Engineering Grounded Semantic Service Definitions from Native Service Specifications

Yu Cao

A dissertation submitted to the University of Dublin, Trinity College

in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

September 2007

DECLARATION

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Name: Yu Cao

Date: 14th September, 2007

PERMISSION TO LEND AND/OR COPY

Name: Yu Cao

Date: 14th September, 2007

ACKNOWLEDGEMENTS

Many thanks are due to my supervisor, Dr. David Lewis, for the considerable time spent assisting me on this project, and for all the valuable advice and guidance offered. Likewise, I would like to thank Kris McGlinn and Ian O'Keefe, for all their work and helpful feedback.

To all my family and friends, and especially my fellow Ubicom classmates, I would like to thank you for all your support and encouragement throughout the course of this dissertation.

Yu Cao

University of Dublin, Trinity College

September 2007

ABSTRACT

The Web Ontology Language for Services (OWL-S) is a semantic markup language for web services to facilitate the automation of web service discovery, invocation, and composition and to improve interoperability. Currently the standard way of generating semantic web service is to convert a Web Service Description Language (WSDL) definition obtained from a web service interface to an OWL-S definition. But the generated OWL-S definition includes no semantics. Semantic information needs to be added manually. Another issue is that there are no links between Java interfaces and generated OWL-S definitions, so maintenance is hard to perform.

This dissertation addressed the problem by developing a tool to help software engineers retrieve and add semantic information from Javadoc to OWL-S definitions from an engineering perspective. An Eclipse plug-in called Semantics Editor was implemented. Semantics Editor shows the roundtrip of identifying concepts and association properties from Javadoc, converting Java classes to OWL-S definitions and adding semantic information to OWL-S definitions. The tool also generates and visualizes a link between Java classes and OWL-S definitions so that it becomes traceable between Java classes and OWL-S definitions.

The evaluation focused on usability of user interface of the tool. A series of tests were designed and conducted. The evaluation results shows that the user interface of the tool reached basic functions but still needs to be improved a lot. While the research results were quite successful and most of the questions posed by the research objectives have been answered, this project still has a big potential for further development.

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

This chapter introduces the dissertation topic and explores the motivation behind the work. It is followed by an examination of the objectives to be achieved by the project, and concludes with a summary of the document structure.

1.1 Motivation

In recent years semantic web service has gained a lot of attention as a means to enable automatic web service discovery, invocation, composition and to also to improve interoperability. The Web Ontology Language for Services (OWL-S) [1] is a semantic markup language for web services to facilitate the automation of these tasks.

A standard way of generating semantic web service is to convert a Web Service Description Language (WSDL) [2] definition obtained from a web service interface to an OWL-S definition. This conversion process [3] only captures information which is contained within the WSDL definition. Therefore, the generated OWL-S definition includes no semantics inside. For example, when an input is referred to a concept, that concept is just a subclass of thing. In order to take full advantage of the functionality of the OWL-S definition, extra semantic information needs to be added manually.

According to normal J2EE development methodology [4], development and deployment phases are done by different software engineers. Deployers do not have enough information to restore missing semantic information after conversion from Java interfaces to OWL-S definitions. Thus it slows the growth of semantic web services. So adding the extra semantic information during the development phase so that enough semantic information can be provided to deployers is recommended to the software developers.

The extra semantic information can be modeled in a variety of formats, e.g. UML [27], XML [28] , Javadoc [16] etc. Some related work has already been done to obtain semantic information from multi-source artifacts which presented in section 2.7.2 Learning Ontologies from Software Artifacts and 2.7.3 Web Service Annotation Using Ontology Mapping. However these work all start from an automation perspective which generates semantic web services automatically. Use an automatic process to generate semantic web services at this stage is not a perfect solution, because multi-source artifacts can have completely different contents. Different naming and terminology cause low accuracy and efficiency of ontology mappings. Another more accurate and efficient approach which starts from engineering perspective was proposed by this dissertation. But the problem is that there is no such tool on the market using an engineering perspective which helps the software engineers retrieve the semantic information from a variety of formats and add it to OWL-S definitions.

Another issue is that after conversion from Java interfaces to WSDL definitions, there are no links between Java interfaces and WSDL definitions. It is not possible to find out which Java interface is corresponding to which service definition in WSDL without looking at the details of conversion. Therefore after conversion from WSDL definitions to OWL-S definitions, it will be very hard to maintain the links between Java interfaces and OWL-S definitions. If an ontology used by an OWL-S definition changes, it is not possible for this to be reflected in the Java interface and vice versa.

1.2 Research Objectives

In the previous section, the following issues were identified:

- Existing methods of conversion from WSDL definitions to OWL-S definitions do not take full advantage of semantic web service definitions. Missing semantic information needs to be added manually.
- No tools help software developers add extra semantics information to the OWL-S definitions.
- Maintenance will be hard to perform because there are no links between Java interfaces and OWL-S definitions

The objective of this research is to provide solutions to the above issues. This work aims to provide a tool to assist software engineers to write OWL-S versions of existing services which are based on Java, and to find a way to model, generate and keep the links between Java interfaces and OWL-S definitions so that definitions of certain components of OWL-S definition are traceable and maintainable.

1.3 Dissertation Roadmap

The whole dissertation is organized as follows:

Chapter 1 – Introduction describes the topic of this dissertation and the objectives of this research. The issues around semantic web service definition area are discussed.

Chapter 2 – State of the Art provides the background information around semantic web services area. Semantic web, semantic web service, technologies used by this project and some related work are studied and introduces.

Chapter 3 – Requirements and Analysis explains the functions should be provided by Semantics Editor based on the requirements and analysis.

Chapter 4 – Design shows the architecture design, data model design and detailed design of Semantics Editor including editor class design and user interface design.

Chapter 5 – Implementation describes the technologies used for implementation and explains the implementation details of important components of Semantics Editor.

Chapter 6 – Evaluation presents the usability evaluation on user interface design of Semantics Editor and the analysis on evaluation results.

Chapter 7 – Conclusions and Further Work presents the contributions of this research and further work.

Chapter 2 State of the Art

This chapter provides the background information around semantic web services area. Semantic web, semantic web service, technologies used by this project and some related work are studied and introduces.

2.1 Semantic Web

2.1.1 Introduction

For the decade years, web is composed content of human readable only texts. For example, search engines are only based on text matching. No context information is involved in searching so that a lot of useless information will also be processed and presented to the search engine users. Semantic Web is an extension to the existing World Wide Web that offers machine readable content. Thus it allows finding, sharing and integrating information more easily.

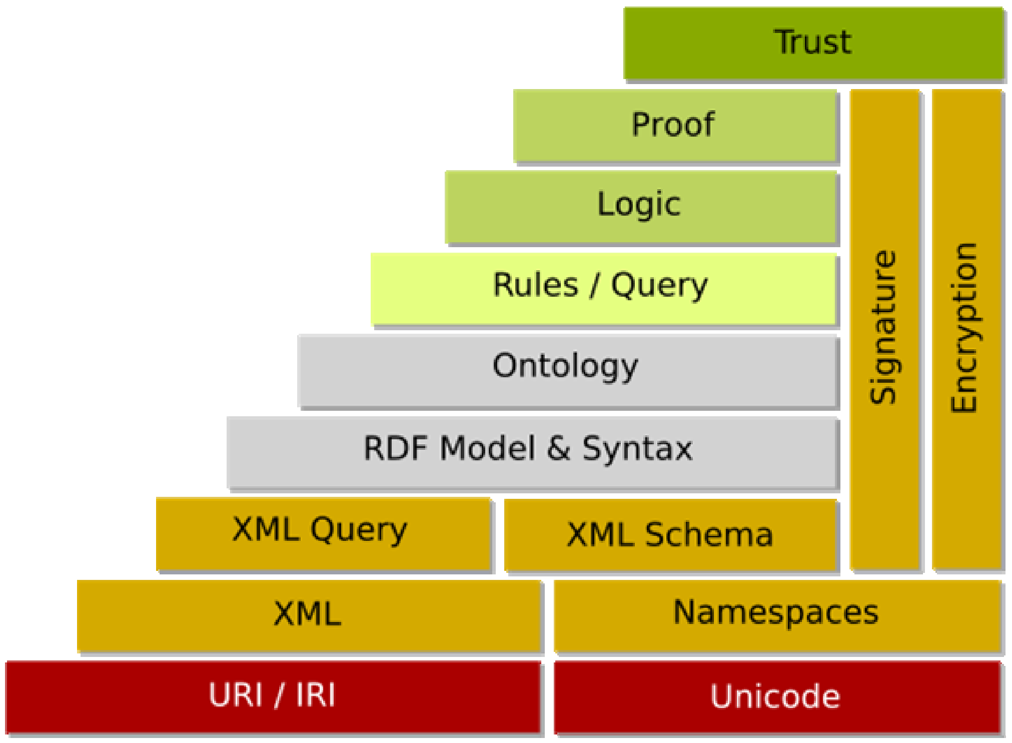


Figure 1 W3C Semantic Web Layer Cake [5]

The semantic web is based on the standards and tools of URI, XML, Namespaces, XML Schema, RDF, Ontology etc. The layer cake diagram Figure 1 published by W3C clearly shows an infrastructure that each layer is built on the lower layer.

2.1.2 Ontology

Ontology is a data model which represents a set of concepts within a domain and describes the relationships between those concepts. It is useful in the area of artificial intelligence, semantic web etc as it provides ability of reasoning about the objects.

A well-formed ontology is one that is expressed in a well-defined syntax that has a well-defined machine interpretation consistent with the above ontology definition.

Ontologies generally describe:

- Individuals: the basic or "ground level" objects (e.g. John, Mary)
- Classes: sets, collections, or types of objects (e.g. people)
- Attributes: properties, features, characteristics, or parameters that objects can have and share (e.g. John’s age is 20 years old)
- Relations: ways that objects can be related to one another (e.g. John is a people)

An example of a basic ontology is shown in Figure 2:

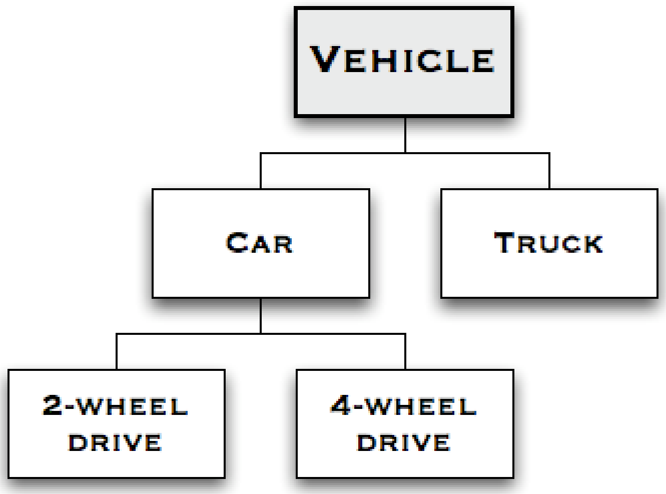


Figure 2 A Basic Ontology [6]

2.1.3 OWL

The web ontology language (OWL) [7] is a language for defining and instantiating ontologies. It is designed specifically for applications to process the ontologies. OWL offers a great machine interpretability of web content by providing additional vocabulary along with a formal semantics. OWL is based on XML, RDF and RDFS.

OWL has three species:

- OWL full is union of OWL syntax and RDF
- OWL DL restricted to FOL fragment
- OWL Lite is “easier to implement” subset of OWL DL

2.1.4 SWRL

Semantic Web Rule Language (SWRL) [8] is a proposal by W3C for additional sophisticated inferencing and reasoning. SWRL is based on OWL DL and OWL Lite with the Unary/Binary Datalog RuleML. It extends OWL axioms to include rules. SWRL is a human readable language which rules are of the form of an implication between an antecedent (body) and consequent (head). SWRL is used to express preconditions in OWL-S definitions.

2.2 Web Service

2.2.1 Traditional Web Service

W3C defined web service as a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols. [9]

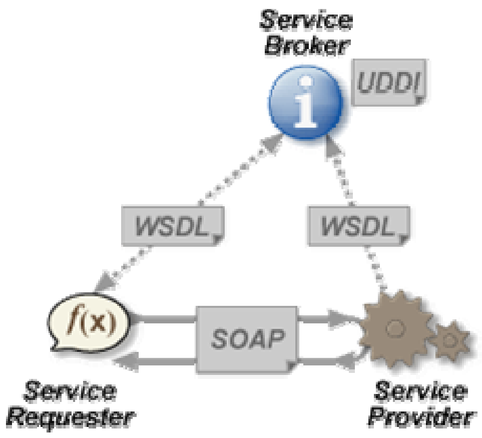


Figure 3 Web Service Architecture [10]

The core specifications are followings:

- **Web Service Definition language (WSDL):** W3C defined that WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services). WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate, however, the only bindings described in this document describe how to use WSDL in conjunction with SOAP 1.1, HTTP GET/POST, and MIME. [11]
- **Simple Object Access Protocol (SOAP):** W3C defined that SOAP is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses. SOAP can potentially be used in combination with a variety of other protocols; however, the only bindings defined in this document describe how to use SOAP in combination with HTTP and HTTP Extension Framework. [12]
- **Universal Description, Discovery and Integration (UDDI):** UDDI is a directory service where service provider and service requester can publish and find the web services. A UDDI registry service is a web service that maintains the information about service providers, service implementations and service metadata. [13]

2.2.2 Semantic Web Service

Semantic web service addressed two problems:

- In order to make service composition possible, developers have to reach some sort of agreement on the interaction of web services. This makes automatic service composition very difficult.
- On the other hand, WSDL can only describe operations and structure of the data.

There are no semantic meanings on data.

Semantic web service solves the problems by providing extra semantic meanings for web services.

2.2.2.1 OWL-S

Web Ontology Language for Services (OWL-S) [1] is an ontology of services that built on existing OWL framework to describe web services as semantic web services. It enables automatic web service discovery, invocation, composition and interoperation.

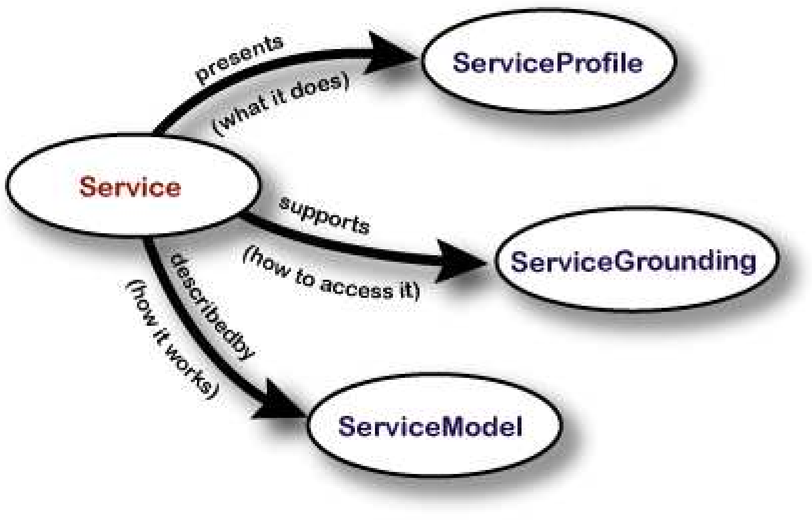


Figure 4 Top level of the service ontology [1]

An upper ontology is shown in Figure 4 to describe OWL-S elements:

- **Service Profile:** The service profile tells "what the service does". it tells the service requester if the service meets the requirements or not, the capability of the service and limitation on the service.
- **Service Grounding:** The service grounding tells “how to access it”. It specifies the details of service invocation. For example: communication protocols, message formats, port numbers etc.
- **Service Model:** The service model tells “how it works”. It tells the service requester how to use the services, preconditions before execution and results after execution.

2.2.2.2 WSDL-S

Web Service Semantics (WSDL-S) [14] is an annotation language for describing semantic web service. It extends WSDL by using extensibility elements of WSDL and adds semantic meaning to WSDL definitions by annotating elements in WSDL definitions.

