SEMANTICALLY-ENRICHED SOFTWARE REQUIREMENTS SPECIFICATION

by

CHENXIAO FAN

(Under the Direction of Krzysztof J. Kochut)

ABSTRACT

The software requirements specification is one of the major phases in the software development cycle. This is due to the fact that different stakeholders who are involved in the development process may lack a common set of goals driving the software development or even lack a common terminology to express the requirements and goals. Therefore, the requirements specification statements and associated use-case descriptions must be formulated as clearly and with as little ambiguity as possible. In this thesis, we propose to take advantage of templates, consisting of attributes and predefined terms, to control the form of requirements statements. We also present a requirements specification software tool assisting developers in creating semi-formal use-case description statements using templates. The tool was developed as a plug-in to Eclipse, a popular software development environment. The created requirements sentences are more structurally uniform and due to the used ontology, easier to understand to stakeholders.

INDEX WORDS: Template, Software Engineering, Semantic Web, Ontology, Requirements Engineering, Eclipse plug-in

SEMANTICALLY-ENRICHED SOFTWARE REQUIREMENTS SPECIFICATION

By

CHENXIAO FAN

BS, NORTHWEST UNIVERSITY, CHINA, 2011

A Thesis Submitted to the Graduate Faculty of The University of Georgia in Partial Fulfillment

of the Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2014

SEMANTICALLY-ENRICHED SOFTWARE REQUIREMENTS SPECIFICATION

by

CHENXIAO FAN

Major Professor:	Krzysztof J. Kochut
Committee:	Khaled M. Rasheed
	Ismailcem Budak Arpinar

DEDICATION

First of all, A special feeling of gratitude to my loving parents, Jun and Enke Fan whose words of encouragement and push for tenacity ring in my ears. I also dedicate this work to my best friends, Clay and Yingtao who have supported me throughout the entire process.

ACKNOWLEDGEMENTS

These years in UGA have been the most colorful years of my life. I cannot thank enough to Dr. Kochut for my work. Thanks for your guidance and encouragement. I am very grateful to my committee members Dr. Rasheed and Dr. Arpinar for their timely availability and suggestions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Requirements engineering

Requirements engineering is treated as one of the most important parts of the software development. Many techniques have been developed in this research area. The importance of requirements process has been identified as the crucial factor of success of a project. This is due to the fact that different participants involved in the development and they need to build a common understanding, as well as goals, scenarios and requirements statements expressed using these techniques [1].

In software engineering, the requirements are detailed descriptions of the needs for a system. In practice, the requirements are split into two groups. The first group includes functional requirements, which defines what the system must do. The other one is the non–functional requirements, which says the quality of system and additional information about system, for example, its response time or load capacity. The more detailed requirement categorization is shown in Figure 1.1. The figure shows the connection between each level of requirements. The business level is the highest level of goals and can be divided into sub-goals as seen by the user. These user goals can also be divided into product level goals [2].

In general, there are four activities in software requirements engineering: 1) requirements gathering and expression, 2) requirements specification, 3) requirements validation, and 4) requirements management [3]. The usual requirements engineering process follows the above steps to create a requirement document. The first and second activities are known as the intermediate steps to obtain requirements. The stakeholders show their intention for planned system and the developers simply build the system according to the requirements. Stakeholders offer examples to resolve the problems in the first stage, and this makes less conflict because it is easier to reach agreement at the business level. Detailed examples, which are scenarios, can be useful for the planned systems in describing user-level requirements. The product level comes as the last one and it includes difficult considerations because the stakeholders are usually not quite sure about the system's functionality at this level. Ambiguities in system's concepts can lead to potential disagreements between various developers. Thus, we are seeking ways to reduce misunderstanding at the beginning of the requirements engineering.

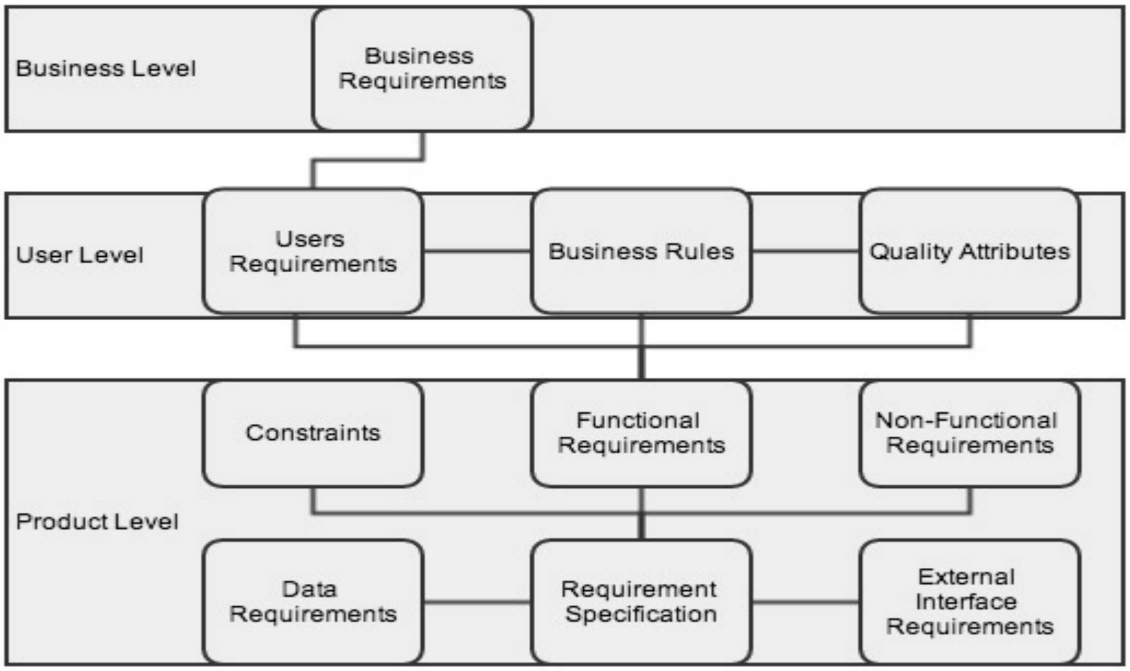


Figure 1.1: Requirement classification

There are several excellent tools and techniques that can help in the requirements engineering process. They assist developers in understanding the stakeholders and building high quality requirement document. One of the most crucial problems in requirements engineering is that most of these documents are written in natural language and it is difficult to process the requirements document by a computer program. If we can improve handing of the semantic aspects of the document, we certainly can obtain an effective tool for requirements' analysis.

1.2 Ontologies

Ontology technologies are widely used in many subjects nowadays. In the context of computer and information sciences, an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members) [3]. Ontologies are typically specified in languages that allow abstraction away from data structures and implementation strategies; in practice, the ontology languages are closer in expressive power to first-order logic than languages used to model databases. For this reason, ontologies are said to be at the "semantic" level [4]. As mentioned above, the ontology offers meaning of word as thesaurus and inference rules on it. Each concept in the ontology represents one atomic element in the domain area and has a specific meaning. That is to say, the thesaurus part of ontology plays role in subject of the semantic aspect.

Considering requirements engineering, the creation of requirements is the demonstration of the exact meaning to stakeholders. The stakeholders will find requirements documents easier to analyze and understand if the requirements, which come from stakeholders' potential needs, are expressed in a uniform format. It also makes correcting problems and establishing universal understanding of a project easier. Well-formulated requirements have certain characteristics, which typically include characteristics are consistency, cohesion, completeness, traceability and verifiability [4].

In this thesis, a domain ontology is used to guide a software engineer in creating use case descriptions. With the help of an Eclipse plug-in [7] we implemented, the developers are able to view and define the meaning of the concepts. What's more, it also aids an analyst or a domain expert identifying the newly found concepts that are relevant and should be added to the existing ontology. These valid concepts are further defined and then added to the domain ontology. The domain ontology provides a common vocabulary, which can be used to improve communication between stakeholders. It can also be used to reasoning over a system, checking information for completeness, consistency and correctness.

Protégé is a free, open source ontology editor and knowledge base framework [5]. Protégé provides support for consistency checking ("can a class have any instances?"), classification ("is A subclass of B?") and instance classification ("which classes do an individual belong to?"). Web Ontology Language (OWL) ontologies consists of:

- Entities, including:
  - Individuals: instances in the domain.
  - Classes: sets that contain individuals.
  - Properties: binary relations.
- Expressions: represent complex notions, e.g. restrictions.
- Axioms: statements that are asserted to be true in the domain.

Usually, properties are divided into two categories. One includes data properties and the other object properties. Data properties can have values of many types: String, Int, Float, Boolean and so on. The restriction types, include, are some (existential), only (universal), min, max and exactly. For a given property P, a minimum cardinality restriction specifies the number of P relationships that an individual must participate in. Figure 1.2 shows how data properties and object properties are represented in an ontology. Object properties specify relationships between individuals, such as “has_a” or “gets_speed_signal_from”. For example: the “online system” has object property “accept payment method” whose domain is “payment method”. The data property is a property related to literals. For example, the concept “online” has data property “hasID”.

In conclusion, domain ontology is capable of specifying knowledge of a domain area. In this thesis, the domain is the problem domain where the system is to be used. The ontology has detailed information about the concepts and relationship of the specified area.

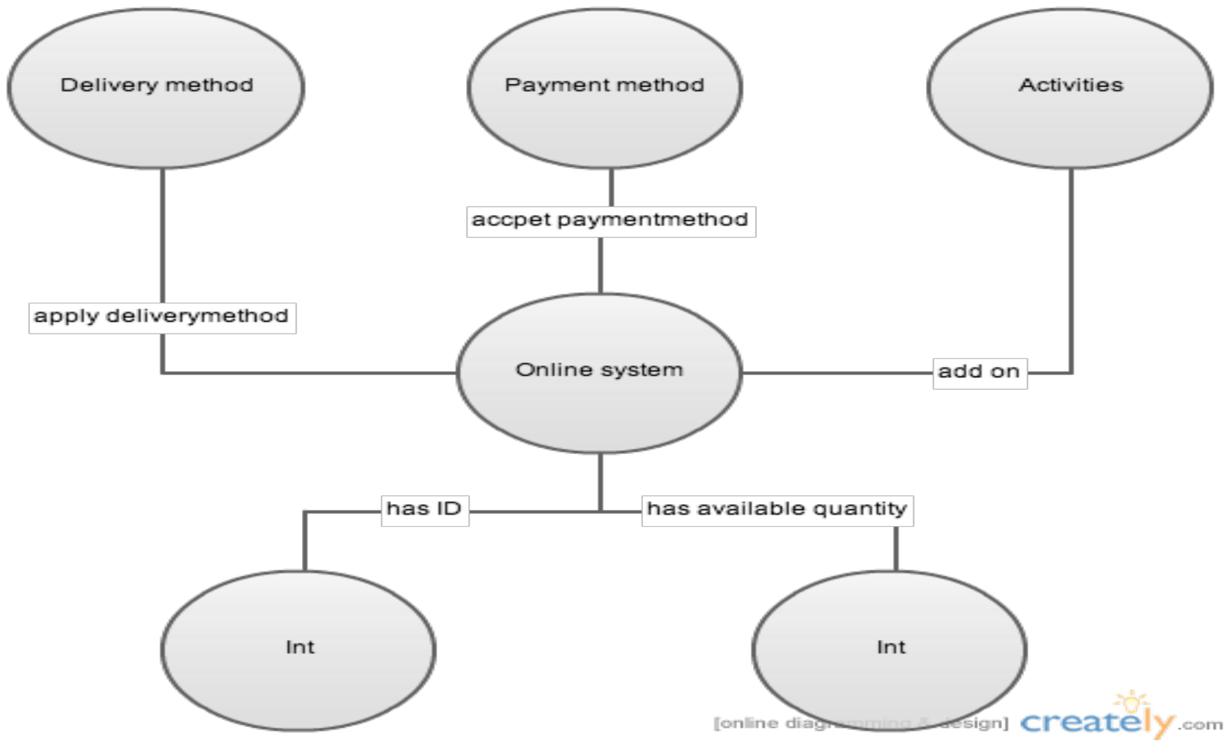


Figure 1.2: Data property and object property.

1.3 Templates

In the industry, requirements are often built using unstructured, free text. Free text offers complete freedom to create the requirements as needed, but with that, the risk of ambiguity and inconsistency increases [6]. Templates are used to structure requirements. Basically, people classify structure expressions into three levels. One is informal structure without any rules applied. Second are semi-formal expressions—particularly the constrained requirements-specification language and templates. The last one includes formal expressions such as state diagrams or other mathematical notations, such as the Z notation [6]. Using template, the requirements can be written in a semi-formal language.

A quite simple example of a template is “ The <entity> shall be able to <capability>”. This is a template to describe the potential entity and its capability. Several templates exist, so software developers can select the most appropriate template for the requirement at hand.

Specific values of attributes, suitable to the domain of the system are filled in. In the previous example, <capability> could be set to “rent a book”. A template with instantiated attributes represents the requirement statement. Figure 1.3 shows the basic structure of a template. The template is a way to be consistent and precise by using same vocabulary across all requirement statements [2].

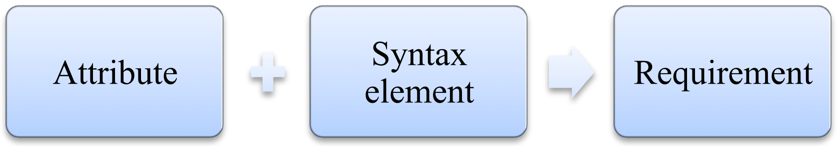


Figure 1.3: Typical contents of a template [2]

In a way, templates are similar to user stories. The differences are that user stories are often written using the following structure: “As a <user> I want to <capability> so that <business value>”. However, a template takes use of predefined attributes and makes instantiating attributes value a part of the process to identify the template that fits the most. Figure 1.4 describes a usage of template. Attributes of template are: User, Capability, Quantity, Time Unit, Operational condition, System function, Action, Entity, State and Effect. To use the example in Figure 1.4, a system’s capability, called distance monitoring, could perhaps be referred to as longitude monitoring or a way to measure the distance. Mixing these terms could not only give widely different results, but could also confuse the developer into believing they refer to three different entities.

