Rapid ontology-based Web application development with JSTL

Angel López-Cima[1], Oscar Corcho[2], Asunción Gómez-Pérez[1]

[1]OEG - Facultad de Informática. Universidad Politécnica de Madrid (UPM) Campus de Montegancedo, s/n. 28660 Boadilla del Monte. Madrid. Spain
{alopez, asun}@fi.upm.es
[2]University of Manchester. School of Computer Science. Oxford Road, Manchester, United Kingdom
Oscar.Corcho@manchester.ac.uk

Abstract. This paper presents the approach followed by the ODESeW framework for the development of ontology-based Web applications. ODESeW eases the creation of this type of applications by allowing the use of the expression language JSTL over ontology components, using a data model that reflects the knowledge representation of common ontology languages and that is implemented with Java Beans. This framework has been used for the development of a number of portals focused on the dissemination and management of R&D collaborative projects.

Introduction

Current Web applications can be designed and implemented with a wide variety of programming languages and underlying frameworks, techniques and technologies (.NET, AJAX, Java, PHP, ASP, JSP, JSTL, JDO, COM, J2EE, etc.). These technologies are widely accepted, what eases their reuse in application development, from code snippets to large applications. Besides, most of them are supported by Web authoring tools (e.g., Macromedia, FrontPage), making it easier for non-experts to create Web applications.

In the context of the Semantic Web there is already a large diversity of tools for editing and managing ontologies, annotating resources, querying and reasoning with them, etc. These tools constitute the basic building blocks of the underlying infrastructure for ontology-based application development [8]. In the past few years there have been also many efforts focused on the provision of technology for the rapid development of ontology-based Web applications. [3] describes the most relevant technologies according to a set of characteristics like their model storage, API paradigm, supported serialization formats, query languages, etc. Most of these efforts are oriented towards the easy management of RDF resources, statements and models, with a lack of approaches for the management of the most usual ontology components (classes, properties, hierarchies, and individuals). This is classified as the *ontology-centric view* in [3], and it is only supported by some of the most common Java APIs (Sesame, Jena, Protégé-OWL, etc.), which operate at a lower infrastructure level, and two other more user and Web oriented APIs, such as Spiral[1] and RAP [16], integrated with .NET and PHP respectively.

On another dimension, authoring tools are also being created for non-experts. Some of the most recent work on this side has been focused on the development of semantic wikis (e.g., [10], [14]), which allow non-experts to include ontology-based annotations (and in some cases even ontology term definitions) of the Web pages that they edit, and on the development of semantic desktops (e.g., [4], [11], [6], [1]).

In this paper we present how the ODESeW framework [10] combines both types of approaches, scripting languages for developers that want to create ontology-based Web applications and ontology-based Web authoring tools for non-experts, in a common framework. On the first aspect, ODESeW proposes the use of Java Beans for the management of ontology components and ontology-based annotations, and JSTL default and extended tags for the visualisation of ontology-based annotations. These technologies can be easily combined with other standard visualisation and navigation ones, hence improving reuse and maintainability. On the second aspect, ODESeW provides forms that are automatically generated from the underlying ontologies and allow a more structured edition of annotations.

The structure of this paper is organized as follows. Section 2 describes the ODESeW architecture. Sections 3, 4 and 5 focus on the most relevant components of this architecture together with some examples from the Person and Organization ontologies in R&D collaborative project. Section 6 provides some conclusions and future work.

---

[1] http://www.semanticplanet.com/library/Spiral/HomePage?from=RdfLib

The ODESeW Architecture

The ODESeW architecture, shown in Fig. **1**, is based on the Model-View-Controller design pattern [7], which is currently widely used for developing Web applications. This pattern is useful to develop applications where the same information may have several visualisations. It divides functionality among three types of objects: the model, the view and the controller.

- The model represents business data, and the business logic that governs its access and modification. In ODESeW, business data is represented with three models, coordinated by the Data Model Manager:
  - The Data Model, which contains domain information, represented by means of ontologies.
  - The User Model, which stores user profiles.
  - The External Information Gateway (EIG), which accesses external resources and annotates them with domain information.
- Views render the contents of models. They access data from the model and specify how that data should be presented. They also update data presentation when their corresponding model changes, and forward user inputs to the controller.
- The controller defines the application behaviour. It dispatches user requests (button clicks, menu selections, form input texts, etc.), also known as user gestures or actions, interpreting and mapping them into actions to be performed by the model. These actions can perform navigation of the user from one view to another and also execute business logic of the application.

