1998

On semi-automatic translation of use cases.

Jinkui. Li

University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

https://scholar.uwindsor.ca/etd/2921

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000 ext. 3208.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6” x 9” black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI
On Semi-Automatic Translation of Use Cases

by
Jinkui Li

A Thesis
Submitted to the Faculty of Graduate Studies and Research
Through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree of
Master of Science at the
University of Windsor

Windsor, Ontario, Canada
1998
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-62240-1
Abstract

This thesis is concerned of automating the translation of the narrative description of use cases into sequence diagrams in the object-oriented analysis. We propose an architecture for a semi-automatic translation system that can translate descriptions of use cases to descriptions of sequence diagrams. The system is based on patterns of sentences in a natural language like English. We present a set of syntactic patterns to guide developers in writing recognizable sentences in use case descriptions and a set of naming rules to guide the system in constructing message names for the generated sequence diagram. We also present a technique for a user of the system to intervene the system translation so that special cases can be processed in a specified way. The sentences in a narrative use case description are converted to a set of message units, which can be used to compose a sequence diagram in UML. A parser does the conversion. A catalog is used to record information derived from the use case description. It also records specific translation decisions made by a user of the system.
To my parents
my brothers and sisters
my wife Ruizhu and my son Yu
Acknowledgements

I would like to acknowledge the support and guidance provided by Dr. Liwu Li, whose time, dedication and effort has contributed in guiding me through this thesis. Without his patience and guidance it would have been impossible to complete this thesis in this time frame. It needs to mention that Dr. Li proposed the idea of semi-automatic translation of use cases and the architecture for the semi-automatic translation implemented in the thesis. I am grateful to Dr. Morrissey and Dr. H. K. Kwan for proving valuable suggestions and comments that aided in my progress.

Special thanks to my wife Ruizhu for her encouragement and support and my lovely son Yu for the relaxation time he gave me.

Last but not least, I would like to thank all my colleagues in the grad lab for providing a friendly atmosphere throughout my Master program.
# TABLE OF CONTENTS

Abstract ........................................................................................................ iv

Acknowledgements ....................................................................................... vi

List of Figures ................................................................................................. xi

List of Tables ................................................................................................. xii

1 Introduction ................................................................................................. 1

1.1 Use Case ................................................................................................. 1

1.1.1 Actor ................................................................................................. 1

1.1.2 Use Case .......................................................................................... 2

1.1.3 Extends and Uses ............................................................................ 4

1.2 UML ........................................................................................................ 5

1.2.1 Concept ......................................................................................... 5

1.2.2 Use Case Diagram ......................................................................... 7

1.2.3 Class and Object Diagram ............................................................ 8

1.2.4 Sequence Diagram ....................................................................... 10

1.3 Translation from Use Case to Sequence Diagram ................................. 13

1.3.1 Necessity of Translating Use Case to Sequence diagram ............ 13

1.3.2 Natural and Programming Languages .......................................... 14

1.3.2.1 Vagueness of Natural Language ............................................. 14

1.3.2.2 Writing Habit and Style ......................................................... 15

1.3.2.3 Syntactic Requirement of Programming Language

   and Sequence Diagram ....................................................................... 16

1.4 Overview of the Thesis ........................................................................... 16
1.4.1 The Objective of This Thesis ........................................ 16
1.4.2 Organization of the Thesis ......................................... 18

2 Syntactic Patterns and Naming Rules .................................. 20

2.1 Basic Principles ......................................................... 20
  2.1.1 The Principles ..................................................... 20
  2.1.2 Explanations on Normalization Principles .................. 21

2.2 Syntactic Patterns ....................................................... 26
  2.2.1 Syntactic requirement of Programming Language .......... 26
  2.2.2 Notation Requirement of Sequence Diagrams ............. 27
  2.2.3 Patterns ............................................................ 28
    2.2.3.1 Patterns Definition ......................................... 28
    2.2.3.2 Matching Patterns with Sequence Diagram ........... 30
    2.2.3.3 Examples ...................................................... 33

2.3 Naming Rules ........................................................... 39
  2.3.1 Source Object and Target Object .............................. 39
  2.3.2 Source Class and Target Class ................................. 40
  2.3.3 Message Name ...................................................... 40
  2.3.4 Argument .......................................................... 42

2.4 Control Information .................................................. 43
  2.4.1 Condition (if-else phrase) .................................... 43
  2.4.2 Iteration .......................................................... 46

2.5 Message Unit .......................................................... 48
  2.5.1 String controlInfo ............................................... 48
List of Figures

Figure 1  Financial trading system .................................................. 8
Figure 2  Class diagram ................................................................. 9
Figure 3  Composite object diagram ............................................... 10
Figure 4  Sequence diagram for switching ...................................... 12
Figure 5  Proposed approach to mechanical translation of use case ...... 18
Figure 6  The architecture for translation of normalized use case .......... 25
Figure 7  Message send in sequence diagram .................................... 26
Figure 8  Condition in sequence diagram ........................................ 28
Figure 9  Iteration in sequence diagram .......................................... 30
Figure 10 If_Else phrase in sequence diagram ................................. 44
Figure 11 Iteration: control information ......................................... 47
Figure 12 Unit of sequence diagram, example 1 ................................ 51
Figure 13 Unit of sequence diagram, example 2 ............................... 52
Figure 14 Unit of sequence diagram, example 3 ............................... 53
Figure 15 Special pattern containing three objects ............................ 66
Figure 16 Sequence diagram for sales use case .................................. 68
Figure 17 Sequence diagram for control use case .............................. 70
List of Tables

Table 1  Syntactic patterns for narrative description of use case ..............30
Table 2  Condition and iteration ................................................. 48
Chapter 1  Introduction

1.1 Use Case

The notion of a use case, first introduced by Jacobson et al in 1994 [16], is an important technique for requirements specification for software system. Since then, lot of papers were published in OOPSLA’95 [3, 7, 8, 11, 24, 27], OOPSLA’97 [9, 14, 29], and some books and journals [2, 5, 18]. Some writers, like Ian Graham, used the term “script” in place of “use case” [13]. Viewpoint and service template [23] and VORD [19] were also used to catch the requirements. Use cases are used to improve understanding of the functionality of a software system. An object-oriented software project may specify the functionality of a software system with use cases. Then, it may create sequence diagrams and other diagrams to describe a logical structure for the system.

Use case model uses actor and use case to specify the functionality of the system, and to define what should take place inside the system. An actor defines what exists outside the system, a use case defines what should be performed by the system. The use case model is described by a number of actors and use cases.

1.1.1 Actor

An actor exists outside the system. It represents a certain role that a user can play with the system. The actor is different from user. We regard an actor as a class, and a user as an instance of this class. Therefore, many users may play the same role with respect to the system, one user may play several actors, and one use case may be performed by several actors.

An instance of an actor carries out a number of different operations on the system. Actors model anything that need to exchange information with the system. They can
model human users and other systems communicating with the system being developed.

The essential thing is that actors constitute anything that is external to the system. Normally, we do not view the functionality of the underlying system (for example, the operating system) as an actor. But in some cases, where the underlying system has an active role with respect to the application, it is very meaningful to model it as an actor. An actor may also be an external system that needs to interact with the system for some reasons. If an external system needs some functions of the system, the external system should be modeled as an actor. For example, an accounting system needs a file generated by the system to be built, the accounting system is an actor.

An actor may start use cases, may participate in use cases when the use cases progress, may receive value or service from some use cases. Since use case represent externally required functionality, they are externally tangible or visible by the potential users.

1.1.2 Use Case

The notion of use case is a primary element in object-oriented project development and planning. A use case is a typical interaction between a user and a computer system. When a user uses the system, she or he performs a behaviorally related sequence of transactions in a dialogue with the system. Such a special sequence is use case. Each use case is a special way of using the system and every execution of the use case may view as an instance of the use case.

A use case class is a description. This description specifies the transactions of the use case. Whenever we start a use case, we view this as we create instances of this class. The set of all use case descriptions specifies the complete functionality of the system. When a
user inputs a stimulus, the use case instance executes and starts a transaction belong to the use case. This transaction consists of different actions to be performed, and is finished when the use case instance again awaits an input stimulus from an actor instance. The use case instance exists as long as the use case is operating. The following are some examples of use cases:

- Withdraw cash
- Deposit
- Make some text bold
- Create an index

Some use cases serve **user goals**; others describe **system interactions**. System interactions reflect user’s interactions with the system in order to achieve their goals. System interaction use cases can be used in project planning. User goal use cases can be used to discover alternative ways to satisfy the goals. To create a use case model, focus on user goals first; then, come up with user goal and system interaction use cases. In the end of each interaction, we can expect a set of system interaction use cases for each user goal identified.

By using use case, we may achieve the following advantage in OOSE (Object-Oriented Software Engineering) Activities

- Use cases help us to focus on the problem, as they constitute a strong tool for defining the system functionality.
- They force the development forward as the use cases are identified and specified, since they are logical and straightforward to find.
• They are also a strong support when developing the subsequent models, since these models are based upon the use cases.

• It is possible to estimate the amount of work in subsequent models since you know how many use cases you have and you can predict the time needed to handle each use case.

1.1.3 Extends and Uses

**Extends** is a relationship between use cases. A basic use case carries out the usual behavior of the system. To avoid cluttering the use case with abnormal situations, we place the normal behavior in a **basic use case** and the unusual behaviors in other use cases, which are called **extension use cases** or simply **extensions**. The basic use case may be extended by the extension use cases. The relationships between a basic use and the extension use cases are extends relationships. To take the advantage of extends relationships in practice,

• Capture the simple, normal use case first.

• For every steps in the normal use case, ask "what could go wrong here?" and "how might this work out differently?"

• Plot the variations as extensions of the basic use case.

**Uses** is another relationship between use cases. The uses-relationships between use cases occurs when a use case can use the functionality or service provided by another use case. Several use cases may need the functionality or service of the same use case. Thus, we can factoring out the common functionality or service and create a use case to represent it, and let other use case **uses** it.

A uses-relationship and an extends-relationship have different intentions. The actor
that is linked to a use case is supposed to communicate both the use case and the extensions of the use case. There may be no actor is associated with a use case that is used by other use cases. The existence of the use case is the result of decision on presentation or organization of use cases. The used use case is an integral part of the using use case. It cannot be omitted from the using use case. An extension may not be needed by an extended use case. The extended use case is complete by itself. The extends-relationships can be used to describe variations of use cases. The uses-relationships can be used to factoring common parts of several use cases into a use case.

1.2 UML

1.2.1 Concept

The powerful modeling language, Unified Modeling Language (UML) [12, 15, 20, 25, 26], was developed in the past several years for representing artifacts created in object-oriented software process. The representation language is still in evolving, and has been widely accepted by the object technology community.

The UML is the successor to the wave of object-oriented analysis and design (OOA&D) methods that appeared in the late 1980s and early 1990s. The UML originates from the methods of Booch [6, 7], Rumbaugh [21, 22] and Jacobson [16, 17], but its reach is wider than that. The UML is called modeling language, not a method, because most method consists of both modeling language and a process. The modeling language is the notation that methods use to express designs, and the process is their advice on what steps to taken in doing a design. Rational Software et al. Released UML version 1.0 in January 1997 and version 1.1 in September 1997. In these versions UML mainly consists of two parts: Semantics and Notation Guide.
UML Semantics defines the rich semantics and expressive syntax of the UML. The UML is layered architecturally and organized by package. Within each package, the model elements are defined in terms of their abstract syntax (using the UML class diagram notation), well-formedness rules (using text and Object Constraint Language expressions), and semantics (using precise text). UML Semantics contains three kinds of packages:

- Foundation packages: core, auxiliary elements, extension mechanisms, and data types.
- Behavioral elements packages: common behavior, collaborations, use cases, and state machines.
- Model management package.

UML Notation Guide defines UML notation and provides supporting examples. The UML notation represents the graphic syntax for expressing the semantics described by the UML metamodel. The graphical notation and textual syntax are the most visible part of the UML (the outside view), used by humans and tools to model systems. These are representation of a user-level model, which is semantically an instance of the UML metamodel. The Notation Guide also summarizes the UML semantics, however, the UML Semantics contains the definitions. This guide contains the following standard diagrams:

- Use case diagram
- Class diagram
- Behavior diagrams:
  - Statechart diagram
• Activity diagram

• Interaction diagrams:
  • Sequence diagram
  • Collaboration diagram

• Implementation diagrams:
  • Component diagram
  • Deployment diagram

In this thesis, we will only use use case diagram, Class diagram, and Sequence diagram.