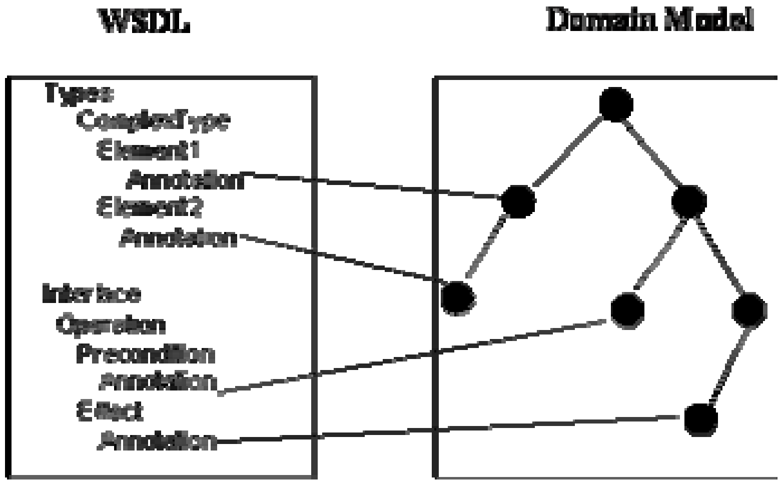


Figure 5 WSDL-S Annotation [14]

The Figure 5 above shows how WSDL-S adds the semantics to WSDL by referencing concepts in an outside domain model.

2.2.3 WSDL to OWL-S Conversion

One issue that OWL-S facing is that for the existing web services it will be a huge amount of work to rewrite the OWL-S version of service definition. So reuse existing WSDL framework to generate OWL-S definition is crucial. As OWL-S has a complementary relationship to WSDL, the conversion can be performed by following mappings (Figure 6): [1]

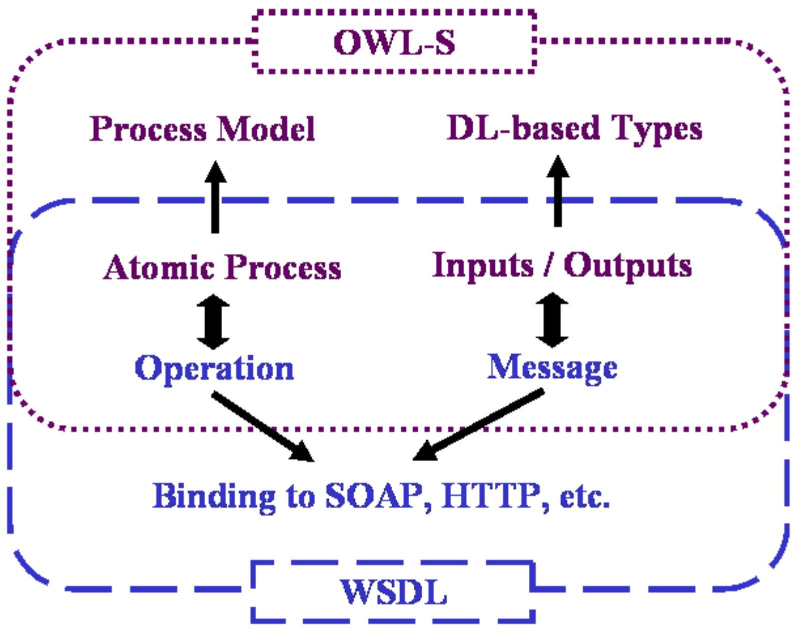
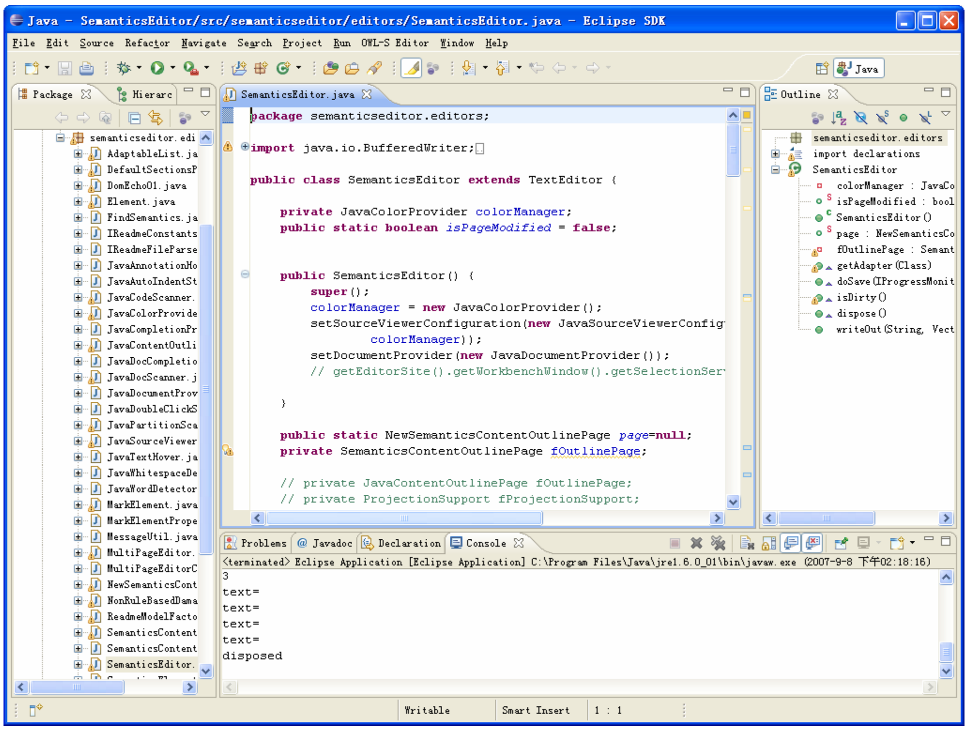


Figure 6 WSDL to OWL-S Grounding [1]

- An OWL-S atomic process corresponds to a WSDL operation.
- The set of inputs and the set of outputs of an OWL-S atomic process each correspond to WSDL's concept of message.
- The types (OWL classes) of the inputs and outputs of an OWL-S atomic process correspond to WSDL's extensible notion of abstract type (and, as such, may be used in WSDL specifications of message parts).

2.4 Eclipse

Eclipse [15] is an open source framework. Eclipse is famous for its origin form of Java IDE (Figure 7) and becomes the main stream of development IDE. Eclipse’s plug-in mechanism makes itself customizable and extensible. Thousands of plug-ins provides different functions which can be freely download from the web. The Eclipse for Rich Client Platform (RCP) and plug-in Developers provides a Plug-in Development Environment (PDE) [23] for developing Eclipse applications which makes developing an Eclipse plug-in really easy.


Java - SemanticsEditor/src/semanticseditor/editors/SemanticsEditor.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run OWL-S Editor Window Help
Package
Hierarc
SemanticsEditor.java
Outline
Problems
Javadoc
Declaration
Console
Writable
Smart Insert
1 : 1

```java
package semanticseditor.editors;

import java.io.BufferedWriter;

public class SemanticsEditor extends TextEditor {

    private JavaColorProvider colorManager;
    public static boolean isPageModified = false;


    public SemanticsEditor() {
        super();
        colorManager = new JavaColorProvider();
        setSourceViewerConfiguration(new JavaSourceViewerConfig
                colorManager));
        setDocumentProvider(new JavaDocumentProvider());
        // getEditorSite().getWorkbenchWindow().getSelectionSer

    }

    public static NewSemanticsContentOutlinePage page=null;
    private SemanticsContentOutlinePage fOutlinePage;

    // private JavaContentOutlinePage fOutlinePage;
    // private ProjectionSupport fProjectionSupport;
```

```
<terminated> Eclipse Application [Eclipse Application] C:\Program Files\Java\jre1.6.0_01\bin\javaw.exe (2007-9-8 下午02:18:16)
3
text=
text=
text=
text=
disposed
```

Figure 7 Eclipse IDE

2.5 Javadoc

Javadoc [16] is a tool that generates the documentation from Java code automatically. Instead of writing and maintaining a separate documentation, software engineers just need to write specially-formatted comments in the Java code. Javadoc will generate nice documentation automatically in HTML form.

Rather than generating the documentation in default format, Javadoc also provides doclets API to allow users create their own format. So Javadoc is a good tool to analyze the structure of Java code.

2.6 MDA

The Model Drive Architecture (MDA) [17] is a software design methodology developed by Object Management Group (OMG). MDA defined a set of guidelines to construct models. Platform-independent model (PIM) is used to describe the overall system specification. Then a platform definition model (PDM) is used to specify the underlying platform information. By applying a transformation approach, PIMs can be translated into platform-specific models (PSM) so that computers can run them.

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns. The core of MDA consists of following main basic concepts:

- **Model:** A model describes the specification of the system. It is often presented in the combination of texts and diagrams e.g. UML.
- **Model-Driven:** MDA aims to increase the power of model in software/system design. Model driven means all designs, constructions, deployments, operations, maintenances and modifications are around models.
- **Platform Independent Model (PIM):** PIM describes system from a platform independent viewpoint. A standard technology to get platform independency is to use the virtual machine technology to be built on top of the platform. A typical example is Java Virtual Machine (JVM) developed by Sun. JVM has different versions and is running on different operation systems. So software and systems that are developed by Java programming language achieved platform independent characteristic.
- **Platform Specific Model (PSM):** PSM describes system from a platform specific viewpoint. It is a specially designed model to be running on a particular platform.
- **Platform Model:** A platform model is a model which contains the specifications and characteristics of a platform e.g. CORBA Component Model.
- **Model Transformation:** Model transformation is the process to convert from one model to another model. In MDA, a PIM is converted to a PSM.

2.7 Related Work

A lot of work have been done around semantic web service area, all these work have their contributions and limitations.

2.7.1 CODE

CMU's OWL-S Development Environment [18] is an Eclipse plug-in which supports the whole OWL-S development processes from Java2OWL-S conversion, OWL-S definitions editing to the deployment and UDDI registration.

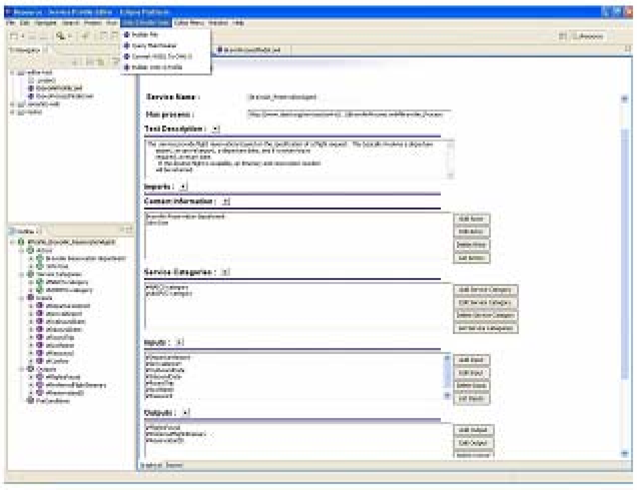


Figure 8 Frame Based Profile Editor

CODE provides an integrated development environment for software engineers to develop semantic web services. It provides frames based editors e.g. profile editor (Figure 8), process editor. It uses Apache's Java2WSDL [21] and CMU's WSDL2OWL-S [3] to achieve the generation of WSDL and OWL-S. CODE uses OWL-S2UDDI [22] to translate OWL-S profile to a UDDI compatible form. Thus it can be automatically registered with an UDDI server, such as the CMU's OWL-S/UDDI Matchmaker.

2.7.2 Learning Ontologies from Software Artifacts

Kalina Bontcheva and Marta Sabou's paper [19] presents an initial prototype of an ontology learning system which facilitates access, maintenance and reuse of software artifacts. The system is able to learn the ontologies from multiple information sources. Follow the steps of term extraction, term pruning, multi-source term enrichment and term matching, the system aims to reuse software artifacts and learn the ontologies automatically from multiple sources.

2.7.3 Web Service Annotation Using Ontology Mapping

Zhang Duo, Li JuanZi and Xu Bin [20] take the same idea as WSDL-S [14] which adds OWL ontologies to WSDL definitions by annotating web services. They provide a set of rules to translate from XML schema to ontologies and develop an algorithm for ontology mapping. Finally they generate a semantic description of web services with the mapping result. The main contribution of their work is it provides a set of rules to translate from XML schema to ontologies which can be implemented to conduct auto-generation of semantic web services.

Chapter 3 Requirements and Analysis

This chapter presents the overall requirements of the tool. Based on analysis, several use cases were introduced.

3.1 Requirements

For the existing tools and approaches to generate semantic web services discussed in chapter two, they do not have the solutions to the following problems:

- The accuracy and efficiency of automatic generating semantic web services is not high
- Maintainability of generated semantic web services is low
- There is no development environment for software engineers to develop semantic web services from an engineering point of view

The aim of this project is to develop a tool to help software engineers to write OWL-S version of existing web services which were based on Java starting from an engineering perspective. The tool is used by software engineers who develop web services. So based on purpose and users of the tool following development requirements were identified:

- The tool will be an Eclipse plug-in
- The tool should be able to show the roundtrip of identifying semantic information from Javadoc, converting Java interfaces to OWL-S definitions and finally adding missing semantic information into OWL-S definitions
- The tool should help the users retrieve semantic information from Javadoc
- The tool should provide a means of storing these semantic information
- The tool should help the users restore the missing semantic information to OWL-S definitions
- The tool should help the users generate preconditions in OWL-S definitions
- The tool should generate the links between Java interfaces and OWL-S definitions

- The tool should allow the users to trace between Java interfaces and OWL-S definitions
- The tool should provide a user interface to allow the users to perform above operations

3.2 Analysis

After a detailed study on requirements listed above, combining with the state of the art study, several use cases were defined to better understand the requirements and goal of the tool.

3.2.1 Identifying a Concept from Javadoc

In order to retrieve missing and add extra semantic information, semantic resources of related web services are crucial to the tool itself. This extra semantic information can be modeled in a variety of formats, e.g. UMLs, XMLs, Javadoc, software manuals, documents etc. In this project, Javadoc is taken as a sort of source of this purpose. Generally when a software engineer is developing a web service and writing the Java code, he will leave some explanations on the attributes, methods etc he identified. And also comments are required for the future maintenance and development. This information can be considered as semantic information for web services at a certain degree.

The idea of identifying a concept is that the tool should allow the users to reuse this information inside Java classes as a source for semantic web services. Users can identify concepts from Javadoc and even link them to other concepts in other existing ontologies.

3.2.2 Identifying Association Properties between Concepts

After identifying concepts, the next step is identifying association properties of that concept. The users should be able to identify the ontology that concept belongs to, the method that concept was bound to, input/output elements in OWL-S definitions that concept is referred to, preconditions and related predicates.

This is how users retrieve the missing and extra semantics from Javadoc. As Javadoc is well structured, it is better not to break the original architecture of Javadoc. So these identified association properties should be stored separately and can be reused by other tools for other purpose in the future.

3.2.3 Binding a Concept to OWL-S Input/Output

Simple conversion from WSDL definitions to OWL-S definitions do not generate semantics due to lack of semantic meanings of WSDL definitions. In order to take full advantage of OWL-S definitions, semantic information needs to be added manually. Though concepts are created as inputs and outputs in OWL-S definitions after conversion, however those concepts just have concept names but without any attributes, properties relationships etc. So binding concepts from other ontologies to OWL-S inputs and outputs can add semantic meanings to OWL-S definitions. Considering the consistency of OWL-S definitions, equivalent class relationships should be created for concepts which do not have semantic meanings.

3.2.4 Creating Unary and Binary Preconditions

Another goal of this project is to create preconditions in OWL-S definitions. As preconditions are not part of WSDL elements, preconditions need to be added manually to OWL-S definitions after conversion from WSDL definitions to OWL-S definitions. Based on current state of Semantic Web Rule Language (SWRL) [8] , the creation of unary and binary preconditions was selected to be a function that provided by the tool to assist users.

The users should be able to specify the class predicates and property predicates with the help of the tool and create the unary and binary preconditions for OWL-S definitions.