Template	The <System function> shall provide <System capability> to achieve <goal>
Attributes	System function: ACC System capability: Distance monitoring Goal: A minimum distance to vehicle in front
Requirements	The ACC shall provide distance monitoring to achieve a minimum distance to vehicle in front.

Figure 1.4: An example of using a template.

1.4 Contributions

There are two main contributions of this thesis to the field of requirements engineering. They are summarized as follow:

1. **Using domain ontology as input to guide creation of requirements**. The domain ontology contains definitions of concepts and possible relationship. It helps to reduce the ambiguity and

reach common understanding. As final result, the requirements are described using a uniformed format within same vocabularies and concepts.

2. **Development of an Eclipse plug-in that utilizes templates and ontologies to guide the creation of requirements statements or use-case descriptions**. The major innovation is to help create formal, easily understandable use case descriptions. The plug-in assists users in creating their own templates and instantiating templates within the concepts in ontology. It also highlights the concept from the ontology in the template instance.

CHAPTER 2

BACKGROUND

2.1 Eclipse

Eclipse is a universal development environment (IDE) first introduced by an IBM Canada project. It contains a base workspace and an extensible plug-in system for customizing the environment. Most frequently, Eclipse is used for computer system development using a programming language, such as Java [7]. It can be used to develop Java applications, various plug-ins and special projects. The Eclipse platform is made up of several components: the platform kernel, the workbench, the workspace, the team component and the help component. The Eclipse architecture is shown in Figure 2.1

The platform kernel task is to get everything started and to load the needed plug-ins. While starting Eclipse, this is the component that runs first. It loads the other plug-ins that people normally think of as Eclipse itself, such as the workbench.

The Eclipse workbench is shown in Figure 2.2. It is the basic graphical interface for working with Eclipse. It includes various toolbars and menus for people to use, and its job is to present those items and the internal windows as shown in Figure 2.2.

The Eclipse workspace

The workspace manages all project resources that includes everything stored on disk or connected machines. Developers' code resides within an Eclipse project. Each project is given its own folder in the workspace directory, which makes it easy to keep track of them. Each project itself can contain many subfolders.

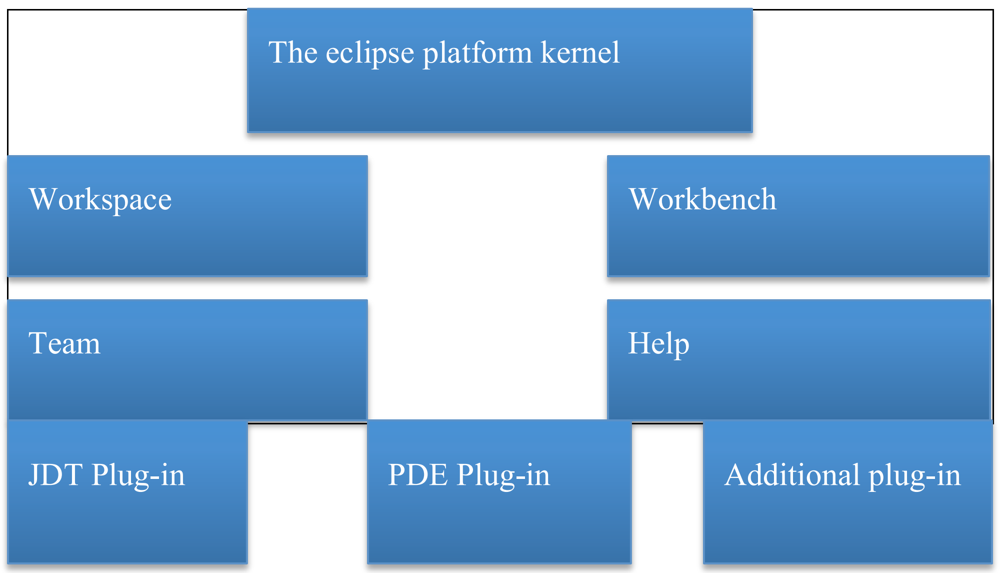
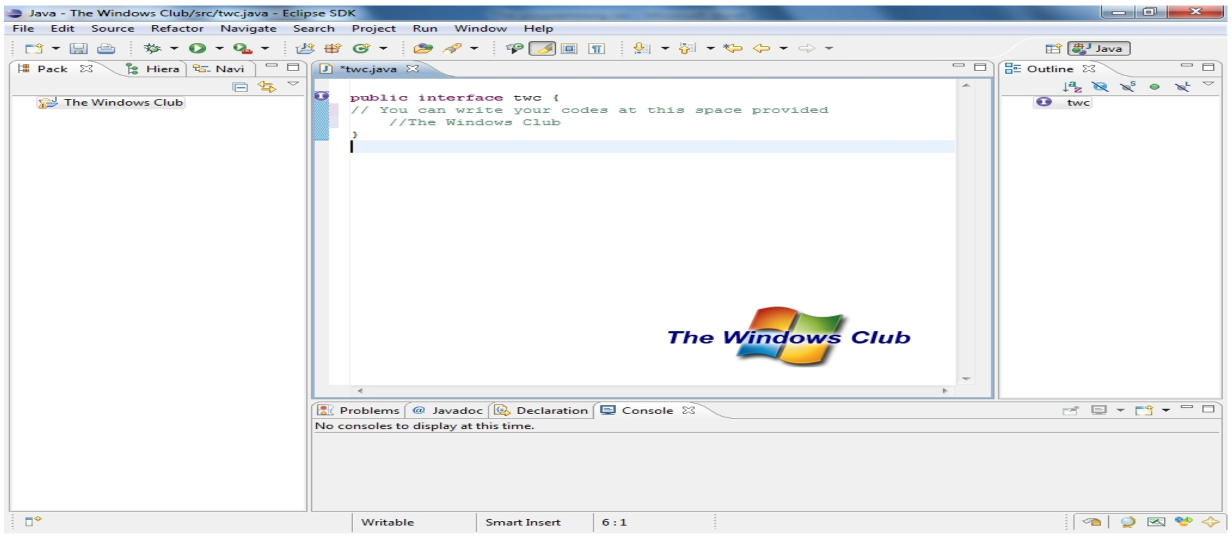


Figure 2.1: The Eclipse architecture [7].

The team component is a plug-in that supports version control in Eclipse. In version control, the program code is checked into or out of a repository as needed so that the changes to that software can be tracked. This is also done so that team members do not overlap or obliterate changes made by other team members.

The help component is an extensible documentation system for providing help; plug-ins can provide HTML documentation with XML-formatted data to indicate how that help documentation should be navigated.


Java - The Windows Club/src/twc.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Pack Hiera Navi
The Windows Club
*twc.java
Outline
twc
Java
The Windows Club
Problems Javadoc Declaration Console
Writable
Smart Insert
6 : 1

```
public interface twc {
// You can write your codes at this space provided
    //The Windows Club
}
```

```
No consoles to display at this time.
```

Figure 2.2: Eclipse workbench.

An Eclipse plug-in is a component that provides a certain type of services within the context of the Eclipse workbench. The Eclipse runtime provides an infrastructure to support the activation and operation of a set of plug-ins working together to provide a seamless environment for development activities. Within a running Eclipse instance, a plug-in is embodied in an instance of some plug-in runtime classes, or plug-in classes. In short, the plug-in class provides configuration and management support for the plug-in instance. A plug-in class in Eclipse must extend org.eclipse.core.runtime. Plug-in is an abstract class that provides generic facilities for managing plug-ins.

In this thesis, we implemented an Eclipse plug-in as a prototype implementation of our requirements specification method. The goal of this implementation is to aid understanding the requirement using semantic techniques. The detailed information will be presented in Chapter 5.

CHAPTER 3

RELATED WORK

Before discussing our approach of dealing with the software requirements, we will talk about several mature or ongoing approaches in requirements engineering. We will also talk about what aspects these approaches can be applied to.

There are several projects dealing with requirements engineering. They mainly concentrate in the following areas: 1) the process of building software requirements ontology, 2) the analysis of source code, 3) the common ontology used to minimize the misunderstanding of requirement concepts, 4) the automatic test case generation from software requirements, 5) software requirement collaboration and management, and 6) the improvement of software maintenance. Surely, there exist other aspects covered by requirement-engineering techniques not discussed here. However, the above six points cover the majority of the requirements' work.

3.1 SWORE

SWORE is an ontology which is used to describe a requirements model with the SoftWiki methodology. The SoftWiki methodology supports a Wiki-based distributed, end-user centered requirements engineering for evolutionary of software development. The core of SWORE includes classes that represent essential concepts of nearly every requirements engineering project. It supports the core concepts such as Requirement, Source, and Stakeholder. It is aligned to external vocabularies including DC-Terms [6], SIOC [28], FOAF [29], SKOS [30] or the tagging ontologies Tags and MUTO [1].

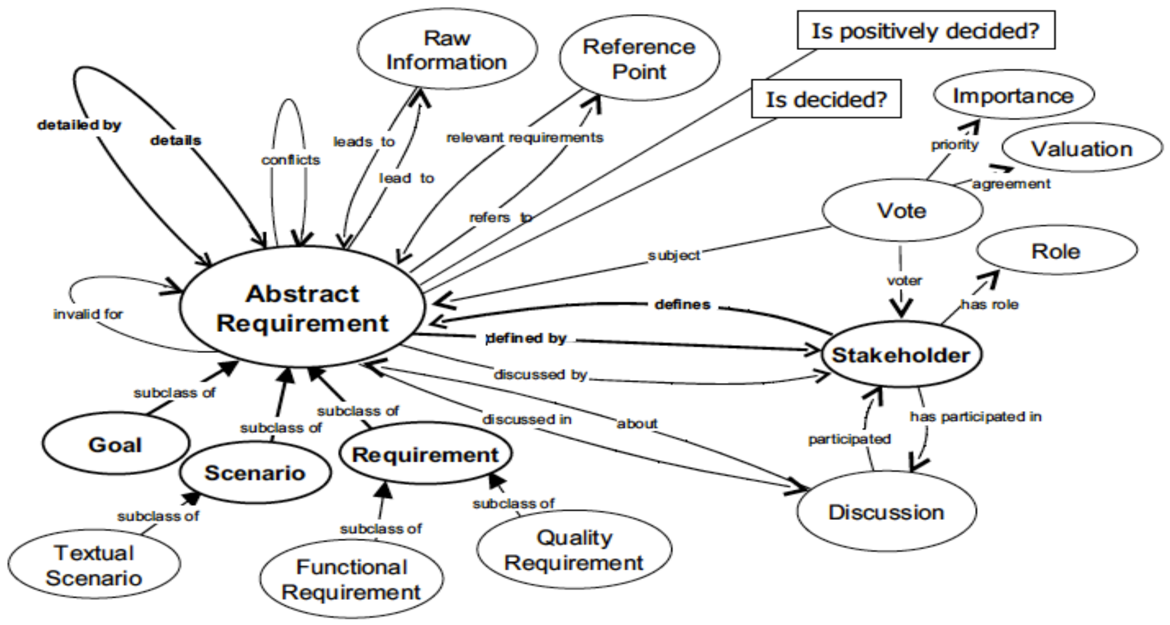
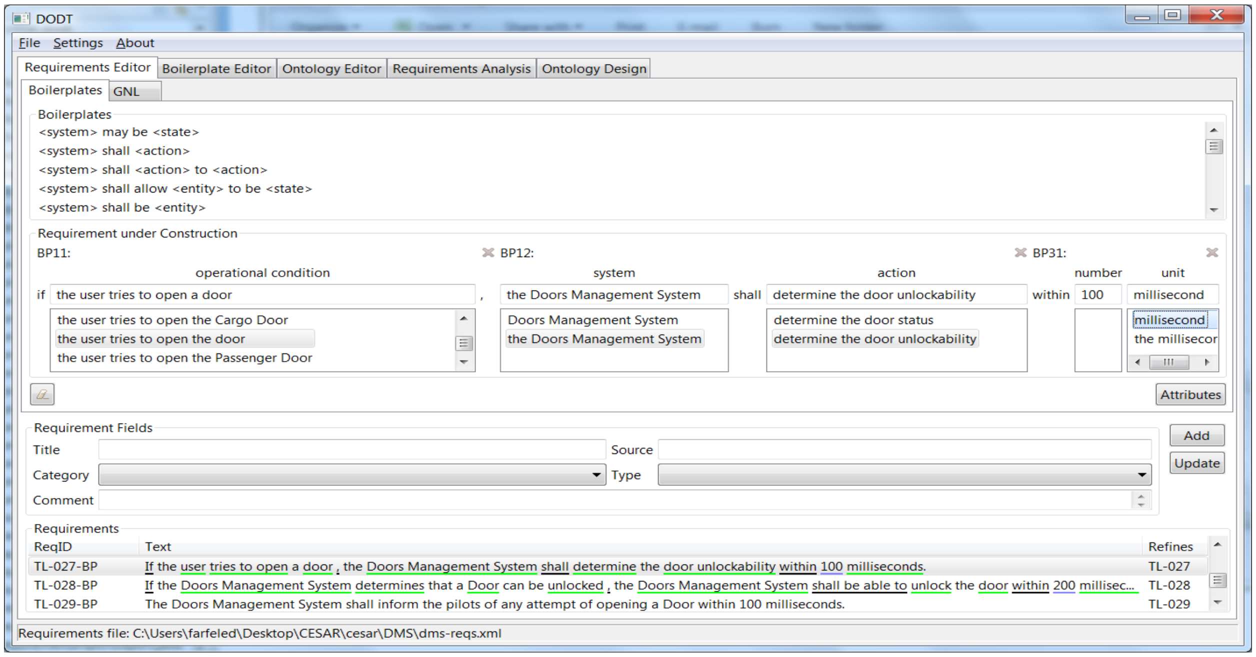


Figure 3.1: Visualization of SWORE [1].

3.2 DODT

DODT is a semi-automatic tool that transforms the natural language requirements into formalized template requirements. The transformation is based on a domain ontology containing basic concepts of the problem area and also upon the natural language processing techniques. The tool reduces the required manual effort for the formalization of requirement statements and further improves the quality of the requirements [8].

DODT provides a list of suggestions to the requirements engineer while creating formulated requirement upon the templates. The provided guidance depends on the attributes that the requirements engineer is currently considering. The idea is to apply an attribute based pre-filter to avoid overwhelming the user with the complete list of ontology entities.