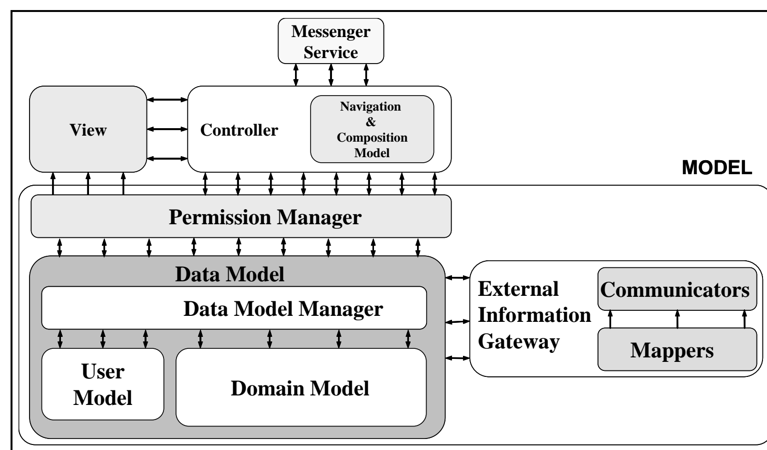

Fig. 1. ODESeW Architecture

The ODESeW architecture also includes other services, such a the Messenger Service, in charge of communicating ODESeW applications with external applications. For instance, the controller may send a message to external applications when an instance is created, updated or removed, or receive a message when an external information source changes. Another important component is the permission manager, which filters user requests and responses according to the user that is accessing the application. The description of how these components work is out of scope of this paper, and can be found in [12].

Data Model

The ODESeW data model uses a frame-based ontology representation model, based on the WebODE [2] knowledge representation ontology, which is the underlying ontology repository used by ODESeW. Fig. 2 shows the components in this model (concepts, attributes, relations, instances, etc.) and the relations between them, and Fig. 2 shows all the attributes and relations of each component of the model. This model has been implemented with Java Beans [9], what enables the use of a large amount of third party Web application development technologies.

The Data Model Manager manages connections to the ontology repositories that store the domain application and the user models. This includes starting up connections, managing their lifetime, and identifying the relevant ontologies to be used. Since the current implementation connects to WebODE, ontologies implemented in a range of languages (including RDF Schema and OWL) can be imported in the system and used.

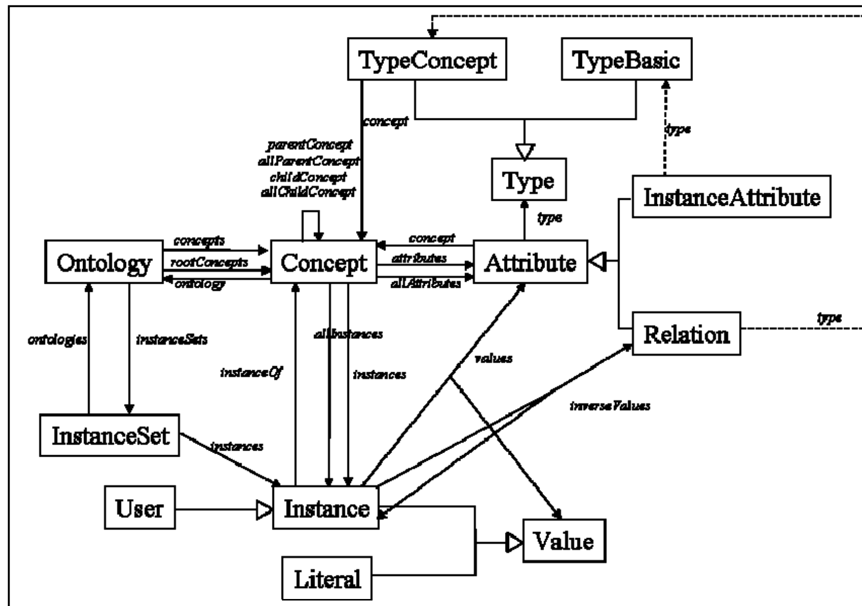

Fig. 2. ODESeW Data Model diagram

View

ODESeW views are implemented with XHTML, for static pages, and with JSP [15], for dynamic pages. The JSP pages use a combination of standard JSTL tags with the Expresion Language [15], and of custom tags to access information from the data model, as illustrated in Fig. 3 and Fig. 5.

The code included in Fig. 3 is included in the page *instance.jsp*, and uses JSTL to execute a set of commands that display the information about a person. The commands are executed are the following: 1) get the instance “Angel López-Cima” from the ontology “Person Ontology”; 2) print the instance name, 3) iterate for all pairs <Attribute, Value[]> of instance values; 4) set the attributes *attribute* and *values* from the pair <Attribute, Value[]>; 5) store in the attribute *multivalue* if the instance has more than one value for the attribute *att*, and in that case create an enumeration list with the following instructions; 6) print the attribute name of the pair <Attribute, Value[]>; 7) iterate for all values in the pair <Attribute, Value[]>; 8) and print a value. Fig. 4 is the page that results from the execution of the code inside this page. To display any other instance, no matter which ontology it comes from or which concept it belongs to, the web developer only needs to change the values in step 1) in Fig. 3.

```
<sew:ontology var="persOntology" name="Person Ontology">            ①
  <sew:instance var="instance" name="Angel López-Cima"/>
</sew:ontology>

<h1>${instance.name}</h1>                                            ②
  <table>
  <c:forEach var="pairAttVals" items="${instance.values}">          ③
    <c:set var="attribute" value="${pairAttVals.key}"/>             ④
    <c:set var="values" value="${pairAttVal.value}">
    <c:set var="multivalue" value="${fn:length(values)>1}"/>        ⑤
  <tr>
    <th align="left" valign="top">${attribute}:</th>                ⑥
    <td align="left">
    <c:if test="${multivalue}"><ul></c:if>
    <c:forEach var="value" items="${values}">                       ⑦
      <c:if test="${multivalue}"><li></c:if>
      ${value}                                                      ⑧
      <c:if test="${multivalue}"></li></c:if>
    </c:forEach>
    <c:if test="${multivalue}"></ul></c:if>
    </td>
  </tr>
  </c:forEach>
  </table>
```

