

1996

An object-oriented diagram editor and code generator

Glenn Fung
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Fung, Glenn, "An object-oriented diagram editor and code generator" (1996). *Master's Theses*. 1365.
DOI: <https://doi.org/10.31979/etd.ckk6-d32c>
https://scholarworks.sjsu.edu/etd_theses/1365

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

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. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

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.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

AN OBJECT-ORIENTED DIAGRAM EDITOR AND CODE GENERATOR

A Thesis

Presented to

The Faculty of Department of Mathematics and Computer Science
San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Glenn Fung

December 1996

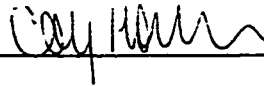
UMI Number: 1382569

UMI Microform 1382569
Copyright 1997, by UMI Company. All rights reserved.

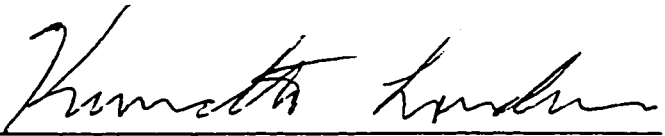
**This microform edition is protected against unauthorized
copying under Title 17, United States Code.**

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

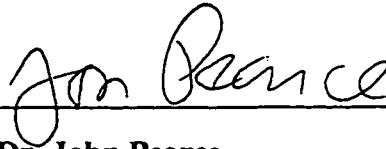
**APPROVED FOR THE DEPARTMENT OF
MATHEMATICS AND COMPUTER SCIENCE**

A handwritten signature in cursive script, appearing to read "Cay Horstmann", written above a horizontal line.

Dr. Cay Horstmann

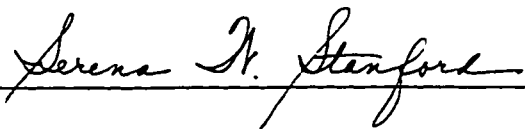
A handwritten signature in cursive script, appearing to read "Kenneth Loudon", written above a horizontal line.

Dr. Kenneth Loudon

A handwritten signature in cursive script, appearing to read "John Pearce", written above a horizontal line.

Dr. John Pearce

APPROVED FOR THE UNIVERSITY

A handwritten signature in cursive script, appearing to read "Serena N. Stanford", written above a horizontal line.

Copyright 1996
Glenn Fung
ALL RIGHTS RESERVED

ABSTRACT

AN OBJECT-ORIENTED DIAGRAM EDITOR AND CODE GENERATOR

by Glenn Fung

There are currently a number of object-oriented CASE tools in the market that capture object-oriented design and generate code in an object-oriented programming language from a design model. However, a review of these tools indicates that the support for the mapping of class diagram notations to C++ programming language syntax and the support of iterative development are still inadequate. This research is based on the CLOUD9 project, in which a CASE tool that provides better support in these areas will be implemented. Concepts used in the process of automating the use of a subset of the Booch's notation are discussed. Key issues related to C++ code generation (such as translation of 1 to many aggregation relationship to C++ code) are identified and various methods for handling these issues are proposed. A class diagram editor is also implemented to illustrate some of the concepts proposed in this research.

ACKNOWLEDGMENTS

I am extremely grateful to my advisor, Dr. Cay Horstmann who showed tremendous patience as I found my way to this research project. He constantly offered invaluable suggestions and criticisms of my work, and provided guidance and insights into the research topic. I am also grateful to Dr. Loudon and Dr. Pearce for their constructive criticisms of the draft of this thesis.

I would like to thank my family for their support, understanding and encouragement over the years, especially my wife Deborah, who patiently reviewed, corrected, criticized my work repeatedly.

TABLE OF CONTENTS

1. INTRODUCTION	1
1.1 OVERVIEW OF THE CLOUD9 PROJECT	2
1.2 DESIGN OF THE CLOUD9 GRAPHICAL TOOL	2
1.3 WORK OF THIS RESEARCH	5
2. OBJECT ORIENTED METHODOLOGY NOTATIONS	7
2.1 BOOCH NOTATION	8
2.1.1 <i>Class Diagram</i>	9
2.1.2 <i>Class</i>	11
2.1.3 <i>Class Relationships</i>	13
2.1.4 <i>Adornments</i>	14
2.1.4.1 <i>Cardinality</i>	15
2.1.4.2 <i>Physical Containment</i>	15
2.1.4.3 <i>Export Control</i>	17
2.1.4.4 <i>Abstract Class</i>	17
2.1.4.5 <i>Static Class</i>	17
2.1.4.6 <i>Virtual Base Class</i>	18
2.1.4.7 <i>Friend Class</i>	18
2.1.5 <i>Class Categories</i>	18
2.1.6 <i>Class Utility</i>	23
2.1.7 <i>Parameterized Class</i>	23
2.1.8 <i>Strengths and Weaknesses of Booch Notations</i>	24
2.2 COAD NOTATION	25
2.2.1 <i>Class-&-Object</i>	25
2.2.2 <i>Attributes</i>	26
2.2.3 <i>Message Connections</i>	27
2.2.4 <i>Services</i>	28
2.2.5 <i>Gen-Spec Structure</i>	28
2.2.6 <i>Whole-Part Structure</i>	29
2.2.7 <i>Subjects</i>	30
2.2.8 <i>Object (Instance) Connection</i>	31
2.2.9 <i>Strengths and Weaknesses of Coad Notations</i>	31
2.3 MARTIN & ODELL NOTATION	32
2.3.1 <i>Object Relationship Diagram</i>	32
2.3.2 <i>Object Generalization Hierarchy Diagram</i>	34
2.3.3 <i>Object Composed-of Diagram</i>	36
2.3.4 <i>Cardinality Constraints</i>	37
2.3.5 <i>Minimum and Maximum Cardinality Constraint</i>	38
2.3.6 <i>Strengths and Weaknesses of Martin & Odell Notations</i>	39
2.4 RUMBAUGH (OMT) NOTATION	40
2.4.1 <i>Object Model</i>	41
2.4.1.1 <i>Object</i>	41
2.4.1.2 <i>Class</i>	41
2.4.1.3 <i>Association</i>	42
2.4.1.4 <i>Aggregation</i>	43
2.4.1.5 <i>Generalization</i>	44
2.4.1.6 <i>Module</i>	45
2.4.2 <i>Strengths and Weaknesses of OMT Notations</i>	45

3. REVIEW OF EXISTING CASE TOOLS	47
3.1 RATIONAL ROSE/C++	50
3.1.1 <i>User Interface</i>	50
3.1.2 <i>Source Code Partitioning</i>	51
3.1.3 <i>Code Generation</i>	52
3.1.3.1 <i>Redundant Member Functions</i>	55
3.1.3.2 <i>Support Iterative Development</i>	55
3.1.4 <i>Synchronization of Code and Diagram</i>	56
3.1.5 <i>Data Storage</i>	57
3.2 TOGETHER/C++	57
3.2.1 <i>User Interface</i>	58
3.2.2 <i>Source Code Partitioning</i>	59
3.2.3 <i>Code Generation</i>	60
3.2.4 <i>Synchronization of Code and Diagram</i>	63
3.2.5 <i>Reverse Engineering</i>	63
3.3 OEW FOR C++	64
3.3.1 <i>User Interface</i>	64
3.3.2 <i>Source Code Partitioning</i>	66
3.3.3 <i>Code Generation</i>	66
3.3.3.1 <i>Reference Relationship (association)</i>	66
3.3.3.2 <i>Containment Relationship</i>	67
3.3.3.3 <i>Forward Declarations</i>	68
3.3.3.4 <i>Schema Generator</i>	68
3.3.4 <i>Synchronization of Code and Diagram</i>	69
3.4 CONCLUSION	69
4. DESIGN AND IMPLEMENTATION OF THE CLASS DIAGRAM EDITOR	71
4.1 MODELING OF CLASS DIAGRAMS	72
4.1.1 <i>Requirements of Class Diagram</i>	73
4.1.2 <i>Implementation of Class Diagram</i>	73
4.1.3 <i>User Interface</i>	75
4.2 MODELING OF CLASS SYMBOLS	75
4.2.1 <i>Requirements of Class Symbol</i>	75
4.2.2 <i>Implementation</i>	76
4.2.3 <i>User Interface</i>	78
4.3 MODELING OF CLASS CONNECTORS	79
4.3.1 <i>Requirements of connector</i>	80
4.3.2 <i>Implementation</i>	80
4.3.3 <i>User Interface</i>	82
4.4 SPECIAL FEATURES	83
5. DESIGN OF THE CODE DOCUMENT	86
5.1 REQUIREMENTS	86
5.2 IMPLEMENTATION	87
5.3 MODELING THE STRUCTURE TREE	88
5.3.1 <i>Module</i>	88
5.3.2 <i>Type Declaration Section</i>	90
5.3.3 <i>Type Declaration</i>	91
5.3.4 <i>Class</i>	92
5.3.5 <i>Field Variable</i>	93
5.3.6 <i>Function</i>	94

6. TRANSLATION OF OO DIAGRAMS TO C++	96
6.1 TRANSLATION OF CLASS SYMBOL	96
6.1.1 <i>Attribute</i>	97
6.1.2 <i>Operation</i>	98
6.2 TRANSLATION OF CLASS RELATIONSHIP	99
6.2.1 <i>Has (aggregate) Relationship</i>	99
6.2.2 <i>Is-a (Inheritance) Relationship</i>	104
6.2.2.1 Multiple Inheritance	105
6.2.2.2 Common Base Class	105
6.2.2.3 Access Control	107
6.2.3 <i>Using Relationship</i>	107
6.3 FORWARD REFERENCE PROBLEM	108
7. SYNCHRONIZATION OF CODE AND OO DIAGRAMS	110
7.1 CLASSES	110
7.2 INHERITANCE	111
7.3 1-TO-1 RELATIONSHIP	112
7.4 1-TO-N RELATIONSHIP	113
7.5 POINTER MEMBER VARIABLES	115
7.6 REFERENCE MEMBER VARIABLES	118
7.7 COMMON CLASS ATTRIBUTES	119
7.8 USING RELATIONSHIP	120
7.9 STATIC MEMBER VARIABLES	122
8. CONCLUSIONS AND FURTHER RESEARCH	124
8.1 CLASS CATEGORIES	125
8.2 GRAND METHODOLOGY OF BOOCH AND RUMBAUGH	125
8.3 USE OF WIZARDS	126
8.4 GENERATED CODE FOR OTHER OO LANGUAGES	126
BIBLIOGRAPHY	127

LIST OF TABLES

Table I. Five Min-Max cardinality constraint combinations	39
Table II. The varies Has relationship and their C++ translations.	104

LIST OF FIGURES

Figure 1. The Application Framework of CLOUD9	3
Figure 2. The design architecture of the Cloud9 project	5
Figure 3. Some elements of the Booch notation	10
Figure 4. The class diagram for the flow-chart editor	11
Figure 5. Association relationship	13
Figure 6. Class & Object , and Class symbols for OOA	26
Figure 7. Gen/Spec symbol in OOA	29
Figure 8. Whole-Part Symbol in OOA	30
Figure 9. Subject notation, collapsed	30
Figure 10. Subject notation, partially expanded	31
Figure 11. An object relationship between person and company	33
Figure 12. An object-generalization hierarchy diagram of class Person	34
Figure 13. An object-generalization hierarchy diagram represented by arrow symbols	35
Figure 14. A complete and incomplete subtype partition symbols	36
Figure 15. An Object composed-of diagram	36
Figure 16. 1 with many cardinality constraint symbol	37
Figure 17. 1 with 1 cardinality constraint symbol	37
Figure 18. Zero or more cardinality constraint	38
Figure 19. Zero or one cardinality constraint	38
Figure 20. Min and max cardinality constraint	39
Figure 21. OMT notation for classes, and an example of a graphics object class	42
Figure 22. The association notation in OOA	43
Figure 23. Aggregation relationship notation in OMT	44
Figure 24. Specialization/Generalization notation of OMT	44
Figure 25. The Class Diagram Editor of Rational Rose	51
Figure 26. Application Framework of the Together/C++	58
Figure 27. Create Subject Dialog in Together/C++	60
Figure 28. The Create Class Dialog in Together/C++	61
Figure 29. The User Interface of OEW	65
Figure 30. The Container Class Specification Dialog	67
Figure 31. The Class Diagram for the Class Diagram Editor	72
Figure 32. Hide>Show Classes Dialog	79
Figure 33. Relationship Specification Dialog	82
Figure 34. Member Type Checking Dialog	84
Figure 35. Container Classes Specification Dialog	85
Figure 36. Translation of class cloud symbol	96
Figure 37. Translation of class attributes	97
Figure 38. Translation of class operations	98
Figure 39. Translation of aggregation relationship (has) symbol with by value containment	100
Figure 40. Translation of aggregation relationship with by reference containment	101
Figure 41. Translation of 1-to-0/1 aggregation relationship	101
Figure 42. Translation of 1-to-many aggregation relationship	102
Figure 43. Translation of inheritance relationship	104
Figure 44. Translation of multiple inheritance relationship	105
Figure 45. Common base class multiple inheritance relationship	106
Figure 46. Class symbols with circular reference	108
Figure 47. Translation of C-- classes to diagram symbols	110

Figure 48. Translation of inheritance classes	112
Figure 49. Translation of a class member variable	113
Figure 50. Translation of a C array member variable	114
Figure 51. Translation of a pointer member variable	116
Figure 52. Translation of a reference member variable	118
Figure 53. Translation of trivial type as a class type	119
Figure 54. Translation of a using relationship	121

CHAPTER 1

INTRODUCTION

A development process typically involves three phases: analysis, design and implementation. In the analysis phase, the requirements of the system are transformed into a description that is independent of the programming languages from the perspective of the classes and objects found in the problem domain. In the design phase, the textual description of the system requirements is transformed into an object-oriented notation. The goal of object-oriented design is to decompose a programming task into classes and to specify the behavior (methods or functions that operate on this data). In the implementation phase, the specifications of the classes made in the design phase are translated into a target programming language such as C++. The software development cycle of a complex software application often involves many coding modifications, leaving very little of its initial design unchanged. An object-oriented CASE (computer-aided software engineering) tool [Taylor] can therefore be indispensable for managing complex design information. A typical CASE tool supports design documentation (or the description of a design through standard forms and diagrams) and code generation (or the translation of design information into a target programming language such as C++). However, existing CASE tools such as Rational Rose [Horst2] or C++ Designer [Horst1] currently do not provide a good match with the target language, nor do they provide sufficient support for iterative development. This research is based on the Cloud9 project. It studies the graphical information in a class diagram created using Booch's method and

analyzes how these information can be translated to C++ and vice versa. Concepts proposed in this research aim to provide better support of design documentation and code generation in CASE tools. It also demonstrates that such concepts can be implemented.

1.1 Overview of the Cloud9 Project

The Cloud9 project is proposed by Dr. Cay S. Horstmann [Horst4] and is a research on a method of object-oriented design documentation and C++ code generation that supports incremental development

This project involves:

1. Building a theoretical foundation for a design methodology that is sufficiently expressive to support iterative code generation.
2. Implementing a Windows-based graphical tool for the entry, revision and display of the object-oriented design information as well as the non-routine portions of the implementation code such as comments.
3. Building a code generator that produces compilable C++ code.

1.2 Design of the Cloud9 Graphical Tool

The Cloud9 graphical tool has four major components: a diagram editor for class diagram editing, a hierarchical structure editor for source code editing, a code document to store the C++ source information, and a code generator to produce C++ source files.

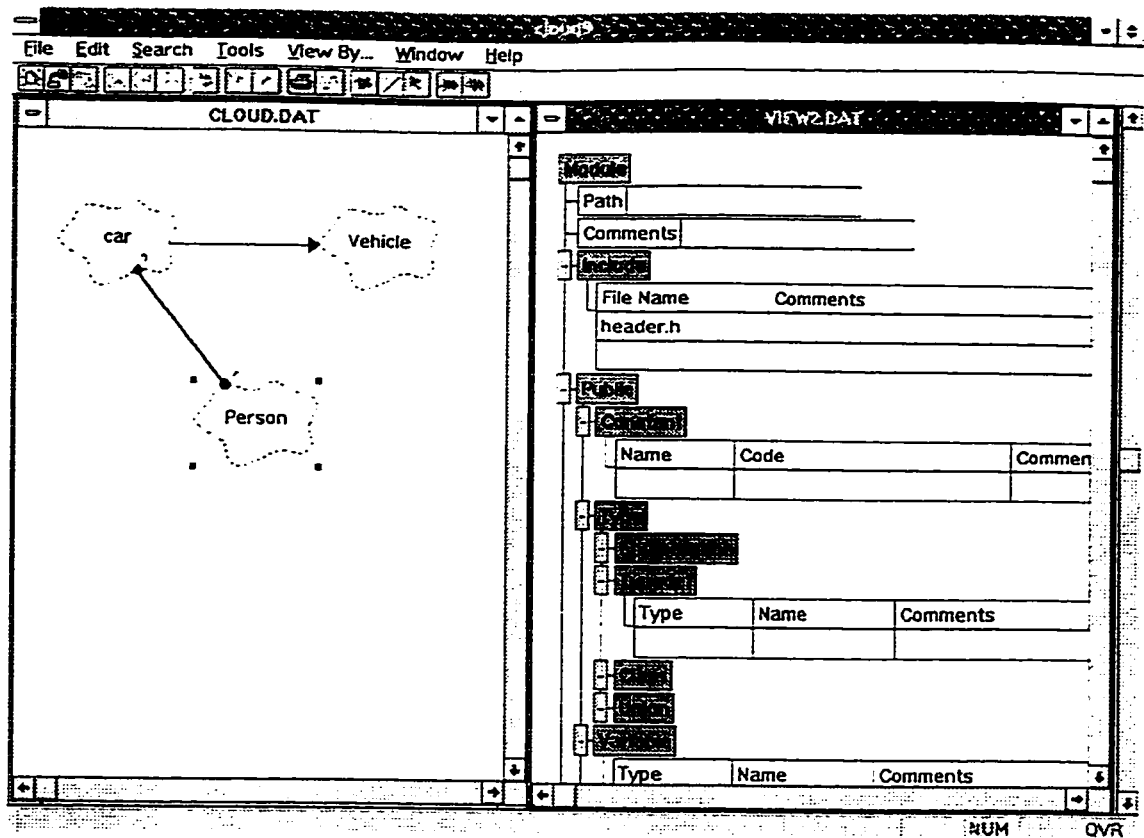


Figure 1. The Application Framework of CLOUD9

This graphical tool will be implemented as an MDI (multiple document interface) application with views as shown below in [Figure 1]. It uses a document/view model to separate the design into two manageable parts, namely the data and its display. A document module is only responsible for managing the data while the view modules are responsible for displaying the data in various views. The code document is the document module; it adds semantic and physical information to the graphical model. It also provides C++ language-specific information to the diagram editor, hierarchical structure editor and the code generator. The diagram editor and hierarchical structure editor are the view modules; they provide a visualization of the model in a class

diagram and a hierarchical structure view. Each view module uses the information store in the code document but has its own graphical details, such as layout, symbols and display information.

The purpose of the diagram editor is to represent the semantics and visualization of a design or implementation. The diagram editor maps the class information from the code document and has its own graphical details such as layout information, symbol notations. The Booch notation is generally useful to model the language-independent concepts such as classes and their relationships. However, some of the more language-intensive aspects of a design or implementation, such as the implementation of member functions, overloaded operators, expressions, etc., are so linguistically based that it makes little sense to capture this information within the diagram. We allow this kind of language-specific information to be entered using the hierarchical structure editor and they are stored uninterpreted as literal strings in the code document.

The hierarchical structure editor provides the editing capability to the C++ code in a hierarchical structure way, much the same as an outline code processor. It allows the programmer to add code for member function bodies and class data fields without using an ASCII text editor. This provides better synchronization of design information and source code, and eliminates the problem of reading in programmer-modified ASCII files since both design and code information are held in the code document.

The diagram editor is synchronized with the hierarchical structure editor. This means

that changes to the class diagram due to the addition, removal or editing of class symbols will be propagated to the code document and be reflected simultaneously in the hierarchical structure editor.

The code generator uses the program information stored in the code document and generates a pair of C++ header and implementation files.

The following diagram illustrates the complete design flow of the Cloud9 project:

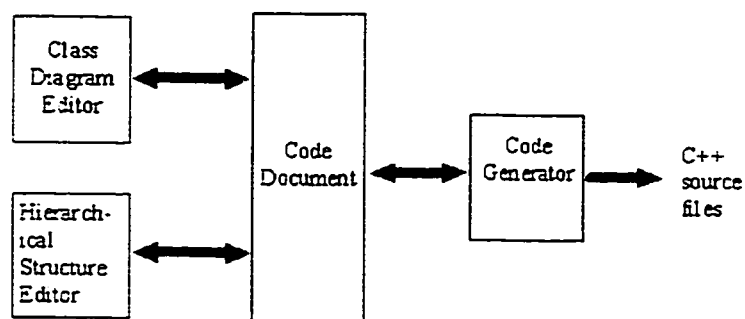


Figure 2. The design architecture of the Cloud9 project

1.3 Work of This Research

The first phase of this research is the study of class diagrams, notations and conventions used in Booch's method, Coad's method, Martin & Odell's method and Rumbaugh's method. This helps formulate the design methodology to be used in the CASE tool of this research.

The second phase involves the review of three existing object-oriented CASE tools, Rational Rose/C++, Together/C++ and OEW for C++ based on their support of

source code partitioning, design documentation, code generation, synchronization of code and their user interface. This is crucial to the evaluation of design features to be implemented.