Boilerplates

```
<system> may be <state>
<system> shall <action>
<system> shall <action> to <action>
<system> shall allow <entity> to be <state>
<system> shall be <entity>
```

Requirement under Construction

BP11:		BP12:			BP31:	
	operational condition	system		action	number	unit
if	the user tries to open a door ,	the Doors Management System	shall	determine the door unlockability	within 100	millisecond
	the user tries to open the Cargo Door	Doors Management System		determine the door status		millisecond
	the user tries to open the door	the Doors Management System		determine the door unlockability		the millisecor
	the user tries to open the Passenger Door					

Requirement Fields

Title		Source	
Category		Type	
Comment			

Requirements

ReqID	Text	Refines
TL-027-BP	If the user tries to open a door , the Doors Management System shall determine the door unlockability within 100 milliseconds.	TL-027
TL-028-BP	If the Doors Management System determines that a Door can be unlocked , the Doors Management System shall be able to unlock the door within 200 millisec...	TL-028
TL-029-BP	The Doors Management System shall inform the pilots of any attempt of opening a Door within 100 milliseconds.	TL-029

Requirements file: C:\Users\farfeled\Desktop\CESAR\cesar\DMS\dms-reqs.xml

Figure 3.2: DODT screenshot [8]

3.3 Semantic assistant

In the Semantic Assistant project, the author investigates how to support users in content retrieval, analysis, and development, by offering context-sensitive NLP services directly integrated with common desktop applications (word processors, email clients, Web browsers), Web information systems (wikis, portals) and mobile applications (based on Android). They are implemented through an open service-oriented architecture, using Semantic Web ontologies and W3C Web Services [9].

The author developed an Eclipse plug-in integrated into the Eclipse development environment and into the Semantic Assistants architecture. It provides a user interface for offering various Natural Language Processing services to the users. In particular, when using Eclipse as a software development environment, people can use novel semantic analysis services, such as named entity detection or quality analysis of the source code comments to software developers.

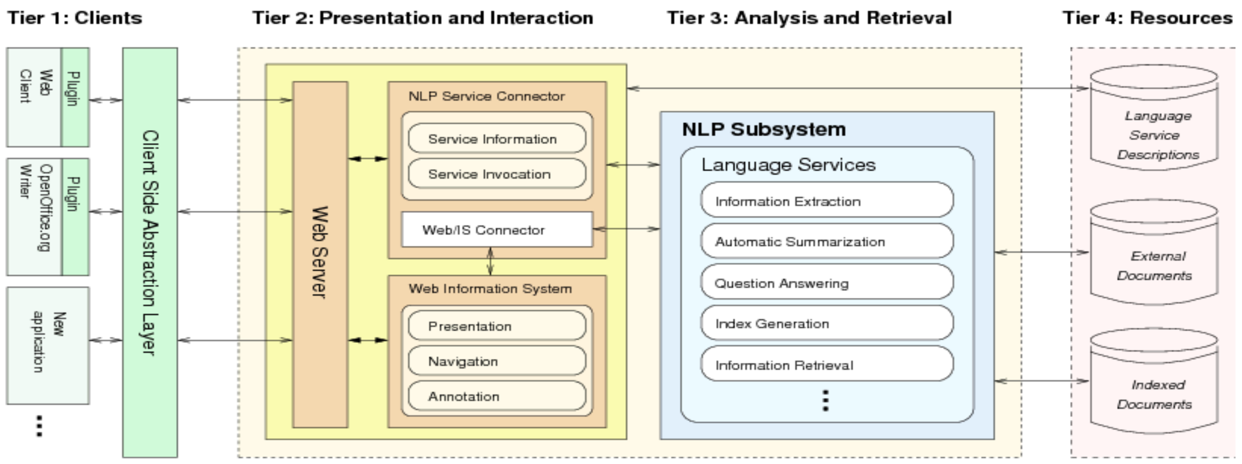


Figure 3.3: Semantic Assistant architecture [9].

3.4 DOORs

DOORs is a requirements management tool that makes it easy to capture, trace, analyze, and manage requirements changes. Control of requirements is key to reducing costs, increasing efficiency, and improving the quality of your products. DOORs offers the following features making it easy to manage the requirements. Developers can manage changes to requirements with either a simple predefined change proposal system or a more thorough, customizable change control workflow through integration with Rational change management solutions. It is possible to link requirements to design items, test plans, test cases, and other requirements for easy and powerful traceability. Business users, marketing, suppliers, systems engineers, and business analysts can collaborate directly through requirements discussions. Also user can use the Open Services for Lifecycle Collaboration (OSLC) specifications for requirements management, change management, and quality management to integrate with systems and software lifecycle tools [10].

CHAPTER 4

SEMANTICALLY ENRICHED REQUIREMENTS

4.1 Controlled requirement language

A bad requirements description language has been recognized by several studies as one of the most critical reasons for project failure [2]. The Standish Group [11] places incomplete and changing requirements second and third on the list of “Project Challenged Factors”. The most common description language for expressing requirements is to use free natural language text. This gives people a lot of flexibility and is easy to learn and use. However, this technique offers almost no support for ensuring the quality of requirements. Thus, both developers and end users have been looking for a new type of language for describing the requirements. As result, the researchers have come up with a template based requirements description language. This language offers the capability to control the variability in formulating requirement statements.

In order to ensure the quality of requirements in development process, some organizations have started to use Controlled Natural Language, a type of language specially designed to address their communicative needs. CNL is a subset of NL that uses:

1. A domain-specific vocabulary, to avoid synonymy (i.e., two different terms referring to the same entity) and ambiguity (i.e., the same term referring to two or more entities in problem domain);

2. A restricted set of grammar rules, which can be general (e.g., ‘write short and simple sentences’), or more formal, counting on grammar rules to constrain the accepted syntactic structure.

Two major groups of CNLs have been identified in application field. They have been designed for different purposes. The most basic purpose is to define a standard to be followed throughout an organization. Another group of CNL is on the more formal side. They help to map from a CNL to a formal representation, which can be understood by machines.

Normally, software requirements can be classified into three categories: formal, semi-formal and informal. Controlled Natural Language is meant to be the middle between formal and informal. Formal requirements are easy to understand for machines, but usually, require a lot of training, and also places several corresponding restrictions on the requirements [12]. UML and the Z notation are examples of formal languages.

The informal representations do not require many restrictions. And they do not have any formulation. Consequently, they are least suitable for being understood by machines. However, informal language is easy to use for stakeholders who do not have any background knowledge in formal requirements [12]. The free text, also known as natural language, is one type of informal language.

The Semi-formal representation is intermediate language. It aids the requirement developers in writing better requirements. The semi-formal language is also regarded as Controlled Natural Language for requirements. It combines the easy of use of informal requirement with the ability to automate work processes surrounding the creation and maintenance the requirements [13].

4.2 Templates

The purpose of using templates is that they help "in knowing how to express certain kinds of requirement in a consistent language" [14]. A template is a free-text based language statement with prescribed words from a standard set. It does not have any constraints and requires minimal expertise. Typically, a template has two parts: attributes and the literal text. Both attributes and sentences are chosen from predefined repository. The reason for providing and maintaining a standard attribute and a syntax repository is to ensure that the same is given to the same composition for all the requirements, and sentences are consistent both throughput the project and between projects within the organization. This is supposed to prevent ambiguities and misunderstandings that can arise if an entity is given different names, or not named properly.

In the example in Figure 1.3, the system distance monitoring could perhaps be referred to by another name. Mixing these entities could probably generate different results and confuse the developer into thinking they are referring to different terms.

We use the attributes whose structure is shown in Figure 4.1. The root node on the top is only used in itself. The root node is parent node of action, entity and quantity. Action is the possible verb or activity used by the potential entity. Entity is a big concept. It includes many terms such as event, goal and many others. The quantity is used for any measure necessary, regardless of whether it is length, weight, duration or any others. It is also important to specify that this can be used in cases where the specific value is not yet determined, in which case it can be instantiated as “TBD”, to be decided. Also, not filling out the quantity can be used in cases where the values are confidential.

By using pre-defined templates, the requirements are likely to be more uniform, preventing vague or ambiguous requirements. New templates can be created according to the user’s demand. As the templates are used over time, the company or organization can gradually achieve uniformity for its requirements, avoiding vague or misunderstanding requirements that lead to costly mistakes.

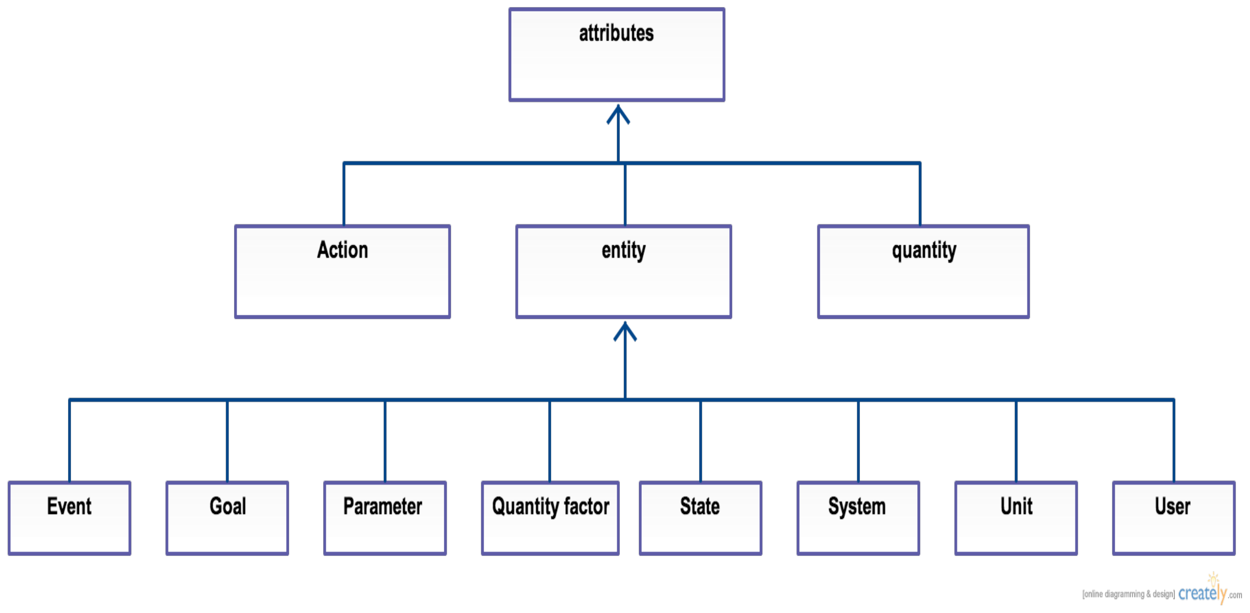


Figure 4.1: Attribute structure [14]

4.3 Template classification

Each template can be classified according to categories listed in Figure 4.2. The words in parenthesis refer to the goal type for the template, such as minimizing or maximizing properties. The following is the complete list of templates, as presented in [8]. The templates are classified into three types; main, prefix, and suffix templates. A main template is able to work as standalone, whereas prefix and suffix templates must be prepended or appended, respectively. The two are sometimes referred to collectively as modes.

Classification of templates
Capability
Capacity (Maximize, Exceed)
Rapidity (Minimize, do not exceed)
Mode (while, if, for …)
Sustainability

Timelines
Operational Constraints
Exception

Figure 4.2: Template classification

Main templates

The main templates are standalone, which means that if the attributes are instantiated they can constitute a requirement on their own. Figure 4.3 shows a few of main templates. Template 1 can be instantiated as " <Laptop> may be < restart>" and No.2 as "<Laptop> shall <prompt a message>".

1	<System> may be <state>
2	<System> shall <action>
3	<System> shall allow <entity> to be <state>
4	<System> shall have <quality factor> of at least <quantity><unit>
5	<System> shall not <action>

Figure 4.3: Main template

Prefix templates

A prefix template needs to be prepended or appended to main template, and can then provide conditions such as " if <event>" or "while <state>". For example, prepending 6 in Figure 4.4 to

5 in Figure 4.3, could create the requirement: " If < In sleep mode>, <laptop> shall not <resume session>."

6	If <Event>
7	If <State>
8	While <State>
9	In order to <Action>

Figure 4.4: Prefix template

Suffix templates

A suffix template is similar to a prefix template. It also needs to be attached to a main template, such as " If <In sleep mode>, <laptop> shall not < resume session> within <5> <seconds>." (10 attached to 6 in Figure 4.4 + 5 in Figure 4.3)

10	With < Quantity><Unit>
11	After <Event>
12	At least <Quantity> times per <unit>

Figure 4.5: Suffix template

4.4 Benefits of using templates

The general goal of using templates is to standardize the formulation of requirements, with regards to both structure and vocabulary, so as to create consistency and prevent ambiguities. The Benefits are summarized as follows [18]:

1. Standardized formulation

The template forces requirements to be expressed in a certain uniform way. It makes similar requirements look alike, which helps the stakeholders (developers, customers, management, etc.) in understanding the requirements better, and also prevents confusion as to what the requirements mean.

2. Preventing ambiguity

The rigidness of a template prevents a vague language and encourages specifying the exact meaning. See example 12, 9, 4 through Figure 4.3 to Figure 4.5.

3. Preventing inconsistency

A clear language and specification of purpose helps developers and others to see the dependencies between requirements, thus preventing inconsistency. See example 9 in Figure 4.4.

4. Uniformity

In addition to the template, the attributes should be selected from a problem domain or a standard repository set, which ensure attributes are named the same throughput the requirements, and also between projects. A common understanding of the requirements reduces the possibility of ambiguities and inconsistencies.

5. Ease of understanding

The template is type of Controlled Natural Language or Semi-formal language. There are certain constraints to the vocabulary and the structure of the sentences, but they are less strict than a formal language such as UML. People have better intuitive understanding of an informal representation than a formal representation. The templates are instantiated to form natural language sentences and should therefore be easier to understand than formal expressions, while still maintaining the benefits of a semi-formal method.

Drawbacks

There have not been much research or experiments preformed yet to measure the performance of using templates. However, some obvious deficiencies include:

1. Reduced flexibility

The template has less freedom to express the requirements as wanted. We already mentioned before, the template is a semi-formal language. Therefore, it has predefined attributes, which the users fill in. On the other hand, it adds constraints to the representations. These constraints may lead the requirements engineers to take shortcuts that may lead to a loss of detail and functionality.