3.2.5 Tracing between Java Interfaces and OWL-S Definitions

Traceability is an important part in this project. The reason to provide traceability between Java interfaces and OWL-S definitions is that software engineers can be involved in developing semantic web services during the development phase. Providing this kind of function can help software engineers observe the changes of OWL-S definitions when Java interfaces are changed. On the other hand, if a link between Java interfaces and OWL-S definitions can be created, it will provide a direct view of the links between Java classes and OWL-S definitions. Maintenance will be easier to perform and development of semantic web services will become more efficient.

The tool should allow tracing between Java interfaces and OWL-S definitions in both directions. Users should be able to find the positions of terms that are identified in Javadoc. On the other hand, users should also be able to find corresponding input, output, and preconditions in OWL-S definitions.

Chapter 4 Design

This chapter begins with the architecture design, the data model design and the user interface design. Based on the analysis results from chapter three, a detailed design is presented including the use case diagram, the work flow design, the package diagram and several class diagrams. Finally, according to the user interface framework of Eclipse platform, a user interface is designed.

4.1 Architecture Design

As the tool is an Eclipse plug-in, so the architecture design of the tool should take Eclipse user interface framework into account. Advantages and limitations should both be considered. As an integrated development environment, Eclipse provides a great extensibility on editors. Thousands of editors for different purposes have been developed and are freely available on the web. Therefore an Eclipse plug-in called Semantics Editor was designed and split into following three parts:

- **Javadoc Editor:** The Javadoc editor is responsible for adding extra semantics by means of editing Javadoc in Java classes. Users are allowed to identify concepts from Javadoc and identify association properties between concepts.
- **Java interfaces to OWL-S definitions conversion:** After having identifying the concepts and association properties, users can use the tool to generate OWL-S definitions from Java interfaces.
- **OWL-S Editor:** The OWL-S editor is used to take extra semantic information identified by Javadoc editor and then add it to the OWL-S definitions generated after conversion.

Figure 9 below shows the architecture of Semantics Editor and how editors interact with extra semantic information:

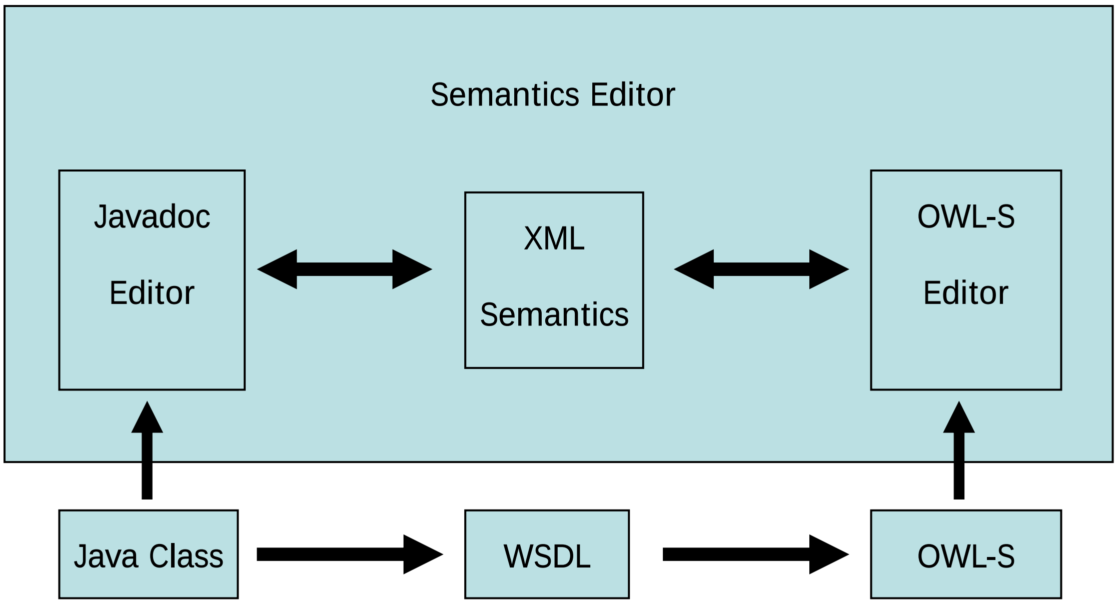


Figure 9 Architecture Design

The users first use Javadoc editor to identify concepts and association properties from Javadoc. This semantic information will be stored in a separate XML file. Then the users convert the Java classes to OWL-S definitions by using the conversion tool. Finally the users use OWL-S editor to add extra semantic information in OWL-S definitions by reading the semantic information from the XML file.

The advantages of this architecture design of Semantics Editor are the followings:

- Saving the extra semantic information identified from Javadoc to a separate XML file will keep the consistency of the original contents of Javadoc. The users can still read the comments and documentations in Java classes as before.
- The separate XML file creates the links between Java classes and OWL-S definitions as both editors use this XML file. The Javadoc editor saves identified semantic information to this file. The OWL-S editor reads this file to retrieve extra semantic information and then adds it to the OWL-S definitions. This makes tracing between Java classes and OWL-S definitions possible.
- As semantic information is stored independent from editors, this makes Semantic Editor portable and extensible. Semantic information stored in XML files can be reused by other applications and systems. The Semantics Editor can also be extended to support multiple resources. For example, by adding a UML editor, the Semantics Editor can support retrieve semantic information from UML diagrams.

4.2 Data Model Design

Design of this tool used the model concept from Model Driven Architecture (MDA) as described in section 2.6 MDA. A data model was designed to be the bridge between Javadoc editor and OWL-S editor. The Semantics Editor was designed to operate around the data model. All operations were designed to manipulate the data stored in this model.

The SemanticsElement class (Figure 10) is a bean class used to present the data model. It contains all fields that can be edit in Javadoc editor. Each field has get and set methods to read and modify the values.

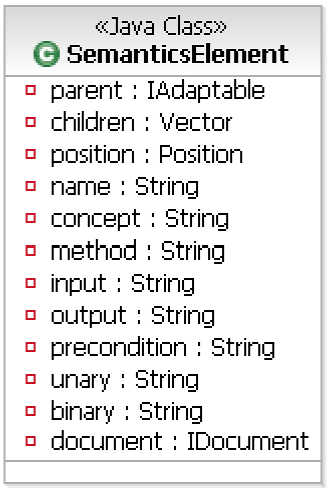


Figure 10 Semantics Element Data Model

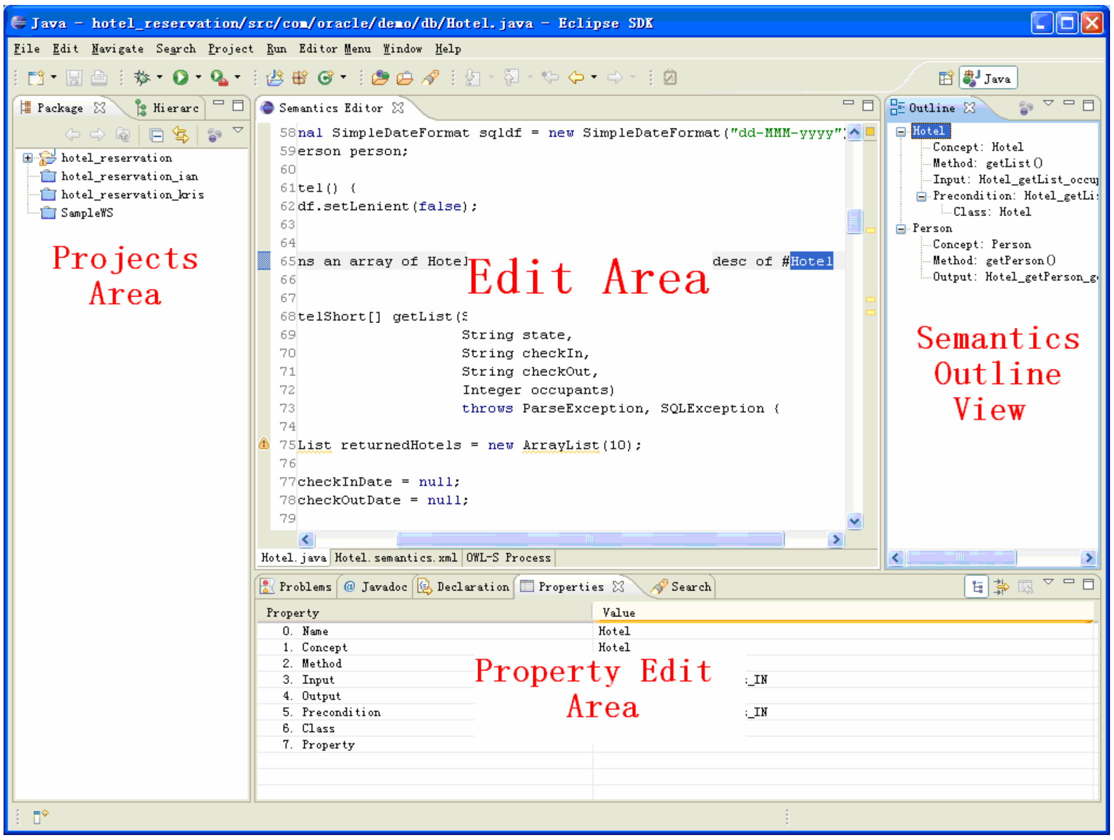
- **parent:** the parent node of current node in a tree structured view
- **children:** the children nodes of current node
- **position:** the corresponding position in Java class of this semantics element
- **name:** the term identified from Javadoc
- **concept:** the concept to which this term can be referred
- **method:** the method to which this semantics element is bound
- **input:** the input to which this element is bound (optional)
- **output:** the output to which this element is bound (optional)
- **precondition:** the precondition to which this element is bound (optional)

- **unary:** the class predicate of the precondition (optional)
- **binary:** the property predicate of the precondition (optional)

4.3 User Interface Design

The user interface design principles of Semantics Editor are the followings:

- **Consistency:** The user interface of Semantics Editor should look like other editors in Eclipse IDE. Users should not need to take a long time to get familiar with the user interface. A standard user interface of editors in Eclipse IDE should be used to keep the consistency.
- **Clearness:** The user interface should be able to speak out. When users look at user interface, it should be clearly shown to the users the functions of each part of the user interface.
- **Simplicity:** The user interface should be easy to use. Interactions between users and Semantics editor should be designed as simple as possible. Complex interactions should be avoided.



```
nal SimpleDateFormat sqldf = new SimpleDateFormat("dd-MMM-yyyy"
erson person;

tel() {
df.setLenient(false);


ns an array of Hotel                 desc of #Hotel


telShort[] getList(S
                    String state,
                    String checkIn,
                    String checkOut,
                    Integer occupants)
                    throws ParseException, SQLException {

List returnedHotels = new ArrayList(10);

checkInDate = null;
checkOutDate = null;

```

```
Hotel
    Concept: Hotel
    Method: getList()
    Input: Hotel_getList_occu
    Precondition: Hotel_getLi
        Class: Hotel
Person
    Concept: Person
    Method: getPerson()
    Output: Hotel_getPerson_g
```

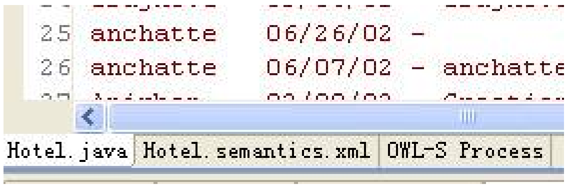
Property	Value
0. Name	Hotel
1. Concept	Hotel
2. Method	
3. Input	_IN
4. Output	
5. Precondition	_IN
6. Class	
7. Property	

Figure 11 User Interface Design

Based on three principles presented above and restrictions on Eclipse platform, the user interface was designed as shown in Figure 11.

The user interface is split into four parts:

- **Projects Area:** This is the area where the users create their projects.
- **Semantics Outline View Area:** Here lists semantics elements and association properties in a tree structured view. A tree structured view was used to visualize the content of the separate XML file. The tree view clearly shows the structure and content of the XML file. In addition, semantics content outline view is the visual presentation of the links between Javadoc editor and OWL-S editor.
- **Property Edit Area:** Here users can edit association properties of the term they identified in edit area. The advantage of using a property edit area is that the structure of Javadoc will be kept. Users can clearly distinguish between Javadoc and semantic information. Users are explicitly informed that they are editing the properties of terms in Javadoc by using this property edit area. Users can easily understand that these properties are semantic information which belongs to the term identified from Javadoc.
- **Main Edit Area:** Here is where the users edit Javadoc, the separate XML file and OWL-S process files. This area is designed based on a tabbed frame structure (Figure 12). By switching between different tabs, users can edit Java classes, the separate XML file where stores semantic information and OWL-S process files. The reason to use a tabbed frame structure is that switching between editors will not change the content of semantics outline view. Thus the user interface shows the meaning that the data in semantics content outline view is shared by all editors.



```
25 anchatte   06/26/02 -
26 anchatte   06/07/02 - anchatte
```

Figure 12 Tabbed Frame Structure

4.4 Detailed Design

4.4.1 Use Case Design

The use case diagram defines the functions of a basic system. It shows what users can do with the system. A user case diagram (Figure 13) shown below clearly describes what users can do with the tool.

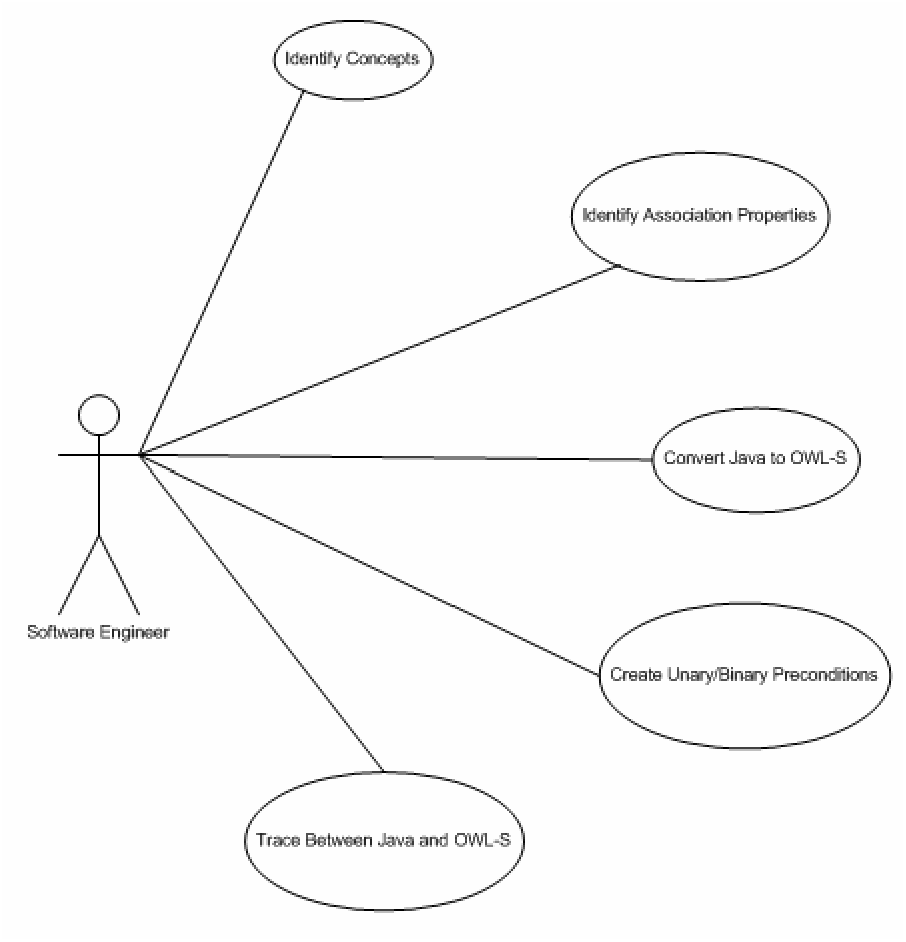

Identify Concepts
Identify Association Properties
Convert Java to OWL-S
Create Unary/Binary Preconditions
Trace Between Java and OWL-S
Software Engineer

Figure 13 Use Case Diagram

Users are allowed to:

- Identifying concepts from Javadoc
- Identifying association properties of those concepts
- Converting Java interfaces to OWL-S definitions
- Creating unary and binary preconditions
- Tracing between Java interfaces and OWL-S definitions

4.4.2 Work Flow Design

The work flow defines a series of sequential steps to reach a certain goal. Based on detailed analysis and five use cases identified in chapter three, a work flow is defined to achieve the goals of Semantics Editor which are retrieving semantic information from Javadoc, adding extra semantic information to OWL-S definitions and tracing between Java classes and OWL-S definitions:

Step 1: Open a Java class that contains web service interfaces using Semantic Editor.