The third phase is the design of portions of a CASE tool for editing object-oriented designs and programs. This involves the following:

1. The design of the code document for storing language-dependent information.
2. The design of a class diagram editor that supports the class component of the Booch's method. This includes the support of cardinalities and sharing for aggregation and usage in the class diagram editor, the suppression of inferred connections of classes, and the inference of class relationships.
3. The mapping from the class diagram notations to C++ programming language syntax.
4. The synchronization of code and object-oriented diagrams.

The last phase is the implementation of a class diagram editor for the entry, revision and display of the object-oriented design information. This demonstrates that concepts presented in this research can be implemented in a CASE tool.

Due to the complexity and time requirements of building a complete CASE tool, only the class diagram editor is implemented in this research. However, the concepts presented here will serve as a good basis for the development and implementation of such a tool.

CHAPTER 2

OBJECT ORIENTED METHODOLOGY NOTATIONS

An object-oriented methodology is a collection of object-oriented mechanisms applied across the software development life cycle from requirements to implementation and unified by some general, philosophical approach. All object-oriented methodologies have a defined set of concepts (class, object, etc.), graphic symbols, diagrams, and rules. We characterize methodologies in the following 4 ways:

1. **Concepts:** An object-oriented methodology must define and deal with concepts such as object, class, state, inheritance, and aggregation. However, there are no standard terms for the object-oriented concepts, and some methodologies use different terms to express the same concepts. For example, in Booch's notation [Booch91], the term "has-a" relationship is defined as an aggregation relationship; whereas in Coad's notation [Coad1] it is called a "whole-part" relationship.
2. **Process:** A methodology describes the steps or processes that must be taken to accomplish tasks such as software design, implementation, reuse, reengineering and testing.
3. **Notation:** Many methodologies require the creation of abstract descriptions of the system under consideration. These can be textual descriptions, graphical notations or some other form of display. One of the major requirements of any methodology is that the set of models form a complete and consistent description of the system.

The expressiveness of the notation is also very important. Each of the methodologies will be studied for its support for such concepts as class, attributes, operations, aggregation, and generalization/specialization. If the notation is not sufficiently expressive, the user is required to use additional form of representation to express the model, for example, attaching text description to the model. This can lead to inconsistent and complex models.

4. **Pragmatics:** A methodology's pragmatics consists of resources, required expertise, accessibility and language suitability. The availability of resources such as text books, users groups or CASE tools that support the method is a major factor in determining how well a methodology can be used for the software process.

In our research, we will focus on how object-oriented graphical notations are used to describe a system. We will mainly study object-oriented models that describe the static structure of the system.

2.1 Booch Notation

Grady Booch's method and notations have undergone significant evolution since initial publication and popularization in [Booch91]. After teaming up with James Rumbaugh, Booch is now producing a grand unified methodology of object-oriented software development. The discussion here is based on his second version of the methodology described in [Booch94]. The Booch method mainly covers the analysis

and design phases of an object-oriented system. In Booch's method, the problem domain is modeled from two different perspectives: the logical model of the system and the physical model of the system. For each perspective, both the static and dynamic semantics are modeled. The logical model is represented in the class and object structures. In the class structure, one constructs the architecture, or the static model. The object structure shows how the classes interact with each other and describes the dynamic behavior of the system. The physical model is represented in the process and module structures. The module and process architecture describe the physical allocation of the classes to modules and processes. The Booch notation is a widely used object-oriented notation that defines a rich set of graphical symbols to describe almost every aspect of the design decisions. It sometimes is criticized for its large set of different symbols. If one works with this method, one will notice that one seldom needs to use all of these symbols and diagrams in the OOA/OOD. On the other hand, the fact that it fully documents every aspects of ones object-oriented code and supports iterative and incremental development is an asset of the notation. The Booch notation contains a Class Diagram, an Object Diagram, a State Event Diagram, a Module Diagram, a Process Diagram, and an Interaction Diagram.

2.1.1 Class Diagram

A class diagram is used to show the existence of classes and their relationships in the logical view of a system. A single class diagram represents a view of the class structure of a part of the system. It is used to indicate the common roles and

responsibilities of the entities that provide the system's behavior in the analysis phase. It is used to capture the structure of the classes that form the system's architecture in the design phase. [Figure 3] shows several graphical notations used for the class diagram in the Booch method. Most of these symbols appear in the development of the example that follows. This example does not illustrate every notation, but does explain most of the important ones. The two most important basic elements of a class diagram are classes and their relationships. We will also study other advanced concepts such as class categories, physical containment, export control, parameterized classes, and class utilities. Both low-level constructs (attributes) and more abstract high-level constructs (class category) can be used together and appear on the same diagram. [Figure 4] is an example that shows a particular view of a flow-chart editor built by a *Flow Chart* class which is made up of nodes and edges. We will use this class diagram to explain the various Booch notations.

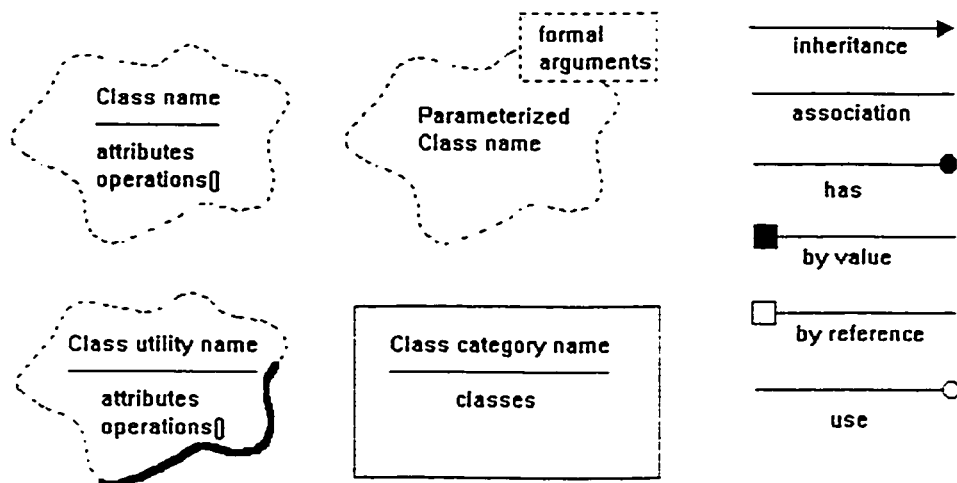


Figure 3. Some elements of the Booch notation

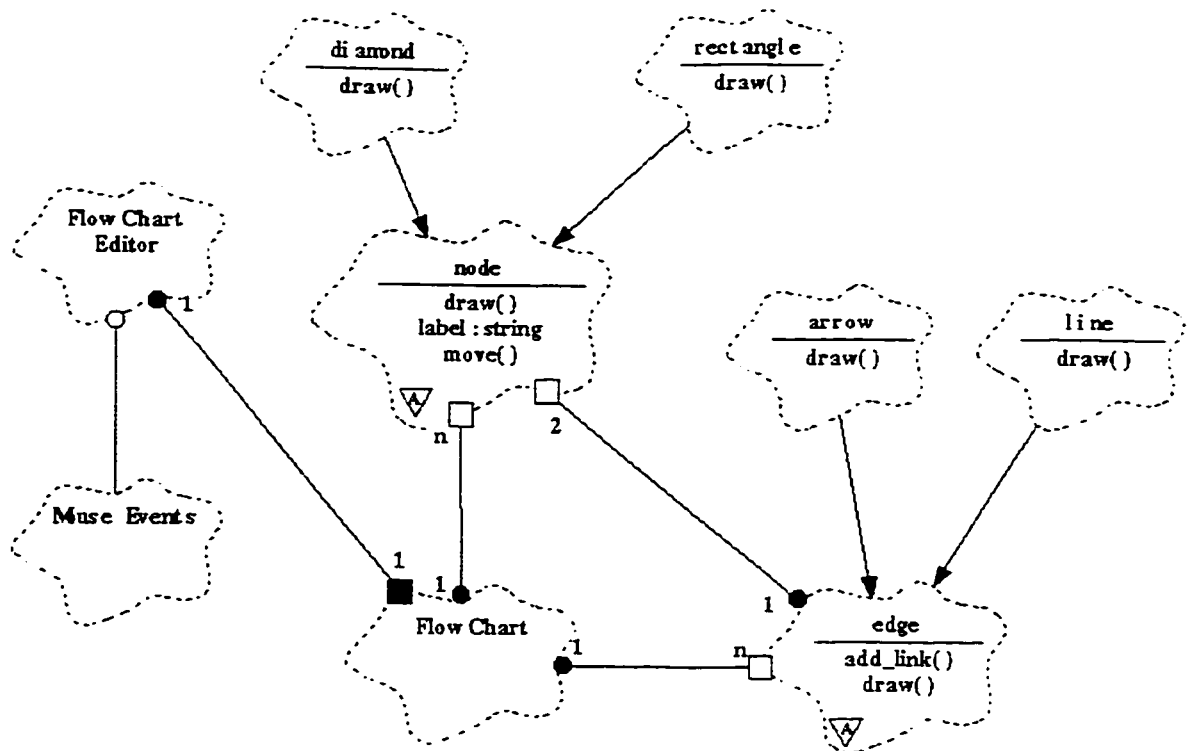


Figure 4. The class diagram for the flow-chart editor

2.1.2 Class

The central class in this example [Figure 4] is the *Flow Chart* class. The *Flow Chart* class is a cloud shape icon with the name “Flow Chart” inside it. Booch uses a special cloud shape to represent a class in the class diagram, as shown in [Figure 3]. Every class requires a name and it must be unique to the enclosing class category, i.e. *Flow Chart*. The class name is placed inside the cloud icon; if the name string is particularly long, it can either be elided or the icon magnified. Furthermore, it is useful to expose the non-trivial attributes and operations associated with a class in the class diagram. In the second edition of his object-oriented book [Booch94], he introduces the additional

notations, the attributes and operations to denote these properties of the class. The syntax of an attribute name may have a name, a class, or both, and an optional default value:

- A Attribute name only
- :C Attribute class only
- A:C Attribute name and class
- A:C = V Attribute name, class and a default value

The syntax of an operation name is distinguished from an attribute by appending a pair of parentheses or its signature:

- F() Operation name only
- R F(Arguments) Operation with a return value and arguments.

Both of the attribute and operation names must be unambiguous in the context of the class. These names are placed inside a cloud icon under a separation line from the class name. If we choose not to show any of these class members at all, we may drop the separating line and show only the class name. In our example, the *node* class has an operation *draw()* which draws the node on the flow chart diagram and an attribute *label* which records the label of the node on the diagram.

2.1.3 Class Relationships

Classes usually relate to other classes in the class diagram. The basic connections among classes include association, inheritance, and aggregation, whose icons are summarized in [Figure 3]. Each such relationship may include a textual label to document the name of the relationship. Relationship names must be unique within their context. The most general relationship between classes is the association relationship. The association icon is a straight line that connects two classes in [Figure 3]. It denotes a semantic connection. It is possible to have more than one association between two classes or for a class to have an association to itself. After forming the general association connection between two classes, we can refine them into an inheritance, has, or using relationship. For example, in a retail point of sale system, we may show a simple association between a Sale class and a Product class [Figure 5]: the class Product denotes the products sold as part of a sale, and the class Sale denotes the transaction through which products were last sold.

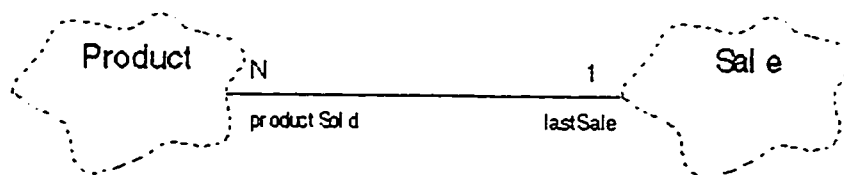


Figure 5. Association relationship

The line with a filled circle at the end in [Figure 3] shows that we have an aggregation. It is sometimes referred to as the *has* relationship. The filled circle end denotes the aggregate. The other end of association denotes the part whose instances are

contained by the aggregate object. In our example [Figure 4], it specifies that a flow-chart editor is made up of various flow-charts. A flow-chart is made up of nodes and edges. An edge is made up of two nodes.

The arrows in [Figure 3] show an inheritance relationship. This is sometimes referred to as the *is-a* relationship. The arrowhead points in the direction of the generalization class, and the opposite end of the arrowhead designates the specialization class. In our example [Figure 4], it specifies that diamonds or rectangles are special nodes in the flow-chart diagram. A line or an arrow is an edge between two nodes. Being specialization classes of edge, line and arrow classes inherit all the relationship sets that edge has. Thus, line is made up of two nodes too.

The line with an open circle at the end in [Figure 3] denotes a client/supplier relationship. It is sometimes referred to as the *using* relationship. It is used to indicate the operation of the client class that have signatures of the instances of supplier class. In [Figure 4], it specifies that a diagram editor uses the mouse events to recognize all the mouse movement and screen updates.

2.1.4 Adornments

Booch further defines a class or a relationship by specifying its access, cardinality, and physical containment. An adornment is a secondary piece of information about some entity in a system's model.

2.1.4.1 Cardinality

Booch uses cardinality adornments to indicate cardinality for classes and relationships.

Class cardinality specifies the number of instances (objects) allowed for that class.

Relationship cardinality specifies the number of links between the client objects and the supplier objects. Booch displays the cardinality value in the constraints portion of the class icon. The cardinality values for a relationship are displayed at the applicable ends of the relationship symbol. In our example [Figure 4], it specifies that a diagram is made up of numbers of nodes and edges. But each edge consists of only of two nodes.

2.1.4.2 Physical Containment

Aggregation does not necessarily mean physical containment of its parts, it can also imply the physical containment of a reference to its parts. Aggregate relationship has two types of physical containment: by value and by reference. The by-reference containment to a *has* relationship between two classes implies that data in one class has a pointer or a reference to an instance of another class. The by-value containment to a *has* relationship between two classes implies that the instance of the aggregate class allocates storage for and has within it an instance of the part class. The choice of physical containment in the aggregate class is usually a detailed, tactical issue.

However, there are two important factors to distinguish the physical containment in the *has* relationship:

1. Physical containment has semantics that play a role in the construction and destruction of an aggregate's parts. Containment by value implies that the construction and destruction of these aggregate parts occurs as a result of the construction and destruction of the aggregate itself. By contrast, if the aggregate only contains a reference to the part, this means that the lifetime of the aggregate parts may be independent of the aggregate.
2. Specification of physical containment is essential for correct code generation from the design model to implementation code. For example, in C++, you can use a pointer or reference to implement the *has* relationship by reference.

Booch uses two different adornments to represent these two types of physical containment on the *has* relationship in the class diagram. The by-value adornment is a black square at the aggregate end of a *has* relationship. The by-reference adornment is an open square at the aggregate end of a *has* relationship. Consider our example in [Figure 4]. The flow-chart editor whose instances physically contain an instance of the class *Flow Chart*. By contrast, each instance of class *Flow Chart* physically contains only a reference or pointer to N instances of class *node* and class *edge*. This implies that a flow-chart has a number of nodes and edges, but the flow-chart does not own them.

2.1.4.3 Export Control

Booch use export adornments to specify the type of access allowed between classes: public, private, protected, or implementation. For example, in C++, members may be public (accessible to all client classes), protected (accessible only to subclasses, friends or itself), or private (accessible to itself or friends). You are allowed to apply export adornments to *has*, *uses*, or *is-a* relationships. Booch specifies an access type for a relationship by adorning the client end of a relationship with the export control symbols.

2.1.4.4 Abstract Class

Booch uses the abstract adornment on a class to identify a base class, defining operations and states that will be inherited by subclasses. An abstract class is a class that has no instances and has one or more abstract operations. The abstract adornment is a triangle with a letter “A” inside and placed at the lower left corner of the class. In our example in [Figure 4], the *node* class is an abstract class.

2.1.4.5 Static Class

Booch uses the static adornment on a *has* relationship to specify that the instance of the part class is owned by the class itself but not by any individual instance. The static adornment is a triangle with a letter “S” inside on the client end of the *has* relationship.

2.1.4.6 Virtual Base Class

Booch uses the virtual adornment to designate a shared base class that will be inherited by descendants of the subclasses in a multiple inheritance situation. The virtual adornment is a triangle-shaped icon with a letter “V” inside. The virtual adornment is placed at the subclass (supplier) end of the inheritance relationship.

2.1.4.7 Friend Class

Booch uses the friend adornment to designate that the supplier class has granted rights to a client class to access its non-public parts. The friend adornment is a triangle-shaped icon with a letter “F” inside.

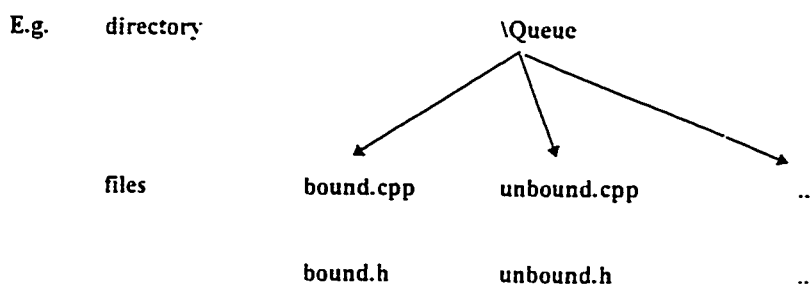
2.1.5 Class Categories

A large project may typically contain hundreds to thousands of classes. It is virtually impossible to organize all these classes into one class diagram. To overcome this problem, Booch introduces a construct to organize classes into separate meaningful class categories. A class category is an aggregate containing classes and other class categories; it is used to partition the logical model of a system. Top-level class diagrams contain only class categories at the highest level of abstraction. That allows programmers to understand and visualize the general architecture of the system. Each class in the system must be a member of a single class category or exist at the top level of the system. A class category icon is a rectangular shape with the name of the category in the upper part and members of the category in the lower part. Each class

category name must be unique and distinct from all other class names. Class categories support *using* relationships. A *using* relationship means that the classes contained in the client class category can use the classes that are exported from the supplier class category. If a class category contains a number of common classes that are used by virtually every class category in the system, for example, a graphical library, Booch introduces a global adornment to be included in the class category icon, instead of displaying the classes in all of the class category diagrams that use it. The global adornment is simply the word “global” placed at the lower left corner of the class category icon.

Most programming languages do not provide direct support for this concept. The concept of class category is often implemented using one of the following methods:

1. Each member of a class category is represented by a set of two files; a header file (.h) that holds the definitions within the class and a source file (.cpp) that holds the implementation of the class. Files from all classes within the same category are organized into a file directory. This directory denotes the class category.



This method organizes class categories into a simple and manageable form. However, class names must be unique in the entire project because no name-resolution method is defined. Nested categories are supported with sub-directories. The only drawback is sharing classes in multiple categories are not allowed since class names must be unique.

2. All classes within a class category appear in a single header and source file.

This pair of files denotes the class category.

E.g.	<pre>class bounded_queue { /* ... */ }; class unbounded_queue { /* ... */ };</pre>	<pre>bounded_queue::add() { /* ... */ };</pre>
	queue.h	queue.cpp

This is similar to method 1. However, nested categories are not supported.

3. The class category is declared as a class. Members of the class category are declared as nested classes inside the class category class. Nested categories are supported.

```

E.g.  class queue
      {
          class bounded_queue
          {
              /* ... */
          };
          class unbounded_queue
          {
              /* ... */
          };
      };
      queue::bounded_queue    bq;

```

Each class in a class category must be qualified by the class category name. This eliminates name conflicts. But it is tedious to reference a class because the name often gets very long. Furthermore, if the definition of a class inside a class category is ever changed, all code using any classes in the class category that contains this modified class will need to be recompiled. Also, all nested classes must be contained in a single file.

4. Only recently a new syntax was added to C++ to provide a new scope into which names can be declared. It is called namespaces [Stroutstrup94]. The main idea of namespaces is to enclose all names inside a class scope to avoid name conflicts. For example:

```
namespace Vender_X
{
    class bounded_queue;
    class unbounded_queue;
    class syn_bounded_queue;
};
```

You can then refer to these classes by qualifying with the namespace:

```
Vender_X::bounded_queue    bq;
```

or you can use the “using” statement to denote a default namespace.

```
using namespace Vender_X;
class bounded_queue {};
```

Namespaces can contain the definitions of global variables, functions, types, classes, and templates. Namespaces are particularly useful in that they allow third party class providers to hide their class names behind a single name space. For example:

```
RogueWave::Set<int> x;
NIH::Set<int> y;
Chi::Set<int> z;
```

But the more important facet of the namespace feature is that it allows the design construct of class category to be expressed in the implementation language.

The class category construct is currently not supported in the Cloud9 project.

2.1.6 Class Utility

For most projects, there usually is a set of functions or procedures that are not member functions of any classes but supplement the overall system. Booch defines these functions and procedures as free subprograms. Booch provides a well-defined construct called class utility for this kind of free subprograms. A class utility is a set of operations that provide additional functions for classes but are not methods of any objects. The notation for a class utility is a cloud with a gray shadow at the lower edge of the cloud to distinguish it from the class cloud. The Class utility icon is shown in [Figure 3]. Class utilities have the same naming and relationship constraints as simple classes.

2.1.7 Parameterized Class

A parameterized class is like a simple class, but with a dashed-line rectangle box in the upper right hand corner denoting its formal parameters. The parameterized class icon is shown in [Figure 3]. A parameterized class is one that serves as a template for other classes; it may not have any instances unless we first instantiate it (that is, fill in the parameters). An instantiated class is adorned with a solid-line box denoting its actual parameters, matched positionally to the corresponding formal parameters in the parameterized class. The instantiation process always involves other concrete classes and requires some *uses* relationship to fill in the template parameter.

2.1.8 Strengths and Weaknesses of Booch Notations

1. Rich Set of Notations

The object model constructs offer good expressive power. The full Booch notation can graphically model almost anything in a system.