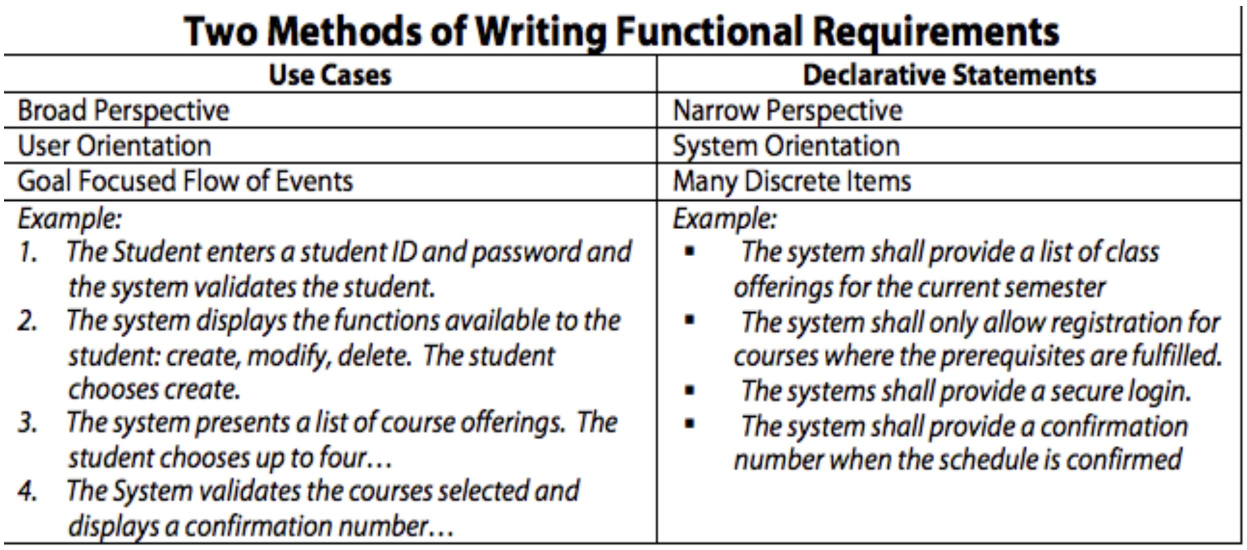
2. Stricter language

A template is more formal than free natural language text. Therefore, it may be argued that templates are more difficult to understand than free text. However, several of the benefits given in Chapter 1 suggest otherwise [19].

4.5 Use Case descriptions and Functional Requirements

In software engineering, there are basically two ways of capturing functional requirements. One is the Use Case form and another is declarative statement describing the functional requirements. Functional requirements statements describe the behavior of the system. This behavior may be expressed as services, tasks or functions the system is required to perform. In a word, functional requirements describe what a software system should do. They also define the fundamental terms related to the potential system and could help track the necessary information required to development process.

Use Cases are derived from the functional requirements statements. Actually, in most situations a single Use Case is typically based on several requirements and a single requirement may appear in several Use Cases. The reason for this is that a use-case does not describe just any activities in the system but a high-level activity triggered by an external source. On the other hand, functional requirements usually describe internal or partial activities and yield functional blocks inside Use Cases.



Two Methods of Writing Functional Requirements

Use Cases	**Declarative Statements**
Broad Perspective	Narrow Perspective
User Orientation	System Orientation
Goal Focused Flow of Events	Many Discrete Items
Example: *1. The Student enters a student ID and password and the system validates the student.* *2. The system displays the functions available to the student: create, modify, delete. The student chooses create.* *3. The system presents a list of course offerings. The student chooses up to four…* *4. The System validates the courses selected and displays a confirmation number…*	*Example:* ▪ *The system shall provide a list of class offerings for the current semester* ▪ *The system shall only allow registration for courses where the prerequisites are fulfilled.* ▪ *The systems shall provide a secure login.* ▪ *The system shall provide a confirmation number when the schedule is confirmed*

Figure 4.6: Comparison of use cases and declarative statements [18]

The idea of Use Cases first appeared in the mid-1980s in [20]. A Use Case describes the proposed functionality of a new system. A Use Case represents a discrete unit of interaction between a user (human or machine) and the system. A Use Case is a single unit of meaningful work; for example, logging in to a system, registering with a system, and creating order are all use cases. Descriptions inside a Use Case define the functionalities, which will be built in to the proposed system. A Use Case may include the functionality of another Use Case or ‘extend’ another Use Case with its own behavior.

A use case diagram is used to graphically depict a subset of the model to simplify the use case description. Typically, there are several use-case diagrams associated with a given model, each of them shows a subset of the model elements relevant for a particular purpose. The same model elements may be shown on several use case diagrams, but each instance must be consistent. If tools are used to maintain the use case model, this consistency constraint is automated, and any changes to the model element (changing the name for example) will be automatically reflected on every use case diagram that shows that element.

Most of the use case models are textual, with the text captured in the use case specifications that are associated with each use case model element. These specifications describe the flow of events of the Use Case. The use case model serves as a unifying thread throughout system development. It is used as the primary specification of the functional requirements for the system, as the basis for analysis and design, as an input to iteration planning, as the basis of defining test cases and as the basis for user documentation basic model elements.

The use case model contains, as a minimum, the following basic elements. Actor: A model element representing each participant. Properties include the actor’s name and a brief description.

Use Case: A model element, which represents each use case. Properties include the use case name and the use case specification. Associations: Associations are used to describe the relationships between actors and the use cases they participate in. This relationship is commonly known as a "communicates-association".

A use case description provides textual details. Briefly speaking, there are three level of use case descriptions. 1) The brief description: it only summarizes the basic information of what system does in response to users' action; 2) Intermediate description: this type of description contains the sequential flow of interaction between user and system and also the constraints to the use case scenario; 3) Full use case description: it expands the intermediate use case description.

In the context of use case description, 'Use cases' are typically related to 'actors'. An actor is a human or machine entity that interacts with the system to perform meaningful work. A use case description generally includes:

1. General comments and notes describing the use case.

2. Requirements, that is what that use case must allow the actor to do, such as <ability to update order>, <ability to modify order>.

3. Constraints, which are rules about what can and cannot be done. They include 1) pre-conditions that must be true before the use case runs, e.g. <create order> must precede <modify order>, 2) post-conditions that must be true once the use case runs, e.g. <order is modifies and consistent>, and 3) invariants which must always be true, e.g., an order must always have a customer number.

4. Scenarios: sequential descriptions of the steps taken to carry out the use case. May include multiple scenarios, to cater for exceptional circumstances and alternate processing paths.

5. Scenarios diagrams: sequence diagrams to depict the workflow.

6. Additional attributes, such as the implementation phase, version number, complexity.

In contrast to use case diagrams, Use Case descriptions capture variation of a use case. Figure 4.7 shows a car rental reservation system use case. In this example, we have three actors who are staff, manager, and customer. They interact through the system, which offers a few functions. It allows the staff to login to the system and maintain business function activities with customers. The customers are able to do all activities involved in car rental business process. The manager is responsible for manage staff and starts up the whole business.

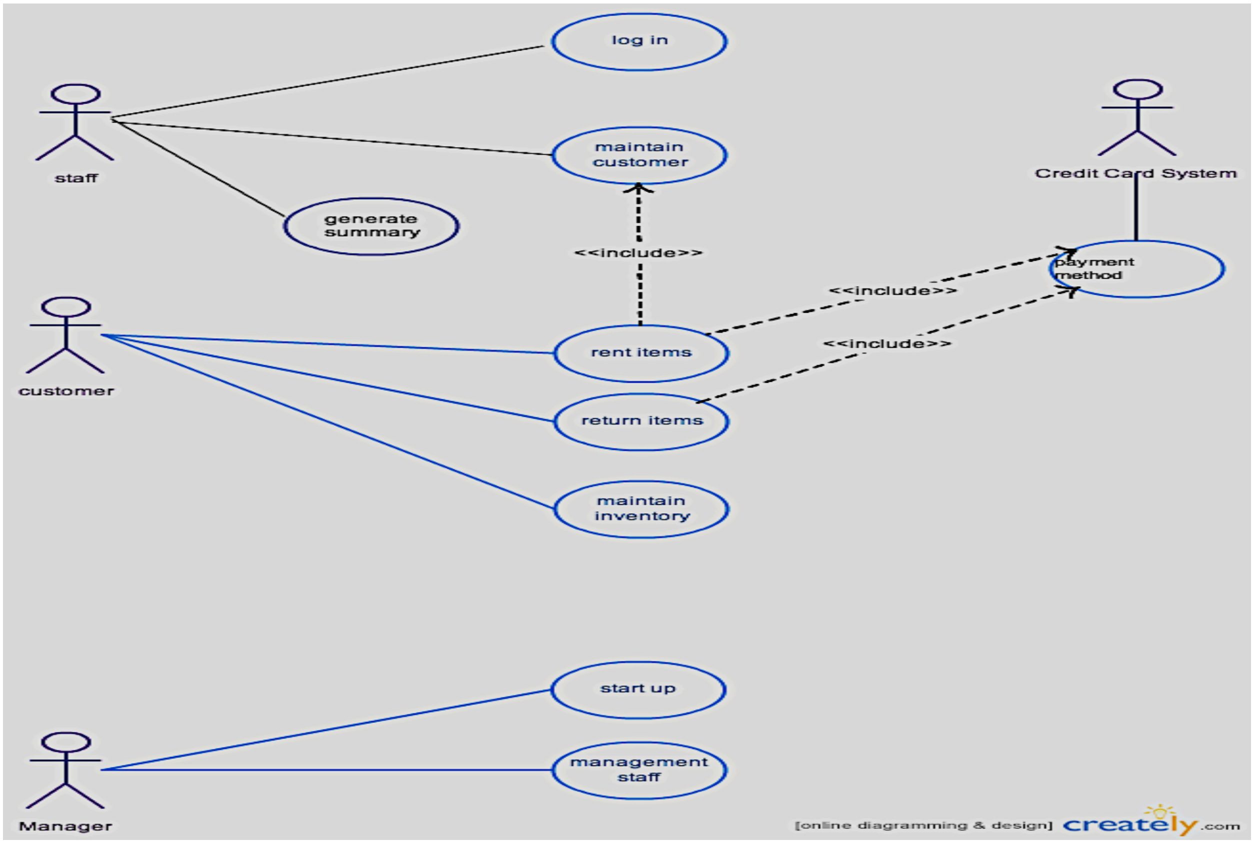


Figure 4.7: Use case diagram of car rental reserve system

4.6 Ontology for requirements specification

The word ontology comes from the Greek ‘ontos’ (being) and ‘logos’ (word) [4]. It denotes the science and the descriptions for the organization, designation and categorization of existence. Carried over to computer science in the field of Artificial Intelligence and Information Technology, ontology is understood as a representational artifact for specifying the semantics or meaning about the information or knowledge in a certain domain in a structured form.

More precisely, ontology is an explicit formal specification of how to represent the entities that exist in a given domain of interest and the relationships that hold among them [16]. In general, for ontology to be useful, it must represent a shared, agreed upon conceptualization. Ontologies have been used in many contexts and for many purposes throughout the years due to, principally, the advent of the Semantic Web. Recently, the use of ontologies in software engineering has gained popularity for two main reasons: (1) they facilitate the semantic interoperability and (2) they facilitate machine reasoning. Researchers have so far proposed many different synergies between software engineering and ontologies. For example, ontologies are used in requirements engineering, software implementation, and software maintenance. An increasing amount of research has been devoted to utilizing ontologies in software engineering, and requirements engineering in particular.

The goal of using techniques in Semantic web towards requirements should be revising the previous requirements process and establishment of well-defined functionalities. The potential use of ontology in Requirements engineering would be: (1) imposing and enabling a particular paradigmatic way of structuring requirements, (2) acquiring structures for domain knowledge, and (3) adding knowledge of the application domain.

The domain ontology contains facts about the domain that are relevant to requirements engineering, i.e., facts that can be used to formulate and to analyze requirements. The domain ontology should be usable to specify requirements for several projects in the same domain. Thus, adding concepts, which are only relevant to a single product, should be avoided. There are three elements about the facts stored in the domain ontology. The following list describes them in Figure 4.8.

In this thesis, ontology is used as a domain expert and guidance in filling in the template attributes in order to formalize the requirement statements. As a result, ambiguity and misunderstanding is reduced and it promotes the completeness and consistency. Furthermore, in this thesis we assume the existence of a suitable domain ontology.

Fact
Concept: Name and definition of the concept itself.
Possible relation between Concepts.
Subclass of Concept
Equivalent class of Concept

Figure 4.8: Domain facts

CHAPTER 5

PROTOYPE IMPLEMENTATION

5.1 Overviews

Most of researches have been conducted in the area of requirements engineering aimed at improving this process. It is helpful to annotate the requirements, to implement collaboration requirements management, and even to generate test cases automatically. The topics of this type of research are varied. It is useful to take a look at these projects and how they achieve the goal of improving requirements process.

The DODT is a tool, which is developed to formalize the requirement statements using templates. But there are still some existing questions: 1) developers have difficulty in getting common understanding of one term in requirement statements, 2) DODT cannot detect the relationships between these concepts, and 3) the standard vocabulary is not updated.

Another tool is called Semantic Assistant, which is an Eclipse plug-in. It uses Natural Language processing techniques to help understanding the JavaDoc [27] inside the source code. It helps people to annotate the meaning of code that is usually hard to read and to make things easy for future work. However, Natural Language processing has some uncertainty and stakeholders who are not programming experts want techniques to aid them in the early stages of the requirements process.

Therefore, we decided to develop a tool with the following capabilities: 1) it should help people understand the requirements in early stages, 2) it should help to formalize the Use Case, 3) it should work with popular development environment Eclipse as a plug-in, and 4) it should use the Semantic Web techniques.

We will focus on the following aspects in this thesis: domain ontologies, templates, and the implementation of the Eclipse plug-in.

5.2 User stories

The requirements are the first concern of this project. We have implemented a tool, which can help collecting, generating the requirements statements and also explaining the meaning of a concept in the requirements statements. Within templates, the requirements are written using the following format: As a <user> I want to <capability or action> so that <benefit or purpose>. This is a user story consisting of whom, what and why. The generic term "user" will be used if the user type is common for customers, developers, and so forth. The sequence of steps is as follow:

S1: Import ontology

The user wants to import existing domain ontology so that it can be used as guidance in filling the attributes and also present the meaning and relationship of concepts.

