Architectural Implications: None

Contracts

Name:	modifyInstanceWritingPermission(ontology)
Responsibilities:	Initiates the use case *Modify Instance Writing Permission*.
Crossed References:	Use case *Modify Instance Writing Permission*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The use case *Modify Instance Writing Permission* has been initiated.

Name:	concept[] := getConceptList(ontology)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the list containing the names of the concepts of a certain ontology of the *Ontology Server*.
Crossed References:	Use case *Set Order Of Attributes*. Use case *Set Instance Description*. Use case *Modify Concept Reading Permission*. Use case *Modify Instance Reading Permission*. Use case *Modify Instance Writing Permission*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The variable *concept[]* contains the list of the names of the concepts of all the ontologies of the KW portal.

Name:	showConceptListForm(concept[])
Responsibilities:	Shows to the user the form containing the list of the concepts appearing in a list.
Crossed References:	Use case *Set Order Of Attributes*. Use case *Set Instance Description*. Use case *Modify Concept Reading Permission*. Use case *Modify Instance Reading Permission.* Use case *Modify Instance Writing Permission.*
Notes:	
Exceptions:	If the list is empty, show a message explaining that there are no concepts in the ontology.
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The list of the concepts contained in the variable has been shown to the user.

Name:	retrieveWritingPermission(concept)
Responsibilities:	Requests to the KW portal the writing permissions of the instances of a concept.
Crossed References:	Use case *Modify Instance Writing Permission.*
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The writing permissions of the instances of *concept* have been requested to the KW portal.

Name:	showConceptWritingPermissions(allowedUser[], user[])
Responsibilities:	Shows the current list of the users with permission to update concepts and another list containing the rest of the users.
Crossed References:	Use case *Modify Instance Writing Permission.*
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The two lists of users (*allowedUse[]* and *user[]*) have been shown to the user.

Name:	updateInstanceWritingPermissions(concept, user[])
Responsibilities:	Updates in the KW portal the list of the users with editing permission on the instances of a certain concept.
Crossed References:	Use case *Modify Instance Writing Permission.*
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The list of the users with updating permission on the instances of *concept* has been updated according to the variable *user[]*.

4.2.3 Ontology Publish Management

Description

With the *Ontology Publish Management*, the *Portal Administrator* decides which ontologies are published in the KW semantic portal. The diagram in *Figure 4-11* describes the three operations related to the *Ontology Publish Management*: *Add Ontology Publication*, *Remove Ontology Publication*. They are respectively explained in sections 4.2.3.1 and 4.2.3.2.

As we saw in *Figure 4-2*, the *Ontology Publish Management* takes the set of ontologies from the *Ontology Repository* by means of the *Retrieve Ontologies* use case.

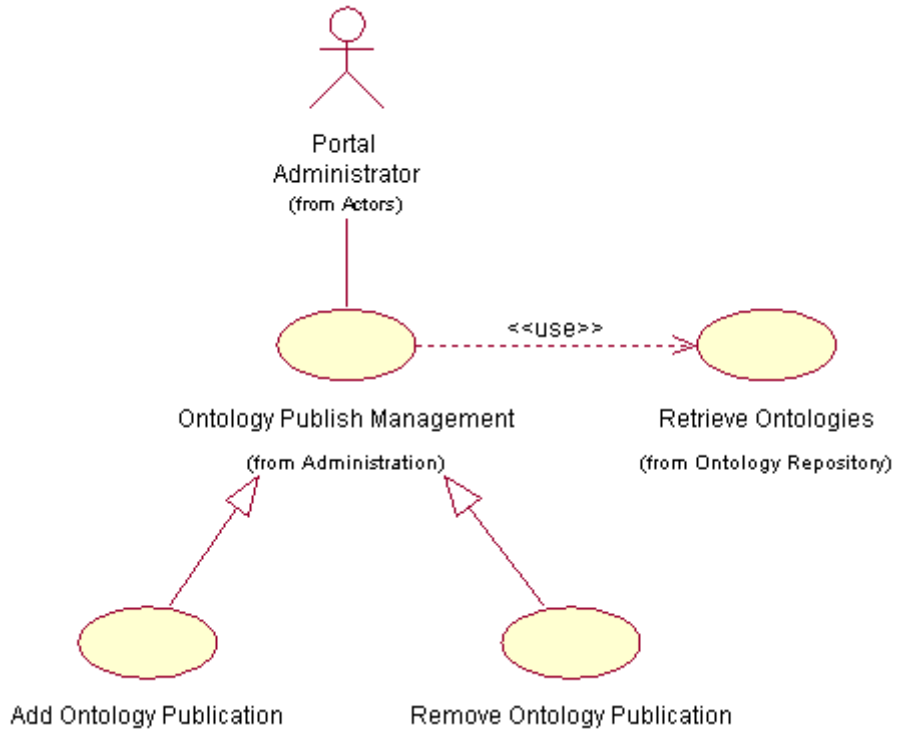

Figure 4-11 *Use case diagram of the Ontology Publish Management use case.*

4.2.3.1 Add Ontology Publication

Description

This use case allows the *Portal Administrator* to publish an ontology in the KW portal.

Flow of events

The flow of events of this use case is shown in *Figure 4-12*. When the *Portal Administrator* requests to publish another ontology, the KW portal must get the list of ontologies from the *Ontology Server*, and show it to the user. Then the *Administrator* may select which ontology wants to be published on the portal.

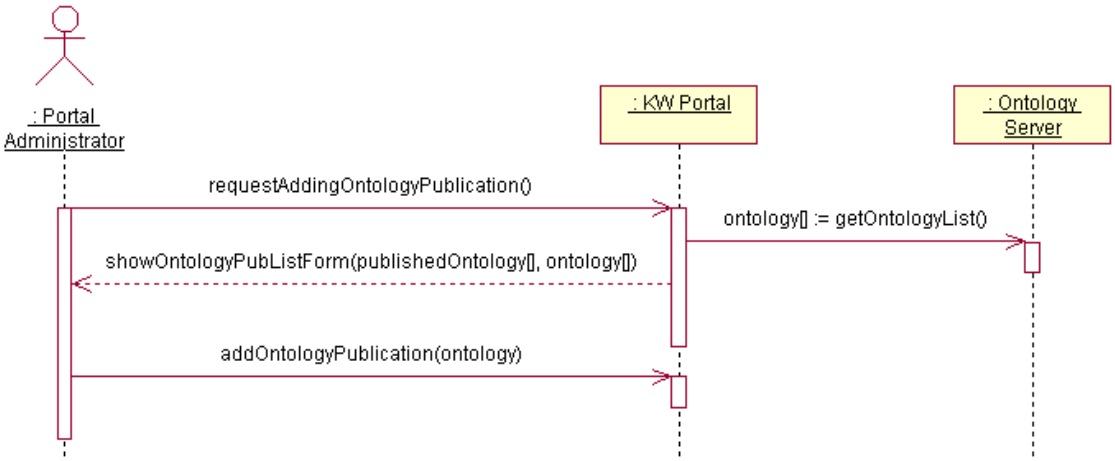

Figure 4-12 *Sequence diagram of the Add Ontology Publication use case.*

Architectural Implications: None

Contracts

Name:	requestAddingOntologyPublication()
Responsibilities:	Initiates the use case *Add Ontology Publication*.
Crossed References:	Use case *Add Ontology Publication*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The use case *Add Ontology Publication* has been initiated.

Name:	ontology[] := getOntologyList()
Responsibilities:	Obtains (from the *Ontology Server*) and returns the list containing the names of all the ontologies implemented in the *Ontology Server*.
Crossed References:	Use case *Add Ontology Publication*. Use case *Remove Ontology Publication*. Use case *Search In Instance Values.*
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The variable *ontology[]* contains the list of the names of all the ontologies of the *Ontology Server*.

Name:	showOntologyPubListForm(publishedOntology[], ontology[])
Responsibilities:	Shows the form with the names of the ontologies currently published in the portal and another list with the names of the rest of the ontologies of the *Ontology Server*.
Crossed References:	Use case *Add Ontology Publication*. Use case *Remove Ontology Publication*.
Notes:	
Exceptions:	If there are no ontologies in the list, show a message explaining that there are no ontologies in the *Ontology Server*.
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	Both lists have been shown to the user.

Name:	addOntologyPublication(ontology)
Responsibilities:	Adds the publication of an ontology in the KW portal.
Crossed References:	Use case *Add Ontology Publication*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The ontology *ontology* has been published in the KW portal.

4.2.3.2 Remove Ontology Publication

Description

This use case allows the *Portal Administrator* to remove the publication of an ontology from the KW portal.

Flow of events

Figure 4-13 describes the flow of events of this use case. It is quite similar to the *Add Ontology Publication* flow. First, the *Portal Administrator* request to remove an ontology publication. Then, the KW portal obtains the list of ontologies from the *Ontology Server* and shows it to the user. Finally, this one selects the ontology whose publication is to be removed.

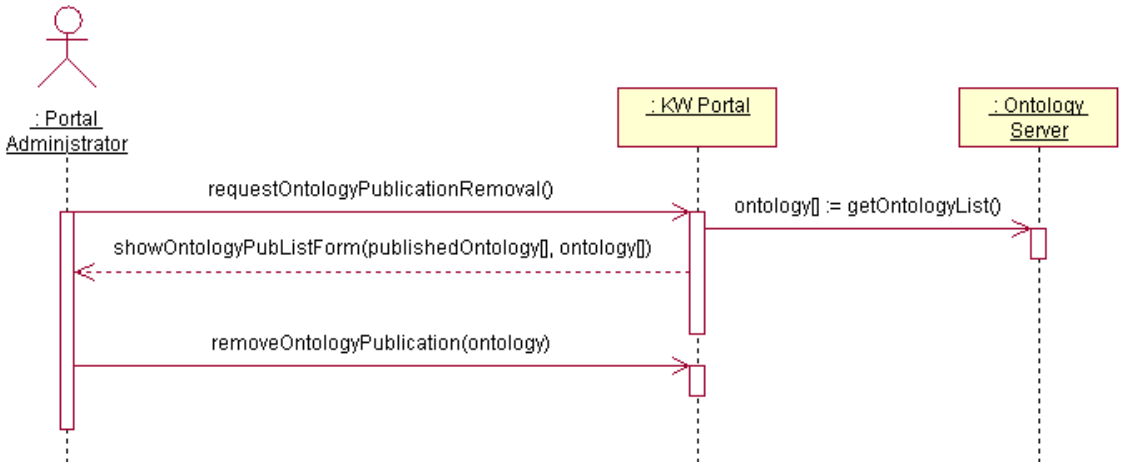

Figure 4-13 *Sequence diagram of the Remove Ontology Publication use case.*

Architectural Implications: None

Contracts

Name:	requestOntologyPublicationRemoval()
Responsibilities:	Initiates the use case *Remove Ontology Publication*.
Crossed References:	Use case *Remove Ontology Publication*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The use case *Remove Ontology Publication* has been initiated.

Name:	ontology[] := getOntologyList()
Responsibilities:	Obtains (from the *Ontology Server*) and returns the list containing the names of all the ontologies implemented in the *Ontology Server*.
Crossed References:	Use case *Add Ontology Publication*. Use case *Remove Ontology Publication*. Use case *Search In Instance Values.*
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The variable *ontology[]* contains the list of the names of all the ontologies of the *Ontology Server*.

Name:	showOntologyPubListForm(publishedOntology[], ontology[])
Responsibilities:	Shows the form with the names of the ontologies currently published in the portal and another list with the names of the rest of the ontologies of the *Ontology Server*.
Crossed References:	Use case *Add Ontology Publication*. Use case *Remove Ontology Publication*.
Notes:	
Exceptions:	If there are no ontologies in the list, show a message explaining that there are no ontologies in the *Ontology Server*.
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	Both lists have been shown to the user.

Name:	removePublishedOntology(ontology)
Responsibilities:	Removes the publication of an ontology in the KW portal.
Crossed References:	Use case *Remove Ontology Publication*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The ontology *ontology* is no longer published in the KW portal.

4.2.4 Attribute Ordering

Description

With this use case, the *Portal Administrator* can set, for each concept, the order in which the attributes of all its instances will be visualized. Once the *Administrator* has set the order of the attributes of a concept, he can impose this order to the subclasses of the concept.

In *Figure 4-14* we can see that the *Attribute Ordering* use case breaks down in two: *Set Order Of Attributes* and *Retrieve Order Of Attributes*. There is also a use case that extends the first one: *Impose Order To Children*.

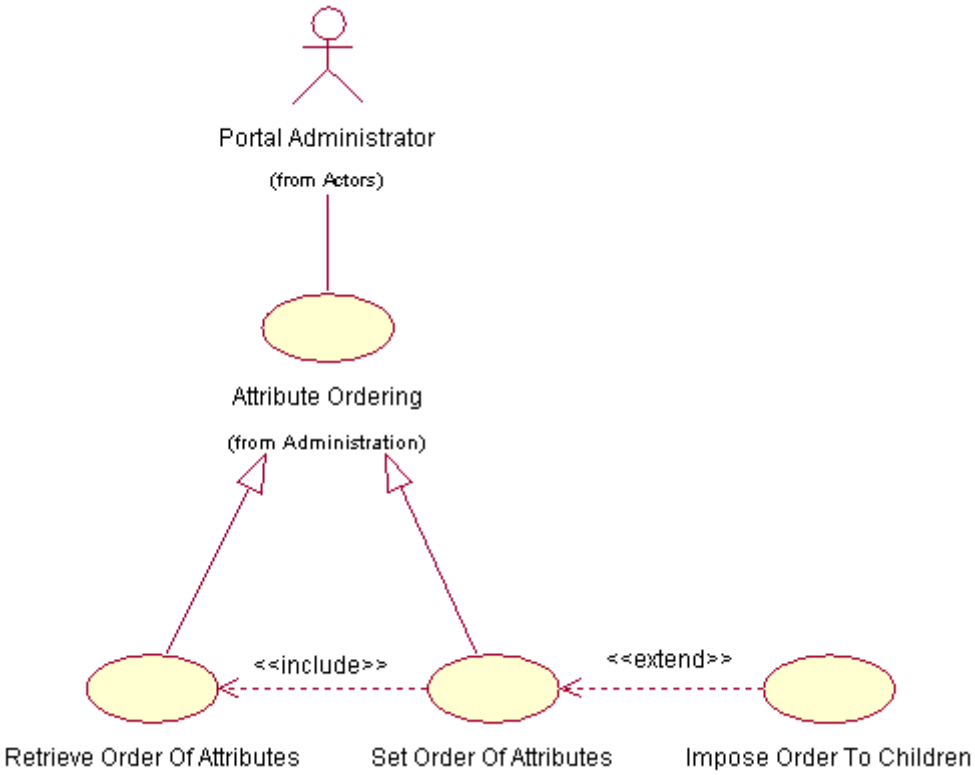

Figure 4-14 *Use case diagram of the Attribute Ordering use case.*

As we can see in *Figure 4-2*, the *Attribute Ordering* use case retrieves the instances and concepts by means of two use cases from the *Ontology Repository*. These are *Instance Retrieval* and *Conceptualization Retrieval*.

4.2.4.1 Set Order Of Attributes

Flow of events

The flow of events of this use case is represented by *Figure 4-15*. There we can see that when the *Portal Administrator* decides to set an order to the attributes of any concept, the portal has to get the concept taxonomy from the *Ontology Server*, and show it to the user. Then, this one selects the concept, and the KW semantic portal will access again the *Ontology Server* in order to obtain the attributes of the concept. Finally, this information will be shown to the user, and (s)he now may update the order of the attributes on his (her) own.

At the bottom of the diagram we can see another operation, called *imposeOrderToChildren*. It represents the use case that extends the *Set Order Of Attributes* use case, as we saw in *Figure 4-14*. The *Portal Administrator* gives the name of the concept and the ordered list of attributes to the KW semantic portal, and this one deals with applying that order to the children of the concept.