2. Support Incremental Development

The Booch notation is particularly applicable to incremental and iterative development. Diagrams are easily evolved and detailed information can be specified during the course of the design process. For example, before we have a clear picture of the relationship between some classes, we can always use the association relationship to generalize the relation. It can later be changed to a *has* relationship or *is-a* relationship by altering the association symbol once the design decision is made. Additional details are provided by adding adornments.

3. Support New C++ Language Features

The extended notation supports new C++ language mechanism, such as templates, namespace and static variables.

4. Models and Methods Lack of Theoretical Foundation

The semantics of the Booch Method's modeling concepts and its corresponding notation are thoroughly described, but no formal mathematical basis for the model is provided.

2.2 Coad Notation

This object-oriented methodology is documented in [Coad1] [Coad2]. Coad's method mainly covers the analysis and initial design phases of an OO system. In this method, the problem domain and system responsibility are directly mapped into a single OOA model with multiple layers. Ultimately this OOA model may consist of as many as five layers(Subject, Class-&-Object, Structure, Attribute, and Service), produced during the five major activities of OOA mentioned in his book. These 5 activities are finding Class-&-Object, identifying structure, identifying subjects, defining attributes, and defining services. These activities guide the analyst from high levels of abstraction (e.g., problem domain, Class-&-Object) to lower level of abstraction (structures, attributes, and services).

2.2.1 Class-&-Object

Object: An abstraction of something in a problem domain, reflecting the capabilities of a system to keep information about it, interact with it, or both; an encapsulation of Attribute values and their exclusive Services.

Class: A description of one or more Objects with a uniform set of Attributes and Services, including a description of how to create new objects in the class.

Class-&-Object: A term meaning a class and the objects in that Class.

The "Class-&-Object" symbol represents both a Class and one or more Objects in that class. The Class symbol is represented by the inner bold rounded rectangle, divided

into three horizontal sections; and the Object symbol is represented by the outer lighter rounded rectangle. [Figure 6]

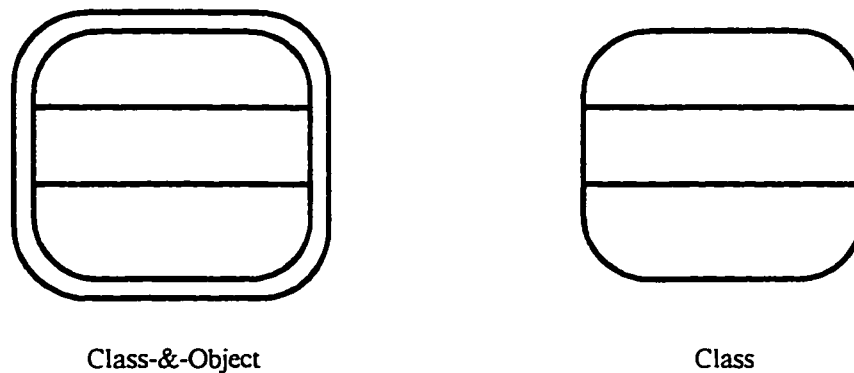


Figure 6. Class & Object, and Class symbols for OOA

The symbol is labeled with its class name placed in the first horizontal section.

Attributes and Services go to the middle and bottom sections.

A variation on the Class-&-Object symbol is the “Class” symbol. It is referred as an “abstract class” in Object-Oriented programming. This symbol is just like the “Class-&-Object” symbol without the Object layer. [Figure 6] represents a class with no objects. Objects may come from its specialization.

2.2.2 Attributes

An Attribute is some data (state information) for which each object in a class has its own value. Attributes are listed in the center section of the Class-&-Object and Class symbols.

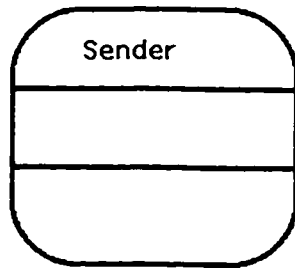
The Class-&-Object symbol is now getting more specific and more detailed with the addition of the Attributes. Attributes describe values kept within an object, and are exclusively manipulated by the services of that object. i.e. private data in C++ Class. If another object in the system needs to access or manipulate the values in an object, it must do so by specifying a message connection corresponding to a service defined by that object. We will review the message connection in the advanced topic section.

2.2.3 Message Connections

A Message Connection models the processing dependency of an Object, indicating a need for Services in order to fulfill its responsibilities. The notation for a Message Connection is a light or dashed arrow that connects both the sender and receiver objects. The arrow points from the sender (Object sends the message) to the receiver (Object receives the message).



For Message Connections that involve more than two Objects, we will have an alternate notation. This notation is used to enhance the graphical layout of the diagram and make the model more readable.



2.2.4 Services

A Service is a specific behavior that an Object is responsible for exhibiting. Services are listed in the bottom section of the Class-&-Object and Class symbols. Services describe what an object does.

2.2.5 Gen-Spec Structure

Gen-Spec Structure notation is shown with a Generalization Class at the top and Specialization Classes below, with lines connecting them. [Figure 7] The notation uses a semi-circle mark to distinguish a Gen-Spec Structure. The notation is directional, with a line drawn outward from the semicircle midpoint to the generalization class. Since it is directional, the classes of the Gen-Spec Structure could be placed anywhere inside the structure; however, the author advises to place the generalization class higher and the specialization classes lower to form a hierarchical structure for easy understanding. The endpoints of the connection lines are connected to the inner class symbol of a “Class-&-Object” symbol to reflect a

mapping between Classes rather than between Objects. A specialization class inherits the attributes and services defined in its generalization class(es).

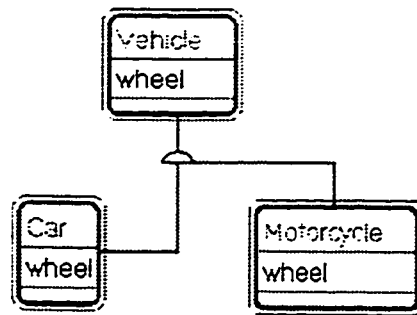


Figure 7. Gen/Spec symbol in OOA

2.2.6 Whole-Part Structure

The Whole-Part Structure is shown with a whole Object at the top, and then a part Object below, with a line drawn between them. The notation uses a triangle mark to distinguish a Whole-Part Structure. The notation is directional with a line drawn outward from the tip of the triangle to the whole object. The endpoints of the connection line are connected to the outer object symbol of a Class-&-Object symbol to reflect a mapping between Objects rather than between Classes. Each end of a Whole-Part Structure line is marked with a range, indicating the number of parts that a Whole may have, and vice versa. In example [Figure 8], a mailbox contains zero to many messages. Observe that the cardinality of the Whole-Part Structure in Coad notation is completely the opposite of the Booch notation.

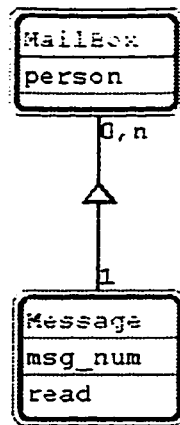


Figure 8. Whole-Part Symbol in OOA

2.2.7 Subjects

A Subject is a mechanism for guiding a reader through a large, complex model.

Subjects are also helpful for organizing work packages on larger projects, based upon initial OOA investigations.

A subject is represented by a shaded rectangular box as shown in [Figure 9]. A subject has a name and a reference tag. i.e. a number. Subjects may contain Class-&-Object and other Subjects. A Class-&-Object may be in more than one Subject. A subject may be fully collapsed [Figure 9], partially expanded [Figure 10] (listing its classes), or fully expanded in the subject layer.

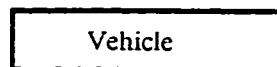


Figure 9. Subject notation, collapsed

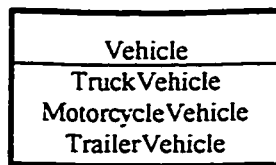


Figure 10. Subject notation, partially expanded

2.2.8 Object (Instance) Connection

An object connection is represented by a solid line connecting two objects. The connection line is marked with a range or limit, indicating the number of other objects that an object may be associated with.

2.2.9 Strengths and Weaknesses of Coad Notations

1. Limited Analysis Notations

There are no constructs to represent certain language-specific features, such as templates and class utility functions. The notation focuses primarily upon objects and classes, and relationships between objects and classes. There are no specific constructs for graphically representing internal class methods and their interactions with one another.

2. Clear and Concise Notation

Unlike the Booch's notation, Coad notation is very concise. It is especially easy to understand, targeted to show a clear and simple representation of classes and relationships between objects and classes.

3. Model Independent

Unlike structured analysis and design, different types of diagrams are not necessary for different phases of development circle. You can use the same set of diagram in analysis and design phases.

2.3 Martin & Odell Notation

This methodology [Martin] is based on the logic and set theory. The authors give a very extensive introduction to all Object-Oriented concepts and turn their focus on the management of the complexity of the system. The methodology strongly focuses on the dynamic behavior of objects. It also develops techniques to identify objects and their relationships. However, only the static models of the methodology are discussed in this thesis.

Martin & Odell methodology is characterized by a specific set of models. Various object diagrams provide a formal graphical notation for modeling objects, classes, and their relationships to one another. There are three types of object diagrams: Object relationship diagrams, Object generalization hierarchy diagrams and Object composed-of diagrams.

2.3.1 Object Relationship Diagram

An object type describes a group of objects with similar properties, common behavior, and common relationships to other objects. It is equivalent to Booch's class term.

Person, company, diagram and car are all object types. An object type is drawn as a

rectangular box. Martin & Odell discourages the idea of including attribute information of the object type in a graphical symbol. Since the diagram focuses on the relationship of the objects, attributes are less important aspects of the model.

An object-relationship diagram is similar to an entity-relationship diagram. It maps the relationships among object types. An object relationship is a conceptual connection between object types. For example, a person *works-for* a company. An object relationship is drawn as a line between two object types. Each relationship is described by a label attached to the line. For example, *works-for* describes the relationship between a person and a company. The inverse of *works-for* could be called *employs*, which relate a company to a person. A label establishes a direction in an object relationship as shown in [Figure 11]. Martin & Odell uses a simple label notation convention in the book, if the relationship direction is from left node to the right node, the label is placed on top of the line; if the relationship direction is from right node to the left node, then the label is placed below the line. If the line is rotated, the label should rotate accordingly. Rectangular boxes, lines and labels are three basic components of an object-relationship diagram.

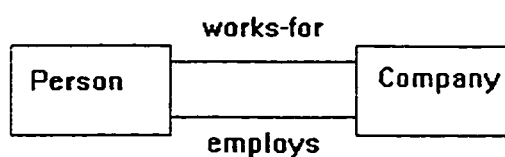


Figure 11. An object relationship between person and company

2.3.2 Object Generalization Hierarchy Diagram

The object relationship diagram can modify to describe the hierarchical structure of the object types. An object-generalization hierarchy diagram describes how object types inherit from each other. An object type can be a more specialized type called subtype and can be a more general type called supertype. For example, Mammal is a supertype of Person, which has its own subtypes, a Female Person or a Male Person. The generalization hierarchy can be expressed as boxes within boxes as shown in [Figure 12].

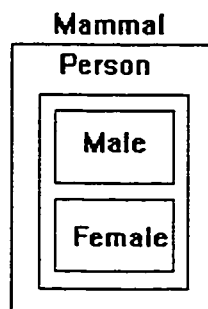


Figure 12. An object-generalization hierarchy diagram of class Person

In an object-generalization hierarchy diagram, overlapping boxes are sometimes hard to draw and maintain. Martin & Odell provides an alternate notation that is simpler than drawing many overlapping boxes within boxes in a hierarchy diagram. [Figure 13] demonstrates that filled arrows make it easy to visually identify the direction of the generalization in a hierarchy. Animal is a supertype of Mammal, which in turn is a supertype of Person.

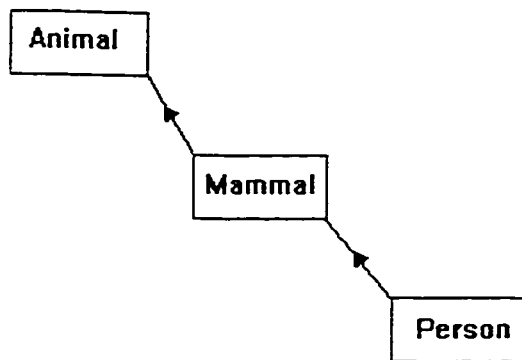


Figure 13. An object-generalization hierarchy diagram represented by arrow symbols

Type partition is defined as a division or partitioning of an object type into disjoint subtypes. Other methods use the terms generalization/specialization or inheritance. A subtype can be partitioned into two classes : complete subtype partition and incomplete subtype partition. A complete subtype partition is a partition that contains all possible subtypes. For example, in [Figure 14] Male and Female are complete subtypes partitions of Person. An incomplete subtype partition is a partition that contains part of the possible subtypes. An incomplete partition is represented by an empty area at the bottom of the partition box. For instances, in [Figure 14] a person is a mammal, a dog is a mammal, and a whale is also a mammal. But the list is incomplete here, a mammal can also be a cat.

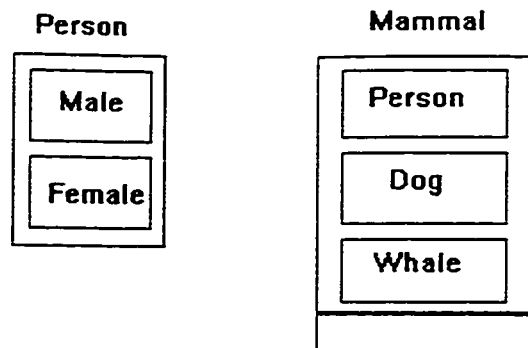


Figure 14. A complete and incomplete subtype partition symbols

2.3.3 Object Composed-of Diagram

An object-composed-of diagram describes the “whole-part” or aggregation relationship in which component objects are associated with assembly objects. An object-composed-of notation is drawn like the relationship notation, except that a small open “C” symbol is used to indicate the assembly end of the relationship.

[Figure 15] shows a portion of an object-composed-of diagram for a word processing program. A document is consisted of many paragraphs, and each paragraph is consisted of many sentences.

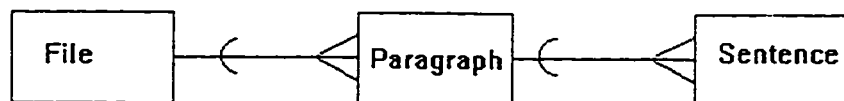


Figure 15. An Object composed-of diagram

2.3.4 Cardinality Constraints

Martin & Odell uses different symbols to specify various cardinality constraints of relationships. A line is drawn between two nodes to indicate they are related. A crow's feet connector from a line to a node is used to specify a 1 with many association. In [Figure 16], the relationship symbol means that each document has 1 or more paragraphs, each of which contains many sentences.



Figure 16. 1 with many cardinality constraint symbol

A 1 with 1 cardinality constraint symbol means that an instance of one object type is associated with one and only one instance of another. In [Figure 17], the 1 with 1 cardinality constraint symbol means that each country has only one capital.

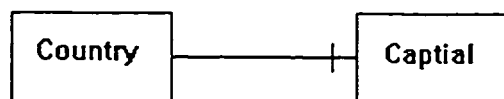


Figure 17. 1 with 1 cardinality constraint symbol

A zero cardinality constraint symbol means that an instance of one object type is not associated with any instances of another. It is also called zero association.

Different cardinality constraint symbols can be combined together to become a new cardinality constraint symbol. For example, in [Figure 18] a mailbox has 0, 1, or many messages. A person has zero or one wife.

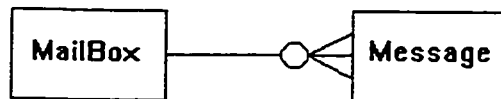


Figure 18. Zero or more cardinality constraint

For example, in [Figure 19] A person has zero or one wife.

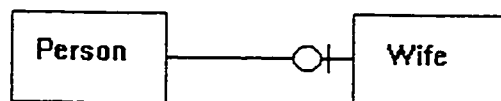


Figure 19. Zero or one cardinality constraint

2.3.5 Minimum and Maximum Cardinality Constraint

The cardinality constraints mentioned in the previous section can also be used to express a maximum and minimum constraint. A pair of cardinality constraint symbols are placed at each end of an association line, the maximum constraint is always placed next to the box to which it refers.

For example, in [Figure 20]. Each customer has zero to many orders and each order is for one and only one customer. When both of the minimum and maximum constraints are equal to one, it means there is only one choice. It uses a pair of l-bar symbols to express this min-max constraint.

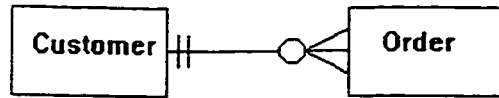


Figure 20. Min and max cardinality constraint

[Table I] shows all five min-max combinations:

Symbol	min	max
	0	1
	1	1
	0	more than 1
	1	more than 1
	more than 1	more than 1

Table I. Five Min-Max cardinality constraint combinations

2.3.6 Strengths and Weaknesses of Martin & Odell Notations

1. Analysis Notations are not Well-defined

This is not to indicate that the notations are not complete, simply that insufficient information is provided to access the completeness. There are no constructs to represent partitioning mechanism, even though Martin & Odell does mention

“realms” and “realm specifications” in their glossary, but no further references are provided to assist in determining how this relates to system partitioning.

2. Least Object-Orientedness

Martin & Odell is the least object-oriented, seemingly presenting slight extensions to information engineering with a heavy behavioral orientation. Booch seems to be the most object-oriented, with his strict emphasis on objects.

2.4 Rumbaugh (OMT) Notation

The Object Modeling Technique (OMT) was developed by James Rumbaugh, et al., at General Electric’s Research and Development Center [Rumbaugh]. The OMT is a methodology that covers the analysis, design, and implementation phases of an object-oriented system. It focuses on creating a model of objects from the real world problem domain and using this model to develop object-oriented software. In OMT methodology, problem domains are classified into 3 different models.

1. Object model - describes the objects in the system and their relationships;
2. Dynamic model - describes the interactions among objects in the system; and
3. Functional model - describes the data transformations of the system.

Each model is applicable during all stages of development and acquires implementation detail as development progresses. A complete description of a system requires all three models. The object model is the most important of the three models; it represents the static, structural, “data” aspects of a system. The dynamic model is a

state machine with events, states, activities, transitions and actions. The functional model is a set of data flow diagrams with the leaf functions being operations and actions. In this research, we are only interested in the static aspect of the method involves classes and their relationships; we will discuss the basic modeling concepts in Object model, present the corresponding OMT notations, and provide examples.

2.4.1 Object Model

An object model describes the problem domain with the static structure of objects, classes and links. The object model includes objects, classes, attributes, operations, links, associations, aggregations, generalization, and cardinality constraints. The object model is represented graphically with object diagrams containing object classes. Classes are arranged into hierarchies sharing common structure and behavior and are associated with other classes. Object diagrams provide a formal graphic notation for modeling objects, classes and their relationships to one another.

2.4.1.1 Object

An object is a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. All objects have identity and are distinguishable.

2.4.1.2 Class

A class describes a group of objects with similar properties (attributes), common behavior (operations), common relationships to other objects, and common semantics. Attributes and operations are specified for classes. An attribute is a data value which

is held by the objects in a class and it should be a pure data value, not an object. An operation is a function or transformation that may be applied to or by objects in a class. OMT uses a rectangular icon which may have as many as three sub-sections to represent a class, and the class name goes in the first section. The second section contains the list of attributes, and the last section contains the list of operations. Attributes and operations may or may not be shown; it depends on the level of detail desired. Each attribute must have a name and may include the data type and default value. Each operation must have a name and may include the argument list and return type. [Figure 21] shows the class notation in OMT. The notation is very similar to the Coad/Yourdon's class notation.

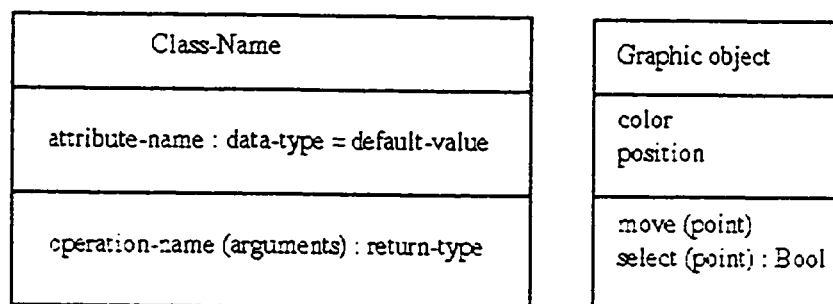


Figure 21. OMT notation for classes, and an example of a graphics object class

2.4.1.3 Association

A link is a physical or conceptual connection between objects. An association describes a group of links with common structure and common semantics. A link is an instance of an association. For example, a person *Works-for* a company. All the links in an association connect objects from the same classes and are bi-directional. OMT

uses a line between classes to represent an association as shown in [Figure 22].

Association names are italicized. Associations can have role names and qualifiers associated with each class connection.

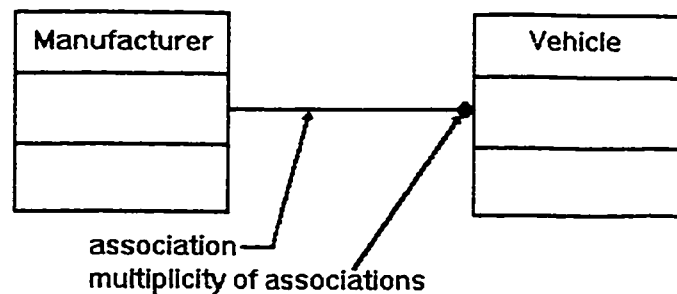
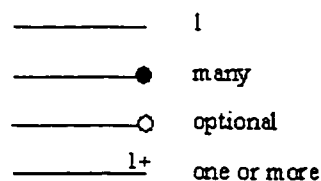


Figure 22. The association notation in OOA

Multiplicity specifies how many instances of one class may relate to a single instance of an associated class. The following is the different kind of multiplicity constraint symbols:



2.4.1.4 Aggregation

An aggregation is a special type of association that represents a “whole-part” relationship. Aggregations are transitive and anti-symmetric. OMT uses a diamond end on an association symbol to represent the aggregations shown in [Figure 23].