### 1.2.2 Use Case Diagram

Use case diagrams are part of the UML that is used to visualize use cases. In use case diagram,

• The standard stereotype icon for an **actor** is the “stick man” figure with the name of the actor below the figure

• A **use case** is shown as an ellipse containing the name of the use case

• The **communication relationship** between an actor and a use case is shown as a solid line between the actor and the use case

• An **extends** relationship between use cases is shown by a generalization arrow from the use case providing the extension to the base use case. The arrow is labeled with the stereotype <<extends>>

• An **uses** relationship between use cases is shown by a generalization arrow from the use case doing the use to the use case being used. The arrow is labeled with the stereotype <<uses>>
Figure 1 shows some use cases for a financial trading system. From Figure 1 we can see:

1. Actor Trader performs three use cases;
2. The same use case, Price Deal, is performed by two actors: Trader and Salesperson;
3. Trader and Salesperson may be played by one person or by two persons;
4. Both Analyze Risk and Price deal uses Valuation;

1.2.3 Class and Object Diagram
Class and Object Diagram are Static Structure Diagrams. Class diagram shows the static structure of the model, in particular, the things that exist, such as classes and types, their internal structure, and their relationship with other things. A class diagram is a graph of Classifier elements connected by their various static relationships. Classifier is the metamodel superclass of Class, DataType, and Interface. A class represents a concept within the system being modeled. Classes have data structure and behavior and relationships to other elements. A class is drawn as a solid-outline rectangle with three compartments separated by horizontal lines:

- The top name compartment holds the class name and other general properties of the class including stereotype
- The middle list compartment holds a list of attributes
- The bottom list compartment holds a list of operations

The simplest connection between classes is an association denoted with a solid line with or without multiplicity. Figure 2 is an example of class diagram. In this figure the multiplicity 1 ... * means that one HashTable class may connect with several String classes.

![Class Diagram](chart.png)

**Figure 2 Class diagram**
An Object represents a particular instance of a class. The object notation is derived from the class notation by underlining instance-level elements. An object is drawn as a solid-outline rectangle with two compartments separated by horizontal lines:

- The top compartment shows the name of the object and its class, all underlined, using the syntax:

  \[ \text{ObjectName : className} \quad \text{or} \quad \text{ObjectName} \quad \text{or} \quad : \text{className} \]

- The second compartment shows the attributes for the object and their values as a list. Each value line has the syntax:

  \[ \text{AttributeName : type = value} \]

Figure 3 is an example of a composite object. A composite object represents a high-level object made of tightly bound parts. In object aframe, there are four objects: fontControl, colorControl, surface, and title. The first two objects are instances of the same class Controller. The connection is labeled with "changes", which means that a operation of "changes" will be called.

1.2.4 Sequence Diagram

Sequence diagram is a subsection of Interaction diagram that is one of three kinds of Behavior diagram. Interaction diagrams are models that describe how groups of objects collaborate in some behavior. Typically, an interaction diagram captures the behavior of a single use case. The diagram shows a number of example objects and the messages that are passed between these objects within the use case. You should use interaction diagram when you want to look at the behavior of several objects within a single use case. If you
Figure 3 Composite object diagram

want to look at the behavior of a single object across many use cases, use a state transition diagram. If you want to look at the behavior of a many objects across many use cases or many threads, use an active diagram.

Interaction diagram has two subsections: sequence diagram and collaboration diagram. Sequence diagrams and collaboration diagrams express similar information, but in different ways. Sequence diagrams show the explicit sequence of messages and are better for real-time specification and for complex scenarios. Collaboration diagrams show the relationships among objects and are better for understanding all of the effects on a given object and for procedural design.

A sequence diagram represents an Interaction, which is a set of messages exchanged among objects within a collaboration to effect a desired operation or result. Within a sequence diagram, an object is shown as a box at the top of a dished vertical line. This
vertical line is called the object’s **lifeline**. The lifeline represents the object’s life during the interaction.

Message is represented by an arrow between the lifelines of two objects. The order in which these messages occur is shown top to bottom on the page. Each message is labeled at minimum with the message name; you can also include the arguments and some control information. You can show self-delegation, a message that an object sends to itself, by sending the message arrow back to the same lifeline.

Two kinds of control information are available in UML:

- **Condition**: indicating when a message is sent
- **Iteration marker**: showing that a message is sent many times to multiple receiver objects

Return is a special message, which indicates the return from a message, not a new message. Returns differ from the regular message in that the arrowhead for a return is a pair of lines and not solid.

Sequence diagrams are also valuable for concurrent processes by sending asynchronous message and using element **activation**. An asynchronous message can do one of the three things:

- Create a new thread, in which case it links to the top of an activation
- Create a new object
- Communicate with a thread that is already running

Figure 4 is a simple sequence diagram for switching, in which you may find objects, lifeline, message including condition, and activation.
1.3 Translation from Use Case to Sequence Diagram

1.3.1 Necessity of Translating Use Cases to Sequence Diagram

A use case is usually associated with a narrative description of an interaction process between actors and a system. Use case is useful in guiding software system development. But, a CASE tool cannot understand use case narrative descriptions. The difficulty in processing use case description is related to the vagueness of natural languages and the prose style of the narrative descriptions of most use cases. It cannot be directly translated into a representation language such as UML or a programming language such as Java.

Effort has been made in translating narrative description of software into executable code. Abbot proposed a technique for transforming informal but precise English description into source code in Ada [1]. The primary contribution of Abbot’s work in the proposed relationship between common nouns and data types. It identifies different types of noun and noun phrases as classes or objects. The key point of Abbot’s technique is that
the English description must be precise enough. The requirement makes it easy to translate the description into a program. However, it is difficult to satisfy the requirement to write English description. Consequently, this technique is useful, but not so flexible.

A software developer manually translates and refines use cases into class diagrams and behavior diagrams in the succeeding phases of a software process. The class and behavior diagrams will be eventually translated into code in a programming language by a CASE tool such as Rational Rose. But the problem of translating the narrative descriptions of use cases into the class and behavior diagrams mechanically remains open. This thesis focus on the issues related to the problem and presents a way to perform the desired translation. As the first step toward a solution to the problem, we shall focus on translating one use case into a sequence diagram.

1.3.2 Natural and Programming Languages

The desired mechanical translation is difficult mainly for the following three reasons.

- The vagueness of natural languages,
- The prosaic writing style of people and
- The rigorous syntactic requirement of representation and programming languages.

This section discusses the mismatch between natural languages, which are used to describe use cases and the formalism enforced by the programming languages such as Java and representation languages such as UML.

1.3.2.1 Vagueness of Natural Language

A natural language such as English has vagueness in expressions. This is an intrinsic attribute of natural language. We can almost do nothing to improve it. The only thing we can do is to make the descriptions of use cases as precise and intuitive as possible.
The following two sentences are a simple example of the vagueness of English.

- Because this is my pencil, I can cut it into two pieces without against the law.
- Because he is my friend, I can cut him into two pieces without against the law.

The same "my" has total different meanings in the two sentences. No parser may find the difference. Here is another example.

- She made me a sandwich.
- She made me a slave.

For the first sentence, we may say "she made a sandwich for me" instead of saying "made me a sandwich", but people often says this way.

1.3.2.2 Writing Habit and Style

Every one has his/her own writing habit and style. The followings are some examples.

1. The operator sends the manager a message.
2. The operator gives the manager a message.
3. The operator passes the manager a message.
4. The operator passes a message to the manager
5. The manager receives a message from the operator.
6. A message is passed by the operator to the manager.
7. The operator sends the manager some information about the price and amount of the item.
8. The operator sends the manager some information that is about the price and amount of the item.
9. The diligent operator quickly sends the smart manager an important message.
Sentence 1 through sentence 3 uses different verbs that have the same meaning in programming. Sentence 3 through sentence 6 uses different sentence structure, but have the same meaning. Sentence 7 has no attribute clause, but sentence 8 does. They have the same meaning. Sentence 1 and sentence 9 convey exactly the same meaning if they are encoded in a computer programming language. For a mechanical translator, the only difference between them is that the latter makes more trouble for the translation. Those different writing habits and styles make the translation difficult.

1.3.2.3 Syntactic Requirement of Programming Languages and Sequence Diagram

Each programming language has its own requirements on the syntax of expressions. Here we exemplify the syntactic requirement with object-oriented programming (OOP). In OOP, we often need to define classes, create objects, and send messages between objects. Message send is the basic action of OOP. It involves four elements: the source object (sender), the target object (receiver), the message name, and the arguments. The sender and the receiver of a message may be the same object.

Message send is the atomic unit of action to be performed by objects. We may also need some control over the execution sequence of actions invoked with messages. Generally speaking, there are three kinds of control information: condition, iteration and concurrency (or parallelism).

To represent message sends, UML sequence diagrams have notations for the four elements: message sender, receiver, message name, and parameter. They can present control information graphically too.

The narrative description of use cases must support those requirements.
1.4 Overview of the Thesis

1.4.1 The Objective of This Thesis

The Objective of this thesis is to give a solution for the three issues mentioned in section 1.3.2, and establish semi-automatic system for translating use case to sequence diagram.

To overcome the vagueness of natural languages, we shall present a set of syntactical patterns, which guide rewriting the prose narrative description of a use case into a normalized description. The normalized description will be translated to a sequence diagram.

From the original or normalized description of a use case, we need to identify objects, actions, and control structures. Their expressions in the use case description are not in the syntax of a representation or programming language. We need naming rules to construct names for the object, action, attributes and control structures.

To record the essential information encoded in a sequence diagram, we shall define message units to record the messages sending within an object or between two objects. In this thesis the terms of sequence diagram and message unit are interchangeable.

The mechanism of use case translation is implemented with a parser, which converts the normalized description of a use case into message units. The parser is supported with a catalog, which automatically records classes produced during parsing. The class records contain class name and related attributes. In addition to the automatically derived information, the catalog also records control information such as decision points and the phrases in the use case description that cannot be automatically translated to sequence
diagram. Therefore, the parser may not be an automatic translator. It replies on information entered by a user to resolve some confused parts in a use case description.

The proposed approach to mechanical translation of use case description is illustrated in Figure 5, which is a UML activity diagram. The input data for the diagram is the original narrative description, the output is Message Units.

![Diagram]

**Figure 5** Proposed approach to mechanical translation of use case
The architecture of the mechanical translation is illustrated in Fig. 6. It consists of a parser, a catalog, a dictionary, and a set of naming rules. The input is normalized descriptions of use case, and the output is a collection of Message Units.

1.4.2 Organization of the Thesis

This thesis is organized into four chapters: Chapter 1 gives a brief overview of the background knowledge involved in this thesis work, which includes use case, UML, the translation from use case to sequence diagram and an overview of the thesis. Chapter 2 defines syntactical patterns, naming rules, message unit, explains the relationship among them. Chapter 3 discusses catalog and parser, creates a parser in Java, parses the narrative descriptions of two use cases and draws sequence diagrams Chapter 4 highlights the conclusions along with the future work.

![Diagram of the architecture for translation of normalized use case](image)

Figure 6. The architecture for translation of normalized use case
Chapter 2  Syntactic Patterns and Naming Rules

Due to the various features of natural languages, it is impossible or very difficult to translate a narrative description of a use case into a sequence diagram or other UML diagrams. Our goal is to mechanically translate the normalized narrative description of use cases into their corresponding sequence diagrams. The normalization is manual by a software developer, who is guided by a set of syntactic patterns. Since the syntactic patterns are close to the sentence patterns of natural language, they can also be used to guide writing narrative descriptions of use cases.

This chapter focuses on the Syntactic Patterns, Naming Rules, Control Information and Message Units.

2.1 Basic Principles

2.1.1 The Principles

As mentioned in Chapter 1, the vagueness of English and the different writing styles make an automatic translation of the narrative descriptions of use cases impossible. We propose normalizing the narrative descriptions so that the generated descriptions of the use cases can be automatically translated to representations of sequence diagrams. The normalization depends on the following basic principles, which should be applied to guide developer to write or rewrite original narrative descriptions for a use case. The principles will be elaborated shortly.

Use simple sentence only; do not use subordinate clause if not definitely necessary.

Use active voice only; do not use passive voice if not definitely necessary.

Use the same verb instead of different verbs for the same meaning in the use cases of the same project.
4. Do not use adjective and ordinary adverb if not definitely necessary.

5. Do not use pronouns for an object name if not definitely necessary.

6. Do not use more than one predicate for one subject.

7. Use the syntactic patterns as defined in Table 1 only, do not use other complicate sentence structures if not definitely necessary.

8. The narrative description for each action must point out the source object, the target object, the action name and the arguments without any vagueness.

9. The control information must be clearly indicated in the narrative description of use cases.

2.1.2 Explanations on Normalization Principles

We now illustrate the basic normalization principles with examples and explanations.

**Principle 1:** Use simple sentence only; do not use subordinate clause if not definitely necessary.

Do not write as:

Customer gave $10 to the cashier who gave the customer $2 as change.

Write as:

Customer gave the cashier $10.

The cashier gave the customer $2 as change.

Comment:

A parser is responsible to identify the structures of sentences in a use case description. The principle makes identifying the structure of sentence pattern easier. In fact, using subordinate clause often invokes another interaction
between objects, just like the above example. We must separate the two
interactions into two sentences.