Step 2: Find a term in Javadoc which can be considered as or referred to a concept and add a '#' symbol right before the term. Then this term will be listed in semantics outline view area.

Step 3: Click on the term and then the property edit area will be open for the users to edit properties. The properties that users can specify are concept, method, input, output, precondition, class predicate and property predicate. The use of these properties has been already discussed in section 4.2 Data Model Design.

Step 4: Identify these properties.

Step 5: Convert Java classes to OWL-S definitions.

Step 6: Switch the tab in edit area to the OWL-S editor and open OWL-S process file with the editor.

Step 7: Create unary or binary preconditions in OWL-S process file by drag and drop from semantics content outline view area to OWL-S process file edit area.

Step 8: Tracing between Java classes and OWL-S definitions can be simply performed by click on elements listed in semantics content outline view. The tool will highlight the related terms in Java classes and OWL-S definitions based on which tab is activated in main edit area.

4.4.3 Packages Design

A total of five packages (Figure 14) were designed for the implementation of Semantics Editor:

- **javadoc_editor:** activator class for Javadoc editor
- **javadoc_editor.editors:** contains classes for functions of Javadoc editor

- **owl_s_edior**: activator class for OWL-S editor
- **owl_s_edior.editors:** contains class for functions of OWL-S editor
- **owl_s_edior.actions:** contains context menu action classes for OWL-S editor

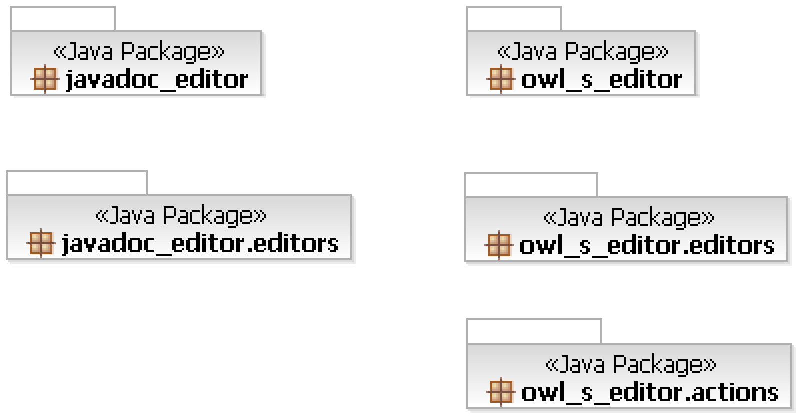


Figure 14 Packages

4.4.4 Editors Design

Considering the architecture design of Semantics Editor, users have to work with Javadoc editor and OWL-S editor together. A tabbed frame structure was designed for the user interface. In main edit area, users can switch between editors to edit different files. So a multiage container was designed to hold three editors: a Javadoc editor, a XML editor and an OWL-S editor.

The XML editor is not mentioned before because it is just an editor for XML files where extra semantic information is stored. It is an Eclipse plug-in built inside Eclipse IDE. It opens the separate XML file which contains extra semantic information. The users can view the information and edit data with this editor. It provides an alternative way of identifying association properties to the users. The users can modify the association properties by modifying the content of this file.

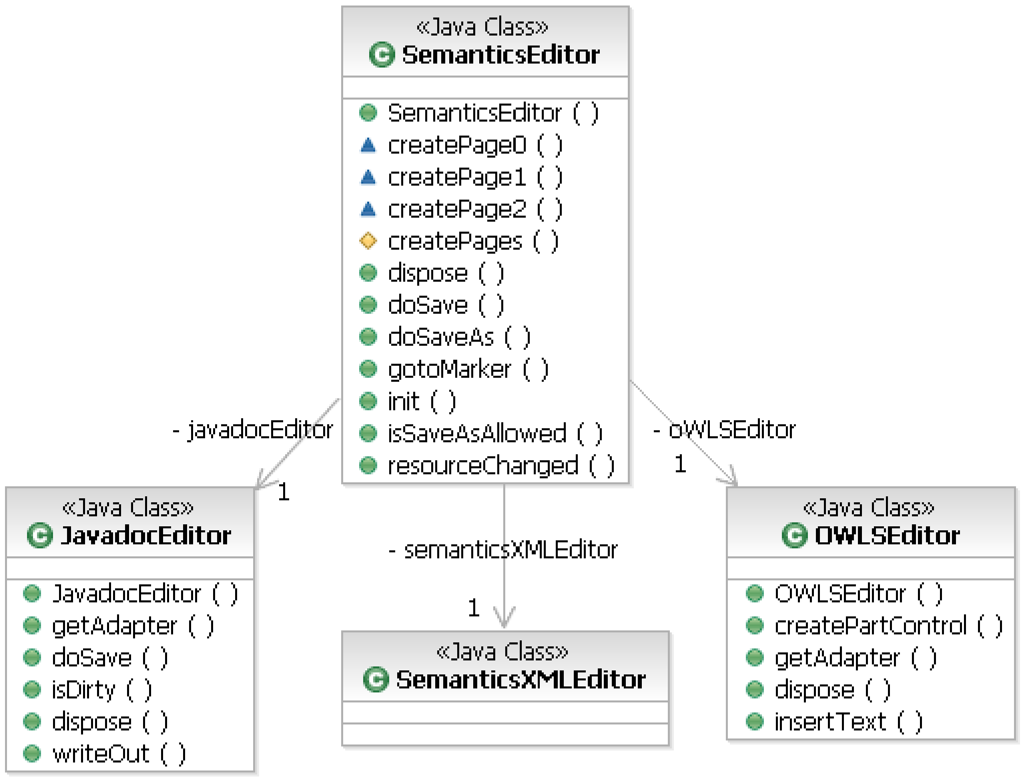


Figure 15 Editors' Relationship

The diagram (Figure 15) above shows the relationships between editors. SemanticsEditor class is the entry class of Semantics Editor. It is the container of other three editors. When the users open a Java class with Semantics Editor (Figure 16), this class will be called. The SemanticsEditor class creates an instance of JavadocEditor class, an instance of XML editor class and an instance of OWLSEditor class.

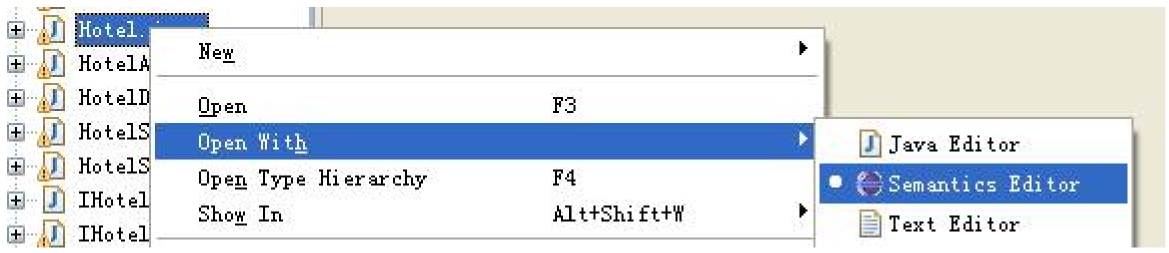


Figure 16 Open with Semantics Editor

The SemantcisEditor class also controls the state and lifecycle of Javadoc editor and OWL-S editor. When the users perform the save operation, this class will call all doSave() methods of Javadoc editor, XML editor and OWL-S editor. Thus all three files in different editors will be saved. When the users close the SemanticsEditor, all instances of three editors will be destroyed.

4.4.5 Javadoc Editor

The Javadoc editor is registered with '.java' file type in Eclipse platform. It is responsible for adding extra semantics by means of editing Javadoc in Java classes. In order to realize this purpose, the following functions were designed:

- The users can identify a concept from Javadoc by adding a '#' symbol right before a term which users think that can be referred to a concept. The Semantcis Editor should realize the changes to the Javadoc.
- The users can identify association properties of that concept by editing the property values in property view window.
- The identified concepts and association properties should be displayed in semantics content outline view window as a tree structured view.
- All values users identified in property edit window should be stored in SemanticsElement beans and saved to a separate XML file.
- Each time users change the concepts and association properties, the outline view window should be refreshed to correspond to the changes and the output XML file should be modified accordingly.
- When the users open a Java class that already contains some semantic information which was added by Semantics Editor before. The old semantic information should be recognized by the editor and displayed correctly in the outline view window.
- The users should be allowed to drag the concepts and association properties from the semantics content outline outline view window in order to perform the drag and drop between concepts and OWL-S definitions.

The class diagram (Figure 17) below shows the details of each class and relationships between them. A more detailed class diagram which has all fields and methods displayed can be found in Appendix B.

JavadocEditor class: The main class of Javadoc editor. It is responsible to control the state and lifecycle of Javadoc editor.

SemanticsContentOutlinePage class: This class is instantiated by JavadocEditor class. It will create outline view for semantic information stored in the separate XML file in the form of a tree structured view.

SemanticsModelFactory class: It is a factory class which used by the whole tool. It makes sure that the whole tool shares and uses same data model object and same semantic information that are saved in the XML file.

DefaultSectionsParser class: It is a parser for reading the content of Javadoc to get terms that identified by the users.

FindSemantics class: It will find existing semantic information in XML files that are identified and saved by the users before.

SemanticsElement class: It is a bean class which used by almost all other classes. It presents the data model designed in section 4.2 Data Model Design.

SemanticsElementProperties class: It is where the users edit association properties of concepts.

TextDragListener class: A test drag listener is registered to semantics content outline view to enable text dragging from semantics content outline view window.

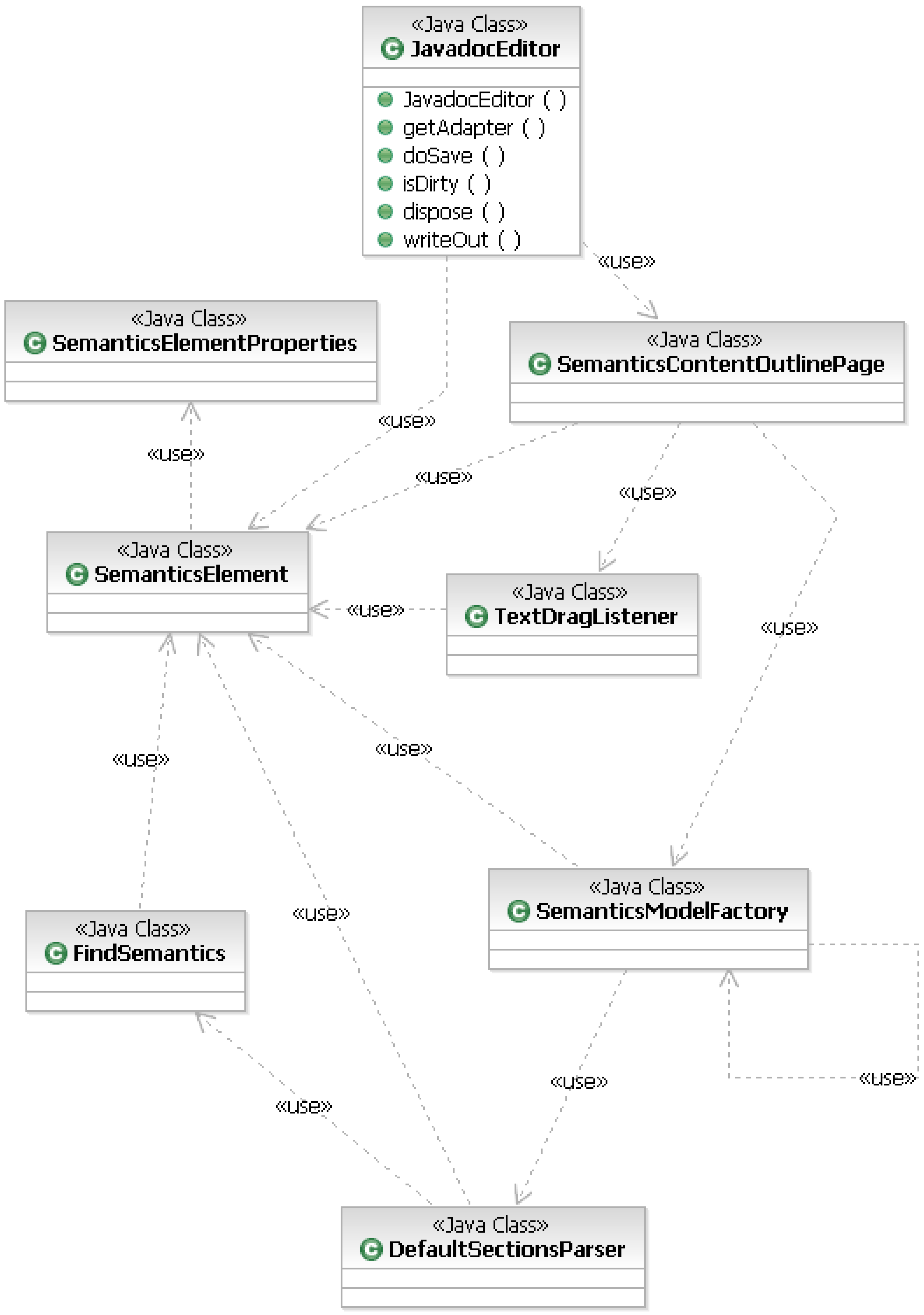


Figure 17 Javadoc Editor Class Diagram

4.4.6 OWL-S Editor

The OWL-S editor is registered with '.owl' file type in Eclipse platform. It is responsible for editing OWL-S process file to create unary and binary preconditions. Two context menu actions were designed to perform unary and binary preconditions generation operations. A MyDropTargetAdapter class was designed to accept drop operations for the OWL-S editor area. The class diagram (Figure 18) below shows the details of each class and relationships between them.

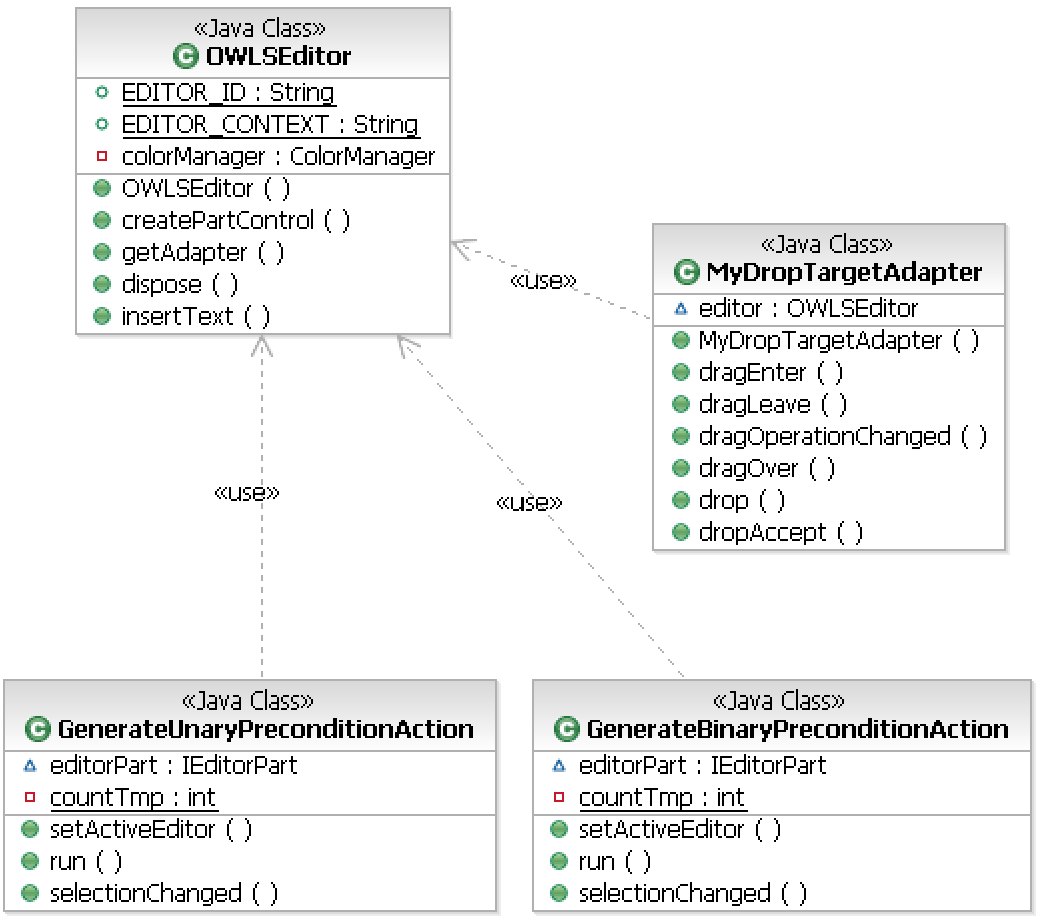


Figure 18 OWL-S Editor Class Diagram

Chapter 5 Implementation

The implementation of the Semantics Editor posed numerous challenges that needed to be overcome in order to satisfy the project objectives set out in chapter one. This chapter describes technologies used for the implementation and the problems overcome in the course of its construction.