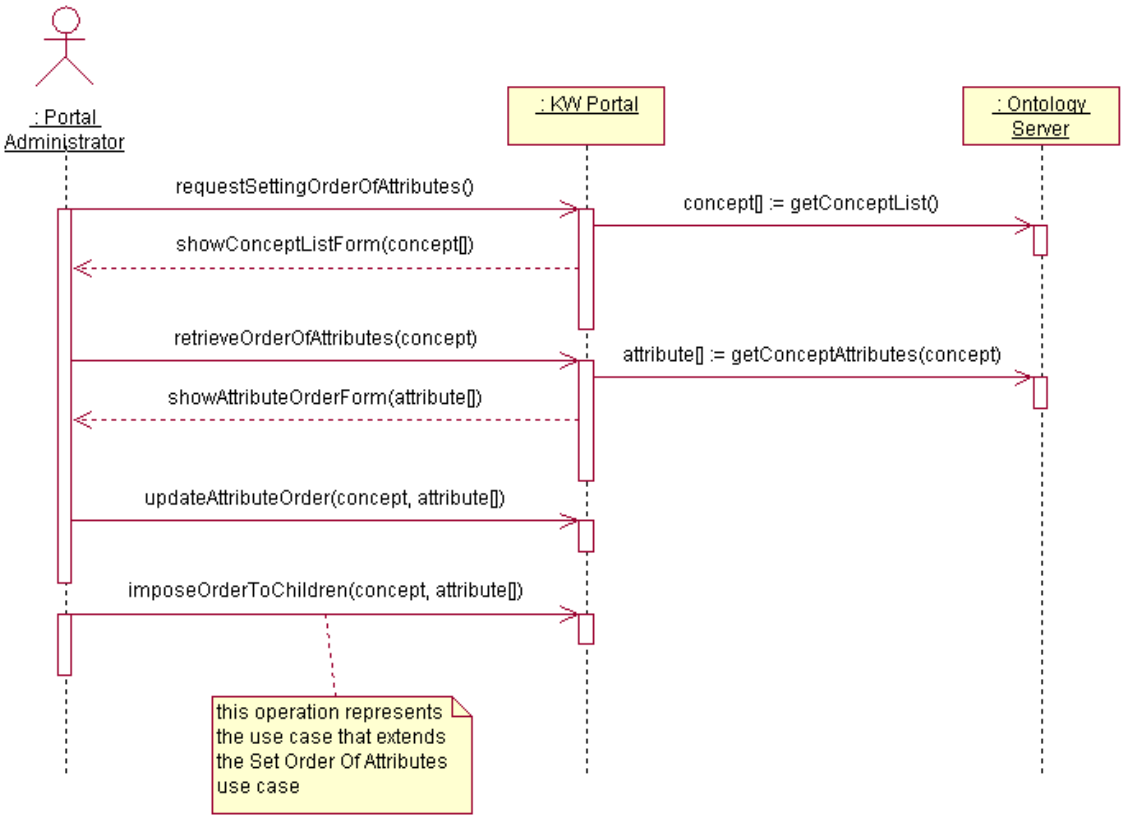

Figure 4-15 *Sequence diagram of the Set Order Of Attributes use case.*

Architectural Implications: None

Contracts

Name:	requestSettingOrderOfAttributes()
Responsibilities:	Initiates the use case *Set Order Of Attributes*.
Crossed References:	Use case *Set Order Of Attributes*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The use case *Set Order Of Attributes* has been initiated.

Name:	concept[] := getConceptList(ontology)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the list containing the names of the concepts of a certain ontology of the *Ontology Server*.
Crossed References:	Use case *Set Order Of Attributes*. Use case *Set Instance Description*. Use case *Modify Concept Reading Permission*. Use case *Modify Instance Reading Permission*. Use case *Modify Instance Writing Permission*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The variable *concept[]* contains the list of the names of the concepts of all the ontologies of the KW portal.

Name:	showConceptListForm(concept[])
Responsibilities:	Shows to the user the form containing the list of the concepts appearing in a list.
Crossed References:	Use case *Set Order Of Attributes*. Use case *Set Instance Description*. Use case *Modify Concept Reading Permission*. Use case *Modify Instance Reading Permission.* Use case *Modify Instance Writing Permission.*
Notes:	
Exceptions:	If the list is empty, show a message explaining that there are no concepts in the ontology.
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The list of the concepts contained in the variable has been shown to the user.

Name:	retrieveOrderOfAttributes(concept)
Responsibilities:	Requests to the KW portal the actual ordering of the attributes of a concept.
Crossed References:	Use case *Set Order Of Attributes*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The order of the attributes of *concept* has been requested to the KW portal.

Name:	attribute[] := getConceptAttributes(concept)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the list of attributes of a concept.
Crossed References:	Use case *Set Order Of Attributes*. Use case *Set Instance Description*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The variable *attribute[]* contains the list of all the attributes of the concept *concept*.

Name:	showAttributeOrderForm(attribute[])
Responsibilities:	Shows the form with the attributes contained in a list, in a certain order.
Crossed References:	Use case *Set Order Of Attributes*.
Notes:	
Exceptions:	If the list is empty, show a message explaining that there are no attributes in the concept.
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The form with the attributes contained in *attribute[]* has been shown to the user in a certain order.

Name:	updateAttributeOrder(concept, attribute[])
Responsibilities:	Updates the order of the attributes of a concept in the KW portal.
Crossed References:	Use case *Set Order Of Attributes*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The concept *concept* has been updated in the KW portal with the new order of attributes specified by the list *attribute[]*.

Name:	imposeOrderToChildren(concept, attribute[])
Responsibilities:	Applies an order of attributes to all the children of a concept.
Crossed References:	Use case *Impose Order To Children*.
Notes:	
Exceptions:	If there is not any concept with that name, show a message explaining the error.
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	All the concepts children of *concept* have been updated with the new order of attributes indicated by *attribute[]*.

4.2.5 Short Instance Description

Description

With this use case, the *Portal Administrator* can define the set of attributes to be used to describe instances of a concept in the instance list visualization, together with the order in which these attributes will appear. As in the previous use case, the description and the order can be imposed to the subclasses of the concept.

As we can see in *Figure 4-16*, the *Instance Description* use case breaks down in two: *Set Instance Description* and *Retrieve Instance Description*. There is also another use case that extends the first one: *Impose Description To Children*.

We saw in *Figure 4-2* that the *Short Instance Description* use case retrieves the instances and concepts by means of two use cases from the *Ontology Repository*. These are *Instance Retrieval* and *Conceptualization Retrieval*.

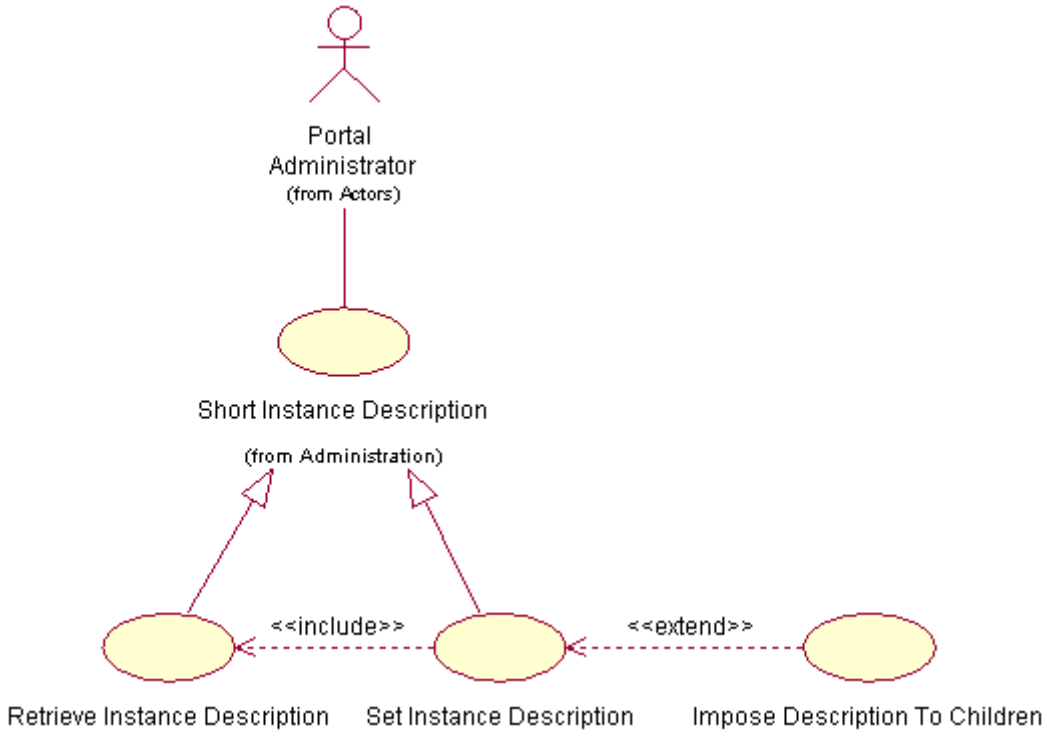

Figure 4-16 *Use case diagram of the Short Instance Description use case.*

4.2.5.1 Set Instance Description

Flow of events

The flow of events of this use case is represented by *Figure 4-17*. There we can see that when the *Portal Administrator* decides to set a description of an instance, the portal has to get the concept taxonomy from the *Ontology Server*, and show it to the user. Then, this one selects the concept, and the KW portal will access again the *Ontology Server* in order to obtain the attributes of the concept. Finally, the portal will show the actual description of the concept to the user, and now (s)he will update it on his (her) own.

At the bottom of the diagram we can see another operation, called *imposeDescriptionToChildren*. It represents the use case that extends the *Set Instance Description* use case, as we saw in *Figure 4-16*. The *Portal Administrator* gives the name of the concept and the list of attributes that will describe it, and the KW portal will internally apply it to the children of the concept.

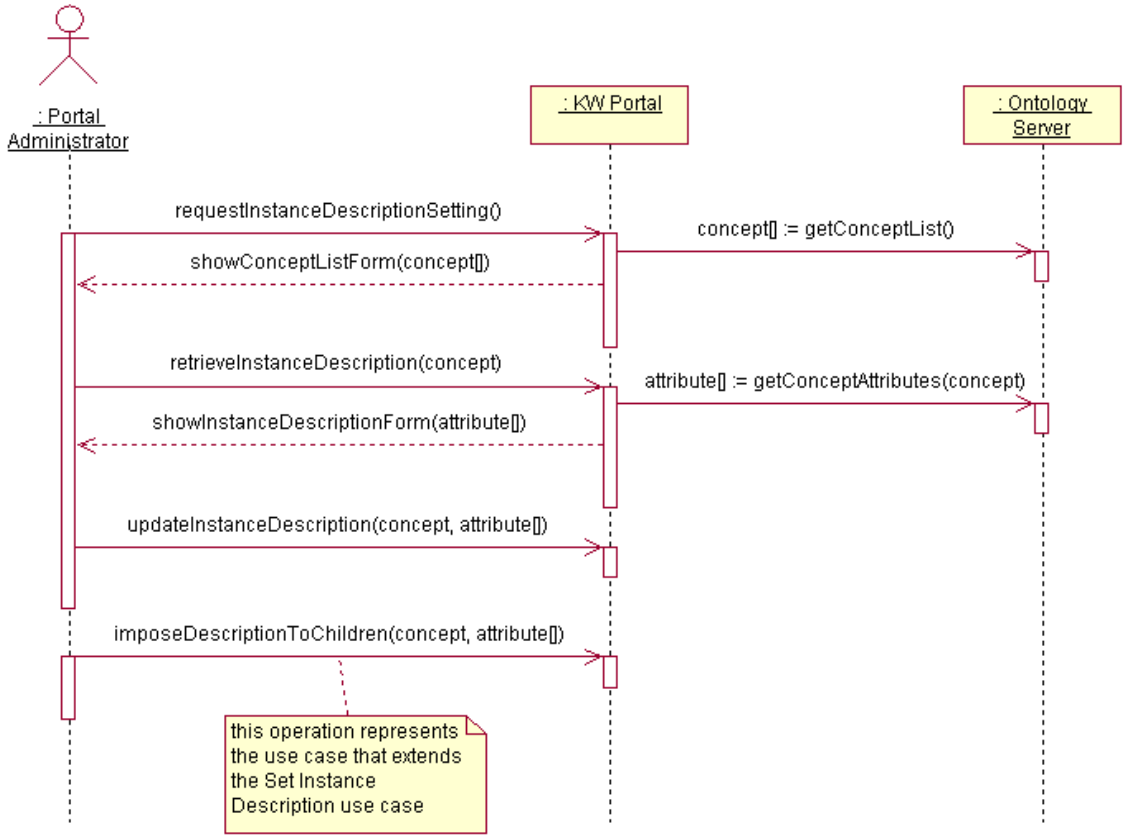

Figure 4-17 *Sequence diagram of the Set Instance Description use case.*

Architectural Implications: None

Contracts

Name:	requestInstanceDescriptionSetting()
Responsibilities:	Initiates the use case *Set Instance Description*.
Crossed References:	Use case *Set Instance Description*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The use case *Set Instance Description* has been initiated.

Name:	concept[] := getConceptList(ontology)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the list containing the names of the concepts of a certain ontology of the *Ontology Server*.
Crossed References:	Use case *Set Order Of Attributes*. Use case *Set Instance Description*. Use case *Modify Concept Reading Permission*. Use case *Modify Instance Reading Permission*. Use case *Modify Instance Writing Permission*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The variable *concept[]* contains the list of the names of the concepts of all the ontologies of the KW portal.

Name:	showConceptListForm(concept[])
Responsibilities:	Shows to the user the form containing the list of the concepts appearing in a list.
Crossed References:	Use case *Set Order Of Attributes*. Use case *Set Instance Description*. Use case *Modify Concept Reading Permission*. Use case *Modify Instance Reading Permission.* Use case *Modify Instance Writing Permission.*
Notes:	
Exceptions:	If the list is empty, show a message explaining that there are no concepts in the ontology.
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The list of the concepts contained in the variable has been shown to the user.

Name:	retrieveInstanceDescription(concept)
Responsibilities:	Requests to the KW portal the actual description of the instances of a concept.
Crossed References:	Use case *Set Instance Description*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The description of the instances of *concept* has been requested to the KW portal.

Name:	attributeList := getConceptAttributes(concept)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the list of attributes of a concept.
Crossed References:	Use case *Set Order Of Attributes*. Use case *Set Instance Description*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The variable *attributeList* contains the list of all the attributes of the concept *concept*.

Name:	showInstanceDescriptionForm(attributeList, description)
Responsibilities:	Shows the form with the attributes contained in a list (all the attributes of a certain concept) and the list of the attributes that represents the current description of a concept.
Crossed References:	Use case *Set Instance Description*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The form containing the two lists of attributes (*attributeList* and *description*) has been shown to the user.

Name:	updateInstanceDescription(concept, newDescription)
Responsibilities:	Updates the description of a concept in the KW portal.
Crossed References:	Use case *Set Instance Description*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The concept *concept* has been updated in the KW portal with the new description specified by the variable *newDescription*.

4.3 Logging

Description
This business use case provides the function of logging the users in the KW semantic portal. A user may log in as a *KW User* (acquiring permissions for entering/editing/modifying information and also requesting contents stored in the portal that are not available for external users) or as a *Portal Administrator* (acquiring permissions for administrating the KW portal).

Flow of events
We can see the flow of events of this use case in *Figure 4-18*. First, the user (a *Portal Administrator* or a *KW User*) requests logging in the portal by providing a name and a password. In case these values are correct, the KW portal will show the main view to the user (the view will depend on the user: an administrator view to the *Administrator* and a KW view to the *KW User*).

