National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

A Program-understanding Schema for C++

by

Pankaj Bedi

An M.Sc. (Systems Science) Thesis

submitted to

the School of Graduate Studies and Research
in partial fulfillment of the requirements for the
Masters of Systems Science degree

University of Ottawa
Ottawa, Ontario
Canada

March 1994

© Pankaj Bedi, Ottawa, Canada, 1994

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services Branch

Direction des acquisitions et des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file   Votre référence

Our file   Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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

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

ISBN 0-315-95887-1

Canada

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

Acknowledgments

I am grateful to my supervisor, Dr. Tuncer Ören, for his guidance and advice in the preparation of this thesis. He was kind and patient, and spent a considerable amount of his precious time in directing the research.

I wish to express my gratitude to my wife, Veena, and my son Gauruv, for their patience and understanding throughout my studies.

Abstract

Program-understanding schemata for understanding the static properties of C++ programs are presented. For understanding purposes, we have selected seven key concepts of the C++ programming language. A C++ program is viewed as being composed of these seven key concepts and the relationships which may exist between them. These seven key C++ concepts and their relationships are represented using concept templates. The information grouped within these concepts is shown using twenty-six display templates. A representation schema and a navigation schema for the concept templates and the display templates are explained in detail.

Table of Contents

Acknowledgments ii

Abstract iii

List of Figures x

Chapter 1 Introduction 1

1.1 Motivation and Aim of the Thesis 1

1.2 Structure of the Thesis 4

Chapter 2 Program-understanding 6

2.1 Program-understanding Application Domains 7

2.1.1 Validation and Verification 7

2.1.2 Software Maintenance 7

2.1.3 Reuse 8

2.1.4 Reverse Documentation 8

2.1.5 Redocumentation 8

2.1.6 Analysis 8

2.1.7 Design Recovery 9

2.2 What is to be Understood and Why 9

2.3 Program-understanding Objectives 11

Chapter 3 Design Methodologies and Software Engineering Tools for C++ . . 13
3.1 Object-Oriented Design Methodologies 13
3.1.1 The Booch Method 13
3.1.1.1 Concepts 13
3.1.1.2 Notations 15
3.1.2 The Rumbaugh Method 16
3.1.2.1 Concepts 16
3.1.2.2 Notations 16
3.1.3 The Wirfs-Brock Method 17
3.1.3.1 Concepts 17
3.1.3.2 Introduced Terminology 18
3.1.3.3 Notations 18
3.2 Survey of Software Engineering Tools for C++ 19
Chapter 4 Program-understanding Applied to C++ 22
4.1 C++ Key Concepts 22
4.2 Relationships Existing Between C++ Key Concepts 24
4.3 Typical Questions Raised During C++ Software
Maintenance 25
4.4 C++ Concepts for Information Discrimination 28
4.4.1 Executable File Concept (C1) 29
4.4.1.1 USES Relationship 29
4.4.1.2 HAS Relationship 29

4.4.1.3 ISA Relationship 29

4.4.2 Object File Concept (C2) 29

4.4.2.1 USES Relationship 30

4.4.2.2 HAS Relationship 30

4.4.2.3 ISA Relationship 30

4.4.3 Source File Concept (C3) 30

4.4.3.1 USES Relationship 30

4.4.3.2 HAS Relationship 31

4.4.3.3 ISA Relationship 31

4.4.4 Method Concept (C4) 31

4.4.4.1 USES Relationship 32

4.4.4.2 HAS Relationship 34

4.4.4.3 ISA Relationship 35

4.4.5 Logical Block Concept (C5) 35

4.4.5.1 USES Relationship 35

4.4.5.2 HAS Relationship 38

4.4.5.3 ISA Relationship 40

4.4.6 Abstract Data Type Template Concept (C6) 40

4.4.6.1 USES Relationship 41

4.4.6.2 HAS Relationship 41

4.4.6.3 ISA Relationship 42

4.4.7 Object Concept (C7) 43

4.4.7.1 USES Relationship 43

4.4.7.2 HAS Relationship 46

4.4.7.3 ISA Relationship 47

Chapter 5 Visual Representation Schema and Navigation Schema 48

5.1 Visual Representation Schema 48

5.1.1 Concept Template 48

5.1.2 Display Template 50

5.2 Navigation Schema 51

5.3 An Example: Representing a C++ Program Based on the Proposed Schema 55

5.3.1 Example Description 56

5.3.1.1 Makefile for Making a "glay" Executable File 56

5.3.1.2 Source File "glay.cc" 57

5.3.1.3 Source File "graph.c" 57

5.3.1.4 Source File "graph.h" 57

5.3.2 Executable File Concept Template C1 57

5.3.3 Object File Concept Template C2 59

5.3.4 Source File Concept Template C3 60

5.3.5 Method Concept Template C4 65

5.3.6 Logical Block Concept Template C5 72

5.3.7 Abstract Data Type Template Concept Template C6 . . . 75
5.3.8 Object Concept Template C7 . . . 80
5.4 Display Templates Description . . . 84
5.4.1 Object Files Display Template (D1) . . . 84
5.4.2 Source Files Display Template (D2) . . . 85
5.4.3 Library Files Display Template (D3) . . . 86
5.4.4 External Objects Display Template (D4) . . . 87
5.4.5 External Variable Display Template (D5) . . . 88
5.4.6 Abstract Data Type (a) Display Template (D6) . . . 89
5.4.7 Global Objects Display Template (D7) . . . 90
5.4.8 Global Variables Display Template (D8) . . . 91
5.4.9 Methods (a) Display Template (D9) . . . 92
5.4.10 Abstract Data Type (b) Display Template (D10) . . . 94
5.4.11 #define Directive Display Template (D11) . . . 95
5.4.12 D12 Methods (b) Display Template (D12) . . . 96
5.4.13 Objects (LHS/RHS) Display Template (D13) . . . 98
5.4.14 Variables (LHS/RHS) Display Template (D14) . . . 99
5.4.15 Logical Block Display Template (D15) . . . 100
5.4.16 Member Object Display Template (D16) . . . 102
5.4.17 Member Method (a) Display Template (D17) . . . 104

5.4.18 Abstract Data Type Inheritance Display Template (D18) 106
5.4.19 Parametric Abstract Data Type Display Template (D19) 107
5.4.20 Parametric Method Display Template (D20) . . . 108
5.4.21 Member Variables Display Template (D21) 108
5.4.22 Member Method (b) Display Template (D22) . . . 110
5.4.23 Another Object's Member Methods (RHS) Display Template (D23) 112
5.4.24 Methods (RHS) Display Template (D24) 113
5.4.25 Objects (RHS) Display Template (D25) 113
5.4.26 Variables (RHS) Display Template (D26) 114

Chapter 6 Conclusion 116
6.1 Implementation 116
6.2 Future Work 118

References 121

Appendix A Program Listing 125
A.1 "makefile" 125
A.2 "glay.cc" 126
A.3 "graph.c" 130
A.4 "graph.h" 136

List of Figures

Figure 4.1 C++ Concepts and Relationship Diagram 28

Figure 5.1 Concept Template 49

Figure 5.2 Display Template 51

Figure 5.3 Navigation Schema 52

Figure 5.4 List of Display Templates 53

Figure 5.5 Display Templates Generated from the Concept Templates . . . 54

Figure 5.6 Concept Templates Generated from the Display Templates . . . 55

Figure 5.7 Concept Template C1 58

Figure 5.8 D1 Display Template (from C1 (1.1.1)) 58

Figure 5.9 Concept Template C2 59

Figure 5.10 D2 Display Template (from C2 (2.1.1)) 60

Figure 5.11 Concept Template C3 62

Figure 5.12 D3 Display Template (from C3 (3.1.1)) 63

Figure 5.13 D2 Display Template (from C3 (3.1.2)) 63

Figure 5.14 D8 Display Template (from C3 (3.2.2)) 64

Figure 5.15 D9 Display Template (from C3 (3.2.3)) 64

Figure 5.16 D11 Display Template (from C3 (3.2.4)) 65

Figure 5.17 Method Concept Template C4 67

Figure 5.18 D12 Display Template (from C4 (4.1.1)) 68

Figure 5.19 D13 Display Template (from C4 (4.1.2)) 68

Figure 5.20 D13 Display Template (from C4 (4.1.3)) 69

Figure 5.21 D14 Display Template (from C4 (4.1.4)) 69

Figure 5.22	D13 Display Template (from C4 (4.2.1))	70
Figure 5.23	D14 Display Template (from C4 (4.2.2))	71
Figure 5.24	D15 Display Template (from C4 (4.2.3))	72
Figure 5.25	Concept Template C5	74
Figure 5.26	D13 Display Template (from C5(5.1.2))	75
Figure 5.27	D13 Display Template (from C5(5.1.3))	75
Figure 5.28	Abstract Data Type Template Concept Template C6	77
Figure 5.29	D6 (from C6 (6.1.2))	77
Figure 5.30	D16 (from C6 (6.2.2))	78
Figure 5.31	D22 (from C6 (6.2.3))	78
Figure 5.32	D17 (from C6 (6.2.4))	79
Figure 5.33	D18 (from C6 (6.3.1))	79
Figure 5.34	Object Concept Template C7	81
Figure 5.35	D22 Display Template (from C7 (7.1.1))	82
Figure 5.36	D23 Display Template (from C7 (7.1.2))	82
Figure 5.37	D25 (from C7 (7.1.4))	83
Figure 5.38	D16 Display Template (from C7 (7.2.1)	83
Figure 5.39	D21 Display Template (from C7 (7.2.2))	83
Figure 5.40	D6 Display Template (from C7 (7.3.1))	84
Figure 5.41	Display Template for Object Files (D1)	84
Figure 5.42	Information Display Template for User Files (D2)	85
Figure 5.43	Display Template for Library Files (D3)	86
Figure 5.44	Display Template for External Objects (D4)	87

Figure 5.45 Display Template for External Variables (D5) 88
Figure 5.46 Display Template for Abstract Data Type (a) (D6) 89
Figure 5.47 Display Template for Global Objects (D7) 90
Figure 5.48 Display Template for Global Variables (D8) 91
Figure 5.49 Display Template for Methods (a) (D9) 93
Figure 5.50 Display Template for Abstract Data Type (b) (D10) 94
Figure 5.51 Display Template for #define Directive (D11) 95
Figure 5.52 Display Template for Methods (b) (D12) 96
Figure 5.53 Display Template for Objects (LHS/RHS) (D13) 98
Figure 5.54 Display Template for Variables (LHS/RHS) (D14) 99
Figure 5.55 Display Template for Logical Block (D15) 101
Figure 5.56 Display Template for Member Objects (D16) 103
Figure 5.57 Display Template for Member Method (a) (D17) 104
Figure 5.58 Display Template for Abstract Data Type Inheritance (D18) . . 106
Figure 5.59 Display Template for Parametric Abstract Data Type (D19) . . 107
Figure 5.60 Display Template for Parametric Function (D20) 108
Figure 5.61 Display Template for Member Variables (D21) 109
Figure 5.62 Display Template for Member Method (D22) 110
Figure 5.63 Display Template for Another Objects's Member Method (D23) 112
Figure 5.64 Display Template for Methods (RHS) (D24) 113
Figure 5.65 Display Template for Objects (RHS) (D25) 114
Figure 5.66 Display Template for Variables (RHS) (D26) 114

Chapter 1 Introduction

1.1 Motivation and Aim of the Thesis

The availability of better tools and methodologies has enabled software engineers to deal with complex systems with more efficiency and productivity. The recent trend of using object-oriented solutions in complex problem domains exemplifies this effect. The object-oriented forward engineering methodology has facilitated the ease with which complex systems can be conceived and developed. However, in the software industry, there is a substantial inventory of software for which very sketchy documentation is available. Improving, updating, and maintaining software is becoming a nightmare for software professionals.

The need for understanding software has become an important issue for those who carry out software documentation, program explanation, software design, maintenance, and reengineering. All these issues can be handled successfully if a program-understanding software system can extract the relevant information from the source code and present this information at a higher level of representation than the code itself.

The motivation for this research lies in developing a program-understanding tool for understanding C++ programs. We have selected the C++ programming language for study primarily because it is widely used in industry. This program-understanding tool should provide helpful information to programmers. The information may also be useful for software redocumentation, maintenance, and reengineering.

To develop such a program-understanding tool, the following components are needed; [8]

1) A parser and a semantic analyzer to perform lexical, syntactic, and semantic analysis of the software code,

2) An information base for storing the information in internal representation form. From this information base, the view composer will compose the visual representation of information for understanding the source code.