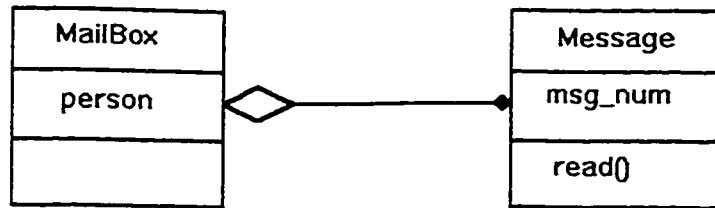


Figure 23. Aggregation relationship notation in OMT

2.4.1.5 Generalization

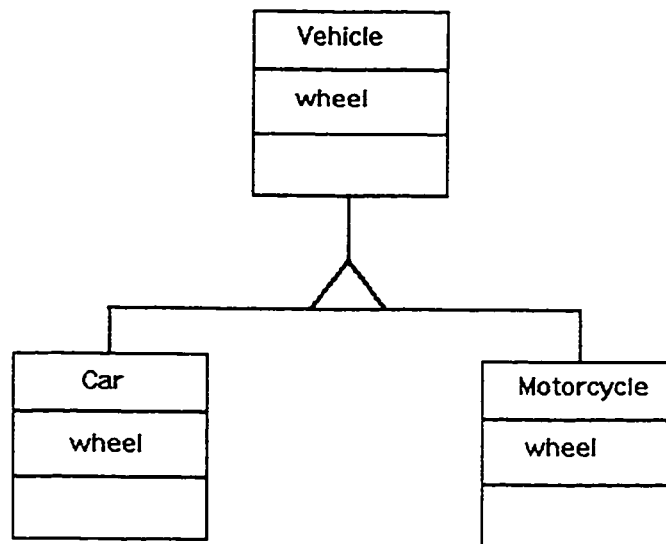


Figure 24. Specialization/Generalization notation of OMT

A generalization is a relationship between a class and one or more refined versions of that class. A generalization can have a discriminator, which is an attribute of an enumeration type that indicates the property of an object being abstracted by the generalization. A subclass may add new operation and attributes. This is called

extension. A subclass may also contain an ancestor attribute. This is called restriction. The generalization symbol is illustrated in [Figure 24].

2.4.1.6 Module

A module is a logical construct for collection of classes, associations, and generalizations. It is similar to the subject in the Coad/Yourdon notation, or class category in Booch.

2.4.2 Strengths and Weaknesses of OMT Notations

1. Analysis Notations are Reasonably Complete

OMT describes a complete methodology for object-oriented analysis which is similar to Booch's methodology, except OMT puts emphasis more on analysis and less on design. Analysis constructs are complete for low-level components, but higher levels of abstraction are not supported as well. There are no constructs to represent certain language-specific features, such as templates and class utility functions.

2. The Models Use Fairly Common Ideas

The Object models are similar to entity-relationship (ER) models. In essence, OMT object modeling is an enhanced form of ER. It improves on ER in the areas of expressiveness and readability.

OMT supports the choice of different object-oriented languages for the implementation of the design. [Rumbaugh] gives practical advice on the implementation of various constructs and the selection of a language.

4. Lack of Integration of the Three Models

The design analysis does begin to define attributes and operations as part of object classes. Integration of the three models is somewhat awkward, especially the functional model because it is not clear how the data flow diagram elements relate to the elements in the object model and the dynamic model.

CHAPTER 3

REVIEW OF EXISTING CASE TOOLS

Several commercial CASE tools support the notations that we have discussed and generate C++ code from the diagrams. Rational Rose is dedicated to the Booch notation, OEW uses the Martin & Odell notation, while Together/C++ focuses on the Coad/Yourdon notation. I had reviewed these three CASE tools in this chapter. Unfortunately, no tool that supports the OMT notation was accessible by me at the time of reviewing. Although each product supports different well-known methods and notations, each of them generates C++ code from the diagrams. Features of the tools will be discussed based on user-interface, source code partitioning, code generation, and the synchronization of code and diagram, i.e. reverse-engineering of existing code and updating of code when the diagrams change.

1. User Interface

Drawing classes and relationships in the diagram should be an easy task for the user. We will review how each tool adds, deletes and edits classes, data members, data functions, and various relationships in the diagram editor.

2. Source Code Partitioning

Some methodologies support the notation of grouping classes together to form higher levels of abstraction. For example, the Booch notation has class categories and the Coad/Yourdon notation has subjects. We would like to study how each CASE tool

implements this feature both internally within the tool and physically in the external file system.

3. Code Generation

A design CASE tool is effective and useful only if it integrates well with code generation. The information that is captured in the design diagram should be translated to the source code of the target language. The translations of class, data members, member functions and inheritance relationships are straightforward processes [Horst2]. All tools in this review should be able to produce correct C++ code. For example, class definitions and data member declarations are included in the header files, the stubs of the member functions are placed in the implementation files. One difficult aspect of code generation is the translation of aggregation relationships. For example, a mailbox has zero or more messages; the translated code should contain a data member message. But what kind of declaration should be generated for this message member?

It can be an array of message

```
Message message[10];
```

or it can be a pointer to a dynamically allocated array of messages

```
Message* message;
```

or it can be declared as a container array class of Message

```
vector<Message> message;
```

Suppose, the aggregation relationship is contained by reference; the data member can be declared as a pointer or reference to the aggregate class. There is no way that the tool would know what the correct generated code is. We will study how these CASE tools handle these ambiguous situations.

4. Synchronization of Code and Diagram

Upon initial code generation, the generated code could be updated with additional coding by the user. It is important that the subsequent code generation does not wipe out those changes and be able to merge in with the old code.

Also it is important for the tool to be able to read the changes in the code or parse the existing code from third parties, and update the diagrams accordingly. This process is called “reverse engineering.” Again it is not an easy task to translate the aggregations in the code to the diagram. For example, the code contains a data member of a container class template:

```
vector<Message> message;
```

The analyzer should be trained to understand the semantics of the vector< > template class and infer an aggregation relationship with 1 to many cardinality in the diagram. Furthermore, it is hard for the tool to identify the differences between data member and aggregation relationships by analyzing the code. For example,

```
class Employee
{
    Date    hire_date;
    Date    birthday;
};
```


The regenerated diagram can either display these data member declarations as an aggregation relationship between classes employee and Date or display them as data attributes in the Employee class. Since the Date class is a common trivial class like an integer or real type, the designer may want to see hire_date and birthday as attributes in the Employee class, instead of as aggregation relationships to Date class in the class diagram.

3.1 Rational Rose/C++

Rational Rose has produced a very extensive tool with complete and strict adherence to the Booch methodology presented in Chapter 2. Rose uses a model to represent the problem domain and the class diagram that contains class categories, classes, and the relationships between them. A model also contains modules, processors, devices, and subsystems which are described in [Booch94] to give a different view of the problem domain.

3.1.1 User Interface

The diagram editor user interface is easy to use (as shown in [Figure 25]). All the common Booch notations are found in a tool bar on the left hand side of the screen. A class model is created first by clicking on the desired class model and then clicking on the drawing area. A flashing class cloud appears at the cursor location on the screen. Class attributes and operations are entered via a dialog that is launched by double-clicking on the cloud shape

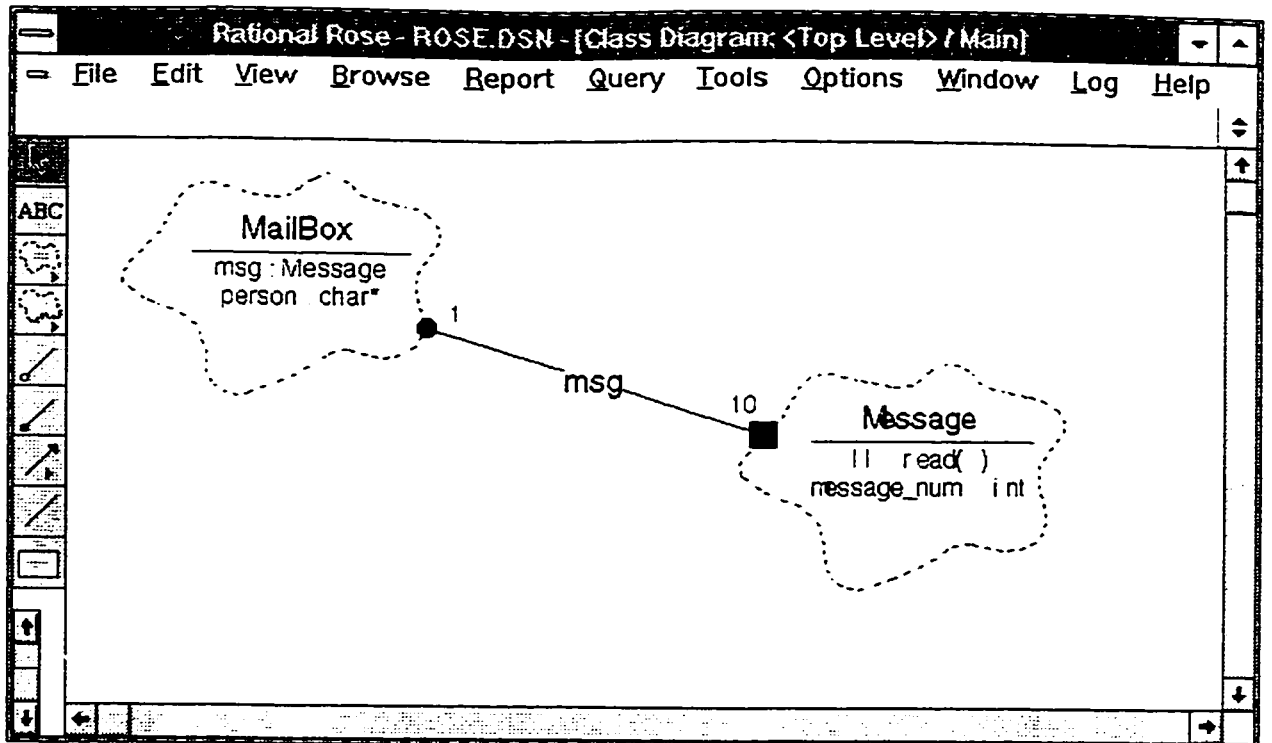


Figure 25. The Class Diagram Editor of Rational Rose

3.1.2 Source Code Partitioning

The generated files are rooted in a single project directory. The project directory represents the top level of the class diagram. Within the project directory, files are created in a hierarchy of subdirectories that corresponds to the class categories in the object model. Source files that reside in the project directory pertain to the class model in the top level of the class diagram. The sub-directory names are derived from the corresponding category names. A pair of header (.h) and implementation (.cpp) files are generated for each class model in the category or the top level class diagram. The name of each file is derived from the name of the corresponding class model.

In the Rational Rose tool, each class diagram is represented by a window. Clicking on the class category rectangular icon on a class diagram will open up a new window for adding classes in that class category. Unfortunately, there is no category hierarchy manager that can display the nested class hierarchy structures. Also the cut and paste of graphical items only works in the current window which means that one cannot copy class information from one class category to another.

3.1.3 Code Generation

Rational Rose correctly generates the code from the class diagram in [Figure 25]. The aggregation relationship between class MailBox and Message is translated into a data member declaration of type class Message in the class MailBox. A header file “mailbox.h” and a implementation file “mailbox.cpp” are generated for the mailbox class symbol in the class diagram. The complete generated code for the class MailBox is listed below:

```
// %X% %Q% %Z% %W%
//
//
// Module Specification MailBox (Package)
//
// File: mailbox.h
//
#ifndef mailbox_h
#define mailbox_h

###begin module.includes preserve=yes
###end module.includes

// Message
#include "message.h"

###begin module.additionalDeclarations preserve=yes
###end module.additionalDeclarations
```

```

// Class MailBox
//
// Concurrency: Sequential
// Persistence: Transient
// Cardinality: n
//
class MailBox
{
public:
    // Constructors
    MailBox();
    MailBox(const MailBox &right);
    // Destructor
    ~MailBox();
    // Assignment Operation
    const MailBox & operator=(const MailBox &right);
    // Equality Operations
    int operator==(const MailBox &right) const;
    int operator!=(const MailBox &right) const;
protected:
private:
    // Get and Set Operations for Has Relationships
    const char* get_person() const;
    void set_person(char*const value);
    const Message get_msg(const int index) const;
    void set_msg(const int index, const Message value);
private: // implementation
    ///begin MailBox::person.has preserve=no
    char*person;
    ///end MailBox::person.has
    ///begin MailBox::msg.has preserve=no
    Message msg[10]; // Data Members for Has Relationships

    ///end MailBox::msg.has
    ///begin MailBox%.declarations preserve=yes
    ///end MailBox%.declarations
};
#endif

```

```

// %X% %Q% %Z% %W%
//
//
// Module Body MailBox (Package)
//
// File: mailbox.cpp
//
///begin module.includes preserve=yes
///end module.includes

// MailBox
#include "mailbox.h"

///begin module.additionalDeclarations preserve=yes

```

```

###end module.additionalDeclarations

// Class MailBox
// Constructors
MailBox::MailBox()
###begin MailBox::MailBox%.hasinit preserve=no
###end MailBox::MailBox%.hasinit
###begin MailBox::MailBox%.initialization preserve=yes
###end MailBox::MailBox%.initialization
{
###begin MailBox::MailBox%.body preserve=yes
###end MailBox::MailBox%.body
}

MailBox::MailBox(const MailBox &right)
###begin MailBox::MailBox%copy.hasinit preserve=no
###end MailBox::MailBox%copy.hasinit
###begin MailBox::MailBox%copy.initialization preserve=yes
###end MailBox::MailBox%copy.initialization
{
###begin MailBox::MailBox%copy.body preserve=yes
###end MailBox::MailBox%copy.body
}

// Destructor
MailBox::~MailBox()
{
###begin MailBox::~MailBox%.body preserve=yes
###end MailBox::~MailBox%.body
}

// Assignment Operation
const MailBox & MailBox::operator=(const MailBox &right)
{
###begin MailBox::operator=%%.body preserve=yes
###end MailBox::operator=%%.body
}

// Equality Operations
int MailBox::operator==(const MailBox &right) const
{
###begin MailBox::operator==%.body preserve=yes
###end MailBox::operator==%.body
}

int MailBox::operator!=(const MailBox &right) const
{
###begin MailBox::operator!=%.body preserve=yes
###end MailBox::operator!=%.body
}

// Get and Set Operations for Has Relationships
const char* MailBox::get_person() const
{
###begin MailBox::get_person%.get preserve=no
return person;
}

```

```

    ///end MailBox::get_person%.get
    }

    void MailBox::set_person(char* const value)
    {
    ///begin MailBox::set_person%.set preserve=no
        person = value;
    ///end MailBox::set_person%.set
    }

    const Message MailBox::get_msg(const int index) const
    {
    ///begin MailBox::get_msg%.get preserve=no
        return msg[index];
    ///end MailBox::get_msg%.get
    }

    void MailBox::set_msg(const int index, const Message value)
    {
    ///begin MailBox::set_msg%.set preserve=no
        msg[index] = value;
    ///end MailBox::set_msg%.set
    }

    // Additional Declarations
    ///begin MailBox.declarations preserve=yes
    ///end MailBox.declarations

```

3.1.3.1 Redundant Member Functions

By default, the generated code automatically contains default constructor, default destructor, copy constructor, and get/set member functions for the private member variables for every class model. However, these member functions are sometimes unnecessary in the definition of the class. Not every class needs a destructor or copy constructor, and not every private member data needs a pair of get/set member functions to access them.

3.1.3.2 Support Iterative Development

Comment delimiter tags in the generated code are used to indicate where one should add code for the implementation of member functions. Each comment delimiter begins

with “`///##”` in the C++ source line. The code changes are preserved in the next iteration of code regeneration. However, any changes to the class definitions such as deleting and renaming classes, must be made through the model editor to regenerate the C++ code. For example, the shaded area in the mailbox.h listing contains the attribute declarations for msg and person. Any changes to these declarations are not preserved in the next iteration of code generation since they are embraced by the comment delimiters with “`preserve=no`” command.

3.1.4 Synchronization of Code and Diagram

The 2.0 version Rose/C++ tool that I reviewed does not have the ability to import existing code. Rose/C++ version 2.5 allows reverse-engineering of an existing project by using a C++ code analyzer; a separate executable provided in the same package. The analyzer lets the user prepare the existing source with comments to separate the portion of code that should or should not be changed by the tool. After the user prepares the source, the analyzer parses it to create a new model file that can be opened in Rose diagram editor. In fact, the analyzer is the middle-man for updating the model when changes occur in the coding stage. The synchronization is somewhat inefficient and inconvenient, because the diagram updating is triggered by a separate program rather than by an integrated feature within the tool. All updated changes have to be reloaded from a file to the diagram editor.

Rose translates each data member of types other than standard (i.e. int, real) into aggregation relationships. It has difficulty translating types such as `array<Message>`.

It treats all these types as classes, i.e. creates bogus classes for `size_t` and `void`. The resulting class diagram ended up containing numerous unnecessary classes. Thus the reverse-engineering process of the tool is not suitable for any real world software development since it requires a lot of corrections by hand.

3.1.5 Data Storage

All the data information of the class diagrams are stored in the internal database which is hidden from the user. Any change to the generated C++ code does not affect the actual class models in the class diagram editor. This kind of data storage makes the code to diagram synchronization difficult to maintain. Rose chooses to use an external program to convert the source files back to the class diagram.

3.2 Together/C++

Together/C++ supports the Coad/Yourdon notation presented in Chapter 2. It is totally different from the Rational Rose in terms of storing the design model information. Together/C++ keeps all the design model information in the form of structured comments in the C++ source files. This approach entirely eliminates the synchronization problem that we have found in Rational Rose. It uses a very fast built-in incremental C++ parser to parse the source files whenever changes are made to the code.

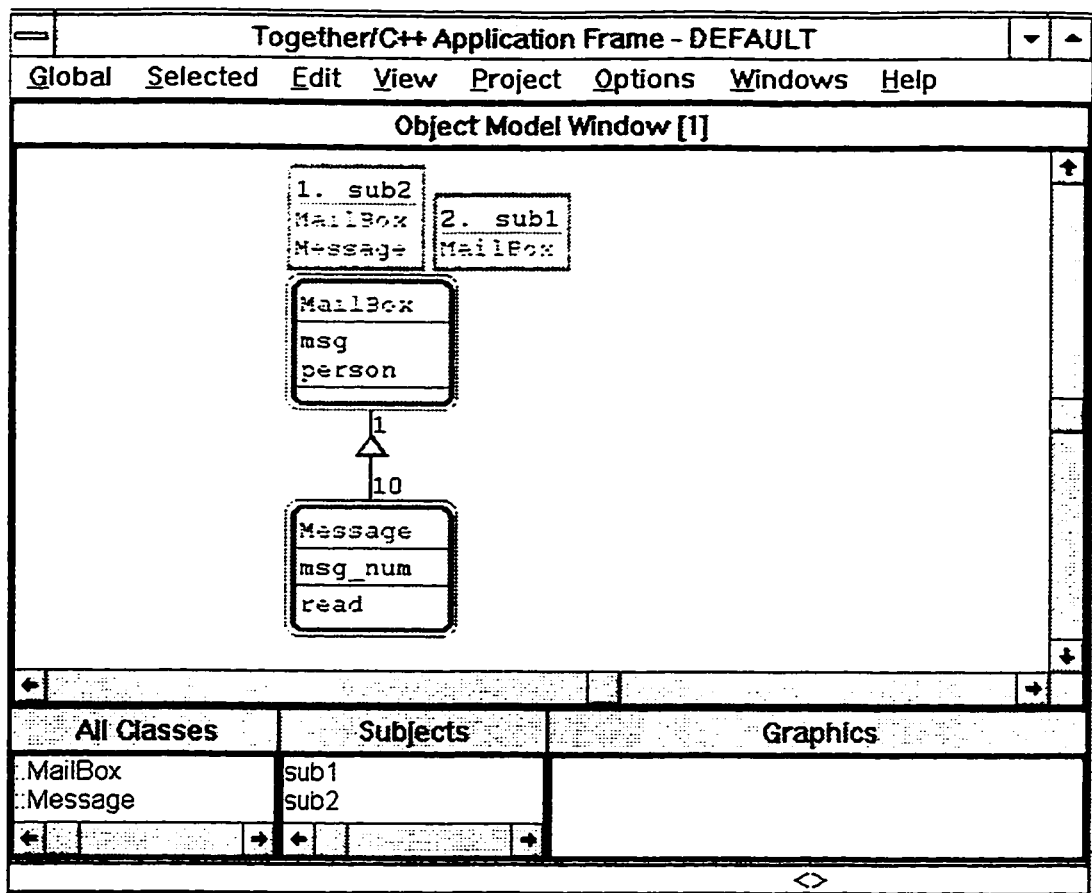


Figure 26. Application Framework of the Together/C++

3.2.1 User Interface

Together/C++ has an automatic layout feature for the class diagram. The automatic placement feature can be turned off if the user wants to manually place and move the classes and connections in the diagram. The user interface is awkward to use and is unlike standard MDI Windows applications (as shown in [Figure 26]). The lack of a tool bar makes the user interface user-unfriendly. The user can create classes and

connections in the diagram editor using hotkeys or by choosing the appropriate command from the pop-up menu.