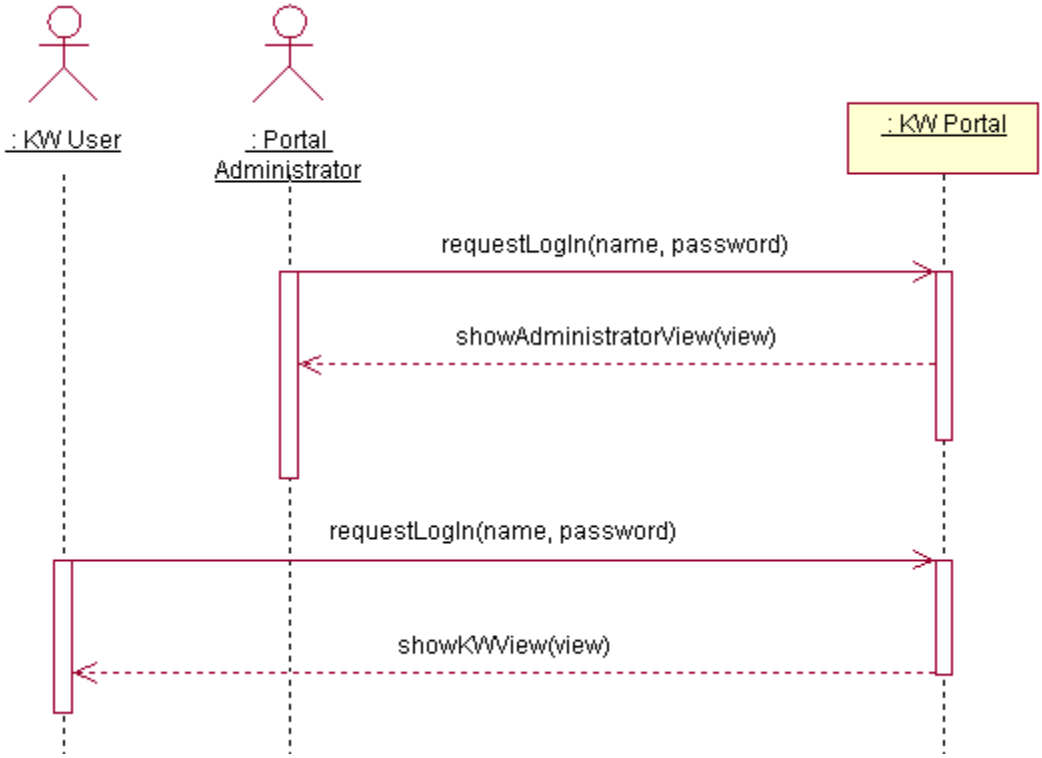

Figure 4-18 *Sequence diagram of the Logging business use case.*

Architectural Implications
The KW semantic portal manages the user session during its life cycle.

Contracts

Name:	requestLogIn(name, password)
Responsibilities:	Requests to log in the KW portal, with the name and password given.
Crossed References:	Use case *Logging*.
Notes:	
Exceptions:	If the name and/or password are incorrect, the user will not be authenticated and a message of error will be shown.
Preconditions:	
Postconditions:	▪ If the name and password authenticates the user as a *Portal Administrator*, (s)he will be logged in the KW portal as it. ▪ If the name and password authenticates the user as an KW User, (s)he will be logged in the KW portal as it.

Name:	showAdministratorView(view)
Responsibilities:	Shows to the user the main administrator view.
Crossed References:	Use case *Logging*.
Notes:	
Exceptions:	
Preconditions:	The user has logged in as a *Portal Administrator*.
Postconditions:	The *Portal Administrator*'s main view has been shown to the user.

Name:	showKWView(view)
Responsibilities:	Shows to the user the main KW view.
Crossed References:	Use case *Logging*.
Notes:	
Exceptions:	
Preconditions:	The user has logged in as a *KW User*.
Postconditions:	The KW main view has been shown to the user.

4.4 Semantic Editing

Description
The *Semantic Editing* business use case consists of providing content to the KW semantic portal by allowing KW users to edit concept instances and the values of their attributes, and to connect such instances by means of relations, even if they belong to different ontologies.

The diagram represented in *Figure 4-19* shows the operations covered by the *Semantic Editing* business use case. This diagram depicts the actor that interact with this use case (*KW User*) and all the more specific use cases associated with it.

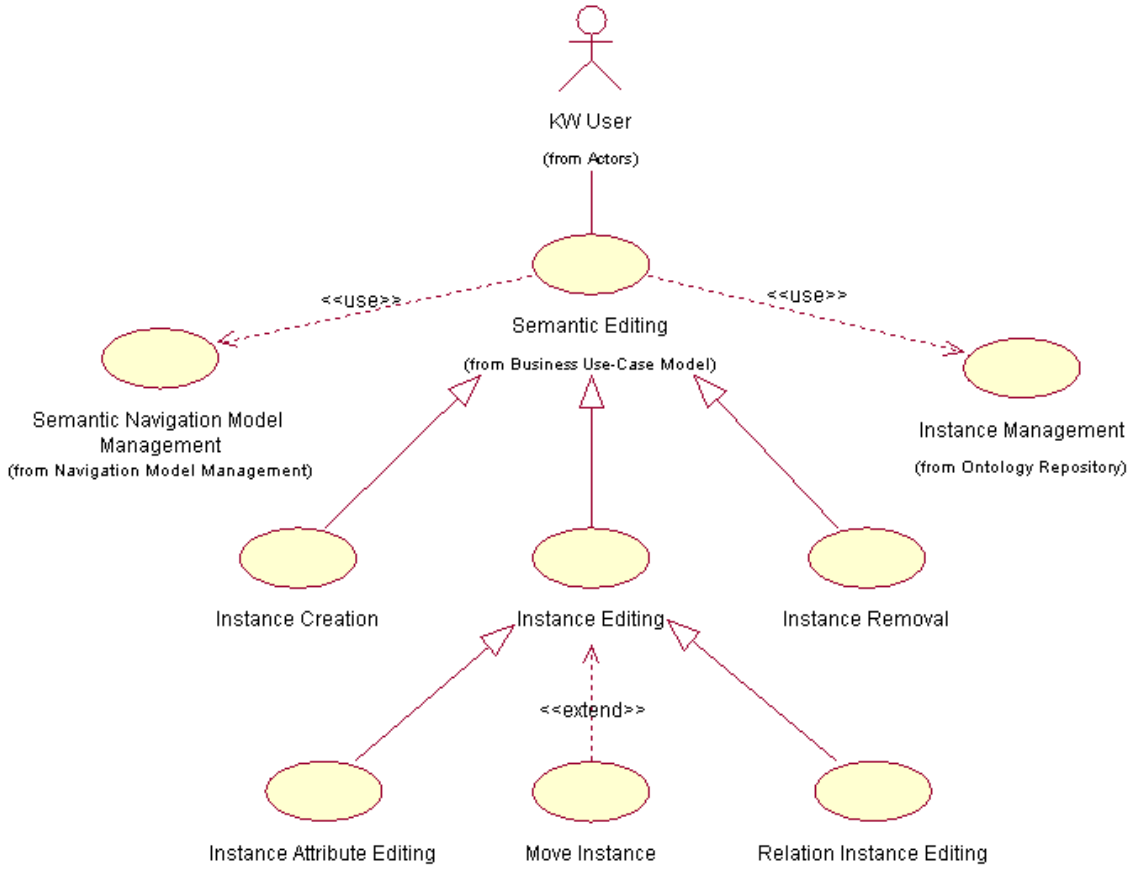

Figure 4-19 *Use case diagram of the Semantic Editing use case.*

The diagram shows that the *Semantic Editing* subsystem will provide functions for: *Instance Creation*, *Instance Editing*, and *Instance Removal*. All of them are respectively explained in sections 4.4.1, 4.4.2 and 4.4.3. The second one, at the same time, breaks down in two more use cases: *Instance Attribute Editing* and *Relation Instance Editing*. There is also another use case that extends the *Instance Editing* use case: *Move Instance* (which moves an instance from one concept to another of the same ontology).

The *Semantic Editing* business use case uses the *Semantic Navigation Model Management* use case to get the form interfaces, as well as the *Instance Management* (from the *Ontology Repository* business use case in the *Ontology Server*) to create, edit and remove instances and relations between instances.

4.4.1 Instance Creation

Description

With this use case, the *KW User* can create an instance of a certain concept.

Flow of events

The flow of events of this use case is specified by the sequence diagram depicted in *Figure 4-20*. When the *KW User* decides to create a new instance, (s)he must fill a form (provided by the KW semantic portal under request) with the convenient values of the attributes. Then, the KW portal formalizes the action by adding the new instance to the *Ontology Server*.

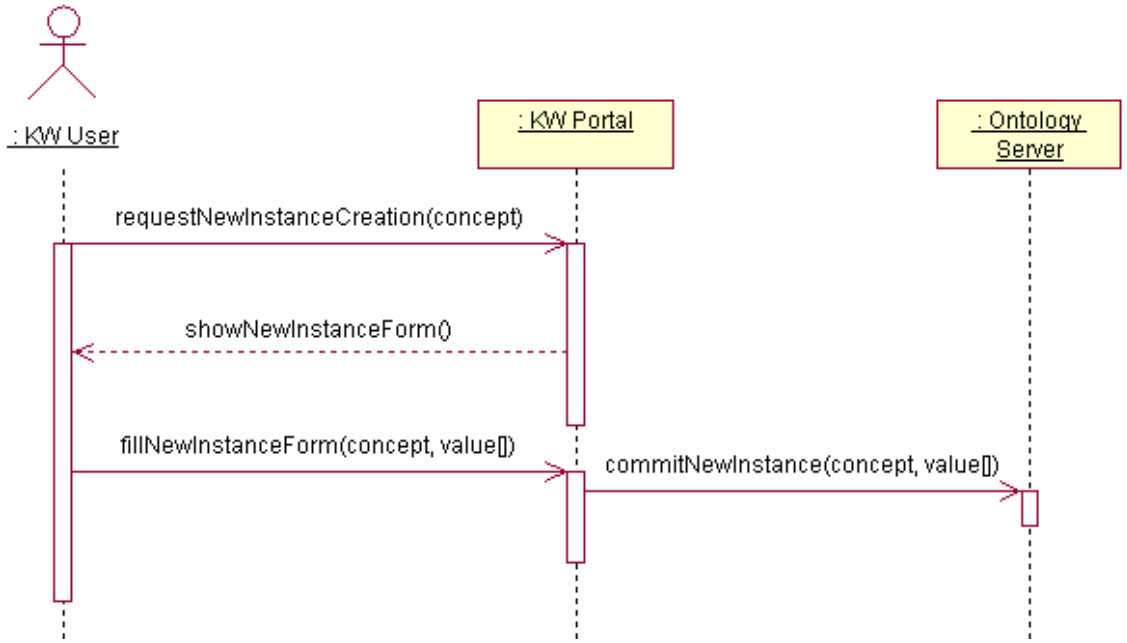

Figure 4-20 *Sequence diagram of the Instance Creation use case.*

Architectural Implications
The *Ontology Server* must control concurrent accesses of the information.

Contracts

Name:	requestNewInstanceCreation(concept)
Responsibilities:	Initiates the use case *Instance Creation*.
Crossed References:	Use case *Instance Creation*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *KW User*.
Postconditions:	The use case *Instance Creation* has been initiated.

Name:	showNewInstanceForm()
Responsibilities:	Shows the form in blank to be filled with the information of the new instance.
Crossed References:	Use case *Instance Creation*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *KW User*.
Postconditions:	The form in blank to be filled with the information of a new instance has been shown to the user.

Name:	fillNewInstanceForm(concept, value[])
Responsibilities:	Reports to the KW portal the information of a new instance.
Crossed References:	Use case *Instance Creation*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *KW User*.
Postconditions:	The information (contained in *value[]*) of a new instance of the concept *concept* has been reported to the KW portal.

Name:	commitNewInstance(concept, value[])
Responsibilities:	Creates a new instanceof the concept *concept* in the *Ontology Server* with the information given.
Crossed References:	Use case *Instance Creation*.
Notes:	
Exceptions:	If there is already an instance with that name in that concept, show a message explaining the error.
Preconditions:	The user is logged in as a *KW User*.
Postconditions:	A new instance of the concept *concept* has been created in the *Ontology Server*, with the data contained in the variable *value[]*.

4.4.2 Instance Editing

Description

With this use case, the *KW User* is capable of modifying the attributes and relations of a certain instance.

Flow of events

The flow of events of this use case is depicted in *Figure 4-21*. When a *KW User* decides to edit a particular instance, the KW portal obtains the instance information from the *Ontology Server*, and then shows it to the user. This way, he now can make the convenient changes in the values of the instance attributes and/or relations. Finally, the KW portal stores the new information of the instance in the *Ontology Server*.

When the *KW User* requests the moving of an instance from one concept to another of the same ontology, the KW portal moves it from the origin to the destination concept.

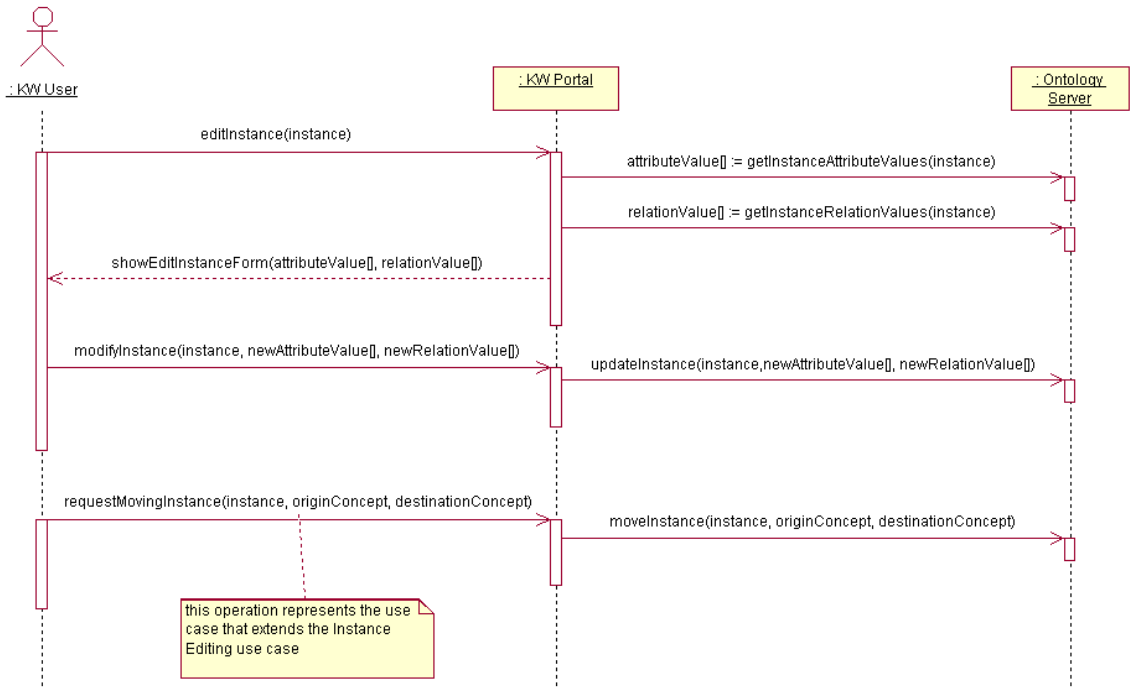

Figure 4-21 *Sequence diagram of the Instance Editing use case.*

Architectural Implications: None

Contracts

Name:	editInstance(attribute)
Responsibilities:	Initiates the use case *Instance Editing*.
Crossed References:	Use case *Instance Editing*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *KW User*.
Postconditions:	The use case *Instance Editing* has been initiated.

Name:	attributeValue[] := getInstanceAttributeValues(instance)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the values of the attributes of an instance.
Crossed References:	Use case *Instance Editing*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *KW User*.
Postconditions:	The variable *attributeValue[]* contains the values of the attributes of the instance *instance*.

Name:	relationValue[] := getInstanceRelationValues(instance)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the values of the relations of an instance.
Crossed References:	Use case *Instance Editing*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *KW User*.
Postconditions:	The variable *relationValue[]* contains the values of the relations of the instance *instance*.

Name:	showEditInstanceForm(attributeValues[], relationValues[])
Responsibilities:	Shows the form containing the current information of an instance.
Crossed References:	Use case *Instance Editing*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *KW User*.
Postconditions:	The form with the information (the values of the attributes and relations) of an instance has been shown to the user.

Name:	modifyInstance(instance, newAttributeValue[], newRelationValue[])
Responsibilities:	Reports to the KW portal the new information of an instance, according to the data given.
Crossed References:	Use case *Instance Editing*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *KW User*.
Postconditions:	The new values of the attributes and relations of the instance *instance* has been reported to the KW portal.

Name:	updateInstance(instance,newAttributeValue[], newRelationValue[])
Responsibilities:	Updates in the *Ontology Server* the information of an instance, according to the data given.
Crossed References:	Use case *Instance Editing*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *KW User*.
Postconditions:	The instance *instance* has been updated in the *Ontology Server*, according to *newAttributeValue[]*, and *newRelationValue[]*.

Name:	requestMovingInstance(instance, originConcept, destinationConcept)
Responsibilities:	Requests to the KW portal the moving of an instance from one concept to another.
Crossed References:	Use case *Instance Editing*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *KW User*.
Postconditions:	The KW semantic portal has been requested to move *instance* from the concept *originConcept* to *destinationConcept*.

