Automating Transition from Use Cases to Class Model

by

Dong Liu

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

CALGARY, ALBERTA

JULY, 2003

© Dong Liu 2003

UNIVERSITY OF CALGARY
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled "Automating Transition from Use Cases to Class Model" submitted by Dong Liu in partial fulfilment of the requirements of the degree of Master of Science.

Supervisor, Dr. B. H. Far,
Department of Electrical and Computer Engineering

Co-Supervisor, Dr. A. Eberlein,
Department of Electrical and Computer Engineering

Dr. Y. Wang,
Department of Electrical and Computer Engineering

Dr. F. Samavati,
Department of Computer Science

Date July 28, 2003
Abstract

Object-Oriented Analysis and Design (OOAD) is currently the most popular software development paradigm. Object identification and class modeling from requirements are among the central activities in object-oriented development. In this thesis, we devise a method that converts the functional requirements into a class model semi-automatically. The functional requirements are specified and represented by use cases. Use-case language schemas are proposed to reduce the complexity, vagueness and ambiguity of natural language. A use-case processing method is applied to analyze the use-case diagrams and the use-case specifications accompanied with the diagrams. The robustness diagrams and the collaboration diagrams are generated as the analysis artefacts, and the class model is generated as the design artifact. A CASE tool named UCDA (Use-Case driven Development Assistant) is implemented to support the methodology. UCDA can automatically generate the diagrams in Rational Rose.
Acknowledgements

First and foremost I wish to thank my supervisor, Dr. Behrouz Homayoun Far, for his clear instruction and encouragement during this project. His perspicacity for research is marvellous and I got lots of ideas from him for the project.

I am grateful to my co-supervisor Dr. Armin Eberlein who gave me precious advices on my research and publications.

Dr. Yingxu Wang supplied some lab equipments to me and I thank him for his help. Dr. Faramarz Samavati gave me comments on this thesis, and I thank him for valuable feedback.

This work was carried out in the Department of Electrical and Computer Engineering, University of Calgary, and I wish to thank my colleagues there for creating such a pleasant and stimulating environment for conducting research work.

Thanks to Mr. Cyprian Ngolah for his comments on the writing of this thesis. Thanks to my friends for either giving me suggestions or other helps during the work.

Finally, thanks to my parents and my girlfriend, for everything.
# Table of Contents

Approval Page .............................................................................................................. ii  
Abstract ......................................................................................................................... iii  
Acknowledgements ........................................................................................................ iv  
Table of Contents .......................................................................................................... v  
List of Tables .................................................................................................................. vii  
List of Figures and Illustrations ..................................................................................... viii  
List of Symbols, Abbreviations and Nomenclature ....................................................... ix  

Chapter One: **INTRODUCTION** ............................................................................. 1  
1.1 Review of Object-Oriented Analysis and Design ................................................. 1  
1.1.1 A brief history and current status ..................................................................... 1  
1.1.2 OOAD concepts ............................................................................................... 2  
1.1.3 RUP approach ................................................................................................. 3  
1.2 Object Model Creation Process ........................................................................... 4  
1.3 The Project and Achievements ............................................................................ 6  
1.3.1 Motivation and state of the art ....................................................................... 6  
1.3.2 The problem .................................................................................................... 8  
1.3.3 The project ...................................................................................................... 8  
1.3.4 The achievements ......................................................................................... 10  
1.3.5 Thesis contributions ....................................................................................... 10  
1.4 Structure of the Thesis ......................................................................................... 10  

Chapter Two: **USE CASE DEVELOPMENT** .......................................................... 13  
2.1 Survey of Use Cases ......................................................................................... 13  
2.1.1 Scenario and use case .................................................................................... 13  
2.1.2 Use cases in UML ......................................................................................... 15  
2.1.3 Use-case specifications ................................................................................. 16  
2.1.4 Use cases for OO project .............................................................................. 18  
2.2 Use-case Language Schemas ............................................................................ 19  
2.2.1 English sentences .......................................................................................... 21  
2.2.2 Statement structures ...................................................................................... 25  
2.3 Normalization of Use-case Specifications ......................................................... 27  
2.3.1 Glossary ........................................................................................................ 28  
2.3.2 Syntactic reconstruction ............................................................................... 29  
2.3.3 Writing reasonable events ............................................................................ 31  
2.3.4 Applying language schemas and the template ............................................. 32  
2.4 Summary ............................................................................................................. 33  

Chapter Three: **USE-CASE DRIVEN ANALYSIS AND DESIGN** ....................... 33  
3.1 Artefacts in Analysis and Design ...................................................................... 33  
3.1.1 Analysis model ............................................................................................. 34  
3.1.2 Design model ............................................................................................... 37  
3.2 Use-case Processing Method ............................................................................. 38  
3.2.1 Object elicitation ......................................................................................... 39
List of Tables

Table 2.1  The RUP use-case specification template .................................................. 16
Table 4.1  The behavior types and the associations ....................................................... 48
Table 4.2  The rules for object identification based on statement patterns ..................... 50
Table 4.3  The rules for message identification based on statement patterns ......... 54
Table 4.4  The robustness diagram validation rules ......................................................... 58
List of Figures and Illustrations

Figure 1.1 Object Model Creation Process (OMCP) .......................................................... 5
Figure 1.2 Object Model Creation Process (OMCP) in a RUP’s sense ............................. 6
Figure 1.3 The structure of our project ........................................................................... 9
Figure 2.1 Notations of use case in UML by Rose .......................................................... 15
Figure 2.2 The structure of noun phrase ......................................................................... 23
Figure 2.3 The structure of verb group ............................................................................. 23
Figure 2.4 The structure of PP ......................................................................................... 24
Figure 2.5 The structure of statement ................................................................................ 24
Figure 2.6 The statement Patterns .................................................................................... 28
Figure 2.7 An actor-system interaction model .................................................................. 31
Figure 3.1 The map of artefacts ......................................................................................... 33
Figure 3.2 The ICONIX Process ....................................................................................... 35
Figure 3.3 Robustness diagram notations ......................................................................... 36
Figure 3.4 The updated map of artefacts ........................................................................... 37
Figure 3.5 The final map of artefacts ................................................................................ 38
Figure 3.6 Messages and classes’ responsibilities ............................................................... 43
Figure 3.7 Include and aggregation ................................................................................... 44
Figure 3.8 Use-case generalization and class generalization ............................................ 45
Figure 4.1 The behavior types in actor-system interactions ............................................. 47
Figure 4.2 The robustness diagram of use case “Withdraw Cash” .................................. 60
Figure 4.3 The new robustness diagram of use case “Withdraw Cash” ........................... 61
Figure 4.4 The collaboration diagram of use case “Withdraw Cash” ............................... 62
Figure 4.5 The classes identified from use case “Withdraw Cash” .................................. 62
Figure 5.1 The use-case diagram of UCDA ...................................................................... 65
Figure 5.2 The activity diagram of use case “Realize use cases” .................................... 66
Figure 5.3 The activity diagram of use case “Validate analysis model” ........................... 67
Figure 5.4 The activity diagram of use case “Generate class model” .............................. 67
Figure 5.5 A multiagent architecture for UCDA ............................................................... 69
Figure 5.6 The four layers of a system under design ......................................................... 69
Figure 5.7 Working with DOM ......................................................................................... 74
Figure 5.8 Rose application and extensibility components .............................................. 78
Figure 5.9 A dialog window .............................................................................................. 81
Figure 5.10 Activation of the add-in in Add-In Manager .................................................. 83
Figure 5.11 The extended menu for the add-in ................................................................. 84
Figure 5.12 The updated four-layer structure with glue layers ....................................... 85
Figure 6.1 The updated methodology structure ............................................................... 88
# List of Symbols, Abbreviations and Nomenclature

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adj</td>
<td>Adjective</td>
</tr>
<tr>
<td>AP</td>
<td>Adjective Phrase</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>ATM</td>
<td>Automated Teller Machine</td>
</tr>
<tr>
<td>AUX</td>
<td>Auxiliary verb</td>
</tr>
<tr>
<td>CASE</td>
<td>Computer Aided Software Engineering</td>
</tr>
<tr>
<td>CD</td>
<td>Confidence Degree</td>
</tr>
<tr>
<td>COM</td>
<td>Component Object Model</td>
</tr>
<tr>
<td>DET</td>
<td>Determiner</td>
</tr>
<tr>
<td>DM</td>
<td>Data Management</td>
</tr>
<tr>
<td>DOM</td>
<td>Document Object Model</td>
</tr>
<tr>
<td>DTD</td>
<td>Document Type Definition</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>INT</td>
<td>Intensifier</td>
</tr>
<tr>
<td>MSXML</td>
<td>Microsoft XML core services</td>
</tr>
<tr>
<td>NL</td>
<td>Natural Language</td>
</tr>
<tr>
<td>NP</td>
<td>Noun Phrase</td>
</tr>
<tr>
<td>OLE</td>
<td>Object Linking and Embedding</td>
</tr>
<tr>
<td>OMCP</td>
<td>Object Model Creation Process</td>
</tr>
<tr>
<td>OMT</td>
<td>Object Modeling Technique</td>
</tr>
<tr>
<td>OO</td>
<td>Object Oriented</td>
</tr>
<tr>
<td>OOAD</td>
<td>Object Oriented Analysis and Design</td>
</tr>
<tr>
<td>OOD</td>
<td>Object Oriented Design</td>
</tr>
<tr>
<td>OOP</td>
<td>Object Oriented Programming</td>
</tr>
<tr>
<td>PD</td>
<td>Problem Domain</td>
</tr>
<tr>
<td>PP</td>
<td>Prepositional Phrase</td>
</tr>
<tr>
<td>REI</td>
<td>Rational Extensibility Interface</td>
</tr>
<tr>
<td>RUP</td>
<td>Rational Unified Process</td>
</tr>
<tr>
<td>SAX</td>
<td>Simple API for XML</td>
</tr>
<tr>
<td>SDML</td>
<td>Standard Generalized Markup Language</td>
</tr>
<tr>
<td>SI</td>
<td>System Interface</td>
</tr>
<tr>
<td>SS</td>
<td>Statement Structure</td>
</tr>
<tr>
<td>SVG</td>
<td>Scalable Vector Graphics</td>
</tr>
<tr>
<td>UCDA</td>
<td>Use-case driven Development Assistant</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling language</td>
</tr>
<tr>
<td>VBA</td>
<td>Visual Basic for Application</td>
</tr>
<tr>
<td>Vgp</td>
<td>Verb group</td>
</tr>
<tr>
<td>VP</td>
<td>Verb Phrase</td>
</tr>
<tr>
<td>W3C</td>
<td>Worldwide Web Consortium</td>
</tr>
<tr>
<td>XMI</td>
<td>XML Metadata Interchange</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>XP</td>
<td>Extreme Programming</td>
</tr>
<tr>
<td>Xpath</td>
<td>XML path language</td>
</tr>
<tr>
<td>XSD</td>
<td>XML schema definition language</td>
</tr>
<tr>
<td>XSLT</td>
<td>Extensible Stylesheet Language Transform</td>
</tr>
</tbody>
</table>
Chapter One: INTRODUCTION

This chapter reviews the history and basic concepts of Object-Oriented Analysis and Design (OOAD). The Object Model Creation Process (OMCP) and an improved model based on OMCP are introduced. The motivation, the structure and the achievements of the project are presented. Related research activities are reviewed. The chapter ends with the structure of the thesis.

1.1 Review of Object-Oriented Analysis and Design

1.1.1 A brief history and current status

The first Object-Oriented Programming (OOP) language, known as Simula-67, was developed in 1967 [1]. OOP became a popular programming paradigm since 1990’s. With the evolution of programming languages, the methodologies used in system analysis and design have been evolving to meet the requests from software projects in the industry. Object-Oriented Analysis and Development (OOAD) has become a very popular software development approach since the 90’s of the last century. Over 60 different Object-Oriented (OO) methods have been proposed [2]; most of them just appeared and disappeared, but some diagrammatic methods have been developed and unified. Booch, Rumbaugh and Jacobson joined together in Rational Software Cooperation (www.rational.com), which was acquired by IBM for $2.1 billion in December 2002. In November 1997, the Unified Modeling Language (UML) was adopted as a standard by OMG (Object Management Group, www.omg.org) with the efforts of Rational [3], which is a milestone in the history of OOAD world. Just like the
methodologies, many CASE (Computer Aided Software Engineering) tools have been
developed. According to Object by Design (www.objectbydesign.com), there are
currently 104 kinds of CASE products supporting UML [4]. Besides UML and its CASE
tool Rational Rose, Rational Cooperation has promoted the Rational Unified Process
(RUP) as both a software engineering process and a process product [5]. RUP provides a
disciplined approach to assign tasks and responsibilities within a development
organization. One significant characteristic of RUP is that it is a use-case driven process,
which is a development of Jacobson’s Objectory process [6]. The approach introduced in
this thesis is based on RUP.

1.1.2 OOAD concepts
An “object” is one of the most essential concepts in OOAD. An object refers to a discrete
entity with a well-defined boundary and identity that encapsulates state and behavior [1].
Each object has its own unique identity, and may be referenced by a unique handle that
identifies it and is used to access it. An object is an instance of a “class”. A class is the
descriptor for a set of objects that share the same attributes, operations, methods,
relationships, and behavior. An attribute is the description of a named information slot of
a specified type in a class. An operation is a specification of a transformation or query
that an object may be called to execute. Each operation has a name and a list of
parameters. A method is a procedure that implements an operation. Behavior describes
the observable effects of an operation or event, including its results. The common
relationships among classes include associations and generalizations. An association is
the semantic relationship between two or more classes that involve connections among
their instances. A generalization is a taxonomic relationship between a more general element and a more specific element.

In OOP, classes are defined first, and then objects can be instantiated from the corresponding classes. When we model the system using OOAD, the process is opposite: the objects are identified first, and then the classes are abstracted from the objects and their attributes, operations and relationships are defined. In Object-Oriented Analysis (OOA), the analysis model is created based on the requirements model. The requirements model describes the boundary of the system and its functionality. The analysis model identifies the analysis classes and describes how they interact to perform specific tasks. In Object-Oriented Design (OOD), the responsibilities of the classes and the relationships between the classes are identified.

1.1.3 RUP approach

The Rational Unified Process provides guidelines, templates and examples for most aspects and stages of iterative software development. RUP captures many of the best practices in modern software development in a form that is suitable for a wide range of projects and organizations. Particularly, RUP emphasizes six practices: develop software iteratively; manage requirements; use component-based architectures; model software visually; verify software quality continuously; and control changes to software. In addition to these practices, there are other key features introduced by RUP: use-case driven development and process configuration and tool support.

RUP applies use cases as the main method to capture and manage requirements. Rational supplies Rational RequisitePro to support requirements capture and
management, *Rational Rose* to visualize use-case diagram in UML and *Rational SoDA* to generate documents. The activity diagram, the collaboration/sequence diagram and the class diagram are used in analysis and design. *Rational Rose* supports the visualization of all these artefacts. RUP is a process framework that can be adapted and extended to suit certain needs. Our approach can be regarded as a tailored RUP.

1.2 Object Model Creation Process

Object elicitation and class modeling are among the central activities in OOAD. However, there are no theoretically or pragmatically well-developed methodologies to help the software developers tackle this problem successfully. In [7], Booch mentions that several developers were asked what were the rules that they applied to identify objects and classes. Stroustrup, the designer of C++, responded “It’s a Holy Grail. There is no panacea.” Gabriel, one of the designers of CLOS (Common Lisp Object System), stated, “That’s a fundamental question for which there is no easy answer. I try things.” Obviously, this task has puzzled the developers since OO was first introduced.