3) A view composer to create visual representations.

However, in order to implement such a tool, we need schemata. The schemata will show how a C++ program should be visually represented in order to achieve understanding. Schemata are most important as they will be the basis for the design specification of the three components needed to develope a program-understanding tool.

The scope of this thesis is twofold:

1) We propose schemata which will be the basis for developing a program-understanding tool for understanding static properties of C++ programs. The proposed schemata will consist of three schemes, namely; an understanding schema for program-understanding, a visual representation schema for presenting information, and a navigation schema for navigation between visual representations,

2) We analyze the feasibility of developing a program-understanding tool based on the proposed schemata.

The criteria used in formulating our understanding schema are as follow:

1) The understanding schema should be as simple as possible in order to reduce complexity of understanding.

2) The terms to which the understanding schema refers should be consistent and intuitive so that they reduce the memory load of the user.

3) The understanding schema should be able to provide any static design information that may have existed as much as possible.

4) The understanding schema should be able to provide help in answering most of the typical questions raised during software maintenance.

We approach program-understanding of C++ programs from two points of view: design methodology and maintenance.

The design methodology point of view is important because the schema should be able to provide the same information as that given by previous design documents (forward engineering methodology). We have studied three prominent object-oriented methodologies (see chapter 3). Our analysis of these methodologies focuses primarily on the static properties of object-oriented software. Our proposed schema provides the design information as given by the Class diagrams, Object diagrams, and Module Diagrams of the Booch method, the Enhanced Entity Relationship (EER) of the Rumbaugh method, and the Class Responsibility Collaboration and Hierarchy Graphs of the Wirfs-Brock method (see section 3.1).

The software maintenance point of view is important in that it should help to answer the questions raised during maintenance of a C++ software. A list of typical questions raised during the maintenance of C++ programs is provided in section 4.3.

For the purpose of understanding the static properties of a C++ program, we have selected seven key concepts of C++ programming language (see section 4.1). We have categorized relationships, which may exist within a C++ key concept, under three categories: USES, HAS, and ISA (see section 4.2). In the proposed schema, we represent these key concepts and relationships existing within them with the help of seven concept templates (i.e., seven concept templates corresponding to seven C++ key concepts and relationships existing within them). To show the detailed specific information within a key concept, we use display templates (see section 5.8). The concept template representing a key C++ concept is created interactively

and groups relevant concepts under one of the three relationships that the C++ concept may have. Display templates which are used for showing the detailed specific information within a concept template are also created interactively. We use navigation schema to facilitate ease and comprehension for generating concept and display templates interactively. The proposed navigation schema describes which display templates are generated from a concept template and which type of concept templates can be reached from a display template.

1.2 Structure of the Thesis

In chapter 2, we present the basic concepts of program-understanding, including an explanation of different types of program-understanding and how they relate to various tasks in the software life-cycle. In section 2.1, we discuss the various tasks which are understanding intensive, such as software maintenance and reuse. In section 2.2, we explain what is to be understood and why, using the question-answering approach of understanding. In the last section of chapter 2, we present a brief summary of objectives to be accomplished by program-understanding.

In chapter 3, we provide a brief discussion of prominent forward engineering object-oriented methodologies and a survey of commercially available software engineering tools for C++.

In chapter 4, we explain what needs to be understood about the object-oriented code of the C++ programming language and the program-understanding schema in order to manage and understand the complexity of a C++ program. We propose that a C++ program is composed of seven key concepts and that these concepts may have three possible relationships. This forms a basis for information discrimination and the basis for building concept templates representing these seven key concepts of C++. In section 4.2, we provide a brief description of the relationships existing among these seven key C++ concepts. In section 4.3, we provide a list of typical questions raised during software maintenance. In section 4.4, we provide a detailed

description of the concept templates representing the seven key concepts of C++. Wherever required, a C++ code fragment is presented to clarify the presentation.

In chapter 5, we provide visual representation schema for concept templates and display templates. The navigation schema presented in this chapter provides the details of which display templates are generated from the concept templates and which concept templates are generated from display templates. In brief, this chapter provides the visual representation schema for concept and display templates as well as the navigation schema for navigating between the concept and display templates.

In chapter 6, we present the work accomplished for implementing a tool based on the proposed schemata. In section 6.2, we discuss future work on this schema.

Chapter 2 Program-understanding

In this chapter, we present the concept of program-understanding, its application domains, what is to be understood and why, and what is to be accomplished by program-understanding.

Whitehead has described different types of understanding in his treatise on Modes of Thought [27]. The following definitions for internal, external, and logical understanding emerge from his presentation.

Internal understanding of a system involves the notion of composition and refers to the system's elements and to their relationship. External understanding treats the system as a unit and refers to its relationships with its environment. Logical understanding starts with the details and passes to the construction achieved.

Understanding has two modes of advance: the gathering of detail within an assigned pattern, and the discovery of a novel pattern with its emphasis on novel detail [27].

There are three broad activities that enhance the understanding of the software under study [6]:

a) the identification of the product's components and their interrelationships;

b) the enhancement of the existing presentation;

c) the creation of a new representation.

Program-understanding is a reverse engineering process in which the high level language source code of a software system is analyzed to recover functionality-oriented information. The information is then represented in another form or at a higher level of abstraction than the code itself. Inherent to program-understanding is the use of knowledge about the programming language needed to parse the software and analyze its semantics.

2.1 Program-understanding Application Domains

Program-understanding may aid the software validation, verification, and maintenance activities of the software life-cycle. It may also aid the reuse, reverse documentation, redocumentation, analysis, and design recovery activities in software reverse engineering.

2.1.1 Validation and Verification

Validation and verification are the processes with which the programmer verifies that the software meets its specification. For this activity, it is important to know how the software performs the various tasks that are specified and where the task-specific code is implemented. Questions such as "What does the code do if a certain condition is true?" [20] need to be answered to understand the program from the validation and verification perspective.

2.1.2 Software Maintenance

Software maintenance is the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment" [2]. Software maintenance activities may vary in type and scope [9].

Normally, the software professionals who maintain the software are not its designers. Furthermore, in many cases the developers and/or the designers are no longer available for consultation. In order to have cost and time-effective maintenance, the maintenance team should consult only the information necessary to make a correct change. This is possible if the maintenance team can enhance their understanding of the software system. Questions which arise in the maintenance environment include "Where is a particular functionality implemented ?", "What is the function of a program fragment?", and "What are the differences between the original version and the changed version of the program?" [18].

2.1.3 Reuse

Reuse involves producing software components which can be used again in building other products. Reuse can occur during maintenance, reengineering, and restructuring. It can occur within a software system, among various software systems, or between a system and a library of reusable components. Typical questions which need to be answered to achieve reuse are: "what does a component do?" and, "where is the component behavior implemented?" [6].

2.1.4 Reverse Documentation

Reverse documentation involves creating a representation different from existing representations. The partial or full information represented in reverse documentation comes from a level of abstraction lower than that associated with the representation being created [6].

2.1.5 Redocumentation

Redocumentation also involves creating a representation about the software that is different from existing representations. The difference between redocumentation and reverse documentation is that, in redocumentation the information used in creation of the representation does not come from a level of abstraction lower than that associated with the representation being created [6].

2.1.6 Analysis

Software analysis in the context of program-understanding relates to characterizing quality and complexity aspects of the software program. It generates the values to be used in various software metrics, e.g. cyclomatic complexity and Halstead's metrics. Characteristics such as " a count of the number of variables or objects in a module" and "identification of the module interface" could be included in analysis [6].

2.1.7 Design Recovery

Design recovery involves creating a new representation at a higher level of abstraction. However, in creating new, meaningful higher level abstraction domain knowledge, external information and deduction are added to the observations of the system under study. According to Ted Biggerstaff , "Design recovery must reproduce all the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus it deals with a far wider range of information than that found in conventional software-engineering representation or code." [6]

2.2 What is to be Understood and Why

This section presents a discussion about what needs to be understood and why it needs to be understood so that it can be claimed that program-understanding has been achieved. The degree of achievement of understanding directly correlates to the extent with which the understanding can be applied to the various application domains of program-understanding.

The understanding capability lies in answering questions which are typically raised in order to successfully carry out the various processes discussed in the above section. "The question-answering model of understanding is supported by the empirical studies of the behavior of programmers during understanding tasks" [18]. The typical questions that arise in understanding [18] are given below. Using identification words such as *what, when and where* to identify question types is only a heuristic device. It is not claimed that each of these English words has a unique inherent meaning that corresponds to our definitions. In fact, the natural meanings of these words appear to be quite flexible and overlapping [18].

What? An example of a What? question is *what does <software module> do?* The answer to this question provides a concise functional summary of the behavior of the module. This

information is needed by software team members reading through a program; it allows them to verify the module and in generally understand the program. This information is also needed for software reuse.

What-If? This type of question is sometimes a special restrictive case of the What? type of question. Restrictive conditions at specified points in the software code that must hold give rise to a What-If? type of question, for example,*what does < software module> do under <condition>*. This type of question provides an answer similar to those provided by unrestricted What? questions, but describes only those execution paths where the condition holds.

When? The logical conditions that cause a particular behavior of a software module relates to a question such as *when does <software module> do <behavior>?* Specifically, such questions ask for the logical conditions which make the module behave in a given manner when a specific type of control path occurs. This maps the conditions onto the execution path, unlike What-if? questions which map execution paths onto conditions. When? type of questions are commonly asked during verification and validation, and bug diagnosis.

Why? This type of question relates the overall contribution of a software module to the system's global behavior. One example is *why does <software module> do <behavior>*. One way to answer this type of question is to identify other modules of the system that may be affected if any changes are made to this module.

Where? The answers to questions such as *where does <system> do <behavior>?* indicate the location at which the implementation of a specified function of the system is done. The answers to these questions are required in verification, code reading, and module recovery and retrieval in reuse.

Difference? The effects of a change where component-new is a changed version of

component-old is provided by questions such as *what are the differences between <software module-old> and <software module-new>?* The answers to such questions are needed when introducing maintenance changes. Reconstruction of design rationales (e.g., explanations of why functions were implemented in a particular manner by inspectors during the formal inspection process) can be viewed as questions about the differences between the original and proposed alternative version.

2.3 Program-understanding Objectives

What do we wish to accomplish by program-understanding? The main objective is to increase overall comprehensibility of the program for both maintenance and new development. There are other objectives as well: we wish to manage complexity, generate alternate views, recover lost information, perform impact analysis, synthesize higher abstractions, and enable reuse. These objectives are discussed below.

Manage complexity: program-understanding is one of the methods to deal with the sheer size and complexity of the system. The use of program-understanding schemata combined with the use of program analyzers provides a way to extract the relevant information. This information can then be used by the decision-makers, developers, and maintainers of the software system to control the product during its evolution.

Generate alternate views: Simon [26] argues that in problem solving a proper representation of the problem can make its solution transparent. It has long been recognized that a pictorial representation aids comprehension. In software development, creating and maintaining pictorial representation is a bottleneck. Program-understanding tools facilitate the generation or regeneration of representations from other forms. From the proposed program-understanding

schemata (see chapter 4) the additional views such as control flow diagram, module diagram, etc., can also be created.

Recover lost information: During the evolution of a large and long-lived system, the information related to design is often lost. Modifications carried out during software maintenance related to bug correction, performance enhancement etc., are seldom reflected in documentation, particularly at a level higher than the code itself. Program-understanding provides a means to deal with the situation even in the absence of past information.

Perform impact analysis: A poor initial design coupled with later modifications in the product can lead to unwanted and unpredictable performance (in some situations) of the product. Program-understanding can provide information beyond what is available with a forward engineering perspective. This information can help in identifying the discrepancies and inconsistencies before they are reported as bugs.

Synthesize higher abstractions: Program-understanding provides the developers with methods and techniques to create alternate views that transcend to higher level abstraction. The generation of higher level abstractions helps in defining to what level the process can be automated.

Enable reuse: The reuse of a system or its components depends greatly on how well it is understood. If full understanding is achieved, the chances of component reuse increases substantially.

Chapter 3 Design Methodologies and Software Engineering Tools for C++

In this chapter, we present a discussion of three prominent object-oriented forward engineering methodologies and a survey of reverse engineering tools for C++. The study of the forward engineering methodology is important because the proposed understanding schema addresses the issues related to these methodologies.

3.1 Object-Oriented Design Methodologies

3.1.1 The Booch Method

The Booch method fully supports object-oriented concepts. For notation, this method uses six basic diagrams made up of a small set of icons. These notation can be made very detailed, as these icons can capture more information through additional annotations. A description of the concepts and the notations [7] used in the Booch method is given below.