**Principle 2:** Use active voice only; do not use passive voice if not definitely necessary.

Do not write as:

$10 was given to the cashier by a customer.

Write as:

A customer gave the cashier $10.

Comment:

This principle follows the expression style of computer programming languages that each statement in a computer program expresses an action in an active but not passive voice. In the passive voice example, we may not know which object is the caller, and which object is being called.

**Principle 3:** Use the same verb instead of different verbs for the same meaning in the use cases of the same project.

Do not write as:

Customer sends the cashier $10.
Customer passes the cashier $20.
Customer gives the cashier $30.

Write as:

Customer sends the cashier $10.
Customer sends the cashier $20.
Customer sends the cashier $30.
Comment:

Take verb “send” as example. A newspaper may use different verbs such as “give” and “pass” for the same meaning in an article. When we are writing descriptions of use cases, only one verb “send” is enough. We shall explain how to relax this requirement for use case descriptions to satisfy human reading pleasures. It seems that the principle is friendlier for a machine than for a human. In OOPL (Object-Oriented Programming Language), the three sentences invoke the same function: send ( parameter ), the only difference is the value of the parameter. If we use different verbs, we will need different function to do the exactly the same thing.

**Principle 4:** Do not use adjective and ordinary adverb if not definitely necessary.

Do not write as:

The diligent operator quickly sends the smart manager an important message.

Write as:

The operator sends the manager a message.

Comment:

Informally speaking, computers and computer programs are cool blood animals. They do not have motion and are not moved by extra adjectives and adverbs. Formally speaking, the state of the art of AI or knowledge engineering has not reached the capability of encoding human intelligence into computer programs easily. The adjectives and adverbs only confuse an ordinary software CASE tool. When we abstract classes and functions from the original sentence, we may receive classes of DiligentOperator and
SmartManager, and function of quicklySend(). But the right things we need to receive are classes of Operator and Manager, and function of send().

About adjective: In certain situations, adjective is definitely necessary. For instance, we need adjectives in the following two sentences to describe the status of an object.

The stack is empty.

Sales Order creates a new item.

**Principle 5**: Do not use pronouns for object name if not definitely necessary.

Do not write as:

The customer gives the cashier item 1.

Later on, he gives him item 2.

Write sentence 2 as:

The customer gives the cashier item 1.

The customer gives the cashier item 2.

Comment:

Using pronoun brings additional work for the parser to find to whom or what the pronoun points. Also, we do not need "Later on", because the parser knows the order of actions by the order of sentences, not by "Later on".

**Principle 6**: Do not use more than one predicate for one subject:

Do not write as:

The cashier checks the price of an item and sends the manager the price of the item.

Write as:
The cashier checks the price of an item.

The cashier sends the manager the price of the item.

Comment:

This principle follows the fact that the syntactic patterns to be used in normalizing use case descriptions do not recognize sentences that contain two verbs. In fact, the original sentence invokes two actions which must be described in two sentences.

According to the discussion on special requirements of programming and sequence diagram, we may abstract out another two principles as follow. The two principles use the vague or fuzzy adjectives vagueness and clearly. They will be clear when we discuss the syntactic patterns later.

**Principle 7:** Use the syntactic patterns as defined in Table 1 only. Do not use other complicate sentence structures if not definitely necessary.

Do not write as:

The cashier received $10 from a customer.

Write as:

A customer gave the cashier $10.

Comment:

We have been careful in identifying a set of syntactic patterns that is expressive and flexible enough for writing use case descriptions and that is normal or formal enough to allow a computer program like the parser to automatically translate the use case descriptions. Table 1 contains the set of syntactic patterns that we have recognized.
Principle 8 and Principle 9 will be illustrated in the next section. They are built on the special requirements of programming and sequence diagram.

2.2 Syntactic Patterns

2.2.1 Syntactic Requirement of Programming Languages

Each programming language has its own requirements on the syntax of expressions. Here we exemplify the syntactic requirement with object-oriented programming language (OOPL). As mentioned in Section 1.3.2.3, In OOPL we often need to define classes, create objects, and send messages between objects. Message send is the basic action of OOP. It involves four elements: the source object (sender), the target object (receiver), the message name, and the arguments. The receiver of a message may be the same as the sender of the message.

Message send is the atomic unit of action to be performed by objects. We need some control over the execution sequence of actions invoked with messages. Generally speaking, there are three kinds of control information: condition, iteration and concurrency (or parallelism).

1. **Condition** indicates a situation under which a certain action may occur. One special condition is synchronization, which has a starting and an ending point. The starting point indicates that a number of activities may proceed in parallel at the same time. The ending point indicates that only after all of the activities are completed, a following activity can occur. Another special condition is a decision point, which switches among different activities based on the truth of a condition.

2. **Iteration** means that a message is sent multiple times to an object, as would happen when we are iterating through a collection.
3. **Concurrency** or **parallelism** means that an object can send asynchronous messages at the same time to several active objects. An asynchronous message does not block the sender. The message sender carries on its processing without waiting for a response from the receiver.

This thesis only contains simple condition and iteration.

![Sequence Diagram](image)

Figure 7 Message send in sequence diagram
a: message send within the same object
b: message send between two different objects

### 2.2.2 Notation Requirement of Sequence Diagrams

UML sequence and collaboration diagrams have notations for the four elements mentioned in Section 2.3.1 to represent message sends. They can present control information graphically as well. We can recognize two kinds of basic message sending style (**MSS**) in sequence diagrams. The first message sending style depicts that an object
sends a message to itself. The second depicts that the message sender and receiver are
different objects. The two kinds of message sending style are illustrated in Figure 7.

In Figure 7(a), a message is sent from anObject to anObject itself, **self-delegation**.
The message sending style in Figure 7(b) differs from Figure 7(a) in that an object named
source sends a message to another object named target. In design level or implementation
level of software development, the sender source invokes a function or method with name
"message" that is defined in the class of Target.

Condition is shown in Figure 8. Only under the condition that [check = true] returns
true, the message in this diagram is sent to the target.

![Sequence Diagram](image)

**Figure 8** Condition in sequence diagram

Iteration is shown in Figure 9. The star * in front of the message name indicates that
the source sends message to each target and each target sends message to anotherObject.

**2.2.3 Patterns**

**2.2.3.1 Patterns Definition**
Although each person has his/her own writing style, an English sentence has intrinsic structure that can be used as a good guide to find a matching syntactic pattern. The Syntactic patterns will be used as writing guides for the narrative description of use cases or as normalizing patterns to transform original descriptions of use cases.

Use case may be used on different levels in software development. The top level is specification level, which emphasis "who does what with whom". On this level the message passed between two objects only describes what needs to be done. This message is not a concrete function or method in OOPL. The bottom level is implementation level, which focuses on "How to do". On implementation level the message passed between object 1 and object 2 is the name of a concrete function or method. This means that object 1 invokes the method on object 2. Analysis and design levels are just between them, therefore, the message has the meaning between them too. The syntactic patterns defined in Table 1 cover the two levels. Although they are simple, they can satisfy a wide range of requirements from use cases and satisfy the requirements of OOPL and sequence diagrams.

To simplify Parser, which will parse these patterns in Table 1, we suggest that only use present tense for all verbs in Table 1, do not use past or future tense. For example, for verb "be", only use "is" or "are", not "was", "were" or "will be"; for verb "send" only use "send" or "sends". In most case, preset tense is enough to describe use cases. By doing this way, the Verb Dictionary will be greatly simplified. We will discuss Verb Dictionary in Chapter 3.

We separate the patterns in Table 1 into type A and type B. The patterns in type A describe sequence diagram shown in Figure 7(a), in which the source and the target are
the same object, self-delegation. The patterns in type B describe sequence diagram shown
in Figure 7(b), in which the source and the target are two different objects.

![Sequence Diagram](image)

**Figure 9** Iteration in sequence diagram

**Table 1** Syntactic patterns for narrative description of use case

<table>
<thead>
<tr>
<th>Type</th>
<th>No.</th>
<th>Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>Subject verb</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>Subject verb object</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>Subject be verb in passive voice</td>
</tr>
<tr>
<td>A</td>
<td>4</td>
<td>Subject be adjective</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>Subject verb a new noun</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>Subject destroy object</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>Subject verb direct object prep. indirect object</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>Subject verb object to verb</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>Subject verb object1 to verb object2</td>
</tr>
</tbody>
</table>

**2.2.3.2 Matching Patterns with Sequence Diagram**

The writing principles and syntactic patterns are used to guide writing narrative
descriptions of use cases that can be recognized and translated by the parser to sequence
diagrams. We need to match the syntactic patterns with the message send styles of
sequence diagrams and object-oriented programs. We also need to find the four elements mentioned in section 2.2.1: source object, target object, message name (action) and arguments. Before Naming Rules for Message Name are defined, we use the term of "action" instead of "message name". By comparing the patterns shown in Table 1 and message send styles, we may identify some correspondence between components in the patterns and in sequence diagram. The correspondence is described as follows.

1. Source Object

   • In both Type A and Type B, the Subject in a pattern represents the source object of a message send in a sequence diagram.

2. Target Object - only in Type B

   • No. 1, the noun in the pattern represents the target object in a sequence diagram.

   • No. 2, the object in the pattern represents the target object in a sequence diagram.

   • No. 3, the indirect object in the pattern represents the target object in a sequence diagram.

   • No. 4, the object in the pattern represents the target object in a sequence diagram.

   • No. 5, the object1 in the pattern represents the target object in a sequence diagram.

3. Action

   • Type A

     • No. 1, the verb represents the action in a sequence diagram.
• No. 2, the verb represents the action in a sequence diagram. If the verb is "check", and the object has the style like "if is empty", the action will be "is empty".

• No. 3, the part after "Subject" represents the action in a sequence diagram.

• No. 4, the verb + adjective represents the action in a sequence diagram.

• Type B

• No. 1, the adjective "new" represents the action in a sequence diagram. This is special for creating a new object.

• No. 2, the verb "destroy" represents the action in a sequence diagram. This is special for destroying an object.

• No. 3, the verb represents the action in a sequence diagram. If the verb is "check(s)", the action may be treated specially.

• No. 4, the first verb should be "tell(s)" or "ask(s)", the second verb represents the action in a sequence diagram.

• No. 5, the first verb should be "tell(s)" or "ask(s)", the second verb represents the action in a sequence diagram.

4. Arguments

• Type A

• No. 1, no argument, or argument is null.

• No. 2, the object represents the argument in a sequence diagram. If the verb in this pattern is "get", the argument is null.

Comment: In OOPL, like Java and C++, function get() only returns a value, does not have argument.
• No. 3, no argument, or argument is null.
• No. 4, no argument, or argument is null.

Type B
• No. 1, no argument, or argument is null.
• No. 2, no argument, or argument is null.
• No. 3, the direct object represents the argument in a sequence diagram. If the verb is "send(s)" or "check(s)", the arguments need to be treated specially.
• No. 4, no argument, or argument is null.
• No. 5, the direct object represents the argument in a sequence diagram. If the verb is "send(s)" or "check(s)", the arguments need to be treated specially. If the verb in this pattern is "get", the argument is null.

Comment: In OOPL, like Java and C++, function get() only returns a value, does not have argument.

2.2.3.3 Examples

This section gives the detail of the patterns through some examples. Before defining Naming Rules, we will not use the term of "Massage name", instead, use the term of "Action".

1. Type A

In Type A, the Subject is both the source object and target object.

• No.1, has no arguments

Example 1: Controller exits

Source object: Controller
Target object: Controller
Action: exits
Argument: null

- No. 2, has arguments

**Example 1:** Checker checks price and quantity

Source object: Checker
Target object: Checker
Action: checks
Arguments: price, quantity

**Example 2:** Checker checks information about price and quantity

Source object: Checker
Target object: Checker
Action: checks
Arguments: price, quantity

**Example 3:** Checker checks if is empty.

Source object: Checker
Target object: Checker
Action: is empty
Arguments: null

**Comment:** special for "check if"

**Example 4:** Checker gets name.

Source object: Checker
Target object: Checker
Action: get

Arguments: null

• No. 3, has no arguments

**Example 1:** Transaction is done

Source object: Transaction

Target object: Transaction

Action: is done

Argument: null

• No. 4, has no arguments

**Example 1:** Stack is empty

Source object: Stack

Target object: Stack

Action: is empty

Arguments: null

2. Type B

In Type B, the source object and the target object are different objects. The Subject in the pattern is always the source object.

• No. 1, has no argument

**Example 1:** Salesperson creates a new order.

Source object: Salesperson

Target object: order

Action: new

Arguments: null
Example 2: Salesperson opens (chooses, selects) a new order.

Source object: Salesperson
Target object: order
Action: new
Arguments: null

- No. 2, has no argument

Example 1: Salesperson destroys order.

Source object: Salesperson
Target object: order
Action: destroys
Arguments: null

- No. 3, has arguments

Example 1: Salesperson checks price and availability with order.