The Object Model Creation Process (OMCP) is a widely applied method to elicit objects and build a class model from requirements [7] and a few systems that implement OMCP are already available [8]. An overview of OMCP is shown in Figure 1.1. The requirements are acquired from the stakeholder through interviews or other methods. Then the developer analyzes the requirements and creates the analysis model. The classes are designed and the analysis model is transformed to the design model at the end of OMCP. When the design model is ready, the implementation model can be developed.
The developers' decisions are supported by domain knowledge, and public and private experiences.

![Diagram of Object Model Creation Process (OMCP)](image)

Figure 1.1 Object Model Creation Process (OMCP).

We apply RUP's features in OMCP and propose a new model of Object Model Creation Process shown in Figure 1.2. The requirements are specified as use cases. The activities comprise finding actors and use cases from the original requirements, structuring the use cases into a use-case model by the system analyst, and documenting the specifications for all the use cases by the requirements specifier. With the change of requirements format, the activity of analysis is specified as use-case analysis carried out by the designer. The activities of design comprise the class design by the designer and the architecture analysis by the architect. The process is iterative, which means some artefacts may be evaluated at the checkpoints, and the activities will be repeated if its output artefacts are not satisfying. This project is to refine the artefacts and the activities in the process and implement a CASE tool to support the methodology and automate most of the activities.
1.3 The Project and Achievements

1.3.1 Motivation and state of the art

As is mentioned in Section 1.2, object elicitation and class modeling are among the central activities in object-oriented software development and have puzzled many developers. The methods used to identify the objects and generate the class model are affected by the way in which the requirements have been represented. Generally, there are two types of requirements: the requirements specified in formal languages and those in natural language (NL). The former one is studied mainly in the research community.
[9-11], while the latter are widely used in industry. We have developed a methodology and a CASE tool to address OOAD based on requirements in NL (English).

Researchers have studied object-oriented design based on natural language requirements since object-oriented programming (OOP) became popular in software development. Abbott proposed an approach to Ada program design on linguistic analysis of informal strategies [12]. Booch’s OOD methodology was developed based on this approach [7]. Abbott’s heuristics was complemented by Saeki, Horai and Enomoto [13], who emphasized the system’s behavior expressed by the verbs in the natural language description for software requirements. Boyd developed the investigation by Cockburn on the application of linguistic metaphors to OO design, and presented an approach on syntactic normalization [14]. Besides the methods, some tools have been developed to support OOAD based on natural language requirements. CoGenTex Inc (www.cogentex.com) developed a methodology and the corresponding prototype tool named LIDA (Linguistic assistant for Domain Analysis), which provides linguistic assistance in the model development process [15]. The tool can process textual documents and help the user to generate the class model visualized in UML (Unified Modeling Language). NIBA (Natural Language Requirements Analysis in German) is an interdisciplinary project between computer scientists and computer linguists at the University of Klagenfurt [16]. The tool can parse the requirements document in German, interpret and transform the output of the parser, validate the result and finally generate the concept model in UML. A common weak spot of many methodologies is that the art of OOAD is not well combined with the techniques for natural language processing. We
developed a methodology based on the good practices in object-oriented software development.

1.3.2 The problem

How to analyze the NL functional requirements and generate the class model are still of great concern in OOAD. Even experienced analysts and developers perceive it as a hard task. Complexity, vagueness and ambiguity of natural language, lack of the domain knowledge and OO experiences and absence of effective OO methods give rise to the difficulties. More research needs to be done in the area of methodologies and CASE tools.

1.3.3 The project

This thesis describes the development work of an intelligent system for requirements elicitation and object-oriented analysis and design. The system captures requirements, specifies the functional requirements in use-cases diagrams and use-case specifications, finds the objects inside the use-case specifications, generates the analysis models and the design models and visualize them in UML. The following points are highlighted by our methodology.

- Apply a use-case driven process.
- Emphasize the behavior when analyzing the NL requirements.
- Model the artefacts in UML.

The structure of this system is shown in Figure 1.3. The system is divided into two parts: The first part supports the requirements elicitation and formalization, and the
second part supports use-case analysis and class model generation. This thesis is mainly about the second part.

Far and Wahono have worked on a project named *OOExpert* for object elicitation and class modeling [8, 17]. An agent-based system is designed and implemented in their project. A formal method named *Object-Based Formal Specification* (OBFS) is proposed and used to formalize the requirements specifications. There are six kinds of agents in the system: requirements acquisition agent, object identification agent, attribute identification agent, association identification agent, behavior identification agent, and object refinement agent.

The differences between *OOExpert* and this project are: (1) *OOExpert* applied a formal method to specify the requirements, but the requirements in this project are use-
case specifications in natural language; (2) OOExpert was based on Booch's OOD (Object-Oriented Design) methodology, but we apply UML and RUP in this project.

1.3.4 The achievements

We propose a methodology based on OMCP and RUP. The methodology combines the techniques for natural language (NL) requirements analysis and good practices of OOAD together. Use-case language schemas are proposed and used to normalize the use-case specifications. A use-case processing method is introduced for use-case realization. The association relationships between formalized action statements and the behavior types are identified and applied to automate the use-case analysis.

1.3.5 Thesis contributions

An experimental CASE tool named UCDA is developed to support the methodology. UCDA can process the use-case specifications and generate robustness diagrams, collaboration diagrams and class diagrams automatically. All the diagrams can be visualized in the workspace of Rational Rose. UCDA can find and trace some missing objects or associations between objects in requirements during the analysis. UCDA is developed as an add-in of Rational Rose, and can be installed and run on Windows systems with Rational Rose installed. Three papers about this project have been published and one has been submitted.

1.4 Structure of the Thesis

The thesis consists of 6 chapters. They are introduced briefly as follows.
Chapter One: Introduction  The basic concepts of Object-Oriented Analysis and Design are reviewed in this chapter. The Object Model Creation Process and its evolution are presented. The motivation, the structure and the achievements of this project are introduced. The structure of the thesis is also introduced.

Chapter Two: Use Case Development  The concepts related to use cases are introduced briefly in this chapter. To normalize the natural language expression in use cases, the use-case language schemas are proposed and discussed. A case study is given at the end of the chapter to show how to rewrite a casually written use-case specification to an organized and well-normalized one.

Chapter Three: Use-Case Driven Analysis and Design  This chapter discusses the artefacts and activities in use-case driven analysis and design. The methodology is some kind of tailored RUP. Robustness analysis and robustness diagram are introduced to the use-case realization. A use-case processing method is introduced to drive the transition from use cases to the class model.

Chapter Four: Automating the Use-case Driven Development  This chapter discusses the approach that automates the use-case processing. A set of rules to identify the objects and their stereotypes are presented. The rules for message identification are also presented. A method is proposed to validate the robustness diagrams. Some missing objects or associations between objects can be found and traced through the validation.

Chapter Five: Implementation  The techniques used to implement the CASE tool are introduced. We use XML to represent all the artefacts and the XML schema to enforce the structure and elements of XML documents. The
manipulation of XML is performed in DOM (Document Object Model) with MSXML (Microsoft XML Core Services). The Rose Extensibility Interface (REI) enables the tool to generate the artefacts and visualize them in *Rational Rose*. The CASE tool is deployed as an add-in of *Rational Rose*.

**Chapter Six: Conclusion**  
The evaluation, contributions and potential future extensions of the project are discussed.
Chapter Two: USE CASE DEVELOPMENT

In this chapter, the basic aspects of use cases are discussed. We propose use-case language schemas to normalize the use-case writing in natural language. The use-case language schemas are developed based on natural language statement structures and special needs for use-case representation. We summarize and list the methods to normalize use-case specifications. An example use-case specification is normalized using the methods.

2.1 Survey of Use Cases

Use cases have been used in software engineering community for over 10 years. With the efforts of the developers and researchers [6, 18-20], use cases have become quite mature.

2.1.1 Scenario and use case

The earliest term related to scenario and use case is usage scenario introduced by Jacobson in 1967 [21]. The usage of scenarios was first introduced in Rumbaugh’s OMT (Object Modeling Technique) methodology in 1990 [22]. Use cases were first made public by Jacobson in 1987 [23], and was discussed in detail in the Objectory methodology [6]. At first, the concepts of both scenarios and use cases were not clearly defined, and there was a cloud of confusion when they were introduced to the software engineering industry and research communities. Some thought that they are the same concepts when describing the system’s behavior [24]. This situation lasted for a long time. Even in 1995, Alistair Cockburn tried to clarify the concepts [25, 26]. Cockburn’s definitions for them are:
**Scenario:** A sequence of interactions happening under certain conditions, to achieve the primary actor's goal, and having a particular result with respect to that goal. The interactions start from the triggering action and continue until the goal is delivered or abandoned, and the system completes whatever responsibilities it has with respect to the interaction. A sequence has no branching.

**Use Case:** A collection of possible scenarios between the system under discussion and external actors, characterized by the goal the primary actor has toward the system's declared responsibilities, showing how the primary actor's goal might be delivered or might fail.

In UML, these terms are defined as [1]:

**Scenario:** A sequence of actions that illustrates behavior. A scenario may be used to illustrate an interaction or the execution of a use-case instance.

**Use Case:** The specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or class can perform by interacting with outside actors.

The definitions in UML are almost the same as Cockburn's. In this thesis, the meanings of the terms are the same as these definitions. The following is an example of use cases that describes that a customer performs a session when using an ATM (Automated Teller Machine) system. It was written by Bjork [30].

*A session is started when a customer inserts an ATM card into the card reader slot of the machine. The ATM pulls the card into the machine and reads it. (If the reader cannot read the card due to improper insertion or a damaged stripe, the card is ejected, an error screen is displayed, and the session is aborted.) The customer is asked to enter his/her PIN, and is then allowed to perform one or more transactions, choosing from a menu of possible types*
of transaction in each case. After each transaction, the customer is asked whether he/she would like to perform another. When the customer is through performing transactions, the card is ejected from the machine and the session ends. If a transaction is aborted due to too many invalid PIN entries, the session is also aborted, with the card being retained in the machine.

The customer may abort the session by pressing the Cancel key when entering a PIN or choosing a transaction type.

2.1.2 Use cases in UML

UML specifies the use-case semantics, notation, diagram and relationships [1, 3]. In UML a use case is shown as an ellipse with its name. A use-case diagram shows the relationships among actors and use cases. There may be associations between use cases and actors. Use cases may be related to other use cases by extend, include or generalization relationships. Because a use case is a kind of classifier in UML, it inherits the generalization relationship from the classifier. A use-case generalization is a taxonomic relationship between a use case (the child) and the use case (the parent) that describes the characteristics the child shares with other use cases that have the same parent. Include and extend are two special relationships in UML to describe the dependency relationship between two use case. An include relationship means that a use case includes the behavior described in another use case, while an extend relationship implies that a use case may extend the behavior described

Figure 2.1 Notations of use case in UML by Rose.
in another use case, ruled by a condition. Include relationship in UML is replaced by use relationship in some cases [18]. The marker that references a location or set of locations within the behavioral sequence for a use case is called extension points, at which additional behavior can be inserted. Figure 2.1 shows all the notations introduced above.

2.1.3 Use-case specifications

Only use-case diagrams in UML cannot fully describe a system’s behavior. Each use case may be accompanied by a use-case specification to specify the detailed information. There have been lots of arguments about whether it is necessary to and how to normalize the use-case specification writing in the industry and research communities [21]. Some prefer writing in no format; however, we suggest the use of well-formatted use-case specification in order to make the use cases easy to write, read and manage. The specification should comprise all the proper information of the use case and be easy to handle. Cockburn summarized the formats had been used. The complexities of the formats are different. The use-case specifiers can choose the proper one for their projects. RUP supplies a use-case specification template within Rational RequisitePro. The information entries are shown in Table 2.1. The template can be

| Table 2.1 The RUP use case specification template. |
| 1. Use Case Name |
| 1.1 Brief Description |
| 2. Flow of Events |
| 2.1 Basic Flow |
| 2.2 Alternative Flows |
| 2.2.1 < First Alternative Flow > |
| ... |
| 3. Special Requirements |
| 3.1 < First special requirement > |
| ... |
| 4. Preconditions |
| 4.1 < Precondition One > |
| ... |
| 5. Postconditions |
| 5.1 < Postcondition One > |
| ... |
| 6. Extension Points |
| 6.1 <name of extension point> |
| ... |
found in Appendix A, and it is applied with some modifications in our project.

The basic flow is the main success flow of events of a use case. Each use case must have one and only one basic flow, though a use case may have several success scenarios. If the condition of an event or step in the basic flow cannot be fulfilled, the basic flow is interrupted, and the alternative flow happens. If the situation turns complex, a use case may have several alternative flows, and an alternative flow may have several sub alternative flows. The writing of use-case specification is a dynamic process. With the progress of the project, the use-case specification may be modified for several times. The alternative flows of a use case will get longer and longer. That introduces difficulty to manage the use cases. A solution is to trim the long alternative flow off the original use case and create a new use case. The new use case is the extension of the original one. If the alternative flow is included in more than one use case, it is better to create a new use case for the flow, so that the flow can be reused.

A precondition is the state of system that must be present prior to a use case being performed. All the use cases must have some preconditions; however the preconditions in the use-case specification can be empty if they are thought as the default conditions of the system. After the preconditions are validated at the start of a use case, they will not be validated again during the performance of the use case. A postcondition is a possible state the system could be right after the use case is executed. A postcondition should not be present before the use case is performed. Postconditions include not only the cases for basic flow but also those for alternative flows. A trigger is the event that makes the use case started. The first step of basic flow acts as the trigger in many cases. Like other
events in the basic flow or alternative flows, a trigger should be an action that is related with stakeholders’ interests and can be observed.

The most important part of a use-case specification is the flow of events including the basic flow and alternative flows. The element of flows is the event or step. Writing effective events and organizing them deserve most efforts in use-case specifications. We propose use-case language schemas to normalize the use-case writing.

2.1.4 Use cases for OO project

The requirements analysts can apply use cases to organize the whole project. They can plan and manage the project by actors or goal levels. Use cases as the functional requirements of the system can be traced across the releases. The changed functionalities and the needed updates for next release should be described in all new versions of use-case specifications. Use cases are useful for test case preparation. Most of the use-case description can be mirrored directly to test cases. This can ensure that the functionalities accomplished are just those wanted by stakeholders. When writing the help documents of the software to release, the technical writers will find that use cases are just what should be written in how-toes. Some researchers think use cases are useful for User Interface (UI) design. They put lots of UI details in the use-case specifications. We suggest that the use cases should not contain UI details. If we divide the system under design into four layers: user interface layer, problem domain layer, data management layer and system interface layer. The most unstable layer is the user interface layer due to the requirements changes. Use cases should focus on the problem domain. Too much information about UI makes the use cases hard to realize and manage.
Designers, especially OO designers, will face some hazards when their design is based on use cases. OO designers need to find the objects from the requirements and then model the system in classes. Use cases record behavioral requirements, and are functional decomposition of the system. Some inexperienced designers create the design classes that mirror the functional decomposition of software requirements. That is a serious mistake for object-oriented analysis and design. Meyer thinks that use cases are not a good tool to find classes. He mentions three risks to apply use cases in OOAD [24]: ordering emphasized by use cases is incompatible with object technology; the use cases are created based on operation modes that existed already, which is not good for a new system; use cases suggest decomposition on processes, while OO needs decomposition on object abstractions. We think that creating use cases based on existing system does not matter, because it is a kind of requirements reuse. If proper methods are applied, the designers can avoid such hazards successfully.