5.1 Technologies Used

The Semantics Editor is an Eclipse plug-in which consists of three editors, an outline view and a property view. The Plug-in Development Environment (PDE) as mentioned in section 2.4 Eclipse was used to assist developing the Eclipse plug-in on Eclipse IDE platform. Java SE Development Kit (JDK) version 6 [24] was used to provide development APIs and Java Runtime Environment (JRE) version 6 [24] was used to provide runtime library. The plug-in was built on Eclipse SDK version 3.3 [25] .

Following APIs from Eclipse SDK were used to develop user interface of Semantics Editor:

- org.eclipse.ui
- org.eclipse.ui.views
- org.eclipse.ui.ide

Following APIs from Eclipse SDK were used to develop user interface of editors and manipulate the texts in Semantics Editor:

- org.eclipse.ui.workbench.texteditor
- org.eclipse.ui.editors
- org.eclipse.jface.text
- org.eclipse.text

The following APIs provide resources of Eclipse workspace e.g. documents and supports for runtime platform, core utility methods and the extension registry.

- org.eclipse.core.resources
- org.eclipse.core.runtime

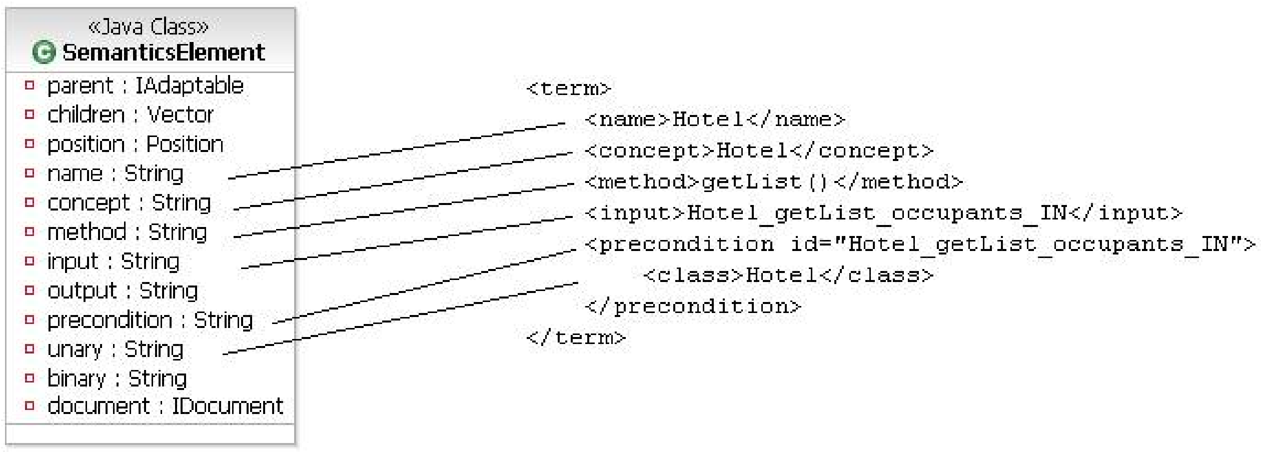
In addition the XML Document Object Model (XML DOM) [29] was used to access and manipulate XML files where Semantics Editor stores extra semantic information. DOM converts XML files to objects so that random access to the data in any level of XML tags becomes possible. Sun's implementation of DOM [30] was used in the project to read in and write out the semantic information stored in the separate XML file.

5.2 Data Model Implementation

The data model is a key to Semantic Editor. All operations are designed to use this data model to perform reading and writing semantic information from the separate XML file. The data model designed in section 4.2 Data Model Design is implemented to the followings:

- One object of SemanticsElement class corresponds to an entry of data in the XML file.
- All data within a term tag in the XML file is defined as one entry.

A data model to XML mapping is clearly shown in Figure 19. All data within the term tag is called one entry and it corresponds to an object of SemanticsElement bean class.



```
<term>
    <name>Hotel</name>
    <concept>Hotel</concept>
    <method>getList()</method>
    <input>Hotel_getList_occupants_IN</input>
    <precondition id="Hotel_getList_occupants_IN">
        <class>Hotel</class>
    </precondition>
</term>
```

Figure 19 Data Model to XML Mapping

5.3 Javadoc Editor

The Javadoc editor is used to edit Javadoc inside Java classes to identify concepts and association properties. The concepts and properties that identified are listed as a tree structured view in semantics content outline view window. Association properties can be displayed and edit by click on the nodes of the tree. A drag operation is allowed to drag elements from the tree.

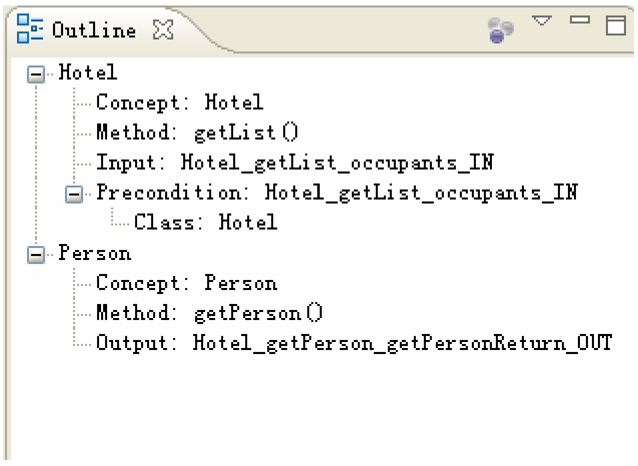
5.3.1 Section Parser

A section parser is implemented to read the content of Javadoc in Java classes to find out identified concepts. The section parser will find all terms that have '#' symbols right before them by searching the whole document. Once a matched term is found, the section parser will create an object of SemanticsElement class with the name field filled as that term. Then it will check if this term has relevant semantic information identified and saved before in the XML file where all semantic information is stored. If it is, relevant fields of SemanticsElement object will be filled. All objects created by the section parser are stored in a vector to be shared by the whole Semantics Editor plug-in.

5.3.2 Semantics Content Outline View

5.3.2.1 Tree Structured View

The semantics content outline view visualizes the semantic information stored in the XML file as a tree structured view. Figure 20 shows an example of semantics content outline view. A tree can show the structure and content of the XML file clearly. For example the term 'Hotel' has the attributes of 'Concept', 'Method', 'Input' and 'Precondition'. In the example of this diagram the precondition is a unary precondition which has only a class predicate 'Hotel'. So only the class predicate 'Hotel' node is listed as a sub-node of 'Precondition' node



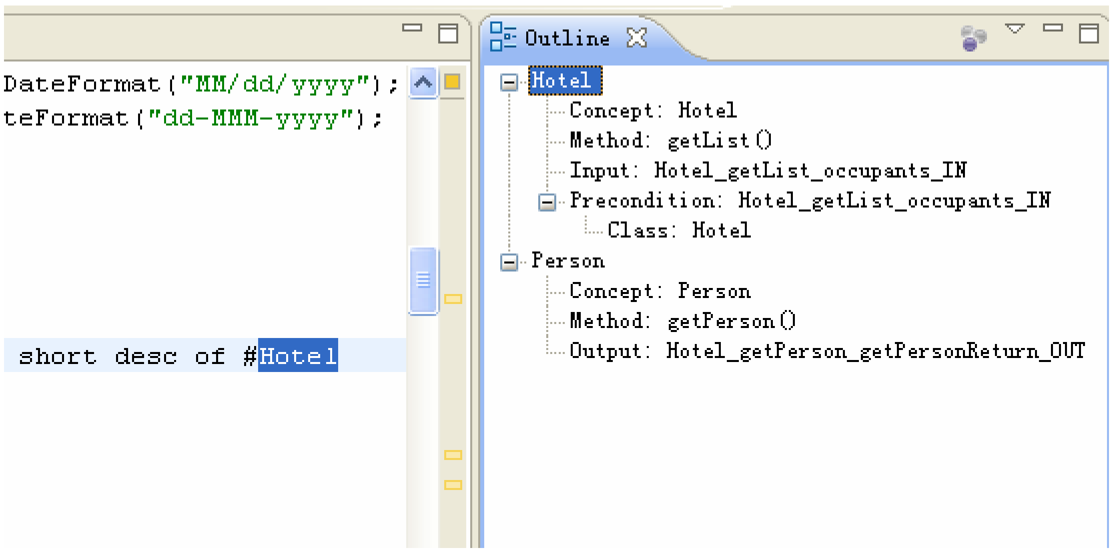
```
Hotel
  Concept: Hotel
  Method: getList()
  Input: Hotel_getList_occupants_IN
  Precondition: Hotel_getList_occupants_IN
    Class: Hotel
Person
  Concept: Person
  Method: getPerson()
  Output: Hotel_getPerson_getPersonReturn_OUT
```

Figure 20 Semantics Content Outline View

Converting from the XML file to a tree is done by using Sun's DOM API [30]. As DOM converts XML files to objects, the data of any level of XML tags can be accessed by invoking getElementByTag() method of objects. As shown in Figure 10, the SemanticsElement has two fields: parent and child. By declaring one SemanticsElement object is the parent or child of another SemanticsElement object, father nodes and child nodes can be easily created for the tree structured view.

5.3.2.2 Traceability

A selection listener is also registered to semantics content outline view to provide traceability. When the user click on an item listed in the semantics content outline view, the selection listener will get the object of the item selected and get data of position field of that object. Then the position of corresponding term in the edit area will be known and that term will be highlighted (Figure 21) by the listener.



```
DateFormat("MM/dd/yyyy");
teFormat("dd-MMM-yyyy");

short desc of #Hotel
```

```
Hotel
  Concept: Hotel
  Method: getList()
  Input: Hotel_getList_occupants_IN
  Precondition: Hotel_getList_occupants_IN
    Class: Hotel
Person
  Concept: Person
  Method: getPerson()
  Output: Hotel_getPerson_getPersonReturn_OUT
```

Figure 21 Traceability

Switching between editors by clicking on different editor tabs in edit area will not cause the changes of semantics content outline view. Thus tracing operations can be performed both in Javadoc editor and OWL-S editor. The semantics content outline view becomes the links between two editors. This is how user interface of Semantics Editor provides the traceability between Java classes and OWL-S definitions.

5.3.2.3 Drag Operation

A drag listener is also registered to semantics content outline view to allow dragging tree nodes from the tree. To the users everything listed in the tree structured view looks like just texts. In fact these are objects of SemanticsElement class. Therefore the drag listener has to convert the objects to plain text data so that it can be accepted by the OWL-S editor. For example, the class predicate field ‘Class:Hotel’ of object ‘Hotel’ has to be converted to text ‘Hotel’.

5.3.3 Semantic Information Reading and Writing

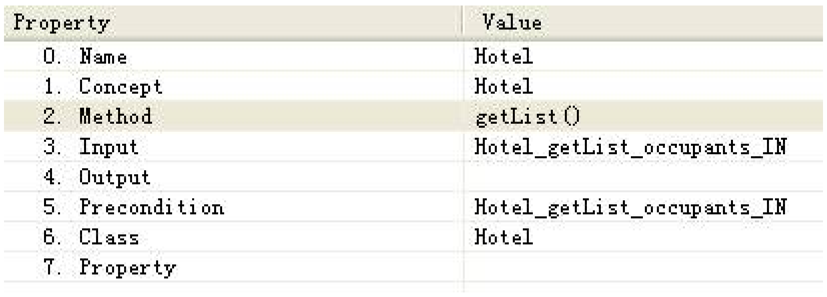
DOM as described in section 5.1 Technologies Used was used to read and write semantic information. Two methods were implemented to realize reading and writing functions respectively.

FindSemantics method is used to read semantic information which is already identified by the users in previous operations. For example, users identify one concept and association properties first. Then they close the project for a break. When they open the project again, previous semantic information should be loaded into semantics content outline view of Semantics Editor. The FindSemantcs method works with the section parser. When the parser finds a term with '#' symbol, it will first use the FindSemantics method to check if this term is already identified before. If it is, the FindSemantics method will retrieve related semantic information for the parser and the parser will create an object SemanticsElement class with relevant fields filled.

WriteOut method is used to write out semantic information to a separate XML file. As mentioned in section 5.3.1 Section Parser that all objects of SemanticsElement class are stored in a vector, this method will take all data in that vector and write out to a XML file according to the XML file structure designed in section 5.2 Data Model Implementation. The WriteOut method will be called when users perform save operation of Semantics Editor. Once the output XML file is modified, the content of semantics content outline view will be refreshed by Semantics Editor.

5.3.4 Properties Editing

As Javadoc is well structured, it is better not to break existing Javadoc structure. So a separate edit window for properties editing was designed in section 4.3 User Interface Design.



Property	Value
0. Name	Hotel
1. Concept	Hotel
2. Method	getList()
3. Input	Hotel_getList_occupants_IN
4. Output	
5. Precondition	Hotel_getList_occupants_IN
6. Class	Hotel
7. Property	

Figure 22 Properties Editing

Figure 22 shows how users can identify association properties for the concepts they identified. Property column shows the property labels. Value column is editable for the users to type in text based value.

A SemanticsElementProperties class is implemented to provide above user interface for the users to edit properties. When the users click on an item listed in semantics content outline view, an object of SemanticsElementProperties class will be created and a property edit area will be open to allow the users to edit the properties of that item. After finish editing the properties in properties edit area, when the users perform the save operation, these properties will be shown as sub-nodes of the term in semantics content outline view as shown in Figure 20. Simultaneously these data will be saved to the separate XML file.

5.4 OWL-S Editor

The OWL-S editor is used to edit OWL-S process file to create unary and binary preconditions. A context menu with two options was registered with the OWL-S editor to generate the templates of unary and binary preconditions. By identifying class and property predicates in properties edit area, the details of preconditions can be completed by drag and drop operations.

5.4.1 Open OWL-S Process File

The OWL-S process [1] file is main file of OWL-S definitions. It is a specification of the way a client may interact with a service. It contains the information of inputs, outputs, atomic processes, preconditions etc. The OWL-S editor modifies this process file to add extra semantic information to OWL-S definitions.

When the first time users use the OWL-S editor, it will first ask users to locate OWL-S process file that was generated by JAVA2OWLS converter as described in section 5.5 JAVA2OWLS Conversion. Figure 23 shows the user interface implementation of the dialog box used to open an OWL-S process file.