S2: Generate requirements

The user wants to generate or collect requirements so that already written requirements can be formalized.

S3: Domain concept information

Developers want to be able to see information about a domain concept used in a feature so that he or she can understand what the feature really means.

S4: Update the ontology

Developers should be able to update the domain ontology.

S5: View

Developers should be able to have an overview of requirements.

S6: Concepts in ontology

Developers should be able to know which concept in requirements is not come from ontology.

S7: Quality of requirement

Customers should be able to measure the quality of requirement statements generated by the tool.

S8: Template

Developers should be able to create various style templates. The tool should help them to store their history of using templates.

S9: Workspace

Developers should be able to build a workspace for the requirements that for different purpose or different projects.

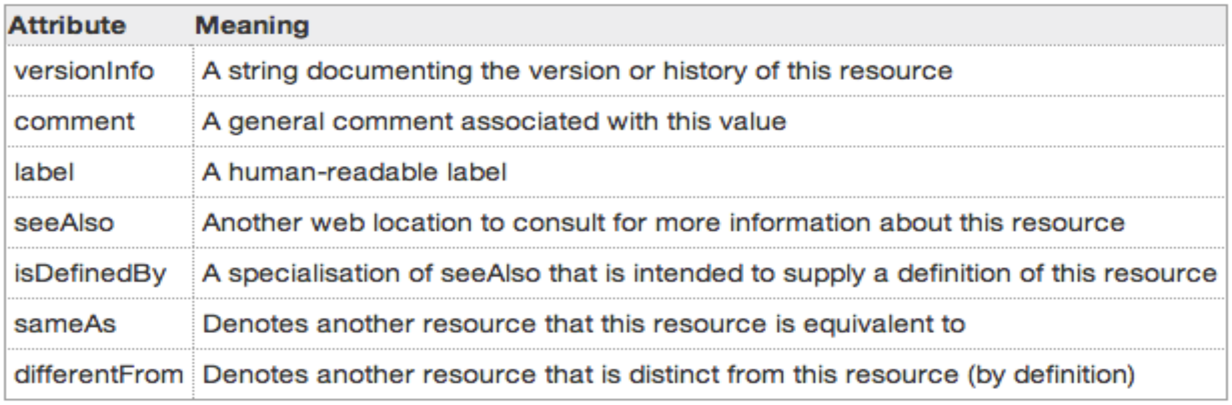
These user stories reflect the basic needs for a tool-assisted requirements statement generator. Furthermore, IEEE Recommended Practice for Software Requirements Specifications states the content and qualities of a good use case description. Therefore we are trying to implement an eclipse based, semantically enriched use case generator according to the practical needs and IEEE recommended practice.

5.3 Plug-in implementation

5.3.1 Ontology Import

Jena is an open source Semantic Web framework for Java [23]. It provides an API to interact with ontology in OWL or RDF. An ontology allows a programmer to specify the concepts and relationships that collectively characterize some domain of interest. In our project, it would be the concepts of potential system's domain. This possible ontology is certainly useful for a requirement process. Since Jena is fundamentally an RDF platform, Jena's ontology support is limited to ontology formalisms built on top of RDF. Specifically this means RDFS, the varieties of OWL.

In order to import an ontology, Jena helps to work with modular ontologies by automatically handling the imports statements in ontology models. We load an ontology document into an ontology model using the OntModel's read method. Each imported ontology document is held in a separate graph structure. Besides, each ontology model has an associated document manager which assists in processing and handling of an ontology document and related concerns. For convenience, there is a global document manager, which is used by ontology models. All of the classes in the ontology API that represent ontology values have OntResource as a common super-class. This makes OntResource is able to store shared functionality for all such classes, and makes a handy common return value for general methods. Some of the common attributes of ontology resources that are expressed through methods on OntResource are shown in Figure 5.



Attribute	Meaning
versionInfo	A string documenting the version or history of this resource
comment	A general comment associated with this value
label	A human-readable label
seeAlso	Another web location to consult for more information about this resource
isDefinedBy	A specialisation of seeAlso that is intended to supply a definition of this resource
sameAs	Denotes another resource that this resource is equivalent to
differentFrom	Denotes another resource that is distinct from this resource (by definition)

Figure 5.1: Common attributes of ontology resources that are expressed through methods on OntResource.

For properties in ontology, Jena has a set of Java classes that allow you to conveniently manipulate the properties represented in an ontology model. A property in an ontology model is an extension of the core Jena API class property and allows access to the additional information that can be asserted about properties in an ontology language. The common API super-class for representing ontology properties in Java is OntProperty. Again, using the pattern of add, set, get, list, has, and remove methods, we can access the following attributes of an OntProperty: Attribute and a sub property of this property. In detail, OWL refines the basic property type from RDF into two sub-types: object properties and data type properties. The difference between them is that an object property can have only individuals in its range, while a data type property has concrete data literals (only) in its range.

In summary, the Jena API offers complete method to interact with ontology. The following code shows how plug-in imports an ontology file.

```
public static void importExtOntology(String externalFilePath){
               SOURCE = path.toString()+"/Ontology/"+Activator.ontology_name;
               folder = path.toString()+"/Ontology";
               NS = "http://www.owl-ontologies.com/2009/DOMCONCEPT#";
               mgr = OntDocumentManager.getInstance();
               base_s = new OntModelSpec( OntModelSpec.OWL_MEM);
               base_s.setDocumentManager(mgr);
               base = ModelFactory.createOntologyModel(base_s);
               String SOURCE_TEMP = externalFilePath;
               base.read("file:"+SOURCE_TEMP);
              try {
                      Writer output = null;
                      File outOntologyDir = new File(folder);
                      File outOntologyFile = new File(SOURCE);
                      if(!outOntologyFile.canWrite()){
                              try {
                                      outOntologyDir.mkdirs();
                                      outOntologyFile.createNewFile();
                              } catch (IOException e) {
                                      e.printStackTrace();
```

```
                                }
                        }
                        output = new BufferedWriter(new FileWriter(outOntologyFile));
                        base.write(output,"RDF/XML-ABBREV");
                        output.close();
```

The following code shows how to load concepts from a domain ontology.

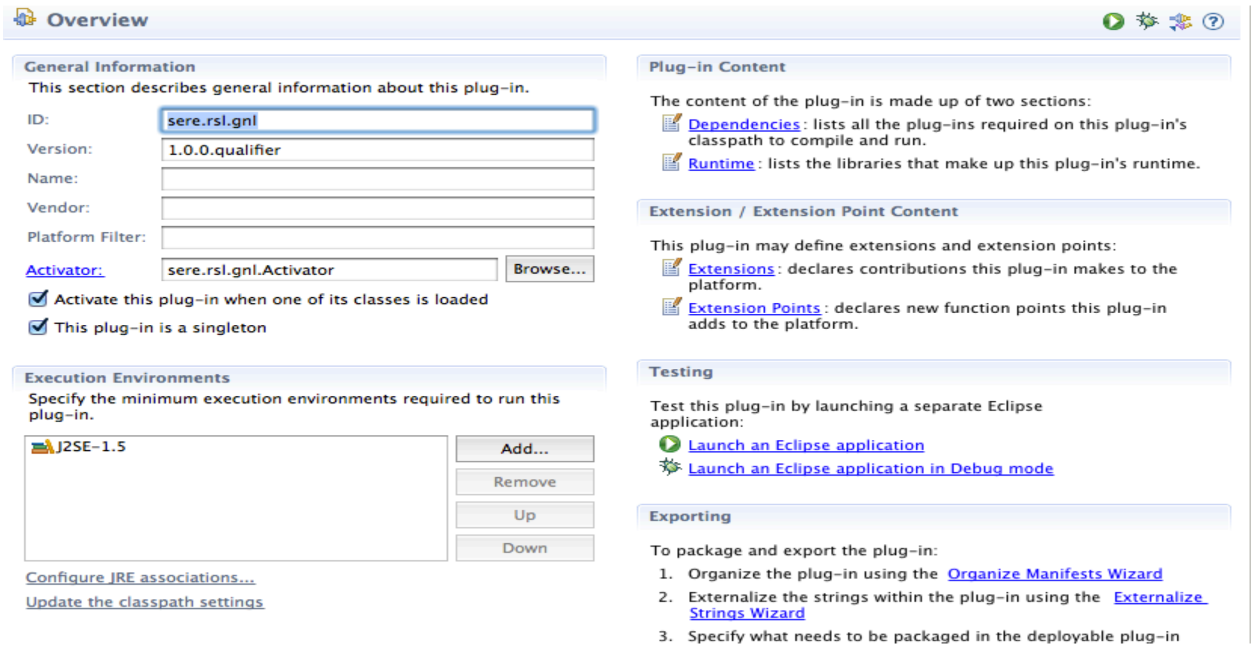
```
public static String [] loadConcepts(){
                String concepts[] = null;
                ArrayList<String> cons = new ArrayList<String>();
                // create the reasoning model using the base
                OntModel infobase = ModelFactory.createOntologyModel(
                                            OntModelSpec.OWL_DL_MEM_RULE_INF, base );
                ExtendedIterator<OntClass> classes = infobase.listClasses();
                while(classes.hasNext()){
                        OntClass c = classes.next();
                        String cn = c.getLocalName();
                        cons.add(cn);
                }
                concepts = new String [cons.size()];
                int i = 0;
                for(String s:cons){
                        concepts[i] = s;
                        i = i+1;
                }
                return concepts;
        }
```

Figure 5.2: Plug-in imports ontology.

5.3.2 Eclipse setup and Installation

Regardless of the operating system, the user needs to install a Java virtual machine (JVM), either a Java Runtime Environment (JRE) or a Java Development Kit (JDK), depending on what is needed with Eclipse. If you intend to use Eclipse for Java development, then a JDK should be installed. In this project, we installed the JDK and its version is 1.6.0_65.

Eclipse can be downloaded from the Eclipse Downloads Page [7]. Eclipse 4.3 is the latest released version. Figure 5.3 shows the properties of our project.



Overview

General Information

This section describes general information about this plug-in.

ID: sere.rsl.gnl
Version: 1.0.0.qualifier
Name:
Vendor:
Platform Filter:
Activator: sere.rsl.gnl.Activator Browse...

☑ Activate this plug-in when one of its classes is loaded
☑ This plug-in is a singleton

Execution Environments

Specify the minimum execution environments required to run this plug-in.

J2SE-1.5

Add... Remove Up Down

Configure JRE associations...
Update the classpath settings

Plug-in Content

The content of the plug-in is made up of two sections:
Dependencies: lists all the plug-ins required on this plug-in's classpath to compile and run.
Runtime: lists the libraries that make up this plug-in's runtime.

Extension / Extension Point Content

This plug-in may define extensions and extension points:
Extensions: declares contributions this plug-in makes to the platform.
Extension Points: declares new function points this plug-in adds to the platform.

Testing

Test this plug-in by launching a separate Eclipse application:
Launch an Eclipse application
Launch an Eclipse application in Debug mode

Exporting

To package and export the plug-in:
1. Organize the plug-in using the Organize Manifests Wizard
2. Externalize the strings within the plug-in using the Externalize Strings Wizard
3. Specify what needs to be packaged in the deployable plug-in

Figure 5.3: Project overview.

Before we can start writing the code, we need to determine how we are going to integrate our project with Eclipse. That is because all extensions to Eclipse are done through plug-ins, Figure 5.4 shows the Eclipse architecture. Plug-ins integrate with each other through extensions on extension points. Eclipse plug-ins typically provide extensions to the platform that support some additional capability or semantics. What is needed is a way for plug-ins to allow other plug-ins to change their behavior in a controlled manner. Eclipse provides an extensibility mechanism that is scalable, avoiding name collisions and does not require compilation of the whole product as a unit, and supports multiple versions of the same component at the same time. Eclipse does this by introducing the notion of a plug-in, which encapsulates functional extensions to the Eclipse platform. Each plug-in has a name, id, provider name, version, a list of other required plug-ins, and a specification for its runtime. A plug-in can also have any number of extension points that provide a portal into which other plug-ins can add their functionality. This is how Eclipse enables other plug-ins to handle the variability supported by your plug-in. In our project, we use several extension points to build our project.

A plug-in is described in an XML file called the plug-in manifest file. This file is always called plug-in.xml, and is always contained in the plug-in sub-directory. The Eclipse Platform reads these manifest files and uses the information to populate and/or update a registry of information that is used to configure the whole platform.

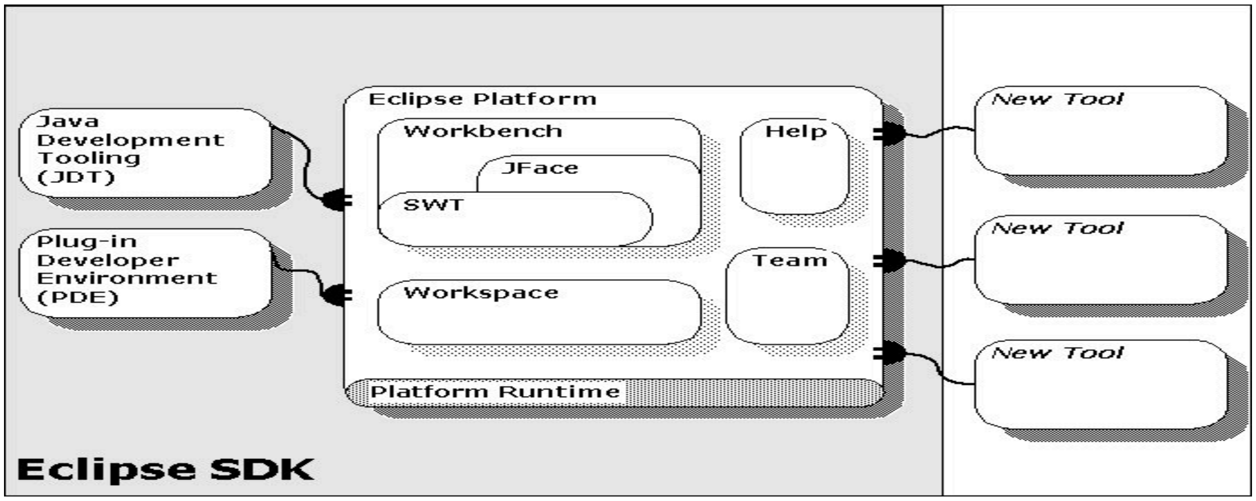


Figure 5.4: Eclipse general architecture [15]

In our Eclipse plug-in, we need a button on the Workbench toolbar. By clicking the button, Eclipse will prompt a separate workbench. New window allows the user to edit the use case description statements, view and edit ontology. The new window also allows the user to create use case description projects. In order to achieve above description, we will need the following extension points: “org.eclipse.ui.actionSets”, “org.eclipse.ui.editors” and “org.eclipse.ui.views”. An action set is a strategy for the addition and removal of menu and toolbar items. A viewer allows plug-ins to add views to the workbench. This strategy is executed if the user explicitly adds the action set to the workbench. An Editor is used by plug-ins to add editors to the workbench.