2.2 Use-case Language Schemas

One of our project’s objectives is to develop a methodology to automate use-case processing. If use-case specifications are casually written in natural language, it will be hard to analyze the use cases automatically. The use-case language schemas are proposed to normalize the use-case specifications.

On the one hand, the use-case statements should be normalized enough to enable automatic processing. On the other hand, the schemas should not introduce too many constraints on natural language writing and reading. We propose four schemas for use-case writing are as follows.
Basic Statement Schema  This schema applies to all the events. It is a set of statement structures that will be discussed shortly. Which statement structure should be applied in an event statement is determined by the context.

If-Then Schema  This schema is used for the conditional event flow. Normally it happens at the beginning of alternative flows. If-clause may consist of one or more condition statements, each of which can be described using the basic statement schema. Then-clause contains a flow of events.

Do-Until Schema  This schema describes the repeated event or sequence of events under a certain condition. Do-clause may contain one event or a sequence of events. Until-clause may consist of one or more condition statements, on which the iteration will stop. While-Do structure can also be used in place of Do-Until. We adopt Do-Until because the condition is tested after the repeated actions in most cases. Some quantifier information can be described in until-clause.

Con-Noc Schema  This schema describes the performance of two or more activities during the same time interval, i.e., the concurrency. This schema starts with a Con and ends with a Noc. There are two or more concurrent events or sequence of event in each Co# clause like following:

\[
\text{Con} \\
\text{Co1} \\
\text{Co2} \\
\text{Noc}
\]

The basic statement schema is the core of these schemas. The If-Then schema and the Do-Until schema describe the conditional event flow and recursive event(s), but each
event is represented by a statement that should follow the basic statement schema. The following sections discuss the basic statement schema linguistically.

2.2.1 English sentences

Syntax is traditionally the name given to the study of the form, positioning and grouping of the elements that make up sentences [27]. It is about the structure of sentences. The structure means

(a) a sentence can be divided into parts, a.k.a. constituents;
(b) there are different kinds of parts, a.k.a. categories of constituents;
(c) each constituent has a certain specifiable function in the structure.

2.2.1.1 Constituents

Sequences of words that can function as constituents in the structure of sentences are called phrases. A phrase is a syntactic structure that consists of more than one word but lacks the subject-predicate organization of a clause. To identify the phases in sentences, the following rules can be followed. To show how the rules work, we use the following sentence to demonstrate.

*The user enters the user name and the password on the log-in page.*

(a) If a sequence of words can be omitted from a sentence leaving another good sentence, this is a good indication that the sequence is a phrase functioning as a constituent in the structure of the sentence. The demo sentence, the sequence on the log-in page can be omitted, and the sentence turns to be

*The user enters the user name and the password.*

It is still a good sentence in the sense of syntax. Not all the phrases are omissible.
(b) If a sequence of words can be replaced with a single word without changing the structure of the sentence, then the sequence of words functions as a constituent of the sentence and is a phrase. The sequence of words *the user* can be replaced by a single word *he* in the sentence and the sentence turns to be

*He enters the user name and the password on the log-in page.*

(c) If a sequence of words can be the answer to ‘WH’ questions, then it is a phrase. ‘WH’ questions refer to the question sentence beginning with *who, which, what, why, where, when, whose* and *how*. For the sample sentence, we can ask

*Question*: What does the user enter on the log-in page?

*Answer*: The user name and the password.

(d) If a sequence of words can be moved to another place of the sentence without destroying the sentence, then it is a phrase. The sample sentence can be changed to the following with movement of a sequence.

*On the log-in page* *the user enters the user name and the password.*

The phrases can usually be identified automatically.

2.2.1.2 Categories

Generally, one constituent may be classified as different categories according to various criteria. A common taxonomy for language constituents is based on their functions. In this section, we introduce some notations and terms related with categories.

**Nouns and noun phrases** A noun phrase (denoted as NP) is a phrase that contains, and is centred on a noun. There is a group of words called determiners (denoted as DET), which are the markers of nouns or noun phrases. Frequently used words like *the, these,*
her, a, few, one, each are all determiners. There is only one noun in a noun phrase that can function as the head. Besides determiners, adjectives/ adjective phrases and prepositional phrases can be part of NPs. Recursion is common in natural language structure. A noun phrase can contain one or several noun phrases inside. The structure of a noun phrase can be shown as Figure 2.2. NOM represents nominal, AP represents adjective phrase, and PP represents propositional phrase.

**Verbs and verb phrases** Verb phrase (denoted as VP) is nearly the most important category among all the categories. A verb phrase may comprise another verb phrase and a prepositional phrase. Each verb phrase must have a verb group (denoted as Vgp) inside. The verb group consists of a lexical verb that is optional preceded and modified by other auxiliary verbs (denoted as AUX). There is a special kind of verb group called phrasal verb. There are many phrasal verbs in English, such as turn on, give up, put down. A phrasal verb consists of a verb and another particle. The structure of verb group is shown in figure 2.3. A verb group can be followed by other constitutes named complements. The type of complements determines the structure of verb phrase, and further determines the structure of the sentence.
**Adjectives, adjective phrases**

Adjectives and adjective phrases function as the modifier of nouns or noun phrases. An adjective phrase may consist of an intensifier (denoted as INT) and an adjective (denoted as Adj). The *very* in a phrase *very nice* is an intensifier.

**Prepositions and prepositional phrases**

Prepositional phrase (denoted as PP) normally consists of a preposition and a noun phrase. The structure is shown in Figure 2.4.

**Co-ordinate phrases**

Co-ordination is common in natural languages. Nouns, pronouns, adjectives, adverbs, prepositions can form co-ordinate phrases, e.g. *teacher and student, hot and tasty, up and down*.

### 2.2.1.3 Functions

Most of the statements can be divided into two parts. The first part is traditionally said to function as a *subject*; and the second as a *predicate*. Some sentences can be divided into three parts: a subject, a predicate and an object. We adopt the structure of subject + predicate in the analysis, because this structure can cover more situations. Normally, the subject is a noun phrase and the predicate is a verb phrase. The structure of a statement is shown in Figure 2.5. The predicates can be various because of the variety of verb phrases. The structure of a sentence is determined by the structure of the predicate, i.e. the structure of the verb phrase functioning as a predicate.

![Figure 2.4](image1.png)  
Figure 2.4 The structure of PP.

![Figure 2.5](image2.png)  
Figure 2.5 The structure of statement.
2.2.2 Statement structures

There are three kinds of sentences in English [29]. The different effects of the sentences on the listener help to distinguish their types. Some sentences tend to make people do things. They are called request sentences, e.g. *Hurry up*. Some sentences tend to make people say things in answer to them. They are called question sentences, e.g. *What's the time?* Some sentences tend to state things to the listener. They are called statement sentences, e.g. *The user enters the use name and the password on the log-in page.* Statement sentences are most frequently used among the three kinds in English writing. Nearly all the sentences in the use-case specification are statements. Although the statement sentences look various, there are still some regular structures.

The structures of all the statements' subjects are almost the same. The predicates have seven kinds of structures that are transitive (a.k.a. monotransitive), intransitive, ditransitive, intensive, complex transitive, prepositional and non-finite [29].

a. Transitive

The verb group (Vgp) in the predicate requires a single noun phrase (NP) to complement it in a transitive structure. The noun phrase functions as the verb group's direct object. The following is an example sentence.

*The system starts a session.*

b. Intransitive

The verb group does not require any further constituent to complement it in an intransitive structure, e.g.

*The session ends.*
c. **Ditransitive**

The verb group requires two noun phrases as its complements in a ditransitive structure. One of the noun phrases functions as the direct object and the other as the indirect object. In the following sentence

*The system sends the network a message.*

*the network* is the indirect object and *a message* is the direct one. The positions of two NPs may be exchanged by introducing a preposition *to* or *for*.

*The system sends a message to the network.*

d. **Intensive**

The head verb in the verb group is a linking verb, and the complement can be a single adjective phrase, or noun phrase, or preposition phrase in this structure. The verbs such as *be, become, turn, get* and so on can function as linking verb. The following is an example.

*The card is not readable.*

e. **Complex transitive**

The verb group is implemented by an NP functioning as a direct object and an AP, or a PP, or another NP as a predicate in complex transitive structure. Only a few special transitive verbs can appear in this structure. In the following example sentence, *the item* is the direct object, and *in the shopping cart* is the predicate.

*The user puts the item in the shopping cart.*

f. **Prepositional**

There is a kind of verbs that must be complemented by a prepositional phrase, such as *look, refer, glance* and so on. This kind of verb groups is called prepositional Vgp, and
the corresponding structure is prepositional structure. The following is an example of this structure.

*The user looks at the monitor.*

g. **Non-finite**

Non-finite (infinite) verb sequences are categorized as a kind of clauses in the literatures [27, 28]. We assign non-infinite as one category of verb phrases to cover more cases of statements. In this structure, the verb group is complemented by an NP that is followed by a non-finite verb group. The non-finite verb groups can be classified into four types: the bare infinitive, the to– infinitive, the passive participle and the –ing infinitive. The bare infinitive can also be regarded as a to– infinitive with the *to* omitted. The following is an example.

*The system makes the card reader to eject the card.*

Figure 2.6 shows all the structures and their structures.

### 2.3 Normalization of Use-case Specifications

The practices for use-case specification normalization are discussed in this section. There are two usage scenarios when the use-case specifications are normalized. The first usage scenario is that the developers apply these practices to writing the use cases. The second is that the use-case specifications are already written and the developers rewrite them using the practices. The use case named “Perform Session” in Section 2.1.1 is normalized in this section as a case study.
2.3.1 Glossary

Different system analysts will use different terms when they prepare the requirements documents. The terms have diverse meanings depending on the domain and personal backgrounds. This makes the communication inside a development team difficult, and
leads to deviations or even mistakes along the development. In the community of Extreme Programming (XP), System Metaphor is one of the twelve practices of XP [31]. It has been revised to Common Vocabulary [32]. A shared vocabulary makes requirements unambiguous so that they can easily be understood by the developers responsible for system analysis and design. In RUP, the glossary is one of the major artefacts [5]. The analysts define a common vocabulary that can be used in all text descriptions of the system, especially in use-case specifications. RUP supplies a template of the glossary, which is in Appendix B.

We also apply the glossary in our approach to reduce the ambiguity of the use-case specifications. The terms and their descriptions are included in the glossary. During use-case writing/rewriting, the terms appearing in the specification are searched for in the glossary. If one entity is represented by different terms in the specification, these terms should be unified to the one in the glossary to keep the consistency. For example, in the specification of a use case named “Perform Session”, an ATM card and a card are the same entity, so ATM card is replaced by card. Both ATM and machine refer to the system under design, we will use system instead of the other terms.

2.3.2 Syntactic reconstruction

The variety of sentence structures gives rise to the complexity of natural language. Sentences are vague when they contain pronouns, or they miss subjects, objects, verb or other functional constitutes. The complexity and vagueness of English sentences can be reduced by syntactic reconstruction. Some software specialists also suggest the
requirements keep simple syntactic characteristics [19, 33]. The good practices related to the syntactic reconstruction are listed as follows.

- Basic statements rather than complex clauses;
- Concrete entity names rather than pronouns;
- Active voices rather than passive voices;
- No ellipses;

**Ellipses**  
omission of implied word: the omission of one or more words from a sentence, especially when what is omitted can be understood from the context.

- No gapping;

**Gapping**  
omission of lexical verb: the omission of lexical verb from the second of two coordinated structures.

- No inversion;

**Inversion**  
a.k.a. anastrophe, change in order: an alteration of the normal order of words or phrases in a grammatical construction, usually for rhetorical effect.

- Fewer auxiliary verbs for tense, aspect or modality.

- [there must be more items to list]  

We can find most sentences in the example need reconstruction. *A session is started when a customer inserts an ATM card into the card reader slot of the machine.* can be modified to *The customer insert a card into the card reader.* and *The system start a session.* *The customer is asked to enter his/her PIN.* can be modified to *The customer enter the PIN on the customer console.*
2.3.3 Writing reasonable events

The most difficult work during use-case writing is to write the actions of actors and the system. Sometimes the use-case specifiers write too many movements of actors or system in the flows. Some specifiers like to describe all the details occurring on the user interfaces. A rough guideline to write effective actions is to write use cases showing the actor’s intent and including a reasonable set of actions. A model for actor-system interaction has been developed to show the actions in an interaction [19]. The model is shown in Figure 2.7. The actor interacts with the system with a goal. First the actor sends a request to the system. The system validates the request, and then changes its states according to the business logic. Finally the system sends a response to the actor. A reasonable action should be the instance of a request, a validation, a change or a response. In some cases, a use case may not include all the four kinds of actions. Sometimes the system is not triggered by the actor’s request. Sometimes the system does not send any responses to the actor after it changes the states. From this model, we can find that the actions are conducted either by the actor or by the system. In use-case writing, the system is reserved as a special term for the system under design. We suggest the subjects of event statements be either an actor or the system.

Figure 2.7 An actor-system interaction model.
2.3.4 Applying language schemas and the template

The use-case language schemas and the use-case template can help normalize the use-case specification. When applying the language schemas, we found that several statement structures are frequently used, but others seldom appear. This is because of the nature of our language and the characteristics of use-case writing. The final version of example use-case specification is

Project Name: ATM

Use Case Name: Perform Session

Date:
Version:
Author:

Description: The costumer perform a session

Actors: customer

Flow of Events:

Basic Flow:

1. the customer inserts a card into the card reader ; //SS: transitive
2. the system starts a session ; //SS: transitive
3. the system pulls the card into the card reader ; //SS: transitive
4. the system reads the card by the card reader ; //SS: transitive
5. the customer enters the PIN on the customer console ; //SS: transitive
6. the system gets the PIN from the customer console ; //SS: transitive
7. the system validates the PIN ; //SS: transitive

Do  //Do-until
8. the system displays the transaction list on the customer console ; // SS: transitive
9. the customer selects one transaction type on the customer console ; //SS: transitive
10. the system performs transaction ; //SS: transitive

until  //Do-until
11. the customer selects abort on the customer console ; //SS: transitive
12. the system ejects the card out of the card reader; //SS: transitive
13. the customer gets the card from the card reader; //SS: transitive
14. the system prints the receipt in the receipt printer; //SS: transitive
15. the customer gets the receipt from the receipt printer; //SS: transitive
16. the system records the session information into the log; //SS: transitive
17. the session ends; //SS: intransitive

*Alternative Flow:*

If //If-then

the card is not readable, //intensive

then //If-then

i. the system ejects the card out of the card reader; //SS: transitive
ii. the system displays an error message on the customer console; //SS: transitive
iii. the customer gets the card from the card reader; //SS: transitive
iv. the session ends; //SS: intransitive

*Special Requirements:*

*PreConditions:*

the system starts. //SS: intransitive

*PostConditions:*

the system can supply service. //SS: transitive

*Extension Points:*

2.4 Summary

In this chapter, many aspects of use cases were surveyed. To normalize use-case specifications, we proposed four schemas of use-case language. Some good practices to normalize use-case specifications were listed in this chapter. The next chapter will discuss use-case processing and class modeling.
Chapter Three: USE-CASE DRIVEN ANALYSIS AND DESIGN

Analysis and design are the workflow steps right after requirements in the development process. The analysis model and the design model for a use-case driven process are specified. The activities to generate the artefacts are unified in a use-case processing method.