3.2.2 Source Code Partitioning

Together/C++ uses sub-directories in the file system to model the subject notation. Grouping classes with the same subject is allowed in Together/C++. However Coad's notation only supports one level of nesting with the subject notation. Unlike Booch notation where one can nest class categories inside another category, no subject nesting is allowed in Together/C++. Each class can be assigned to one or more subjects and Together/C++ provides a selective subject display filter that allows user to hide or show subjects that he/she wants to display. When creating a subject in the diagram editor, a pop-up "create subject" dialog is opened to prompt for the directory name of the newly created subject (as shown in [Figure 27]). The user can also specify a matching pattern for the subject. By default, any newly created class matches the subject pattern that is automatically assigned to that subject. For example, all the classes under the project Cloud9's diagram editor unit start with the prefix DE_, while the classes under the hierarchy structure editor start with the prefix HS_. The main advantage of this feature is that it allows the user to partition the project source files into different sub-projects. For example, all files related to the diagram editor are prefixed by DE_.

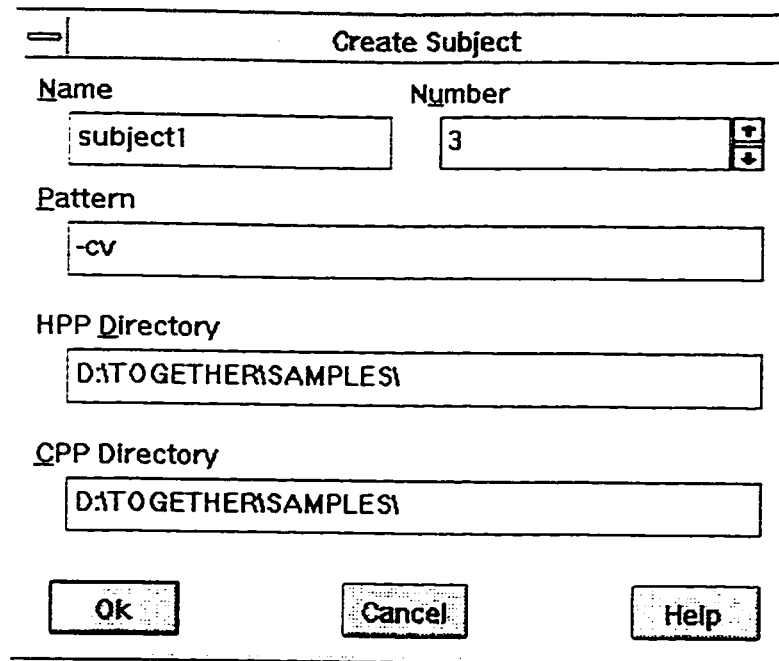


Figure 27. Create Subject Dialog in Together/C++

3.2.3 Code Generation

Together/C++ maps each class-object symbol to a pair of C++ header (“xxx.hpp”) and body (“xxx.cpp”) source files. When the user creates classes in the diagram editor, a pop-up dialog is opened to ask for specific class information for the new class-object. Using the name of the class the user entered, the tool will automatically create the filenames for the corresponding C++ header and body files. Users can also enter their own filenames and directories. As shown in [Figure 28], Together/C++ chooses to use C++ source code as the storage media. It keeps all design information together with the source code. Information which can be mapped to C++ source code unambiguously, is stored as C++ source code redundancy free. All other information is stored in the form of special C++ comment structures. These comments structures

serve as implementation guidelines and are understood by the internal parser which is able to re-engineer all information directly from the source code. In the example below, the shaded area is the design information generated by Together/C++. It specifies the aggregation relationship between class Mailbox and Message, the relationship cardinality, and the subject name to which the class belongs.

The image shows a dialog box titled "Create Class with Objects". It contains several input fields and buttons. The "Name" field contains "MailBox" and the "Subject" field contains "sub1". The "HPP File" field contains "MailBox.HPP" and there is a "Select HPP File" button. The "HPP Directory" field contains "F:\GLENM\THESES\ITG1\SUB1\". The "CPP File" field contains "MailBox.CPP" and there is a "Select CPP File" button. The "CPP Directory" field contains "F:\GLENM\THESES\ITG1\SUB1\". At the bottom, there are "Ok", "Cancel", and "Help" buttons.

Name	Subject
MailBox	sub1

HPP File: MailBox.HPP [Select HPP File]

HPP Directory: F:\GLENM\THESES\ITG1\SUB1\

CPP File: MailBox.CPP [Select CPP File]

CPP Directory: F:\GLENM\THESES\ITG1\SUB1\

[Ok] [Cancel] [Help]

Figure 28. The Create Class Dialog in Together/C++

```

#include "message.hpp"
class MailBox
{
    /*design part Message class Message <"1", "10">*/
    /*design in subject "sub1" */
    /*design in subject "sub2" */
    person;
};

```

The generated C++ code does not include the data member declaration for the whole-part relationship between class MailBox and Message. Instead it includes the comments of the design information of the whole-part links. The programmer needs to open the header file from the editor and adds the appropriate data member declaration for this relationship. For example,

```

    Message msg[10];

```

As mentioned earlier, the code generation for the aggregation relationship poses ambiguous decisions on the declaration of the data members. Together/C++ chooses to allow the user to fill in the appropriate declaration code in the generated C++ source code.

When the designer adds an attribute in the diagram editor, the C++ type is not entered into the design information. Instead the designer has to enter this specific type in the generated C++ source code through an editor. For example,

```

    char* person;

```

The same approach is used for the member function return types.

3.2.4 Synchronization of Code and Diagram

At run time of Together/C++, all the design model and source code information are parsed and stored in a transient repository which is like an extended form of a symbol table. The transient repository is never updated directly. Any change to the model causes the code generator to update the C++ source code. Then the built-in parser will re-parse the affected code and update the repository. Repository updates trigger updates of all views of changed data. Since there is only one location for the design information (the C++ source files), the synchronization of code and diagram is basically transparent in Together/C++.

Using the C++ source as the storage media allows the user to use other file tools (i.e. an external editor). Moreover, it makes the editing of the class member data and functions easier. By using the source editor, the designer can enter the data member and functions directly into the source files from an editor instead of going through numerous dialogs in the diagram tool. For example, the designer can edit the class definitions in the C++ source code from an external editor and Together/C++ reparses the source and updates the diagram.

3.2.5 Reverse Engineering

Reverse engineering does not support the conversion of Whole-Part relationships. Existing C++ source code is parsed and represented by the Coad notation in the

diagram editor. Only classes, attributes, services and Gen-Spec structures are extracted. Whole-Part relationships may be entered by the diagram editor or text editor afterwards. They are stored as special comments in C++ code. The obvious short-coming of this approach is the inability to translate whole-part relationships. It makes the diagram after parsing the existing C++ code less useful and complete. All data members are translated as attributes in the class.

3.3 OEW for C++

OEW is quite distinct from the other two reviewed CASE tools because of its well-planned features. It supports the James Martin & Odell notation presented in Chapter 2. It also supports database development and third party class libraries. It lets the programmer import third party C++ class libraries for building GUIs. OEW will parse the header files of the libraries, and convert the class information into diagrams. The designer can then inherit from these GUI classes and create their own derived classes. It is the only tool I have seen that supports template container classes in the aggregation relationships. The code generation also support class persistence by generating additional macros in the class definitions.

3.3.1 User Interface

OEW has two different views of the diagram as shown in [Figure 29]. The hierarchical view is the default view window that shows the inheritance relationship between classes and the class properties of the classes (attributes, methods). The

relationship view shows the relationships of the classes. There are three types of relationships one can create in the relationship window: reference (association), containment (aggregation) and inheritance. Each of these relationship icons are shown with different colors, making the graphs more readable. OEW also provides various choices of filters; e.g. show/hide public “slots,” show Methods only etc. “Slots” in OEW refers to a compartment containing attributes or methods. OEW also comes with a built-in full-syntax highlighting source code editor and a built-in C++ parser. It allows the user to edit code within the OEW environment and allows changes made externally to be imported easily by the C++ parser.

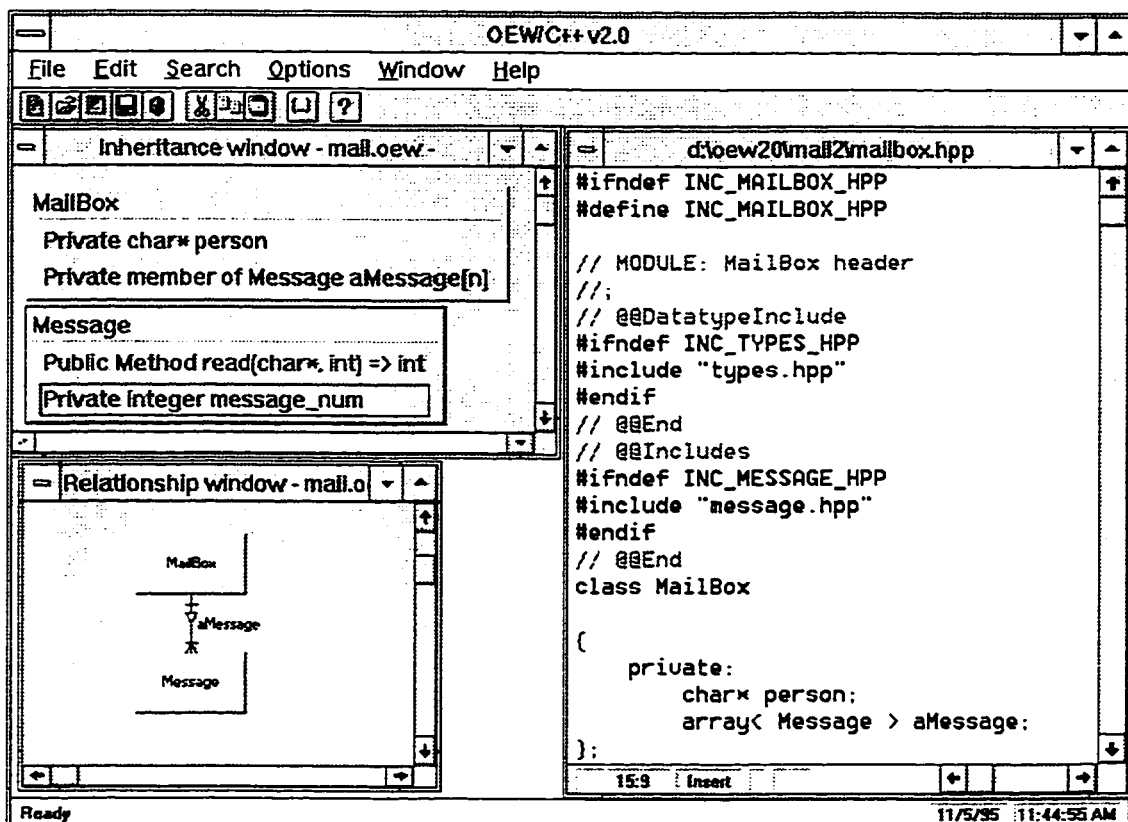


Figure 29. The User Interface of OEW

Furthermore, one can define different views of the relationship model that focuses on parts of the entire model. Displaying selected class icons from the current view is an option.

3.3.2 Source Code Partitioning

Martin & Odell notation does not support class categories. But one can open multiple relationship windows displaying different parts of the entire diagram.

3.3.3 Code Generation

The class names are used to name the source files to be generated, but one can override the default filenames with the Module dialog.

3.3.3.1 Reference Relationship (association)

OEW supports association relationship, i.e. the general many-to-many relationship between two classes. It generates the macros representing each side of the 1-to-many relationship between associated classes. For example, a car manufacturer makes many models of vehicle. OEW make a reference (association) relationship between class Vehicle and Manufacturer, and generated macros shown below implement this relationship.

```
class Vehicle
{
    os-relationship_1_m(Vehicle, _maker, Manufacturer, _vehicles, Manufacturer*) _maker;
}

class Manufacturer
{
    os-relationship_1_m(Manufacturer, _vehicles, Vehicle, _maker, os_list) _vehicles;
}
```

3.3.3.2 Containment Relationship

The designer can train OEW to recognize special data types and container classes by using the data type dialog. (See [Figure 30]).

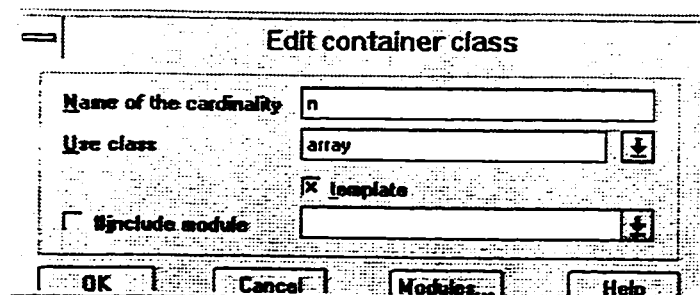


Figure 30. The Container Class Specification Dialog

The cardinality specified during the creation of the containment relationship determines the actual data member declaration. OEW will scan the container class lists and match the cardinality of the containment relationship to the one that are specified in the container class. For example, if the user specifies the array template container class with a cardinality n and the same cardinality n in the containment relationship, the generated code will look like this :

```
array<Message> msg;
```

If the cardinality is a constant number, say 10, the default generated code is an array.

```
Message msg[10];
```

If the user has other container classes, say a list with cardinality m, and the user uses the m as the cardinality in the containment relationship, the produced code will be like:

```
list<Message> msg;
```

Each container class is mapped to a unique cardinality identifier.

3.3.3.3 Forward Declarations

All the generated source files include the global type header file, which includes the user defined data type declarations and the forward declarations of all the classes.

This global inclusion prevents any circular class reference errors. For example, global header file `types.hpp` is shown below.

```
// MODULE: Global types;
// @@Includes
// @@End
// @@Forwards
class MailBox;
class Message;
// @@End
#endif
```

3.3.3.4 Schema Generator

Object-oriented databases use extended C++ syntax, which allow the preprocessor of the database to generate database schema. OEW supports the translation of a C++ class hierarchy (or part of the class hierarchy) into the corresponding schema of the database by Object Store¹, POET². Additional database macros are added to the classes that one might instantiate using persistent `new`. For example,

```
static void dummy()
{
    OS_MARK_SCHEMA_TYPE( MailBox );
    OS_MARK_SCHEMA_TYPE( Message );
}
```

¹ ObjectStore is an object-oriented database system by Object Design, Inc..

² POET is an object-oriented database by POET Software.

3.3.4 Synchronization of Code and Diagram

OEW keeps the design information in a single ObjectBase, which allows the tool to relate class definitions and implementation code to actual source files. The built-in parser allows changes made to the generated code externally to be imported into the tool. It also keeps track of the time-stamp of the generated files. If it detects that a file has been changed externally, OEW alerts the user to import that module rather than regenerating it. For example, after the initial code generation from the design class diagram, the programmer can use Borland C++ IDE for maintaining source code and building the application. Once it compiles, he/she can import those changes into the OEW ObjectBase.

It correctly regenerates the data member declaration of types such as `array<Message>` to a “1 to many” aggregation relationship between Mailbox and Message.

3.4 Conclusion

While all three products contain rules to prevent illegal connections on a diagram and provide correct generated C++ source code on classes, attributes, methods and inheritance, none of the tools reviewed so far provide all the features that a programmer wants from a CASE tool. Rational Rose has the most user-friendly interface and is committed completely to the Booch notation. However the lack of built-in support of reverse-engineering makes the tool less appealing than the other two CASE tools. Together/C++ comes close to being the perfect tool for storing all the design information in the C++ source. It eliminates problems of synchronization of

code and diagram. Also the ability to edit the class information (data member and function) from an ASCII editor is most desirable to most programmers. It is the only tool that comes with version control. Unfortunately, the user interface is cumbersome to use.

OEW is a fascinating product. It is the only tool with a container class trainer that understands the semantics of container class templates and regenerates the correct 1 to many aggregation relationship from the C++ source code (i.e. `array<Message> msg;`). However, the use of Martin/Odell notation makes the graphical diagram less expressive and not as complete as Rational Rose for supporting C++. The Martin/Odell notation has no hierarchical clustering, and class hierarchy is only supported in class inheritance. As in Rational Rose, collections of classes can be assigned to different categories. OEW covers more aspects of development with the use of a database, analyzing third party GUI libraries (OWL, MFC) as well as application modeling.

CHAPTER 4

DESIGN AND IMPLEMENTATION OF THE CLASS DIAGRAM EDITOR

The diagram editor is an essential component of the CASE tool in CLOUD9. It is the utility for drawing class diagrams using the class component of the Booch notations. Typically, classes are inserted to a diagram by adding clouds, and relationships are inserted by adding connectors. Therefore, the class diagram, class symbols and class connectors need to be modeled. I will identify the requirements of these objects and will discuss the design and implementation of this editor. Since the connector type used for representing a relationship depends on the type and attributes of the relationship, a relationship specification dialog box is built for the input of this information. Details of data members and member functions of a class can be entered via the hierarchical structure editor. Since the class diagram editor and the hierarchical structure editor are linked internally, this information must be reflected in the class diagram. In this design, class relationships are inferred. If a class has a field of another known class, an aggregation is inferred. If an operation has an argument or return value of another class, a usage relationship is inferred. To help the editor determine whether the type of a class data member is a simple data type, a class, or a container class template, the member type checking dialog box and the container specification dialog box are implemented. To avoid cluttering of information in the class diagram, the hide/show dialog box is implemented for selective suppression of display of relationships.

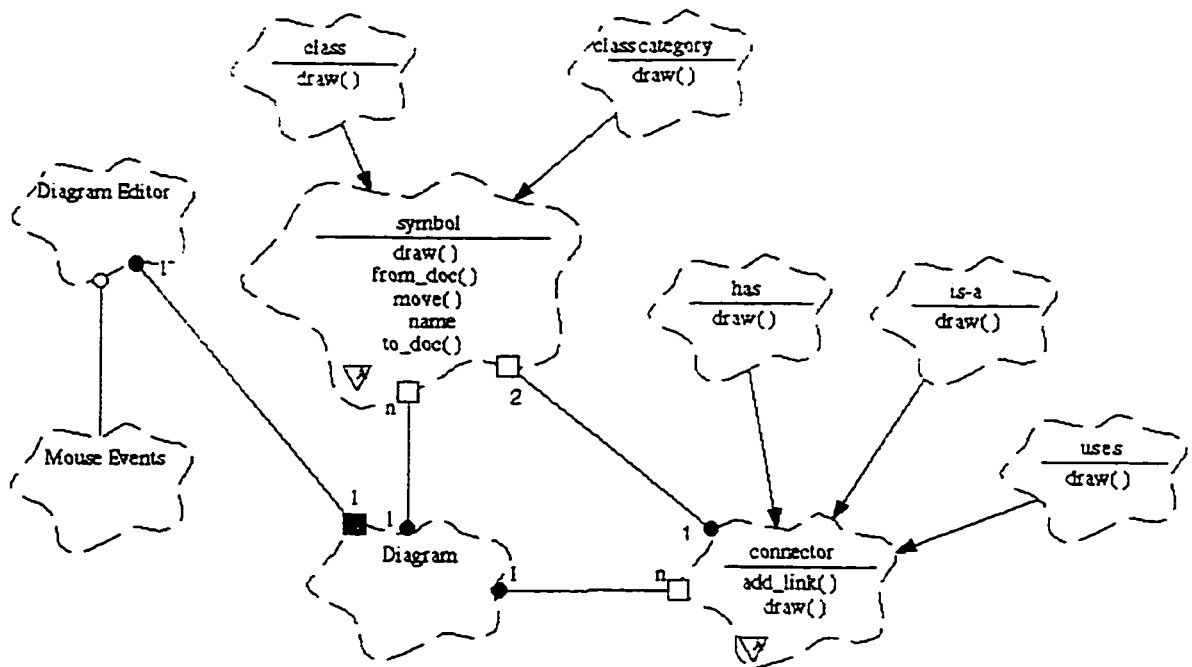


Figure 31. The Class Diagram for the Class Diagram Editor

4.1 Modeling of Class Diagrams

[Figure 31] shows a class hierarchy that models the things in a diagram editor. It shows that a class *Diagram* is an area upon which the user can place graphic items like symbols and connectors. The abstract class *Symbol* defines the common data and functionality of all class symbols that can be placed on the diagram editor. The classes that descend from *Symbol*, *Class* and *Class Category* all overload the `draw()` member function to draw specific kinds of shapes. The class *Connector* defines the common data and functionality of all types of relationships between classes. The class *Diagram* is really the central class of the diagram editor tool.

4.1.1 Requirements of Class Diagram

1. Serve as a drawing board with a coordinate system.
2. Remember the symbols and connectors that it contains.
3. Translate information represented in the diagram into a format that both the hierarchical structure editor and the code generator can easily and efficiently decode.
4. Reverse-engineer a class diagram from a code document.

4.1.2 Implementation of Class Diagram

Since the application uses a multiple-document interface (MDI), the first requirement can be easily met by modeling the diagram as a view window into which graphical objects can be added. The logical coordinate system that I use assumes (0,0) to be the upper left corner of the diagram. The X coordinate increases to the right and the Y coordinate increases down the diagram. The actual display symbol sizes are fixed in logical units. But the actual number of pixels represented in each unit varies with diagram scale and target device.

To handle the second requirement, the diagram object will maintain a dynamic array of class symbols and another dynamic array of connectors. These two arrays will grow or shrink as class objects and relationships are being added or deleted. The member data `_cloud_array[]` and `_connector_array[]` is defined to hold the information of the symbols and connectors the diagram contains to satisfy requirement 2.

The third requirement involves parsing the lists of symbols and connectors and saving the results in a code document. The code document will then be used to generate the class hierarchy and to generate C++ code. The implementation of the code document will be discussed in the next chapter. To fulfill requirement 3, member functions such as AddCloudToDoc (Chi_String name), AddFieldCloudToDoc (Chi_String name, type, name) and AddParentCloudToDoc (Chi_String name, parent_name) are implemented to enter design information to code document.