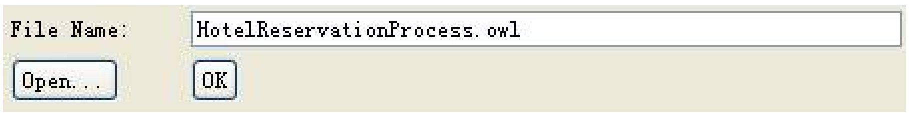
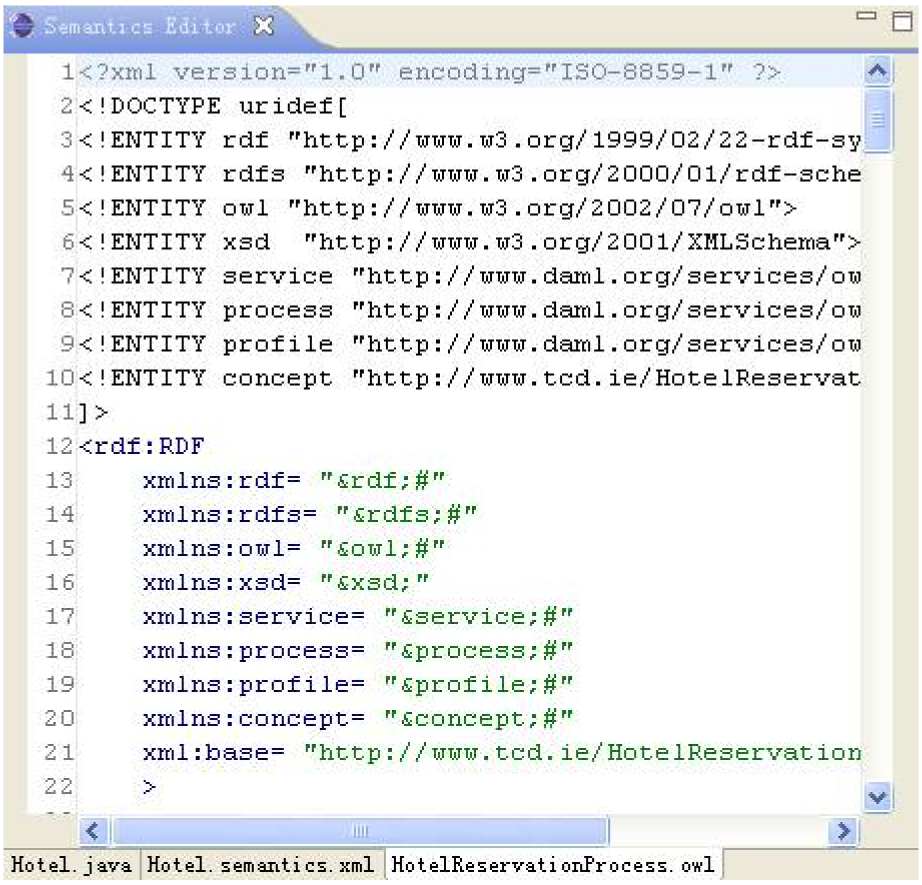

File Name: HotelReservationProcess.owl

Figure 23 Open OWL-S Process File Dialog

After the users specify the location of OWL-S process file and click on the OK button, the OWL-S process file will be open. Then users can start to add semantic information to the OWL-S definitions by modifying the OWL-S process file (Figure 24).



```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE uridef[
<!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-sy
<!ENTITY rdfs "http://www.w3.org/2000/01/rdf-sche
<!ENTITY owl "http://www.w3.org/2002/07/owl">
<!ENTITY xsd  "http://www.w3.org/2001/XMLSchema">
<!ENTITY service "http://www.daml.org/services/ow
<!ENTITY process "http://www.daml.org/services/ow
<!ENTITY profile "http://www.daml.org/services/ow
<!ENTITY concept "http://www.tcd.ie/HotelReservat
]>
<rdf:RDF
    xmlns:rdf= "&rdf;#"
    xmlns:rdfs= "&rdfs;#"
    xmlns:owl= "&owl;#"
    xmlns:xsd= "&xsd;"
    xmlns:service= "&service;#"
    xmlns:process= "&process;#"
    xmlns:profile= "&profile;#"
    xmlns:concept= "&concept;#"
    xml:base= "http://www.tcd.ie/HotelReservation
    >
```

Figure 24 Editing OWL-S Process File

5.4.2 Context Menu Registration

Two kinds of preconditions can be created by Semantics Editor: unary preconditions and binary preconditions. In order to make the user interface as easier as possible, a means of using a context menu was implemented. Two actions for the context menu were registered as an extension to popup menus in the configuration of Semantics Editor plug-in. When the users want to create preconditions, they just need to right click on the OWL-S editor area. Two extra options for precondition generation will be popped up. The users can select generating skeletons of unary or binary preconditions as shown in Figure 25 on their choice. After the users click on either ‘Generate Binary Precondition Skeleton’ or ‘Generate Unary Precondition Skeleton’, the skeleton will be generated and relevant texts will be inserted to the OWL-S process file at the position of cursor in edit area.

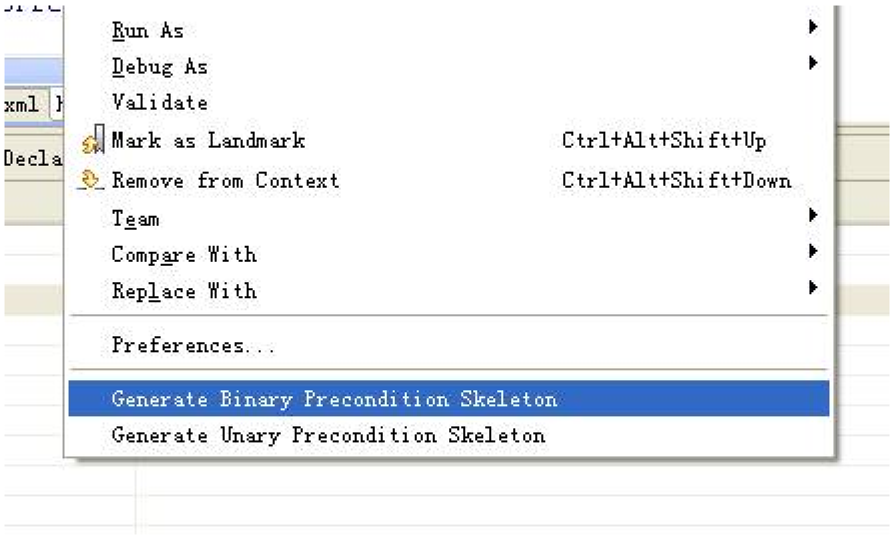

Run As
Debug As
Validate
Mark as Landmark Ctrl+Alt+Shift+Up
Remove from Context Ctrl+Alt+Shift+Down
Team
Compare With
Replace With
Preferences...
Generate Binary Precondition Skeleton
Generate Unary Precondition Skeleton

Figure 25 Preconditions Context Menu

5.4.3 Unary/Binary Preconditions Creation

Above operation only generates the skeletons of unary and binary preconditions. The complete preconditions can be generated by dragging the related items from semantics content outline view and dropping to the skeletons of preconditions. In this project, the users need to drag the class predicates for unary preconditions and both the class and property predicates for binary preconditions. These predicates can be identified in properties edit area as described in section 5.3.4 Properties Editing.

5.4.4 Drop Target Adapter

In order to realize drag and drop function for preconditions generation, the OWL-S editor should accept any text droppings. As objects have been already converted to text data as described in section 5.3.2.3 Drag Operation, what OWL-S editor needs to do is listening to any text drop operations. A drop target adapter class was implemented to let the OWL-S editor listen to the text drop operations on OWL-S editor. When the users drag an item from semantics content outline view and drop to the OWL-S editor. This drop target adapter is activated. It will find out the current position of user’s cursor and insert the text into the OWL-S process file.

5.5 JAVA2OWLS Conversion

As introduced in section 2.7.1 CODE, CODE uses Apache’s Java2WSDL tool and CMU’s WSDL2OWL-S tool to achieve the generation of WSDL definitions and

OWL-S definitions. This project reused this part of implementation from CODE.

A JAVA2OWLS converter was integrated with Semantics Editor. A menu was registered to Eclipse IDE. When the users open the Java classes with Semantics Editor, a menu as shown in Figure 26 can be found. By click on the menu option, the users can activate JAVA2OWLS converter tool.

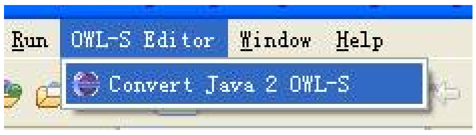
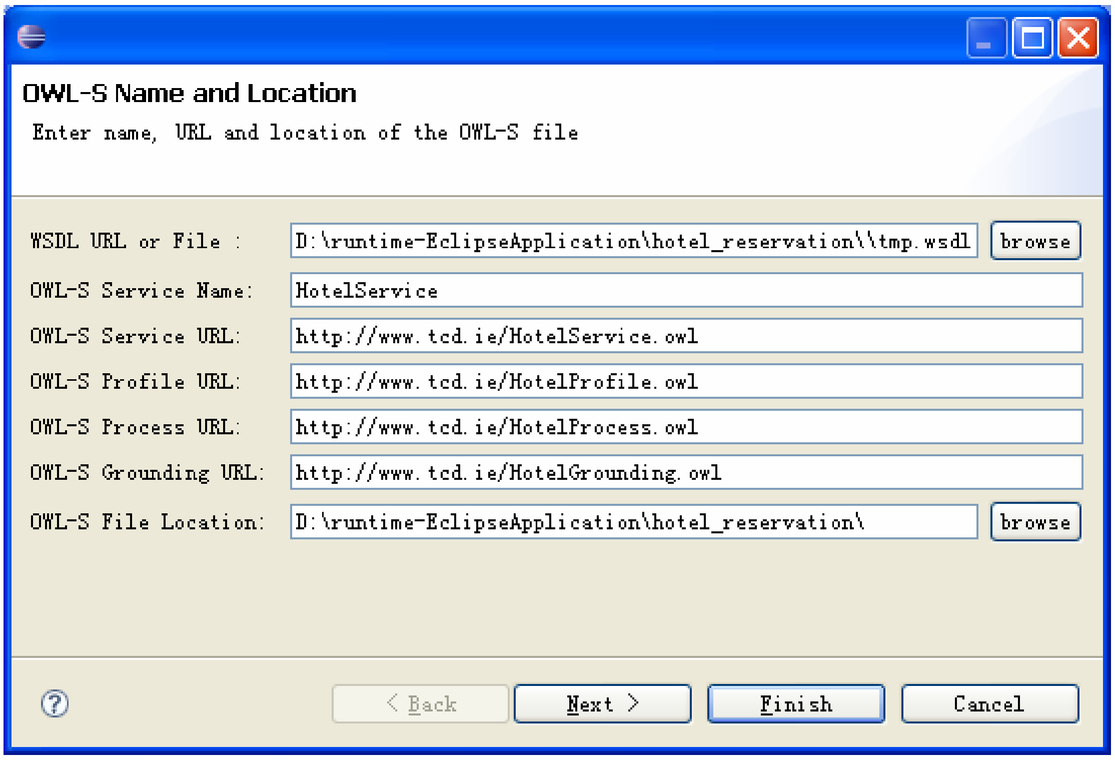


Figure 26 Java to OWL-S Converter Menu

The JAVA2OWLS converter is pretty simple to use. It will generate WSDL definitions first and then convert WSDL definitions to OWL-S definitions. Four OWL-S files will be generated (Figure 27). They are service, profile, process and grounding files. In this project, the OWL-S process is used by OWL-S editor to create unary and binary preconditions as it is the main file of OWL-S definitions.



OWL-S Name and Location

Enter name, URL and location of the OWL-S file

WSDL URL or File :	D:\runtime-EclipseApplication\hotel_reservation\\tmp.wsdl
OWL-S Service Name:	HotelService
OWL-S Service URL:	http://www.tcd.ie/HotelService.owl
OWL-S Profile URL:	http://www.tcd.ie/HotelProfile.owl
OWL-S Process URL:	http://www.tcd.ie/HotelProcess.owl
OWL-S Grounding URL:	http://www.tcd.ie/HotelGrounding.owl
OWL-S File Location:	D:\runtime-EclipseApplication\hotel_reservation\

Figure 27 JAVA2OWLS Converter

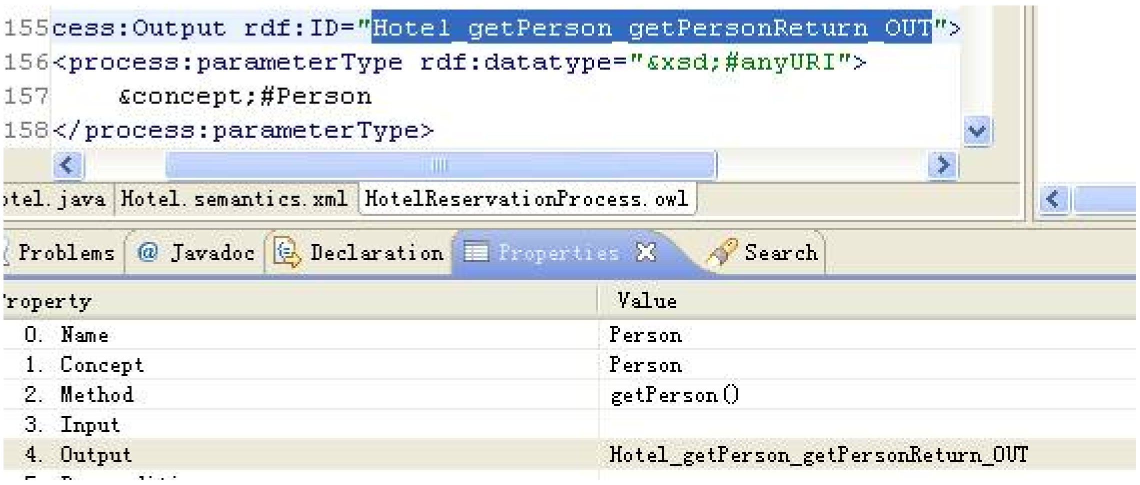
5.6 Co-operations between Editors

The Semantics Editor consists of a Javadoc editor, a XML editor, an OWL-S editor, a semantic outline view window and a property edit window. As a whole, editors have to co-operate with each other to realize the traceability and drag and drop functions.

5.6.1 Tracing

By tracing the users can find out elements in OWL-S definitions which are bound to the relevant terms identified in Javadoc. Section 5.3.2.2 Traceability only shows how to perform tracing operations between Java classes and OWL-S definitions. Before perform tracing, the links between Java classes and OWL-S definitions have to be created first.

Creation of links is done by identifying association properties of concepts. Users have to get the ID of the element in OWL-S where they want to create the link. Figure 28 shows an example of creating the link for an output process in OWL-S definitions. The output process has already been bound to the 'Person' concept in previous operations. So the link between Java classes and OWL-S definitions has to be created. The output process has an ID of 'Hotel_getPerson_getPersonReturn_OUT'. So what users need to do to create the link is filling the 'output' property of that concept with the value of this output process ID. Then the link between Java classes and OWL-S definitions is created. Same operations can be applied to create the links for inputs and preconditions. Filling the relevant properties the links will be created.



```
cess:Output rdf:ID="Hotel_getPerson_getPersonReturn_OUT">
<process:parameterType rdf:datatype="&xsd;#anyURI">
    &concept;#Person
</process:parameterType>
```

roperty	Value
0. Name	Person
1. Concept	Person
2. Method	getPerson()
3. Input	
4. Output	Hotel_getPerson_getPersonReturn_OUT

Figure 28 Create the Link

A listener is registered to semantics content outline view. As mentioned in section 4.3 User Interface Design, switching between tabs in edit area will not change the content of semantics content outline view area. The semantics content outline view is the visual presentation of the links between Java editor and OWL-S editor. So the listener can listen to selection operations happen in all editors in the edit area. The algorithm of tracing operation is simple. Once a selection operation happens in the editor area, the listener will capture the text content of the selection. It will look up the semantic information stored in the XML file. If a matched text is found in the XML file, the listener will find corresponding tree node in semantics content outline view and highlight it. As IDs of processes are used to create the links, duplications can be avoided. All editors can perform same operations. Thus the links between Java classes and OWL-S definitions are created and tracing can be performed.

5.6.2 Drag and Drop

Drag and drop operation is a fast and easy to learn technique for users to perform tasks. As mentioned in section 5.3.2.3 Drag Operation and 5.4.4 Drop Target Adapter, the drag listener and drop adapter have been already registered to semantics content outline view and OWL-S editor. The drag and drop operation can and only can be performed between these two windows. Mis-operations can be avoided.