3.1 Artefacts in Analysis and Design

The use-case diagrams, use-case specifications, the analysis model and the design model are central artefacts for requirements, analysis and design in RUP [5]. Use cases have been discussed in Chapter 2. The analysis model is an object model describing the realization of use cases and serves as an abstraction of the design model. The design model is also an object model describing the realization of use cases and serves as an abstraction of the implementation model and its source code. The analysis model is an optional artifact according to the RUP, which means it can be tailored away from the process. However, we think the analysis model is necessary for use-case driven OOAD. A map of artefacts is shown in Figure 3.1.

Figure 3.1 The map of artefacts.
3.1.1 Analysis model

In RUP, the analysis model contains the results of use-case analysis, especially instances of the analysis classes. Analysis classes represent an early conceptual model of classes. In UML, analysis classes are represented as classes, and have three stereotypes: <<boundary>>, <<control>> and <<entity>> [3]. An analysis class has such properties as a name, a description, responsibilities and attributes. A collaboration realizes a use case in UML [1, 34]. It describes the context in which the implementation of a use case executes, i.e. the arrangement of objects and links that exist when the execution begins, and the instances that are created or destroyed during execution. The behavior sequences may be specified in interactions, shown as sequence diagrams or collaboration diagrams. A sequence diagram shows object interactions arranged in time sequence. It shows the objects participating in an interaction and the sequence of messages exchanged. A message is a signal or the call of an operation. The receipt of a message instance is normally considered an instance of an event.

A collaboration diagram shows interactions organized around objects and their links within a collaboration. Unlike a sequence diagram, a collaboration diagram explicitly shows the relationships among the roles. A collaboration diagram does not show time as a separate dimension, so the sequence of messages and the concurrent threads are determined using sequence numbers. Sequence diagrams and collaboration diagrams represent the same information, but show it in different ways. In Rational Rose, the two diagrams can be converted to one another easily. Collaboration diagrams are adopted for all the collaborations in our approach, because we are concerned with the messages
around the objects more than the sequence. Collaborations comprise nearly all the information that is obtained from analysis and required by design. However, a gap exists between use cases and collaboration/sequence diagrams. In collaborations, the analysis classes and their stereotypes are supposed to be known beforehand, but actually not. Therefore we need to introduce another kind of artefact to bridge the gap, and it is so-called the *robustness diagram*.

Doug Rosenberg and Kendall Scott have published two books about use-case driven object-oriented modeling [35, 36]. They introduced a process named ICONIX for object-oriented analysis and design. An overview of the ICONIX process is shown in Figure 3.2. The major contribution of ICONIX is the application of the robustness analysis and the robustness diagram to OOAD, which is the difference between ICONIX and generic RUP approaches. They claimed that robustness analysis is missing from RUP. However some have debated and said that the robustness analysis is still alive in the RUP [37].

![Figure 3.2 The ICONIX Process. [36]](image)
We adopt the robustness analysis and the robustness diagram in our approach. The robustness diagram acts as a bridge between a use-case model and a collaboration artifact. A difference between our methodology and ICONIX is that we suggest user interface details should not be included in the use-case specifications. Too much user interface description in use cases makes the requirements brittle. If the design for user interfaces is changed, the use cases will have to been changed accordingly, which introduces potential defects to the requirements.

Robustness Diagram

A diagram that shows the object instances with their stereotypes and the associations among them in a use case. The stereotypes are <<boundary>>, <<control>> and <<entity>>. Boundary objects model the presentation of the system. Control objects model the behavior of the system. Entity objects model the information in the system. An association models the connection among the objects with stereotypes at the association ends. In a robustness diagram, the association may or may not have navigability. A navigability suggests a possible message direction from one association end to another.

Notation

\[\begin{array}{ccc}
\text{<<boundary>>} & \text{<<control>>} & \text{<<entity>>} \\
\end{array}\]

association

association with navigability

Figure 3.3 Robustness diagram notations.
**Robustness Analysis**

A set of activities including use-case analysis, object elicitation, stereotype assignment and association identification. The output of robustness analysis is the robustness diagram.

The map of artefacts in Figure 3.1 can be updated as shown in Figure 3.4.

![Figure 3.4 The updated map of artefacts.](image)

### 3.1.2 Design model

Design classes and the relationships among them are the central artefacts generated by design activities. Design classes are derived from analysis classes with the operations, relationships and more properties identified. The stereotypes are optional for design classes. A class diagram shows design classes and the relationships among them. The relationships include association, generalization, dependency and constraint. A generalization relationship defines a taxonomic relationship between a more general class and a more specific class. Aggregation and composition are two forms of associations,
and specify whole-part relationships between classes. Subsystems, interfaces and protocols are also design artefacts. In our methodology only classes and the class diagram(s) are generated. More detailed design can be carried out by the developers based on the model generated by our tool. The final version of artifact map is shown in Figure 3.5.

![Diagram](image_url)

**Figure 3.5** The final map of artefacts.

### 3.2 Use-case Processing Method

To perform use-case driven analysis and design, we propose a use-case processing method as follows:

**For each use case**

- *a.* elicit the analysis classes, identify their stereotypes, and generate the robustness diagram;

- *b.* decompose the system’s behavior, distribute the behavior to analysis classes, and generate the collaboration diagram.

**For the analysis classes identified**

- *c.* describe the responsibilities;
d. describe the associations and establish them in the class diagram;

e. find the generalization relationships and establish them in the class diagram.

The hazards of use cases for object-oriented projects have been discussed in Section 2.1.4 of this thesis. A hazard is that use cases are typically functional decomposition of the system, while decomposition by objects is required in OOAD. The purpose of use-case driven OOAD is to change a system described by use cases to one described by objects/classes in object decomposition. To achieve this, the developers identify the objects in the requirements, decompose the system’s behavior, distribute the behavior to the proper analysis classes, and finally structure the classes with relationships.

3.2.1 Object elicitation

An object is an entity with well-defined boundary and identity that encapsulates state and behavior. We can find such properties of an object from the concept:

- An object should have state and behavior.
- An object should be either an event recognizer or an event responder.
- An object should not be redundant, because of unique identity.

Shlaer and Mellor suggest that candidate objects are usually derived from one of the following sources [38].

- Tangible things Cars, telemetry data, pressure sensors, etc.
- Roles Mother, teacher, politician, etc.
- Events Landing, interrupt, request, etc.
- Interactions Loan, meeting, intersection, etc.

From the perspective of data modeling, Ross offers a similar list [39]:

- People Human who carry out the function
- **Place**  
  Areas set aside for people or things

- **Things**  
  Physical objects, or groups of objects, that are tangible

- **Organizations**  
  Formally organized collections of people, resources, facilities, and capabilities having a defined mission, whose existence is largely independent of individuals

- **Concepts**  
  Principles or ideas not tangible per se; used to organize or keep track of business activities and/or communications

- **Events**  
  Things that happen, usually to something else at a given date and time, or as steps in an ordered sequence

Norman presents a list much similar to Shlaer and Mellor's, and names the method "conglomeration Strategy" [40].

- **Tangible items**  
  Such as vehicles, furniture, credit/debit card, insurance policies, buildings, animals, scanners, keyboards, displays, conveyors, bins, invoices, shipping documents, sales receipts, ATM transaction receipts, paychecks, patient charts, maps, and so on.

- **Roles**  
  People or organization such as student, teacher, administrator, patient, clerk, employee, nurse, homeowner, department, region, sale office, division, and so on.

- **Incidents/interactions**  
  Incidents happen at a specific point in time such as paying bills, an airplane flight, payday for employees, register for classes, football or other game, maintenance call, telephone call, and so on.

- **Specification**  
  A tabular (row/column) quality such as list of sales offices, list of U.S. state abbreviations, list of Standard Industry Codes (SIC), tax rate table, and so on.

These are very general guidelines. For a certain system under design, the domain knowledge is more important. The glossary is used in use-case writing to reduce the
ambiguity of the natural language and it is used in object elicitation in our approach. If an entity appearing in the use case can be found in the glossary, it is a candidate object that may correspond to a class in the concept model. Early approaches to find objects focus on noun analysis. However, verb analysis should also be a way to find the objects from requirements, because the behavior is a key characteristic of objects and is represented by verb phrases in use-case specifications. In our approach, the use-case specifications are analyzed based on statement structures. We present some rules describing the associations among the syntax, candidate objects, and their behaviors in Chapter 4.

3.2.2 Analysis class stereotypes

The idea of three stereotypes was introduced by Jacobson [6]. He believes that the objects can be classified to three types: entity, boundary (interface at first) and control. The types can respectively be mapped to three aspects of a system: information, presentation and behavior. He thinks that these three aspects are orthogonal with each other, which means that the couplings among classes can be reduced to the least extent if each class is designed to represent only one aspect. The analysis using these stereotypes suggests a robust structure for an object-oriented system. Robustness means the system developed in such a structure can adapt to the unexpected requests of changes due to requirements changes. It also means easy maintenance and good extensibility. A robust design can reduce the hazards that may occur in the latter phases of software development. The robustness analysis and the robustness diagram are derived from these ideas.

A boundary class is used to model the system interface, which interacts with actors directly. It handles the communication between the system’s surroundings and the inside
of the system. The system's surroundings are the actors interacting with the system. Only through the boundary classes can either the primary actors or the supporting actors access the system under design. An entity class models a real world entity, which is typically independent of the surroundings. To increase its cohesion, an entity class cannot interact with actors or the boundary class directly. The control class is responsible for the flow of events in use cases and is application-dependent. It connects the boundary class and the entity class together. Determining the control classes for a system is subjective.

Once an object is identified, an analysis class will be created for that kind of object. The class's stereotype is determined based on the context. Chapter 4 lists the rules applied to determine the analysis classes' stereotypes. The analysis classes appearing in the robustness diagrams must have stereotypes and are displayed by the notations introduced in Section 3.1.1 of this chapter. Sometimes, the stereotype of one class may be changed during the analysis process, but the class can have only one stereotype when the analysis is finished.

3.2.3 System's behavior and classes' responsibilities

The behavior is the observable effects of an operation or event, including its results. The behavior of a system is described in use cases. The actions or events in the basic flow, alternative flows and the postconditions are all the system's behavior. The classes' behaviors are subsets of the system's behavior and are observed when one class supplies services to another class. The use-case realization is to separate the concerns of the use-case specifiers of the system (represented by use-case diagrams and use-case specifications) to the concerns of the designers (objects and classes) of the system. The
use-case realization provides artefacts that typically consist of the collaboration or sequence diagrams which express use cases in terms of collaborating objects.

In collaboration or sequence diagrams, the system's behavior is implemented by groups of objects that exchange messages within a context to accomplish a purpose. The purpose represents a goal of the stakeholder(s). A sequence of messages within a diagram is called an interaction corresponding to the interaction in use cases. A message is the conveyance of information from one object (or instance) to another with the expectation that activity will ensue. A message may be a signal or the call of an operation. A signal is an asynchronous communication between objects. The receipt of a signal is an event that is intended to trigger transitions in the receiver's state machine. A call is to invoke an operation of an object. Either a signal or a call is an action. A message consists of a sender, a receiver and an action. The receiver has the responsibility for the execution of the action. The sender and the receiver can be the same object. The messages in collaboration diagram can help to identify the classes' responsibilities. Figure 3.6 shows a part of a collaboration diagram and the classes with the responsibilities identified. The collaboration diagram is the realization of a use case named "Withdraw Cash" for an ATM system.

![Collaboration Diagram](image)

Figure 3.6 Messages and classes' responsibilities.
3.2.4 Relationship identification

Association and generalization relationships are two kinds of relationships to be identified. Association relationships can be found from the association among the analysis classes in robustness diagrams or the links among the classes in collaboration diagrams. An aggregation is a form of association that specifies a whole-part relationship between an aggregate (a whole) and a constituent part.

Some of the aggregation relationships can be found from the concept model of the system. Some can be found from the use-case relationships. If one use case includes another use case, the core control classes in the use cases are likely to have an include relationship. Figure 3.7 is an example. The use case “Perform Session” includes another use case “Perform Transaction”. Session is the core control class identified from “Perform Session”. Transaction is the core control class identified from “Perform Transaction”. Because there may be several transactions in a session, there is an aggregation relationship between the class Transaction and the class Session.

An inheritance is represented by a generalization relationship in UML. Like associations, some of the generalization relationships can be found in concept model. Lots of senior OO designers prefer to create complex class hierarchies with some abstract classes. It is hard to create such hierarchy automatically. Design patterns are helpful in some cases [41]. Generalizations can also be identified from use-case relationships. If one use case has a generalization relationship with another use case, the core control classes
of the use cases are likely to have a generalization relationship as well. Figure 3.8 shows an example. The use case “Transaction” is specified into four use cases: “Withdraw Cash”, “Transfer Money”, “Deposit Money” and “Inquire Account”. Correspondingly the class Transaction is inherited by Withdrawal Transaction, Transfer Transaction, Deposit Transaction and Inquiry Transaction.

Figure 3.8 Use case generalization and class generalization.

3.2.5 Use cases to scenarios

Although the analysis and design are said to be use-case driven, they are actually scenario driven. The collaboration diagram or the sequence diagram is the key artifact connecting the system described by behaviors and that described by objects. They are all scenario diagrams. The term, scenario diagram, is used by Rational Rose development team. When we generate the scenario diagram, we need to decide how many diagrams to generate for a use case. Generally, a collaboration or sequence diagram is required for each scenario in the use case. There will be many same messages in the diagrams
generated from scenarios of one use case, but only one is enough to identify the receiver's responsibility and the rest are redundant. In our approach, the scenarios of a use case are represented in the same collaboration diagram if they share the same analysis classes.

3.3 Summary

We specify the analysis model as robustness diagrams and collaboration diagrams. The design model is specified as class diagrams in our approach. A use-case processing method is applied to generate such artefacts. However the methods discussed in this chapter is not adequate to automate the use-case processing, and Chapter 4 will present the approach to automate the transition from use cases to the class model.
Chapter Four: AUTOMATING THE USE-CASE DRIVEN DEVELOPMENT

Automation has two meanings in this project: first, the process can be automated by the methodology logically; secondly, the process is supported by the tool and automated practically. This chapter focuses on the first point. The rules for object identification, message identification and robustness diagram validation are discussed.

4.1 Behavior Types in Interactions

Our methodology emphasizes the analysis on behavior when processing the use-case statements in natural language. We propose the behavior types in the interactions between the actors and the system. Recall the actor-system interaction model introduced in Chapter 2. We define 4 behavior types that can cover most cases occurring during the interactions. They are Request, Validation, Change, and Response shown in Figure 4.1.

![Figure 4.1 The behavior types in actor-system interactions.](image)

The behavior described by a statement in use-case specifications must belong to one of the behavior types. The different behavior types are related with different associations between objects with stereotypes in the robustness diagram. Table 4.1 shows the relationships between the behavior types and associations in the robustness diagram. The
relationships are helpful for object identification and stereotype assignments. Our methodology identifies the objects and assigns their stereotypes at the same time.

Table 4.1 The behavior types and the associations.