Fig. 3. *Instance.jsp*, a simple view of an instance.

Angel López-Cima

Full Name:	Angel López-Cima
Photo:	/semanticportal/servlet/download?ontology=Person+Ontology&concept=Phl
e-mail:	alopez@fi.upm.es
Date of Birth:	16/10/1976
Role:	Portal Administrator
Country:	Spain
City:	Madrid
Zip code:	28027
Street Address:	Campus Montegancedo, sn
Telephone:	+34-91 336 3670
Gender:	Male
participates in:	Knowledge Web
belongs to:	Universidad Politécnica de Madrid
works in:	WP1.6: Semantic Portal Structure
is involved in:	• T1.2.1 Utility of ontology-based tools • T1.6.1 Semantic portal analysis requirements and design • T1.6.2 Semantic portal ontology prototype development • T1.6.3 Semantic portal prototype development • T1.6.4 Semantic Portal Unit and integration testing • T1.6.5 Content annotation and management

Fig. 4. Simple visualization of an instance of the concept Person.

The code included in Fig. 5 executes the following commands: 1) get the concept "Person" from the ontology "Ontology Person"; 2) print the name of the concept; 3) iterate for all direct and indirect instances of the concept; 4) print the name of an instance. Fig. 6 is the page that results from its execution. To display any other concept, the web developer only needs to change the value in 1) in Fig. 5.

```
<sew:ontology var="persOnto" name="Person Ontology">          ①
  <sew:concept var="concept" name="Person"/>
</sew:ontology>
<h1><c:out value="${concept}"/></h1>                          ②
<ul>
<c:forEach var="instance" items="${concept.allInstances}">    ③
  <li>${instance.name}</li>                                   ④
</c:forEach>
</ul>
```

Fig. 5. *Concept.jsp*, a simple view of a concept

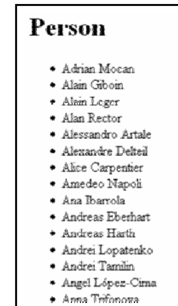

Fig. 6. Simple visualization of the list of instances of the concept Person.

Controller

The ODESeW Controller receives user requests, which contain the actions to be performed, and completes or checks requests with the information model in the Data Model (including both the domain and the user models). Then it reads and executes the navigation and composition model, described below, and returns the next view that should be rendered for the user.

We will describe first the ODESeW Navigation and Composition Model, and then the steps followed by the Controller to execute actions.

The Navigation and Composition Models

The **navigation model** represents the navigation of a user through the application. This model is explicitly separated from the design of views so that changes in the navigation do not affect the implementation of views. Besides, it allows representing declaratively the navigation of a user, enabling in this way an easy study of the behaviours of the user of an application.

The navigation model is a directed named graph in which nodes represent views and edges represent navigation actions from one view to another.

- Nodes have 2 attributes: "*precondition*" and "*view URL*". The first one specifies preconditions to allow the execution of a view and the second one specifies the location of the view. If the precondition is empty, it is considered to be true. If the *viewURL* attribute is empty this means that the view is abstract. That is, it is a view that cannot be rendered directly and has to be specialised by other views so as to be used by the Controller.
- Edges identify actions that can be performed from a view. Besides redirecting users from a view to another, edges are attached to a task execution: instance edition, instance removal, message sending, etc. An edge may not have an origin node, that is, the action represented by this edge can be executed from all pages, represented or not in the navigation model. Besides, edges can be concatenated to perform different tasks in a navigation step.

The navigation model also allows describing specialisation/generalisation relations between two views (defined with the subclass-of relationship). A view is a specialisation of another if it visualises the same content as its parent view but providing more specific visualisation items. For instance, a default view may be used to render any instance and for other more specific instances, such as instances of persons, instances of publications, etc., other more specific views can be created.

Fig. 7 shows an example of a navigation model with 9 views defined and several types of actions and specialisation/generalisation relations defined between them.

The **composition model** is similar to the navigation model, though its rationale is different: it allows including a set of views inside another and is normally used when complex sets of information have to be presented at once.

One common example of the use of the composition model is for displaying a default view that render attribute values and for other more specific types of values, such as e-mail addresses, URLs, image files, sound files, video files, etc., other more specific views can be created.