3.1.1.1 Concepts

Objects and Classes: In Booch's method, an object is something that has its own identity, state, and behavior. A class is a template for a group of objects that share a common structure and behavior. In this method, we have the concept of generic classes. A generic class acts as a template to other classes. This method also supports the concept of metaclass. A metaclass is a template whose instances are *class objects*. A class object's attributes provide the information common to an entire group of objects of one class and an operation to create new instances of that class. All these types of classes are represented by a single icon. However, the distinction among these classes is made on the basis of the relationships in which they participate. An abstract class defines the partial definition of a class which is used in inheritance.

Inheritance: Inheritance is a relationship between classes in which the features of one class are partially defined in terms of other classes. A class which inherits is called a *subclass* and the class or classes (multiple inheritance) from which a class inherits are called *superclasses*. The Booch method separates the notion of subtype inheritance from derived inheritance. The subtype inheritance allows a subclass to behave like its superclass for all the operations of the superclass. The derived inheritance introduces a new type of class.

Visibility: The Booch method provides a general use relationship for specialization into use in interface and use in implementation. In the case of use in interface the used objects are accessible as parameters, whereas in use in implementation the use of objects is completely hidden in the using class. During interaction of objects it is possible to show whether objects that interact are visible by one being the component of another, being passed as a parameter, or being in lexical scope.

Lifetime: While dealing with static systems of objects, all the objects have the same lifetime as the system. If that is not the case then the method must contain some facility for dynamically *creating* objects, e.g. by instantiating a class. The Booch method shows the persistence of objects by indicating in the template either for an instance or, for all instances, in the class. The timing diagram of the dynamic model captures the creation and the destruction of objects.

Concurrency: The diagrams in the Booch method label the objects which have their own thread of control as active. The Booch method does not address the issue of internal object concurrency [4].

Communication: Objects consist of a loosely coupled model of communication which provides both information flow and synchronization. The Booch methodology captures the following types of message passing: simple, synchronous, balking, time-out, and asynchronous.

3.1.1.2 Notations

There are six basic diagrams used in the Booch method. These diagrams are made up of a small set of icons. Additional notations are used in these icons to capture additional information.

Class Diagram: A class diagram is used to show the existence of classes and the relationships between them. Class categories can be created by organizing class diagrams into chunks. This categorization helps in understanding complicated class diagrams. A class diagram is a part of the logical design of a system.

Object Diagram: An object diagram is used to show the existence of an object and its relationships (e.g., message passing) between them. The mechanism denotes which one object can pass messages to other objects. These include component, parameter, or lexical scope. An object diagram is part of the logical design of the system.

State Transition Diagram: A state transition diagram shows the state space of a class, the events that cause a transition from one state to another, and the action triggered as a result of state change. It is a part of the dynamic model.

Timing Diagram: A timing diagram shows the dynamics of message passing in an object diagram. The suggested notations for showing the dynamics are: the arcs of an object diagram, pseudocode, and a timing diagram similar to that used in hardware to indicate flow of control between objects and methods.

Module Diagram: A module diagram shows the allocation of classes and objects to modules. In order to make understanding of complicated module diagrams simpler, module diagrams can be broken up into chunks called subsystems. A module diagram is part of the physical design of a system.

Process Diagram: A process diagram is used to capture the allocation of processes to physical processors. Process diagram is also part of the physical design of a system.

3.1.2 The Rumbaugh Method

The Rumbaugh method fully supports the concepts of object-oriented software development. The methodology is well defined and encompasses analysis, design, and implementation, the notations used are concise and taken from SA/SD and Harel and [4].

3.1.2.1 Concepts

Objects and Classes: The Rumbaugh method support objects, classes, and metaclasses.

Inheritance: Both single and multiple inheritance is supported by this method, as well as properties such as whether subclasses have overlapping feature.

Visibility: A rich set of aggregation primitives, including recursion, is provided.

Lifetime: Minimal support for object creation, destruction, and persistence is provided.

Concurrency: The expression of inter and intraobject concurrency is supported.

Communication: An asynchronous model of communication is employed.

3.1.2.2 Notations

This method uses three notations to capture object, dynamic, and functional models of the system.

Enhanced Entity Relationship (EER): The EER capture the main entities of the system under development and their static relationship

Harel Statechart: The Harel Statechart captures the sequence of events, states, and operations that occur between system of objects.

Data Flow Diagram(DFD): The DFD capture the flow of values from external inputs, through operations and internal data stores, to external output.

Various structuring mechanisms for the diagrams are possible: object and event classes can be arranged into a hierarchy, and state transition and data flow diagrams can be nested.

3.1.3 The Wirfs-Brock Method

The Wirfs-Brock method fully supports object-oriented concepts. The process is better suited for an individual developer rather than a team of developers because it is informal and exploratory. The method can be used for analysis or high-end design [4].

3.1.3.1 Concepts

Object and Classes: Both objects and classes are supported with standard definitions.

Inheritance: Both abstract and concrete classes and single and multiple inheritance are supported.

Visibility: The using relationship in visibility is supported.

Lifetime: The lifetime of object instances is not explicitly discussed. Consequently, managing dynamic object creation and deletion is not supported. It is implicitly assumed that object instances can be created and destroyed. Object persistence is not supported.

Concurrency: No consideration is given as to whether objects are active or passive, so no support is provided for concurrency in the method.

Communication: An abstract level discussion is provided for communication between objects. Communication uses the client-server model. No detail is provided for communication type (e.g., synchronous).

3.1.3.2 Introduced Terminology

Responsibilities: Responsibilty is defined in terms of the knowledge an object maintains and the actions an object performs. The first of these is refined into attributes and second into method signatures.

Collaborations: (Collaborations represent) Requests from a client to a server in fulfillment of a client responsibility.

Contract: A set of requests that a client can make to a server. The server is bound to respond to these requests.

3.1.3.3 Notations

Class Responsibility Collaboration (CRC): CRC cards are used throughout the design process to record information related to classes. The super- and subclasses are recorded on a CRC card, as is the responsibility of a class together with collaborator classes. Toward the end of the design process, the CRC card is developed into a class specification.

Subsystem Card (SC): A short description of a subsystem is recorded on the SC. Contracts required by clients external to the subsystem are noted with the class within the subsystem that supports the contract. The SC card is developed into a subsystem specification towards the end of the design process.

Hierarchy Graphs (HGs): HGs are a standard representation for inheritance hierarchies. Single and multiple inheritance can be denoted, as well as concrete and abstract classes.

Venn Diagram (VD): VDs are used as a tool to explore and refine inheritance hierarchies. Each class is viewed as being a set of responsibilities. Common responsibilities are drawn in the overlapping part of Venn diagram and independent parts in the non-overlapping parts.

Collaboration Graphs (CGs): CGs are used to display and analyze the path of communication between classes. Classes, inheritance relationships, contracts, and collaborations are represented in CGs.

Contract Specifications (CSs): CSs are templates used to fill out the details of the contracts. A contract specification contains name of the server and clients that collaborated to fulfill the contract, as well as an informal description of what the contract does.

3.2 Survey of Software Engineering Tools for C++

For software development, the most basic tool required is a compiler. Initial work on C++ was done under the UNIX environment; however, much has also been done to produce an equivalent compiler for MS-DOS. Borland, Zortech and other companies have produced C++ compilers with impressive lists of features, such as editing, project management, an extensive C function library, and a library of ready-to-use classes.

In order to produce good quality software much more than a compiler is needed. The following paragraphs present a survey of commercially available tools to aid software development in C++.

C++/Softbench, a workstation product from Hewlett Packard, was one of the first tools developed. Described in detail by Arnisted [3] it can be considered a CASE tool. C++/Softbench was designed to address issues related to the design and management of large projects written in C++. C++/Softbench consists of seven tools and class libraries. It consists of C++ developer, Static Analyzer, Program Builder, Program Editor, Program Debugger, Development Manager, and Mail. The graphical construction of classes including an inheritance hierarchy, is achieved with the C++ developer tool. Using a graphical browser, classes can be added or deleted and inheritance hierarchy can be changed. Source code templates are generated automatically from

the graphical representation. Viewing and editing of the members of a class is also allowed by the C++ developer. The Static Analyzer of C++/Softbench allows the user to see program information that is specific to C++, such as the names of all the classes. Some of the feature which are common to the most recent compiler are also present in C++/Softbench. A program with many source files can be compiled using the Program Builder. The user can choose to replace the Program Editor with another editor of his choice. The version control and integration of other tools with C++/Softbench is done by the Development Manager. C++/Softbench falls in the category of CASE tool.

Teamwork/OOD from Cadre Technologies is another workstation tool which consists of a graphical editor for the design stage and a C++ code generator which automates the production of code from the design [16].

Interactive Development Environments (IDE) has also developed a set of workstation CASE tools aimed at C++, including Software through Pictures. It is an analysis and design tool using a wide variety of methods including Booch's earlier approach to object-oriented analysis [24] and the traditional SA/SD approach. OOSD/C++, also developed by IDE, allows the programmer to start with a graphical design editor and through the use of a C++ reuse library at the design phase and end up with a design suitable for implementation in C++ [11].

C-DOC, from Software Blacksmiths, is a program analyzer and documentor for C and C++ under MS-DOS environment. Its features include path complexity diagram for each method, reformatting of code, a hierarchy tree for called and calling methods, a table of contents, and cross reference of identifiers. This product falls into the reverse engineering category [29].

GUI_MASTER, from Vleermuis Software Research (Netherlands), is a tool which automatically produces a graphical user interface for a C++ program under the MS-DOS environment.

It is a forward engineering tool that writes a skeleton program with pop-up menus. All code other than user interface has to be written by the user [30].

Objectcraft allows the developers to do visual programming and automatically generates C++ code [13]. Improvements include writing C++ methods within ObjectCraft, importing a C++ files, and printing a visual programming diagrams [15].

It is found from the survey that none of the tools provide a consistent visual representation schema satisfying both the object-oriented and structured concepts of software design and programming. In our research, we have proposed an understanding schema satisfying both object-oriented and structured requirements in software design and programming

Chapter 4 Program-understanding Applied to C++

In this chapter, we present descriptions of some key C++ concepts and the relationships existing within them with other concepts. In addition, we provide a list of typical questions (based on section 2.2) raised during the maintenance of C++ software.

Deciding how to answer typical questions raised during software maintenance provides a guideline for proposing concept and display templates. For constructing concept templates, we identify the key concepts of C++ and the different types of relationship which can exist between those concepts. Each concept template represents a C++ concept and the relationships which may exist between it and other concepts, namely the USES, HAS, and ISA relationships.

4.1 C++ Key Concepts

For the purposes of this thesis, a C++ program is understood to be any program written in C++. In order to understand a C++ program, we must first know what a C++ program looks like, conceptually. We have identified the following as key C++ concepts in the context of program-understanding:

Executable File: An executable file of a C++ program is a file which is directly executable from the command prompt of a computer system. This file is created either by linking together various object files by means of specific command line arguments to a compiler or by directly compiling one program.

Object File: An object file of a C++ program is a file which is created by compiling a source file with include files (i.e., header files) by means of specific command line arguments to the compiler. The same header files may be shared with other object files.

Source File: A source file of a C++ program is a file where the implementation code is written to perform one or more specified tasks. In order for the program to achieve the specified tasks, the software is written in a source file with statements for including header and library header files, declaring global objects and variables, declaring abstract data type templates, declaring methods, implementing methods, etc.

Method: There are three types of methods in the C++ programming language: ordinary methods, member methods, and parametric methods. A member method is declared in an abstract data type template and can be called only by an object which is an instance of that abstract data type or by an object such that its data type template is declared as *friend* to an abstract data type template. Other types of methods can be called anywhere in a source file as long as they are declared before they are called. Methods have unique names, with the exception of member methods and parametric methods, which are *overloaded* and *parameterized* respectively. However, *overloaded* functions declared in an abstract data type must have a unique parameter list. At run time, these methods have their own separate temporary memory address spaces, which are released once the last statement of the method is executed. A task is performed by calling methods in a particular order. In a method, code is written; for calling other methods, local/global objects and/or variables are declared and updated or used in updates.