<table>
<thead>
<tr>
<th>Behavior Type</th>
<th>Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request</td>
<td><img src="image" alt="Request" /></td>
</tr>
<tr>
<td>Validation</td>
<td><img src="image" alt="Validation" /></td>
</tr>
<tr>
<td>Change</td>
<td><img src="image" alt="Change" /></td>
</tr>
<tr>
<td>Response</td>
<td><img src="image" alt="Response" /></td>
</tr>
</tbody>
</table>

![actor, ![boundary object, ![control object, ![entity object](image)](image)](image)

4.2 Statement Analysis

Object identification is the first step of the use-case processing method introduced in Chapter 3. The process of object identification is as follows.

a. The action statements in the flow of events are parsed into phrases.

b. The statement structures are identified, and the rule set is found according to the statement structures. The rules and their comments are shown in Table 4.2.

c. The suitable rule is found from the rule set, and the objects and their stereotypes are identified.

d. The object is added to the object set, if it was not identified and added before. If the new identified stereotype of an object is different from the old one, the confidence degree is checked, and the stereotype is reassigned.
Sometimes, an object may be identified as different stereotypes, if there are conflicts among the rules. We assign a confidence degree to each rule. The higher the confidence degree of a rule is, the more confident the reasoning based on the rule is. Therefore if there is a conflict between the reasoning results based on two rules, the one with higher confidence degree is accepted. The confidence degrees of the rules are listed in Table 4.2.

When the objects inside the statements are identified, the statements will be processed again and the association among the objects are identified. The objects with the stereotypes and the associations among them comprise the robustness diagram. If a statement contains an actor and an object or two objects, an association can be identified from the statement. The association identified is added to the set of associations, if it is not in the set yet. With objects and the associations among them identified, the robustness diagram is generated.

Another kind of artifact to generate is the collaboration diagram that consists of the objects, the links among them and the messages. In practice, only the messages are required at this moment because the analysis classes are already found and the information of links is included in the messages. The rules for message identification are listed in Table 4.3. When the messages are identified, the collaboration diagram can be generated. The class diagram can be generated with the responsibilities of the classes distributed following the use-case processing method.
Table 4.2 The rules for object identification based on statement structures.

<table>
<thead>
<tr>
<th>Statement Structure</th>
<th>Symbol</th>
<th>Structure</th>
<th>Rule</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transitive SS1</td>
<td></td>
<td><img src="image" alt="Diagram" /></td>
<td>If Subject/NP//Noun(head) is an actor, and Predicate/PP/NP//Noun(head) is in the glossary and Predicate/PP/Prep is a special preposition, then Predicate/PP/NP//Noun(head) is an object with the stereotype as boundary; (CD high) and Predicate/VPSS/NP//Noun(head) is in the glossary, then Predicate/VPsp/NP//Noun(head) is an object with the stereotype as entity. (CD median)</td>
<td>An actor sends a request to the system through a boundary object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intransitive SS2</td>
<td></td>
<td><img src="image" alt="Diagram" /></td>
<td>If Subject/NP//head noun is system, and Predicate/PP/NP//Noun(head) is in the glossary and Predicate/PP/Prep is a special preposition, then Predicate/PP/NP//Noun(head) is an object with the stereotype as boundary or entity; (CD median) and Predicate/VPSS/NP//Noun(head) is in the glossary, then Predicate/VPsp/NP//Noun(head) is an object with the stereotype as entity. (CD median) and Predicate//Vgp//Verb(head) is a special verb, then Predicate/VPsp/NP//Noun(head) is an object with the stereotype as control; (CD high)</td>
<td>The system sends a response to the actor through a boundary object. The system validates the request or changes the state.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The system changes the state.
<table>
<thead>
<tr>
<th>Row</th>
<th>Diagram</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ditransitive SS3</td>
<td><img src="image" alt="Diagram" /></td>
<td>If Subject/NP//Noun(head) is an actor or <em>system,</em> and Predicate/NP2//Noun(head) is in the glossary, then Predicate/NP2//Noun(head) is an object with the stereotype as boundary; (CD high) and Predicate/NP1//Noun(head) is in the glossary, then Predicate/NP1//Noun(head) is an object with the stereotype as entity; (CD median)</td>
<td>The actor or the system sends a request or a response to the system or an actor through a boundary object.</td>
</tr>
<tr>
<td>Intensive SS4a</td>
<td><img src="image" alt="Diagram" /></td>
<td>Not available.</td>
<td>Seldom used in use-case writing.</td>
</tr>
<tr>
<td>Intensive SS4b</td>
<td><img src="image" alt="Diagram" /></td>
<td>If Subject/NP//Noun(head) has been identified as an object, then AP is an attribute of the object; (CD median)</td>
<td>The statement describes the object's attribute.</td>
</tr>
<tr>
<td>Intensive SS4c</td>
<td><img src="image" alt="Diagram" /></td>
<td>If Predicate/PP/NP//Noun(head) has been identified as an object, and Predicate/PP/Prep is a special preposition, then Subject/NP//Noun(head) is an attribute of the object; (CD median)</td>
<td>The statement describes the association relationship between two objects.</td>
</tr>
<tr>
<td>Complex Transitive SS5a</td>
<td><img src="image" alt="Diagram" /></td>
<td>If Subject/NP//Noun(head) is <em>system,</em> and Predicate/NP1//Noun(head) is in the glossary, then Predicate/NP1//Noun(head) is an object with the stereotype as entity, and Predicate/NP2//Noun(head) is a state of Predicate/NP1//Noun(head); (CD median)</td>
<td>The system changes an entity object's state.</td>
</tr>
<tr>
<td>SS5b</td>
<td>Statement</td>
<td>Subject</td>
<td>Predicate</td>
</tr>
<tr>
<td>SS5c</td>
<td>Statement</td>
<td>Subject</td>
<td>Predicate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subject</td>
<td>Predicate</td>
</tr>
<tr>
<td>Prepositional SS6</td>
<td>Statement</td>
<td>Subject</td>
<td>Predicate</td>
</tr>
<tr>
<td>Non-finite SS7a</td>
<td>Statement</td>
<td>Subject</td>
<td>Predicate</td>
</tr>
<tr>
<td>SS7b</td>
<td>If Subject/NP/Noun(head) is <em>system</em>, and Predicate/VPss/NP/Noun(head) is in the glossary, then Predicate/VPss/NP/Noun(head) is an object with the stereotype as entity; (CD median)</td>
<td>The system keeps an entity object in a certain state.</td>
<td></td>
</tr>
<tr>
<td>SS7c</td>
<td>The system changes an entity object’s state.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:**


2. / Child operator; selects immediate children of the left-side collection. A/B denotes B that is immediate child of A. // Recursive descent; select the first found children in the zero or more levels of the left-side collection. A/B denotes B that is the first found children in the zero or more levels of B.

3. Only the structure of Predicate/VPss is showed in the structure diagrams.

4. Special prepositions include *on, to, into* and *from.*

5. Special verbs include *start and end.*

6. This rules set is not complete.
Table 4.3 The rules for message identification based on statement structures.

<table>
<thead>
<tr>
<th>Statement Structure</th>
<th>Symbol</th>
<th>Structure</th>
<th>Rule</th>
</tr>
</thead>
</table>
| **Transitive**      | SS1    | ![Diagram](image) | If Subject/NP//Noun(head) is an actor or *system* and Predicate/PP/NP//Noun(head) is a boundary object, then the message is Predicate/Vgp//Verb(head) + Predicate/VPss/NP//Noun(head), the sender is Subject/NP//Noun(head) and the receiver is Predicate/PP/NP//Noun(head).
|                     |        |           | If Subject/NP//Noun(head) is an actor or *system*, and Predicate/PP/NP//Noun(head) is a boundary object, and Predicate/PP/Prep is *from*, then the message is Predicate/Vgp//Verb(head) + Predicate/VPss/NP//Noun(head), the sender is Predicate/PP/NP//Noun(head) and the receiver is Subject/NP//Noun(head).
|                     |        |           | If Subject/NP//Noun(head) is *system*, and Predicate/PP/NP//Noun(head) is an entity object, then the message is Predicate/Vgp//Verb(head) + Predicate/VPss/NP//Noun(head), the sender is the corresponding control object and the receiver is Predicate/PP/NP//Noun(head).
|                     |        |           | If Subject/NP//Noun(head) is *system* and Predicate/VPss/NP//Noun(head) is a control object, then the message is Predicate/Vgp//Verb(head), both the sender and the receiver are Predicate/VPss/NP//Noun(head). |
| **Intransitive**    | SS2    | ![Diagram](image) | If Subject/NP//Noun(head) is a control object, then the message is Predicate/Vgp//Verb(head), both the sender and the receiver are Subject/NP//Noun(head). |
| Ditransitive | SS3 | Statement: Subject, Predicate (NP, Vgp, NP1ToNP2)  
If Subject/NP/Noun(head) is an actor or system and Predicate/NP2/Noun(head) is a boundary object, then the message is Predicate/Vgp/Verb(head) + Predicate/NP1/Noun(head), then sender is the actor or the corresponding control object and the receiver is Predicate/NP2/Noun(head). |
|---|---|---|
| Intensive | SS4 | Statement: Subject, Predicate (NP, Vgp, NP1ToNP2)  
No message identified. |
| Complex Transitive | SS5a | Statement: Subject, Predicate (NP, Vgp, NP1ToNP2)  
If Subject/NP/Noun(head) is system, and Predicate/NP1/Noun(head) is an entity object, then the message is Predicate/Vgp/Verb(head) + Predicate/NP2/Noun(head), the sender is the corresponding control object and the receiver is Predicate/NP1/Noun(head). |
| SS5b | Statement: Subject, Predicate (NP, Vgp, NP1ToNP2)  
If Subject/NP/Noun(head) is system, and Predicate/NP1/Noun(head) is an entity object, then the message is Predicate/Vgp/Verb(head) + Predicate/AP/Adj(head), the sender is the corresponding control object and the receiver is Predicate/NP1/Noun(head). |
| SS5c | If Subject/NP//Noun(head) is *system*, and Predicate/VPss/PP/NP//Noun(head) is an entity object, then the message is Predicate/Vgp//Verb(head) + Predicate/NP1//Noun(head), the sender is the corresponding control object and the receiver is Predicate/VPss/PP/NP//Noun(head).

If Subject/NP//Noun(head) is an actor or *system*, and Predicate/VPss/PP/NP//Noun(head) is a boundary object, then the message is Predicate/Vgp//Verb(head) + Predicate/NP1//Noun(head), the sender is Subject/NP//Noun(head) and the receiver is Predicate/VPss/PP/NP//Noun(head).

| Prepositional SS6 | No message identified. |
| Non-finite SS7 | If Subject/NP//Noun(head) is system, and Predicate/NP//Noun(head) is an entity object, then the message is Predicate/VP | PresentParticiple | PastParticiple//Verb(head), the sender is the corresponding control system and the receiver is Predicate/NP//Noun(head). |

Note:


2. / Child operator; selects immediate children of the left-side collection. A/B denotes B that is immediate child of A. // Recursive descent; select the first found children in the zero or more levels of the left-side collection. A//B denotes B that is the first found children in the zero or more levels of B. | Set operation; returns the union of two sets of nodes. A | B denotes A or B.

3. This rules set is not complete.
4.3 Robustness Diagram Validation

Based on the methodology discussed in this chapter, most activities in analysis and design can be automated or semi-automated. A robust process should be iterative and incremental, so validation is necessary for the methodology to guarantee the generated artefacts are correct. The validation of class model can be performed by testing the codes implementing the classes, which is not discussed in this thesis. In this section, we propose a method to validate the analysis model, especially the robustness diagram.

There are some constraints for the objects and the associations in a robustness diagram. These constraints are based on the semantics of the robustness diagram. We use them to validate the robustness diagrams that were generated automatically or have been modified by the user. The constraints are as follows.

- There must be at least one actor in a robustness diagram.
- There must be at least one control object in a robustness diagram.
- There must be at least one boundary object in a robustness diagram.
- An actor can only associate with the boundary object(s).
- A boundary object can associate with the actor(s) or the control object(s).
- A control object can associate with the boundary object(s), the entity object(s) or other control object(s).
- An entity object can only associate with the control object(s).
- Either an actor or an object must have at least one association with other object.

The robustness diagram generated from a use case is validated according to these constraints. If the validation is not successful, there must be some mistakes to be
corrected in the diagram. The possible cases and the corresponding validation results are shown in Table 4.4.

Table 4.4 The robustness diagram validation rules.

<table>
<thead>
<tr>
<th>Case</th>
<th>Validation result</th>
<th>Suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td>Not allowed. One actor may be an object with the stereotype as boundary; or there may be an Object&lt;&lt;Boundary&gt;&gt; + Object&lt;&lt;Control&gt;&gt; + Object&lt;&lt;Boundary&gt;&gt; between the actors.</td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image3" alt="Diagram" /></td>
<td>Allowed</td>
<td><img src="image4" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image5" alt="Diagram" /></td>
<td>Not allowed. There may be an Object&lt;&lt;Boundary&gt;&gt; between them.</td>
<td><img src="image6" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image7" alt="Diagram" /></td>
<td>Not allowed. There may be an Object&lt;&lt;Boundary&gt;&gt; + Object&lt;&lt;Control&gt;&gt; between them.</td>
<td><img src="image8" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image9" alt="Diagram" /></td>
<td>Not allowed. There may be an Object&lt;&lt;Control&gt;&gt; between them.</td>
<td><img src="image10" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image11" alt="Diagram" /></td>
<td>Allowed.</td>
<td><img src="image12" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image13" alt="Diagram" /></td>
<td>Not allowed. There may be an Object&lt;&lt;Control&gt;&gt; between them.</td>
<td><img src="image14" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image15" alt="Diagram" /></td>
<td>Allowed.</td>
<td><img src="image16" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image17" alt="Diagram" /></td>
<td>Allowed.</td>
<td><img src="image18" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image19" alt="Diagram" /></td>
<td>Not allowed.</td>
<td><img src="image20" alt="Diagram" /></td>
</tr>
</tbody>
</table>

: actor, : boundary object, : control object, : entity object

The missing of the objects or the associations between objects is a kind of requirements incompleteness. We can trace such requirements faults back to a use case and even to the steps of a use-case specification. The validation of analysis model is supported by the CASE tool we developed.
4.4 A Case Study

In this section, a use case is processed using the methodology as a case study. It is not well organized and normalized at first. The following is the description of a use case named “Withdraw Cash” written by Bjork [30].

A withdrawal transaction asks the customer to choose a type of account to withdraw from (e.g. checking) from a menu of possible accounts, and to choose a dollar amount from a menu of possible amounts. The system verifies that it has sufficient money on hand to satisfy the request before sending the transaction to the bank. (If not, the customer is informed and asked to enter a different amount.) If the transaction is approved by the bank, the appropriate amount of cash is dispensed by the machine before it issues a receipt. (The dispensing of cash is also recorded in the ATM's log.) A withdrawal transaction can be cancelled by the customer pressing the Cancel key any time prior to choosing the dollar amount.