Name:	moveInstance(instance, originConcept, destinationConcept)
Responsibilities:	Moves an instance from one concept to another in the *Ontology Server*.
Crossed References:	Use case *Instance Editing*.
Notes:	
Exceptions:	If there is already an instance with that name in the destination concept, show a message explaining the error.
Preconditions:	The user is logged in as a *KW User*.
Postconditions:	The instance *instance* has been moved from the concept *originConcept* to the concept *destinationConcept* in the *Ontology Server*.

4.4.3 Instance Removal

Description

This use case allows the KW user to remove an instance from an ontology.

Flow of events

The flow of events in this use case is quite simple. We can see it in *Figure 4-22*. First, the *KW User* requests to remove certain instance. Then, the KW portal eliminates the instance (specified by the KW user) from the *Ontology Server*.

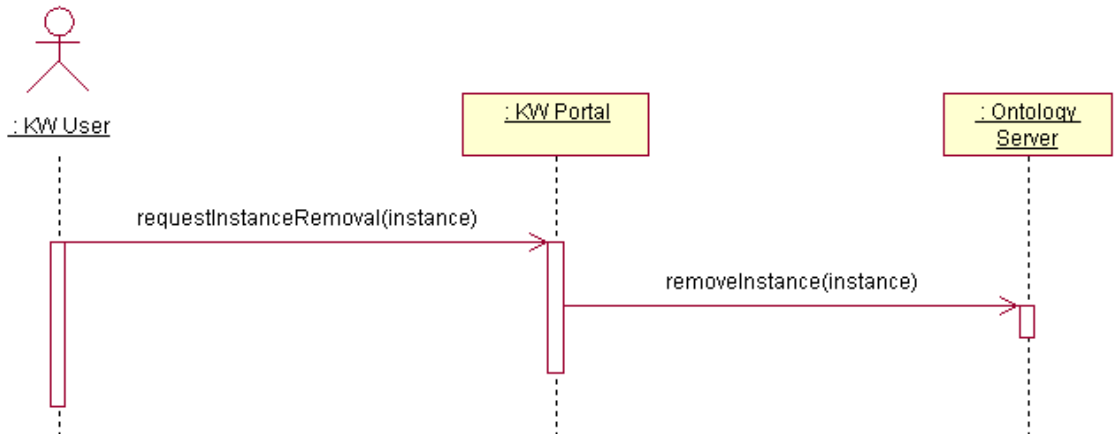

Figure 4-22 *Sequence diagram of the Instance Removal use case.*

Architectural Implications: None

Contracts

Name:	requestInstanceRemoval(instance)
Responsibilities:	Initiates the use case *Instance Removal*.
Crossed References:	Use case *Instance Removal*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *KW User*.
Postconditions:	The use case *Instance Removal* has been initiated.

Name:	removeInstance(instance)
Responsibilities:	Removes an instance from the *Ontology Server*.
Crossed References:	Use case *Instance Removal*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *KW User*.
Postconditions:	The instance called *instance* is no longer in the *Ontology Server*.

4.5 Semantic Browsing

Description
This business use case allows the user (*KW* or *Guest*) to navigate through the KW semantic portal. At the same time, it provides the appropriate views for each user and for each situation. The diagram depicted in *Figure 4-23* describes the operations carried out in the *Semantic Browsing* function. As we said before, there are two actors: the *KW User*, and the *Guest User*.

The *Semantic Browsing* use case includes two more use cases: *Semantic Navigation* and *Semantic Visualization*. Both of them use another two use cases from the *Ontology Repository*: *Instance Retrieval* and *Conceptualization Retrieval*, in order to get the instances and concepts from the ontologies implemented in the *Ontology Server*.

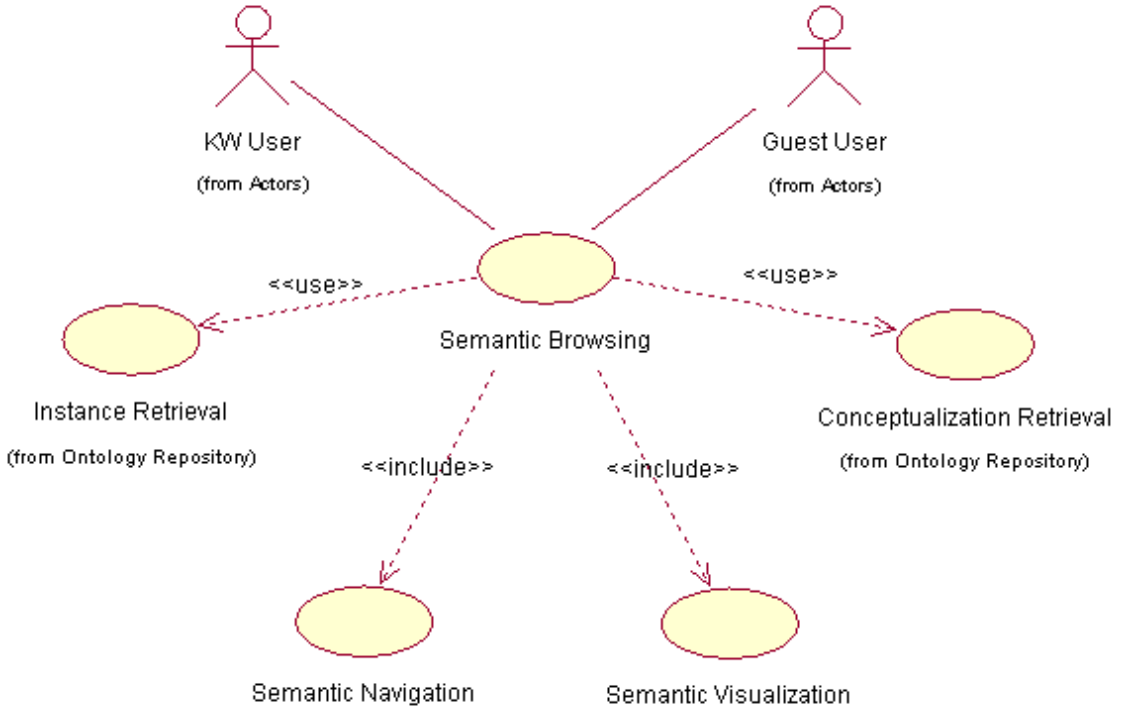

Figure 4-23 *Use case diagram of the Semantic Browsing use case.*

4.5.1 Semantic Navigation

Description

With this use case the user (*KW* or *Guest*) is able to navigate semantically through the KW portal by clicking on the hyperlinks. Hyperlinks represent instances, relations, concepts or ontologies. This navigation will be restricted by the user's permissions.

Flow of events

The flow of events of this use case is shown in *Figure 4-24*. When the user (*Guest* or *KW*) clicks on a hyperlink, (s)he is actually giving to the KW portal the current view and the requested action. Afterwards, the portal will look up the navigation model (which is a preloaded ontology, so accessing the *Ontology Server* is unnecessary) and return to the user the destination view.

Although we will explain it later in section 4.9, we must mention that the navigation model is implemented as an ontology inside the *Ontology Server*. The concepts of this ontology are views, and the user can go from one to another by means of "actions". This way, when the KW portal requests the destination view, all it has to provide are an origin view and a certain action.

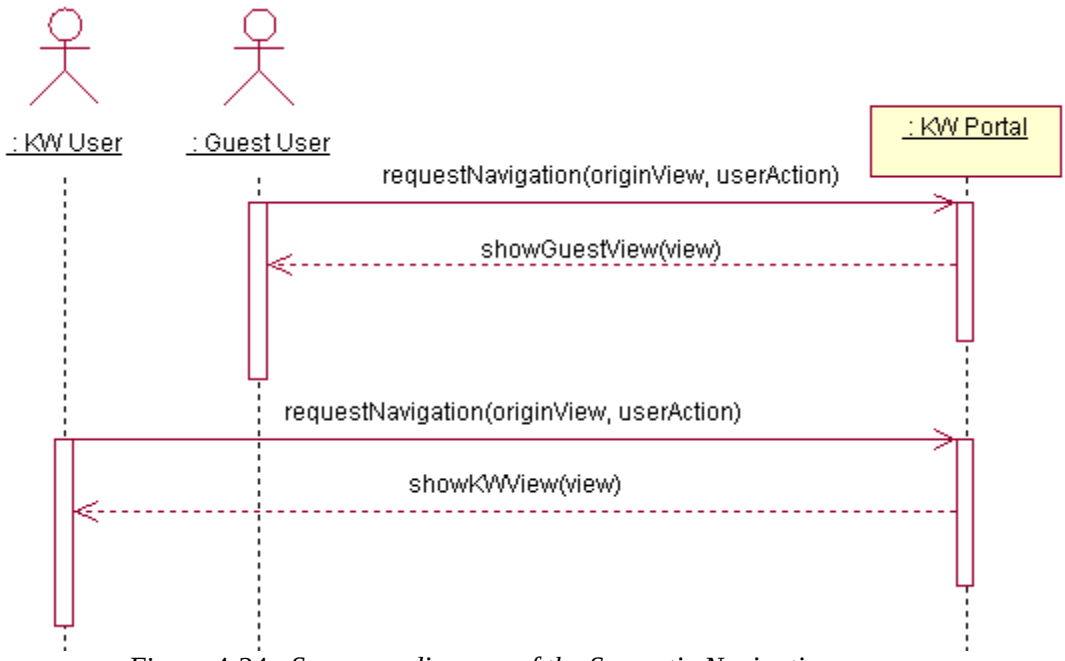

Figure 4-24 . *Sequence diagram of the Semantic Navigation use case.*

Architectural Implications
These functionalities will be provided by the KW portal. The navigation model must be loaded during the KW semantic portal setup.

Contracts

Name:	requestNavigation(originView, userAction)
Responsibilities:	Requests to the KW portal the destination view, given an origin view and a certain action.
Crossed References:	Use case *Semantic Navigation.*
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The use case *Semantic Navigation* has been initiated.

Name:	showGuestView(view)
Responsibilities:	Shows a certain view to the *Guest User*.
Crossed References:	Use case *Semantic Navigation.*
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	A certain view has been shown to the *Guest User*.

Name:	showKWView(view)
Responsibilities:	Shows a certain view to the *KW User*.
Crossed References:	Use case *Semantic Navigation.*
Notes:	
Exceptions:	
Preconditions:	The user has logged in as a *KW User*.
Postconditions:	A certain view has been shown to the *KW User*.

4.5.2 Semantic Visualization

Description

This use case provides the appropriate views for each user and for each situation, that is, depending on the user's permissions and its situation in the current navigation model.

The diagram depicted in *Figure 4-25* describes the *Semantic Visualization* use case. As we can see, there are two types of *Semantic Visualization*: *Guest Visualization* (for *Guest Users*) and *KW Visualization* (for *KW Users*).

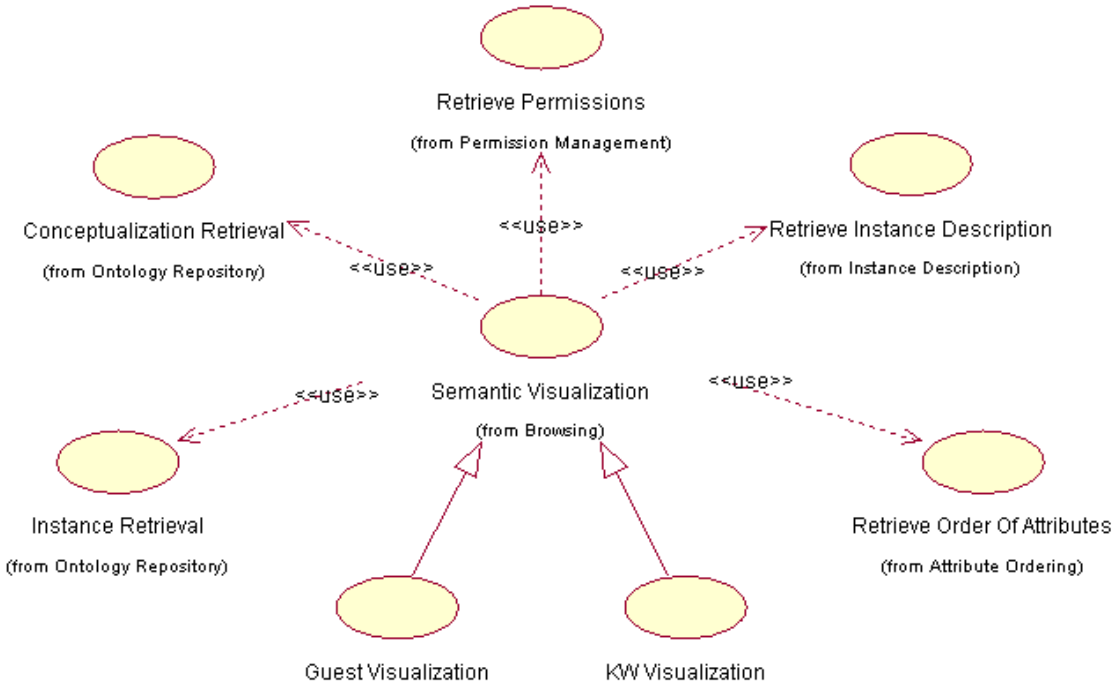

Figure 4-25 *Use case diagram of the Visualization use case.*

The *Semantic Visualization* also uses six use cases: *Conceptualization Retrieval* and *Instance Retrieval* (both from *Ontology Repository*) to get the visualization models, *Retrieve Permissions* (from the *Permission Management* use case inside the *Administration* business use case) to get the user permissions, *Retrieve Instance Description* (from the *Instance Description* use case inside the *Administration* business use case) to get the instance descriptions, and finally, *Retrieve Order Of Attributes* (from the *Attribute Ordering* use case inside the *Administration* business use case) to get the order of the instance attributes.

Flow of events

The flow of events of this use case is depicted in *Figure 4-26*. In this diagram we can see that the user requests the semantic visualization of a view. Then, the KW portal access the *Ontology Server* in order to obtain the info of all the ontology terms included in the view.
Finally, the portal will show that information to the user depending on its reading permissions.
Although the flow of events is the same for both kind of users, each one of them will use a different use case: while a *Guest User* will use the *Guest Visualization*, a *KW User* will use the *KW Visualization*.

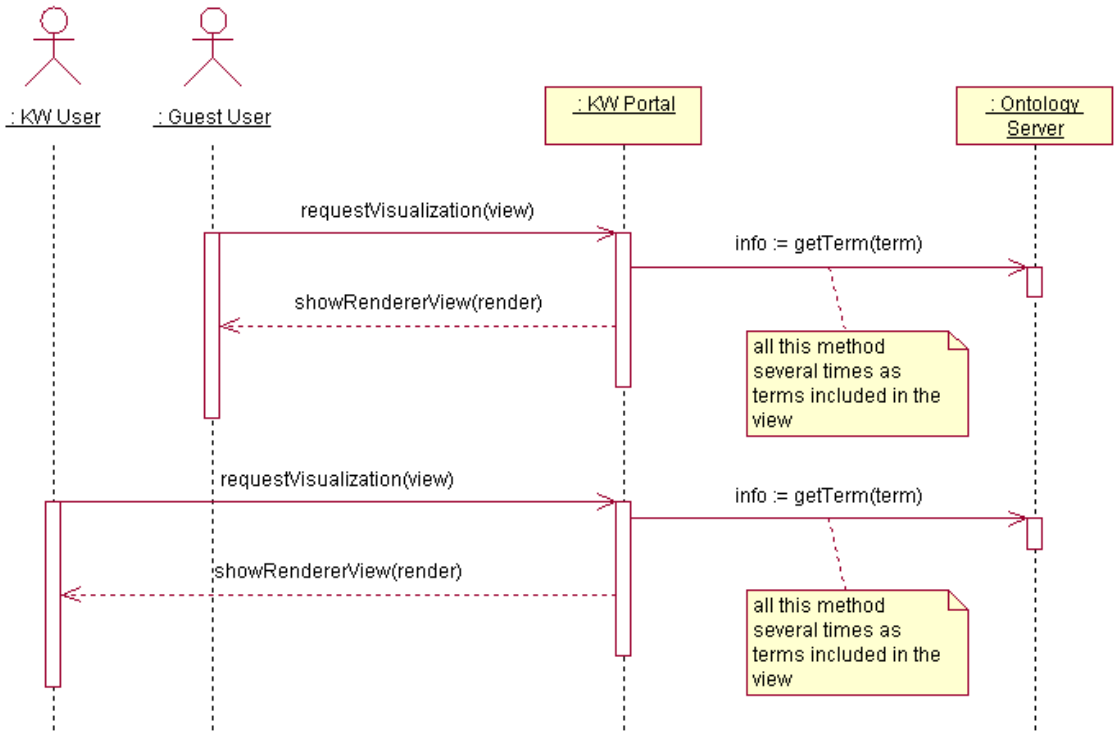

Figure 4-26 *Sequence diagram of the Semantic Visualization use case.*

Architectural Implications: None

Contracts

Name:	requestVisualization(view)
Responsibilities:	Requests to the KW semantic portal the semantic visualization of a view.
Crossed References:	Use case *Semantic Visualization.*
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The semantic visualization of view has been requested to the KW semantic portal.