The action set is declared and given a label. The label (defined in the plug-in.xml) is used to define the extension properties such as icon, id, contribute classes, class path. The extension points are illustrated as below:

```
<extension point="org.eclipse.ui.actionSets">

      <actionSet
            label="sre Action Set"
            visible="true"
            id="sere.actionSet">
         <action
               class="sere.rsl.gnl.views.actions.ReqDashboardToolbarAction"
```

```
                        icon="icons/direction.png"
                        id="sere.rsl.gnl.views.actions.ReqDashboardToolbarAction"
                        label="SRE action"
                        toolbarPath="SREgroup"
                        tooltip="new use case description">
                  </action>
               </actionSet>
            </extension>
```

The above definition will create a new toolbar in the Eclipse workbench with selected icon. It also locates the contribute class which implements the behaviors of toolbar. The Figure 5.5 shows the result.

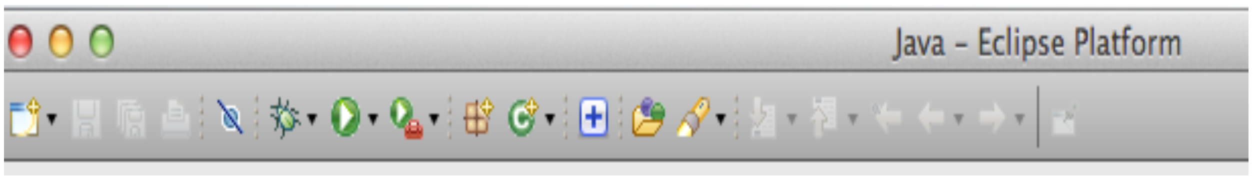


Figure 5.5: A new toolbar button shown in Eclipse workbench.

Other extension points are defined in the same way. Now, we already had fundamental file for the Eclipse plug-in. We declared a plug-in, which is the root element of a plug-in manifest file. The properties of the plug-in defines the plug-in's name, id, version, and provider name.

The plug-in runtime element is how you tell the platform where to find the classes in your plug-in. Essentially, the required libraries and the runtime elements go together to specify the needed "class" for the plug-in in the definition of extension points. This approach allows each plug-in to have its own class independent from any other plug-ins. For the above extension points and other graphic user interface elements, we are using following packages as shown in Figure 5.6.

Package:
org.eclipse.ui
org.eclipse.core.runtime
org.eclipse.jface.text

org.eclipse.ui.editor
org.eclipse.ui.views
org.eclipse.ui.ide
org.eclipse.swt

Figure 5.6: User package

In the user package, we can see that we require plug-in org.eclipse.ui. This allows the platform to find classes while running. In our project, we also refer some outside libraries such as Standford NLP library and Jena. This type of information is defined in the runtime package. Figure 5.7 shows the runtime package.

Runtime package
lib/jena-2.6.2.jar,
lib/stanford-parser.jar,

Figure 5.7: Runtime libraries.

The Activator class provides methods for accessing static resources within the plug-in, and for accessing and initializing plug-in-specific preferences and other state information. It is specified in the plug-in manifest, the activator is the first class notified after the plug-in loads and the last class notified when the plug-in is about to shut down. Thus, we can say the Activator class controls the plug-in's life cycle. The following code shows the Activator class in our project.

```
package sere.rsl.gnl;

public class Activator extends AbstractUIPlugin {

        public static final String PLUGIN_ID = "sere.rsl.gnl";

        private static Activator;

        public static String [] CONCEPTS = null;
```

```
                public static String [] CONCEPTSextended = null;
                public static String[] TAGS = null;
                private static RSLDashboard dashboard;
                private static RSLTemplateeditor; template;
                private static RSLListManager rslListManager;
                private static RSLSaveManager rslSaveManager;
                private static RSL activeRSL;
                public static String plausibleRSLName = "SYSTEM-REQ#";
                public static String plausibleRSLDescription = "#";
```

The extension points' functionality is defined in the contribute class whose path is defined in the.xml. This path tells the platform where to find the contribute class while running. Except for the contribute classes and their associated extension points. The creation and editing of templates is the main focus of our Eclipse plug-in. The class named template model implements these activities according to the template's definition. The Template model defines five lists for template's elements such as preamble, attributes, recursion, model and concrete syntax. The extension point org.eclipse.ui.editor implements the editor interface in the Eclipse workbench. The editor allows people to create a template and instantiate a selected template. It is possible to fill the attributes' value only while instantiating a template. Before starting to edit a template, the user has to import an ontology. Our plug-in loads the selected ontology and its concepts. All concepts are saved in a temporary list. The implementation is described in session 5.3.1. The ontology guides the template instantiation by suggesting the value. If value is similar to a concept in the list, editor will display this concept as a suggestion. Figure 5.8 shows the editors composition in workbench. Figure 5.9 shows the ontology guidance.

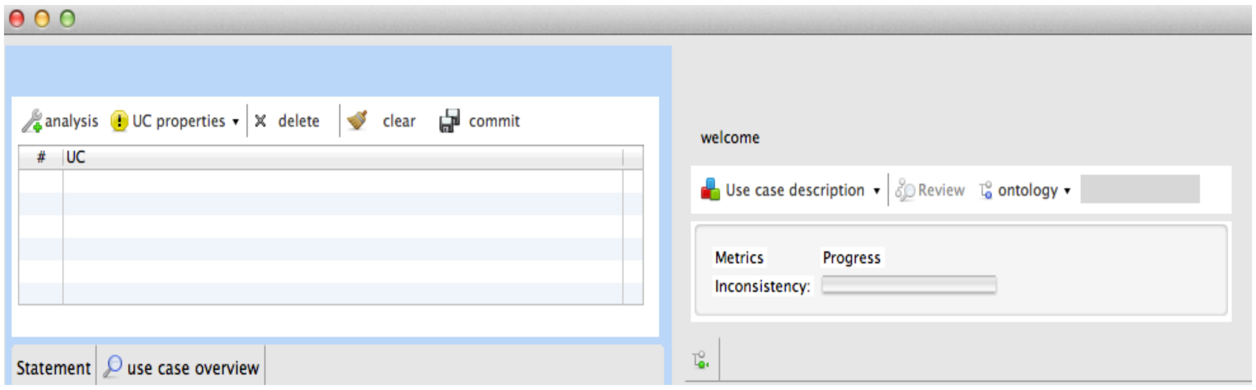


Figure 5.8: Screenshot of the Eclipse plug-in.

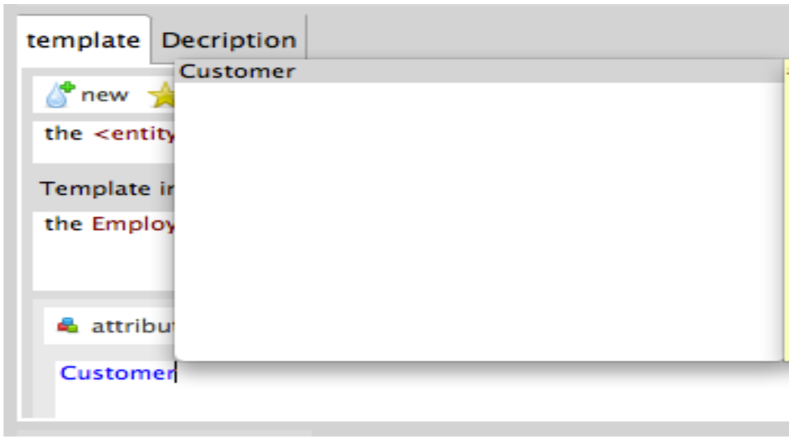


Figure 5.9: The ontology's guidance to template instantiation.

We use SWT (Standard Widget Toolkit) for implementing graphic user interface. We have already registered our workbench and other user interfaces by declaring them in extension points. Each extension point carries the actual contribute class. The following listing shows the code defining an SWT toolbar with the text "new use case".

```
package sere.rsl.gnl.views.actions;

import org.eclipse.jface.action.IAction;

import org.eclipse.jface.dialogs.MessageDialog;

import org.eclipse.jface.viewers.ISelection;

import org.eclipse.ui.IWorkbenchWindow;

import org.eclipse.ui.IWorkbenchWindowActionDelegate;

import sere.rsl.gnl.editors.forms.RSLDashboard;
```

```
public class ReqDashboardToolbarAction implements IWorkbenchWindowActionDelegate {
        private IWorkbenchWindow window;
                   public ReqDashboardToolbarAction() {
        }
                   public void run(IAction action) {
                new SREDashboard().run();
        }
        public void selectionChanged(IAction action, ISelection selection) {
        }
        public void dispose() {
        }
        public void init(IWorkbenchWindow window) {
                this.window = window;}}
```

The following code is the contribute class of above toolbar button. It defines the behavior, which is creating a new window, through clicking the button.

```
public class ReqDashboardAction extends Action{
        public static final String ID = "rsl.rsllist.actions.create.rsl";
        public final REQView view;
        public static NewReqInputDialog newReqInputDialog;
        public ReqDashboardAction() {
                this(null);
        }
        public ReqDashboardAction(REQView view) {
                this.view = view;
                setText("use case dashboard");
                setToolTipText("specification and analysis of requirements");
                setId(ID);
        setImageDescriptor(ResourceManager.getPluginImageDescriptor(Activator.getDefault()
            , "icons/application-monitor.png"));
        setDisabledImageDescriptor(ResourceManager.getPluginImageDescriptor(Activator.getD
efault(), "icons/application-monitor.png"));
                setEnabled(true);
        }
        public void run() {
                new RSLDashboard().run();
        }
}
```

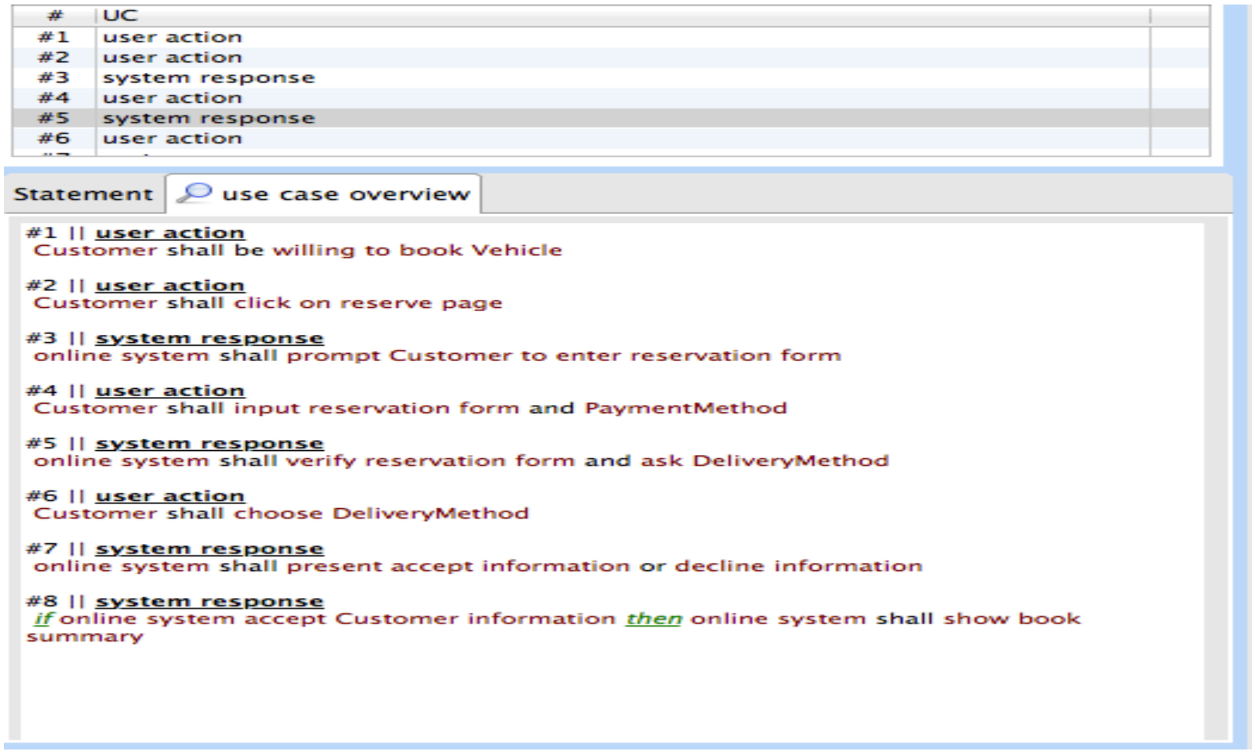
Similarly, there are several graphical user-interface elements used in our project, such as Toolbar, Tablefolder, Progressbar, Button, and Label. The Eclipse plug-in is informed about the selection by its linked listener. This feature is most implemented using the selection service of Eclipse. The following code shows the listener for updating the main window.

```
public void insertReqActionsToolbar(Composite template_composite){

                req_operations_bar = new ToolBar(template_composite,
                   SWT.HORIZONTAL|SWT.SHADOW_OUT| SWT.WRAP | SWT.RIGHT);

                   ToolItem review_item = new ToolItem(req_operations_bar, SWT.PUSH);
                review_item.setToolTipText("analyse requirements");
                review_item.setText("analysis");
                Image enable_review_image =
                      ResourceManager.getPluginImageDescriptor(Activator.getDefault(),
                   "icons/wrench--plus.png").createImage();
                review_item.setImage(enable_review_image);
                review_item.addSelectionListener(new SelectionListener(){
                        public void widgetDefaultSelected(SelectionEvent e) {
                        }
                        public void widgetSelected(SelectionEvent e) {
                                SpecReview.updateViewDashboard();
                        }
                });
```

One of the goals of this project is to create a formalized use case description. We have already talked about how to create a template and how to instantiate a template. The instantiated templates are maintained in the list named “use case statements”. This list is associated with another list named “rsl”, which records the ongoing use case description projects. When a use case description project selected from rsl list, the Eclipse plug-in loads this project’s properties such as name, ID, description, pre-condition and post-condition. All the information is shown briefly in the workbench. The Eclipse plug-in viewer implements the session of displaying the use case description statements. Figure 5.10 shows the Eclipse plug-in viewer.