The fourth requirement involves traversing the structure tree, adding class symbols and connectors to the class diagram lists and assigning locations for the symbols and connectors. To fulfill requirement 4, member function doc_class_to_cloud (Document* theDoc) is used to place the initial classes from the document to the diagram screen. An automatic placement algorithm is used to calculate the initial position of the class symbols.

This listing shows the C++ interface for the CV_Diagram class.

```
class CV_Diagram
{
private :
    Chi_Array<CV_Cloud>    _cloud_array;           // list of class symbols on the
diagram
    Chi_Array<CV_Connector> _connector_array;     // list of connectors on the diagram
public:
    void    Display(TDC& dc);                    // Paint the diagram
    void    DocClassToCloud(Document* theDoc);   // places the initial classes into
```

```

// diagram
void UpdateCloudInDoc (int index, Chi_String name, Chi_String old_name);
void DeleteCloudFromDoc (int index, Chi_String name);
void AddParentCloudToDoc (Chi_String name, Chi_String parent_name);
void AddFieldCloudToDoc (Chi_String name, Chi_String field type, Chi_String
filed_name);
void AddCloudToDoc (Chi_String name); // Insert class into document
// database
};

```

4.1.3 User Interface

When the application is executed, a blank diagram view is initially displayed. To create a new diagram view, select File-New from the menu. To load a diagram from an existing C++ project, select File-Open from the menu. The actual application user interface is shown in [Figure 1].

4.2 Modeling of Class Symbols

In this implementation, a class symbol can only be used to represent a class. It is represented as a graphical icon of rectangular shape or cloud shape in a class diagram.

4.2.1 Requirements of Class Symbol

1. Maintain class information that is common to all class symbols. (e.g. class name).

2. Maintain methods and properties pertinent to the graphical representation of a class symbol. (e.g. relative position in the class diagram, the frame enclosing the symbol, and how the shape should be drawn).
3. Maintain information for a class. (e.g. list of attribute names, list of method names and arguments, members).
4. Be extensible so that it can be used to represent other class symbols supported in Booch notations, such as parameterized classes or class categories.

4.2.2 Implementation

To fulfill requirement 4, the class symbol is defined as an abstract base class `CV_Symbol` which holds information specified in requirements 1 through 2. Another class, `CV_Cloud`, which will be used to represent the cloud symbol, is derived from this base class `CV_Symbol` to satisfy requirement 3.

This listing shows the C++ interface for the `CV_Symbol` abstract base class.

```
class MemberHideShow
{ // Hide Show Class
public:
//etc ...
private:
    Chi_String  _name; // the member name
    Bool        _hide; // hide or show flag
};
```

The member `_name` is used to store the full declaration of a member variable or function in a class including the access right. For example, string name “public int wheel” is used to specify the public member variable `wheel` with type `int`.

```
class CV_Symbol
{
public :
    BOOL          contains(const TPoint& point) const;
    BOOL          intersects(const TRect& rect) const;
    void          move(int dx, int dy);           // moves symbol w/ offset
    virtual void  draw(TDC& dc) const = 0;       // draw the symbol
    virtual void  draw_focus(TDC& dc) const = 0; // draw the selected symbol
    virtual void  display_name(TDC& dc) const = 0; // draw the symbol name
    TRect         invalirect() const;

// etc ...

private :
    Chi_String    _name;           // Class name
    BOOL          _hide;           // Hide or show the symbol
    TRect         _frame;         // Rectangle frame around symbol
    double        _scale;         // Current scale logical units/pixel
};
```

This listing shows the C++ interface for the `CV_Cloud` class.

```
class CV_Cloud : public CV_Symbol
{
public :
```

```

void            insert_member(Chi_String cs);    // insert data members
virtual void    draw(TDC& dc) const;           // draw the symbol
virtual void    draw_focus(TDC& dc) const;     // draw the selected symbol
virtual void    display_name(TDC& dc) const;   // draw the symbol name
virtual void    display_methods(TDC& dc) const; // draw the operations
virtual void    display_members(TDC& dc) const; // draw the attributes

// etc ...

private :

    static POINT          Logical_coor[61];

    Chi_Array<MemberHideShow>  _members;    // list of methods & attributes
};

```

4.2.3 User Interface

- To add a symbol, first click on the cloud button in the class diagram editor toolbar to change into symbol creation mode. Each subsequent click on the target class diagram will produce a new class symbol. After a symbol has been added, the hierarchical structure diagram will be updated and the programmer can then edit the name or enter attributes and methods of the newly added class using the hierarchical structure editor.
- To move a class symbol, first click on the selection button in the toolbar to change into edit mode, then click and drag the target class symbol to the new location.
- To delete a class symbol, switch to edit mode, click on the target class symbol and press the Delete key. To avoid information-cluttering in the class diagram,

hide/show a class symbol, its links and attributes by using the Hide/Show Filter dialog box. To access this dialog, select Edit-Hide/Show from the menu.

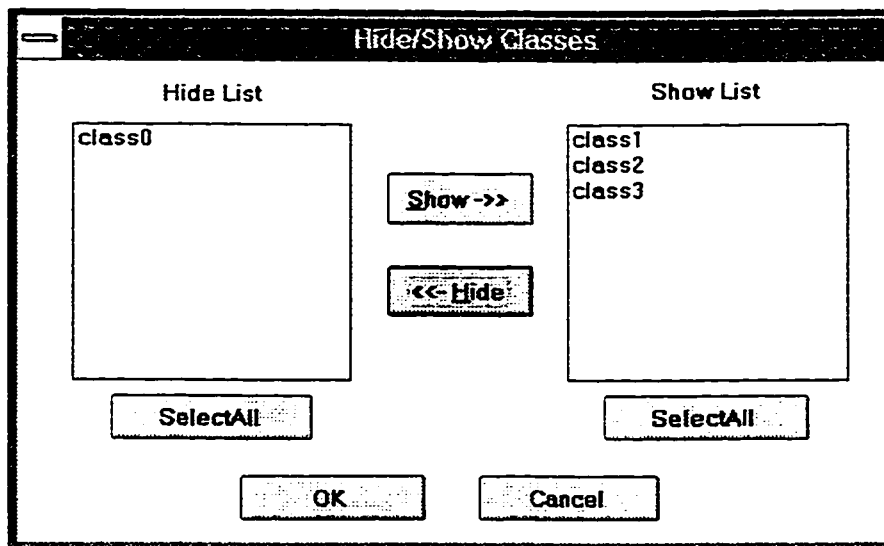


Figure 32. Hide/Show Classes Dialog

4.3 Modeling of Class Connectors

A class connector connects two class symbols and is built relative to the symbols. It is represented by an arrow or a line with special adornments of filled or unfilled squares or circles. There are three types of relationships to be modeled, namely the *has* relationship, the *is-a* relationship and the *uses* relationship. A class connector representing a *has* relationship is a line with a solid circle on one end. A class connector representing an *is-a* relationship is shaped like an arrow. A class connector representing a *uses* relationship is a line with an open circle on one end. The positions of the graphical adornments are determined relative to the connector's ends.

A connector is defined as an abstract base class. Each relationship connection is a specific connector derived from it. When a connector is created in the class diagram, the user will be asked to supply specifications of the relationship. The relationship specification information is directly related to the shape and kinds of adornments on the connectors.

4.3.1 Requirements of connector

1. Maintain relationship information. (e.g. connector name, type, cardinality, clouds it connects).
2. Maintain methods and properties pertinent to the graphical representation of a connector symbol. (e.g. relative position in the class diagram and how the connector should be drawn).

4.3.2 Implementation

The design of the connector class `CV_Connector` is similar to that of the symbol class `CV_Symbol`. However a class symbol variant like `CV_Cloud` is represented as a derived class whereas a connector variant is distinguished only by the values of the connection type, aggregation type and cardinality. The class diagram in [Figure 31] shows the class connector as an abstract class to provide clarity in the design.

The name of the cloud is used to specify the starting and ending connections of the connector. The reason we choose to use the name instead of the pointer to the cloud

is because of the address of these clouds can be changed dynamically. The clouds are stored in a dynamic array in the diagram class (see section 4.12).

This listing shows the C++ interface for the connector.

```
class CV_Connector
{
public :
    enum ConType {USE, HAS, INHERIT};
    enum ValType {BYREF, BYVAL}; // ARM - added for labels
    enum QtyType {ONE, N, SOME}; // ARM - added for labels, i.e. 1->n
    BOOL          contains(const TPoint& point) const;
    void          assign_name(ClassLinkXfer &xfer, int index);
    virtual void  draw(TDC& dc) const = 0;           // draw the connector
    virtual void  draw_focus(TDC& dc);              // Display as selected.
// etc ...
private :
    CV_String     _startConnect; // connection starts
    CV_String     _endConnect;   // connection ends
    Chi_String    _name;        // connector label
    double        _scale;       // current scale logical units/pixel.
    ConType       _type;        // kind of relationship
    ValType       _vtype;       // by reference or by value
    QtyType       _qtytype;
    TPoint        _i1, _i2, _i3, _i4;
    BOOL          _hidden;
};
```


4.3.3 User Interface

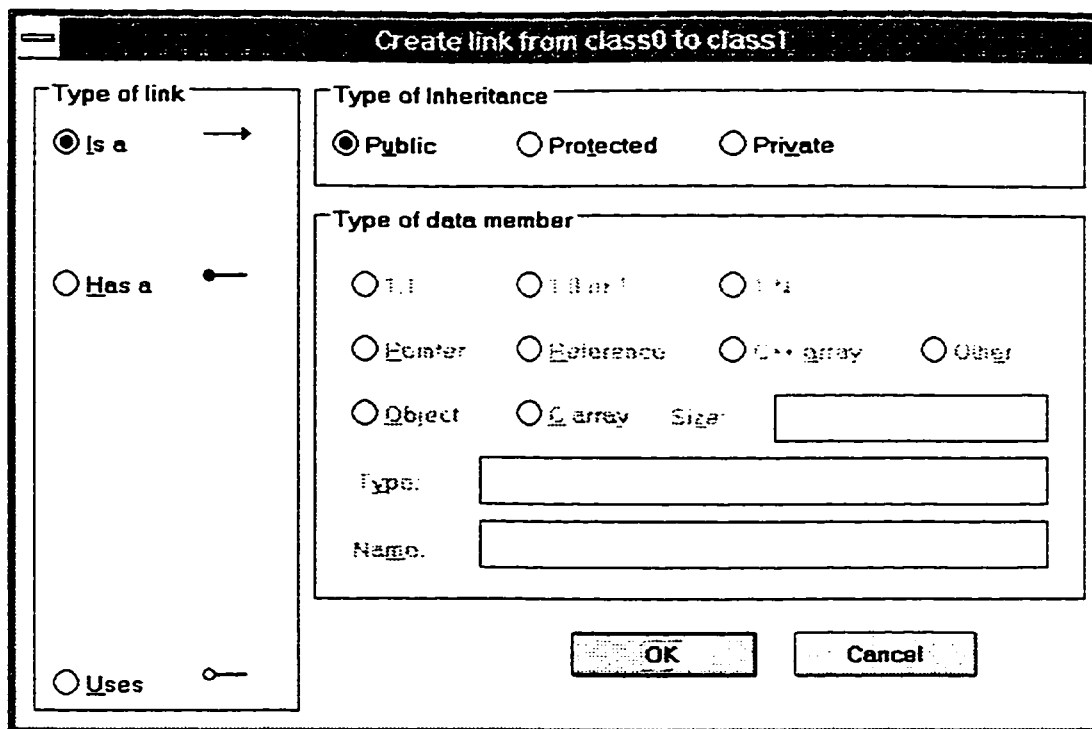


Figure 33. Relationship Specification Dialog

- To add a connector, first click on the connector button in the class diagram editor toolbar to change into connector creation mode. The user should then click on the first target class symbol, drag the mouse to the second target class symbol and release. After a connection has been made, the relationship specification dialog will be displayed. The user must specify the type of relationship and related properties for proper C++ translation. For the *is-a* relationship, the user can choose from one of the three possible types: public, protected, or private. The

default type of inheritance is public. For the *has* relationship, the user can choose different cardinality, containment, and the possible C++ translation.

- To edit a relationship, double-click on the connector to access the relationship specification dialog.
- To delete a relationship, switch to edit mode, click on the target connector and press the Delete key.

4.4 Special Features

In the object-oriented world, the type of a data member is always a class. In reality, it can be a simple data type such as integer. To avoid cluttering the information in a class diagram, all standard data type classes such as integer and string, and all *has* relationships involving these data types are not shown in the class diagram. The user can use the member type checking dialog to specify additional user-defined classes that should be hidden from view.

The member class type added via this dialog is treated as a simple type field and the corresponding data member is translated into an attribute of the class.

The following C++ class interface is used to keep track of the types of class members registered using this dialog.

```
class MemberTypeFilter {  
    Chi_Array<Chi_String> filter_list;  
    Boolean match_filter(Chi_String class_name); }  

```

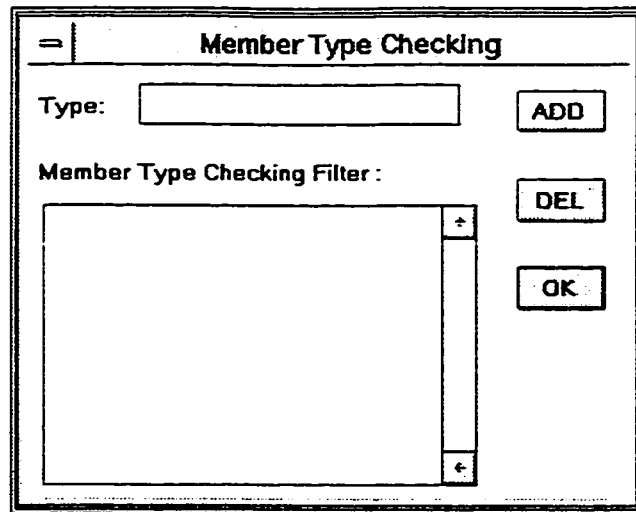


Figure 34. Member Type Checking Dialog

The type of a data member can be a simple data type, a class, or a container class. With the Member Type Checking dialog, we can differentiate between simple data types and a class.

It is common in C++ to model a 1 to many aggregation relationship by using a container class template. For example,

```
class MailBox {
    array<Message> m;           // a C++ array of class Message
};
```

The declaration of “array<Message> m” can be treated as 1 to 1 or 1 to many aggregation relationship between MailBox and Message. because the tool do not understand the semantic of the container class template array. In order to understand the semantic of a container class, we need the Container Class Specification dialog.

This dialog is used to specify a set of container class templates and their corresponding cardinality. If a programmer uses one of the specified container classes in a data member declaration, the graphical editor will reverse-translate this data member declaration into a *has* relationship with the cardinality specified in the dialog. When the graphical editor performs the C++ to Diagram translation, it will check the data type to see if it matches one of the container class names in the container class list. If it is on the list, the tool will recognize the data type as a container class template. The standard C++ library classes will be preloaded into the list when they are available.

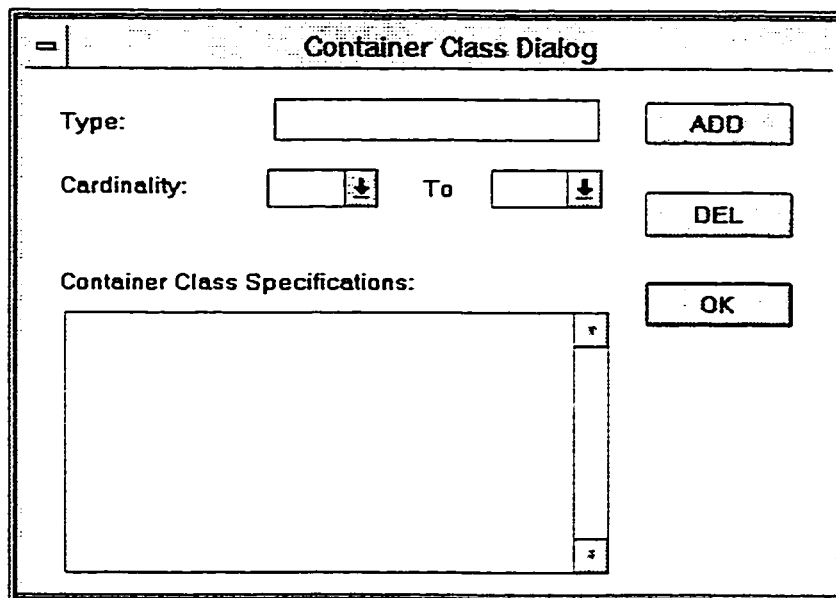


Figure 35. Container Classes Specification Dialog

CHAPTER 5

DESIGN OF THE CODE DOCUMENT

This chapter describes the basic design concepts of the code document in the Cloud9 project. The code document will serve as an OO database for transferring design data to and from the diagram editor and hierarchical structure editor, and for communicating with the code generator to produce C++ source code.

5.1 Requirements

A successful design of the code document should meet the following requirements:

1. It should be complete and informative in describing a C++ program.
2. It should be fast in mapping to graphical symbols in the diagram editor.
3. It should be easy to generate a C++ source program.
4. It should provide searching capability for locating classes in the hierarchical structure by a given qualifying class name. (i.e. classA::classB)
5. It should provide a persistence mechanism for saving and reloading from an external file.
6. It should provide capabilities for adding, deleting, and changing a class entry.

5.2 Implementation

One obvious approach to designing the data structures of the code document is simply to use the C++ grammar symbols as the tree node objects and build a parse tree from the bottom up. Unfortunately, this approach produces too many objects, and it gives very little insight to the class hierarchy. Although it provides a complete set of data objects, it does not give real insight into the class membership, behavior and relationship among classes. It also requires a parser to recognize the C++ grammar to build the parse tree.

Another approach is to implement a *structure* tree. This is a tree structure in which each node of the tree represents an important structure of the C++ source program. Each node of the structure tree is an object that contains member data for storing the C++ source of the corresponding structure and member functions that prescribe the semantic information of the object. Using the structure tree as the data structure of the code document provides a better description of the hierarchical structure of the program. It is also easy to translate from the hierarchical structure editor to the code document. Since the structure tree resembles a C++ program, it easily satisfies the first three requirements.

To satisfy requirement 4, a member function `search_class()` is implemented to provide searching capability. This function should be able to search recursively into the program tree structure and find the corresponding class information by a given valid qualifying class name.

To fulfill requirement 5, all classes in code document are inherited from a `Chi_Persistent` class which provides an automatic method to save and load the data directly from an external file.

The last requirement involves traversing the tree structure, adding, deleting, or modifying a class node.

5.3 Modeling the structure tree

To build the *structure* tree, one would need to model the structure of a C++ program.

The core of this model will consist of the following classes:

1. **Module:** The top root node of the *structure* tree is the module. A module models a single C++ source file.
2. **Class:** A class models the C++ class definition. It encapsulates member data and member function definitions.
3. **Variable:** It models the data members of each class.
4. **Function:** This class models the member functions of each class.

These classes and their relationships form the foundations of the structure tree.

5.3.1 Module

A *module* has a module name denoting the source file name and a path name denoting the source file location. Each module has a list of include files, comments, a private module body section, a public module body section, and a C++ code section that can

be used to integrate with other application-specific code. e.g. special macros required by some framework. This module class usually corresponds to file in a conventional system. A module maps well into the Booch OO concept of class category.

The following listing shows the C++ interface for the DOC_Module class:

```
class DOC_Module : public Chi_Persistent
{
public:
    DOC_Class* search_class(Chi_String) const; // search within the module for a class
    // etc.
private:
    DOC_Name    _name;           // module name
    DOC_Path    _path;          // module path
    DOC_Comment _comment;       // any comments
    DOC_ModuleSection _public; // public module section
    DOC_ModuleSection _private; // private module section
    DOC_Code    _code;          // other application-specific code, i.e. framework
    Chi_Array<DOC_IncludeFile> _include_file; // all include files
};
```

Each private/public module body section consists of a list of constant definitions, a type declaration section, a list of global variable declarations, and a list of global function declarations.

The following listing shows the C++ interface for the DOC_ModuleSection class:


```

class DOC_ModuleSection : public Chi_Persistent
{
public:
    DOC_Class* search_class(Chi_String) const; // search within the module section for a class
    // etc.

private:
    Chi_Array<DOC_EnumConst>    _constant;        // compile-time constants
    DOC_TypeSection            _type_section;    // type declarations
    Chi_Array<DOC_Variable>    _variable;        // global variables
    Chi_Array<DOC_Function>    _function;        // global functions
    friend class DOC_Module;

};

```

5.3.2 Type Declaration Section

A type declaration section consists of a list of enumeration definitions, a list of type declarations, a collection of unions, and a collection of classes.

The following listing shows the C++ interface for the DOC_TypeSection class:

```

class DOC_TypeSection : public Chi_Persistent
{
public:
    DOC_Class* search_class(Chi_String) const; // search within the type section for a class
    // etc.

private:
    Chi_Array<DOC_Enum>        _enum;            // enumeration definitions
    Chi_Array<DOC_Typedef>    _typedef;        // user-define types
    Chi_Array<DOC_Union>      _union;          // union definitions

```

```

        Chi_Array<DOC_Class*>    _class;        // class definitions
        friend class DOC_Module;

};

```

Program structures containing recursive structures often produce a circular reference. The problem arises when the type declaration section of the class declaration section contains another list of class declarations. This kind of structure produces a circular reference. In order to avoid this problem, we declare a list of pointers to the class declaration objects in the type declaration section.

5.3.3 Type Declaration

A type declaration consists of a type name, a name, and comments.