Chapter 6 Evaluation

This chapter introduced the usability evaluation conducted to test the usability of Semantics Editor. An analysis of user interface is presented based on the statistic of data collected from post-test questionnaire. Finally some comments left by the users are described and discussed.

6.1 Overview

Evaluation is a way of determining if a software product satisfies its requirements. Evaluation can assess extent of system functionality, assess effect of interface on user and identify specific problems. Usability test is a sort of tests for measuring effectiveness, efficiency and satisfaction with which users accomplish tasks. Think aloud [31] is a means of gathering data in usability tests. Users are asked to describe what they are doing and why, what they think is happening etc while they are performing tasks. One of the advantages is that it requires little expertise for testers. Think aloud techniques can also provide a useful insight and can show how system is actually used.

The main problems Semantics Editor addressed are retrieving and adding semantic information from Javadoc to OWL-S definitions using an engineering point of view and generating and visualizing the links between Java classes and OWL-S definitions. So usability of the user interface of Semantics Editor is considered to be a main evaluation aspect. In order to evaluate whether the Semantics Editor is efficient, effective and satisfying for those who use it, a usability test was designed. The test followed a standard usability test methodology. A collection of forms, checklist and other useful documents was used for conducting usability tests and user interview. Details can be found in Appendix A Usability Evaluation Documents. Think aloud technique was applied to the whole test to gather the data. All conversations were recorded for further analysis.

A total of five people attended the usability test. They are two Ph.D students from KDEG group and three master students from Ubicom class. The tests were undertaken in a quite room in Lloyd building in case noise distracted users and caused inveracity in the evaluation results.

6.2 Evaluation Approach

Users were provided a test document for them to read during the test. It contains a pre-test questionnaire, an introduction to Semantics Editor, the scenario users were set, three short tasks and a post-test questionnaire.

6.2.1 Pre-Test Questionnaire

A pre-test questionnaire was designed and used to gather users' background information. They were asked to explain their experiences using Eclipse IDE and OWL-S definitions. This information was used to categorize user level in the further analysis.

6.2.2 An Introduction to Semantics Editor

A brief written description about functions of Semantics Editor was presented to users to let them get a brief idea on the tool itself as well as what they are going to do. Users were asked to understand everything in the descriptions before they went to next steps. The test will not continue if users have any questions.

6.2.3 The Scenario Setting

The users were supposed to be software engineers who work for a web service provider. In order to provide better services to get more customers, they were required to develop semantic web services based on existing web services. The architect found the Semantics Editor is a very good tool to generate semantic web service definitions. Semantics Editor was recommended to the software engineers to use.

6.2.4 Three Short Tasks

All use cases defined in Chapter three were split into a total of three short tasks for the users to complete. For each task, there is an instruction to tell the user how to finish it. The tasks were designed to allow users to have a full experience on all functions of Semantics Editor. After completing these three short tasks, the users will go through all functions of the tool.

When the users were using the tool to complete the tasks, the users were asked to describe what they are doing and why, what they think is happening etc. The conversations were recorded whilst note taking was also used.

6.2.5 Post-Test questionnaire

A post-test questionnaire was designed and used to get the feedback after the test. Users were asked to mark the tool for overall performance and for each use case. Marks were used for further analysis. They were also asked to leave some comments for each step the way user interface was designed. Also users can leave their suggestions on which way the user interface can be improved.

6.3 Evaluation Results

6.3.1 Background Information Statistic

From the pre-test questionnaires, users' background information is collected. Background information has a huge influence on the accuracy of the evaluation results. As Semantics Editor is an Eclipse plug-in and specially designed for software engineers and generating semantic web services. The data for users' experience on both Eclipse IDE and OWL-S definitions is collected and shown in Figure 29 and Figure 30.

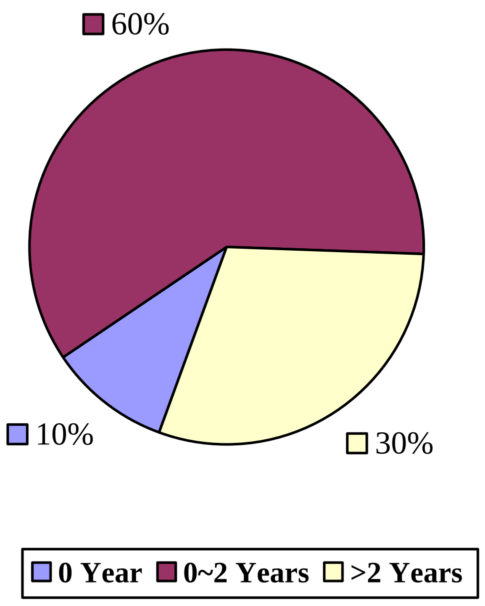


Figure 29 Users' Experience on Eclipse IDE

Figure 29 shows that 60 percent of the users have less than two years experience on

Eclipse IDE and 30 percent of the users even have more than two years experience. Only 10 percent of the users did not use the Eclipse IDE as their development environment before. The more experience on Eclipse IDE the less time will be spent getting familiar with the user interface of Semantics Editor. Then the users can focus more on using the tool itself.

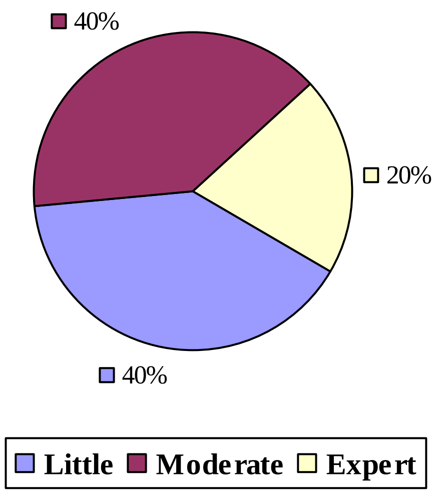


Figure 30 Users' Experience on OWL-S

Figure 30 shows that 80 percent of the users have experience on OWL-S definitions. This is important because only feedbacks from the users who are familiar with OWL-S definitions are useful to evaluate the design of each use case. These feedbacks can be used to improve the interaction design and the way how the tool works

The background information collected from users shown in Figure 29 and Figure 30 ensures that the results of evaluation tests conducted from selected users are trustable, valuable and significant.

6.3.2 Results and Analysis

A post-test questionnaire is used to be filled in by all the users. The data is collected and presented below.

Figure 31 shows that the overall satisfaction on user interface is moderate. A total of 60 percent of the users thought Semantics Editor is easy to use. 20 percent among them even thought it is very easy to use. Only 10 percent of the users thought the tool is very difficult to use. This means the overall user interface design of Semantics Editor is successful.

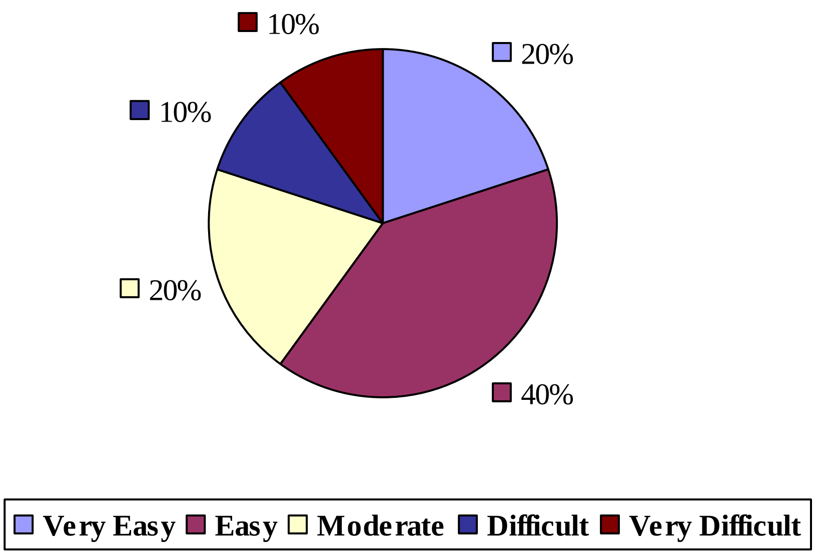


Figure 31 Overall Performance

A table of the usability statistic is shown below (Table 1). The data is collected by asking the users to mark the design of each use case. The marks shown below are the average marks got for each use case of Semantics editor. Five points is the full mark.

Table 1 Use Case Statistic

Function	**Marks**
A. Identify a concept from Javadoc	3.2
B. Identify an association property of a concept	2.4
C. Bind a concept to OWL-S input/output	3.8
D. Create class precondition (Unary predicate)	3.8
E. Create property precondition (Binary predicate)	3.8
F. Tracing between Java interface and OWL-S definition	4.2

A chart diagram of above statistic table is created to show the results. Figure 32 shows that the way of identifying an association property of a concept is almost not satisfied by all users. It got the lowest mark 2.4. This is because Semantics Editor at the moment only shows how the tool works and the round trip behind it. As it is only a prototype version, users spent a long time understanding the user interface. For example, users spent a long understanding fields that can be edit in property edit window. It has to admit that current user interface of Semantics Editor does not present the properties editing function perfectly. Explanations have to be made first before the users start to use the tool to reduce the confusion. A more significant way of identifying association properties should be designed. Also users were tired of typing the texts for association properties repeatedly. The user interface needs to be well designed and improved in the future versions. Improvements are discussed in section 7.2 Further Work.

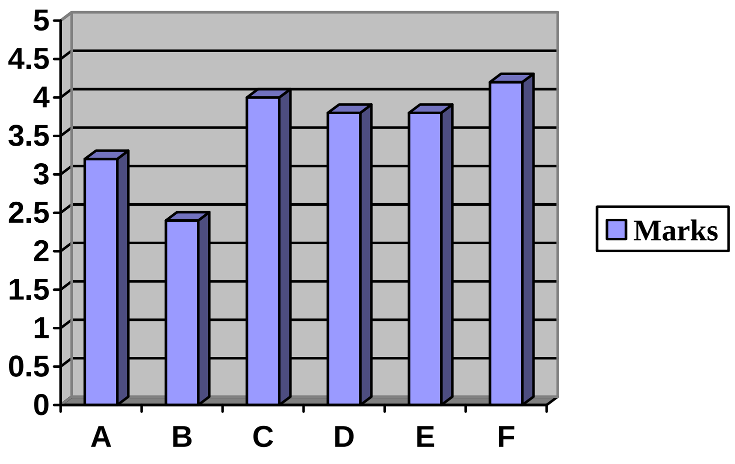


Figure 32 Use Case Performance

It was also found that most of the users showed interest in the traceability provided by the tool. It got the highest mark 4.2. Traceability is a pretty import part in the whole project. Most of the users thought it is a very good function and it will be useful in the OWL-S definitions editing as well as Java coding.

6.3.3 User Comments

Users were encouraged to leave their comments as this is direct feedback from end users and comments will be useful to develop a more user friendly user interface in the future.

6.3.3.1 More Support

The Semantics Editor that was used in evaluation test is just a prototype version. So some users felt that manual text entry reduces the benefit of some of the features since it no longer feels like the IDE is supporting the users. It feels like simply using a text editor.

6.3.3.2 More Automation

More automation should be built in with the tool. For example, auto suggestion of methods that are associated with concepts would be helpful. Perhaps the use of the class view (tree) would aid in this (dragging between class view and semantics outline).

Auto suggestion of methods since Javadoc is associated with a specific method when user is identifying concepts.

6.3.3.3 More Structured

Currently identifying a concept is done by using adding a '#' symbol. It will be more sensible to use something more in line with Javadoc syntax e.g. '@concept' would be better.

A more structured workflow should also be applied. Perhaps if all or most of the concepts were defined first, with the IDE assisting with method associations, then the later stages (adding preconditions) would be easier.

Chapter 7 Conclusions and Further Work

This chapter concludes the dissertation and work involved. Further development on the tool is discussed and described.

7.1 Conclusions

This dissertation involved the state of the art study, an investigation on problems around semantics web service definitions area, an analysis to the problems presented and finally a complete solution was provided to solve the problems.

Different from other approaches which aims to use automation processes to learn ontologies and add semantic meanings to WSDL definitions. This work takes another approach which starts from an engineering perspective to generate semantic web services.

A complete solution was provided. An Eclipse plug-in called Semantics Editor was designed and implemented for retrieving and adding the semantics information from Javadoc which considered as a kind of source to OWL-S definitions. In addition a link between Java interfaces and OWL-S definitions can be generated and visualized by using the Semantics Editor. A data model was designed and can be easily extended and reused according to the concept of model driven architecture.

In conclusion, Semantics Editor is a tool which starts from an engineering point of view to help software engineers add the missing and extra semantics to OWL-S definitions. It also generates and visualizes the links between Java interfaces and OWL-S definitions to enable traceability and maintainability.

Semantics Editor provides the following functions:

- Identifying a concept from Javadoc
- Identifying an association property between concepts
- Binding a concept to OWL-S input/output
- Creating class precondition (unary)
- Creating properties precondition (binary)

- Converting Java interfaces to OWL-S definitions
- Tracing between Java interfaces and OWL-S definitions

7.2 Further Work

While the research results were quite successful and most of the questions posed by the research objectives have been answered, this project still has a big potential for further development.

7.2.1 Full OWL-S Editor Support

At the moment, Semantics Editor only supports creating unary and binary preconditions. But in fact there are more elements in OWL-S definitions e.g. effects, complex preconditions etc. A full OWL-S editor support should be provided. But there is also a problem among the OWL-S community itself. Currently they have not decided which rule language to be the standard language to be supported by OWL-S. Maybe in the next release, this will be decided. Then a full OWL-S editor support can be built into the existing framework of Semantics Editor.

The ideal OWL-S editor should providing following functions:

- **Syntax assistance**: Syntax assistance can help the users get rid of remembering the syntax of elements in OWL-S definitions. With the help of IDE, development period will be shortened and efficiency of development will be improved.
- **Semantics and syntax validation:** Validation is important to ensure OWL-S definitions are correct and consistent. An example of validation can be: validate if a concept referred in the OWL-S definitions is exist and correct.
- **Visualization of contents:** At the moment, when creating the links between Java classes and OWL-S definitions, the tool requires users to find out where the relevant inputs, outputs and preconditions are in OWL-S definitions first. Some users thought it is a little bit difficult to find them rapidly as OWL-S definitions are huge and complex. It is better to visualize the content of the OWL-S definitions, e.g. use a tree structured view, so that the elements can be found easily.
- **Full preconditions generation support:** Skeletons of preconditions mentioned in

section 5.4.2 Context Menu Registration are hard coded at the moment. It is not a very good way of implementation. If more kinds of preconditions need to be defined in the future, the code of Semantics Editor has to be modified. It will be better to get skeletons of preconditions from text files so that users can customize the skeletons by creating text files.

7.2.2 Integrating with an OWL Editor

Due to the time and resource problems, the current version of Semantics Editor is in a prototype stage. It does provide a complete solution, but this solution is text based. For example, the users have to type in the texts to identify association properties. This meant that some of the users could not understand what it is they were doing in the usability evaluation. An OWL editor should be integrated with the Semantics Editor to support a more user friendly graphical user interface. For example, when identifying a concept in Javadoc, instead of manually typing the text users can simply drag the concept from an existing OWL ontology in the OWL editor and drop this into Javadoc editor. And users can also create their own ontology with the help of OWL editor for the web service they developed. Integrating with an OWL editor will make Semantics Editor easier to use and make the user interface meaningful.

7.2.3 Working with Multi-source of Semantic Information

There is a logistic issue behind this project is that if software engineers are lazy and do not leave a very good documentation in the Javadoc. The tool seems to be useless. As a pre-requisition, Semantics Editor requires a very good documentation to be used as a sort of source for semantic information.

As mentioned earlier, semantic information can be also obtained from other sources e.g. XML, UML etc and a lot of attempts have been done to retrieve semantic information from a multi-source of artifacts. As discussed in section 4.1 Architecture Design, starting from an engineering perspective, the Semantics Editor can be extended by adding a XML editor, a UML editor etc to make different kinds of sources be available for obtaining semantic information with the benefit of existing framework.