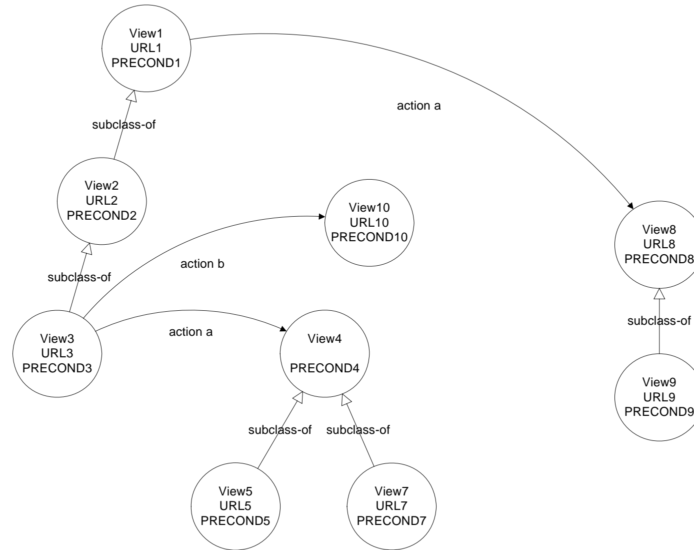

Fig. 11. Example of a navigation model

The elements used in the composition model are the same as those for the navigation model: views are represented as nodes, with the attributes "*precondition*" and "*view URL*"; views can be specialized with other views; and actions are represented as edges. The only constraint is the type of actions that can be represented in this model, which only consists in the action of inclusion of a view inside another.

Controller Execution

Actions received by the Controller contain two elements: task and control flow operation. The task is the specific operation to be performed, while the control flow operation specifies what to do after the execution of the task.

Developers can use any of the default tasks provided by ODESeW or create new ones, either from scratch or by reusing and extending any of the default ones. The following default tasks are available: *sewView*. It renders the view specified in the user request by redirecting users to it; *sewRemove*. It deletes the set of concept and relation instances specified in the user request; *sewEdit*. It updates or creates the set of concept and relation instances specified in the user request; *sewSearch*. It searches for a set of concept and relation instances that satisfy the query; *sewRouter*. It is used to execute another action from a list specified in the user request. These actions have a guard condition, and the sewRouter task selects the first one whose guard condition is satisfied; *sewLogin*. It authenticates a user and loads his/her profile in the user session.

With respect to control flow operations, there are four available: *Forward*: the user request is concatenated to another action or view; *Redirect*: the user request ends and a new user request starts. This new request consists in showing another view or performing another action; *Include*: the execution of a new action or view is included in the original view or action, so that it will be performed later; *Empty*: the execution ends without any more control flow actions. In fact, a view is actually defined as a rendering action, optionally followed by other additional include actions, and which has an empty control flow at the end.

When a user requests an action from a view, the Controller executes the navigation model, following these steps:

1. Identify the view from which the user request is originated, and find it in the navigation model.
2. Find the requested action in the source view. The action can be defined explicitly in the source view or in any of its ancestor views.
3. Select the target view for the requested action. In the navigation model, an action applied to a specific view may have several target views, and at least one of them has to be selected. To perform this selection, the Controller verifies whether the precondition of any of the target views specified in the

action is satisfied given the request parameters. If no precondition is satisfied, an exception raises and the error is reported to the portal administrator.

4. Find whether any of the specialisations of the selected target view is also valid. Once the controller found a valid candidate view, it will try to find another one among its specialisations. To do this, the Controller checks the preconditions of the view specialisations. If any of them is satisfied, that view is a new valid candidate view and the Controller repeats this step with its children views, until a valid view does not have more specialisations or none of the preconditions of its specialisations are satisfied. The last valid candidate view is the final target view.

Let us see an example based on the navigation model presented in Fig. 11. Let us assume that the user requests the **action a** from the view **View3**, and that the parameters of the request satisfy the preconditions **Precondition4**, **Precondition8** and **Precondition9** and do not satisfy the preconditions **Precondition5** and **Precondition7**.

First, the Controller finds the source view (**View3**). Taking into account that the user wants to perform **action a**, the possible candidate views are the **View4** and **View8**.

The first candidate to be checked is **View4**. However, **View4** is abstract, so the Controller has to check the preconditions of its specialisations (**View5** and **View7**). Neither of them satisfies the preconditions, so View4 nor its specialisations are valid target views. Hence, **View4** is discarded by the Controller and the next candidate view (**View8**) is analysed. The Precondition8 is satisfied, hence the **View8** is a valid candidate view. Then, the Controller starts looking for its specialisations (**View9**). The precondition of **View9** is also satisfied and, since **View9** does not have specialisations, the final target view for the execution of action from **View3** is the **View9** (see Fig. 12).

Both models, the navigation and composition, can be included and executed under the technologies Struts[2] or JSF [5].