The type name is the C++ type expression such as `int`, `int*`, `int&`, `int const&` and `int(*)`(int). Not all types are currently implemented.

The following listing shows the C++ interface for the `DOC_Typedef` class:

```

class DOC_Typedef : public Chi_Persistent
{
public:
    // etc.

private:
    DOC_TypeName    _type;        // original data type name
    DOC_Name        _name;        // user-defined data type name
    DOC_Comment     _comment;     // comment on the user-defined

};

```

5.3.4 Class

A class *Class* represents the static semantics of C++ class declaration in a type declaration section. A class *Class* includes a name, a comment, a list of parent class names, three visibility class sections, a friend section and a code section. The list of parent class names specify the list of superclasses in the inheritance relationships. The three visibility class sections are public, protected, and private. Each of these specifies the visibility properties of attributes and operations within the class. A class has an uninterpreted code section which contains application-specific code as building blocks of an application framework.

The class *Class* also contains two member functions: `search_class()` and `get_parent()`. The `search_class()` function is used to search for the sub-class which is nested within this class scope. The `get_parent()` function is used to get a collection of classes that are superclasses to this subclass.

The following listing shows the C++ interface for the `DOC_Class` class:

```
class DOC_Class : public Chi_Persistent
{
public:
    DOC_Class* search_class(Chi_String) const; // search within the class for a sub-class
    Bool get_parent(int, DOC_Parent&) const; // find the corresponding DOC_Parent
    // etc.
private:
    DOC_Name        _name;        // class name;
```

```

    DOC_Comment      _comment;      // comment on the class;
    Chi_Array<DOC_Parent> _parent;    // parent classes;
    DOC_ClassSection  _public,       // public class section
                        _protected,  // protected class section
                        _private;    // private class section
    DOC_FriendSection _friend;       // friend section
    DOC_Code          _code;         // other application-specific code, i.e.
framework.
    friend class DOC_Module;
};

```

Each Class section consists of a type section, a list of constant declarations, a list of attributes (fields and variables) and a list of operations. An attribute is either a variable declaration or a field declaration. A variable declaration has an uninterpreted initial value whereas field declarations cannot be initialized. An operation can be an inline function or a non-static function.

5.3.5 Field/Variable

A *field* has a type name, a variable name, and a comment. It can not be initialized.

A *variable* is similar to a field object except that a variable can be initialized. The initial value is treated as uninterpreted C++ code.

The following listing shows the C++ interface for the DOC_Field class and the DOC_Variable class:

```

class DOC_Field : public Chi_Persistent
{
public:
    // etc.

private:
    DOC_TypeName _type;        // type of the field
    DOC_Name     _name;       // name of the field
    DOC_Comment  _comment;    // comment on the field
};

```

```

class DOC_Variable : public Chi_Persistent
{
public:
    // etc.

private:
    DOC_TypeName _type;        // type of the variable
    DOC_Name     _name;       // name of the variable
    DOC_Comment  _comment;    // comment on the variable
    DOC_Code     _value;      // value of the variable
};

```

5.3.6 Function

A Function has a function name, a return type, comments, a list of arguments, and a code section for the function body. A function body is the C++ source code enclosed by braces. It also has a modifier member to specify if the function is a const member function, virtual function, inline function, or pure virtual function.

The following listing shows the C++ interface for the DOC_Function class:

```
class DOC_Function : public Chi_Persistent
{
public:
    Chi_String rtn_full() const;           // return the full function declaration
    Bool argument(int, DOC_Variable&) const; // find the corresponding function argument
    enum {NONE = 0, CONST = 1, VIRTUAL = 2,
          INLINE = 4, PURE = 8}; // modifier bitmap
    // etc.
private:
    DOC_Name    _name;           // function name
    int         _modifier;       // function modifier
    DOC_TypeName _rtn_type;       // function return type
    DOC_Comment _rtn_comment;     // function return comment
    DOC_Comment _comment;         // function comment, including its purpose
    DOC_Comment _remark;         // additional function remark
    Chi_Array<DOC_Variable> _argument; // function arguments
    DOC_Code    _body;           // function body
};
```

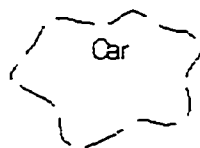
CHAPTER 6

TRANSLATION OF OO DIAGRAMS TO C++

Each class diagram has a set of classes that represents the corresponding problem domain. A class diagram is translated to a program module in C++ code. Each program module becomes a header (.h) file and a source (.cpp) file in a project. A class symbol in the class diagram becomes a class definition in the C++ program module. This chapter will explain how to translate a class diagram to the structure tree. Since the structure tree resemble the structure of the C++ code, we use the C++ code as the target in the translation to provide clarity in showing.

6.1 Translation of Class Symbol

Each graphical cloud shape class symbol represents a class in a class diagram. A class symbol is directly mapped to a C++ class in the program module. The class name of the class symbol is translated to the name of the C++ class as shown in [Figure 36]. Since the graphical editor enforces that every class name must be unique to the class diagram, it also ensures the C++ class is distinct in the namespace of the program module. The class name is the key index for searching classes in the program module.



Translate to C++ \longrightarrow
`class Car { /* ... */};`

Figure 36. Translation of class cloud symbol

6.1.1 Attribute

As described in Chapter 2.1, an attribute denotes a part of an aggregate object. An attribute may have a type associated with it. An attribute is translated into a private data member of the corresponding class shown in [Figure 37]. The attribute type is the type field of the data member declaration; the attribute name is directly translated to the name field of the data member declaration. The attribute name is checked for uniqueness within the class by the graphical editor. Therefore, it ensures no data naming conflicts in the data member declaration. Some CASE tools will automatically generate a pair of public `get()` and `set()` member functions for each private data member (e.g. Rational Rose). We thought this is redundant in the attribute translation and should let the user add the functions if it is truly necessary.

Example:

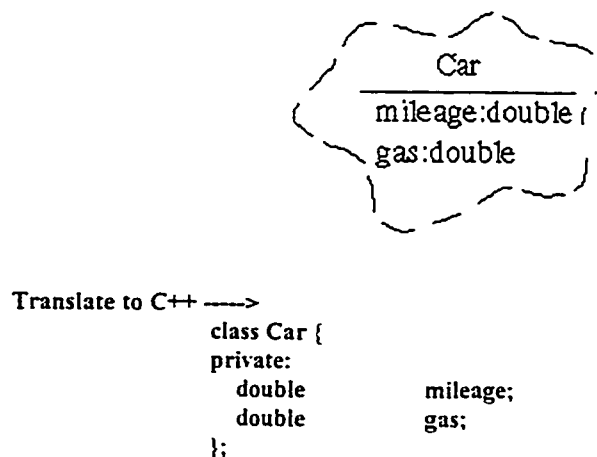


Figure 37. Translation of class attributes

6.1.2 Operation

Each operation must have an operation name. It may have an optional return type and argument list associated with it. An operation is translated into a public member function of the corresponding class shown in [Figure 38]. The operation name and return type are translated to the name and return type of the member function. If there are a list of arguments, it will be translated into the member function prototype.

Example:

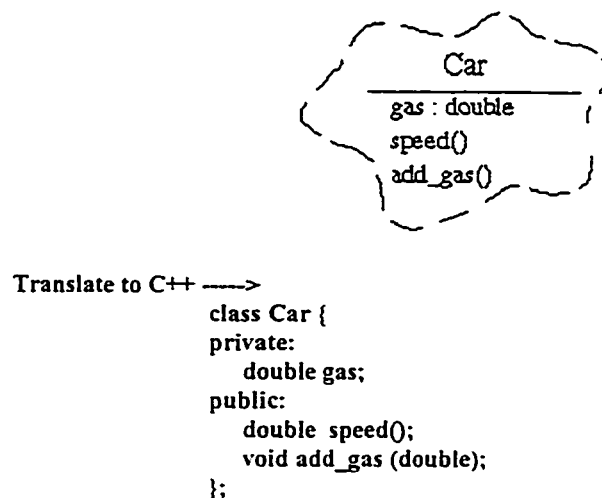


Figure 38. Translation of class operations

As we described in Chapter 2.1, the Booch notation for the attributes and operations is simplified to isolate special syntax of various languages. For example, in C++, we may wish to declare certain attributes as static members, declare certain operations as virtual or pure virtual functions, and designate certain attributes and operations in the public, protected and private declaration. These language specific details are not supported in the current version of Cloud9, these could be included in the future

enhancement to the software. By default, we translate all attributes to private member data and all operations to public member functions.

6.2 Translation of Class Relationship

A class relationship symbol represents a relationship between two classes in the class diagram. A class relationship can be one of the three kinds: *has*, *is-a*, or *uses*.

6.2.1 Has (aggregate) Relationship

A *has* relationship denotes a whole-part hierarchy in the OO class diagram. A *has* relationship is represented in C++ as a data member in the whole (aggregate) class. The part object is declared as a private data member in the whole class definition. The label of the *has* relationship is the name of the data member variable and the name of the part class is the type of the data member in the data member declaration. A number of factors affect the actual C++ code that is translated from the *has* relationship. Each *has* relationship must associate with a physical containment and cardinality adornments. These adornments give additional specifications to the way the data member is declared. The physical containment adornment denotes the physical containment of the aggregation. There are two kinds of containment: by value or by reference. The containment of the relationship determines whether the data member holds the actual data or pointers to the data. If the *has* relationship uses the by value containment, the data member is declared as an instance of the part class. If the *has* relationship uses the by reference containment, the data member can be

declared as a pointer or a reference to the part object depending on the user choice. The simplest case is when the cardinality is 1 to 1 and the containment is by value, which translates to a simple field whose type is the part class as shown in [Figure 39]. For example, a car has a motor.

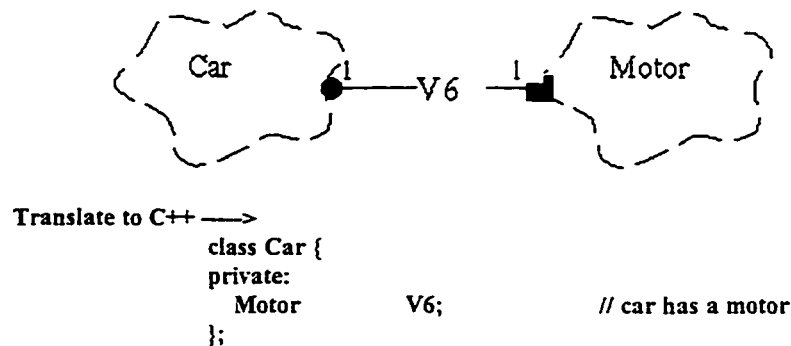


Figure 39. Translation of aggregation relationship (has) symbol with by value containment

The cardinality adornment denotes the cardinality of the aggregation. There are three kinds of cardinality across a has relationship: 1 to 1 relationship, 1 to 0/1 relationship, or 1 to many relationship. The cardinality of the relationship determines the data member type. The 1 to 1 relationship means each whole class contains exactly one part object. In C++, the 1-to-1 has relationship can be translated to a data member declaration in the whole class of either a part object or reference to the part object depending on the physical containment. If the 1-to-1 *has* relationship is by reference containment, the data member is declared as a reference to the part object because a reference variable must be bound to an object. For example, a *Dialog* class has a reference to its owner *Window* object *wdc* as shown in [Figure 40].

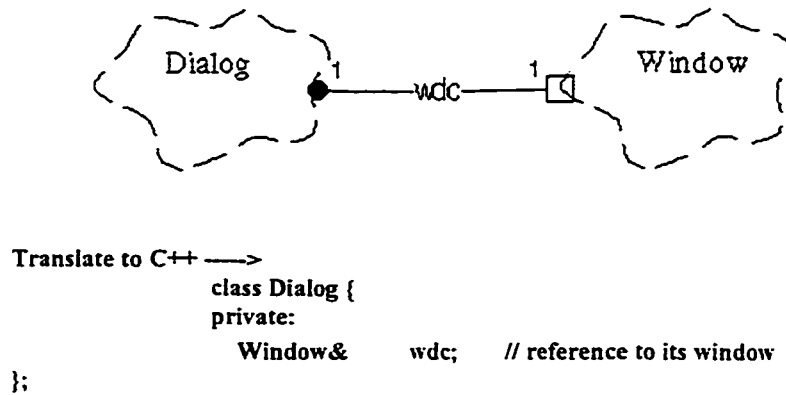


Figure 40. Translation of aggregation relationship with by reference containment

A 1-to-0/1 has relationship means the whole class contains either one or none of the part object. In C++, the 1-to-0/1 has relationship can be modeled by declaring the data member as a pointer to the part object, because only a pointer can have a null value and thus it can be bound to one or none object. For example, a *list* has a *link* to the head link of the list or the current link of the list as shown in [Figure 41]

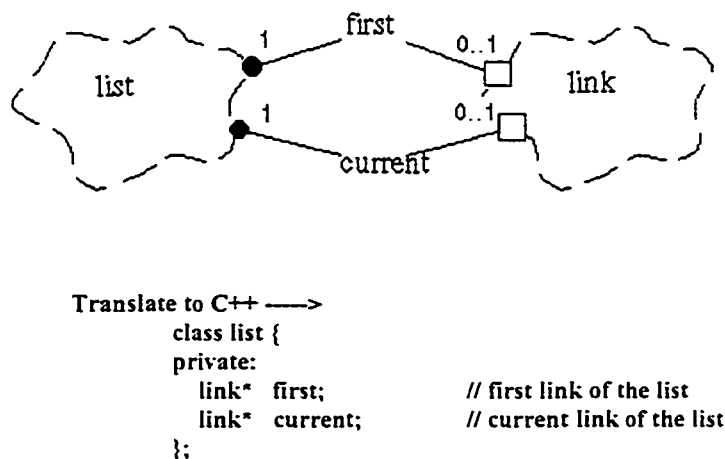


Figure 41. Translation of 1-to-0/1 aggregation relationship

The 1 to many relationship means each whole class contains more than one copy of the part object. A more complex type must be used to hold the data. It usually requires an array or a container class to represent the 1 to N cardinality. For example, a *MailBox* class has many *Messages* as shown in [Figure 42].

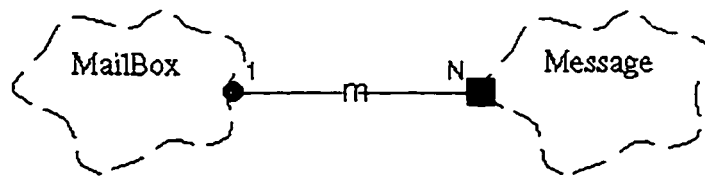


Figure 42. Translation of 1-to-many aggregation relationship

In C++, we can model this 1-to-many cardinality in many different ways. The data member variable can be declared as a C array of the part object if the size of the many relation is known. The user specifies the size of the array at the time of creating the *has* relationship symbol. For example, each *MailBox* has up to ten *Messages*.

```
class MailBox {  
    private:  
        Message    m[10];        // an array of 10 Message objects  
};
```

If the user wants to use a C++ dynamic array to model the relationship, the data member variable can be declared as a standard C++ array template with a type parameter of the part class. For example,

```

class MailBox {
private:
    array<Message> m;           // a C++ array of class Message
};

```

The C++ array template described above is one of the default choices in the relationship specification form. Other user defined class templates or types could also be used. One can set the other types to refer to ones own container classes when one wants to use a non-array container class, such as a list or a set. For example,

```

class MailBox {
private:
    list<Message> m;           // a C++ list of class Message
};

```

When the 1-to-many has relationship is created in the OO class diagram, users can specify the specific implementation method for the 1 to many cardinality adornment through the relationship specification form. The default implementation method for the 1-to-many relationship is an C array. Other complex types must be entered into the editor by the user.

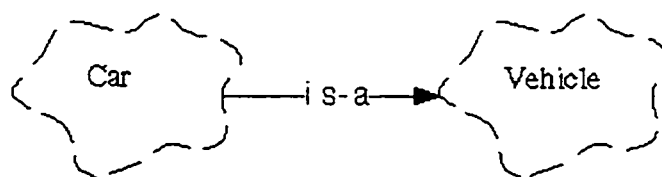
The kind of implementation method and corresponding translations for the has relationship is summarized in [Table 2] for the various combinations of the cardinality and physical containment.

Cardinality	By value containment	By reference containment
1 to 1	T	T& (reference to T)
1 to 0 or 1		T* (pointer to T)
1 to N	T[] (array of T) or array<T> (C++ array) or Chi_list<T> (user defined type)	T*[] (array of pointers to T) or array<T*> (C++ array) or Chi_list<T*> (user defined type)

Table II. The various Has relationship and their C++ translations.

6.2.2 Is-a (Inheritance) Relationship

An *is-a* relationship denotes an inheritance relationship between a superclass and a subclass. A superclass is the base class and the subclass is the derived class. As shown in [Figure 43], the base class name is added after the : in the subclass class declaration during translation.



Translate to C++ →

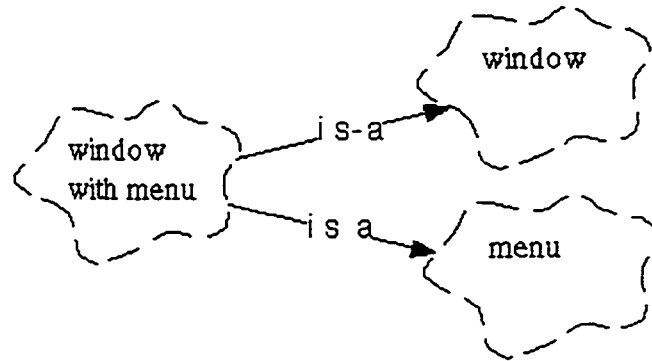
```

Class Car : public Vehicle {
    /* ... */
};
  
```

Figure 43. Translation of inheritance relationship

6.2.2.1 Multiple Inheritance

A class symbol can have more than one “is-a” relationship; this is commonly called multiple inheritance. The translation is similar to that of single inheritance in that the base class name is specified after `:` in the subclass class declaration. For example, as shown in [Figure 44], class *window with menu* is inherited from class *window* and class *menu*.



Translate to C++ \longrightarrow

```
class window_w_menu :
    public window,
    public menu
{ /* ... */};
```

Figure 44. Translation of multiple inheritance relationship

6.2.2.2 Common Base Class

Although it is impossible to derive directly from the same base class more than once, it is easy to inherit indirectly from a common base class along a different inheritance path. Consider an *item* class that is a common base class to both *window* and *menu*. Therefore, *window_w_menu* derives twice from *item* as shown in [Figure 45].

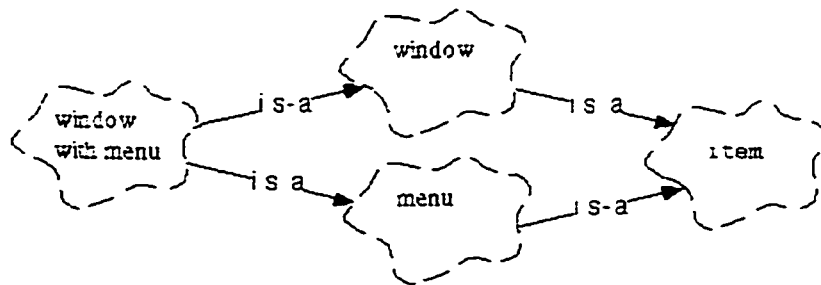


Figure 45. Common base class multiple inheritance relationship

C++ requires the declaration of the inheritances as virtual to merge common indirect base classes into one, otherwise the generated code may not compile. For example,

```
class window : virtual public item
{
  /* ... */
};
```

```
class menu : virtual public item
{
  /* ... */
};
```

and the *window_w_menu* class derives in the usual way from both *window* and *menu*:

```
class window_w_menu :
  public window,
  public menu
{
  /* ... */
};
```

The translator must check for an indirect base class and automatically add the keyword *virtual* to the previous ancestor classes' declarations along the different inheritance paths from the same indirect base class.

6.2.2.3 Access Control

An is-a relationship can be a private, protected, or public inheritance relationship. The access control of the “is-a” relationship is specified in the relationship specification form at the time when the connector is created in the diagram. Like an attribute in C++, a base class can be declared private, protected, or public in the derived class declaration. For example

```
class line : public shape {  
    /* ... */  
}
```

Here we define a public “is-a” relationship between the classes line and shape. In translation, the keyword public, protected or private is added after the : and before the base class name in the class definition. The default access control for the *is-a* relationship is public.

6.2.3 Using Relationship

A using relationship denotes that a class is being used in another class, but the relationship itself doesn’t specify how it is being used. The used class can be a signature in the member function interface or a local variable declaration of the member function. There is no direct translation to the C++ code from the using relationship symbols other than generating code to include the header file containing the declarations of the used class. Although a using relationship has no important

effect on the generated code, it must be a part of the OO class diagram because it still provides a valuable visual insight of the how the classes are related to each other.

6.3 Forward Reference Problem

In the class diagram, if there is a cycle consisting of two or more *has* relationships by reference, a forward declaration is needed for one of the classes. A class object can be declared as a data member only if its class definition has already been seen. A forward declaration permits pointers and references to objects of the class to be declared as data members. For example, if there are two classes in the class diagram and each class is an aggregated class in the *has* relationship with the other, as shown in [Figure 46], then a forward declaration is needed for one of the classes to break the cycle.