Source object: Salesperson
Target object: order
Action: checks
Arguments: price, availability

Example 2: Salesperson checks information about price and availability with order.

Source object: Salesperson
Target object: order
Action: checks
Arguments: price, availability
Comment: To simplify Parser, we suggest do not use

Object 1 checks arguments with object 2
only use

Object 1 tells object 2 to check arguments

For instance, instead of saying:

Salesperson checks price with order
we say

Salesperson tells order to check price

Example 3: Salesperson checks if is ready with order.

Source object: Salesperson
Target object: order
Action: is ready
Arguments: null

Example 4: Salesperson sends price to order.

Source object: Salesperson
Target object: order
Action: sends
Arguments: price

Example 5: Salesperson sends information about price and availability to order.

Source object: Salesperson
Target object: order
Action: sends
Arguments: price, availability

**Example 6**: Salesperson sends "prepare" message to order.

Source object: Salesperson

Target object: order

Action: sends

Arguments: "prepare"

**Comment**: To simplify Parser, we only use `send(s)` for this type. For instance, we do not say:

Salesperson **submits** information about price and availability **to** order.

Only say:

Salesperson **sends** information about price and availability **to** order.

- No. 4, has no argument

  **Example 1**: Salesperson tells order to prepare.

  Source object: Salesperson

  Target object: order

  Action: prepare

  Arguments: null

  **Example 2**: Salesperson asks order to prepare.

  Source object: Salesperson

  Target object: order

  Action: prepare

  Arguments: null
**Comment:** To simplify Parser, we suggest only use **send(s)** - **to**, not use **ask(s) - to** or other similar verb for Type B, No. 4 and No. 5. Therefore, we should never use the description as Example 2.

- No. 5, has argument

**Example 1:** Salesperson tells order to check price.

Source object: Salesperson
Target object: order
Action: check
Arguments: price

**Example 2:** Salesperson tells order to **get** price.

Source object: Salesperson
Target object: order
Action: get
Arguments: null

### 2.3 Naming Rules

While translating a narrative description into a sequence diagram, we must name the source object, target object, action (message name) and arguments. Also, we need to name the classes, which will be stored in **catalog**.

#### 2.3.1 Source Object and Target Object

The following is Naming Rules for Source Object and Target Object

1. Object name is the concatenation of the words in the object part of a sentence.
2. Any article is ignored. This means we do not distinguish "the", "a" and "an".
3. The first letter of each word in the concatenation must be capitalized except the 
   first word.
4. The first letter of the first word in the concatenation must be in lowercase.

Example 1: A sales department creates a new checker.
Source object: salesDeparment
Target object: checker

Example 2: The sale department destroys the checker.
Source object: saleDeparment
Target object: checker

2.3.2 Source Class and Target Class

In OOPL, any object is an instance of a class. The names of classes will be stored in a 
catalog. In sequence diagram, we often need class names to point out from which class an 
object is created. In the example item:Item, "item" is an object which is an instance of 
class "Item". The following is Naming Rules for Source Class and Target Class.

1. Rules 1 - 3 for naming source and target object are effective for naming source 
   class and target class.

2. The first letter of the first word in the concatenation must be capitalized.

Example 1: A sales department creates a new checker.
Source class: SalesDeparment
Target class: Checker

2.3.3 Message Name

In section 2.2.3 we use term of "action" instead of "message name", because we 
have not defined the naming rules for action at that time. "Action" points out what will
occur between the related objects; "message name" is the name of the action. The message name will appear in the sequence diagram. The following is Naming Rules for Message Name.

1. Message name is the concatenation of the words in the action part in a sentence;
2. Rules 2 - 4 for naming source and target object are effective for naming messages.
3. Special case for verb "get" and "set":

   If the verb in Type A, No.2 or the second verb in Type B, No.5 is "get" or "set", the concatenation of "get"/"set" and the followed object names the message.

Comment:

- In OOPL, like Java and C++, we often name access functions as
  getForObject(), setObject( object ).

- In this case the object of the sentence often contains only one unit like "price" or "customer name", not "price and customer name".

Example 1: A sale department creates a new checker.

Message Name: new

Example 2: A sales department destroys checker.

Message Name: destroy

Example 3: Salesperson gets name.

Message Name: getName

Example 4: Salesperson sets name.

Message Name: setName

Example 5: Salesperson check if is empty.

Message Name: isEmpty
Example 6: Sales order sends item to item ordered.
Message Name: send

Example 7: Sales order sends information about price and item to item ordered.
Message Name: send

Example 8: Sales order sends "is ready" message to item ordered.
Message Name: send

2.3.4 Argument

The following is Naming Rules for arguments.

1. Argument is the concatenation of the words in each part of the argument part in a sentence.

2. Rules 2 - 4 for naming source and target object are effective for naming argument.

3. If the argument is a string, the string must be capsulated with a pair of double quotation marks.

4. Special case for verb "get":
   If the verb in Type A, No.2 or the second verb in Type B, No.5 is "get", the argument for this action is null.

Comment:

- In OOPL, like Java and C++, we often name get function as getobject(), and does not put the object into the argument position.

- In this case the object of the sentence often contains only one unit like "price" or "customer name", not "price and customer name".

Example 1: A sale department creates a new checker.

Arguments: null
Example 2: A sale department destroys checker.
Arguments: null

Example 3: Salesperson gets customer name.
Arguments: null

Example 4: Salesperson sets customer name.
Arguments: customerName

Example 5: Salesperson checks name, customer address and credit limit.
Arguments: name, customerName, creditLimit

Example 6: Sales order sends item to item ordered.
Arguments: item

Example 7: Sales order sends information about price and item to item ordered.
Arguments: price, item

Example 8: Sales order sends "is ready" message to item ordered.
Arguments: "is ready"

2.4 Control Information

As mentioned in section 2.2.1 message send needs to contain control information: condition, iteration and concurrency (or parallelism). Condition can be synchronization or decision point. This thesis only covers simple condition (if-else phrase) and iteration.

2.4.1 Condition (if-else phrase)

If-else phrase may have any number of layers. To find a way for showing condition, let us consider the example shown bellow.

If [condition 1] {

    Object 1 send argument 1 to object 2

}
Object 2 send argument 2 to object 3

If [condition 2] {
    Object 3 send argument 3 to object 4
    If [condition 3]{
        Object 4 send argument 4 to object 5
    }Else
        Object 4 exits
    } Else
        Object 1 exists
}

Figure 10 If_Else phrase in sequence diagram
The sequence diagram of this example is shown in Figure 10. From this figure it can be seen that we only need to pay attention to the actions that most close to condition (if or else). There are two reasons to guarantee the logical execution:

1. The sequence diagram itself is a sequence, which shows the order of execution.
2. Each message contains source and target object, which guarantee the matching of "if" and "else". In other words, we do not match "if" and "else" by counting them or by counting the corresponding "{" and "}", we match them by the sender name.

Therefore, the rule for showing condition may defined as

1. Use if-else phrase only;
2. The "if" phrase must be positioned at the very beginning of a sentence.
3. The "else" phrase must be positioned at the very beginning of a sentence.
4. Show the condition in a pair of square brackets.
5. Show the action statement in a pair of braces.
6. The "if" phrase must be expressed as: if [condition] { the most closed message send }.
7. The else phrase must be expressed as: else { the most closed message send }.

Accounting to this rule, we may rewrite the example given above as

1. If [condition 1] {Object 1 send argument 1 to object 2}.
2. Object 2 send argument 2 to object 3.
3. If [condition 2] {Object 3 send argument 3 to object 4}.
4. If [condition 3]{Object 4 send argument 4 to object 5}.
5. Else {Object 4 exits}.
6. Else {Object 1 exists}
The example in Section 3.3.2 demonstrate that if we describe control information by these rules, our parser may successfully parse them, even there are some nested loops.

The last question for this section is the condition itself: how to write condition expression. The answer depends on the level at which you are building your use case. If we are building a use case of high level, for example specification or analysis level, we may write the condition in English, say, [the stock is already empty]. If it is a low level use case, especially at implementation level, we must use Boolean expression of the language we are using, say, Java. The condition may like \[ a \text{Variable} = 3 \], or \[ a > 2 \&\& b < 3 \].

2.4.2 Iteration

Iteration means a message is sent multiple times to an object, as would happen when we are iterating through a collection. To find a way showing iteration, let us consider an example first.

1. Object 1 sends "message 1" message to each object 2.
2. Each object 2 sends "message 2" message to object 3.
3. Object 3 sends "message 3" message to object 4.

This sequence diagram of this example is shown in Figure 11. From this figure it can be seen that, just like condition control, we only need to mark the actions that starts the iteration. Therefore, we define the rule showing iteration as the following.

1. The expression style must be:
   \[ \text{[For each anObject]} \{ a \text{ message send sentence } \} \]

2. In the sentence encapsulated within braces, the target object must exactly match "anObject"
In this rule, we encapsulate "For each ..." in a pair of square brackets to make parsing easy. Now we may rewrite the example as:

1. [For each object 2 ] {object 1 sends "message 1" message to object 2.}
2. object 2 sends "message 2" message to object 3.
3. object 3 sends "message 3" message to object 4.

It is important to point out that there is some mistake in Sentence 1. In fact we do not send message to each object 2. We send message to each instance of same class. Each object 2 is not the same object, they are different instances of same class, but with same object name. To avoid this mistake, we should name the object 2 in sequence diagram as :Object 2. This way give the class name, from which each object is instantiated. We may write the descriptions as we defined, every one will understands it, therefore, it is clear enough.
After constructing the sequence diagram like Figure 11, we may use rational Rose to process the iteration. Rational Rose will create a thread for each new instance and process it. The iteration will be finished only when all the threads are finished or an "exit" is shown. Since this thesis only covers the translation from narrative descriptions of use case to sequence diagram, we will not discuss the details of Rational Rose.

Condition and iteration are summarized in Table 2.

<table>
<thead>
<tr>
<th><strong>Condition</strong></th>
<th>If [condition] {the most closed message send} Else {the most closed message send}</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Iteration</strong></td>
<td>[For each anObject] {a message send sentence}</td>
</tr>
</tbody>
</table>

### 2.5 Message Unit

Message Unit, translated from narrative descriptions of use cases, is established to record a message sending within an object or between two objects. To be able to translate into Message Units, the narrative descriptions must be written or rewritten as consistent with the Syntactic Patterns defined in section 2.2. Message Unit is an object which consists of five parts: **controlInfo**, **sourceObject**, **targetObject**, **messageName**, and Vector **argVector**. Each time we read a normalized sentence of the narrative descriptions, a Message Unit is created by a parser in cooperation with Naming Rules. The collection of Message Units is the formal description of a sequence diagram of a use case, in other words, Message Units are equivalent to sequence diagrams.

#### 2.5.1 String controlInfo
controlInfo is a string that supports the control information for an action to occur. As mentioned early, this thesis only involves the condition (if-then phrase) and iteration (multiple receiver objects).

For condition (if-then phrase), the control information takes the same format as condition expression, for example,

"if [quantity > 5]",

"if [check = "true"], or

"else".

Those strings will be recorded in a catalog (if necessary), and be drawn in a sequence diagram.

For iteration, the control information string takes the format as

"For each order line",

"For each checker".

Those strings will be recorded in a catalog (if necessary), too. But when we draw sequence diagram, we use a symbol "*" instead of the string as shown in Figure 11.

2.5.2 Vector argVector

When we send message within an object or between two objects, we often need to pass some arguments. For example, in the sentence

1. Item sets price.

"price" is a argument, which is passed within the object "item". Another example,

2. Object 1 sends name and price to object 2.

Both "name" and "price" are argument, which are passed between "object 1" and "object 2".

49
3. Checker exists.

Considering these sentences, we may find:

- We may have any number of arguments. In sentence 1, we have one argument; in sentence 2, we have two, and in sentence 3, we have no argument.

- The arguments may have any data type. "name" is a String, "price" is a Double.

To store those arguments, we need a suitable holder, which may store any number of elements, and any type of elements. Vector is the best choice. The type of element stored in a vector is Object (Java), and we may cast any other type into Object. We name the vector as argVector to notify that this vector is used to store arguments.

This vector takes the following style:

\[ \text{argVector(} \text{arg0, arg1, \ldots, argn} \text{)} \]

where \( \text{argi} ( i = 0 \text{ to } n ) \) is argument name. For sentence 1, we have

\[ \text{argVector( price)} \]

for sentence 2,

\[ \text{argVector(name, price)} \]

for sentence 3

\[ \text{argVector()}. \]

2.5.3 Object messageUnit

Object messageUnit is a complete record of a message sending. This object has five parameters and takes the following styles:

\[ \text{messageUnit(} \text{sourceObject, targetObject, messageName, argVector, controlInfo} \text{)} \]

where
sourceObject is a string, the source object name, which is in consistence with Naming Rules for object;

targetObject is a string, the target object name, which is in consistence with Naming Rules for object;

messageName is a string, the message name, which is in consistence with Naming Rules for message name;

argVector is a vector containing arguments;

controlInfo is a string containing control information for an action to occur.