Name:	info := getTerm(term)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the information of a certain term.
Crossed References:	Use case *Semantic Visualization.*
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The variable *info* contains the information of the term *term*.

Name:	showRendererView(render)
Responsibilities:	Shows to the user the semantic visualization of a view.
Crossed References:	Use case *Semantic Visualization.*
Notes:	▪ The portal will show the information to the user depending on his reading permissions. ▪ Each user will be given a different visualization: *Guest Visualization* for the *Guest User*, and *KW Visualization* for the *KW User*.
Exceptions:	
Preconditions:	
Postconditions:	The visualization of a view has been shown to the user.

4.6 Semantic Searching

Description
This business use case implements the search engine that allows querying for information in one or in all the ontologies of the portal. As we can see in *Figure 4-27*, the *Semantic Searching* function is represented by two use cases: *Search In Term Names* and *Search In Instance Values*. Both of them use another two use cases: *Instance Retrieval* and *Conceptualization Retrieval* (from *Ontology Repository* in the *Ontology Server*) in order to get the concepts and instances from the ontologies implemented in the *Ontology Server*.

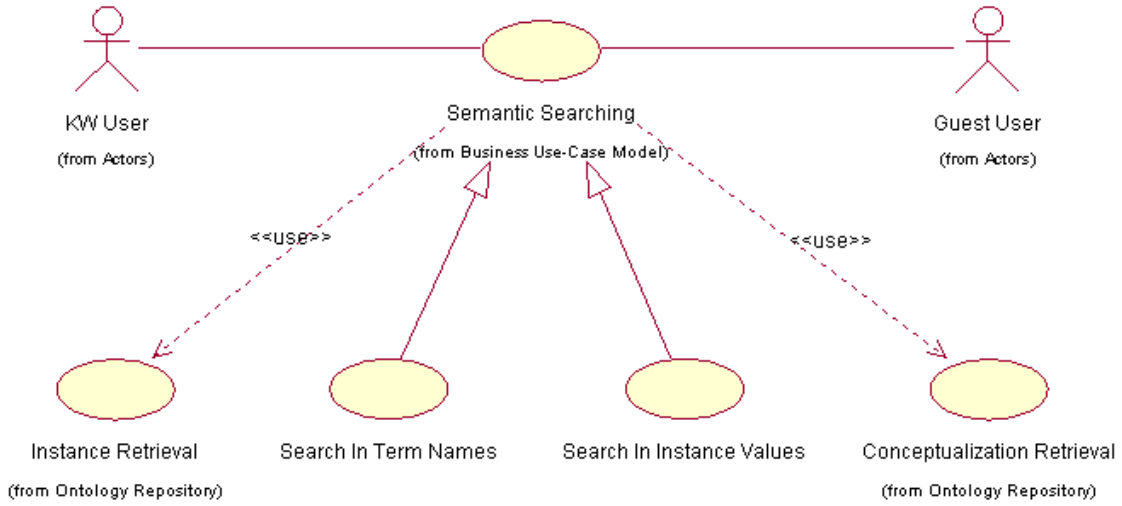

Figure 4-27 *Use case diagram of the Semantic Searching use case*

4.6.1 Search In Term Names

Description
With this use case, the search engine looks for instances or concept names that contain the keywords specified in the query.

Flow of events
The flow of events of this use case is depicted in *Figure 4-28*. First, the user (*KW* or *Guest*) gives a list of terms. Then, the KW portal accesses the *Ontology Server* in order to get the instances or concept names that match with the list of terms. Finally, the KW portal shows the result of the search to the user, and he now can access (*KW* and Guest *Users*) or edit that info (just *KW Users*).

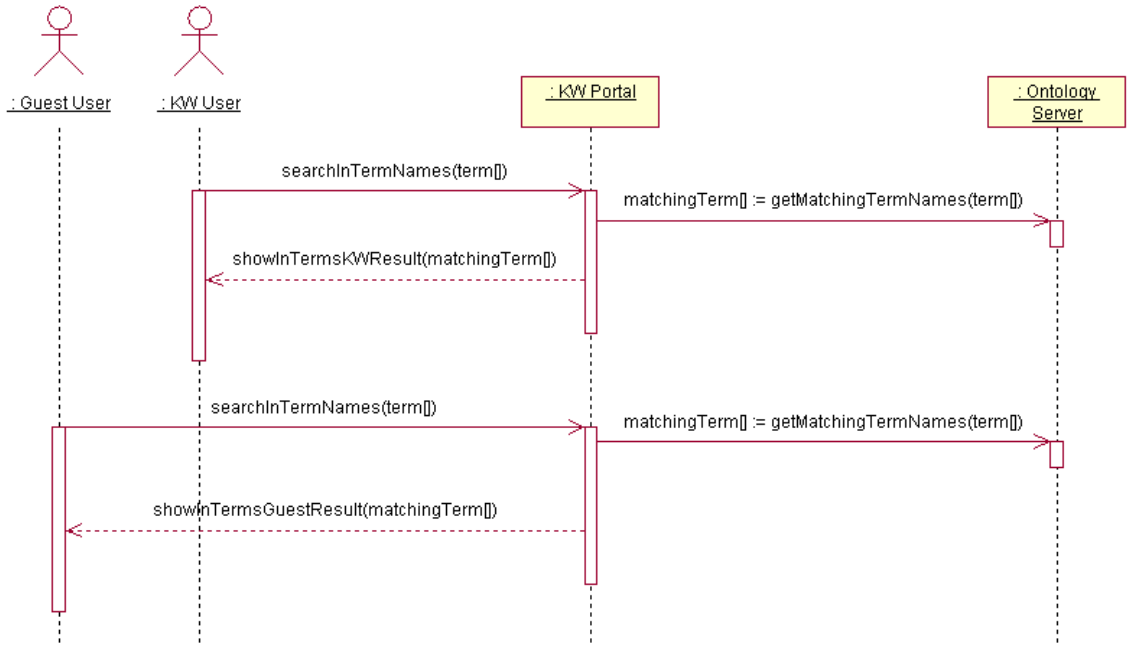

Figure 4-28 *Sequence diagram of the Search In Term Names use case.*

Architectural Implications: None

Contracts

Name:	searchInTermNames(term[])
Responsibilities:	Initiates the use case *Search In Term Names*.
Crossed References:	Use case *Search In Term Names.*
Notes:	
Exceptions:	If no terms have been introduced, show a message explaining the error.
Preconditions:	
Postconditions:	The use case *Search In Term Names* has been initiated.

Name:	matchingTerm[] := getMatchingTermNames(term[])
Responsibilities:	Obtains (from the *Ontology Server*) and returns the terms that match with the ones given.
Crossed References:	Use case *Search In Term Names.*
Notes:	
Exceptions:	No results will be returned if terms are missing.
Preconditions:	
Postconditions:	The variable *matchingTerm[]* contains the list of terms that match with the words contained in *term[]*.

Name:	showInTermsKWResult(view)
Responsibilities:	Shows to the *KW User* the results of a conventional search, according to the data given.
Crossed References:	Use case *Search In Term Names.*
Notes:	
Exceptions:	If the list of terms is empty, show a message indicating that there has been no matches.
Preconditions:	The user has logged in as a *KW User*.
Postconditions:	The results of the "in terms" search has been shown to the *KW User*, giving him the option of accessing or editing them.

Name:	showInTermsGuestResult(view)
Responsibilities:	Shows to the *Guest User* the results of a conventional search, according to the data given.
Crossed References:	Use case *Search In Term Names.*
Notes:	
Exceptions:	If the list of terms is empty, show a message indicating that there has been no matches.
Preconditions:	
Postconditions:	The results of the "in terms" search has been shown to the *Guest User*, giving him just the option of accessing them.

4.6.2 Search In Instance Values

Description

The KW portal provides an advanced search function by means of a query form. The fields to be filled in the query form are attributes and relations taken from the ontology we are querying. Once the user introduces the values he is looking for, the search engine returns the instances that satisfy the conditions imposed in the attributes values specified in the form.

Flow of events

The flow of events of this use case is depicted in *Figure 4-29*. First, the KW portal gets the list of ontologies from the *Ontology Server*, with which he creates the initial advanced search form, showing it then to the user (*Guest* or *KW*). The user selects the ontology and the KW portal accesses again the *Ontology Server* in order to get its general attributes. This info composes the advanced search form, which now can be filled by the user on his own. Finally, the KW portal gets the matching terms from the *Ontology Server* and delivers the results to the user, so he can access (*Guest* and *KW Users*) or edit them (just *KW Users*).

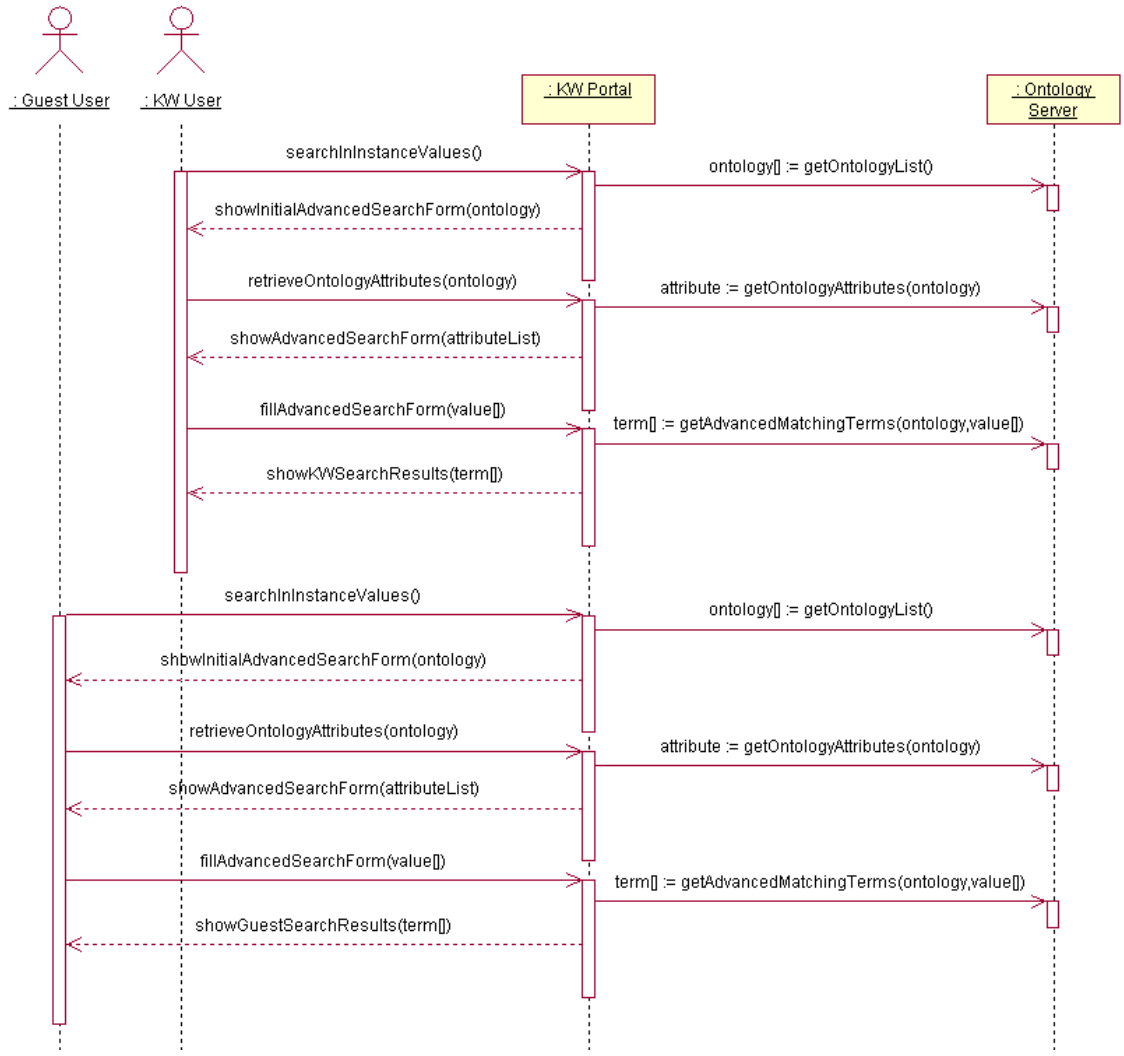

Figure 4-29 *Sequence diagram of the Search In Instance Values use case.*

Architectural Implications: None

Contracts

Name:	searchInInstanceValues()
Responsibilities:	Initiates the use case *Search In Instance Values.*
Crossed References:	Use case *Search In Instance Values.*
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The use case *Search In Instance Values* has been initiated.

Name:	ontology[] := getOntologyList()
Responsibilities:	Obtains (from the *Ontology Server*) and returns the list containing the names of all the ontologies stored in the *Ontology Server*.
Crossed References:	Use case *Add Ontology Publication*. Use case *Remove Ontology Publication*. Use case *Search In Instance Values.*
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The variable *ontology[]* contains the list of the names of all the ontologies of the *Ontology Server*.

Name:	showInitialAdvancedSearchForm(ontology[])
Responsibilities:	Shows the form with the list of ontologies, from which the user should select one of them.
Crossed References:	Use case *Search In Instance Values.*
Notes:	
Exceptions:	If the list is empty, show a message explaining that there are no ontologies stored in the *Ontology Server*.
Preconditions:	
Postconditions:	The advanced search form with the list of the ontologies has been shown to the user.

Name:	retrieveOntologyAttributes(ontology)
Responsibilities:	Requests to the KW portal the attributes of an ontology.
Crossed References:	Use case *Search In Instance Values.*
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The attributes of *ontology* has been requested to the KW portal.

Name:	attribute[] := getOntologyAttributes(ontology)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the list of the attributes of an ontology.
Crossed References:	Use case *Search In Instance Values.*
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The variable *attribute[]* contains all the attributes of the ontology *ontology*.

Name:	showAdvancedSearchForm(attribute[])
Responsibilities:	Shows to the user the form containing the fields in blank related to the attributes of an ontology.
Crossed References:	Use case *Search In Instance Values.*
Notes:	
Exceptions:	If the list is empty, show a message explaining that the ontology has no attributes.
Preconditions:	
Postconditions:	The form (with the fields in blank) related to the attributes of an ontology has been shown to the user.

Name:	fillAdvancedSearchForm(value[])
Responsibilities:	Reports to the KW portal the values of the attributes to be searched in the *Ontology Server*.
Crossed References:	Use case *Search In Instance Values.*
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The advanced search form has been filled and its values has been reported to the KW portal.

Name:	term[] := getAdvancedMatchingTerms(ontology,value[])
Responsibilities:	Obtains (from the *Ontology Server*) and returns the terms of an ontology that match with the values of the attributes given.
Crossed References:	Use case *Search In Instance Values.*
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The variable *term[]* contains the list of the terms of the ontology *ontology* that match with the values of the attributes represented in *value[]*.

Name:	showKWSearchResults(term[])
Responsibilities:	Shows to the *KW User* the results of a search, according to the data given.
Crossed References:	Use case *Search In Instance Values.*
Notes:	
Exceptions:	If the list is empty, show a message indicating that there has been no matches.
Preconditions:	The user has logged in as a *KW User*.
Postconditions:	The results of the search has been shown to the *KW User*, giving him/her the option of accessing or editing them.