Logical Block: A logical block is a section of code in a method for which memory address space is allocated separately at run time. The objects and/or variables (if any) that are declared within a logical block are not accessible outside of that block. A logical block is created using braces; the braces "{" and "}" signify the start and end of a logical block, respectively. Depending upon how the start of a logical block is written, these blocks can be classified as selection blocks (e.g., if, if-else), recursive blocks (e.g., while, do-while), and simple logical

blocks by using the braces (e.g., { *<software code>* }). In a logical block, apart from its local declarations, code is written for making calls to other methods and updating local/global objects or variables. Logical blocks can also be nested; e.g., an *if-else* logical block can be placed within a *while* logical block.

Abstract Data Type Template: An abstract data type template is a declaration written by using one of the following *type specifiers: class, struct, or union.* Within an abstract data type template, data members, member methods and their attributes (e.g., *public* etc.), and other attributes (e.g. *friend <abstract data type>*) are declared. An abstract data type template can also inherit from other abstract data type templates. An abstract data type template can access the members of other abstract data type templates by using the keyword *"friend"* in the declaration. As the name indicates, these are templates for creating objects and by themselves do not require any memory.

Object: An object is an instance of an abstract data type. An object occupies memory as per the declaration of its template. An object can invoke its member methods and the member methods of a *friend* abstract data type template. The scope of an object depends upon its location and type of declaration, e.g., external, local to a method.

4.2 Relationships Existing Between C++ Key Concepts

We have used three relationships, namely USES, HAS, and ISA, to show the relationships between C++ concepts (see section 5.2). These relationships are (if applicable) context sensitive. An interpretation of these relationships is given below.

The USES relationship indicates that one C++ concept interacts with another, but does not contain it. For example, an executable file (see section 5.4.1) interacts with other object files to create itself, but it does not contain any object files nor any specialization of any

executable files. The context sensitivity of a USES relationship can be illustracted in the following way: if a method is called from within a logical block (see section 5.4.5.1) of a method, then the called method will be shown only as *<logical block >* USES *< method>*, not as *<method>* USES *<method>*. It is important to know under what condition a method is called by another method. If we show the above situation as *<method>* USES *<method>* because this call is made from within a method, this information is erroneous because method is called only when it reaches that logical block. To elaborate further, if we suppose the logical block to be an *if* block, then the significance of this information will be lost.

The HAS relationship indicates that one C++ concept contains another C++ concept. For example, a method is always contained in a source file. The context sensitivity of the HAS relationship is shown as follows: if a data member is declared in an abstract data type template and in turn this abstract data type is declared in a source file, then this data member will be only shown as *<abstract data type template>* HAS *<data member>* not as *<source file>* HAS *<data member>*. The latter representation is misleading because it is not a source file but rather the data template which owns the data member.

The ISA relationship indicates that one C++ concept is a specialization of one or more other C++ concepts. For example, all objects of an abstract data type template are separate specializations of that abstract data type template. In the case of inheritance, an abstract data type template which is inheriting from other templates (multiple inheritance) is a specialization of these templates.

4.3 Typical Questions Raised During C++ Software Maintenance

The discussion provided in section 2.2 provides the basis for formulating these questions. For example, question of the type *what does <software module> do?* forms a basis for question such

as what objects are declared and how they update other objects and what objects or variable update them.

- How is the executable file of a C++ program made? That is, which object files are linked together to create the executable file of a C++ program?

- How is each object file compiled? That is, which C++ source files are compiled together to create an object file?

- What is the significance of the code written in a source file? That is, which other source files are included, which global objects are declared, what code for which methods is written in a source file, etc.?

- What is the significance of the code written in a method? That is which local variables and objects are declared, what methods are called from this method (and how they are called), which variables and objects are updated and which variables or objects update them, how many and which types of logical blocks (selection or repetitive blocks, the blocks made using "{ }") are written in a method, etc.?

- What is the purpose of the code written in a logical block ? That is, which objects and variables are local within a logical block, is there a nesting of blocks, which objects and variables are updated and how, etc.?

- How and what abstract data types are declared? That is, from which abstract data type does an abstract data type inherit, which abstract data types are *friends* (C++ term) to an abstract data type, what declarations are made in an abstract data type, etc.?

- What objects are declared? That is, what object is an instance of what abstract data type?

- Where are objects declared? That is the scope of an object a source file, or is it local to a method, or to a logical block in a method, etc.?
- How are objects used? That is, what objects are updated by an object/variable, and where they are updated, etc.?

4.4 C++ Concepts for Information Discrimination

In this section, we present seven key C++ concepts for information discrimination based on the relationships, namely, USES, HAS, ISA which may exist within these concepts (Figure 4.1). Details of the information grouped under the three relationships are discussed in detail for each concept. When required, a C++ code fragment is provided as an example.

C++ Concept \ Relationship	USES	HAS	ISA
Executable File C1	YES	—	—
Object File C2	YES	—	—
Source File C3	YES	YES	—
Method C4	YES	YES	YES
Logical Block C5	YES	YES	—
Abstract Data Type Template C6	YES	YES	YES
Object C7	YES	YES	YES

Figure 4.1

C++ Concepts and Relationship Diagram

4.4.1 Executable File Concept (C1)

The executable file concept interacts with the object file concept without containing it. Therefore, the executable file concept of C++ shows the USES relationship that exists for this concept (Figure 4.1). This concept groups the information related to different object files which are used to create the executable file of a C++ program. This information can be obtained only when the executable file is created using the UNIX *makefile* utility (i.e., the information grouped under C1 is obtained by analyzing *makefile*).

4.4.1.1 USES Relationship

The USES relationship of executable file concept provides information about the object files that are linked together in the creation of the executable file.

4.4.1.2 HAS Relationship

As the executable file does not directly contain another C++ concept, no HAS relationship exists for this concept.

4.4.1.3 ISA Relationship

As the executable file is not a specialization of any executable file, no ISA relationship exists for this concept.

4.4.2 Object File Concept (C2)

The object file concept interacts with the source and header file concept without containing it. Therefore, the object file concept shows the USES relationship. C2 groups the information related to the source files and header files that are compiled together to create an object file. This information can be obtained only when the executable file of a C++ program is created using the UNIX *makefile* utility.

4.4.2.1 USES Relationship

The USES relationship of C2 provides information about the source files and header files that are compiled together to create an object file.

4.4.2.2 HAS Relationship

An object file does not directly contain another C++ concept, therefore no HAS relationship exists for this concept.

4.4.2.3 ISA Relationship

As the object file is not a specialization of another object file, therefore no ISA relationship exists for this concept.

4.4.3 Source File Concept (C3)

The source file concept is not a specialization of another C++ concept. However, the source file concept contains other C++ concepts within itself and interacts with other concepts of C++. Therefore, the source file concept of C++ (Figure 4.1) shows only the USES and HAS relationships. C3 groups the information related to the source code written in a source file. This information is obtained by analyzing the source code. This concept groups the information up to the declaration level, i.e., a method written in a source file is shown by the relationship *<source file>* HAS *<method name>* and details about a method are shown by method concept template.

4.4.3.1 USES Relationship

The USES relationship of C3 provides information about:

- Library files, as a list of library files which are referenced by an **#include** < > declaration in a source file.

- Source files, as a list of source files which are referenced by an **#include " "** declaration in a source file.
- External objects, as a list of objects which are referenced in a source file but are created outside this source file, i.e., objects declared using an **extern** storage specifier.
- External variables, as a list of variables which are referenced in a source file but are created outside this source file, i.e., variables declared using an **extern** storage specifier.

4.4.3.2 HAS Relationship

The HAS relationship of C3 provides information about:

- Global objects, as a list of objects having global scope in a source file.
- Global variables, as a list of variables having global scope in a source file.
- Methods, as a list of methods for which the code is written in a source file.
- Macro expansions, as a list of macro expansions declared in a source file, e.g., ***#define MAX 100.***
- Abstract data type template, as a list of abstract data type template names for which the code is written in a source file.

4.4.3.3 ISA Relationship

As a source file is not a specialization of another source file, therefore no ISA relationship exists for this C++ concept.

4.4.4 Method Concept (C4)

The method concept shows all the three relationships, USES, HAS, and ISA (Figure 4.1). C4 groups the information related to the code written in a method, i.e., the logical blocks, their

types and interface, the objects and variables local to this method, etc. For a logical block, C4 provides the information only up to declaration of that logical block. This information is shown via the HAS relationship, i.e., the information about the code written in a logical block is shown logical block concept (C5).

4.4.4.1 USES Relationship

The USES relationship of C4 provides information about:

- Another method, as a list of methods called from within a method, e.g.,

```
void myfunc()
{
  ...
  yourfunc ();
  ...
}
```

- Global objects, as a list of global objects which are used on the left or right hand side of the assignment statements within a method, e.g.,

```
void myfunc ()
{
  ...
  A=B;

  /* "A" global object used on the lhs
   "B" global object used on the rhs */

}
```

- Local objects, as a list of local objects declared within a method used on the left or right hand side of the assignment statements, e.g.,

```
void myfunc()
{
  Edgelist edges;
  edges = Edges;

}
```

- Global variables, as a list of global variables which are used on the left or right hand side of the assignment statements within a method, e.g.,

```
void myfunc ()
{
  ...
  a=b;

}
```

- Local variables, as a list of local variables (variables declared within a method) used on the left or right hand side of the assignment statements, e.g.,

```
void myfunc ()
{
  ...
  a=b;

}
```

4.4.4.2 HAS Relationship

The HAS relationship of method concept C4 provides information about:

- Local objects, as a list of objects which are declared within a method,.e.g.,

```
void myfunc()
{
  Edgelist edge;//local object
  ...
}
```

Although this information looks similar to the information given by *<method> USES <local objects>*. However, it provides useful information in relation to the objects that are declared within a method and those which are actually referenced within a method, i.e., it provides information about those objects which are unnecessarily declared within a method.

- Local variables, as a list of variables which are declared within the method, e.g.,

```
void myfunc()
{
  int number;//local variable
  ...

}
```

- Logical block, as information about how many logical blocks exist in the code of the method, e.g.,

```
void myfunc ()
{
  {
    int i;
```

```
        while (i < 30)
          i++;
      }

    }
```

- Local abstract data type templates, as a list of the abstract data type templates that are local to this method.

4.4.4.3 ISA Relationship

The ISA relationship of method concept provides the parameterization information of a method.

4.4.5 Logical Block Concept (C5)

The logical block concept shows the USES and HAS relationships (Figure 4.1). C5 groups the information related to code written in a logical block, e.g., nesting of logical blocks, the objects and variables that are local to this logical block. For nesting of logical blocks, C5 provides information only up to the declaration of next logical block. This information is shown via the HAS relationship, i.e., the information about a logical block within a logical block is grouped again by creating another logical block concept (see section 5.4).

4.4.5.1 USES Relationship

The USES relationship of logical block concept C5 provides information about:

- Method, as a list of methods called from within this logical block, e.g.,

```
void myfunc()
{
  ...
    {
      if (a==b)
      {
        yourfunc ();
        printf("HELLO\n";
      }

  //yourfunc called only from "IF" Block
      ...
    }

}
```

- Global objects, as a list of objects which are used on the left or right hand side of the assignment statements within a logical block, e.g.,

```
void myfunc ()
{

  {
    ...
      while (i<10)
      {
        A[i] = B[i];
    // A[i] and B[i] not declared here
           i++;
      }
  }
}
```

- Local objects, as a list of local objects (the objects declared within the method) used on

the left or right hand side of the assignment statements, e.g.,

```
void myfunc()
{
  {
    if ( ptr )
    {
      Edgelist edges;
      edges = Edges;
    }
 //edges not visible outside the "IF" block
   }
}
```

- Global variables, as a list of global variables which are used on the left or right hand side of the assignment statements, e.g.,

```
void myfunc ()
{
  {
    ...
    for (i = 0; i< 10;i++)
    {
      a=b;
      b=c;
    }
    // a and b are global variables
   }
}
```

- Local variables, as a list of local variables (the variables declared within the method) used on the left or right hand side of the statements, e.g.,

```
void myfunc()
{
```

```
    {
      int i;
      switch (i)
      {
        case 1: printf("%d\n",number+i);
        default: break;
      }
   /* Variable number is not visible outside
    this switch block */
    }
}
```

4.4.5.2 HAS Relationship

The HAS relationship of C4 provides information about:

- Local objects, as a list of local objects which are declared within the logical block. e.g.,

```
void myfunc()
{
  {
    while (i > 0)
    {
      Edgelist *ptr = &edge;
      ptr->nxt = NULL;
      cin << i;
    }
  // ptr not visible outside while Block
    ...
  }
  ...
}
```

- Local variables, as a list of local variables which are declared within the method, e.g.,

```
void myfunc()
{
      {
         if (!ptr)
         {
         int number=99;
         ptr.num = number;
         }
 // number not visible outside "IF" block
      } ...
 }
```

- Logical block, as information about how many logical blocks are written within the logical block itself, e.g.,

```
void myfunc()
{
  int a;
  a=b;
    if (a > 99)
    {
      {// Logical Block 1
        int b;
        {// Logical Block 2
          int c;
        }
      }
    }
  /*"IF" block has two logical block of type
   "{ }" */
}
```

- Local abstract data type templates, as a declaration made for abstract data type templates

which are local to a logical block; i.e.,

```
void myfunc( )
{
  int i;
  i = 10;
  {
    if (process == TRUE)
    {
      class Myclass { ... };
      Myclass yourclass;
      yourclass.message;
    }
  /* Myclass template not visible outside
    "IF" block */
  }
}
```

4.4.5.3 ISA Relationship

As logical block is not a specialization of another logical block, therefore no ISA relationship exists for this concept.