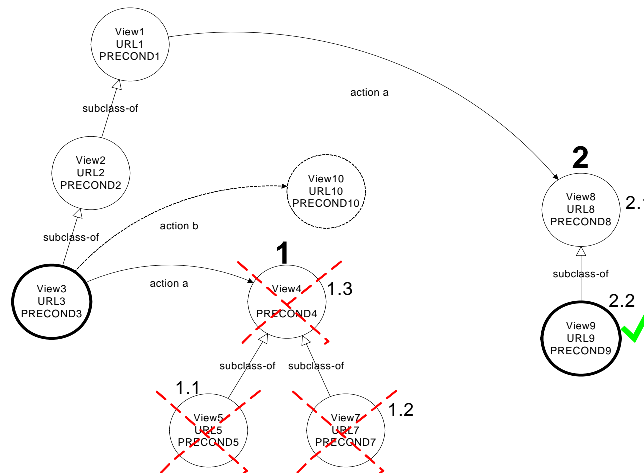

Fig. 12. Example of a navigation model execution.

Examples of composition and navigation

Now we show how we can improve the visualisation of the examples presented in Fig. 3 and in Fig. 5 by including specialised visualisations for images, emails and relations to other instances, using the composition and navigation models described above.

Using the Composition Model, we include a hierarchy of visualization of values where each node of the hierarchy is a type of attribute and a visualization of instances inside the visualization of a concept that lists all instances in a concept Fig. 15 shows a Navigation Model for visualizing values in an instance and instances in the list of instances from a concept and Fig. 16 it shows the JSP code that visualized each type of attributes and the instance.

Executing the page presented in the Fig. 3, but substituting the line marked with number 8) with the following line:

```
<sew:out action="includeView" value="${value}" attribute="${att}"/>
```

and Fig. 5, but substituting the line marked with number 4) with the following line:

```
<sew:out action="includeView" value="${instance}"/>
```

the portal displays to the user the visualization presented in Fig. 17 in the left side.

---

[2] http://struts.apache.org/index.html

When the web server executes the page *instance.jsp* and finds the tag `<sew:out>`, the ODESeW controller looks for the best visualization that matches the original view *instance.jsp*, executing the action *includeView* with the parameters depending on the attribute of the value and the actual value to be displayed. And when the web server executes the page *concept.jsp* and finds the tag `<sew:out>`, the ODESeW controller includes *refInstance.jsp* inside the page *concept.jsp*.

In this way, web developers delegate to the Composition Model how to visualize each type of information, saving a lot of time that would be needed to create a large set of if-then-else or switch-case commands in all dynamic pages that are used to visualize, in a specific format, attribute values. In the last example, if we remove the lines marked by 1) in Fig. 3 and in Fig. 5 , the view is fully reusable for displaying any instance and concept, and if the user click on any destination instance, ODESeW executes the same *instance.jsp* but displaying another instance, as shown in Fig. 17.

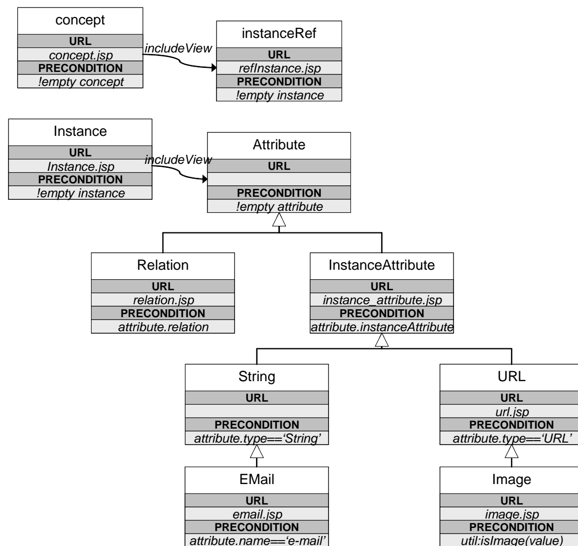

Fig. 15. Composition Model for value visualizations

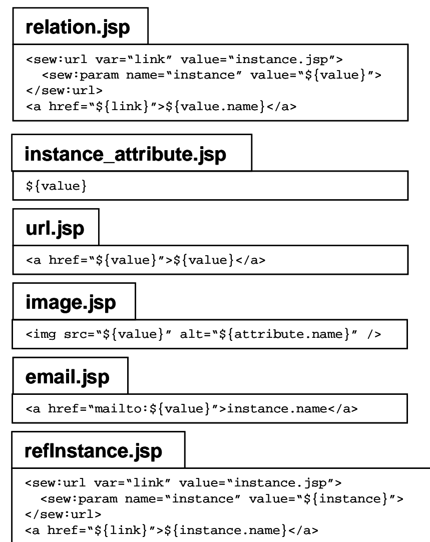

Fig. 16. Value visualizations

Now, the *instance.jsp* and *concept.jsp* pages can be used to create generic views for displaying any instance and concept in the Data Model. However, in most of the cases, the generic view is not what the web developer wants to display in all cases. ODESeW allows web designers to create specific visualizations for different types of information in the Data Model in the same way as it is done with the Composition Model, but using the Navigation Model instead.