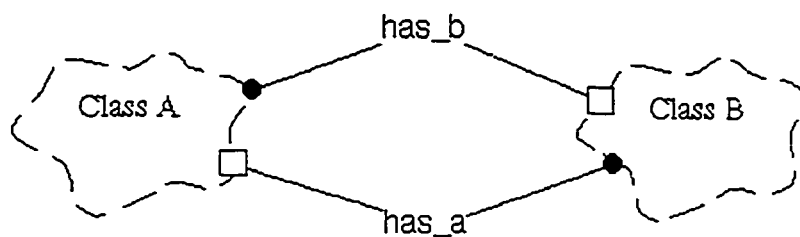


Figure 46. Class symbols with circular reference

Here is a definition of class A and B using a forward declaration:

```
class B;
class A
{
    B* has_b;           // pointer to B object
};
class B
{
    A* has_a;           // pointer to A object
};
```

Unfortunately, the current version of Cloud9 does not support the translation of circular *has* relationship with the use of forward declaration. Users have to edit the translated source code to add in the forward declaration of the class if there is a circular reference.

CHAPTER 7

SYNCHRONIZATION OF CODE AND OO DIAGRAMS

7.1 Classes

The basic idea is that we map each C++ class to a single class symbol (cloud) in the class diagram, so that the C++ class is a one to one mapping to the graphical class model. In [Figure 47] we map the C++ classes directly to the class symbols, resulting in the creation of two graphical class symbols *Customer* and *Account*.

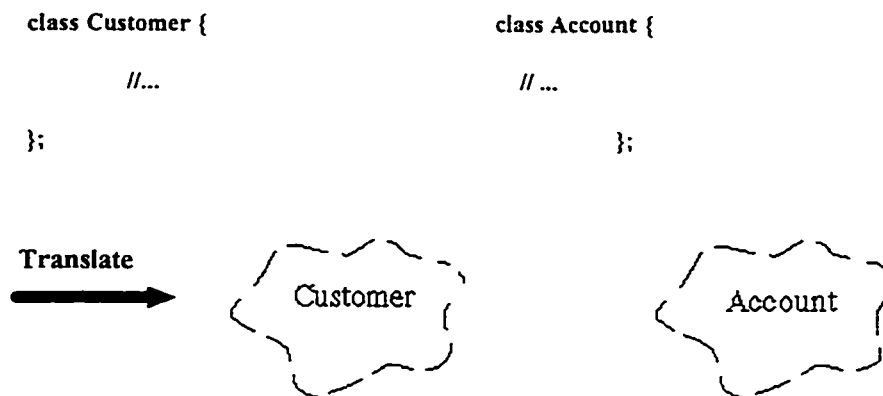


Figure 47. Translation of C++ classes to diagram symbols

Unlike many other CASE tools, which show the attributes and operations of a class in the class diagram, the declarations of the member variables and operations are not shown in the class diagram in Cloud9. Instead we provide a hide/show class member dialog to allow user to select member variables and operations that he/she wishes to be shown in the graphical cloud shape. There are two reasons for hiding the member attributes and operations in the default mode.

1. In the diagram editor, we are only interested in the high level abstract modeling of the objects and classes. The high level abstraction of the design involves class creation and defining class relationships, but the implementation details of the classes are not important at this time. They can be added later through the hierarchical structure editor. Therefore, only members that are instances of other classes are shown as aggregation relationships. All other member variables that are not class instances are omitted from the class diagram by default.
2. The class symbol (cloud shape) is often too small to fit all the member variables and operation declarations. We do have a feature in Cloud9 to show the class symbol as a rectangle shape. The rectangular box can display more class member information than a cloud. (It also resembles a CRC card.) [Wilkinson]

7.2 Inheritance

When defining classes, C++ programmers can derive new classes from existing classes through inheritance. For example,

```
class Car : public Vehicle { /* ... */};
```

The class *Car* inherits the structure and behavior of class *Vehicle*. *Vehicle* is called a base class of *Car*, also called a parent class of *Car*. *Car* is a derived class from *Vehicle*. Base classes can be modified by public, private and virtual keywords. We model this inheritance relationship as shown in [Figure 48]. The inheritance icon appears as an arrow and the arrowhead points to the base class, and the opposite end

of the arrow designates the derived class. Also according to the C++ grammar rules, a class may have one (single inheritance) or more (multiple inheritance) base classes.

For example,

```
class A : public B, public C { /* ... */};
```

In multiple inheritance, we have an inheritance relationship icon connecting from the subclass to each of the base classes.

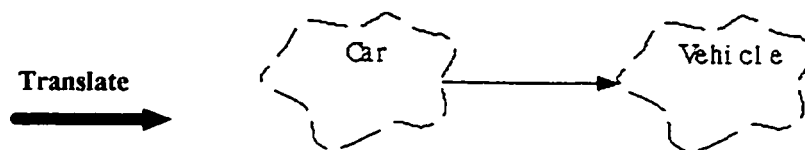


Figure 48. Translation of inheritance classes

7.3 1-to-1 Relationship

The attribute *v6* in the class *Car* below is shown as a cloud icon on the diagram. We would like to model this attribute as an aggregation relationship between class *Car* and *Motor*.

```
class Car {  
public:  
    Motor v6;  
};
```

The process starts by parsing the type name of the member variable declaration. If the type is not one of the simple types, such as int, float, etc., and the type name is one of

the class names in the diagram, we define an aggregation relationship icon for this C++ member variable. The icon ends with a filled circle that connects the aggregate class (*Car*). The opposite end connects with the “part” class (*Motor*). For example, we model this aggregation relationship as shown in [Figure 49]. First, we search for the class *Motor* in the class diagram and make an aggregation relationship icon from class *Car* to class *Motor* with the relationship labeled *v6*. The cardinality adornment for this relationship is 1-to-1.

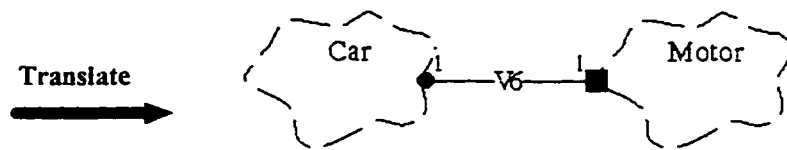


Figure 49. Translation of a class member variable

7.4 1-to-N Relationship

A member of a class can be a collection of other objects. For example,

```
class MailBox {
    Message m[10];           // an array of 10 Message objects
};
```

Each mailbox is allowed to store up to a maximum of ten messages. The message queue is modeled as a C array of 10 messages in class *MailBox*. We translate the C array to a 1-to-many aggregation relationship in the class diagram as shown in [Figure 50]. The 1-to-N cardinality adornment is used with the aggregation relationship

symbol. The label “N” is attached to the end of the relationship icon which connects to the “part” class (*Message*).

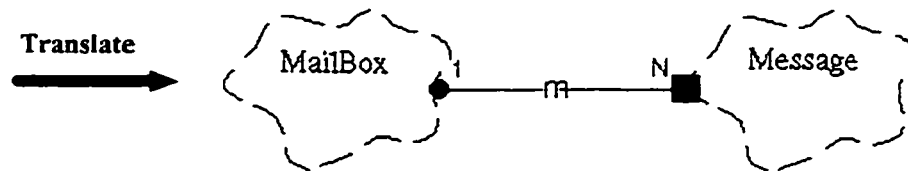


Figure 50. Translation of a C array member variable

It is also possible to model this member variable by using an ANSI C++ array template. For example,

```
class MailBox {
    array<Message> m;           // a C++ array of class Message
};
```

In order for Cloud9 to understand the semantics of the collection classes, Cloud9 has a built-in filter to recognize some of the standard collection class names such as vector, set, and list which are included in the standard ANSI C++ library. It treats the member variable of a collection class as a 1 to many aggregation relationship. At this point, the tool can only recognize some common ANSI standard C++ template classes. If the programmer uses other third party collection classes, he/she has to train the tool to recognize these special class template names. The tool provides a container class dialog that allows users to enter their favorite or proprietary collection of class names.

Once the collection class name and its relationship properties such as cardinality are entered into the dialog, (see [Figure 30]) , the tool will translate the corresponding variable declaration according to the information which the user entered through the container class dialog. For example, we can use a linked list template class to model the message queue and have the same graphical translation as an C array variable as shown in [Figure 50].

```
class MailBox {  
    list<Message> m;           // use other collection class. i.e. list  
};
```

7.5 Pointer Member Variables

In C++, we often declare member variables as pointers to other objects. There are some situations in which we would like to dynamically allocate memory for an object from the heap. The object itself is stored in the dynamic memory and the object pointer holds the accessible address. For example,

```
class Teller {  
    Customer* cust; // contains a pointer to instance of class Customer  
};
```

Here the class *Teller* physically contains a reference to an instance of class *Customer*. The “Customer* cust” syntax indicates that the variable *cust* is a pointer of type Customer* used for accessing *Customer* objects.

We use physical containment by reference and 1-to-0/1 cardinality adornments with an aggregation relationship symbol to model the C++ pointer member variable declaration, as shown in [Figure 51]. The open square (by reference) icon and the label “0/1” are attached to the end of the aggregation relationship icon which is connected to the part class. It is possible for a pointer variable to be bound to no objects, the 1-to-0/1 cardinality adornment is appropriate in this situation.

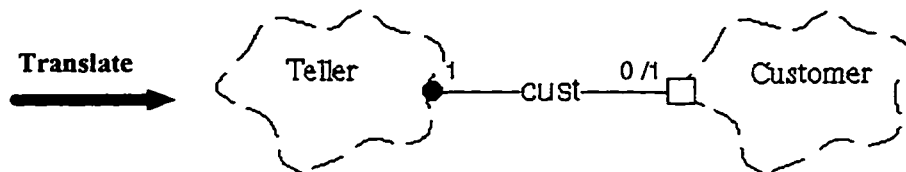


Figure 51. Translation of a pointer member variable

Unfortunately, in C++, a pointer variable can have three other different meanings besides dynamic storage. The above transformation could pose incorrect or inadequate semantic description of the C++ class.

1. Object sharing.

Pointers are a C++ mechanism for sharing ownership of an object. In C++, variables are used to hold object values and pointers are used for object access. Object sharing is achieved by binding two or more pointer variables to the same object.

```
Class Window {
    Database* db;
```

```

    }

    Window W1, W2;
    w1->db = dbase;
    w2->db = dbase;

```

Attribute db in both window W1 and W2 share the same database dbase.

2. Polymorphism.

When pointers are used for polymorphism, objects of a class or its derived class are bound to the same common base class pointers. It is possible to assign an object of any derived class to a pointer variable of its base class.

```

class Graph {
    Chart* _chart;
}

```

where Chart may be a bar chart or pie chart.

3. Dynamic C array.

A C pointer variable can also use as a dynamic C array.

```

Class Company {
    Employee* e_array ;
    Company() {};
}

```

The original transformed 1-to-0/1 aggregation relationship graphical symbol is inadequate to describe the other three meanings. Most importantly, it fails completely to model for the C dynamic array. A C dynamic array means a 1-to N aggregation relationship. However, the tool itself has no knowledge of what the programmer intends to do with a pointer. It is impossible for the tool to describe the exact meaning of a pointer variable. The programmer has to alter the translated graphical symbol if he/she intends to use a pointer as a dynamic C array.

7.6 Reference Member Variables

In C++, we often declare member variables as reference to other objects. For example,

```
class Dialog {
    Window &wdc;           // contains a reference to an instance of class window
};
```

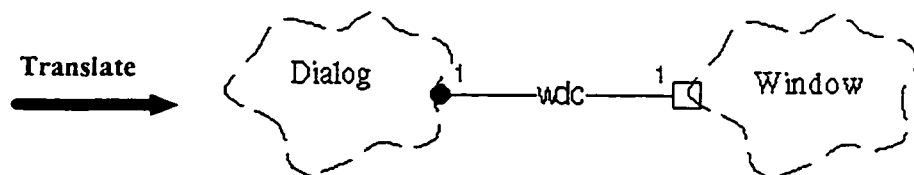


Figure 52. Translation of a reference member variable

The “Window &wdc;” syntax indicates that the member variable *wdc* is a reference to an instance of the class *Window*. A C++ reference variable is very similar to a pointer

variable, except that it has to be bound to a single object. A pointer variable can be bound to no object. Therefore, we use a physical containment by reference and 1-to-1 cardinality adornments with an aggregation relationship symbol to model these kinds of C++ member variable declarations, as seen in [Figure 52]. The open square (by reference) icon is attached to the end of the aggregation relationship icon that is connected to the reference “part” class.

7.7 Common Class Attributes

If a class *Employee* contains member *birthday* and *hire_day* of class *Date*, how do we choose between using aggregation relationships of class *Employee* and *Date*, and attributes of type class *Date* to describe these attributes? For example,

```
class Employee {  
    Date    birthday;  
    Date    hire_day;  
};
```

Do we really want a diagram like this ?

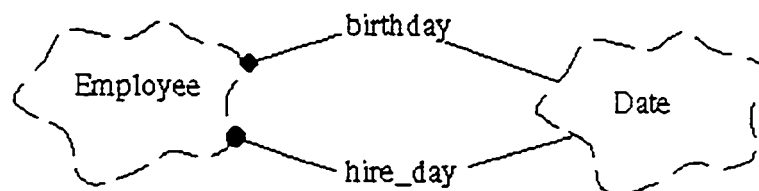


Figure 53. Translation of trivial type as a class type

Probably not, because the type *Date* is a trivial type that is used in many other classes. We would probably want to treat it as a simple atomic type such as `int` and `float`. Most programmers would probably hide the class definition of *Date* from the class diagram, and any relationship associated with it. We offer two solutions to this problem. As the first solution, we have designed a hide/show class feature in Cloud9 that allows users to specify the desired classes to be shown on the class diagram. With this feature, the tool will continue to translate the member variables `birthday` and `hire_day` as aggregation relationships between class *Employee* and class *Date*. After the translation, the user can then select the class *Date* to be concealed from the diagram. The diagram editor will automatically remove the class *Date* and any relationships associated with it from the class diagram. As the second solution, we have designed a type checking feature that allows users to specify certain classes or class templates as trivial simple types. Simple data types are not translated to classes in the class diagram. During the translation, the tool will examine each member variable declaration. If the variable type is among the types included in the type checking list, the member variables will become attributes in the enclosing class.

7.8 Using Relationship

In C++, objects are often passed as arguments through the member function interface, so that the member function can obtain the service from this object. In Booch notation, we called this form of object usage the *using* relationship. In our example of the *Diagram* class, the class *Cloud* appears as part of the signature in the member

function interface *Diagram::AddCloud(Cloud& cloud)*, and thus we can say that *Diagram* uses the services of the *Cloud*. In [Figure 54], shows the using relationship between the *Diagram* and *Cloud* is represented via a graphical link between these classes in the class diagram. An open circle is attached to the end connected to the usage class.

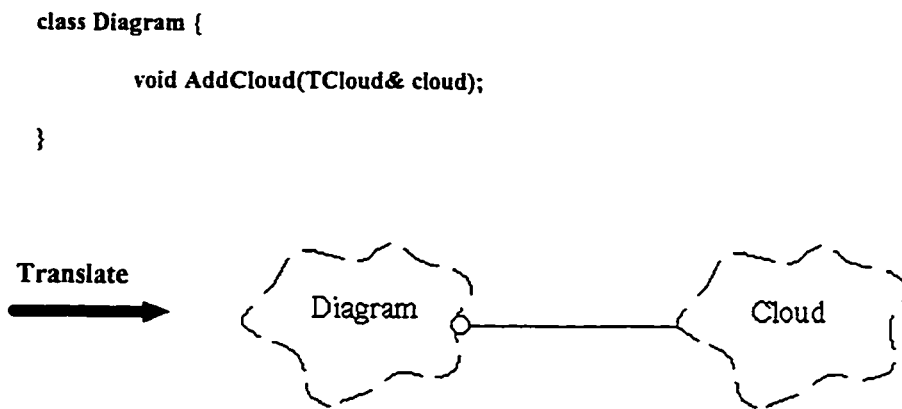


Figure 54. Translation of a using relationship

Furthermore, a class declaring a local variable of another class in the implementation of its member function, is also defined as a using relationship between these corresponding classes. For example, the *Printout::Print()* function has a local variable arial of type class *Font* in its implementation. This *Font* object (arial) is only used by the member function and is not part of the *Printout* instance. Thus we say *Printout* is using the service of *Font*. Both forms of usage are represented by the same symbol in Booch notation.

```

Printout::Print()
{
    Font arial("Arial");
}

```



```
    /* ... */  
}
```

In all the examples we have given up to this point, it is very straight-forward to recognize these kinds of using relationships: if we see an argument declaration of a class in the member function interface or a local variable of a class in the function body, then we immediately recognize these C++ constructs as using relationships between the corresponding classes.

However, sometimes it is very difficult to recognize these relationships if it is an indirect usage. For example, if a variable of class A is declared in the global function g() and member function X::f() calls g() to perform certain operations, then it is clear from the logical view that class X is using the service provided by class A, since the global function g() uses class A. But the tool will not recognize this kind of indirect using relationship.

```
X::f()  
{  
    g();  
}  
g()  
{  
    A a;  
}
```

7.9 Static Member Variables

In C++, a static member variable has a special property that distinguishes itself from other member data. Each instance of a class owns its copy of the member data, but a static data item is owned by the class itself and it is shared by all instances of the class.

In Booch notation, a special adornment represents static data members. Booch treats the static member data as a special aggregation relationship between the class and the static member class.

CHAPTER 8

CONCLUSIONS AND FURTHER RESEARCH

In this research, I have reviewed existing CASE tools that support various Object-Oriented methodologies and generate Object-Oriented diagrams. I have shown that there are ambiguities in the code generation of aggregation relationships that prevent the generation of perfect C++ code. Furthermore, my findings indicate that the synchronization problems between OO diagram and generated source code make it difficult to implement iterative development techniques that are both dependable and easy to use.

The various methods proposed in this research address these problems:

1. C++ code generation
2. OO diagram and C++ code synchronization
3. Usage of Container Class Templates
4. Checking of common user defined data types

These implementation methods are sufficient to correct some of the problems found in existing CASE tools.

This research leaves open a number of issues for future research efforts. The most pressing one is the support of multiple C++ modules. Further research should aim to substantially improve the capability of supporting more than one module using

namespace and file directories. The support for importing third party C++ class library for building window applications is also essential. The ability to read and load the library and present the C++ classes graphically within the tool will greatly enhance the usability of the tool. Extending the support for the Booch notations such as class category and parameterized class is also crucial.

8.1 Class Categories

Further research is also needed into the support for class categories in Booch notation. The discussion in Chapter 2 shows that the class categories can be implemented by using the namespace mechanism in C++. Until recently there has been little opportunity to experiment with the usage of namespace mechanism in C++, because of the lack of compiler support. It may reveal additional problems in the actual implementation.

8.2 Grand Methodology of Booch and Rumbaugh

Almost everyone are in complete agreement on the basics of object orientation with regard to classes, objects, polymorphism, inheritance, and encapsulation. Booch and Rumbaugh have joined forces recently to create a grand methodology that will have a common process, a standard notation and a meta-model. They intend to iron out the differences in their methods and create a standard notation and a meta-model so that irrelevant differences can be put aside. Further research is needed to support this notation.

8.3 Use of Wizards

In the software industry, it is very common to use a “Wizard” to guide the user to develop standard things. For example, Microsoft Word now uses “Wizards” to help users to develop typical documents such as resume, letter, etc. Conceivably, a diagram wizard could employ similar techniques to help the user to determine common design decisions, such as if a class needs to define a constructor or destructor, if a common base class should be virtual, or if an abstract class should be defined, etc. It could eliminate common design flaws, such as failing to define a copy constructor for a class that allocates on the heap, failing to make a common base class virtual before the generated code gets to the C++ compiler.

8.4 Generated Code for Other OO Languages

The diagram editor is developed as a language-independent front end and can generate code for other OO languages (such as Eiffel, Smalltalk, etc.) from the cloud diagram.

BIBLIOGRAPHY

- [Booch91] Grady Booch, Object-Oriented Design With Applications, Benjamin-Cummings, 1991.
- [Booch94] Grady Booch, Object-Oriented Analysis and Design With Applications, 2nd ed., Benjamin-Cummings, 1994.
- [Coad1] Peter Coad and Edward Yourdon, Object-Oriented Analysis, 2nd ed., Prentice Hall, 1991.
- [Coad2] Peter Coad and Edward Yourdon, Object-Oriented Design, Prentice Hall, 1991.
- [Ellis and Stroustrup], Margaret Ellis and Bjarne Stroustrup, The Annotated C++ Reference Manual, Addison-Wesley, 1990.
- [Horst1] Cay Horstmann, "A look at C++ designer, An OMT CASE Tool and Code Generator," C++ Report, Vol. 5, No. 7, pp. 55-60, 1993.
- [Horst2] Cay Horstmann, "OO design tools with C++ code generation," C++ Report, vol. 5, pp 51-58, 1994.
- [Horst3] Cay Horstmann, Mastering Object-Oriented Design in C++, John Wiley & Sons, Inc., 1995.
- [Horst4] Cay Horstmann, "The CLOUD9 Project for Iterative Object-Oriented Design Documentation and Code Generation", San Jose State University, April 1994.
- [Martin] James Martin and James O. Odell, Object-Oriented Analysis and Design, Prentice Hall, 1992.
- [Rumbaugh] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, Object Modeling and Design, Prentice Hall, 1991.
- [Stroustrup91] Bjarne Stroustrup, The C++ Programming Language, 2nd Ed., Addison-Wesley, 1991.
- [Stroustrup94] Bjarne Stroustrup, The Design and Evolution of C++, Addison-Wesley, 1994.
- [Taylor and Hecht] D. K. Taylor and A. Hecht, "Using CASE for Object-Oriented Design with C++", Computer Language, Vol. 7, No. 11, pp. 49-57, Nov 1990.
- [Wybolt] Nicholas Wybolt. "Experiences With C++ and Object-Oriented Software Development", ACM Software engineering notes, Vol .15, No. 2, pp. 31-39. 1990.
- [Wilkinson] Nancy M. Wilkinson, Using CRC Cards An Informal Approach to Object-Oriented Development. SIGS Books. 1995.