4.4.6 Abstract Data Type Template Concept (C6)

The abstract data type template concept (C6) shows all the three relationships, USES, HAS, and ISA (Figure 4.1). C6 groups the information related to declarations made in the abstract data type template, e.g., declarations of data members and member methods. For inheritance and friend abstract data type templates, C6 provides information only up to a declaration of those templates; i.e., information about these templates is grouped in another abstract data type template concept.

4.4.6.1 USES Relationship

The USES relationship of abstract data type concept C6 provides information about:

- Abstract data type: an abstract data type uses another abstract data type without encapsulating it, e.g.,

```
class Graph
{
  public:
       friend class FriendClass;
       ...
};
```

- Method: An abstract data type uses a method (non-member method).

4.4.6.2 HAS Relationship

The HAS relationship of abstract data type concept C6 provides information about:

- Abstract data type template nesting: an abstract data type declaration contains another abstract data type declaration. A template has another template defined within itself, e.g.,

```
class Graph
{
  ...
  struct Node
  {
    ...
  };
  //Abstract data type nesting
};
```

- Data members: an abstract data type has a data member defined in it, e.g.,

```
class Graph
{
  int number; // data member number
        ...
};
```

- Member method: an abstract data type has member methods in it, e.g.,

```
class Graph
{
  int nodeSize (...);
  //nodeSize is the member method
};
```

4.4.6.3 ISA Relationship

The ISA relationship of abstract data type concept C6 provides information about:

- Inheritance: an abstract data type is derived from another abstract data type, therefore the inherited abstract data type is a specialization of another abstract data type, e.g.,

```
class Graph: Mgraph /* Graph inherits
            from Mgraph to
            construct itself*/
{
  ...

};
```

- Parametric abstract data type template: an abstract data type template is constructed using parameters, i.e., it is a specialization of another parametric abstract data type template.

4.4.7 Object Concept (C7)

The object concept (C7) shows all the three relationships, USES, HAS, and ISA (Figure 4.1). C7 groups information related to how an object is used, instances of a given abstract data type template, etc.

4.4.7.1 USES Relationship

The USES relationship of object concept C7 provides information about:

- Member method: an object uses its own member methods in the source code, e.g.,

```
class Graph
{
  void sortGraph ();
  ...
}

Graph myGraph;
main ()
{
  ...
  cin << mygraph;
  myGraph.sortGraph( );
//mygraph uses its own method sortGraph
  ...
}
```

- Other object's member method: an object uses another object's member method, e.g.,

```
class NodeList
{
  int  totalNodes;
  int  listSize();
  ...
};
```

```
class MNodeList
{
  int nodes;
  ...

};
main ()
{
   NodeList mynodes;
   MNodeList yournodes;
     ...
   cin << mynodes;
   yournodes.nodes = mynodes.listSize();
   ...
}
```

- Method: an object uses the method on the RHS of an expression to update its value,

  e.g.,

```
class NodeList
{
   int  totalNodes;
   int  listSize();
   ...
};

int listSize (NodeList *);

main ()
{
   NodeList mynodes, *ptr;
   ...
   cin << mynodes;
   ptr = &mynodes;
   mynodes.totalNodes = listSize(ptr);
   ...
}
```

- Object: an object uses an object on the RHS to update its value, e.g.,

```
class NodeList
{
  int  totalNodes;
  int  listSize();
  ...
};

class MNodeList
{
  int nodes;
  ...
};

main ()
{
   NodeList mynodes;
   MNodeList yournodes;
   ...
   cin << mynodes;
   yournodes.nodes = mynodes.totalNodes;
   ...
}
```

- Variable: an object uses a variable on the RHS of the expression to update its value, e.g.,

```
class NodeList
{
   int  totalNodes;
   int  listSize();
   ...
};

class MNodeList
```

```
{
   int nodes;
   ...
};
main ()
{
   int initialize = 10;
   NodeList mynodes;
   MNodeList yournodes;
   ...
   cin << mynodes;
   yournodes.nodes = initialize;
   ...
}
```

4.4.7.2 HAS Relationship

The HAS relationship of the object concept C7 provides information about:

- Data members: an object has data members in it, e.g.,

```
class NodeList
{
   int  totalNodes;
   int  nodeNum;
   char  nodeLabel;
   NodeList *nxt;
   ...
};

main ()
{
   NodeList myNodes;
   ...
   cin << mynodes;
   /* mynodes has data members totalNodes,
    nodeNum, nodeLabel, *nxt */
   ...
```

```
}
```

4.4.7.3 ISA Relationship

The ISA relationship of the object concept C7 provides information about:

- Instance: an object is an instance of an abstract data type template, e.g.,

```
class NodeList
{
   int  totalNodes;
   int  nodeNum;
   char  nodeLabel;
   NodeList *nxt;
   ...
} ;
main ()
{
   NodeList myNodes;
 //myNodes is an instance of class NodeList
   ...
   cin << mynodes;
}
```

Chapter 5 Visual Representation Schema and Navigation Schema

In this chapter, an example showing how a C++ program will be shown is provided to present visual representation schema for representing concepts (concept template) and displaying information grouped within those concepts (display template), and navigation schema for navigating among the concept templates and the information display templates. The details of all the display templates which are generated is given in section 5.4.

5.1 Visual Representation Schema

5.1.1 Concept Template

The concept template for representing a C++ key concept is divided into three major parts (see Figure 5.1). The first horizontal part, with the heading "Access Path," is shown in the upper right hand side of the template. The entry "Access Path" shows from which display templates (details given below) a concept template can be generated. The entry is shown either as "NONE" or as a display template identification number. The second horizontal part of the template shows the identification number and name of the concept template. The third horizontal part of the template is divided into three columns, with each column representing one of the three relationships contained within a concept. The entry for these three columns shows either "NONE," if a particular relationship does not exist in that concept, or the **"heading"** under which the information is grouped. The identification number of a display template generated to display the information grouped under a heading, is shown just below that heading. For convenience, we have used a three-digit identification number scheme for these headings. The first digit of the scheme represents the concept template id, the second digit represents the relationships

(1->USES, 2->HAS, 3->ISA) within a concept, and the third digit represents the running serial under a particular relationship. The details of seven concept templates are explained using example (see section 5.3).

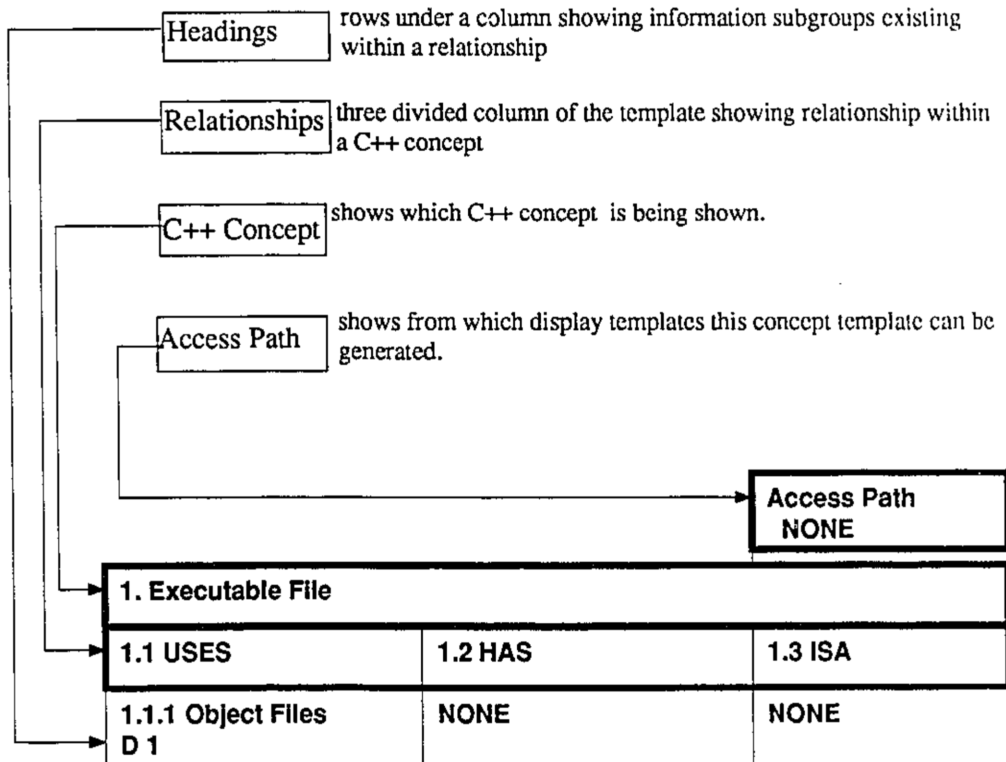
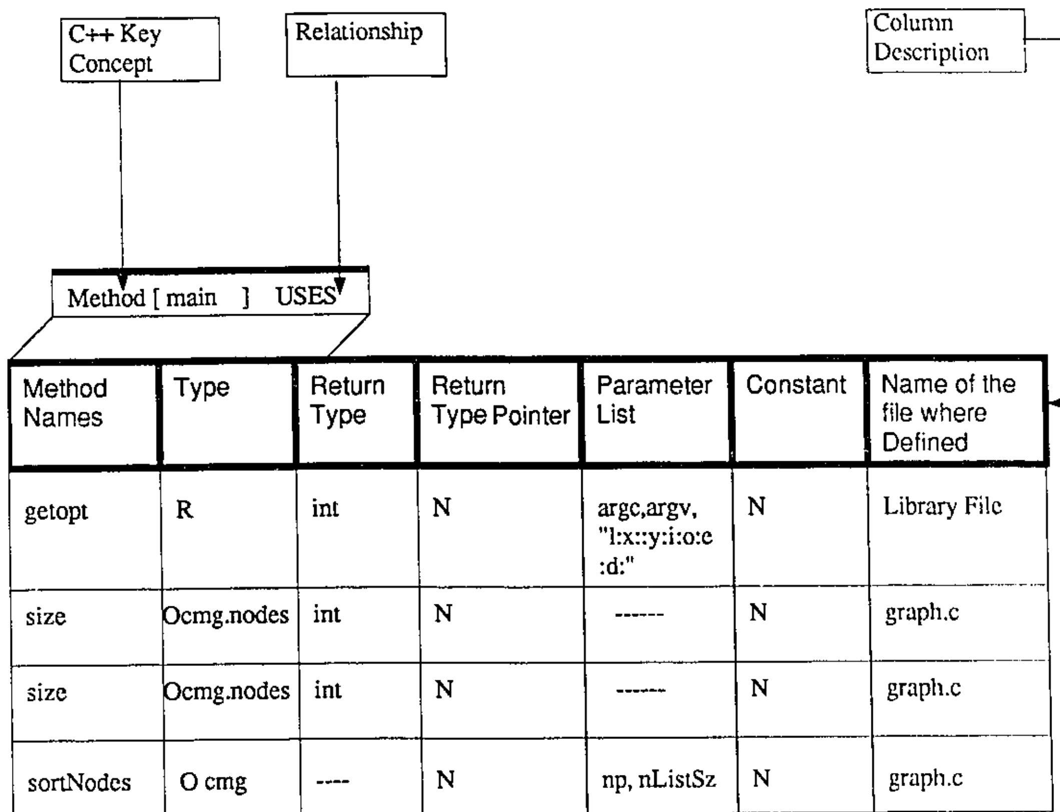


Figure 5.1

Concept Template

5.1.2 Display Template

Display Template: The display templates are used to display the information grouped under a **"heading"** (see previous paragraph) within the C++ concept. In the proposed tool specification, the display templates (Figure 5.2) are generated only from the concept templates. The display template is divided into two horizontal portions: the upper and lower portion. The upper portion is shown as a stack (wherever applicable), and shows the concept and the relationship from which this display template is generated. The space between the "[ ]" is used to display the name of the specific concept for which the information is displayed, e.g., method's name, logical block's number, etc. The lower portion of the template is arranged as a table with the appropriate headings for the columns. The details of these columns for the display templates D1 through D26 is given in section 5.4. The same display templates are used wherever the same information needs to be displayed. The information presented in the display template can also be presented in sorted order on any of the columns as a sort key. Sorting will further enhance the grouping of the information within a column.