In the Knowledge Web portal, we set a navigation model with specific views for displaying persons involved in the project, organization partners in the project, instances of persons and instances of organizations. The resulting navigation and visualization is presented in Fig. 18 and the result of their execution is shown in Fig. 19.

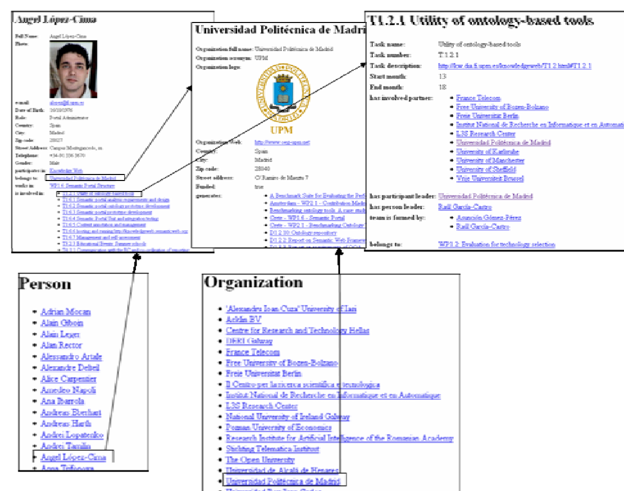

Fig. 17. Visualization of different instances with the same *instance.jsp* and *concept.jsp*

To execute the navigation through the global navigation action *viewTerm*, we substitute the value of the attribute *value* of the tag `<sew:out>` of the files *relation.jsp* and *refInstance.jsp* shown in Fig. 16 from the value *instance.jsp* to the value */sew/viewTerm* (e.g. `<sew:out var="link" value="/sew/viewTerm">…`). In this way, when the portal receives a request for executing the action *viewTerm*, and the parameters contain a class or subclass of *Organization*, then it shows the page */concept/organization.jsp*. If the parameters contain a class or subclass of *Person*, then it shows the page */concept/person.jsp*. If the parameters contain any other class, then it shows the generic concept visualisation page *concept.jsp*. Similarly this will happen with instances of *Organization*, *Person*, etc.

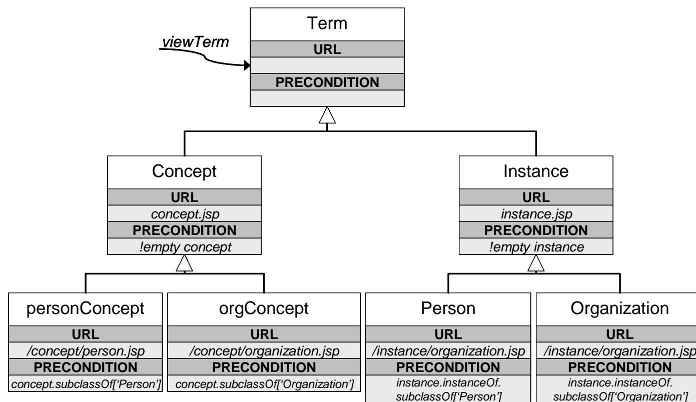

Fig. 18. Navigation Model

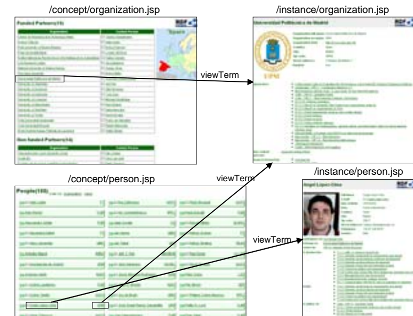

Fig. 19. Visualization of different objects in the Data Model.

Instance edition

ODESeW also provides a set of tags that help to create specific forms for editing instances, together with actions to store those modifications inside the Data Model. The set of custom JSTL tags provided by ODESeW are extensions of the well-known HTML tags for creating form objects, as shown in Table 2.

In the case that the form is editing an instance or a group of instances, all the ODESeW tags of the HTML form object have 3 new attributes: *instance* (here the web designer specifies which instance is being edited); *attribute* (here the web designer specifies which instance attribute is being edited); and *mode* (here the web designer specifies whether the value input by the user will be appended to the existing values or will replace the old values stored in the ontology). Besides, in this case, the attribute *value* is used to specify a destination instance of a relation.

Table 2. ODESeW tag for HTML forms

Tag	Attributes	Description
<sew:form>	Specification of a form and how information will be submitted.	
	name, action, enctype, method, target, all event triggers	Standard attributes from the <form> tag
<sew:input> <sew:textarea> <sew:button> <sew:select>	Object form: text input, checkbox, option, button, text area, a combo list and a multivalue list	
	accept, accesskey, align, alt, border, checked, cols, datafld, datasrc, disabled, id, ismap, maxlength, name, readonly, rows, size, src, style, tabindex, type, usmap, all event triggers	Standard attributes from the <input>, <textarea>, <button> and <select> tag.
	value	Standard attribute, but it can contain any term of the Data Model.
	instance	Sets the instance that is being edited
	attribute	Sets the instance attribute being edited
	mode	Sets whether the value in this tag will be appended [`append`] in the instance or will be used to replace the old values [`update`].
<sew:option>	Option in a combo box	
	disabled, label, selected	Standard attributes from <option> tag
	value	Standard attribute, but it can contain any term of the Data Model.