7.2.4 Integrating with an OWL-S Discovery and Execution Platform

An OWL-S discovery and execution platform can be integrated with the tool to make the tool become a semantic web services development and execution environment.

Then this environment will cover the whole development and execution processes from web services development, to semantic web services generation, discovery and execution.

A decentralized discovery and execution platform [26] for composite semantic web services has been developed in Java last year by Dominik Roblek. Therefore a typical further work can be done is the integration of Dominik’s platform.

References

[1] *D Martin, M Burstein, J Hobbs, O Lassila, D McDermott, S Mcllraith, S Narayanan, M Paolucci, B Parsia, T Payne, E Sirin, N Srinivasan, K Sycara, "OWL-S: Semantic Markup for Web Services", W3C Member Submission 22 November 2004, W3C, available at http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/.*

[2] *E Christensen, F Curbera, G Meredith, S Weerawarana, "Web Services Description Language (WSDL) 1.1", W3C Note 15 March 2001, W3C, available at http://www.w3.org/TR/2001/NOTE-wsdl-20010315 /.*

[3] *M Paolucci, N Srinivasan, K Sycara, T Nishimura, "Towards a Semantic Choreography of Web Services: from WSDL to DAML-S", 2003, The Robotics Institute, Carnegie Mellon University, USA, available at http://www-cgi.cs.cmu.edu/~softagents/papers/isws_ieee_03.pdf.*

[4] *KA Gabrick, DB Weiss, "J2EE and XML Development", 2002, Manning Publications.*

[5] *Semantic Web Layers, available at http://www.w3.org/2006/Talks/1023-sb-W3CTechSemWeb/Overview.html#(19).*

[6] *A Basic Ontology, available at http://en.wikipedia.org/wiki/Image:OntologyBasic.png.*

[7] *MK Smith, C Welty, DL McGuninness, "OWL Web Ontology Language", W3C Recommendation 10 February 2004, W3C, available at http://www.w3.org/TR/2004/REC-owl-guide-20040210/.*

[8] *I Horrocks, PF Patel-Schneider, H Boley, S Tabet, B Grosof, M Dean, "SWRL: A Semantic Web Rule Language Combining OWL and RuleML", W3C Member Submission 21 May 2004, W3C, available at http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/.*

[9] *W3C Working Group, "Web Services Architecture Requirements", Note.Technical report, W3C.*

[10] *Web Service Architecture, available at http://en.wikipedia.org/wiki/Image:Webservices.png.*

[11] *A Ankolekar, D Martin, D McGuninness, S McIlraith, M Paolucci, B Parsia, "OWL-S' Relationship to Selected Other Technologies ", 2004, W3C, available at*

http://www.daml.org/services/owl-s/1.1/related.html#wsdl.

[12] *A Ankolekar, D Martin, D McGuninness, S McIlraith, M Paolucci, B Parsia, "OWL-S' Relationship to Selected Other Technologies ", 2004, W3C, available at http://www.daml.org/services/owl-s/1.1/related.html#soap.*

[13] *A Ankolekar, D Martin, D McGuninness, S McIlraith, M Paolucci, B Parsia, "OWL-S' Relationship to Selected Other Technologies ", 2004, W3C, available at http://www.daml.org/services/owl-s/1.1/related.html#uddi.*

[14] *R Akkiraju, J Farrell, J Miller, M Nagarajan, MT Schmidt, A Sheth, K Verma, "Web Service Semantics - WSDL-S", W3C Member Submission 7 November 2005, W3C, available at http://www.w3.org/Submission/2005/SUBM-WSDL-S-20051107/.*

[15] *Eclipse, available at http://www.eclipse.org/.*

[16] *Javadoc, available at http://java.sun.com/j2se/javadoc/.*

[17] *Object Management Group, Technical Guide to Model Driven Architecture: The MDA Guide v1.0.1", 2003, OMG, available at http://www.omg.org/docs/omg/03-06-01.pdf.*

[18] *N Srinivasan, M Paolucci, K Sycara, CODE: A Development Environment for OWL-S Web services, 3rd International Semantic Web Conference (ISWC2004).*

[19] *K Bontcheva, M Sabou, "Learning Ontologies from Software Artifacts: Exploring and Combining Multiple Sources", Workshop: 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE 2006), International Semantic Web Conference (ISWC'06).*

[20] *Z Duo, L Juan-Zi, X bin, "Web service annotation using ontology mapping", Service-Oriented System Engineering, 2005. SOSE 2005. IEEE International Workshop.*

[21] *Apache, "Axis User's Guide Version 1.2", 2005, available at http://ws.apache.org/axis/java/user-guide.html#Java2WSDLBuildingWSDLFromJava.*

[22] *N Srinivasan, M Paolucci, K Sycara, "Adding OWL-S to UDDI, implementation and throughput", To appear in proceeding of Semantic Web Service and Web Process Composition 2004.*

[23] *Eclipse for RCP/Plug-in Developers, available at http://www.eclipse.org/downloads/moreinfo/rcp.php*

[24] *Java SE Development Kit (JDK) version 6 and Java Runtime Environment (JRE) version 6, available at http://java.sun.com/javase/6/.*

[25] *Eclipse Europa, available at http://www.eclipse.org/europa/.*

[26] *D Roblek,"Decentralized Discovery and Execution for Composite Semantic Web Services", 2006, Department of Computer Science Technical Report, University of Dublin, Trinity College.*

[27] *Unified Modeling Language (UML), available at http://en.wikipedia.org/wiki/Unified_Modeling_Language/.*

[28] *Extensible Markup Language (XML), available at http://en.wikipedia.org/wiki/XML/.*

[29] *DOM Interest Group, "Document Object Model (DOM)", 2005, W3C, available at http://www.w3.org/DOM/.*

[30] *Sun, "Java API for XML Processing (JAXP)", available at http://java.sun.com/webservices/jaxp/index.jsp.*

[31] *KA Ericsson, HA Simon,"Protocol Analysis: Verbal Reports As Data", 1992, NetLibrary.*

Appendices

Appendix A Usability Evaluation Documents

<u>The Usability Process</u>

Usability evaluations seek to determine if the people who use the product can do so quickly and easily to accomplish their own tasks. Usability applies to every aspect of the product in which a person interacts, such as hardware, software, menus, icons, messages, documentation, and help. Evaluations are designed to solicit feedback from participants, focusing on areas of concern identified by our customers. An evaluation typically involves several participants, each of whom represents a typical user.

Once all evaluation sessions are completed, I will compile the feedback received from each participant, along with the notes.

Do you have any questions?

<u>**Pre-Test Questionnaire**</u>

This questionnaire is designed to gather information about your experience on Eclipse IDE and semantic web service. Please circle the number that most clearly expresses how you feel about a particular statement.

1. How many years have been using Eclipse IDE:

A. 0

B. 0~2

C. More than 2 years

2. Do you like to use Eclipse as your main development environment:

A. Yes

B. No

C. I don't care.

3. How much do you know about semantic web service:

A. Expert

B. Moderate

C. Little

4. How much do you know about OWL-S:

A. Expert

B. Moderate

C. Little

Semantics Editor Overview

The Semantics Editor allows individuals to generate semantic web service definition from Java code as well as Java doc. Also the tool helps the user maintaining the link between the Java class and semantic web service definition. The user can trace the definitions that specified with the help of the tool through out Java class and semantic web service definition.

What you can do with the tool?

- Identify a concept from Javadoc
- Identify an association property of a concept
- Bind a concept to OWL-S input/output
- Create class precondition (Unary predicate)
- Create property precondition (Binary predicate)
- Tracing the extra semantics you have added between Java class and OWL-S file

Do you have any questions?

The Setting

You are a software engineer. The company you work for is a service provider which is developing a simple hotel reservation system. The architect required you to be able to generate semantic web service definition (OWL-S) for the final product.

The main classes to look at :

-com.oracle.demo.Hotel.java (contains operations to do the reservations).

-com.oracle.demo.HotelAdmin.java (contains operations to administer the database).

Do you have any questions?

Task 1

You are writing the Java code. During the coding process, you have added some extra semantic information in the form of Javadoc. Now you are going to:

1. Open the Java class with Semantics Editor
2. Identify a concept from Javadoc
3. Bind the concept with a specific method
4. Trace the concept you identified inside the Javadoc

Instructions:

1. To open Java file with Semantics Editor:

a) right click on the filename → open with → Semantics Editor

2. To identify a concept from Javadoc

a) add a '#' symbol right before the term.

b) save and you will see the term appears in the outline view window

3. To bind the concept with a specific method,:

a) select the term in the outline view window

b) edit the value of Method in property view window

4. To perform tracing, simply click on the term in the outline view window

Note: if the outline and property view windows are not active, you can open them from Menu: Window show view

<u>Task 2</u>

After having generated OWL-S semantic web service definition using the existing tool, you are now free to bind the input/output with the concept you identified inside Javadoc.

1. Open the OWL-S Process file using the Semantics Editor
2. Bind the input/output with the concept
3. Trace the input/output inside OWL-S edit window

Instructions:

1. To open the OWL-S Process file using the Semantics Editor

a) click on 'OWL-S Process' tab right below the edit area

b) click on open button and find your OWL-S Process file

c) click on OK button to open

2. To bind the input/ouput with the concept

a) copy the ID of input/output in the OWL-S Process file to the clipboard

b) select the concept in outline view window

c) paste the value to input/output in property view window

3. To perform tracing

a) simply double click on or select ID of input/output in OWL-S Process file

b) you will see the associated concept highlighted in outline view window

Task 3

Another important part of semantic web service definition is precondition. When converting from WSDL to OWL-S, this information is normally lost and can not be recovered. You are going to recover this information and generate the links.

1. Identify unary/binary predicate
2. Generate unary/binary precondition
3. Bind the precondition with the concept
4. Trace the precondition inside OWL-S edit window

Instructions:

1. To identify unary/binary predicate

a) click on the concept in outline window

b) fill in the class/property value (if necessary) in property view window

2. To generate unary/binary precondition

a) copy the ID of certain process which has a precondition missing

b) go to an appropriate position (inside AtomicProcess tag) and right click

c) select either unary or binary precondition generation command

d) drag over the unary/binary predicate from outline view window to the appropriate position inside edit window

3. To bind the precondition with the concept

a) copy the ID of precondition in the OWL-S Process file to the clipboard

b) select the concept in outline view window

c) paste the value to precondition in property view window

4. To perform tracing

a) simply double click on or select ID of precondition in OWL-S Process file

b) you will see the associated concept highlighted in outline view window

Post-Test Questionnaire

This questionnaire is designed to tell us how you feel about Semantics Editor you used today. Please circle the number that most clearly expresses how you feel about a particular statement. Write in any comments you have below each question.

1. Using the Semantics Editor was:

1	2	3	4	5
Very Difficult	Difficult	Neither Easy Nor Difficult	Easy	Very Easy

- Identify a concept from Javadoc

1 2 3 4 5

- Identify an association property of a concept

1 2 3 4 5

- Bind a concept to OWL-S input/output

1 2 3 4 5

- Create class precondition (Unary predicate)

1 2 3 4 5

- Create property precondition (Binary predicate)

1 2 3 4 5

- Tracing the extra semantics you have added between Java class and OWL-S file

1 2 3 4 5

Comment:

____

____

____

____

2. Is the tool helpful?

1	2	3	4	5
Very Useless	Useless	Neither Helpful Nor Helpless	Helpful	Very Helpful

- Identify a concept from Javadoc

1 2 3 4 5

- Identify an association property of a concept

1 2 3 4 5

- Bind a concept to OWL-S input/output

1 2 3 4 5

- Create class precondition (Unary predicate)

1 2 3 4 5

- Create property precondition (Binary predicate)

1 2 3 4 5

- Tracing the extra semantics you have added between Java class and OWL-S file

1 2 3 4 5

Comment:

3. If I could change the Semantics Editor program I would:

Appendix B Detailed Class Diagram

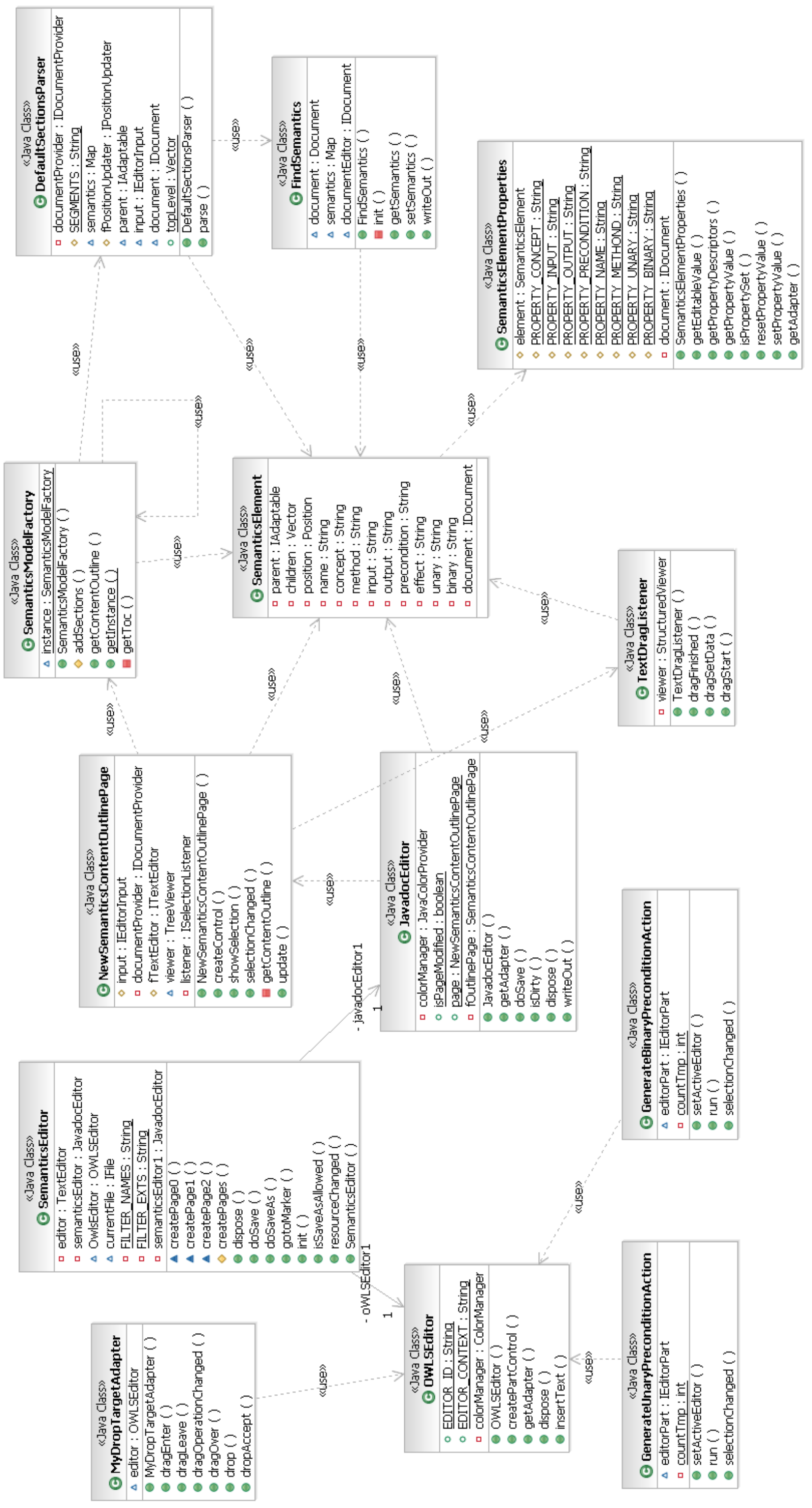


Figure 33 Detailed Class Diagram

Appendix C Commonly Used Abbreviation

OWL-S	Web Ontology Language for Services
WSDL	Web Service Description Language
Eclipse SDK	Eclipse Software Development Kit
Eclipse IDE	Eclipse Integrated Development Environment
UML	Unified Modeling Language
XML	Extensible Markup Language
API	Application Programming Interface