An object messageUnit is an equivalent of a unit in sequence diagram. This object clearly tell us from which object draw an event arrow to which object, and how to label the event arrow. The messageUnit works as blocks, with which we may build a whole sequence diagram.

2.5.4 Examples

For each of the following examples, give

- A sentence written in consistence with syntactic patterns;
- String controlInfo, Vector argvector and Object messageUnit;
- A unit of sequence diagram, which represents this sentence.

Example 1

Sentence: Cashier sends information about price, quantity and date to Controller

controlInfo = ""

argVector( price, quantity, date)

messageUnit( cashier, controller, sends, argVector, controlInfo )

The corresponding unit of sequence diagram: Figure 12.
Figure 12 Unit of sequence diagram, example 1

Example 2

Sentence: If [stack is not empty] {stack pops}

ccontrollInfo = if [stack is not empty]

argVector( )

messageUnit( stack, stack, pop, argVector, controllInfo )

The corresponding unit of sequence diagram: Figure 13.

Figure 13 Unit of sequence diagram, example 2
Example 3

Sentence: [For each object 2 ] {object 1 sends "message 1" message to object 2.}

controlInfo = [For each object 2 ]
argVector("message 1")
messageUnit(object 1, object 2, send, argVector, controlInfo )

The corresponding unit of sequence diagram: Figure 14.

Two concrete examples will be shown in Chapter 3. Each of those two examples describes a use case. In those two examples, we will demonstrate that our parser may successfully parse normalized descriptions, each sentence will generate a messageUnit, and that the control information will control the flow of sequence diagram.

Figure 14 Unit of sequence diagram, example 3
Chapter 3 Parser and Catalog

Chapter 2 focuses on how to write normalized descriptions of use cases - writing all sentences in consistent with the patterns in Table 1. This chapter focuses on how to process those normalized narrative descriptions - creating a parser and parsing the narrative descriptions. The parser reads each sentence; generates Message Units, which are the equivalent of a unit of sequence diagram. The parser will work together with the Naming Rules defined in Chapter 2 and a dictionary of verbs. The classes and their attributes will be stored in a catalog. The catalog will also store control information and special description, if necessary.

This chapter discusses parser and catalog, creates parser program, parses two use cases and gives the result diagrams.

3.1 Parser

Parser is a program coded in a programming language, such as Java, C, C++. The program takes the normalized narrative description of a use case as its input. Parser reads the narrative description sentence by sentence. From each sentence Parser finds out the source object, the target object, the action, the arguments and the control information. The output of parsing is a collection of Message Units, which is the equivalent of a sequence diagram, and a catalog, which records the parsing results and some special information.

3.1.1 Libraries

While reading a sentence, the parser will find out the source object, the target object, the action, the arguments and the control information. The key action of parsing is checking verbs. After finding out a verb in the sentence, the parser will know where
target object is located, and if there are arguments in the sentence. For example, parser reads the following description:

1. Cashier sends customer name to Controller.

First parser finds a verb - "sends", then it knows that the next word must be arguments or part of them, and that there must be a proposition "to", and the word following this "to" must be the target object or a part of it. Another example:

2. Stack is empty.

After finding the verb "is", parser knows that this is a self-delegation description. i.e., the source object and the target object are the same object.

To check verbs, parser needs a Verb Dictionary. Parser checks each word in the sentence with Verb Dictionary to determine if it is a verb. We may either use existing Verb Dictionary provided by some software, or create our own. The following is the steps for creating Verb Dictionary by hands:

1. Create a Verb file;
2. Read the narrative description, find a verb;
3. If the verb is already in the file, skip it and continue reading;
4. Else put it into the file; and continue reading;
5. Read until the end of the whole description.

This is the simplest way for creating Verb Dictionary, and also an effective way. We may create the dictionary while write each sentence. The size of the dictionary will be very small, may be only several ten words. Of course, the smaller the size is, the faster the parser works.
**Preposition Dictionary** is another useful dictionary for parsing. The preposition, say "to" or "with", found in reading a sentence points out that its object is the target object of this description. In the above example, sentence 1, the word in behind of the preposition "to" is the target object - controller. We may establish Preposition Dictionary by the same way and at the same time as we create Verb Dictionary. This dictionary may only contain several words, but it will be big enough for parsing almost all descriptions.

**Adjective Dictionary** is also a necessary dictionary for parsing pattern of Type A, No. 4, which contains an adjective. We may establish Adjective Dictionary by the same way and at the same time as we create Verb Dictionary and Preposition Dictionary.

### 3.1.2 Generating Message Units with Naming Rules

The output of parsing process is a collection of Message Units. A Message Unit contains String controlInfo, sourceObject, targetObject and messageName, and Vector argVector. This means that to generate a message unit, the parser needs to find out all objects, arguments, action and control information, and name them with help of Naming Rules. Also, the parser needs to name the classes from which the source objects and target objects are created. To do so, we build the architecture of the parser program as follow.

1. Class Driver containing methods main(), checkTypes(), makeUnit() and printUnit()
   - main():
     - Use FileReader (Java) to read file;
     - Use while loop to read each line (a sentence);
1. Use StringTokenizer to separate the line into controlInfo part and the rest part;
   - Put controlInfo into String controlInfo, pass the rest part to checkType();
   - checkType():
     - Find Type A or B by checking special words such as "new", "destroy", "send" and "tell";
     - Find Type No. by invoking checkAs(), checkBs(), checkVerb() and checkAdjective();
   - makeUnit():
     - Pass the rest part to a pattern class to which it belongs, let the corresponding pattern class make messageUnit
   - printUnit():
     - Print the unit;

2. Patterns class For each pattern we have a class, for example class B3 for pattern Type B, No. 3. The method makeUnit() of those classes finds source part, target part and action part, and send them to class NameMaker to name them. This method also finds argument part and sends it to class Argmaker to make Vector argVector.

3. Class NameMaker names sourceObject, targetObject, messageName and arguments according to Naming Rules.

4. Class ArgMaker finds each argument in the passed argument part, sends them to NameMaker to name them, and puts them into a vector.
5. Class Catalog is a Hashtable. The key of the hashtable contains class name; while its value is a Vector containing the attributes of the class (for the simplest situation).

We implemented the parser in Java and listed it in Appendix A.

Let us look at the following example

1. Order sends item name and price to order line.

Driver class reads this sentence, finds the verb "send", and passes it to class B3 (pattern: Type B, No. 3). By invoking makeName() on an object of NameMaker, B3 finds and names:

- Control information: null
- Source object: order
- Target object: orderLine
- Message name: send
- Arguments: itemName, price

B3 records these data into messageUnit of this sentence. B3 also generates class Order and class OrderLine, records them into the catalog of this use case. Meantime, B3 puts orderLine into Order class, and puts itemName and price into both Order class and OrderLine class.

For a description with control information, Driver class check "if [condition]" or "[For each anObject] " first, records the control information into String controlInfo, and passes the sentence capsulated within the pair of braces into corresponding pattern class. The following is an example. For the sentence

1. [For each order line] { Order sends item name and price to order line}
Driver class records "for each order line" in String controlInfo, and passes the followed sentence into class B3.

3.2 Catalog

As mentioned in section 3.1, parsing the description of a use case generates a collection of Message Units. Besides this, the parser creates classes, which contain class name, attribute and actions. Therefore, we need a Catalog to record the generated classes. Current syntactic patterns defined in this thesis are not a complete set of patterns, especially for control information. The parser cannot parse the undefined patterns, which need to be stored in the catalog. During parsing, the parser may meet some ambiguous or vague descriptions, and some special text, which cannot be automatically translated to a message Unit. The catalog needs to record those descriptions, too. As a conclusion, we may say that Catalog is semi-automatically established during parsing normalized narrative descriptions of use cases. Generally speaking, Catalog mainly record the information about classes, it also records special control information and ambiguous or vague descriptions and special text.

3.2.1 Classes

A use case may involve a number of objects (source and target objects), each of which is an instance of a class. A class contains class name, attributes, actions and even some relationship with other classes, for example "extends" or "implements" in Java. This section only covers class name, attributes and actions.

3.2.1.1 Class Name

During parsing, the parser use the following steps to generate class names and record them into Catalog.
1. Read an action description, in which there must be one or two objects;

2. Find the object parts according to the corresponding syntactic pattern;

3. Name each object according to Naming Rule;

4. Change the first letter of each object name to capital letter to name its corresponding class;

5. Check the Catalog to see whether or not the class name is already in the Catalog, if it is, skip this class name; else add it into the Catalog;

6. Go to the next description and do step 1 – 5 until all action descriptions are parsed.

3.2.1.2 Attributes

A class may define attributes. An attribute belongs to a certain class. In one sentence of the descriptions of a use case, we may find some arguments. Those arguments are the attributes of their related classes. The question is which class, source class or target class, is their enclosing class. Our answer to this question is both, both the source class and the target class. To understand this issue, look the following example first.

1. Checker sends item to Controller.

To be able to send "item" to Controller, the Checker must have a "item" already. This means that class Checker has an attribute named "item". Similarly, to be able to receive "item" from Checker, Controller must have a holder to hold the "item". This means that class Controller has an attribute named "item" too. If we implement this sentence in Java, we may use access method of set ( type arg ) as the following.