#	UC
#1	user action
#2	user action
#3	system response
#4	user action
#5	system response
#6	user action

#1 || user action
Customer shall be willing to book Vehicle

#2 || user action
Customer shall click on reserve page

#3 || system response
online system shall prompt Customer to enter reservation form

#4 || user action
Customer shall input reservation form and PaymentMethod

#5 || system response
online system shall verify reservation form and ask DeliveryMethod

#6 || user action
Customer shall choose DeliveryMethod

#7 || system response
online system shall present accept information or decline information

#8 || system response
if online system accept Customer information *then* online system shall show book summary

Figure 5.10: The viewer of use case description statements.

CHAPTER 6

CASE STUDY AND EVALUATION

In this chapter, we will demonstrate how our Eclipse plug-in helps to create requirements descriptions. The background is about to writing a use case description in online car rental industry. This car rent and online reservation system is developed to provide the following services.1) Customer can reserve a vehicle online form anywhere in the world. 2) Every work process activity is done by electronic means, with no need of hardcopies.

6.1 Methodology

6.1.1 Natural language statements

We begin with regular method to build the use case description. User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it operate. This language describes user's goals or tasks that the users must be able to perform with the system. User's requirements therefore describe what the actor will be able to do with the system.

Functional requirements are statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations. It specifies the software functionality that the developers must build into the product to enable users to accomplish their tasks. There are total 4 sub-systems of this online car rental system. We will talk about each sub-system and demonstrate how use case descriptions are created.

Functional requirements:

1. The system must allow the customer to register for reservation.

2. The system shall allow the customer to view detail description of particular vehicle.

3. The system must notify on selection of unavailable vehicles while reservation.

4. The system shall present an option for advanced search to limit the vehicle search to specific categories of vehicles search.

5. The system must allow the customers to select specific vehicle using different search category while reservation.

6. The system must view list of available vehicles during reservation.

7. The system shall allow the customers to cancel reservation using reservation confirmation number.

8. The system shall allow the employee to update reservation information.

9. The system shall allow the employee to view reservations made by customers.

10. The system shall presents information on protection products and their daily costs, and requests the customer to accept or decline regulation terms during reservation.

11. The system must be able to provide a unique reservation conformation number for all successfully committed reservations.

12. The system must be able to display reservation summary for successfully committed reservation.

13. The system shall allow customer to select vehicles in the list.

14. The system shall allow customer staff to Search vehicles by specific record.

15. The system shall allow staff to display all lists of vehicle.

16. The system shall allow staff to display all available Vehicles.

17. The system shall allow staff to display all customers rent record

18. The system must provide printable summary for successful committed rent.

Use Case diagram for above functional requirements shown in Figure 6.1.

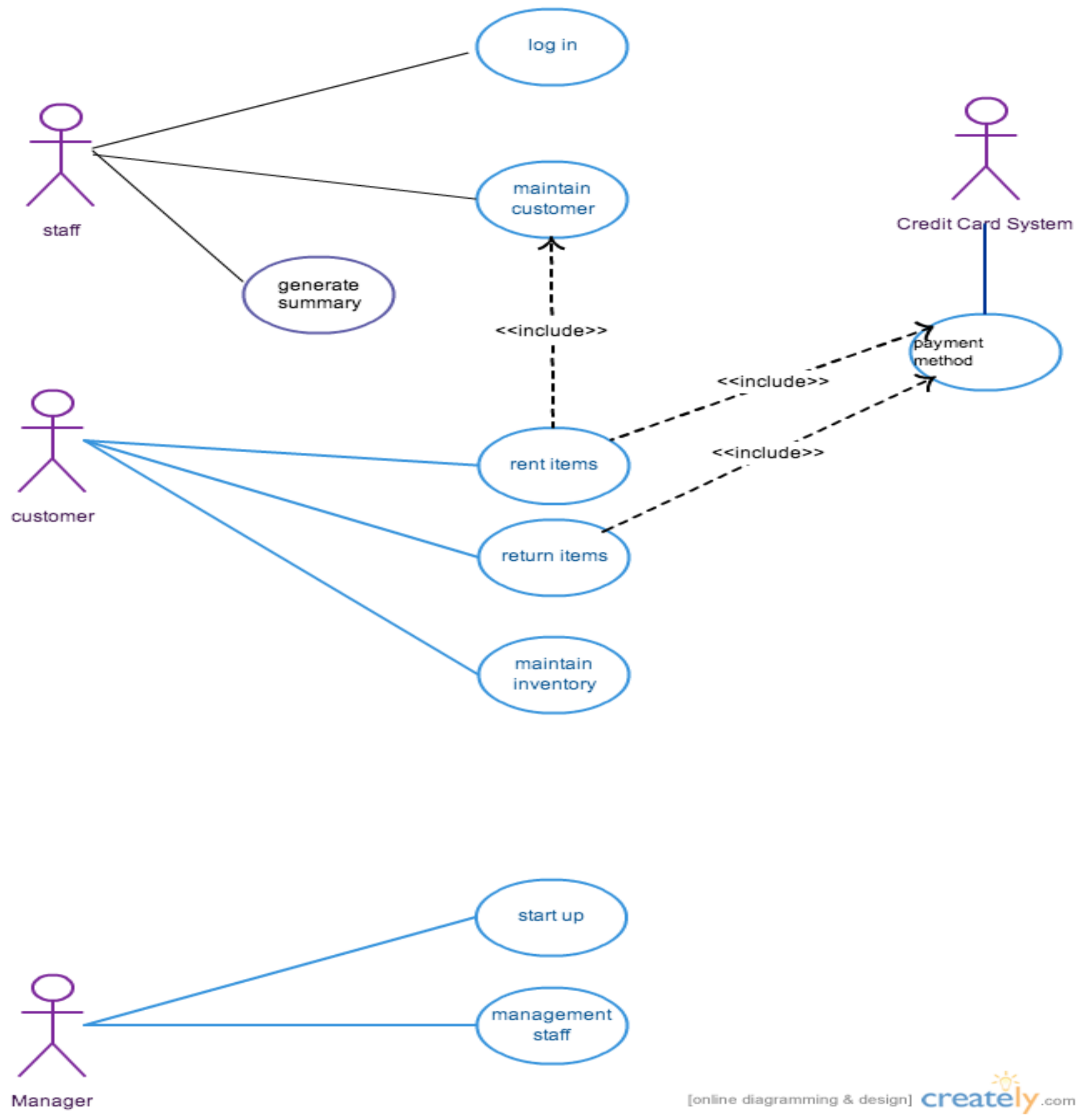


Figure 6.1: Use case diagram

After creating use case diagram, developers need to present an overview of the steps inside the use case you are dealing with use case specification. The use case description will be based on the use diagram. The main purpose of the use case description is to specify any constraints that must be met in order to start the use case, to specify any business rules related to the use case steps, and to show the event flow of the use case steps.

Table 6. 1: Use Case - Login

Use-case Number	UC-01	
Use-Case Name	Log in	
Priority	High	
Actor	Staff	
Description	This use case describes how Staffs to login into the car rental System.	
Precondition	None	
Post-condition	If the use case was successful, the actor is now logged into the car rental system. If not, the system state is unchanged.	
Basic course of Action	**User Action**	**System Response**
	1. The staff is on the home page to login to the system. 3. The staff enters username and password, clicks on Login Button.	2. The system prompts the staff to enter Username, Password. 4. The system verifies that all the fields have been filled out and are valid. 5. The staff successfully logged in the system. 6. Use case Exits
Alternate course of Action	6.1 If all fields are not filled out and not matched to the username and password the system notifies the actor a message Verify Username or Password and then goes back or returns to step 4 of basic course of Action to enter again.	

Table 6.2: Use case – Search Vehicle

Use-Case Number	UC-05	
Use-Case Name	Search Vehicle	
Priority	Medium	
Actor	Staff and customer	
Description	This use case permits staff and customer to search vehicle from the vehicle list in order to display.	
Precondition	UC-3, UC-2	
Post-condition	Display	
Basic course of	User Action	System Response

Action	1. The staff or customers click on search vehicle link. 3. The staffs or customers select one of the following lists from the combo Box, Vehicle Brand. Vehicle Type. Vehicle Model or default is All. 5. Users Click on search button.	2. The system displays combo box to select search for a vehicle. 4. The system displays all information about the vehicle based on selected list. 6. Use case Exists.
Alternate course of Action	4.1 If any lists are not selected from the combo box system goes back or returns to step 3 of basic course of Action to select from the combo box.	

Table 6.3: Use Case - Reserve Vehicle

Use-case Number	UC-02	
Use-Case Name	Reserve vehicle	
Priority	High	
Actor	Customer	
Description	This use case permits customers to reserve and make schedule for renting vehicle, based on the availability of the vehicle.	
Precondition	Customer wants to reserve a vehicle and reservation details about customer have to be entered.	
	Customers reserve successfully	
Basic course of Action	**User Action**	**System Response**
	1. The customer clicks reservation page. 3. The customer enters the following information customer (full name, ID/Passport No, Country, Mobile number and selects vehicle plate number, Pickup date & return date) 5. The customer clicks reserve button to reserve.	2.The system notifies staff and then prompts the customer to fill a reservation form. 4. The system checks all required information had been filled and the date entered dates are valid 6. The system organizes the information and sends it to staff. 7. The system shows the customer that the reservation has been completed, and presents the customer a reservation confirmation number. 8. Use case ends.
Alternate course of Action	5.1 If the customer enters invalid date and time, the system goes back to step 4 to enter the valid date and time. 5.1 If the customer fills invalid information, the system goes back to step 4 to enter the invalid field again. 6.1 If the customer declines the agreement, the system displays a message that reservation canceled.	

Table 6.4: Use case – rent registration

Use-case Number	UC-03	
Use-Case Name	Rent Registration	
Priority	High	
Actor	Staff	
Description	This use case permits to register rental information of the customers and the vehicle that the customer rents.	
Precondition	UC-1	
Post-condition	Customer rent information	
Basic course of Action	**User Action**	**System Response**
	1. The customer wants to take the reserved vehicle. 2. The staff open rent page. 4.The staff enters Full name, Nationality, Country, City, Identification Number, Phone, Plate No, Down Payment, Daily Price, Rent Date, Return Date, Total Rent Day, Total Payment, Refund	3. The system displays a form to be filled out for renting the vehicle. 5. The system displays successful rent summary
Alternate course of Action	None	

All things above are the regular routine for building use case model for potential development. The use case descriptions are written in free text and all various concepts that may refer to the same thing exist in the description document.

6.1.2 Ontology guided use case descriptions

We are going to handle the same scenario shown in functional requirements. As mentioned before, the first step is importing ontology. We use the Vehicle ontology for this purpose. This ontology describes all vehicles for e-commerce. The ontology is designed to use in combination with GoodRelations, a standard vocabulary for the commercial aspects of offers for sale, rental, repair, or disposal. There are a total of 11 classes, 49 object properties and data properties in the ontology. The details of this ontology are shown below:

Table 6.5: Classes in VSO and their definitions.

Brand	A specification of Vehicle
Business entity	The instances involved in the car rental business. It might be organization or person.
Business function	The specification of business services.
Day of week	Use to specify which day the opening hour refers to.
Delivery method	The standard procedure to transfer the service to its final destination based on customer's request
Location	The address of available services
Offering	It bundles to the business function which refers to the detailed service information
Opening hour	The service available time
Payment method	It specifies how customers pay for the services.
Price	The tag price for services.
Product	The detailed information for services.

Table 6.6: Object properties:

Accept payment method	"business entity" and "payment method"
Booking requirements	"business entity" and "business function"
Apply delivery method	"business entity" and "delivery method"
Available at	"business entity" and "location"
Has brand	"business entity" and "brand"
Has opening hours day of week	"business entity" and "day of week"
Has opening hours	"business entity" and "opening hour"
Offers	"business entity" and "offering"
Add on	"business entity" and "business function"
Has business function	" business entity" and "business function"
Has price specification	" brand" and "price"
Has available quantity	"business entity" and " brand"

Table 6.7: Data properties:

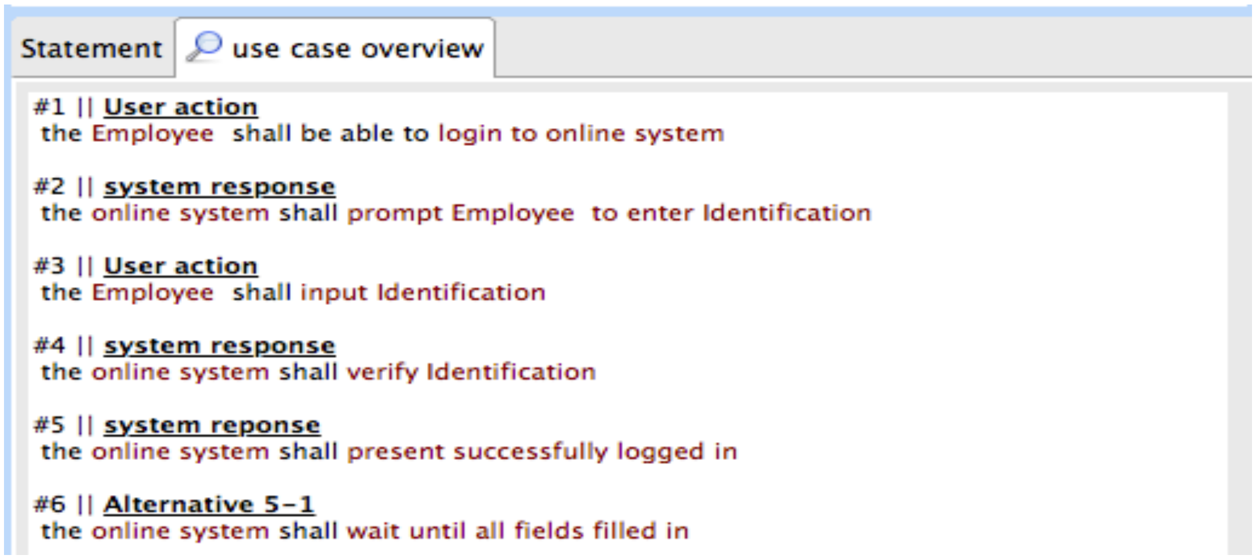
Customer ID	“Business entity” has an unique identifier
Brand ID	Each vehicle “brand” has an unique identifier
Opening duration	The number of opening hours of “ business entity”
Has maximum value	“business entity” has maximum number of vehicle
Condition	“ brand” condition measurement
Eligible location	“location “ identifier
Product ID	“ product” identifier

Based on the above ontology, our tool generates the use case description under the template and ontology. To instantiate the template, people first need to create a suitable template for the corresponding use case description statement. The tool will automatically save the created template to a history list. After composing the template, there are several attributes without values. The ontology will guide the user to fill the value of attributes in the template. Except for attributes guided by the ontology, people are able to instantiate the attributes with reasonable values. Furthermore, each description has some features. You have to select the value for these features from the list.