The normalized specification is as follows:

Actors: customer, bank

Flow of Events:

Basic Flow:

1. the system starts withdrawal transaction; //Transitive
2. the customer selects the account on the customer console; //Transitive
3. the system gets the account from the customer console; //Transitive
4. the customer selects the amount on the customer console; //Transitive
5. the system gets the amount from the customer console; //Transitive
6. the system generates the withdrawal transaction information; //Transitive
7. the system sends the withdrawal transaction information to the bank; //Ditransitive
8. the bank sends the withdrawal transaction approval to the system; //Ditransitive
9. the system dispenses the cash in the cash dispenser; //Transitive
10. the customer gets the cash from the cash dispenser; //Transitive
11. the system records the withdrawal transaction information into the log; // Transitive
12. the withdrawal transaction ends; //Intransitive

Alternative Flow:
If
the bank does not approve the withdrawal transaction, //Transitive
then
i. the system displays an error message on the customer console; //Transitive

ii. the system records the withdrawal transaction information into the log; // Transitive

iii. the withdrawal transaction ends; //Intransitive

The specification is processed using the methodology. Fig. 4.2 shows the robustness diagram that is generated according to the use-case specification.

![Robustness Diagram](image)

Figure 4.2 The robustness diagram of use case “Withdraw Cash”.

The robustness diagram is validated, and it is found that there should be a boundary object between the bank and the Withdrawal transaction class. The use-case specification is reviewed and steps 7 and 8 in the basic flow are revised as follows:

7. the system sends the withdrawal transaction information to the network connection; //Ditransitive
8. The bank gets the withdrawal transaction information from the network connection; //Transitive
9. The bank sends the withdrawal transaction approval to the network connection; //Ditransitive
10. The system gets the withdrawal transaction approval from the network connection; //Transitive

A new robustness diagram is generated according to the revised use-case specification shown in Fig. 4.3. The collaboration diagram is shown in Fig. 4.4, and the class diagram containing the identified classes from the use case is shown in Fig. 4.5. All the diagrams are automatically generated by UCDA, whose implementation will be discussed in next chapter.

Figure 4.3 The new robustness diagram of use case “Withdraw Cash”.
Figure 4.4 The collaboration diagram of use case “Withdraw Cash”.

Figure 4.5 The classes identified from use case “Withdraw Cash”.
4.5 Summary

To automate the transition from use cases to the class model, the rules to identify the objects, their stereotypes, and the messages in collaboration diagrams were discussed. The rules guarantee the use-case process can be automated logically. The method to perform the robustness diagram validation was presented. The practical automation of the process is carried out by the CASE tool. The implementation of the tool is discussed in the next chapter.
Chapter Five: IMPLEMENTATION

A CASE tool is designed and implemented to support the methodology discussed in the preceding chapters. We name the tool UCDA (Use-Case driven Development Assistant). The requirements and analysis of UCDA are discussed. The techniques for artifact representation, manipulation and visualization are presented. UCDA is seamlessly integrated with Rational Rose and deployed as an add-in of Rose.

5.1 Analysis and Design

5.1.1 Requirements and analysis

Tool-support is one of the key features of the Rational Unified Process. We also emphasize this point in our approach. The methodology discussed in this thesis is fully supported by UCDA that is discussed in this chapter. The features supplied by UCDA are listed as follows:

- Parse and analyze the use-case specifications.
- Parse, generate and store robustness diagrams, collaboration diagrams and class diagrams.
- Visualize the diagrams in Rational Rose.
- Validate the analysis model.

The inputs of UCDA are use-case diagrams and use-case specifications. The outputs are the generated artefacts in text formatting files and those visualized in Rose. The system's scope is determined by its features, inputs and outputs. The visualization of diagrams is performed by Rational Rose, and UCDA generates the diagrams and displays them in
Rose. Rational Rose plays the role of a support actor for UCDA under design. UCDA's behavior is shown by the use-case diagram in Figure 5.1.

![Use-case diagram of UCDA]

Figure 5.1 The use case diagram of UCDA.

Besides the features listed above, there are 2 more requirements for UCDA. The first requirement is that UCDA needs to integrate with the tool for another part of our project. As is introduced in Section 1.3.2 of this thesis, our project comprises two parts: the first part will generate the use-case specifications, and the second part is UCDA. UCDA needs to read the output of the first part, i.e., the use-case specifications. The second is that UCDA needs to integrate with Rational Rose. Because UCDA needs to access the application of Rose, the diagrams and other elements loaded in it, the techniques used to implement UCDA must support the extensibility of Rose. These issues will be discussed in detail in the following sections.

For the use case “Realize use cases”, the preconditions are that the use-case diagrams have been created in Rose and the use-case specification files are linked to the corresponding use cases. The postconditions of this use case are that the robustness
diagrams and the collaboration diagrams are generated and visualized in *Rose*. The activity diagram for this use case is shown in Figure 5.2.

![Activity Diagram](image)

**Figure 5.2** The activity diagram of use case "Realize use cases".

For the use case "Validate analysis model", a precondition is that the robustness diagrams have been generated, and the postconditions are that no mistakes are found in the robustness diagrams or the mistakes found are recorded in the log. The activity diagram of this use case is shown in Figure 5.3.

For the use case "Generate class model", a precondition is that the collaboration diagrams have been generated, and the postconditions are that the classes’ behaviors are
identified and the class diagram is generated. The activity diagram of this use case is shown in Figure 5.4.

Figure 5.3 The activity diagram of use case "Validate analysis model".

Figure 5.4 The activity diagram of use case "Generate class model".
5.1.2 A brief multiagent architecture of UCDA

Just like the name, UCDA are supposed to play the role of an assistant for the developers. There are three developer roles in the development process discussed in this thesis: analysis model designer for use-case realization and analysis model design, analysis model reviewer for analysis model review and validation and the class designer for class model design. UCDA is responsible for all the three roles. To perform the tasks, UCDA can carry out some autonomous behavior based on some knowledge and for some goals. We can apply agent-based paradigm to analyze the system.

Software agent was proposed to model the software that is autonomous, interactive, adaptive, mobile and intelligent. Basically we consider software agents as design patterns for software. The agent-based paradigm supplies methods to analyze and design the software that can be modeled as agents. The details about agent-based analysis and design are not discussed in this thesis. UCDA can be modeled as a multiagent system. The structure of UCDA as a multiagent system is shown in Figure 5.5. Analysis model designer agent will analyze the use-case model and use-case specifications and generates the robustness diagrams and the collaboration diagrams for the developer. The diagrams are visualized in Rational Rose. The analysis model reviewer will validate the robustness diagrams and records the results in the log. The class model designer agent will design the analysis classes in the analysis model and generate the class model. Again, the class diagram will be visualized in Rose. The knowledge supplier agent will supply the knowledge for the other agents. The knowledge is mainly the glossary for the domain under discussion.
In implementation, the agents can be considered as modules. Although the tool discussed in this thesis executes locally, an agent-based architecture will help modify the tool to a distributed one easily. The modules are developed separately, but there are some common issues for all the modules, e.g., the user interface, the date management and the system interface to Rational Rose. A four-layer model is applied to analyze these issues.

5.1.3 Analysis based on 4-layer model

To build a robust system, a four-layer model is applied to analyze the system [42]. The four layers are the User Interface (UI) layer, the Problem Domain (PD) layer, the Data Management (DM) layer and the System Interaction (SI) layer shown in Figure 5.6. The 4-layer model separates the concerns for different features of the system. With respect to UCDA under discussion, the interfaces through which the user will access the system belong to the UI layer; storage of use-case
specifications and other artefacts belong to the DM layer; the interface to *Rational Rose* application, diagrams and other model elements in *Rose* belong to the SI layer; and use-case analysis, artifact generation and validation belong to the PD layer. Since the logic of PD has been discussed in the previous chapters, the other 3 layers are discussed in this chapter.

The data to manage for UCDA include the glossaries, use-case diagrams, use-case specifications, robustness diagrams, collaboration diagrams and class diagrams. The structures and elements in use-case specifications, robustness diagrams and collaboration diagrams are so complex that we use XML (Extensible Markup Language) to represent all the artefacts UCDA will read or write outside of *Rose*. The representation and manipulation of artefacts are discussed in Section 5.2.

UCDA needs to access the *Rose* application, the diagrams and other model elements loaded in *Rose* via a system interface between UCDA and *Rose*. *Rational Rose* provides Rose Extensibility Interface (REI) for the developers to extend and customize its capabilities to meet specific software development needs. We perform this task by programming REI. The system interface is discussed in Section 5.3.

VBA (Visual Basic for Application) is applied to implement the user interfaces of UCDA. The dialogues and messages are supported by Windows platform services. The details about VBA for UI development are discussed in Section 5.4. The user can modify the diagrams visualized in *Rose* that helps UCDA to perform some functions of UIs.
5.2 Artifact representation and manipulation

To cooperate with the tool that generates use-case specifications, the use-case specifications are formalized strictly by the approaches introduced in Chapter 2 and the structure and elements are enforced by an XML schema. The schema is designed based on the use-case template and the use-case language schemas. The structure and elements of glossaries, robustness diagrams and collaboration diagrams are also enforced by XML schemas. The schemas of robustness diagrams and collaboration diagrams mirror their notations in UML to be directly visualized in Rational Rose. The system navigates and builds the artefacts according to their schemas.

5.2.1 XML and XML schema

The Extensible Markup Language (XML) is a subset of Standard Generalized Markup Language (SGML), an ISO standard [43], originally designed to meet the challenges of large-scale electronic publishing. XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and among applications [44]. XML provides a uniform method for describing and exchanging structured data that is independent of applications or vendors. XML makes the content and the presentation of information separate. For these advantages, we adopt XML for data representation and storage in the implementation. XML 1.0 is currently a W3C (World Wide Web Consortium, www.w3c.org) Recommendation.

An XML document represents and stores certain information in elements that are assembled in a structure. An element is specified by tags and attributes. When the data represented by XML documents or streams are exchanged on web or among applications,
the documents or streams must be validated to ensure that the information can be uniquely understood. The Document Type Definition (DTD) and the XML schema are both the techniques to enforce the structure and the elements in XML documents. Although DTD appeared early than the XML schema, the schema is more widely used, and offers more powers in date expression to the users. The XML Schema definition language (XSD) is the current W3C specification for XML schemas. The XML schemas in our approach are defined by XSD. The schemas for the use-case specification, the robustness diagram and the collaboration diagram are in the Appendix C.

Object Management Group (OMG) proposed a specification named XML Metadata Interchange (XMI) for defining, interchanging, manipulating and integrating XML data and objects [45]. Although some vendors of UML CASE tools claim that their products supported XMI-based UML model exchange, it is not well implemented. We found that even the model exported from Rose using the add-in suggested by Rational could not be fully imported into Together ControlCenter (www.togethersoft.com), which is claimed to support XMI. Therefore we define our own schemas for UML diagrams.

Recursion appears frequently in natural language. The XML schema for use-case specifications needs to deal with recursive relationships, e.g. a noun phrase (NP) comprises another noun phrase. There are two methods to define such a relationship. One method is to have global declarations of the elements, and refer to those global declarations using the ref attribute on xs:element, as follows:

```xml
<xsl:element name="NP">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="DET"/>
    </xs:sequence>
  </xs:complexType>
</xsl:element>
```
Another method is to have named complex types that will be referred to when the element is declared:

```xml
<xs:element name="NP" type="NP"/>
<xs:complexType name="NP">
  <xs:sequence>
    <xs:element type="DET" />
    <xs:element type="NP" />
  </xs:sequence>
</xs:complexType>
</xs:element>
```

The second method is applied to the schema for use-case specifications in the Appendix C.

### 5.2.2 DOM and XPath

To manipulate the XML documents, several programming interface standards have been developed. The Simple API (Application Programming Interface) for XML (SAX, www.saxproject.org) was the first widely adopted API for XML in Java [42], and now supported by most programming languages. SAX is a publicly developed standard for the events-based parsing of XML documents. SAX defines an abstract programmatic interface that models the XML information set through a linear sequence of familiar method calls.

The Document Object Model (DOM, http://www.w3.org/DOM/) provides another standard programming model for working with XML. It is fully described in the W3C DOM specification. The DOM was designed to:
- Provide a standard way to programmatically build, navigate or update and transform the contents of XML documents.

- Establish a core set of vendor- and language-neutral application programming interfaces that could be used to meet the most common needs for developers working with and generating well-formed XML input and output streams.

DOM is adopted to handle the XML documents in the system. The reasons to select DOM rather than SAX are as follows:

- The XML documents are not large, so to build the in-memory DOM tree will not damage the performance of UCDA and the platform system.

- Complex searches need to be implemented, which is difficult for SAX.

- XML documents will be created, modified or saved during use-case processing and artifact generation. SAX is designed mainly for reading, not writing XML documents.

When UCDA carries out use-case realization, the use-case specifications and the glossary are loaded into the memory and parsed into a tree by DOM. The new artefacts

---

**Figure 5.7 Working with DOM.**
are created and modified in the memory, and saved as XML files. Figure 5.7 shows the
tasks involved in XML document manipulation via DOM.

Addressing an element or a set of elements with certain properties is frequently
executed when an XML document is processed. The XML Path Language (XPath,
http://www.w3.org/TR/xpath) provides a language for addressing parts of an XML
document. It is fully described in the W3C XML Path Language (XPath) specification.

XPath was designed to:

- Provide an efficient, compact, logical syntax for referring to path or node tree
  locations within XML documents.
- Remain an effective language tool for referencing the contents of XML
documents separate of any markup used within them.
- Offer a standard library of functions for working with strings, numbers, and
  Boolean expressions when writing programs that process XML documents.
- Work closely with Extensible Stylesheet Language Transformations (XSLT) and
  other languages or programming interfaces designed for working with XML.

XPath can cooperate with DOM to address and process the elements in an XML
document. XPath makes the expression of the elements with certain tag names, attributes,
text content compact, e.g., all the nodes that are “Object” with the “Name” as “Card” and
the “Stereotype” as “Entity” can be expressed in XPath as:

\[
\text{Object[Name = "Card" and Stereotype = "Entity"]}
\]
5.2.3 MSXML

In our approach, DOM and Xpath are implemented using Microsoft XML Core Services (MSXML) 4.0. MSXML 4.0 allows customers to build XML-based applications that provide a high degree of interoperability with other applications that adhere to the XML 1.0 standard [46]. The reasons that we choose MSXML in the project are as follows:

- MSXML features DOM, XML schema and Xpath. These techniques are all required for our application to represent and manipulate the artefacts.

- MSXML is released as COM (Component Object Model) and can be easily accessed on Windows system. Our application need to support programming Rose Extensibility Interface (REI), which means it must support automation. We use Visual C++ (VC++) to program automation, and MSXML can be used in VC++ environment. Java supplies APIs implementing DOM and Xpath, but it is very hard to integrate the program in Java to that in VC++.

The Component Object Model (COM) is a specification developed by Microsoft for building software components that can be assembled into programs or add functionality to existing programs running on Microsoft Windows platforms. MSXML can be used in any programming environment that supports COM. We program in VC++ to process the use-case specifications in XML and generate the artefacts. In VC++ 6.0, the interfaces of DOM and XPath are accessed via smart pointers. The smart pointers are created when the MSXML COM is imported into the program:

```cpp
#import <msxml4.dll>
```

The namespace is declared to make the imported identifiers accessible.

```cpp
using namespace MSXML2;
```
The COM library is initialized by calling CoInitialize().

    CoInitialize(NULL);

The object is created by calling the CreateInstance() method of the declared class.

    //pXMLDOC is a smart pointer with type as XMLDOMDocument2.
    //__uuidof(DOMDocument40) retrieves the GUID (Global Unique
    //Identifier) of MSXML DOM 4.0.
    IXMLDOMDocument2Ptr pXMLDoc;
    HRESULT hr = pXMLDoc.CreateInstance(__uuidof(DOMDocument40));

The attributes and methods of the COM objects are accessed via the smart pointer, e.g.