In the case that the form is requesting an attribute that contains a term of the Data Model (ontology, concept, attribute or instance), the attribute *value* can contain one of these terms, which is passed as a parameter to the action of the form.

These tags include a set of *javascript* functions that are in charge of generating hidden controls in the form that are used in an editing instance action from ODESeW to collect the values of the instances entered in the form. These *javascript* functions do not generate noise behaviour in the generated HTML and allow web designers to include their own *javascript* functions as usual.

Fig. 22 shows the composition model for editing instances, and Fig. 24 and Fig. 25 show that forms are specified very similarly to how they are done in the normal visualization shown in Fig. 19, and that they can generate a generic form for any instance.

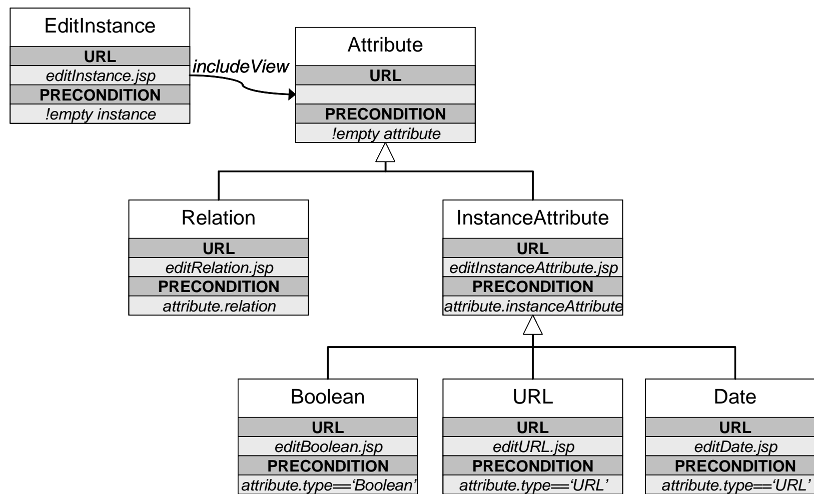

Fig. 22. Composition Model for editing instances

The code include in the Fig. 24 executes the following commands: 1) sets the parameter "*instance*" of the form as a new variable of the form with the name "*instance*" and this new instance is an instance of the concept represented by the variable "*concept*"; 2) prints a text field in the form asking for the name of the instance represented in the variable "*instance*"; 3) iterates among all attributes of the concept of the instance; 4) prints the name of the attribute; 5) prints the name of the type of the attribute; 6) prints the minimum and the maximum cardinality of the attribute; 7) calls the *includeView* action specifying the instance and the attribute to include an HTML object form to edit values for that attribute in that instance. The command in 7) calls the composition model shown in Fig. 22 iteratively among all attributes of the concept of the instance and presents as a result the visualization shown in Fig. 25.

```
<sew:form name="form" action="/sew/sewEdit" debug="false">
  <sew:instance name="instance" var="instance" instanceOf="${concept}"/>
<table border=1>
  <tr>
    <td><b>Instance Name:</b></td>
    <td colspan="3">
      <sew:input type="text" name="instanceName" size="80"
instance="${instance}" attribute="name"/>
    </td>
  </tr>
  <tr>
    <th>Attribute</th>
    <th>Range</th>
    <th>Cardinality</th>
    <th>Value</th>
  </tr>
  <c:forEach var="att" items="${concept.allAttributes}"
varStatus="status">
    <c:set var="att" value="${att.key}"/>
  <tr>
    <td>${att.name}</td>
    <td>${att.type.name}</td>
    <td>(${att.minCardinality}-${att.maxCardinality})</td>
    <td>
      <sew:out value="${instance}" type="${att}"/>
    </td>
  </tr>
  </c:forEach>
</table>
<input type="submit">
</sew:form>
```

Fig. 24. editInstance.jsp for editing instances.

Instance Name:	Angel López-Cima		
Attribute	**Range**	**Cardinality**	**Value**
Full Name	String	(1,1)	Angel López-Cima Clear
Photo	URL	(0,1)	/semanticportal/servlet/c <- -> URL of the file: http:// Or upload a file: upload
e-mail	String	(0,N)	alopez@fi.upm.es <- ->
Homepage	URL	(0,1)	<- -> URL of the file: http:// Or upload a file: upload
Date of Birth	Date	(1,1)	16/10/1976
Role	String	(1,N)	Portal Administrator <- ->
Country	String	(1,1)	Spain Clear
City	String	(1,1)	Madrid Clear
Zip code	String	(0,1)	28027 Clear
Street Address	String	(1,1)	Campus Montegancedo, sn Clear
Telephone	String	(0,1)	+34-91 336 3670 Clear
Fax	String	(0,1)	Clear

Fig. 25. Visualization of editInstance.jsp

Conclusions and future work

In this paper we have presented how the ODESeW framework allows the rapid development of ontology-based Web applications using standard Web development technologies, such as Java Beans, custom JSTL tags and the Expression Language. This allows developers that are familiar with those technologies to access ontology-based information to generate ontology-based applications easily. The main advantages are on the visualisation of ontology components (including instances), on the advanced navigation and composition models, and on the edition of instances using standard HTML form edition technologies.