Name:	showGuestSearchResults(term[])
Responsibilities:	Shows to the *Guest User* the results of a search, due to the data given.
Crossed References:	Use case *Search In Instance Values.*
Notes:	
Exceptions:	If the list is empty, show a message indicating that there has been no matches.
Preconditions:	
Postconditions:	The results of the search has been shown to the *Guest User*, giving him just the option of accessing them.

4.7 Semantic Content Visualization

This business use case allows a *Software Agent* to obtain the semantic visualization of a concept or instance in a certain semantic web language (OWL or RDF).

4.7.1 Content Generation in Semantic Web Languages

Description

This use case allows a *Software Agent* to obtain the semantic visualization of a concept or instance in a certain semantic web language. At this point, there are two languages supported by the KW semantic portal: OWL [Dean and Schreiber, 2003] and RDF [Lassila and Swick, 1999]. Therefore, as we can see in *Figure 4-30*, the *Content Generation in Semantic Web Languages* function breaks down in two use cases: *OWL Translation* and *RDF(S) Translation*.

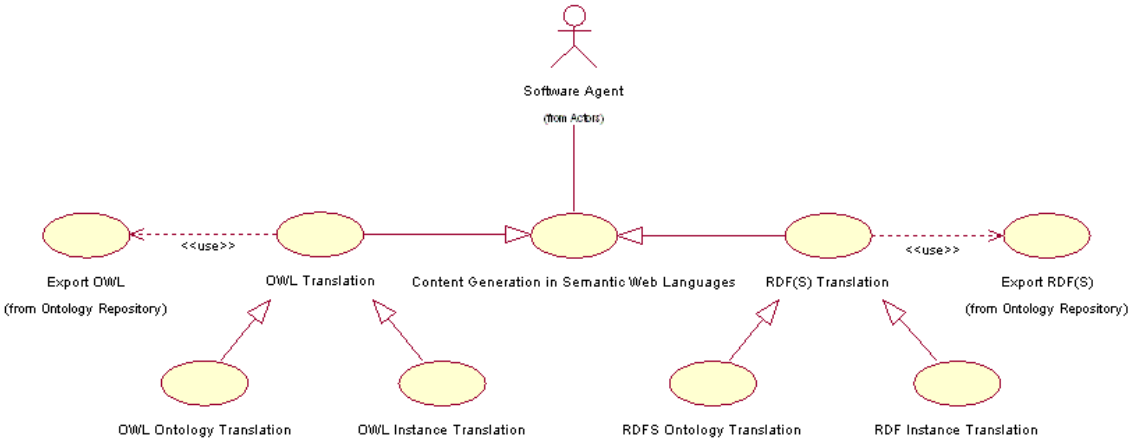

Figure 4-30 *Use case diagram of the Content Generation in Semantic Web Languages use case*

4.7.1.1 OWL Translation

Description

With this use case, the user can get OWL code of concepts and instances, for what it is divided in two more use cases: the *OWL Ontology Translation* and the *OWL Instance Translation*. They are clearly explained in sections 4.7.1.1.1 and 4.7.1.1.2 respectively. Both of them use *Export OWL* (from the *Ontology Repository*).

4.7.1.1.1 OWL Ontology Translation

Flow of events

The flow of events for this use case is shown in *Figure 4-31*. When the *Software Agent* requests the OWL visualization of an ontology, the KW portal imports it from the *Ontology Server* and returns the code to the *Software Agent*.

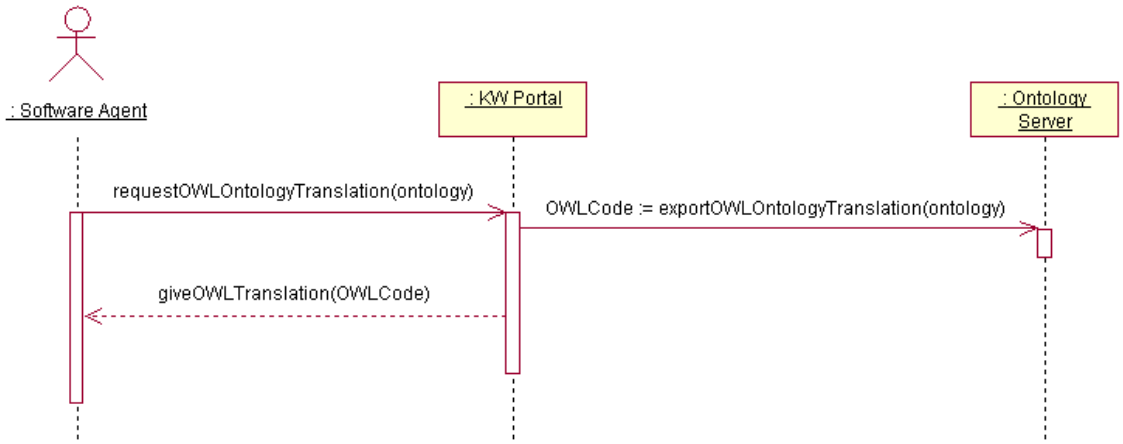

Figure 4-31 *Sequence diagram of the OWL Ontology Translation use case.*

Architectural Implications: None

Contracts

Name:	requestOWLOntologyTranslation(ontology)
Responsibilities:	Requests the OWL code of an ontology.
Crossed References:	Use case *OWL Ontology Translation*.
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The OWL code of the ontology *ontology* has been requested to the KW portal.

Name:	OWLCode := importOWLOntologyTranslation(ontology)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the OWL code of an ontology.
Crossed References:	Use case *OWL Ontology Translation*.
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The variable *OWLCode* contains the OWL code of *ontology*.

Name:	giveOWLTranslation(OWLCode)
Responsibilities:	Gives a certain OWL code to the user.
Crossed References:	Use case *OWL Ontology Translation*. Use case *OWL Instance Translation*.
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The OWL code contained in the variable has been shown to the user.

4.7.1.1.2 OWL Instance Translation

Flow of events

The flow of events for this use case is shown in *Figure 4-32*. When the *Software Agent* requests presenting a list of instances in OWL, the KW portal imports them from the *Ontology Server* and returns the code to the *Software Agent*.

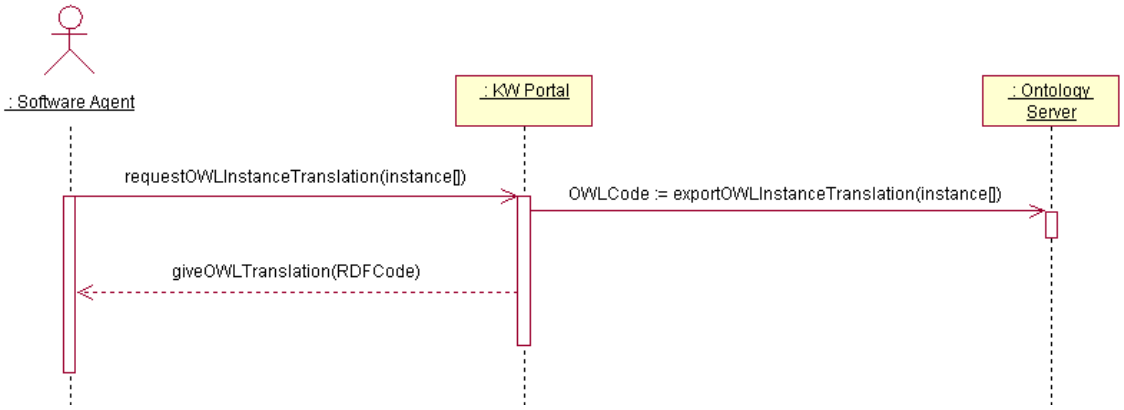

Figure 4-32 *Sequence diagram of the OWL Instance Translation use case.*

Architectural Implications: None

Contracts

Name:	requestOWLInstanceTranslation(instance[])
Responsibilities:	Requests the RDF code of a list of instances that follows an OWL ontology.
Crossed References:	Use case *OWL Instance Translation*.
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The RDF code of the list of instances has been requested to the KW portal.

Name:	OWLCode := exportOWLInstanceTranslation(instance[])
Responsibilities:	Obtains (from the *Ontology Server*) and returns the OWL code of a list of instances.
Crossed References:	Use case *OWL Instance Translation*.
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The variable *OWLCode* contains the OWL code of the list of instances.

Name:	giveOWLTranslation(RDFCode)
Responsibilities:	Gives a certain RDF code to the user.
Crossed References:	Use case *OWL Ontology Translation*. Use case *OWL Instance Translation*.
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The RDF code contained in the variable has been shown to the user.

4.7.1.2 RDF(S) Translation

Description
With this other use case, users can get the RDF(S) code for ontology concepts and instances. Therefore, there are two use cases that implement the two functionalities. As we saw in *Figure 4-30*, both use another use case from the *Ontology Repository*: *Export RDFS*.

4.7.1.2.1 RDFS Ontology Visualization

Flow of events
The flow of events of this use case is shown in *Figure 4-33*. When the *Software Agent* requests an RDFS code, the KW portal imports it from the *Ontology Server* and returns the code to the *Software Agent*.

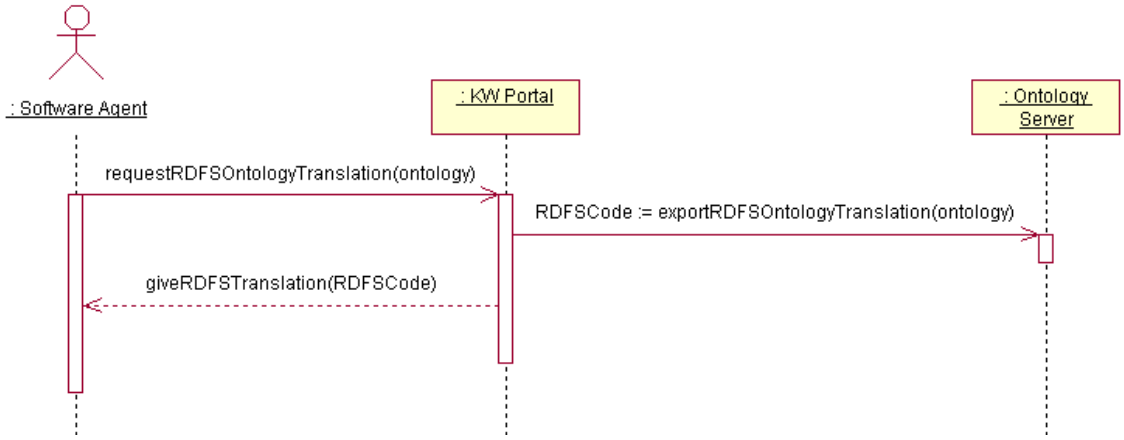

Figure 4-33 *Sequence diagram of the RDFS Ontology Translation use case.*

Architectural Implications: None

Contracts

Name:	requestRDFSOntologyTranslation(ontology)
Responsibilities:	Requests the RDF code of an ontology.
Crossed References:	Use case *RDFS Ontology Translation*.
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The RDFS code of the ontology *ontology* has been requested to the KW portal.

Name:	RDFSCode := exportRDFSOntologyTranslation(ontology)
Responsibilities:	Obtains (from the *Ontology Server*) and returns the RDFS code of an ontology.
Crossed References:	Use case *RDFS Ontology Translation*.
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The variable *RDFSCode* contains the RDF code of *ontology*.

Name:	giveRDFSTranslation(RDFSCode)
Responsibilities:	Gives a certain RDFS code to the user.
Crossed References:	Use case *RDFS Ontology Translation*.
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The RDFS code contained in the variable has been shown to the user.

4.7.1.2.2 RDF Instance Translation

Flow of events

The flow of events for this use case is shown in *Figure 4-34*. When the *Software Agent* requests the RDF visualization of an instance, the KW portal imports it from the *Ontology Server* and returns the code to the *Software Agent*.

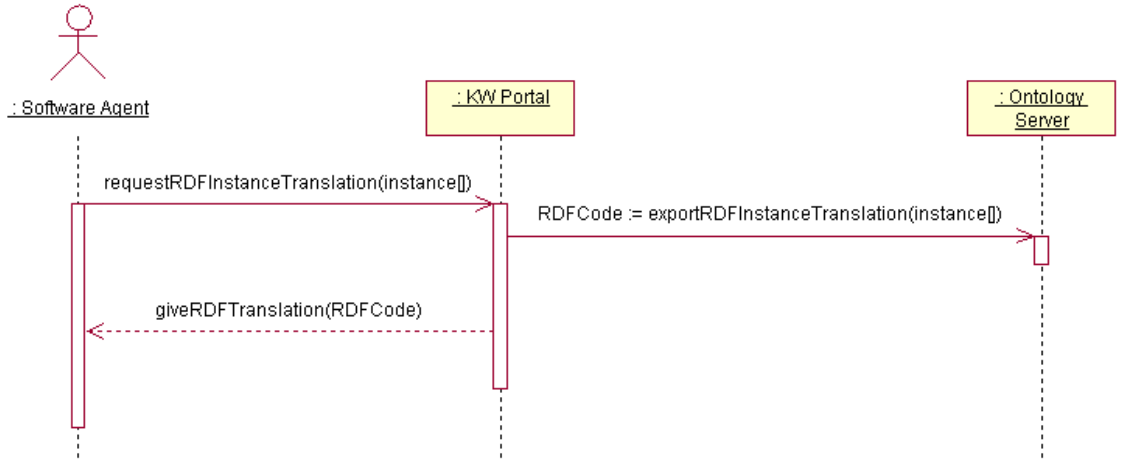

Figure 4-34 *Sequence diagram of the RDF Instance Translation use case.*

Architectural Implications: None

Contracts

Name:	requestRDFInstanceTranslation(instance[])
Responsibilities:	Requests the RDF code of a list of instances.
Crossed References:	Use case *RDF Instance Translation*.
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The RDF code of the list of instances has been requested to the KW portal.

Name:	RDFCode := exportRDFInstanceTranslation(instance[])
Responsibilities:	Obtains (from the *Ontology Server*) and returns the RDFS code of a list of instances.
Crossed References:	Use case *RDF Instance Translation*.
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The variable *RDFCode* contains the RDF code of the list of instances.

Name:	giveRDFTranslation (RDFCode)
Responsibilities:	Gives a certain RDF code to the user.
Crossed References:	Use case *RDF Instance Translation*.
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The RDF code contained in the variable has been shown to the user.

4.8 Interoperability

Description

The *Interoperability* use case provides functions for exporting internal content and importing data from an *External Semantic Information Source*. Both functionalities have two types of execution: *batch mode* and *runtime mode*. Batch mode executes iteratively at the frequency indicated by the user. Runtime mode executes just once, at the moment of the request. As we see in *Figure 4-35*, there are two use cases that represents the two functionalities, and will be explained in sections 4.8.1 and 4.8.2.

First, we have the *Import Resource* use case, which breaks down in three: *Import FOAF*, *Import Ontoweb*, and *Import BibTeX* . All of them use the *External Semantic Information Source* (from where the data should be imported), and the *Import Ontology* use case from the *Ontology Repository*. The only actor with permission for importing resources is the *Portal Administrator*.

Then, there is the *Export Content*, which breaks down in three use cases: *Export FOAF*, *Export Ontoweb* and *Export BibTeX*. It also uses two use cases from the *Ontology Repository*: *Export OWL* and *Export RDF(S)*. Both *KW Users* and *Guest Users* are capable of exporting content.

Given that the KW semantic portal can import/export content from/to *Ontoweb Portal*, *FOAF*, and *BibTeX Resources*, there will be three kinds of wrappers, one for each one of them.