**Example Checker-Controller**

```java
public class Checker{
```
```java
AType item;

item = someValue;

...

Controller controller = new Controller();

Controller.set( item );

...

}

public class Controller {

    AType item;

    ...

    public Controller() {

        ...

    }

    ...

    public void set( AType anItem ) {

        this.item = anItem;

    }

    ...

}

Obviously, "item", the argument in sentence 1, is an attribute of both classes Checker and Controller.
Since arguments are derived from the narrative description, it is impossible to determine their types. The attribute types can only be decided when implement the program.

Since a source object needs to talk to a target object, the source must know the target. This means the target object is an attribute of the source class. From the example given above, we know that the object "controller", which is an instance of class Controller, is an attribute of class Checker.

During parsing, the parser uses the following steps to add attributes into Catalog.

1. Read an action description;
2. find the arguments imbedded in the description, and name them according to the Naming Rules;
3. Find the source object and the target object in the description and name them;
4. Name the corresponding source classes and target class;
5. Check with Catalog, find whether or not the first argument is already in the two classes, if it is, skip this argument; otherwise, add it to the two classes; and check the next argument, until all arguments are added or skipped;
6. Check with Catalog, find whether or not the target object is already in the source class, if it is, skip it, otherwise, add it to the source class;
7. Go to the next description and do step 1 – 6 until all action descriptions are parsed.

3.2.1.3 Actions

A class defines its actions (function in C++, or method in Java). The messageName in a Message Unit may either be the actual method name in Java, or only a message passed
within an object or between two objects. In the latter case, one or several methods must be imbedded in the message. This depends on at what level you are building your use case.

At high level, for example specification or analysis level, the message contains one or several methods, but before implementing the use case, no one knows what those methods are. Therefore, it is impossible to add actions into a class recorded in the catalog while parser parses the action description of a high level use case. In fact, we mainly design use case at this level, so it is not necessary to record those messages in catalog.

On the other hand, at low level, say implementation level, the messageName is the name of a concrete method, we need to record it into the catalog. For example, we change sentence 1 in section 3.2.1.2 to:

2. Checker tells Controller to set item.

This sentence means that an object of class Checker invokes method set() on an object of class Controller. Example Checker-Controller of section 3.2.1.2 implements this description. In this case, we must add method set into class Controller in the catalog. We may use similar steps to add methods into catalog as we add attributes. But we only add method into class Controller, not into both.

Since this thesis mainly focuses on high level use case, we will not add actions into catalog.

3.2.2 Control Information

Some control information, such as condition (if-else phrase) and iteration, is already defined in Chapter 2. Parser can easily catch it and put it in Message Unit. If necessary, this information can be put into catalog too. Some other Control information, such as
Synchronization, Decision Point and Concurrency (Asynchrony), is often complicate and is not easy to be normalized. It is difficult to translate such non-normalized control information into formal expression of sequence diagram automatically. Therefore, it is suggested that those complicate ones should be stored in the Catalog, and rewrite it into a formal expression in some way by hand. To be able to easily find the original sentence and the rewritten condition expression, we emphasize that

- Those special sentences in the normalized narrative description must be numbered, and the rewritten sentences must have the same number in the catalog.

The following example describes how to make coffee. This is an example of synchronization.

1. Find beverage
   2.1 Put coffee in filter
   2.2 Add water to reservoir
   2.3 Get cups

3. Pour Coffee

These three activities numbered 2.1 through 2.3 can occur in parallel. This means that their order is irrelevant. I could put coffee in the filter first, then add water to the reservoir, and then get cups. Or, I could get the cups, then add the coffee to the filter. I can also do these activities by interleaving. I could get a cup, add some water to the reservoir, get another cup, get some more water, and so forth. Or I could do some of this simultaneously: pour the water in with one hand while I reach for a cup with another hand. Any of these is correct, according to the description of this use case. In this case the
best way to deal with these descriptions is to record them into a catalog, and draw a Activity Diagram instead of Sequence Diagram.

3.2.3 Ambiguous/Vague/Special Description

As any other language, English has its own ambiguousness and vagueness. If an action description is written in an ambiguous or vague style, it is difficult or even impossible to automatically translate it to a Message Unit. Catalog must record this kind of descriptions.

To describe some special action, sometimes we need to use more complicate sentence structure that is not covered by our Syntactic Patterns. For those special descriptions, we need to record them in the Catalog, and translate them into formal Message Unit by hand, if possible.

It is worth emphasized that our Syntactic Patterns are not completed ones. Those patterns cannot cover all the sentences written by any person with his own habit. I even believe that no one can create a complete syntactic patterns system that can cover any sentence; I also believe that it is not necessary to create such system. By using the Catalog, we give programmers more flexibility, and may perform successful translation from the narrative description to the formal Message Units. That is why we name our translation system as semi-automatic translation system, not full-automatic one.

The following sentence is the last example for catalog.

1. Checker sends price of item to cotroller.

This is a reasonable description of a message send, but it is not covered by the Syntactic Patterns defined in this thesis. This sentence invokes three objects: checker, item and controller. Our regular patterns only have one or two objects. Therefore, the catalog
should record this special description. When we draw sequence diagram unit for this sentence, we may separate it into two regular expressions, like

1. Checker asks Item to get price.
2. Checker sends price to Controller.

Figure 15 shows the sequence diagram unit of this complicated description.

![Sequence Diagram](image)

Figure 15 Special pattern containing three objects

### 3.3 Two Use Cases

#### 3.3.1 Use Case 1 - Sales

This Sales use case describes sale process. Sales Order receives name, employee number and ID from Salesperson and checks them; receives customer information and sends it to CustInfo. The latter checks customer credit with Accounting and sends the results back to Sales Order. Sales Order receives an item from Salesperson and creates a new Item Ordered. The latter checks item availability and price with Inventory and sends the results back to Sales Order. Sales Order accepts the order and sends some information to Accounting and Inventory. The following is the normalized description. Its sequence
diagram drawn by Rational Rose is shown in Figure 16. Our parser parsed this use case, and listed the output and the catalog in Appendix B.

Salesperson creates new sales order
Salesperson sends information about name, employee number and ID to sales order
Sales order checks name, employee number and ID
Salesperson sends information about customer name and customer address to sales order
Sales order creates new CustInfo
Sales order sends information about customer name and customer address to CustInfo
CustInfo tells Accounting to check customer credit
Accounting sends "customer credit established" message to CustInfo
CustInfo sends "customer credit established" message to sales order
Salesperson tells sales order to order first item
Sales order creates a new item ordered
Item ordered tells Inventory to check item availability and price
Inventory sends "Item availability established" message to item ordered
Item ordered sends information about item availability and price to sales order
Salesperson sends order to sales order
Sales order sends order information to Accounting
Sales order sends order information to Inventory

3.3.2 Use Case 2 - Control Information

This control Information use case demonstrates how to describe control information: condition and iteration. The Order Entry sends a "prepare" message to an Order. The
Figure 16. Sequence diagram for Sales use case
Order then sends "prepare" to each Order Line on the Order. Each Order Line checks the given Stock Item. If this check returns "true", the Order Line removes the appropriate quantity of Stock Item from stock. Otherwise, the quantity of Stock Item has fallen bellow the reorder level, and the Stock Item requests a new delivery. The following is the normalized description. Its sequence diagram drawn by Rational Rose is shown in Figure 17. Our parser parsed this use case, and listed the output and the catalog in Appendix C.

Order Entry tells Order to prepare.

[For each Order Line] {Order tells Order Line to prepare}.

Order Line tells a Stock Item to check item.

If [check = "true"] {Order Line tells the Stock Item to remove item}

Stock Item checks if is empty.

If [needToReorder = "true"] {Stock Item creates new Reorder Item}

Else {Order Line sends "not available" message to Order}
1: prepare

- 2: * prepare

- 3: check(item)

- 4: if [check="true"]
  remove(item)

- 5: isEmpty

- 6: if [needToReorder="true"]
  new

- 7: else sends("not available")

Figure 17 Sequence diagram for control use case
Chapter 4 Conclusions and Future Work

This chapter summarizes the conclusions of this thesis work and outlines the future work.

4.1 Conclusions

In this thesis we have defined a semi-automatic translation system of use cases. This system contains Basic Normalization Principles, Syntactic Patterns, Naming Rules, Message Units, Catalog and Parser. Two simple examples demonstrated that this system is a member-complete system and it can successfully semi-automatically translate normalized narrative descriptions of use cases into sequence diagrams of UML.

This semi-automatic translation system of use cases is mainly based on analyzing elements of sentence. Any natural language has the same set of elements. The only difference among different languages is the position of each element. For example, in English, objects often locate behind of predicate (Verb); but in Japanese, objects often locate in front of predicate (Verb). If we modify the Syntactic Patterns in Table 1 a little, the whole system may be applied to Japanese descriptions of use cases. In other words, this system, or its basic idea, is language-independent.

Based on the work carried out in this thesis we have the following conclusions:

1. This semi-automatic translation system of use cases is a member-complete system.

   We may need to modify each member of this system to cover more complicate cases, but we do not need to add or delete any member.

2. Basic Normalization Principles is a complete set of writing rules, which satisfies the requirements of OOPL and sequence diagram of UML, and guides the definition of Syntactic Patterns.
3. Syntactic Patterns defined in this thesis are a set of simple and reliable patterns, which covers a wide range of normalized narrative descriptions of use cases, although is still not a complete set.

4. Naming Rules name classes, objects, arguments and messages. These rules are in consistence with most popular OOPL such as Java and C++.

5. Message Unit, including String controlInfo, Vector argVector and Object sendMessage, is an equivalent of sequence diagram unit. This message unit covers all the information a description may give.

6. Catalog records information on classes, such as class name, attributes and actions, and special issues. This catalog is the basis for building class diagrams of UML.

7. The parser coded in Java covers most of the patterns defined in Chapter 2. It can successfully translate normalized narrative descriptions of a use case into a set of Message Unites, i.e., into a sequence diagram.

8. This semi-automatic translation system of use cases is mainly based on analyzing elements of sentence. Therefore, this system, or its basic idea, is language-independent.

**4.2 Future Work**

This thesis focuses on semi-automatic translation of narrative descriptions of use cases into sequence diagram of UML. In this area, we have the following future work to do.

1. Expand syntactic patterns so that they may cover more complicate descriptions, which may invoke three objects. For example, we need a pattern to describe:

- Object 1 sends an attribute of object 2 to object 3.
2. Find a way to process complicate control information, such as Decision Point, Asynchronous Message and Synchronization.

3. Build more powerful Parser. The current parser works good, but is not powerful enough to parse very complicate sentences.

Sequence diagram is only one of many Behavior Diagrams in UML. Use cases may be translated into other diagrams such as Collaboration Diagram or Activity Diagram. To do so, we need

1. Expand the existing patterns to cover such situation, or

2. Create new set of patterns for each kind of Behavior Diagrams.

3. Rebuild the current parser or build a new parser for such diagrams.

Translating use cases into Behavior Diagrams of UML is only one aspect of Use Case Management System (UCMS). To build a complete Use Case Management System, we need more work for the following aspects.

1. Repository including
   - Template for Actor and Use Case
   - Class Diagrams
   - Behavior Diagrams

2. Navigation to navigate process

3. Different Abstraction Level of use case
   Consistency between the text-described use cases and their Class Diagrams and Behavior Diagrams
APPENDIX

Appendix A: Parser

//Title: The Parser of Patterns

//Version:

//Copyright: Copyright (c) 1998

//Author: Jinkui Li

//Description: This is the parser in my thesis

package Parser;

import java.io.*;

import java.util.*;

import Parser.*;

public class Driver{
    
    public static void main(String[] args){
        
        Driver driver = new Driver();

        int ch;

        StringBuffer buf = new StringBuffer();

        String inputString = "";

        Vector unit = null;//new Vector();

        try {
            /* create a reader to read file */

            FileReader reader = new FileReader("Control.txt");

            FileReader reader = new FileReader("Sales.txt");

            //
StringTokenizer tokenizer = null;
Vector v = new Vector();

while( (ch = reader.read()) != -1 ){ // read file to end
    buf.append((char)ch);
    while( (ch = reader.read()) != '\n' ) // read a line