This framework has been used to generate several Semantic Web portals: Knowledge Web (EU Network of Excellence) at http://knowledgeweb.semanticweb.org; OntoGrid (EU project portal) at http://www.ontogrid.eu; NeOn (EU project portal) at http://droz.dia.fi.upm.es/neon; Ontology Engineering Group (research group portal) at http://www.oeg-upm.net; and Red Temática en Web Semántica (Spanish Network) at http://www.redwebsemantica.es.

Most of the future work on this framework will be devoted to the development of the External Information Gateway (EIG). This component accesses external resources, annotates them with the domain ontologies in the ODESeW data model and gives access to these annotated resources as if they were part of the internal ODESeW data model. The specification and design of this component are already finished, and the implementation is currently in progress.

Acknowledgements

This work has been supported by the EU IST Network of Excellence Knowledge Web (FP6-507482).

References

[1] A. Cheyer, J. Park, R. Giuli. “IRIS: Integrate. Relate. Infer. Share”. ISWC Workshop on Semantic Desktop. 2005.

[2] JC. Arpírez, O. Corcho, M. Fernández-López, A. Gómez-Pérez. “WebODE in a nutshell”. AI Magazine 24(3):37-48. Fall 2003

[3] C. Bizer, D. Westphal. “Developers Guide to Semantic Web Toolkits for different Programming Languages”. http://sites.wiwiss.fu-berlin.de/suhl/bizer/toolkits/. Last updated: January 2007.

[4] D. Karger, K. Bakshi, D. Huynh, D. Quan, V. Sinha. “Haystack: A General Purpose Information Management Tool for End Users of Semistructured Data”. CIDR 2005

[5] E. Burns, R. Kitain. “JavaServer Faces”. JSR-000252. http://jcp.org/en/jsr/detail?id=252

[6] G. Tummarello, C. Morbidoni, M. Nucci, “Enabling Semantic Web communities with DBin: an overview”, Proceedings of the Fifth International Semantic Web Conference ISWC 2006, November 2006, Athens, GA, USA

[7] E. Gamma, R. Helm, J. Vlissides, R. Jhonson. “Design Patterns: Elements of Reusable Object-Oriented Software”. Boston: Addison-Wesley, 1995.

[8] R. García-Castro, MC Suárez-Figueroa, A. Gómez-Pérez, D. Maynard, S. Costache, R. Palma, J. Euzenat, F. Lécué, A. Léger, T. Vitvar, M. Zaremba, D. Zyskowski, M. Kaczmarek, M. Dzbor, J. Hartmann, S. Dasiopoulou. “D1.2.4 Architecture of the Semantic Web Framework”. Knowledge Web technical report, December 2006.

[9] G. Hamilton. “JavaBeans v1.01”. 1997 http://java.sun.com/products/javabeans/

[10] J. Fischer, Z. Gantner, S. Rendle, M. Stritt, L. Schmidt-Thieme. “Ideas and Improvements for Semantic Wikis”. 3rd European Semantic Web Conference (ESWC 2006), Budva, Montenegro.

[11] L. Sauermann. “The Gnowsis Semantic Desktop for Information Integration”. IOA Workshop of the WM2005 Conference. 2005

[12] A. López-Cima, O. Corcho, A. Gómez-Pérez.. “A platform for the development of Semantic Web portals”. In: Proceedings of the 6th International Conference on Web Engi-neering (ICWE2006). Stanford, July 2006.

[13] A. López-Cima, O. Corcho, MC. Suárez-Figueroa, A. Gómez-Pérez. “The ODESeW platform as a tool for managing EU projects: the KnowledgeWeb case study”. In: Proceedings of the 15th International Conference on Knowledge Engineering and Knowledge Management Managing Knowledge in a World of Networks (EKAW2006). Podebrady, October 2006.

[14] M. Krötzsch, D. Vrandečić, M. Völkel. “Wikipedia and the Semantic Web - The Missing Links”. Proceedings of the WikiMania2005.

[15] P. Delisle, J. Luehe, M. Roth. “JavaServer Pages”. JSR-000245. http://jcp.org/en/jsr/detail?id=245

[16] R. Oldakowski, C. Bizer, D. Westphal. „RAP: RDF API for PHP”. ESWC2005 workshop on Scripting for the Semantic Web (SFSW2005). Heraklion, Crete, May 2005.