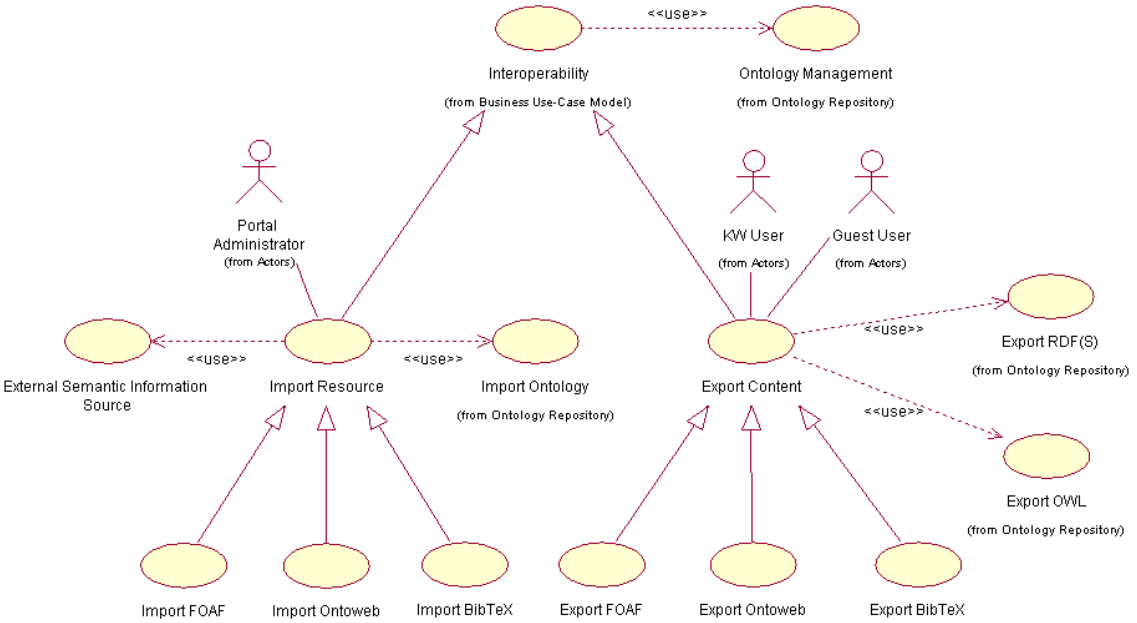

Figure 4-35 *Use case diagram of the Interoperability use case.*

4.8.1 Import Resource

Description

With this use case, the user is capable of importing an external resource. The *Portal Administrator* will provide the location of the external resource and the import mode. If this is batch mode, the user should also provide the frequency of the import. Then, the appropriate wrapper will translate the data (from *Ontoweb*, *FOAF* or *BibTeX*) to an ontology that follows the *WebODE Knowledge Representation Model* and use the *Import Ontology* services if needed (in case of RDF(S) and OWL external resources).As we see in *Figure 4-35*, it uses the *Ontology Management* use case in order to retrieve concepts and instances from the *Ontology Server*.

Flow of events

The flow of events of this use case is depicted in *Figure 4-36*. First, the *Portal Administrator* initiates the importation by adding an external resource to the KW portal. He provides the location of that resource, the import mode (*batch* or *runtime*) and the wrapper to be used in the translation. If the user has selected batch mode, the KW Portal will import the data at the frequency provided. If he has selected runtime mode, the portal will import just once, in the moment of the request.

The real importation is made by the KW portal, as we see in the second operation, in which it retrieves the data from the *External Semantic Information Source*. Then, internally, the wrapper will translate it into an ontology that follows the *WebODE Knowledge Representation Model*.

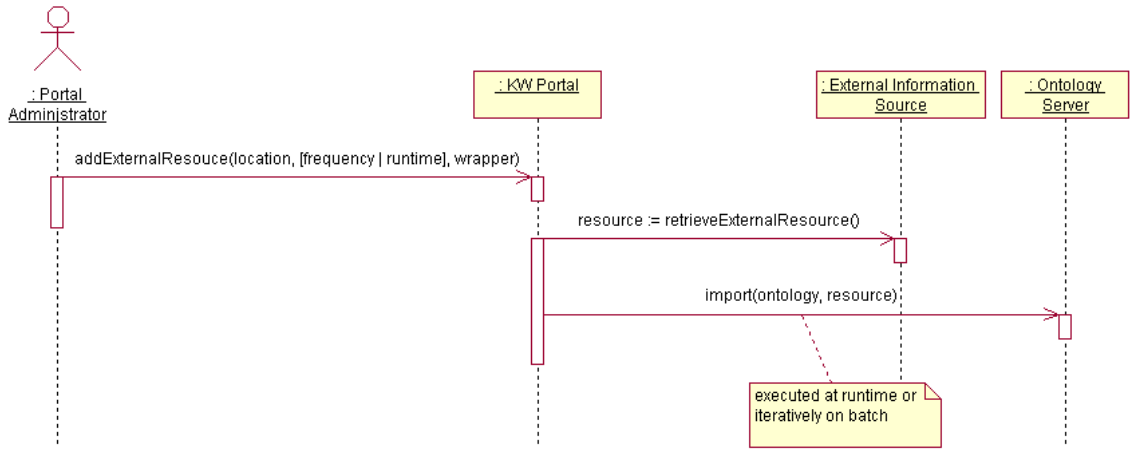

Figure 4-36 *Sequence diagram of the Import Resource use case.*

Architectural Implications

The importation on batch mode implies to create a scheduler of the tasks to be imported.

Contracts

Name:	addExternalResouce(location, [frequency | runtime], wrapper)
Responsibilities:	Adds an external resource to the KW portal, providing its location, the import mode (batch or runtime) and the wrapper to be used in the translation.
Crossed References:	Use case *Import Resource*.
Notes:	
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The external resource (from location) has been added to the KW portal, in batch or runtime mode, and using the wrapper *wrapper* for the translation.

Name:	resource := retrieveExternalResource()
Responsibilities:	Retrieves the data from an *External Semantic Information Source*.
Crossed References:	Use case *Import Resource*.
Notes:	▪ If batch mode, this operation will execute with the frequency indicated. ▪ If runtime mode, this operation will execute just at this moment.
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The variable *resource* contains the data of the *External Semantic Information Source*.

Name:	import(ontology, resource)
Responsibilities:	Imports a resource from an ontology of the *Ontology Server*.
Crossed References:	Use case *Import Resource*.
Notes:	▪ If batch mode, this operation will execute with the frequency indicated. ▪ If runtime mode, this operation will execute just at this moment.
Exceptions:	
Preconditions:	The user is logged in as a *Portal Administrator*.
Postconditions:	The variable *resource* contains the resource imported from the ontology *ontology*.

4.8.2 Export Content

Description

With this use case, the *Guest User* is capable of exporting a certain content to the outside. The *Guest User* will provide the location of the internal resource and the export mode. If this is batch mode, the *Guest User* should also provide the frequency of the export. Then, the appropriate wrapper will translate the data (from an ontology that follows the *WebODE Knowledge Representation Model*) to *Ontoweb*, *FOAF* or *BibTeX* formats.

Flow of events

The flow of events of this use case is quite simple, as we see in *Figure 4-37*. There is just one operation, initiated by the *KW User* or by the *Guest User*, that requests to export internal content, in *batch* or *runtime mode*, and using a certain wrapper for the translation. If the user has selected batch mode, the KW portal will export the data at the frequency provided. If he has selected runtime mode, the portal will export just once, in the moment of the request. Therefore, the wrapper will internally translate the internal content to the supported language.

It is important to mention the fact that the KW portal does not interact here with an external actor, it just supplies information (keeping it internally in a certain language) that may be retrieved by somebody external.

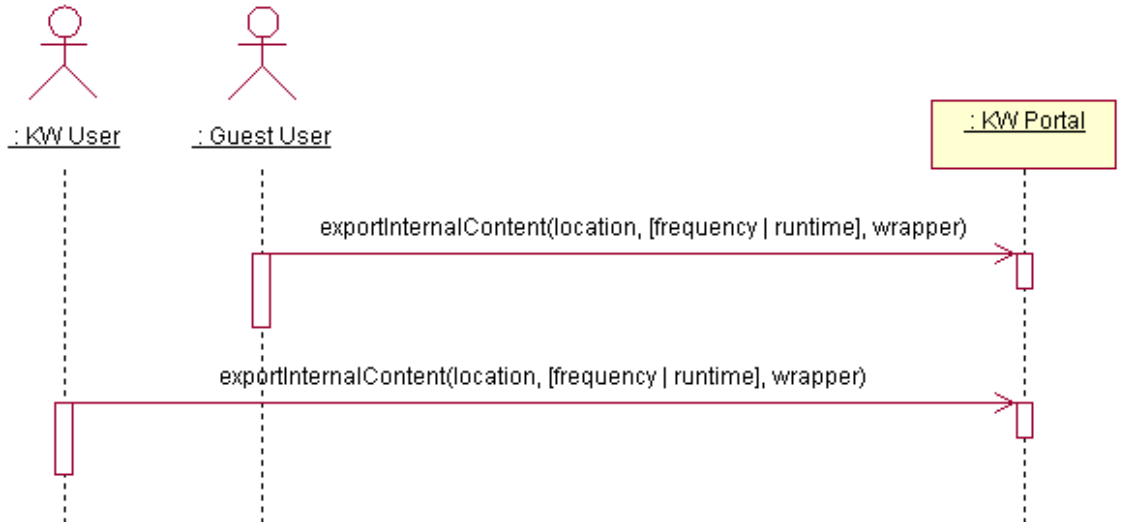

Figure 4-37 *Sequence diagram of the Export Content use case.*

Architectural Implications: None

Contracts

Name:	exportInternalContent(location, [frequency | runtime], wrapper)
Responsibilities:	Requests the export of internal content, in batch or runtime mode, and using a certain wrapper for the translation.
Crossed References:	Use case *Export Content*.
Notes:	If the user (*KW* or *Guest*) has selected batch mode, the KW portal will take *frequency* as the frecuency of the execution.
Exceptions:	
Preconditions:	
Postconditions:	The KW portal has exported an internal content (from *location*) in batch or runtime mode, and using the wrapper *wrapper* for the translation.

4.9 Semantic Navigation Model Management

Description

The *Semantic Navigation Model Management* use case allows the *Web Designer* to manage the navigation models. A navigation model is implemented as an ontology inside the *Ontology Server*. It may be seen as an state diagram, in which the states are represented by concepts of the ontology (which actually are views), and the transitions between states are defined by relations between concepts (which actually are actions between different views). This way, each view has a name, description, precondition to be accomplished to retrieve the view and its location (URL). All these attributes are represented as concept attributes (class attributes) of the views.

The *Semantic Navigation Model Management* provides the following functionalities: add, update and remove a navigation model. As we can see in *Figure 4-38*, the *Semantic Navigation Model Management* only uses another use case, *Conceptualization Management* from the *Ontology Repository*.

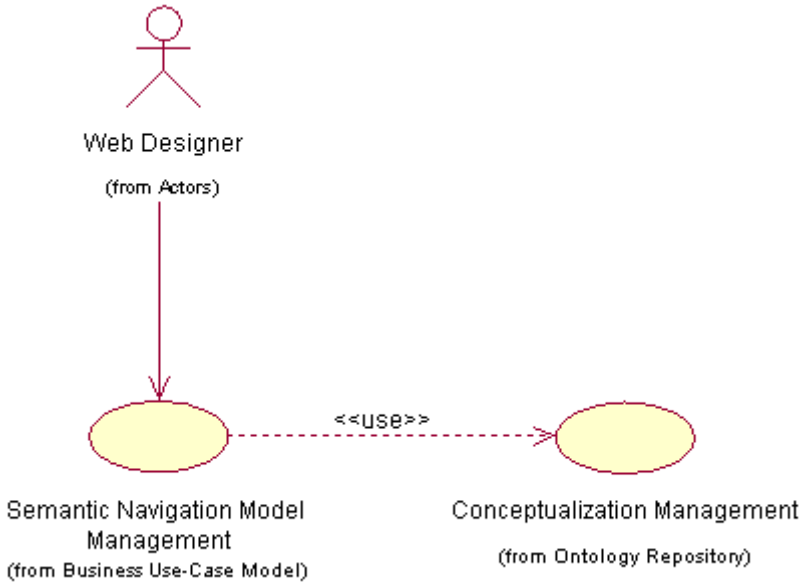

Figure 4-38 *Use case diagram of the Semantic Navigation Model Management use case.*

As we said before, this use case is in charge of managing the navigation models implemented in the KW portal. A navigation model allows the user to navigate correctly through the portal. It is important to mention that we have decided to implement the navigation model together with all its views, so we have also modelled the navigation model as an ontology which is stored inside the *Ontology Server*. This way, changes to be done to any navigation model will mean accessing to the *Ontology Server* to modify the corresponding ontology.

Flow of events
The flow of events of this use case is specified by the sequence diagram depicted in *Figure 4-39*. There we can see that the *Web Designer* can decide whether to create, update or remove a navigation model.

Therefore, when creating a navigation model, the KW semantic portal must access the *Ontology Server* in order to create a new ontology, and the concepts and relations indicated by the user as values.

The same happens when updating a navigation model. The KW Portal must access the *Ontology Server* in order to update the name of an existing ontology, or its concepts and relations as indicated by the values provided by the *Web Designer*.

Finally, when the *Web Designer* decides to remove a navigation model, the only thing the KW portal must do is remove the ontology from the *Ontology Server*.

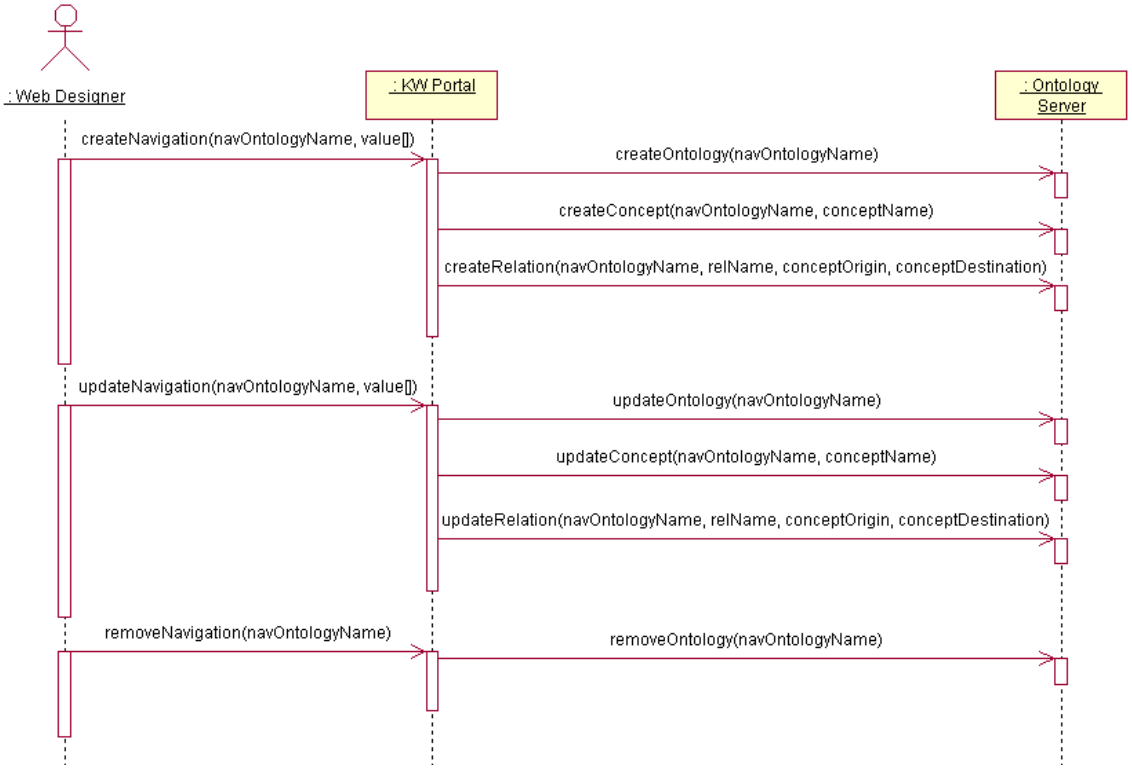

Figure 4-39 *Sequence diagram of the Semantic Navigation Model Management use case.*

Architectural Implications: None

Contracts

Name:	createNavigation(navOntologyName, value[])
Responsibilities:	Creates a new navigation model in the KW portal.
Crossed References:	Use case *Semantic Navigation Model Management*.
Notes:	
Exceptions:	If there is already a navigation model with that name, show a message explaining the error.
Preconditions:	
Postconditions:	A new navigation model called *navOntologyName* has been created in the KW portal with the data contained in *value[]*.

Name:	createOntology(navOntologyName)
Responsibilities:	Creates an ontology in the *Ontology Server*.
Crossed References:	Use case *Semantic Navigation Model Management*.
Notes:	The navigation model is implemented as an ontology inside the *Ontology Server*.
Exceptions:	If there is already an ontology with that name in the *Ontology Server*, show a message explaining the error.
Preconditions:	
Postconditions:	A new ontology called *navOntologyName* has been created in the *Ontology Server*.