Method Names	Type	Return Type	Return Type Pointer	Parameter List	Constant	Name of the file where Defined
getopt	R	int	N	argc,argv, "l:x::y:i:o:e :d:"	N	Library File
size	Ocmg.nodes	int	N	------	N	graph.c
size	Ocmg.nodes	int	N	------	N	graph.c
sortNodes	O cmg	----	N	np, nListSz	N	graph.c

Figure 5.2

Display Template

5.2 Navigation Schema

A concept template is generated directly and also from some display templates. However, display templates are generated only from the concept templates (Figure 5.3).

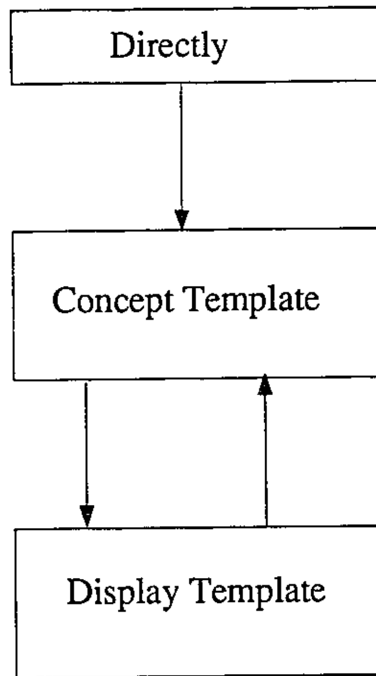


Figure 5.3

Navigation Schema

To make the presentation of the navigation schema clear, we have provided three figures:

1) A comprehensive list of the display templates generated (Figure 5.4)

2) A list of the specific display templates which are generated the concept templates (Figure 5.5)

3) A list of the concept templates which are generated from the display template columns (Figure 5.6).

Display TemplateID	Description
D 1	Object files
D 2	Source Files
D 3	Library Files
D 4	External Objects
D 5	External Variables
D 6	Abstract Data Types (a)
D 7	Global Objects
D 8	Global Variables
D 9	Methods (a)
D 10	Abstract Data Types (b)
D 11	#define Directive
D 12	Methods (b)
D 13	Objects (LHS/RHS)
D 14	Variable (RHS/LHS)
D 15	Logical Block
D 16	Member Objects
D 17	Member Methods (a)
D 18	Abstract Data Type Inheritance
D 19	Parameteric Abstract Data Types
D 20	Parameteric Functions
D 21	Member Variables
D 22	Member Methods (b)
D 23	Other Object's Member Methods (RHS)
D 24	Methods (RHS)
D 25	Objects (RHS)
D 26	Variables (RHS)

Figure 5.4

List of Display Templates

Concept Template	Display Templates Generated		
	Relationships		
	USES	HAS	ISA
Executable File C1	D1	None	None
Object File C2	D2	None	None
Source File C3	D2, D3, D4, D5	D7, D8, D9, D10	None
Method C4	D12, D13, D14	D13, D14, D15, D10	D20
Logical Block C5	D12, D13, D14	D13, D14, D15, D10	None
Abstract Data Type Template C6	D9, D6	D10, D16, D21, D17	D18, D19
Object C7	D22, D23, D24, D25, D26	D16, D21	D6

Figure 5.5

Display Templates Generated from the Concept Templates

Display Template ID.#.	Concept Templates Generated	Display Templates Column#
D1	C2	1
D2	C2	1
D3	None	
D4	C3 C6 C7	6 2 1
D5	C3	5
D6	C3 C6	3 1
D7	C6	2
D8	None	
D9	C4 C6 C7	1 2 2
D10	C6	1
D11	None	
D12	C3 C4 C6	1 7 3
D13	C6 C7	2 1
D14	None	
D15	C5	1
D16	C6 C7	2 1
D17	C4 C6	1 2
D18	C6	1
D19	None	
D20	None	
D21	None	
D22	C3 C4	2 1,3
D23	C3 C4 C7	3 2,4 1
D24	C3 C4	2 1,3
D25	C3 C4 C7	2 3 1
D26	C3 C4	2 3

Figure 5.6

Concept Templates Generated from the Display Templates

5.3 An Example: Representing a C++ Program Based on the Proposed Schema

In this section, we present the representation of a C++ program using the proposed schema. The example taken is of a tool, GLAY, developed for Bell Canada [5]. (GLAY was developed as

an example tool to be incorporated in a repository. It is used for aligning the X and Y coordinates of the components of a directed graph. The system aligns the arcs and centre points of the nodes of a directed graph onto a user defined grid. GLAY adjusts the X and Y coordinates to the nearest grid point coordinates. The X and Y coordinates of all the other associated objects of the nodes and arcs (e.g. any associated labels) are also adjusted, maintaining the relative position with respect to their associated node or arc.) The example consists of a `makefile` for creating the executable file and the three source files. A listing of the `makefile` and the source files is given in A.1 through A.4. For representation purposes, we have chosen one case for each of the seven key C++ concepts for concept templates. The display templates generated from these concept templates are also shown. Using the same basis, the complete program representation can be easily shown. The source code for the selected cases in the source files (see Listings A.1 through A.4) is shown in bold typeface. Each of the selected cases is shown in detail to answer the following questions:

1) How concept templates are generated?

2) How does the concept template look for the selected case?

3) What display templates are generated from a selected concept template?

4) What information is shown in display templates?

5) What other concept templates are generated from the display templates?

5.3.1 Example Description

5.3.1.1 Makefile for Making a "glay" Executable File

For the concept template, in the makefile (see Listing A.9.1):

1) `glay` is used for the executable file concept,

2) `glay.o`is used for the object file concept,

3) `glay.cc` is used for the source file concept.

5.3.1.2 Source File "glay.cc"

For the concept template purpose, in the source code (see Listing A.9.2);

1) `main (int argc, char *argv[])` is used for the method concept,

2) `Graph cmg;` is used for the abstract data type template and object concept,

3) `for (int i=0; i < nListSz; ++i)` is used for the logical concept.

5.3.1.3 Source File "graph.c"

The name of the source file "graph.c" (see Listing A.9.3) will appear in one of the display templates generated from the source file concept template for `glay.cc`.

5.3.1.4 Source File "graph.h"

From the file "graph.h" (see Listing A.9.4) `class Graph` is used to represent the abstract data type template concept.

5.3.2 Executable File Concept Template C1

The executable file concept template (C1) groups information related to the executable file concept of a C++ program. The order in which the object files are displayed corresponds to the order in which they appear in the *makefile*. The executable file concept template for `glay`(Figure 5.7) in the example is generated only by parsing the `makefile` (A.1). The concept template cannot be generated from any of the display templates. From this concept template, only D1 (Figure 5.8) display template is generated. From the display template D1, the object file concept templates are generated. To represent one object file concept template, we have selected `glay.o` object file.

Access Path NONE

1. Executable File		
1.1 USES	**1.2 HAS**	**1.3 ISA**
1.1.1 Object Files D 1	NONE	NONE

Figure 5.7

Concept Template C1

Executable File [glay] USES

Object File Names
glay.o
IML.o
IMLtool_glay.o
IM.o
general.o
ctools.o

Figure 5.8

D1 Display Template (from C1 (1.1.1))

5.3.3 Object File Concept Template C2

The object file concept template (C2) groups information related to the object file concept of a C++ program. The template is generated by choosing an object file from the information display template D1 generated from executable file concept template . This information is possible only when a object file is made by using the UNIX *make* utility. A list of source files is displayed (display template D2) from 2.1.1 of the object file concept template. The information display template provides the names of the source files used to create the object file and their extensions.

The object file concept template (Figure 5.9) for `glay.o` is generated by choosing *glay.o* from the display template D1 (Figure 5.8). From 2.1.1 of C2 (Figure 5.9), the display template D2 (Figure 5.10) is generated, showing the list of source files used in making the object file `glay.o` .

		Access Path D 1
2. Objects		
2.1 USES	**2.2 HAS**	**2.3 HAS**
2.1.1 Source Files D 2	**NONE**	**NONE**

Figure 5.9

Concept Template C2

Object File [glay.o] USES

Source File Names	Extension	C++ or C
glay	cc	C++
IML	.h	C++
general	.h	C++
IM	.h	C++
ctools	.h	C++
args	.h	C++

Figure 5.10

D2 Display Template (from C2 (2.1.1))

5.3.4 Source File Concept Template C3

The source file concept template (C3) groups information about the source file concept of a C++ program. C3 is generated by choosing source file from the information display template D2 through 2.1.1 of the object file concept template, or from the appropriate columns of the display templates D4, D5, D6, D12, D22, D23, D24, D26 (see section 5.4) . The information about other C++ key concepts contained within the source file concept is displayed either directly by the other information display templates, or subsequent to the generation of the concept templates therein. For example, the details of an external object (a C++ key concept) within a source file are displayed directly by display template D4, whereas the details of a method within a source file are displayed after generating concept template C4 from display template D9.

The details of the display templates generated from concept template C3 are given below.

From 3.1.1 of C3 a list of library files as well as information about whether they belong to a C or C++ library (display template D3) is shown. Activation of 3.1.2 of C3 gives a list of the source files which are included in this file and their extensions (display template D2). Activation of 3.1.3 of C3 gives a list of the external objects, their properties and the names of the source files where they are declared (display template D4). Activation of 3.1.4 of C3 gives a list of the external variables, their properties and the names of the source files where they are declared (display template D5).

Activation of 3.2.1 of C3 gives a list of the global objects created in the source file and their attributes (display template D7). Activation of 3.2.2 of C3 gives a list of the global variables declared in the source file and their attributes (display template D8). Activation of 3.2.3 of C3 gives a list of the methods for which the code is written in the source file and their attributes (display template D9). Activation of 3.2.4 of C3 gives the details of the *#define* directive in the source file (display template D11). Activation of 3.2.5 of C3 gives a list of the data abstraction templates for which code is written in the source file (display template D10).

The concept template C3 (Figure 5.11) for `glay.cc` is generated by choosing *glay* from the D2 display template (Figure 5.10). Concept template C3 is generated only when the source file is a C++ file (C++ compilers also allow the inclusion of C files). In our example, the information display templates D4, D5, D7, and D10 will not be generated, as the source file does not contain the declarations needed for generating these templates. The information displayed by generating the display templates D3 (activated from 3.1.1), D2 (activated from 3.1.2), D8 (activated from 3.2.2), D9 (activated from 3.2.3), and D11 (activated from 3.2.4) is shown in Figures 5.12 through 5.16.

		Access Paths "Direct" D2,D4,D5,D6,D12, D22,D24,D26,D27
3. Source File		
3.1 USES	3.2 HAS	3.3 ISA
3.1.1Library Files D 3	3.2.1 Global Objects D 7	None
3.1.2 Source Files D 2	3.2.2 Global Variables D 8	
3.1.3 External Objects D 4	3.2.3 Methods D 9	
3.1.4 External Variables D 5	3.2.4 #define Directtive D 11	
	3.2.5 Abstract Data Type D 10	

Figure 5.11

Concept Template C3

Source File [glay.cc] USES

Library File Names	C++ or C
iostream	C++
fstream	C++
strings	C++
stdlib	C++
stdio	C++
stddef	C++

Figure 5.12

D3 Display Template (from C3 (3.1.1))

Source File [glay.cc] USES

Source File Names	Extension	C++ or C
graph	c	C++

Figure 5.13

D2 Display Template (from C3 (3.1.2))

Source File [glay.cc] HAS

Global Variable Names	Type	Storage Type	Pointer	Array
nodeFlg	int	A	N	N
edgeFlg	int	A	N	N
gridX	int	A	N	N
gridY	int	A	N	N

Figure 5.14

D8 Display Template (from C3 (3.2.2))

Source FIle [glay.cc] HAS

Method Names	Type	Return Type	Return Type Pointer	Parameter List	Constant
main	R	int	N	int argc char *argv[ ]	N

Figure 5.15

D9 Display Template (from C3 (3.2.3))

Source File [glay.cc] HAS

Strings to be replaced	Replaced by string	Scope
YES	1	10
SEG	99	11
NO	0	12

Figure 5.16

D11 Display Template (from C3 (3.2.4))

5.3.5 Method Concept Template C4

The method concept template (C4) groups information related to method concept of a C++ program. The template is generated by choosing a method name from the information display template D9, activated from 3.2.3 of C3, or from the display templates D12, D17, D22, D23, D24, D25, D26 generated through appropriate concept templates (see section 5.4). The information about other C++ concepts contained within method concept is displayed either directly by other information display templates, or subsequent to generating the concept templates therein. For example, the details of a local object (a C++ key concept) within a method are displayed directly by display template D13, whereas the details of a logical block (a C++ key concept) within a method are displayed after generating the logical block concept template (C5) from display template D15.