    //Get the root of loaded XML document represented by pXMLDoc.
    IXMLDOMElementPtr pRoot;
    hr = pXMLDoc->get_documentElement(&pRoot);

MSXML is installed on most Windows systems by default. Different versions are
installed and work in a side-by-side mode, which means the newer version will not cover
the old one, and they can be used at the same time. UCDA is deployed with the support
of MSXML 4.0. If MSXML 4.0 is not installed on the system, UCDA’s install program
will install the application and register it without affecting other applications using
MSXML on the system.

5.3 Artefact Visualization and Accessing in Rose

The artefacts generated are represented and stored in XML by UCDA. Because the
diagrams need to be visualized in Rational Rose, we need to deal with the interface
between UCDA and Rose. Rose also helps to manage the artefacts and enable the
developer to modify the models graphically. UCDA can access the diagrams and other
elements in *Rose* model via Rose Extensibility Interface (REI). *Rational Rose* is a component-based application and defined in REI model. The structure of Rose application and extensibility components are shown in Figure 5.8. Rose application refers to the functionalities of *Rational Rose* system. Diagrams refer to all the diagrams displayed in *Rose* workspace. Rose model elements refer to actors, use cases, classes, the relationships among the classes, etc. All these resources are accessible by Rational Rose Automation or Rational Rose Script. Both methods are used in programming UCDA.

![Figure 5.8 Rose application and extensibility components. [47]](image)

### 5.3.1 Rose automation

Automation, formerly known as OLE (Object Linking and Embedding) Automation, is a Microsoft-designed technology that enables an application to expose objects and their properties for use by other applications. Nearly all the applications developed by Microsoft currently support automation, e.g., Microsoft Word, Microsoft Access, Microsoft Visio, etc. To have a good extensibility of their applications, some vendors other than Microsoft also support this feature. *Rational Rose* supplies Rose Extensibility Interface (REI) to support automation. REI allows the application to display and
manipulate diagrams and other Rose model elements. The Rose application acts as the automation server when the CASE tool as an automation client reads or modifies the Rose diagrams and other Rose elements. Automation can be either local or remote. Our tool functions locally, and can be extended to remote mode if needed. We use VC++ to implement automation.

REI supplies more than 130 COM objects to be accessed using automation. There are two methods to use REI classes in VC++: one method is using ClassWizard to find the class to be imported from the Rose’s type library file, rationalrose.tlb; another is to use #import to import the COM objects. Because only the MFC-based project created by VC++ can make use of ClassWizard, we use the second method when developing the program. First the type library is imported and no namespace needs to be declared for it.

```
#include <Program Files\Rational\Rose\rationalrose.tlb>  //No namespace
```

The COM library is initialized by calling CoInitialize().

```
CoInitialize(NULL);
```

The running Rose application object is found by calling GetActiveObject().

```
//CLSIDFromProgID() retrieves the class ID from the given //Programmatic ID, "Rose.Application".  //&lpUnk is the pointer to the requested Rose application object.
hr = CLSIDFromProgID(L"Rose.Application", &RoseID);
hr = GetActiveObject(RoseID, NULL, (LPUNKNOWN *)&lpUnk);
```

The Interface of the Rose application object is obtained by query.

```
//lpDispatch is the interface pointer.
hr = lpUnk->QueryInterface(IID_IDispatch, (LPVOID
The current model in the Rose application is obtained by

```c
IRoseApplicationPtr roseApp(lpDispatch);
IRoseModelPtr theModel = roseApp->GetCurrentModel();
```

All the diagrams and other model elements can be accessed via the current model. The interface between the module for use-case realization and Rose is performed by Automation in VC++.

### 5.3.2 Rose script

The Rose Scripting language is an extended version of the Summit BasicScript language that is developed based on Microsoft Visual Basic for Application (VBA). VBA is a hosted language and part of the Visual Basic family of development tools. VBA is actually an essential element of the retail version of VB, providing the vast majority of language elements used in VB [48]. *Rational Rose* provides the Rose Script Editor for the scripting development environment. The editor is started by clicking either `Tools > New Script` or `Tools > Open Script in Rose`.

It is very easy to access the Rose model in Rose Scripting language. *RoseApp* is reserved for the running Rose application. The current loaded model in *Rose* is accessed by

```vbscript
Dim theModel As Model
Set theModel = RoseApp.CurrentModel
```

All the diagrams and other model elements can be accessed via `theModel`. The Rose script can be saved as EBS file, or be compiled to EBX file. Both EBS and EBX files can be run in *Rose* environment. Compiled script files can run faster and protect the source
codes. The subsystems for analysis model validation and class model generation are programmed in Rose Scripting language, and compiled into EBX files for release.

5.4 User Interface

An advantage of programming in Rose script is that the services of Windows platform can be used directly [49]. User interface service is a package of the Windows platform services and comprises a package for Windows User Interface. With the components of Windows User Interface, the data exchange, the resources, the user input, and the windowing can be handled by programming in VBA. All the Graphical User Interfaces (GUI) of UCDA are developed using VBA with the support of Windows Platform SDK (Software Development Kit). The implementation of GUIs becomes compact in this way, and the program dealing with UIs does not couple with those dealing with business logic and Rose system interface. The following is a segment of VBA code for the dialog window shown in Figure 5.9.

```
Begin Dialog Realization,,196,104,"Realize Use Cases",_ 
.RealizeProc
CancelButton 144,86,40,14
ListBox 6,16,118,40,Usecases$,Usecases
Text 7,6,112,8,"Use Cases with XML Specification",.Text1,,8
PushButton 144,63,40,14,"Realize",.Realize
Text 7,61,57,8,"Glossary Address",.Text2,,8
PushButton 8,88,36,12,"Browse",.Browse
DropListBox 7,72,117,48,GlossaryFile$,GlossaryFile
End Dialog
```

Figure 5.9 A dialog window.
5.5 Developing an Add-In for Rose

Our tool could be deployed as a standalone application on Windows platforms, but the visualization of UML diagrams need to be supported by Rational Rose. To cooperate with Rational Rose and make use of the power of visual modeling, UCDA is deployed as an add-in of Rational Rose. To develop an add-in for Rational Rose, we need to consider at least these issues:

- Register and manage the add-in on the Windows system.
- Extend Rational Rose menus to enable the access to the add-in.
- Define custom data types.
- Integrate and release the programs.

A registry file for the add-in can be created and saved as REG file. The registry information will be added into the system’s registry by executing the following code in a batch file when the add-in is installed.

```
regedit /S \InstallPath\ISSUCC.reg
@echo off
cls
\InstallPath\ISSUCC.reg is the path and filename of the REG file. The content of the REG file is as follows:
```

Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SOFTWARE\Rational Software\Rose\AddIns\ISSUCDA]
"Active"="Yes"
"Company"="ISS, Univ. of Calgary"
"Copyright"="Copyright 2003, ISS Univ. of Calgary"
"HelpFileName"="UCDAHELP"
"InstallDir"="C:\Don\Add-in"
"LanguageAddIn"="No"
"MenuFile"="ISSUCDA.mnu"
"Version"="1.0"

After the system’s registry is updated by this file, we can use the Add-In Manager in *Rational Rose* to manage the add-in. To select the add-in to activate it in *Rose* is shown in Figure 5.10.

![Add-In Manager](image)

**Figure 5.10** Activation of the add-in in Add-In Manager.

The menu of *Rational Rose* would be extended to add the items that will trigger the add-in’s functions when they are clicked. For our add-in, three items are added, i.e. Use-case realization, Robustness validation and Class model generation. The menu file named `ISSUCDA.mnu` is as follows:

```plaintext
Menu Tools
{
    Separator
    Menu "Use Case to Class Model"
    {
        option "Use case Realization"
        {
            RoseScript C:\Don\Add-in\UseCaseRealization\realize.ebx
        }
    }
}
```
The extended menu is shown in Figure 5.11.

![Figure 5.11 The extended menu for the add-in.](image)

No new data type besides the robustness diagram is introduced to Rational Rose by the add-in. Because the robustness diagram is not specified by UML, Rational Rose does not support its visualization. The facility of the class diagram is used for the robustness diagram in our approach.

All the components that have been discussed need to be integrated together to be released. That is carried out by another layer named glue layer besides the four layers in the original structure. The glue layer is added to the problem-domain-centered structure.
to integrate the different layers into a whole system. Figure 5.12 shows the updated four-layer structure for the CASE tool. The glue layer between Data Management (DM) layer and Problem Domain (PD) layer is MSXML. VBA acts as the glue layer between UI (User Interface) layer and PD layer. The System Interface (SI) layer is accessed via automation and Rose script.

![Figure 5.12 The updated four-layer structure with glue layers.](image)

The user interface of the module for use-case realization is implemented in Rose script, and the business logic and data access are implemented in VC++ and released as DLL component that would be called by Rose script. The other two subsystems are implemented in Rose script.

### 5.6 Summary

The implementation of the CASE tool is discussed in this chapter. The requirements, the design based on the requirements, and the detail techniques used in coding are briefly introduced. The four-layer model is applied in the analysis and design of UCDA, and we try to get a robust system via component-based development. UCDA can support the methodology introduced in this thesis, and can be extended if the methodology is updated.
Chapter Six: CONCLUSION

This chapter presents the evaluation of UCDA approach, and reviews the contributions. The potential future extensions of the methodology and the tool are discussed.

6.1 Evaluation of UCDA

Our methodology comprises the good practices from both natural language requirements analysis and use-case driven analysis and design. We emphasize behavior analysis and suggest that object identification should company with the identification of object stereotypes based on the relationship between the action statements in use cases and the behavior types in an interaction. The methodology can help the analysts and the designers get a robust class model with the operations identified, while most of the other approaches supply only concept models.

UCDA can accelerate the transition from requirements to the class model. With the tool, even a developer with only a little experience and knowledge of OOAD can get the design model directly from the requirements represented in natural language. The automatically generated artefacts supply the original models for modification. Once the use-case specifications are modified because of the requirements changing, UCDA will generate the updated analysis and design models for the developer in a few minutes. Although these models still need to be revised before implementation, they give the developer a better vision of the system. This can reduce the time to market for the project in industry. UCDA generates the models with a robust structure emphasized, and gives the developer a good start point for design. UCDA can help the developer to check their analysis model and give corresponding suggestions, which make the review task faster.
6.2 Review of Contributions

The four contribution of the project is as follows:

- We proposed a use-case driven process for object elicitation and class modeling based on natural language requirements in OOAD. Our methodology combines the approaches of NL processing and the good practices of OO software development together, which is not addressed by many other approaches.
- I proposed the use-case language schemas for use-case specification normalization. The schemas are based on natural language statement structures and the characteristics of use-case modeling. The use-case language schemas can normalize the use-case specification without introducing too many constraints to natural language expressions.
- I proposed a method to identify the objects together with their stereotypes. The association relationships among the statement structures, the behavior types and the objects with stereotypes are presented in this thesis. The method emphasized the behavior analysis rather than the concept analysis.
- I proposed a method to validate the analysis model. The method can find and trace the missing objects or associations between objects in analysis phase, and give the developer suggestions that how to revise the analysis models and which use case to be checked.

6.3 Potential Future Extensions

The work described in this thesis may be extended in such points as:
Software architecture is important for OO software development. It should be performed based on both the functional and non-functional requirements before the class model is designed. A set of activities can be added into our methodology to carry out architecture analysis and design. The updated methodology is shown in Figure 6.1.

![Diagram showing updated methodology structure]

**Figure 6.1** The updated methodology structure.

The developer may want to validate the class model in design without developing the code based on the model. The statechart diagram is helpful to test the class model logically. If the statechart diagrams can be generated automatically, the class models can be validated easily. UCDA can generate the class model, and in future the statechart diagram generation can be developed and added into it.

Object identification is domain-dependent process. The domain knowledge is very important for the effectivity of the object model generated. We applied the glossary to represent part of the domain knowledge. But it would be better if the rules can adapt to the domain and represent part of the domain knowledge. A
separate rule base module can be developed to support such requirements in future work.

- UCDA visualizes the UML diagrams in *Rational Rose*, which is not convenient for the users who have no *Rose* installed on their systems. Because each diagram generated by UCDA is represented and stored in an XML file, XSLT (Extensible Stylesheet Language Transform) and SVG (Scalable Vector Graphics) can be used for the graphical representation of the diagrams. This feature will make UCDA handier for the users.

- The tool can be modified and developed as a distributed application to support a development organization other than individual designers. A group of developers can cooperated on a project at the same time and update the artefacts concurrently over the network. A multiagent architecture is suitable for such requests. The architecture discussed in Section 5.1 needs to be evolved to perform this feature.
List of Publications


References


[49] Microsoft, Microsoft Platform SDK. Available:

Appendix A: RUP Use Case Specification Template

Use Case Specification: <Use-Case Name>

1. Use Case Name

1.1 Brief Description

[The description should briefly convey the role and purpose of the use case. A single paragraph should suffice for this description.]

2. Flow of Events

2.1 Basic Flow

[This use case starts when the actor does something. An actor always initiates use Cases. The use case should describe what the actor does and what the system does in response. It should be phrased in the form of a dialog between the actor and the system. The use case should describe what happens inside the system, but not how or why. If information is exchanged, be specific about what is passed back and forth. For example, it is not very illuminating to say that the Actor enters customer information; it is better to say the Actor enters the customer’s name and address. A Glossary of Terms is often useful to keep the complexity of the use case manageable; you may want to define things like customer information there, to keep the use case from drowning in details. Simple alternatives may be presented within the text of the use case. If it only takes a few sentences to describe what happens when there is an alternative, do it directly within the flow of events section. If the alternative flows are more complex, use a separate section to describe it. For example An Alternative Flow describes how to describe more complex alternatives. A picture is sometimes worth a thousand words (though there is no substitute for clean, clear prose). If it improves clarity, feel free to paste graphical depictions of user interfaces, process flows, or other figures into the use case to improve its clarity. If a flow chart is useful to present a complex decision process, by all means use it! Similarly for state-dependent behavior, a state-transition diagram often clarifies the behavior of a system better than pages upon pages of text. Use the right presentation medium for your problem, but be wary of using terminology, notation or figures that your audience may not understand. Remember that your purpose is to clarify, not obscure.]

2.2 Alternative Flows

2.2.1 < First Alternative Flow >

[More complex alternatives should be described in a separate section, which is referred to in the basic flow of events section. Think of the alternative flow sections like alternative behavior — each alternative flow represents alternative
behavior (many times, because of exceptions that occur in the main flow). They may be as long as necessary to describe the events associated with the alternative behavior. When an alternative flow ends, the events of the main flow of events are resumed unless otherwise stated.]

2.2.1.1 < An alternative sub-flow >
[Alternative flows may in turn be broken down into sub-sections if it improves clarity.]

2.2.2 < Second Alternative Flow >
[There may be, and most likely will be, a number of alternative flows in a use case. Keep each alternative separate to improve clarity. Using alternative flows improves the readability of the use case, as well as preventing use cases from being decomposed into hierarchies of use cases. Keep in mind that use cases are just textual descriptions, and their main purpose is to document the behavior of a system in a clear, concise and understandable way.]

3. Special Requirements
[A Special Requirement is typically a non-functional requirement that is specific to a use case but is not easily or naturally specified in the text of the use case’s event flow. Examples of special requirements include legal and regulatory requirements, application standards, and quality attributes of the system to be built, including usability, reliability, performance or supportability requirements. Additionally, other requirements such as operating systems and environments, compatibility requirements, and design constraints should be captured in this section.]

3.1 < First special requirement >

4. PreConditions
[A precondition (of a use case) is the state of the system that must be present prior to a use case being performed.]

4.1 < Precondition One >

5. PostConditions
[A postcondition (of a use case) is a list of possible states the system can be in immediately after a use case has finished.]

5.1 < Postcondition One >

6. Extension Points
[Extension points of the use case.]

6.1 <name of extension point>
[Definition of the location of the extension point in the flow of events.]
Appendix B: RUP Glossary Template

Glossary

1. Introduction
   [The introduction of the Glossary should provide an overview of the entire document. Present any information the reader might need to understand the document in this section. This document is used to define terminology specific to the problem domain, explaining terms which may be unfamiliar to the reader of the use-case descriptions or other project documents. Often, this document can be used as an informal data dictionary, capturing data definitions so that use-case descriptions and other project documents can focus on what the system must do with the information. This document should be saved in a file called Glossary.]

1.1 Purpose
   [Specify the purpose of this Glossary.]

1.2 Scope
   [A brief description of the scope of this Glossary; what Project(s) it is associated with, and anything else that is affected or influenced by this document.]

1.3 References
   [This subsection should provide a complete list of all documents referenced elsewhere in the Glossary. Each document should be identified by title, report number (if applicable), date, and publishing organization. Specify the sources from which the references can be obtained. This information may be provided by reference to an appendix or to another document.]

1.4 Overview
   [This subsection should describe what the rest of the Glossary contains and explain how the document is organized.]

2. Definitions
   [The terms defined here form the essential substance of the document. They can be defined in any order desired, but generally alphabetic order provides the greatest accessibility.]

2.1 <aTerm>
   [The definition for <aTerm> is presented here. As much information as the reader needs to understand the concept should be presented.]

2.2 <anotherTerm>
   [The definition for <anotherTerm> is presented here. As much information as the reader needs to understand the concept should be presented]

2.3 <aGroupofTerms>
[Sometimes it is useful to organize terms into groups to improve readability. For example, if the problem domain contains terms related to both accounting and building construction (as would be the case if we were developing a system to manage construction projects), presenting the terms from the two different sub-domains might prove confusing to the reader. To solve this problem, we use groupings of terms. In presenting the grouping of terms, provide a short description that helps the reader understand what <aGroupOfTerms> represents. Terms presented within the group should be organized alphabetically for easy access.]

2.3.1 <aGroupTerm>
[The definition for <aGroupTerm> is presented here. Present as much information as the reader needs to understand the concept.]

2.3.2 <anotherGroupTerm>
[The definition for <anotherGroupTerm> is presented here. Present as much information as the reader needs to understand the concept.]

2.4 <aSecondGroupofTerms>

2.4.1 <yetAnotherGroupTerm>
[The definition for the term is presented here. Present as much information as the reader needs to understand the concept.]

2.4.2 <andAnotherGroupTerm>
[The definition for the term is presented here. Present as much information as the reader needs to understand the concept.]
Appendix C: XML Schemas

C.1 Use Case Specification Schema

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:complexType name="PP">
        <xs:annotation>
            <xs:documentation>Preposition phrase</xs:documentation>
        </xs:annotation>
        <xs:sequence>
            <xs:element name="Prep" type="xs:string">
                <xs:annotation>
                    <xs:documentation>Preposition</xs:documentation>
                </xs:annotation>
            </xs:element>
            <xs:element name="NP" type="NP"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="AP">
        <xs:annotation>
            <xs:documentation>Adjective Phrase</xs:documentation>
        </xs:annotation>
        <xs:sequence>
            <xs:element name="INT" minOccurs="0">
                <xs:annotation>
                    <xs:documentation>Intensifier</xs:documentation>
                </xs:annotation>
                <xs:simpleContent>
                    <xs:extension base="xs:string">
                        <xs:attribute name="POS" type="xs:string"/>
                    </xs:extension>
                </xs:simpleContent>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```
<xs:element name="Adj" type="xs:string">
  <xs:annotation>
    <xs:documentation>Adjective</xs:documentation>
  </xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
<xs:simpleType name="DEP">
  <xs:annotation>
    <xs:documentation>Determiner</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="TO">
  <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="SP" final="restriction">
  <xs:annotation>
    <xs:documentation>Statement Pattern</xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="SP1"/>
    <xs:enumeration value="SP2"/>
    <xs:enumeration value="SP3"/>
    <xs:enumeration value="SP4a"/>
    <xs:enumeration value="SP4b"/>
    <xs:enumeration value="SP4c"/>
    <xs:enumeration value="SP5a"/>
    <xs:enumeration value="SP5b"/>
    <xs:enumeration value="SP5c"/>
    <xs:enumeration value="SP6"/>
    <xs:enumeration value="SP7a"/>
    <xs:enumeration value="SP7b"/>
    <xs:enumeration value="SP7c"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="Actor"/>
an Actor has Name and Description:

The first character of actor's name should be lower-cased, the same as what in the statements.

The first character of actor's name should be lower-cased, the same as what in the statements.
<xs:element name="NP" type="NP"/>
</xs:sequence>
</xs:group>
<xs:group name="NOM3">
<xs:annotation>
  <xs:documentation>NOM+PP</xs:documentation>
</xs:annotation>
<xs:sequence>
  <xs:element name="NP" type="NP"/>
  <xs:element name="PP" type="PP"/>
</xs:sequence>
</xs:group>
<xs:complexType name="NP">
  <xs:annotation>
    <xs:documentation>Nominar</xs:documentation>
  </xs:annotation>
  <xs:choice>
    <xs:group ref="NOM1"/>
    <xs:group ref="NOM2"/>
    <xs:group ref="NOM3"/>
    <xs:element name="Noun" type="xs:string"/>
  </xs:choice>
</xs:complexType>
<xs:complexType name="Vgp">
  <xs:annotation>
    <xs:documentation>Verb Group</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="AUX" minOccurs="0">
      <xs:annotation>
        <xs:documentation>Auxiliary</xs:documentation>
      </xs:annotation>
      <xs:complexType>
        <xs:sequence>
          <xs:element name="AuxVerb" type="xs:string" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:complexType>
  <xs:element name="Verb" type="xs:string"/>
  <xs:element name="PhrasalVerb" type="xs:string"/>
</xs:choice>
</xs:sequence>
</xs:complexType>
<xs:group name="VP1">
  <xs:annotation>
    <xs:documentation>Transitive: Vgp+NP</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="Vgp" type="Vgp"/>
    <xs:element name="NP" type="NP"/>
  </xs:sequence>
</xs:group>
<xs:group name="VP3">
  <xs:annotation>
    <xs:documentation>Ditransitive: Vgp+NP1+TO+NP2</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="Vgp" type="Vgp"/>
    <xs:element name="NP1" type="NP"/>
    <xs:element name="TO" type="TO"/>
    <xs:element name="NP2" type="NP"/>
  </xs:sequence>
</xs:group>
<xs:group name="VP4">
  <xs:annotation>
    <xs:documentation>Intensive: LinkingVgp+NP/AP/PP</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="LinkingVgp" type="Vgp"/>
    <xs:element name="NP" type="NP"/>
  </xs:choice>
</xs:group>
<xs:element name="AP" type="AP"/>
<xs:element name="PP" type="PP"/>
</xs:choice>
</xs:sequence>
</xs:group>
<xs:group name="VP5">
<xs:annotation>
<xs:documentation>Complex Transitive: Vgp+NP1+NP2/AP/PP</xs:documentation>
</xs:annotation>
<xs:sequence>
<xs:element name="Vgp" type="Vgp"/>
<xs:element name="NP1" type="NP"/>
<xs:choice>
<xs:element name="NP2" type="NP"/>
<xs:element name="AP" type="AP"/>
<xs:element name="PP" type="PP"/>
</xs:choice>
</xs:sequence>
</xs:group>
<xs:group name="VP6">
<xs:annotation>
<xs:documentation>Prepositional: Vgp+PP</xs:documentation>
</xs:annotation>
<xs:sequence>
<xs:element name="Vgp" type="Vgp"/>
<xs:element name="PP" type="PP"/>
</xs:sequence>
</xs:group>
<xs:group name="VP7">
<xs:annotation>
<xs:documentation>Non-Finite: Vgp+NP1+TO-VP/Present-Participle/Past-Participle</xs:documentation>
</xs:annotation>
<xs:sequence>
<xs:element name="Vgp" type="Vgp"/>
<xs:element name="NP" type="NP"/>
<xs:choice>
  <xs:sequence>
    <xs:element name="TO" type="TO"/>
    <xs:element name="VP" type="VP"/>
  </xs:sequence>
  <xs:element name="PresentParticiple" type="xs:string"/>
  <xs:element name="PastParticiple" type="xs:string"/>
</xs:choice>
</xs:sequence>
</xs:group>
<xs:complexType name="VPsp">
  <xs:annotation>
    <xs:documentation>Verb Phrase with Statement Pattern</xs:documentation>
  </xs:annotation>
  <xs:choice>
    <xs:group ref="VP1"/>
    <xs:element name="Vgp" type="Vgp">
      <xs:annotation>
        <xs:documentation>Intransitive</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:group ref="VP3"/>
    <xs:group ref="VP4"/>
    <xs:group ref="VP5"/>
    <xs:group ref="VP6"/>
    <xs:group ref="VP7"/>
  </xs:choice>
  <xs:attribute name="SP" type="SP" use="required"/>
</xs:complexType>
<xs:complexType name="VP">
  <xs:annotation>
    <xs:documentation>Verb Phrase</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="VPsp" type="VPsp"/>
    <xs:element name="PP" type="PP" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="Statement">
  <xs:annotation>
    <xs:documentation>The basic sentence pattern.</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="Subject" type="NP"/>
    <xs:element name="Predicate" type="VP"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="FlowOfEvents">
  <xs:annotation>
    <xs:documentation>Including Basic Flow and Alternative Flow.</xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="BasicFlow"/>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="FlowStep" type="Statement" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="AlternativeFlow" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="If" type="Statement"/>
        <xs:element name="Then" type="FlowOfEvents"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:complexType>
<xs:element name="UseCaseSpecification">
<xs:annotation>
  <xs:documentation>All the use case specifications will have the same root as this.</xs:documentation>
</xs:annotation>
<xs:complexType>
  <xs:sequence>
    <xs:element name="ProjectName" type="xs:string">
      <xs:annotation>
        <xs:documentation>Put the name of the project here.</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="UseCaseName" type="xs:string">
      <xs:annotation>
        <xs:documentation>Put the name of the use case here.</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="Date" type="xs:date">
      <xs:annotation>
        <xs:documentation>The Date's format is YYYY-MM-DD, such as 2001-10-15.</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="Version" type="xs:decimal">
      <xs:annotation>
        <xs:documentation>Put the version of the specification document here.</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="Author" type="xs:string">
      <xs:annotation>
        <xs:documentation>The author's name.</xs:documentation>
      </xs:annotation>
    </xs:element>
    <xs:element name="Description" type="xs:string">
      <xs:annotation>
        <xs:documentation></xs:documentation>
      </xs:annotation>
    </xs:element>
  </xs:sequence>
</xs:complexType>
<xs:documentation>The description briefly convey the role and purpose of the use case.</xs:documentation>

<xs:element name="Actors">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Actor" type="Actor" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="FlowOfEvents" type="FlowOfEvents">
  <xs:annotation>
    <xs:documentation>Including basic flow and alternative flow.</xs:documentation>
  </xs:annotation>
</xs:element>

<xs:element name="SpecialRequirements">
  <xs:annotation>
    <xs:documentation>A special requirement is typically a non-functional requirement that is specific to a use case but is not easily or naturally specified in the text of the use case's even flow.</xs:documentation>
  </xs:annotation>
</xs:element>

<xs:element name="PreConditions">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="PreCondition" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
C.2 Robustness Diagram Schema

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:simpleType name="Stereotype" final="restriction">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Boundary"/>
      <xs:enumeration value="Control"/>
      <xs:enumeration value="Entity"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="Actor">
    <xs:sequence>
      <xs:element name="PostConditions">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="PostCondition" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="ExtensionPoints">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="ExtensionPoint" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```
<xs:sequence>
  <xs:element name="Name" type="xs:string"/>
  <xs:element name="Description" type="xs:string" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="Object">
  <xs:sequence>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="Stereotype" type="Stereotype"/>
  </xs:sequence>
</xs:complexType>
<xs:element name="RobustnessDiagram">
  <xs:annotation>
    <xs:documentation>Comment describing your root element</xs:documentation>
  </xs:annotation>
</xs:complexType>
<xs:element name="ProjectName" type="xs:string">
  <xs:annotation>
    <xs:documentation>Put the name of the project here.</xs:documentation>
  </xs:annotation>
</xs:element>
<xs:element name="UseCaseName" type="xs:string">
  <xs:annotation>
    <xs:documentation>Put the name of the use case here.</xs:documentation>
  </xs:annotation>
</xs:element>
<xs:element name="Version" type="xs:decimal">
  <xs:annotation>
    <xs:documentation>Put the version of the specification document here.</xs:documentation>
  </xs:annotation>
</xs:element>
<xs:element name="Author" type="xs:string"/>
<xs:annotation>
  <xs:documentation>The author's name.</xs:documentation>
</xs:annotation>
<xs:element>
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Description" type="xs:string">
        <xs:annotation>
          <xs:documentation>The description briefly convey the role and purpose of the use case.</xs:documentation>
        </xs:annotation>
      </xs:element>
      <xs:element name="Actors">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Actor" type="Actor" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Objects">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Object" type="Object" maxOccurs="unbounded"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Links">
        <xs:annotation>
          <xs:documentation>The objects found from the Flow of Events, with their names and stereotype included.</xs:documentation>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
The associations between the objects, with ElementA and ElementB.

ElementA(Supplier) c-------- ElementB

type="Stereotype" use="required"/>

ElementB

type="Stereotype" use="required"/>

ElementA

type="Stereotype" use="required"/>

ElementB
C.3 Collaboration Diagram Schema

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">

<xs:complexType name="Class">
    <xs:sequence>
        <xs:element name="Name" type="xs:string"/>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="Link">
    <xs:sequence>
        <xs:element name="Role1" type="xs:string"/>
        <xs:element name="Role2" type="xs:string"/>
    </xs:sequence>
</xs:complexType>

<xs:element name="CollaborationDiagram">
    <xs:annotation>
        <xs:documentation>Collaboration Diagram</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name="ProjectName" type="xs:string"/>
            <xs:element name="UseCaseName" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>
<xs:documentation>Put the name of the use case here.</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="Version" type="xs:decimal">
<xs:annotation>
<xs:documentation>Put the version of the specification document here.</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="Author" type="xs:string">
<xs:annotation>
<xs:documentation>The author's name.</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="Description" type="xs:string">
<xs:annotation>
<xs:documentation>The description briefly convey the role and purpose of the use case.</xs:documentation>
</xs:annotation>
</xs:element>
<xs:element name="Classes">
<xs:complexType>
<xs:sequence>
<xs:element name="Class" type="Class" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="Links">
<xs:complexType>
<xs:sequence>
<xs:element name="Link" type="Link" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="Messages">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Message" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Name" type="xs:string"/>
            <xs:element name="Sender" type="xs:string"/>
            <xs:element name="Receiver" type="xs:string"/>
            <xs:element name="Sequence" type="xs:integer"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>