Name:	createConcept(navOntologyName, conceptName)
Responsibilities:	Creates a new concept in a certain ontology of the *Ontology Server*.
Crossed References:	Use case *Semantic Navigation Model Management*.
Notes:	The navigation model is implemented as an ontology inside the *Ontology Server*.
Exceptions:	If there is already a concept with that name in that ontology, show a message explaining the error.
Preconditions:	
Postconditions:	A concept called *conceptName* has been created in the ontology *navOntologyName*.

Name:	createRelation(navOntologyName, relName, originConcept, destinationConcept)
Responsibilities:	Creates a new relation in a certain ontology of the *Ontology Server*.
Crossed References:	Use case *Semantic Navigation Model Management*.
Notes:	The navigation model is implemented as an ontology inside the *Ontology Server*.
Exceptions:	If there is already a relation with that name in that ontology, show a message explaining the error.
Preconditions:	
Postconditions:	A relation called *relName* has been created in the ontology *newOntologyName*, from the concept *originConcept* to *destinationConcept*.

Name:	updateNavigation(navOntologyName, value[])
Responsibilities:	Updates a navigation model in the Knowledge Web Portal.
Crossed References:	Use case *Semantic Navigation Model Management*.
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The navigation model *navOntologyName* has been updated in the KW portal with the data contained in *value[]*.

Name:	updateOntology(navOntologyName)
Responsibilities:	Updates an ontology in the *Ontology Server*.
Crossed References:	Use case *Semantic Navigation Model Management*.
Notes:	The navigation model is implemented as an ontology inside the *Ontology Server*.
Exceptions:	
Preconditions:	
Postconditions:	The ontology *navOntologyName* has been updated.

Name:	updateConcept(navOntologyName, conceptName)
Responsibilities:	Updates a concept in a certain ontology of the *Ontology Server*.
Crossed References:	Use case *Semantic Navigation Model Management*.
Notes:	The navigation model is implemented as an ontology inside the *Ontology Server*.
Exceptions:	
Preconditions:	
Postconditions:	The concept called *conceptName* has been updated in the ontology *navOntologyName*.

Name:	updateRelation(navOntologyName, relName, conceptOrigin, conceptDestination)
Responsibilities:	Updates a relation in a certain ontology of the *Ontology Server*.
Crossed References:	Use case *Semantic Navigation Model Management*.
Notes:	The navigation model is implemented as an ontology inside the *Ontology Server*.
Exceptions:	
Preconditions:	
Postconditions:	The relation called *relName* has been updated in the ontology *newOntologyName*, from the concept *originConcept* to *destinationConcept*.

Name:	removeNavigation(navOntologyName)
Responsibilities:	Removes a navigation model from the KW portal.
Crossed References:	Use case *Semantic Navigation Model Management*.
Notes:	
Exceptions:	
Preconditions:	
Postconditions:	The navigation model *navOntologyName* is no longer in the KW portal.

Name:	removeOntology(navOntologyName)
Responsibilities:	Removes an ontology from the *Ontology Server*.
Crossed References:	Use case *Semantic Navigation Model Management*.
Notes:	The navigation model is implemented as an ontology inside the *Ontology Server*.
Exceptions:	
Preconditions:	
Postconditions:	The ontology *navOntologyName* is no longer in the *Ontology Server*.

4.10 Web Designing

Description

As we see in *Figure 4-40*, there is only one actor interacting with this use case. It allows the *Web Designer* to create views with his designing tools and deploy them manually in the KW portal.

Figure 4-40 *Use case diagram of the Web Designing use case*

Therefore, ODESeW will not provide tools for the designing of views.

5 Analysis

The Analysis activity is intended to obtain a high-level system architecture. The main functional blocks and their interactions are identified on this phase. This phase is comprised of the following activities:

- **Integration Environment**, for describing the platforms needed to run the KW Semantic Portal.
- **Software Architecture**, for describing the main elements that comprise the software architecture system.
- **UML class and sequence diagrams**, for detailing the sequence diagrams including the identification of classes and their methods.

This section only contains the first two points of this phase: the integration environment (section 5.1) and the software architecture (section 5.2).

5.1 Integration environment

In this section we will describe the hardware and software platforms needed to run the KW Semantic Portal.

5.1.1 Description of the target integration platform

The platform that will be used for the integration is the following:

- Hardware:
  - Pentium IV 2.4 Ghz
  - 512Mb of RAM
- Software:
  - Windows 2000
  - Professional. Service Pack 3
  - J2SDK 1.4.1_03[1]
  - J2SDKEE 1.3.1[2]
  - Apache Ant 1.5.1[3]
  - AXIS[4]
  - Resin 2.1[5]
  - Oracle 8.0.5
  - Log4J 1.2.8[6]
  - JUnit 3.8.1[7]
  - ActivePerl 5.6.1 build for MSWin32-x86-multithread[8]
  - Red Hat Linux 9[9]
  - WebODE

[1] http://java.sun.com/j2se/1.4.1/download.html
[2] http://java.sun.com/j2ee/1.3/download.html
[3] http://ant.apache.org/
[4] http://ws.apache.org/axis/
[5] http://www.caucho.com/download/index.xtp
[6] http://jakarta.apache.org/log4j/docs/index.html
[7] http://www.junit.org/index.htm
[8] http://www.activestate.com/Products/ActivePerl/
[9] http://www.redhat.com/

5.2 Software architecture

ODESeW has been built in the framework of WebODE, a scalable ontology engineering workbench that gives support to the ontology building methodology METHONTOLOGY.

As shown in *Figure 5-1*, the KW semantic portal is one of the two main front-end applications of the WebODE workbench. The other one is the WebODE ontology editor, which integrates all the ontology editing and management functions of the platform.

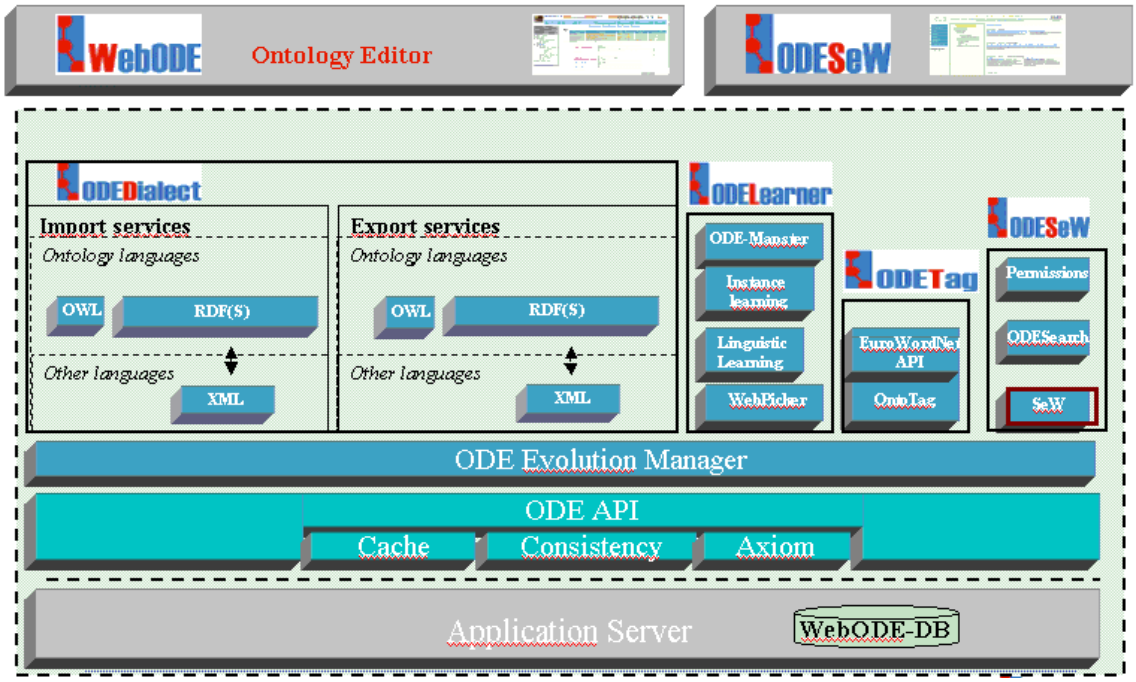

Figure 5-1. *KW Portal architecture*

WebODE is platform-independent, since it is completely implemented in Java. To allow scalability and easy extensibility, it is supported by an application server, so that services can be easily created and integrated in the workbench by means of a management console. One important advantage of using this application server is that it allows deciding which users or user groups may access each of the services of the workbench.

The figure also shows the most relevant services currently available in the WebODE workbench. The core of the WebODE's ontology development services are: the cache, consistency and axiom services, and the ontology access service (ODE API), which defines an API for accessing WebODE ontologies. One of the main advantages of this architecture is that these services can be accessed remotely from any other application or any other instance of the WebODE workbench.

Furthermore, ontologies are stored in a relational database, so they can manage huge ontologies quite efficiently. And it is also easily extensible, so that the database manager can be changed, or any backend system can be plugged in the bottom of the architecture. Finally, WebODE also provides backup management functions for the ontologies stored in the server.

The figure shows that the import, export and evaluation services are running on top of the ontology access service. These services import ontologies from XML, RDF(S) and OWL, to WebODE; and export ontologies from WebODE to XML, RDF(S) and OWL.

Once described the main characteristics of the WebODE workbench, we will proceed to describe the services used by the ODESeW application. To implement ODESeW, we have built

three more services on top of the ODE API, as shown in the right of the figure: *ODESearch*, *permission* and *SeW*.

• *ODESearch* allows querying the WebODE ontologies, by means of keywords or using the attributes of the ontology concepts as templates, as will be explained in section 3.3.

• The *permission* service is in charge of managing security in the access to the concepts, instances and attributes of the ontologies. It will manage both read and write access permissions to the content stored.

• *SeW* gives support to the administration functions of the ODESeW application. It allows selecting which ontologies will be published in the portal, which types of users can access it (administrators, guest users, etc.), how instances in the ontology will be visualized in the portal, etc. These functions are described in section 3.4.

There are many advantages of having built ODESeW on top of the WebODE workbench. First of all, ODESeW can use any of the WebODE workbench services. For example, with the ontology import services we can import other ontologies in the workbench, and these new ontologies can be easily selected for publication in the KW semantic portal. Consequently, we can create a complete new knowledge portal (including its Intranet and its Extranet) in a very short period of time.

Another advantage is that we can edit any of the ontologies published with ODESeW using the WebODE ontology editor, and observe at run-time the modifications in the knowledge portal, which means that there is auto-synching of the portal with respect to the ontology.

6 Conclusions

In this deliverable we have presented the Business Model, the Requirement Analysis Model and part of the Analysis phase of the Knowledge Web Semantic Portal (http://knowledgeweb.semanticweb.org). In order to achieve these activities, we have followed the Rational Unified Process methodology (RUP) [Kruchten, 99], defined in Section 2.

In the Business Model (Section 3) we have identified the actors that interact with the KW portal, the business use cases that represent its functionalities and finally, the business object model, in which we describe separately the ODESeW objects and the WebODE's ones.

The Requirement Analysis Model (Section 4) defines the detailed requirement specification of the ODESeW technology, according to the main groups of functions identified in the former section.

Finally, in Section 5, we have carried out the first two points of the Analysis phase: the integration environment and the software architecture.

7 References

[Annex I] *Knowledge Web, Network of Excellence. Annex 1 "Description of Work"*. November, 2003

[Arpirez et al., 2001] Arpírez JC, Corcho O, Fernández-López M, Gómez-Pérez A. *WebODE: a Scalable Workbench for Ontological Engineering.* First International Conference on Knowledge Capture (KCAP'01). ACM Press (1-58113-380-4). pp: 6-13. October 2001

[Arpirez et al., 2003] Arpírez J.C, Corcho O, Gomez-Pérez A, Fernández-López M (2003) *WebODE In A Nutshell*. AI Magazine (Fall 2003), 24 (3) pp: 37-47

[Blázquez et al., 1998] Blázquez M, Fernández-López M, García-Pinar JM, Gómez-Pérez A (1998) *Building Ontologies at the Knowledge Level using the Ontology Design Environment*. In: Gaines BR, Musen MA (eds) 11th International Workshop on Knowledge Acquisition, Modeling and Management (KAW'98). Banff, Canada, SHARE4:1–15

[Brickley and Guha, 2000] Brickley D, Guha R (2000) *Resource Description Framework (RDF) Schema Specification 1.0* Candidate Recommendation, World Wide Web Consortium

[Corcho et al., 2003] O. Corcho, A. Gómez-Pérez, A. Lopez-Cima, V. López-García, M.C. Suárez-Figueroa *ODESeW. Automatic Generation of Knowledge Portals for Intranets and Extranets.* 2nd International Semantic Web Conference (ISWC2003) Industrial Track. Sanibel Island, Florida, USA. October 2003. The Semantic Web - ISWC 2003 (LNCS2870) pp: 802-817

[Dean and Schreiber, 2003] Dean M, Schreiber G (2003) *OWL Web Ontology Language Reference*. W3C Working Draft.

[E. Valle, M. Brioschi, 204] Emanuele Della Valle, Maurizio Brioschi: Toward a Framework for Semantic Organizational Information Portal. ESWS 2004: 402-416

[F. Bellas et al., 2004] Fernando Bellas, Daniel Fernández and Abel Muiño. A flexible framework for engineering "My" portals. 13th international conference on World Wide Web 2004: 234 - 243

[Fernández et al., 1997] Fernández-López M, Gómez-Pérez A, Juristo N (1997) *METHONTOLOGY: From Ontological Art Towards Ontological Engineering*. Spring Symposium on Ontological Engineering of AAAI. Stanford University, California, pp: 33–40

[Fernández-López et al., 1999] Fernández-López M, Gómez-Pérez A, Pazos A, Pazos J (1999) *Building a Chemical Ontology Using Methontology and the Ontology Design Environment*. IEEE Intelligent Systems & their applications 4(1) pp: 37–46

[Fluit et al., 2003] Fluit C, Sabou M, van Harmelen, F. (2003) *Visualizing the Semantic Web*. Chapter 3: Ontology-based Information Visualization, Springer-Verlag, pp: 36-37

[Gómez-Pérez et al., 1995] Gómez-Pérez A, Juristo N, Pazos J (1995) *Evaluation and assessment of knowledge sharing technology*. In: Mars N (ed) Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing (KBKS'95). University of Twente, Enschede, The Netherlands. IOS Press, Amsterdam, The Netherlands, pp: 289–296

[Gómez-Pérez et al., 2003] Gómez-Pérez A, Fernández-López M, Corcho O (2003) *Ontological Engineering* Springer-Verlag. 2003

[Gómez-Pérez, 1998] Gómez-Pérez A (1998) *Knowledge Sharing and Reuse*. In: Liebowitz J (ed) Handbook of Expert Systems. CRC Chapter 10, Boca Raton, Florida

[J. Hartmann, Y. Sure, 2004] J. Hartmann and Y. Sure. An Infrastructure for Scalable, Reliable Semantic Portals. In: IEEE - Intelligent Systems, pp. 58-65. IEEE, 2004.

[Kruchten, 1999] Kruchten Philippe (1999). *The Rational Unified Process: An Introduction*. Englewood Cliffs, NJ: Prentice--Hall.

[Lassila and Swick, 1999] Lassila O, Swick R (1999) *Resource Description Framework (RDF) Model and Syntax Specification*. W3C Recommendation.

[R. Keller et al., 2004] Richard Keller, Daniel Berrios, Robert Carvalho, David Hall, Stephen Rich, Ian Sturken, Keith Swanson, Shawn Wolfe. SemanticOrganizer: A Customizable Semantic Repository for Distributed NASA Project Teams. ISWC 2004: 767-781

[Rojas, 1998] Rojas MD (1998) *Ontologías de iones monoatómicos en variables físicos del medio ambiente*. Proyecto Fin de Carrera. Facultad de Informática, Universidad Politécnica de Madrid, Madrid, Spain

[Y. Jin et al., 2003] Yuhui Jin, Stefan Decker, Gio Wiederhold. OntoWebber: Building Web Sites Using Semantic Web Technologies. Submitted to the Twelfth International World Wide Web Conference, 20-24

[Y. Lei et al., 2004] Yuangui Lei, Enrico Motta, John Domingue: OntoWeaver-S: Supporting the Design of Knowledge Portals. EKAW 2004: 216-230