        buf.append((char)ch); // fill the buffer
    inputString = new String(buf); // hold the line
    buf.setLength(0); // clean the buffer
    inputString = inputString.trim();
    int len = inputString.length();

    if( len > 0 ){
        if( inputString.charAt(len-1) == '.' )
            inputString = inputString.substring(0, len-1);
        String[] theLine = new String[2];
        theLine = driver.checkControl( inputString.trim() );
        String control = theLine[0];
        System.out.println("**************************************************************************");
        System.out.println("ControlInfo = " + control);
        tokenizer = new StringTokenizer(theLine[1], "\t");
        String[] words = new String[30];
        int i = 0;
        while (tokenizer.hasMoreTokens())
words[i++] = tokenizer.nextToken();

int size = i;

int type = driver.checkType(words, size);

v = driver.makeUnit(type, words, size);

driver.printUnit(v);

}

} // while

reader.close();

Catalog.printTable();

} catch( IOException IOe ){

System.out.println("IOe: " + IOe);

}

}

private String[] checkControl(String input){

String controlInfo = "null";

int index1 = 0;

int index2 = 0;

if( (index1 = input.indexOf('{}') != -1 ){

controlInfo = input.substring(0, index1).trim();

index2 = input.indexOf('}');

input = input.substring(index1+1, index2).trim();

}
String[] aLine = new String[2];
aLine[0] = controlInfo;
aLine[1] = input;
return aLine;
}

private int checkType( String[] words, int size ){
    int flag = 0;
    for( int index = 0; index < size; index++ ){
        String word = words[index];
        if( word.equals("new") )
            flag = 21;
        else if( word.equals("destroy") || word.equals("destroys") )
            flag = 22;
        else if( word.equals("send") || word.equals("sends") )
            flag = 23;
        else if( word.equals("tell") || word.equals("tells") )
            flag = 45;
        if( flag != 0 )
            break;
    }

    if( flag == 0 )
return checkAs( words, size );

else if ( flag == 45 )
    return checkBs( words, size );
else
    return flag;
}

private int checkAs( String[] words, int size ){
    int flag = 12;
    for( int index = 0; index < size; index++ ){
        String word = words[index];
        if( word.equals("is") || word.equals("are") ){
            if(checkDictionary("Adjectives.txt", words[size-1]) ){
                flag = 14;
            }else{
                flag = 13;
            }
        }
        break;
    }else{
        if(checkDictionary( "Verbs.txt",words[size-1]) ){
            flag = 11;
        }
        break;
    }
private int checkBs(String[] words, int size)
{
    int flag = 0;
    for (int index = 0; index < size; index++)
    {
        if (words[index].equals("to");
            int toIndex = index;
        if (toIndex + 2 == size)
        {
            flag = 24;
        } else{
            flag = 25;
        }
    }
    return flag;
}

public boolean checkDictionary(String Dictionary, String word)
{
    try {
        /* create a reader to read file */

    }}
FileReader reader = new FileReader(Dictionary);

StringTokenizer tokenizer = null;

StringBuffer buf = new StringBuffer();

String inputString = "";

int ch;

while( (ch = reader.read())!= -1 ){  // read file to end
    buf.append((char)ch);
    while( (ch = reader.read()) ! = '\n' )  // read a line
        buf.append((char)ch);  // fill the buffer
    inputString = new String(buf);  // hold the line
    buf.setLength(0);  // clean the buffer
    inputString = inputString.trim();
    int len = inputString.length();
    if( len > 0 ){
        tokenizer = new StringTokenizer(inputString, "\\t");
        while (tokenizer.hasMoreTokens())
            if( tokenizer.nextToken().equals(word) )
                return true;
    }
}

reader.close();

} catch( IOException IOe ){

}
System.out.println("IOe: "+ IOe);

}

return false;

}

private Vector makeUnit( int type, String[] words, int size ){
    Vector v = new Vector();
    switch( type ){
        case 11:
            A1 a1 = new A1();
            v = a1.makeUnit(words,size);
            break;
        case 12:
            A2 a2 = new A2();
            v = a2.makeUnit(words,size);
            break;
        case 13:
            A3 a3 = new A3();
            v = a3.makeUnit(words,size);
            break;
        case 14:
            A4 a4 = new A4();
            v = a4.makeUnit(words,size);
    }
break;

case 21:
    B1 b1 = new B1();
    v = b1.makeUnit(words, size);
    break;

case 22:
    B2 b2 = new B2();
    v = b2.makeUnit(words, size);
    break;

case 23:
    B3 b3 = new B3();
    v = b3.makeUnit(words, size);
    break;

case 24:
    B4 b4 = new B4();
    v = b4.makeUnit(words, size);
    break;

case 25:
    B5 b5 = new B5();
    v = b5.makeUnit(words, size);
    break;

default:
    break;
private void printUnit( Vector v ){
    System.out.println("sourceObject = " + v.elementAt(0) );
    System.out.println("targetObject = " + v.elementAt(1) );
    System.out.println("messageName = " + v.elementAt(2) );
    Vector args = (Vector) v.elementAt(3);
    System.out.println("argVector : ");
    for (Enumeration e = args.elements() ; e.hasMoreElements() ;) 
        System.out.print( "   "+ e.nextElement());
    System.out.println();
}

>Title:  The Parser of Patterns

//Version:

//Copyright:  Copyright (c) 1998

//Author:  Jinkui Li

//Description:  This is the parser in my thesis

package Parser;

import java.util.Vector;
public class A1 {

    public A1() {} 

    public Vector makeUnit(String[] words, int size) {
        Driver driver = new Driver();
        Vector unit = new Vector();
        NameMaker maker = new NameMaker();
        int index = 0;
        for (int i = 0; i < size; i++) {
            if (driver.checkDictionary("Verbs.txt", words[i]))
                index = i;
        }

        /* source object */
        String[] source = new String[index];
        for (int i = 0; i < index; i++)
            source[i] = words[i];
        String caller = maker.makeName(source, index);
        unit.addElement(caller);

        /* target object = source object */
        String receiver = caller;
        unit.addElement(receiver);
    }
}
/* action */

// the verb represents the action
String[] operation = new String[1];
operation[0] = words[index];
String action = maker.makeName(operation, 1);
unit.addElement(action);

/* arguments */
Vector argVector = new Vector();
argVector.addElement("null");
unit.addElement(argVector);

Catalog.setTable(caller, receiver, argVector);
return unit;

}
package Parser;

import java.util.Vector;

public class A2 {

    public A2(){

    }

    public Vector makeUnit( String[] words, int size ){

        Vector unit = new Vector();
        Vector argVector = new Vector();
        NameMaker maker = new NameMaker();
        int index = 0;
        int flag = 1;
        for( int i = 0; i < size; i++ ){
            if( words[i].equals("check") || words[i].equals("checks") ) {
                index = i;
                if( words[i+1].equals("if") )
                    flag = 0;
            }
        }

    /* source object */

    String[] source = new String[index];
    for( int i = 0; i < index; i++ )
        source[i] = words[i];

    }
String caller = maker.makeName(source, index);
unit.addElement(caller);

/**< target object = source object*/
String receiver = caller;
unit.addElement(receiver);

if(flag == 0){
 /**< action */
  /**< the part after "check if" represents the action */
  int actionSize = size - index - 2;
  String[] action = new String[actionSize];
  for(int i = 0; i < actionSize; i++)
    action[i] = words[i + index + 2];
  String operation = maker.makeName(action, actionSize);
  unit.addElement(operation);

  /**< arguments */
  argVector.addElement("null");
  unit.addElement(argVector);
} else if(flag == 1){
 /**< action */
  /**< the verb check/checks represents the action */
String[] operation = new String[1];
operation[0] = words[index];
String action = maker.makeName(operation, 1);
unit.addElement(action);

/* arguments */
int argSize = size - index - 1;
String[] args = new String[argSize];
for( int i = 0; i < argSize; i++)
    args[i] = words[index+1+i];
ArgMaker argMaker = new ArgMaker();
argVector = argMaker.getArgs( args, argSize );
unit.addElement(argVector);
}
Catalog.setTable(caller, receiver, argVector);
return unit;
}

>Title: The Parser of Patterns
-Version:
-Copyright: Copyright (c) 1998
-Author: Jinkui Li
//Description: This is the parser in my thesis

package Parser;

import java.util.Vector;

public class A3 {
    public A3() {}

    public Vector makeUnit(String[] words, int size) {
        Vector unit = new Vector();
        NameMaker maker = new NameMaker();
        int index = 0;
        for (int i = 0; i < size; i++) {
            if (words[i].equals("is") || words[i].equals("are") )
                index = i;
        }

        /* source object */
        String[] source = new String[index];
        for (int i = 0; i < index; i++)
            source[i] = words[i];
        String caller = maker.makeName(source, index);
        unit.addElement(caller);

        /* target object = source object*/
    }
}
String receiver = caller;

unit.addElement(receiver);

/* action */

// the part after caller represents the action
int actionSize = size - index;

String[] action = new String[actionSize];

for (int i = 0; i < actionSize; i++)
    action[i] = words[i + index + 2];

String operation = maker.makeName(action, actionSize);

unit.addElement(operation);

/* arguments */

Vector argVector = new Vector();

argVector.addElement("null");

unit.addElement(argVector);

Catalog.setTable(caller, receiver, argVector);

return unit;

}
package Parser;

import java.util.Vector;

public class A4 {
    public A4() {}

    public Vector makeUnit(String[] words, int size) {
        Vector unit = new Vector();
        NameMaker maker = new NameMaker();

        int index = 0;
        for (int i = 0; i < size; i++) {
            if (words[i].equals("is") || words[i].equals("are"))
                index = i;
        }

        /* source object */
        String[] source = new String[index];
        for (int i = 0; i < index; i++)
            source[i] = words[i];
        String caller = maker.makeName(source, index);
        unit.addElement(caller);
    }
}
/* target object = source object*/

String receiver = caller;

unit.addElement(receiver);

/!* action */

// the part after caller represents the action

int actionSize = size - index;

String[] action = new String[actionSize];

for( int i = 0; i < actionSize; i++ )
    action[i] = words[i + index + 2];

String operation = maker.makeName(action, actionSize);

unit.addElement(operation);

/!* arguments */

Vector argVector = new Vector();

argVector.addElement("null");

unit.addElement(argVector);

Catalog.setTable(caller, receiver, argVector);

return unit;

}
package Parser;

import java.util.Vector;

public class B1 {

    public B1(){
    }

    public Vector makeUnit( String[] words, int size ){
        Vector unit = new Vector();
        NameMaker maker = new NameMaker();
        int verbIndex = 0;
        int newIndex = 0;
        for( int i = 0; i < size; i++ ){
            if( words[i].equals("create") || words[i].equals("creates") ||
                words[i].equals("open") || words[i].equals("opens") ||
                words[i].equals("start") || words[i].equals("atarts") )
                verbIndex = i;
            if( words[i].equals("new") )
                newIndex = i;
        }
    }
}
/* source object */

String[] source = new String[verbIndex];
for( int i = 0; i < verbIndex; i++ )
    source[i] = words[i];
String caller = maker.makeName(source, verbIndex);
unit.addElement(caller);

/* target object */

int targetSize = size - newIndex - 1;
String[] target = new String[targetSize];
for( int i = 0; i < targetSize; i++ )
    target[i] = words[i + newIndex + 1];
String receiver = maker.makeName(target, targetSize);
unit.addElement(receiver);

/* action */

// "new" represents the action
unit.addElement("new");

/* arguments */

Vector argVector = new Vector();
argVector.addElement("null");
unit.addElement(argVector);
Catalog.setTable(caller, receiver, argVector);

    return unit;

}

}

//Title: The Parser of Patterns

//Version:

//Copyright: Copyright (c) 1998

//Author: Jinkui Li

//Description: This is the parser in my thesis

package Parser;

import java.util.Vector;

public class B2 {

    public B2(){}

    public Vector makeUnit( String[] words, int size ){

        Vector unit = new Vector();

        NameMaker maker = new NameMaker();

        int index = 0;

        for( int i = 0; i < size; i++ ){

            if( words[i].equals("destroy") || words[i].equals("destroys") )

                index = i;

            }
/* source object */

String[] source = new String[index];
for( int i = 0; i < index; i++ )
    source[i] = words[i];

String caller = maker.makeName(source, index);
unit.addElement(caller);

/* target object */

int targetSize = size - index - 1;
String[] target = new String[targetSize];
for( int i = 0; i < targetSize; i++ )
    target[i] = words[i + index + 1];

String receiver = maker.makeName(target, targetSize);
unit.addElement(receiver);

/* action */

// only a verb represents the action
String[] operation = new String[1];
operation[0] = words[index];

String action = maker.makeName(operation, 1);
unit.addElement(action);


/* arguments */

Vector argVector = new Vector();
argVector.addElement("null");
unit.addElement(argVector);

Catalog.setTable(caller, receiver, argVector);
return unit;
}
}

>Title: The Parser of Patterns

//Version:

//Copyright: Copyright (c) 1998

//Author: Jinkui Li

//Description: This is the parser in my thesis

package Parser;

import java.util.Vector;

public class B3 {
  public B3() {}

  public Vector makeUnit( String[] words, int size ){
    Vector unit = new Vector();
    

NameMaker maker = new NameMaker();

int sendIndex = 0;
int toIndex = 0;

for( int i = 0; i < size; i++ ){
    if( words[i].equals("sends") || words[i].equals("send") )
        sendIndex = i;
    if( words[i].equals("to") )
        toIndex = i;
}

/* source object */

String[] source = new String[sendIndex];
for( int i = 0; i < sendIndex; i++ )
    source[i] = words[i];
String caller = maker.makeName(source, sendIndex);
unit.addElement(caller);

/* target object */

int targetSize = size - toIndex - 1;
String[] target = new String[targetSize];
for( int i = 0; i < targetSize; i++ )
    target[i] = words[i + toIndex + 1];
String receiver = maker.makeName(target, targetSize);
unit.addElement(receiver);

/ * action */

// only a verb represents the action
String[] operation = new String[1];
operation[0] = words[sendIndex];
String action = maker.makeName(operation, 1);
unit.addElement(action);

/ * arguments */

Vector argVector = new Vector();
StringBuffer buf = new StringBuffer();
int argSize = 0;
int startIndex = 0;
if( words[sendIndex+1].charAt(0) == "" ){
    for( int i = sendIndex+1; i < toIndex-1; i++ )
        buf.append( words[i] + " " );
    argVector.addElement( buf.toString().trim() );
} else {
    if( words[sendIndex+1].equals("information") ){
        argSize = toIndex-sendIndex-3;
        startIndex = sendIndex + 3;
} else{
    argSize = toIndex-sendIndex-1;;
    startIndex = sendIndex + 1;
}

String[] args = new String[argSize];
for( int i = 0; i < argSize; i++)