The activation of 4.1.1 of C4 provides a list of the methods which are called from within the method. The display template displays their names, properties, and the files in which they

are defined (display template D12). Activation of 4.1.2 of C4 provides a list of the objects used on the left hand side of the assignment statements (display template D13) and their properties. Activation of 4.1.3 of C4 provides a list of the objects used on the right hand side of the assignment statements (display template D13) and their properties. Activation of 4.1.4 of C4 provides a list of the variables used on the left hand side of the assignment expression (display template D14) and their properties. Activation of 4.1.5 of C4 provides a list of the variables used on the right hand side of the assignment expression (display template D14) and their properties. The distinction of variables and objects on the left hand side (LHS) and the right hand side (RHS) of the assignment statement is made because LHS variables and objects are updated, whereas RHS variables and objects may update the LHS variables and objects.

Activation of 4.2.1 of C4 provides a list of the objects which are declared in a method (display template D13) and their properties. Activation of 4.2.2 of C4 provides a list of the variables which are declared in a method (display template D14) and their properties. Activation of 4.2.3 of C4 provides a list of the logical blocks, as running serial numbers with their starting and ending line numbers in the source file (display template D15). Activation of 4.2.4 of C4 provides a list of the abstract data types which are declared locally in the method (display template D10).

Activation of 4.3.1 of C4 provides the parametric details of the method of which a method is a specialization (display template D20).

The concept template C4 (Figure 5.17) for `main(...)` in the source file `glay.cc` is generated by choosing *main* from the D9 display template (Figure 5.15). The information display templates D10 and D20 will not be generated, as the method does not contain the necessary declarations for generating these templates. The information grouped in this concept template is

displayed by generating the display templates D12 (activated from 4.1.1), D13 (activated from 4.1.2, 4.1.3, and 4.2.1), D14 (activated from 4.1.4 and 4.2.2), and D15 (activated from 4.2.3) as shown in Figure 5.18 through 5.24.

		Access Paths D9, D12, D17, D22, D23, D24, D25, D26
4. Methods		
4.1 USES	4.2 HAS	4.3 ISA
4.1.1 Methods D 12	4.2.1 Objects (Local) D 13	4.3.1 Parameterization D 20
4.1.2 Objects LHS D 13	4.2.2 Variables (Local) D 14	
4.1.3 Objects RHS D 13	4.2.3 Logical Block D 15	
4.1.4 Variables LHS D 14	4.2.4 Abstract Data Type D 10	
4.1.5 Variables RHS D 14		

Figure 5.17

Method Concept Template C4

Method [main] USES

Method Names	Type	Return Type	Return Type Pointer	Parameter List	Constant	Name of the file where Defined
getopt	R	int	N	argc,argv, "l:x::y:i:o:e :d:"	N	Library File
size	Ocmg.nodes	int	N	------	N	graph.c
size	Ocmg.nodes	int	N	------	N	graph.c
sortNodes	O cmg	----	N	np, nListSz	N	graph.c

Figure 5.18

D12 Display Template (from C4 (4.1.1))

Method [main] USES

Object Names	Type	Storage Type	Pointer	Constant	Array	Scope
g	Graph	A	Y	N	N	113

Figure 5.19

D13 Display Template (from C4 (4.1.2))

Method [main] USES

Object Names	Type	Storage Type	Pointer	Constant	Array	Scope
cmg	Graph	A	N	N	N	33

Figure 5.20

D13 Display Template (from C4 (4.1.3))

Method [main] USES

Variable Names	Type	Storage Type	Pointer	Constant	Array	Scope
xFlg	int	A	N	N	N	28
yFlg	int	A	N	N	N	28
deposFlg	int	A	N	N	N	29
extrFlg	int	A	N	N	N	29
localinFlg	int	A	N	N	N	29
localoutFlg	int	A	N	N	N	30
outf	char	A	Y	N	N	26
inf	char	A	Y	N	N	26
depFile	char	A	Y	N	N	26
extFile	char	A	Y	N	N	26
nLisdtSz	int	A	N	N	nListSz	47
eListSz	int	A	N	N	eListSz	48

Figure 5.21

D14 Display Template (from C4 (4.1.4))

Method [main] HAS

Object Names	Type	Storage Type	Pointer	Constant	Array	Scope
cmg	Graph	A	N	N	N	33
r	Repository	A	Y	N	N	95
g	Graph	A	Y	N	N	113
np	Node	A	N	N	nListSz	49
snp	Node	A	N	N	nListSz	49
ep	Edge	A	N	N	eListSz	50

Figure 5.22

D13 Display Template (from C4 (4.2.1))

Method [main] HAS

Variable Names	Type	Storage Type	Pointer	Constant	Array	Scope
xFlg	int	A	N	N	N	28
yFlg	int	A	N	N	N	28
deposFlg	int	A	N	N	N	29
extrFlg	int	A	N	N	N	29
localinFlg	int	A	N	N	N	29
localoutFlg	int	A	N	N	N	30
outf	char	A	Y	N	N	26
inf	char	A	Y	N	N	26
depFile	char	A	Y	N	N	26
extFile	char	A	Y	N	N	26
nLisdtSz	int	A	N	N	nListSz	47
eListSz	int	A	N	N	eListSz	48
aflg	int	A	N	N	N	25
lclv	char	A	Y	N	N	26
inFile	fstream	A	N	N	N	21

Figure 5.23

D14 Display Template (from C4 (4.2.2))

Method [main] HAS

Logical Block#	Type	Argument List	Starting Line #	End Line#	Nesting Level
1	while	c=getopt(argc,argv,"l:x:y:o:e:d:" != -1	36	70	0
2	if	localinFlg == YES	71	80	0
3	if	localoutFlg == YES	81	92	0
4	if	deposFlg || extrFlg	84	94	0
5	if	extrFlg == YES && localinFlg == YES	95	100	0
6	if	localinFlg == YES	102	106	0
7	if	extrFlg == YES	108	112	0
8	if	extrFlg != YES && localinFlg == YES	114	117	0
9.	if	!xFlg || yFlg	119	122	0
10	for	int i=0; i< nListSz; ++i	133	136	0
11	for	i=0; i< eListSz; ++i	138	141	0
12	if - else	aflg == YES : ----	144	150	0
13	for	i=0; i < eListSz; ++i	152	156	0
14	for	i=0; i < nListSz; ++i	158	162	0
15	if	localoutFlg !=YES && deposFlg!=YES	160	164	0
16	if	deposFlg == YES	166	169	0
17	if	localoutFlg == YES	171	174	0

Figure 5.24

D15 Display Template (from C4 (4.2.3))

5.3.6 Logical Block Concept Template C5

The logical block concept template (C5) groups information related to the logical block concept of a C++ program. The template is generated by choosing a logical block from the information display template D15 activated either from 4.2.3 of C4 or 5.2.3 of C5. The

information about other C++ concepts contained within the logical block concept is displayed either directly by other information display templates, or subsequent to generating the concept templates therein. For example, the details of a local object (a C++ key concept) within the logical block are displayed directly by display template D13, whereas the details of the method (a C++ key concept) called from within the logical block will be displayed after generating the method concept template C4 from display template D12.

Activation of 5.1.1 of C5 provides a list of the methods used in this method and their attributes, and the file in which they are defined (display template D12). Activation of 5.1.2 of C5 provides a list of the objects used on the left hand side of the assignment statements (display template D13) and their properties. Activation of 5.1.3 of C5 provides a list of the objects used on the right hand side of the assignment statements (display template D13) and their properties. Activation of 5.1.4 of C5 provides a list of the variables used on the left hand side of the assignment statements (display template D14) and their properties. Activation of 5.1.5 of C5 provides a list of the variables used on the right hand side of the assignment statements (display template D14) and their properties.

Activation of 5.2.1 of C5 provides a list of the objects which are declared within the logical block (display template D13) and their properties. Activation of 5.2.2 of C5 provides a list of variables which are declared in this block (display template D14) and their properties. Activation of 5.2.3 of C5 provides a list of the logical blocks as running serial numbers and their starting and ending line numbers in the source file (display template D15). Activation of 5.2.4 of C5 provides a list of the abstract data types which are declared locally in the logical block (display template D10).

The concept template C5 for logical block concept (Figure 5.25) for

```
•   ( for (int i=0; i < nListSz; ++i)
```

in the method `main(...)` of the source file `glay.cc` is generated by choosing *10* from the D15 display template (Figure 5.24). Only the information display template D13 (Figure 5.26 through 5.27) is generated from 5.1.2 and 5.1.3 of concept template C5 (Figure 5.25), as the logical block does not contain the declaration needed for generating other display templates.

		Access Paths **D 15**
5. Logical Block		
5.1 USES	**5.2 HAS**	**5.3 ISA**
5.1.1 Methods **D 12**	**5.2.1 Objects (Local)** **D 13**	**NONE**
5.1.2 Objects LHS **D 13**	**5.2.2 Variables (Local)** **D 14**	
5.1.3 Objects RHS **D 13**	**5.2.3 Logical Block** **D 15**	
5.1.4 Variables LHS **D 14**	**5.2.4Abstract Data Type** **D 10**	
5.1.5 Variables RHS **D 14**		

Figure 5.25

Concept Template C5

Logical Block [10] USES						
Object Names	Type	Storage Type	Pointer	Constant	Array	Scope
np	Node	A	N	N	i	49

Figure 5.26

D13 Display Template (from C5 (5.1.2))

Logical Block [10] USES						
Object Names	Type	Storage Type	Pointer	Constant	Array	Scope
cmg.nodes	Graph	A	N	N	i	43

Figure 5.27

D13 Display Template (from C5 (5.1.3))

5.3.7 Abstract Data Type Template Concept Template C6

The abstract data type template concept template (C6) groups information related to the data abstraction concept of a C++ program. The template is generated by choosing an abstract data type name from display template D10 generated from one of the following concept templates; 3.2.5 (C3), 4.2.4 (C4), 5.3.4 (C5), or 6.2.1 (C6), and the information display template D6 generated from 6.1.2 (C6). Information about the other C++ concepts contained within the abstract data type template concept is displayed either directly by other information display templates, or subsequent to generating the concept templates therein. For example, the details

of a member object (a C++ key concept) within the abstract data type will be displayed directly by the display template D16, whereas the details of the *friend* abstract data type template will be displayed after generating another abstract data type template concept template (C6) from display template D6.

Activation of 6.1.1 of C6 provides a list of the methods used in the abstract data type (display template D9). Activation of 6.1.2 provides a list of the abstract data type templates declared as a *friend* to the abstract data type template (display template D6).

Activation of 6.2.1 of C6 provides a list of the abstract data types defined (nesting) within the abstract data type (display template D10). Activation of 6.2.2 of C6 provides a list of the member objects and their details (display template D16). Activation of 6.2.3 of C6 provides a list of the member variables and their details (display template D21). Activation of 6.2.4 of C6 provides a list of the member methods and their details (display template D17).

Activation of 6.3.1 of C6 will provide the inheritance details of the abstract data type (display template D18). Activation of 6.3.2 of C6 will provide the details about parametric abstract data type (display template 19).

For the chosen set in the example, the concept template C6 (Figure 5.28) for `graph` abstract data type template is generated from the display template D13 (Figure 5.19, 5.20, 5.22, 5.26, 5.27). These templates in turn are generated from either one of the following: C4 (4.1.2), C4 (4.1.3), C4 (4.2.1), C5(5.1.3), and the display template D6 (Figure 5.40) generated from C7 (7.3.1). The information is displayed by generating the display templates D6 (activated from 6.1.2), D16 (activated from 6.2.2), D21 (activated from 6.2.3), D17 (activated from 6.2.4), D18 (activated from 6.3.1). as shown below (Figure 5.29 through 5.33). The display templates D9, D10, and D19 will not be generated because of the absence of the necessary declaration in the

abstract data type template.