To start the plug-in, if there is an existing project that needs use case description, the user can create a new use case project by right clicking on project explore. On the other hand, it is possible to start it with just clicking on the tool bar.

For use case descriptions we already showed from the Table 6. 1 to Table 6.4, the following shows the results. The number in front each statement is the unique identifier. Its order describes the event flow order.

Use case login:



Statement | use case overview

#1 || User action
the Employee shall be able to login to online system

#2 || system response
the online system shall prompt Employee to enter Identification

#3 || User action
the Employee shall input Identification

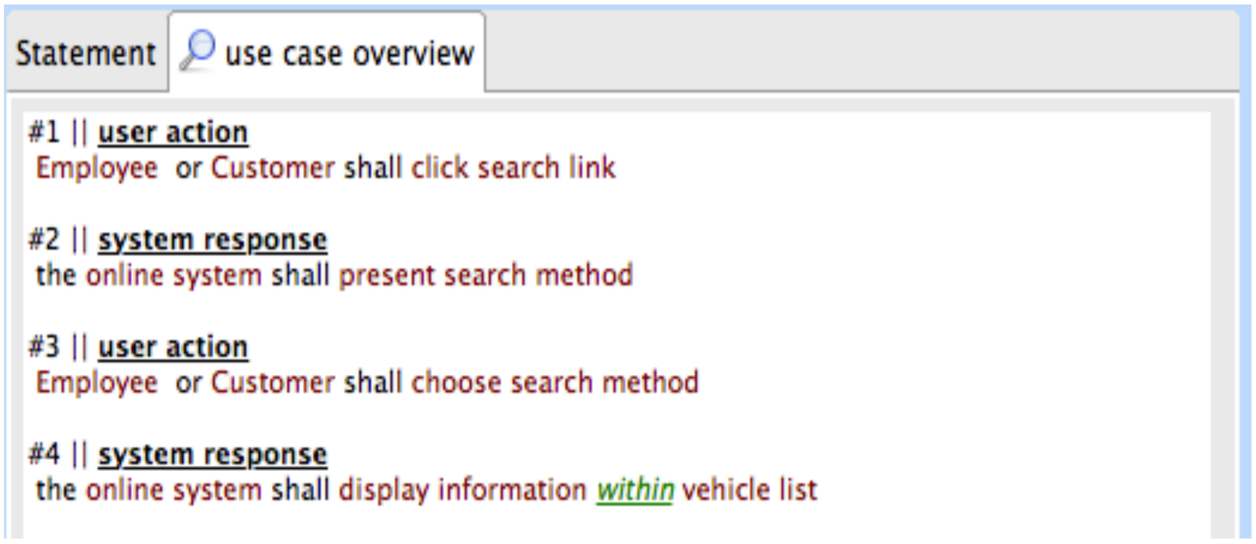
#4 || system response
the online system shall verify Identification

#5 || system reponse
the online system shall present successfully logged in

#6 || Alternative 5-1
the online system shall wait until all fields filled in

Figure 6.2: Screenshot of use case login.

Use case search:



Statement | use case overview

#1 || user action
Employee or Customer shall click search link

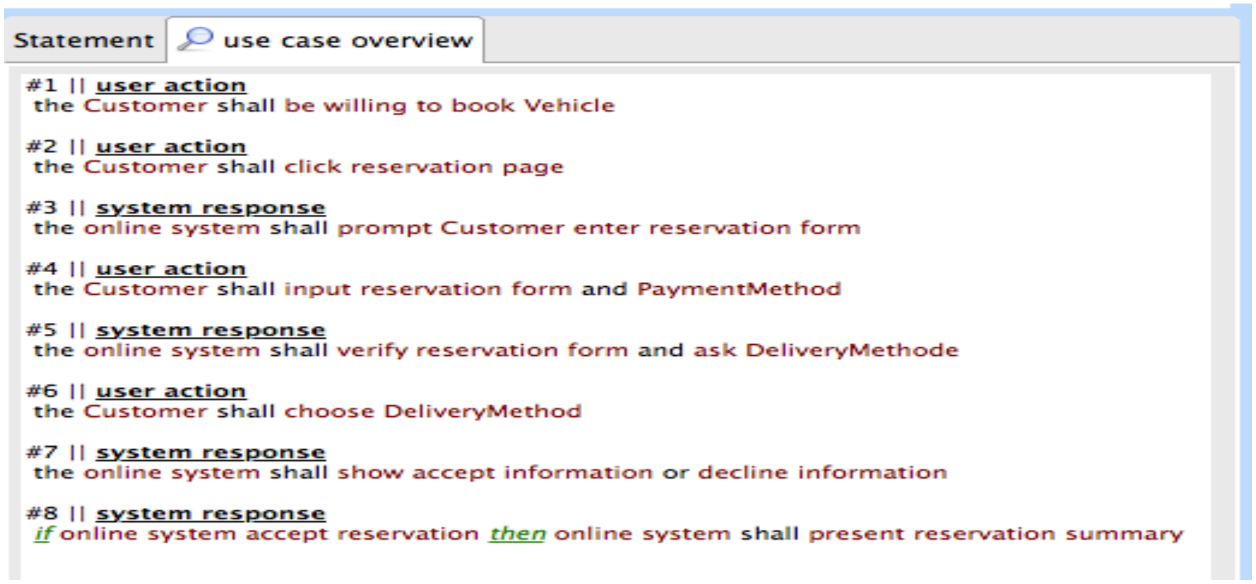
#2 || system response
the online system shall present search method

#3 || user action
Employee or Customer shall choose search method

#4 || system response
the online system shall display information within vehicle list

Figure 6.3: Screenshot of use case search.

Use case reserve:



#1 || user action
the Customer shall be willing to book Vehicle

#2 || user action
the Customer shall click reservation page

#3 || system response
the online system shall prompt Customer enter reservation form

#4 || user action
the Customer shall input reservation form and PaymentMethod

#5 || system response
the online system shall verify reservation form and ask DeliveryMethode

#6 || user action
the Customer shall choose DeliveryMethod

#7 || system response
the online system shall show accept information or decline information

#8 || system response
if online system accept reservation *then* online system shall present reservation summary

Figure 6.4: Screenshot of use case reservation

For each of the use case statement, the term from ontology will be labeled in blue color. For example: the Employee shall be able to login to online system. The concepts: employee and online system come from the ontology. Besides the detailed statements, the other information such as use case description and pre- or post- conditions will be shown briefly in the window and can be viewed completely by clicking UC properties.

CHAPTER 7

CONCLUSION AND FUTURE WORK

The objective of this thesis was to take advantage of using templates and a domain ontology to generate use case descriptions. The created tool should assist the user in generating use case descriptions directly. This project used part of the DODT project, which helps in importing ontology and creating template session. The advantage of the tool-assisted approach is that it provides a range of possibilities, as the ontology can be used in different ways to help the requirements engineer. When the user writes a use case description, the ontology can be used to give universal understanding of the concept, and the template can be used to formalize the use case description. The ontology is also used to give suggestions for writing use case descriptions. The main disadvantage of the approach is the lack of automation. The users have to compose their templates manually and also create the template instances manually. In conclusion, we have developed an Eclipse plug-in that combines templates and ontology. The approaches identified involved different degrees of semantics. The domain concepts and relationships are easily accessible and provide a common vocabulary for the requirement domain.

Recommendations for further work will be divided into whether the continued work would be performed in an academic or industrial setting. In an academic setting, the major point of interest is allowing people to use a universal ontology such as DBpedia (constructed of Wikipedia) or Freebase [26]. This is the task of suggesting the most universal meaning based on what is already written. A similar task is to find a way to determine when the states are specified "sufficiently", meaning that each step in the created descriptions is valid and contains enough information so that it actually makes sense from a domain perspective. Also, we could increase the amount of automation to reduce the manual work of users on template composition. In an industrial setting, a company should pursue a more direct approach. The implementation could be tailored to whatever enterprise-wiki the company uses, with the goal of using the domain ontology as a natural tool in the development process. The wiki can make the use of the ontology more accessible to everyone, and domain information can be stored here. It will also support collaboration between different users.

The Eclipse plug-in project has been a proof of concept. We demonstrated its usefulness through the case study. However, a comprehensive user study is needed to assess the advantages of this plug-in over regular natural language requirements descriptions. Therefore, we hope to conduct a comprehensive human based evaluation to the created plug-in.

REFERENCES

[1] Riechert, T. L. (2007). Towards semantic based requirements engineering. *Proceedings of the 7th International Conference on Knowledge Management.* Graz.

[2] Undheim, O. (2011). *Semi-automatic Test case generation.* Norway: Norwegian university of science and technology.

[3] Farfeleder, S. M. (2011). Ontology-driven guidance for requirements elicitation. *The Semanic Web: Research and Applications* , 212-226.

[4] Gruber, T. (n.d.). *Ontology*. Retrieved February 11, 2014, from ontology defination: http://tomgruber.org/writing/ontology-definition-2007.htm

[5] University, standford. *Protege.* http://protege.stanford.edu/ (accessed March 20, 2014).

[6] Bande, Roshan, and K. N. Hande. "Analysis of requirement using ontology: Survey." *International Journal of Engineering*, 2012: 10.

[7] Fundation, eclipse. *eclipse.* https://www.eclipse.org/ (accessed March 22, 2014).

[8] Farfeleder, S., Moser, T., Krall, A., Stalhane, T., Zojer, H., & Panis, C. "DODT: Increasing requirements formalism using domain ontologies for improved embedded

systems development." *Design and Diagnostics of Electronic Circuits & Systems* . IEEE, 2011. 271-274.

[9] Semantic Software Lab. *Semantic Assistant.* http://www.semanticsoftware.info/ (accessed February 10, 2014).

[10] IBM. "Getting start with rational DOORs." http://publib.boulder.ibm.com/infocenter/rsdp/v1r0m0/topic/com.ibm.help.download.doors.doc/pdf92/doors_getting_started.pdf (accessed April 20, 2014).

[11] The Standish Group International, Inc. *THE CHAOS REPORT.* The Standish Group International, Inc., The Standish Group International, Inc., 2009.

[12] Castañeda, Verónica, Luciana C. Ballejos, and Maria Laura Caliusco. "Improving the Quality of Software Requirements Specifications with Semantic Web Technologies." *WER.* santa fe: WER, 2012.

[13] ASD. *ASD-STE100 Simplified Technical English.* brussels: ASD, 2013.

[14] Elizabeth Hull, Ken Jackson and Jeremy Dick. *Requirements Engineering.* Springer, 2005.

[15] Maryland, University of. "Introduction to Eclipse." http://www.csee.umbc.edu/courses/undergraduate/341/fall08/Lectures/Eclipse/intro-to-eclipse.pdf (accessed March 25, 2014).

[16] Siegemund, Katja, Edward J. Thomas, Yuting Zhao, Jeff Pan, and Uwe Assmann. "Towards ontology-driven requirement engieering." *Workshop Semantic Web Enabled Software Engineering at 10th International Semantic Web Conference* . BONN: ISWC, 2011.

[17] Zoer. Norway: NATU, 2011.

[18] Johannessen, Vegard. *CESAR - text vs. boilerplates* . Thesis, NATU, 2012.

[19] Carew, D., Exton, C., and Buckley, J. "An empirical investigation of the comprehensibility of requirements specifications." *International Symposium on Empirical Software Engineering (ISESE).*, 2005.

[20] Jacobson, Ivar. *Use Case 2.0. The guide to use case succeed.* Ivar jacobson international, 2011.

[21] W, lee . and zhao. "Domain Requirements Elicitation and Analysis – An Ontology-Based Approach ." *Proceedings of the First International Multi-Symposiums on Computer and Computational Science* , 2006: 805-813.

[22] Shibaoka, M., Kaiya, H. & Saeki, M. "GOORE: Goal-Oriented and Ontology Driven Requirements Elicitation Method." *ER workshops.* 2007. 225-234.

[23] W3g. *OWL, web ontology language.* www.w3g.org (accessed March 20, 2014).

[24] Omoronyia, I., Sindre, G., Stålhane, T., Biffl, S., Moser, T., & Sunindyo, W. *In Requirements Engineering: Foundation for Software Quality.* Springer Berlin Heidelberg, 2010.

[25] Delia Rusu, Lorand Dali, Blaž Fortuna, Marko Grobelnik, Dunja Mladenić. "TRIPLET EXTRACTION FROM SENTENCES." *In Proceedings of the 10th International Multiconference.* Information Society-IS , 2007. 8-12

[26] Wikipedia, The Free Encyclopedia. Retrieved November 18, 2007, from www.wikipedia.org.

[27] Oracle. Javadoc tool. Retrieved April 28, 2014, from Javadoc documentation: http://www.oracle.com/technetwork/java/javase/documentation/index-137483.html#usingHead

[28] SIOC. SIOC project. Retrieved April 28, 2014, from http://sioc-project.org/

[29] W3C. Friend of Friend project. Retrieved April 28, 2014, from http://www.foaf-project.org/

[30] Simple knowledge organization. Simple knowledge organization. Retrieved April 28, 2014, from http://www.w3.org/2004/02/skos/