    args[i] = words[startIndex+i];

ArgMaker argMaker = new ArgMaker();
argVector = argMaker.getArgs( args, argSize );
}

unit.addElement(argVector);

Catalog.setTable(caller, receiver, argVector);

return unit;
}

>Title:    The Parser of Patterns
//Version:
//Copyright: Copyright (c) 1998
//Author:    Jinkui Li
//Description: This is the parser in my thesis

100
package Parser;

import java.util.Vector;

public class B4 {
    
    public B4() {}

    public Vector makeUnit( String[] words, int size ){
        Vector unit = new Vector();
        NameMaker maker = new NameMaker();
        int tellIndex = 0;
        int toIndex = 0;

        for( int i = 0; i < size; i++ ){
            if( words[i].equals("tells") || words[i].equals("tell") )
                tellIndex = i;
            if( words[i].equals("to") )
                toIndex = i;
        }

        /* source object */
        String[] source = new String[tellIndex];
        for( int i = 0; i < tellIndex; i++ )
            source[i] = words[i];
        String caller = maker.makeName(source, tellIndex);
        unit.addElement(caller);
    }
}
/* target object */

int targetSize = toIndex - tellIndex - 1;

String[] target = new String[targetSize];

for( int i = 0; i < targetSize; i++ )
    target[i] = words[i + tellIndex + 1];

String receiver = maker.makeName(target, targetSize);

unit.addElement(receiver);


/* action */

// only a verb represents the action

String[] operation = new String[1];

operation[0] = words[toIndex+1];

String action = maker.makeName(operation, 1);

unit.addElement(action);

/* arguments */

Vector argVector = new Vector();

argVector.addElement( "null" );

unit.addElement(argVector);

Catalog.setTable(caller, receiver, argVector);

return unit;
package Parser;

import java.util.*;

import Parser.*;

public class B5 {

public B5() {

}

public Vector makeUnit( String[] words, int size ){

Vector unit = new Vector();

NameMaker maker = new NameMaker();

int tellIndex = 0;

int toIndex = 0;

for( int i = 0; i < size; i++){

if( words[i].equals("tells") || words[i].equals("tell") )

tellIndex = i;

}
if ( words[i].equals("to") )
    toIndex = i;
}

/* source object */
String[] source = new String[tellIndex];
for ( int i = 0; i < tellIndex; i++ )
    source[i] = words[i];
String caller = maker.makeName(source, tellIndex);
unit.addElement(caller);

/* target object */
int targetSize = toIndex - tellIndex - 1;
String[] target = new String[targetSize];
for ( int i = 0; i < targetSize; i++ )
    target[i] = words[i + tellIndex + 1];
String receiver = maker.makeName(target, targetSize);
unit.addElement(receiver);

/* action */
// only a verb represents the action
String[] operation = new String[1];
operation[0] = words[toIndex+1];
String action = maker.makeName(operation, 1);
unit.addElement(action);

/* arguments */
Vector argVector = new Vector();
int argSize = size-toIndex-2;
int argIndex = 0;
String[] args = new String[argSize];
for( int i = toIndex+2; i < size; i++){
    args[argIndex++] = words[i];
}
ArgMaker argMaker = new ArgMaker();
argVector = argMaker.getArgs( args, argSize );
unit.addElement(argVector);

Catalog.setTable(caller, receiver, argVector);
return unit;
}
package Parser;

public class NameMaker {

    public NameMaker() {
    
    }

    public String makeName( String[] words, int size ){

        StringBuffer buf = new StringBuffer();
        char first = ' ';
        int startIndex = 0;

        if( words[0].equals("a") ||
            words[0].equals("an") ||
            words[0].equals("the") ||
            words[0].equals("A") ||
            words[0].equals("An") ||
            words[0].equals("The") ) {
            startIndex = 1;
        } else {
            startIndex = 0;
        }

        for( int j = startIndex; j < size; j++ ){
            first = words[j].charAt(0);
        }
    
}
if ( j == startIndex ){
    if( Character.isUpperCase(first) ){
        buf.append( (char)(first+32) );
        buf.append(words[j].substring(1));
    }else{
        buf.append(words[j]);
    }
}else{
    if( Character.isLowerCase(first) ){
        buf.append( (char)(first-32) );
        buf.append(words[j].substring(1));
    }else{
        buf.append(words[j]);
    }
}

return buf.toString();

//Title:    The Parser of Patterns
//Version:
//Copyright: Copyright (c) 1998
package Parser;

import java.util.*;

public class ArgMaker {
    public ArgMaker() {
    }

    public Vector getArgs(String[] words, int size) {
        Vector vector;
        String s = getNewString(words, size);
        vector = getArguments(s);
        return vector;
    }

    private String getNewString(String[] words, int size) {
        StringBuffer buf = new StringBuffer();
        for (int i = 0; i < size; i++) {
            if (words[i].equals("and"))
                words[i] = ", ";
            buf.append(words[i] + " ");
        }
        String s = buf.toString().trim();
private Vector getArguments( String s){
    int len = s.length();
    boolean hasMore = true;
    int index = 0;
    String anArg = new String();
    Vector vector = new Vector();
    String argument = "";

    while(hasMore){
        if( (index= s.indexOf(','))!= -1 ){
            anArg = s.substring(0, index).trim();
            s = s.substring(index+1).trim();
        }else{
            anArg = s;
            hasMore = false;
        }
        argument = makeAnArgument(anArg);
        vector.addElement(argument);
    }

    return vector;
}
private String makeAnArgument(String anArg) {
    NameMaker maker = new NameMaker();
    StringTokenizer tokenizer = new StringTokenizer(anArg, " \\	");
    String[] ws = new String[30];
    int i = 0;
    while (tokenizer.hasMoreTokens())
        ws[i++] = tokenizer.nextToken();
    int number = i;
    return maker.makeName(ws, number);
}

//Title: The Parser of Patterns

//Version:

//Copyright: Copyright (c) 1998

//Author: Jinkui Li

//Description: This is the parser in my thesis

package Parser;

import java.util.Hashtable;

import java.util.Vector;
import java.util.Enumeration;

public class Catalog {

    public static Hashtable table = new Hashtable();

    public Catalog() {
    }

    public static void setTable(String source, String target, Vector args) {
        String sourceClass = source.substring(0, 1).toUpperCase() + source.substring(1);
        String targetClass = target.substring(0, 1).toUpperCase() + target.substring(1);
        Vector vT = (Vector) args.clone();
        Vector vS = (Vector) args.clone();
        if (!vS.contains(target)) {
            vS.addElement(target);
        }
        doGet(targetClass, vT);
        doGet(sourceClass, vS);
    }

    private static void doGet(String key, Vector v) {
        if (table.containsKey(key)) {
            Vector args = (Vector) table.get(key);
            String arg = null;
        }
    }
}
for (Enumeration e = v.elements(); e.hasMoreElements();)
{
    arg = (String)e.nextElement();
    if ( !args.contains(arg) )
        args.addElement(arg);
}
} else {
    table.put(key, v);
}

public static void printTable()
{
    System.out.println("***************************");
    System.out.println("*********** Catalog ***********");
    System.out.println("***************************");
    String key = "";
    Vector vec = null;
    for (Enumeration e = table.keys(); e.hasMoreElements();)
    {
        key = (String)e.nextElement();
        System.out.println("Class Name = " + key);
        vec = (Vector)table.get(key);
        String attribute = "";
        for (Enumeration arg = vec.elements(); arg.hasMoreElements();)
        {
            attribute = (String)arg.nextElement();
        }
if( !attribute.equals("null") )
    
    System.out.println( "Attribute: " + attribute );

}

System.out.println( "--------------------------------" );

}

}

Appendix B: Output and Catalog of Sales use Case

C:\JBuilder2\java\bin\javaw.exe -classpath
"F:\\UW\cs\Classes;C:\JBuilder2\lib\swingall.jar;C:\JBuilder2\lib\jbc12.0.jar;C:\JBuilder2\lib\jbc12.0-res.jar;C:\JBuilder2\lib\jg13.1.0.jar;C:\JBuilder2\lib\vjorb.jar;C:\JBuilder2\lib\vjapp.jar;C:\JBuilder2\lib\vjgk.jar;C:\JBuilder2\java\lib\classes.zip"
Parser.Driver
AppAccelerator(tm) 1.1.034 for Java (JDK 1.1), x86 version.
Copyright (c) 1998 Borland International. All Rights Reserved.

**********************
ControlInfo = null
sourceObject = salesperson
targetObject = salesOrder
messageName = new
argVector :
    null
**********************

ControlInfo = null
sourceObject = salesperson
targetObject = salesOrder
messageName = sends
argVector :
    name  employeeNumber  ID
**********************

ControlInfo = null
sourceObject = salesOrder
targetObject = salesOrder
messageName = checks
argVector :
name employeeNumber ID
**********************************************************************
ControlInfo = null
sourceObject = salesPerson
targetObject = salesOrder
messageName = sends
argVector :
   "customerName customerAddress"
**********************************************************************
ControlInfo = null
sourceObject = salesOrder
targetObject = custInfo
messageName = sends
argVector :
   null
**********************************************************************
ControlInfo = null
sourceObject = salesOrder
targetObject = custInfo
messageName = check
argVector :
   "customerCredit"
**********************************************************************
ControlInfo = null
sourceObject = accounting
targetObject = custInfo
messageName = sends
argVector :
   "customer credit established"
**********************************************************************
ControlInfo = null
sourceObject = custInfo
targetObject = salesOrder
messageName = sends
argVector :
   "customer credit established"
**********************************************************************
ControlInfo = null
sourceObject = salesperson
targetObject = salesOrder
messageName = order
argVector:
  firstItem
  ************************************
ControlInfo = null
sourceObject = salesOrder
targetObject = itemOrdered
messageName = new
argVector:
  null
  ************************************
ControlInfo = null
sourceObject = itemOrdered
targetObject = inventory
messageName = check
argVector:
  itemAvailability  price
  ************************************
ControlInfo = null
sourceObject = inventory
targetObject = itemOrdered
messageName = sends
argVector:
  "Item availability established"
  ************************************
ControlInfo = null
sourceObject = itemOrdered
targetObject = salesOrder
messageName = sends
argVector:
  itemAvailability  price
  ************************************
ControlInfo = null
sourceObject = salesperson
targetObject = salesOrder
messageName = sends
argVector:
  order
  ************************************
ControlInfo = null
sourceObject = salesOrder
targetObject = accounting
messageName = sends
argVector:
  orderInformation
ControlInfo = null
sourceObject = salesOrder
targetObject = inventory
messageName = sends
argVector:
   orderInformation

************* Catalog *************

Class Name = ItemOrdered
Attribute: itemAvailability
Attribute: price
Attribute: inventory
Attribute: "Item availability established"
Attribute: salesOrder

Class Name = Inventory
Attribute: itemAvailability
Attribute: price
Attribute: "Item availability established"
Attribute: itemOrdered
Attribute: orderInformation

Class Name = Accounting
Attribute: customerCredit
Attribute: "customer credit established"
Attribute: custInfo
Attribute: orderInformation

Class Name = SalesOrder
Attribute: name
Attribute: employeeNumber
Attribute: iD
Attribute: salesOrder
Attribute: customerName
Attribute: customerAddress
Attribute: custInfo
Attribute: "customer credit established"
Attribute: firstItem
Attribute: itemOrdered
Attribute: itemAvailability
Attribute: price
Attribute: order
Attribute: orderInformation
Attribute: accounting
Attribute: inventory

Class Name = CuslInfo
Attribute: customerName
Attribute: customerAddress
Attribute: customerCredit
Attribute: accounting
Attribute: "customer credit established"
Attribute: salesOrder

Class Name = Salesperson
Attribute: salesOrder
Attribute: name
Attribute: employeeNumber
Attribute: iD
Attribute: customerName
Attribute: customerAddress
Attribute: firstItem
Attribute: order

Appendix C: Output and Catalog of Control Information use Case

C:\JBUILDER2\java\bin\javaw.exe -classpath
"F:\UW\cs\Classes;C:\JBUILDER2\lib\swingall.jar;C:\JBUILDER2\lib\jbc12.0.jar;C:\JBUILDER2\lib\jbc12.0-res.jar;C:\JBUILDER2\lib\jg13.1.0.jar;C:\JBUILDER2\lib\vbjorb.jar;C:\JBUILDER2\lib\vbjapp.jar;C:\JBUILDER2\lib\vbjgk.jar;C:\JBUILDER2\java\lib\classes.zip"
Parser.Driver
AppAccelerator(tm) 1.1.034 for Java (JDK 1.1), x86 version.
Copyright (c) 1998 Borland International. All Rights Reserved.

**************************************************
ControlInfo = null
sourceObject = orderEntry
targetObject = order
messageName = prepare
argVector :
    null
**************************************************
ControlInfo = [For each Order Line]
sourceObject = order
targetObject = orderLine
messageName = prepare
argVector : null

************************************
ControlInfo = null
sourceObject = orderLine
targetObject = stockItem
messageName = check
argVector :
  item

************************************
ControlInfo = If [check = "true"]
sourceObject = orderLine
targetObject = stockItem
messageName = remove
argVector :
  item

************************************
ControlInfo = null
sourceObject = stockItem
targetObject = stockItem
messageName = isEmpty
argVector :

************************************
ControlInfo = If [needToReorder = "true"]
sourceObject = stockItem
targetObject = reorderItem
messageName = new
argVector :
  null

************************************
ControlInfo = Else
sourceObject = orderLine
targetObject = order
messageName = sends
argVector :
  "not available"

************************************
********** Catalog **********
************************************

Class Name = OrderEntry
Attribute: order

Class Name = StockItem
Attribute: item
Attribute: stockItem
Attribute: reorderItem

Class Name = ReorderItem

Class Name = Order
Attribute: orderLine
Attribute: "not available"

Class Name = OrderLine
Attribute: item
Attribute: stockItem
Attribute: "not available"
Attribute: order
BIBLOGRAPHY


[9] Buhr, R.J.A., High Level Design with Use Case Maps, OOPSLA ’97, October 5-9, 1997 - Atlanta, Georgia

[10] Desmond, F., Catalysis: Framework and Component based Modeling (with UML and OMG compliant models), OOPSLA ’97, October 5-9, 1997 - Atlanta, Georgia


[27] Vemulapalli, Chandra, A use case FAQ, OOPSLA‘95 Workshop, Austin, Texas, 15 October 1995.

[28] Vlissides, J., An Introduction to Design Patterns, OOPSLA ’97, October 5-9, 1997 - Atlanta, Georgia

[29] Wirfs-Brock, Rebecca and McKeen, Alan, Responsibility-Driven Design, OOPSLA ’97, October 5-9, 1997 - Atlanta, Georgia
VITA AUCTORIS

Jinkui Li was born in 1947 in Harbin, China. He graduated from Harbin Third High School in 1966. From there he went on to Harbin Shipbuilding Engineering Institute where he obtained a Bachelor degree in Mechanical Engineering in 1982, and went on to Harbin Institute of Technology where he obtained a Master degree in Mechanical Engineering in 1984 and a Doctor degree in Materials Engineering in 1989. He is currently a candidate for the Master’s degree in Computer Science at the University of Windsor and hopes to graduate in the Fall of 1998.