Access Paths D6, D10, D13		
6. Abstract Data Type Template		
6.1 USES	**6.2 HAS**	**6.3 ISA**
6.1.1 Method **D9**	**6.2.1 Abstract Data Type** **D10**	**6.3.1 Inheritance** **D18**
6.1.2 Abstract Data Type **D6**	**6.2.2 Members Objects** **D16**	**6.3.2 Parameteric Type** **D19**
	6.2.3 Member Variables **D21**	
	6.2.4 Member Methods **D17**	

Figure 5.28

Abstract Data Type Template Concept Template C6

Abstract Data Type Template [Graph] USES

Abstract Data Type Names	Type	Declared in File
RObject	class	IML.h

Figure 5.29

D6 (from C6 (6.1.2))

Abstract Data Type Template [Graph] HAS						
Member Object Names	Type	Storage Type	Pointer	Constant	Array	Access
edges	EdgeList	A	N	N	N	P
nodes	NodeList	A	N	N	N	P

Figure 5.30

D16 (from C6 (6.2.2))

Abstract Data Type Template [Graph] HAS						
Member Variable Names	Type	Storage Type	Pointer	Constant	Array	Access
gtype	int	A	N	N	N	PUB

Figure 5.31

D22 (from C6 (6.2.3))

Abstract Data Type Template [Graph] HAS

Member Method Names	Return Type	Return Type Pointer	Parameter List	Constant	Virtual	Inline	Access
layout	int	N	Node *, int nListSz, int grdiX, int gridY	N	N	N	PUB
read	void	N	istream & ins	N	N	N	PUB
write	void	N	ostream& outs	N	N	N	PUB
sortNodes	int	N	Node *np, int nListSz	N	N	N	PUB
read_frm_stream	int	N	istream &ins	N	N	N	PUB
write_frm_stream	int	N	ostream &outs	N	N	N	PUB

Figure 5.32

D17 (from C6 (6.2.4))

Abstract Data Type Template [Graph] ISA

Inherited Abstract Data Type Names	Virtual	Access
IO_type	Y	PUB
RObject	Y	PUB

Figure 5.33

D18 (from C6 (6.3.1))

5.3.8 Object Concept Template C7

The object concept template (C7) groups information related to the object concept of a C++ program. The concept template is generated by choosing an object from one of the following information display templates: D4, D7 activated from 3.1.3 or 3.2.1 (C3), D13 activated from 4.1.2, 4.1.3, 4.2.1 of C4, and 5.1.2. 5.1.3, 5.2.1 of C5, and from 6.2.2 of abstract data type concept template C6. The information about other C++ concepts contained within the object concept will be displayed either directly by other information display templates, or subsequent to generating the concept templates therein. For example, the details of a member object (a C++ key concept) within the object will be displayed directly by the display template D16, whereas the details about its abstract data type instance is displayed after generating another abstract data type template concept template (C6) from display template D6.

Activation of 7.1.1 of C7 provides the details about the use of its own member method (display template D22). Activation of 7.1.2 of C7 provides the details about another object's member method used by the object on the right hand side of the assignment statements (display template D23). Activation of 7.1.3 of C7 provides the list of methods used by the object on the right hand side of the assignment statements (display template D24). Activation of 7.1.4 of C7 provides a list and details of the other objects used by the object on the right hand side of the assignment statements (display template D25). Activation of 7.1.5 of C7 provides a list and details of the variables used by the objects on the right hand side of the assignment statements (display template D26).

Activation of 7.2.1 of C7 provides a list and details of member objects which the object has (display template D16). Activation of 7.2.1 provides a list and details of member variables which the object has (display template D21).

Activation of 7.3.1 of C7 provides the instance type of the object (display template D6).

The concept template C7 (Figure 5.34) for the object cmg is generated from D13 (Figure 5.20, 5.22) and D12 (Figure 5.18). Further information grouped in the concept template C7 is generated by display templates D22 (activated from 7.1.1), D23 (activated from 7.1.2), D25 (activated from 7.1.4), D16 (activated from 7.2.1), D21 (activated from 7.2.2), and D6 (activated from 7.3.1), as shown below (Figure 35 through 5.40). The display templates D24 and D26 are not generated because of the absence of the necessary declarations in the code.

		Access Paths D4, D7, D13
7. Object		
7.1 USES	7.2 HAS	7.3 ISA
7.1.1 Own Member D22 Method	7.2.1 Member Objects D16	7.3.1 Abstract Data Type D6 Instance
7.1.2 Other Objects's D23 Member Method RHS	7.2.2 Member Variables D21	
7.1.3 Method RHS D24		
7.1.4 Object RHS D25		
7.1.5 Variable RHS D26		

Figure 5.34

Object Concept Template C7

Object [cmg] USES

Member Method Names	Source File Name	Used In Methods	Line #s	Nesting Level
sortnodes	glay.cc	main	164	0
layout	glay.cc	main	166	1

Figure 5.35

D22 Display Template (from C7 (7.1.1))

Object [cmg] USES

Object Names	Member Method Names	Source File Name	Method Name	Line #	Nesting Level
nodes	size	glay.cc	main	148	0
edges	size	glay.cc	main	149	0

Figure 5.36

D23 Display Template (from C7 (7.1.2))

Object [cmg] USES (RHS)

Object Names	Source File Name	Used In Methods	Line #	Nesting Level
ep	glay.cc	main	184	1
np	glay.cc	main	190	1

Figure 5.37

D25 (from C7 (7.1.4))

Object [cmg] HAS

Member Object Names	Type	Storage Type	Pointer	Constant	Array	Access
edges	EdgeList	A	N	N	N	P
nodes	NodeList	A	N	N	N	P

Figure 5.38

D16 Display Template (from C7 (7.2.1)

Object [cmg] HAS

Member Variable Names	Type	Storage Type	Pointer	Constant	Array	Access
gtype	int	A	N	N	N	PUB

Figure 5.39

D21 Display Template (from C7 (7.2.2))

Object [cmg] ISA

Abstract Data Type Names	Type	Declared in File
Graph	class	graph.h

Figure 5.40

D6 Display Template (from C7 (7.3.1))

5.4 Display Templates Description

This section present the details of all the display templates generated from concept templates such as concept templates from which these display templates are generated and the information these display templates contain. When needed, an example is provided to make the presentation clear.

5.4.1 Object Files Display Template (D1)

The object files template (D1) is generated (Figure 5.41) only from the USES relationship of the executable file concept template C1 (1.1.1 of the C1).

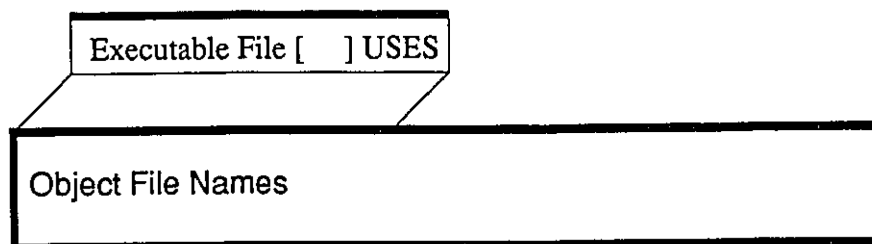


Figure 5.41

Display Template for Object Files (D1)

The display template D1 will display the following information:

- Object File Names: The names of the object files are listed in the order in which they appear in the makefile.

5.4.2 Source Files Display Template (D2)

The source files display template (D2) is generated from the USES relationship of the object file concept templates C2 (2.1.1 of the C2). From the first column of this template, concept template C2 is generated for another C++ source file (Figure 5.42).

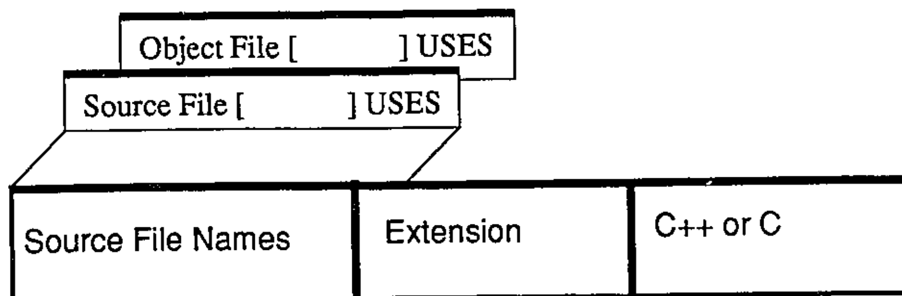


Figure 5.42

Information Display Template for User Files (D2)

The display template D2 displays the following information:

- Source File Names: The source file names are listed in the first column, in the order in which they appear in the makefile.
- Extensions: The extensions of the source files are shown in the second column, e.g., for myfile.h, "h" is the extension of the file.

- C++ or C: This column shows the entry as either C++ or C, depending on whether the source file is written in C++ or C. Files with the extension 'C' generate neither concept templates nor display templates.

5.4.3 Library Files Display Template (D3)

The library files template (D3) is generated only from the USES relationship of the source file concept template (3.1.1 of the C3). No other concept template is generated from this display template (Figure 5.43).

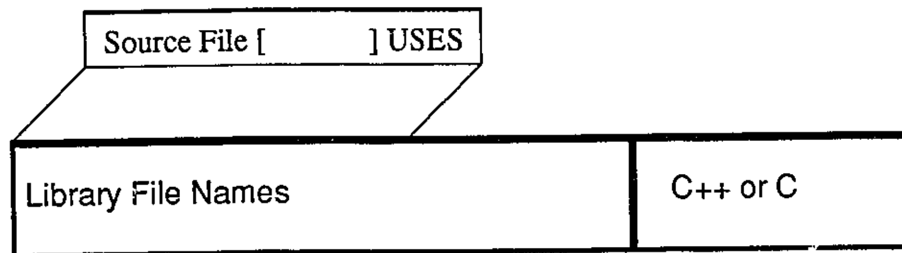


Figure 5.43

Display Template for Library Files (D3)

The display template D3 displays the following information:

- Library File Names: The first column shows a list of the library file names in the order in which they appear in the program.
- C++ or C: This column shows the entry as either C++ or C, depending on whether the library file belongs to a C++ or C library.

5.4.4 External Objects Display Template (D4)

The external objects template (D4) is generated only from the USES relationship of the source file concept template (3.1.3 of the C3). From the sixth column of this display template, the source file concept template C3 is generated, provided that the source file is a C++ source file. And from the first and second columns of the template, concept templates C7 and C6 can be generated, respectively (Figure 5.44).

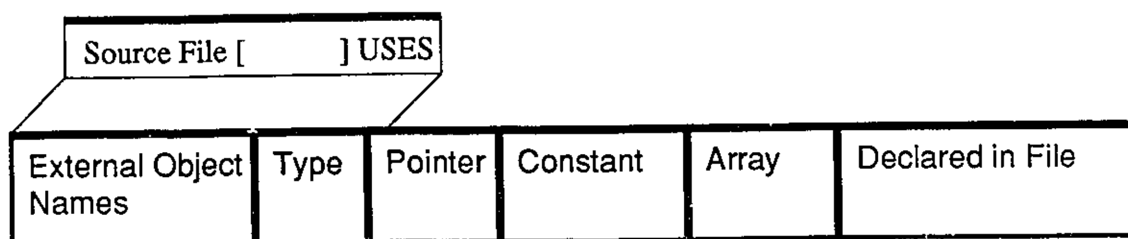


Figure 5.44

Display Template for External Objects (D4)

The display template D4 displays the following information:

- External Object Names: The first column shows a list of the external object names in the order in which they are declared in the file.

- Type: The second column shows the type of the object, e.g., for

```
extern EdgeList edges;
```

  "EdgeList" appears in the second column.

- Pointer: The third column shows the entry as either "Y" (yes) or "N" (no) for the pointer.

- Constant: The fourth column shows the entry as either "Y" (yes) or "N" (no) for the constant.

- Array: The fifth column show the entry as either the array size or "N" (no), e.g., for

```
extern EdgeList edges[20][20];
```

  the entry is " 20,20 ".

- Declared in file: The sixth column shows the entry for the file name in which the object has been declared.

5.4.5 External Variable Display Template (D5)

The external variable template (D5) is generated only from the USES relationship of the source file concept template (3.1.4 of the C3). From the last column of the template, the source file concept template C3 for another source file is generated (Figure 5.45).

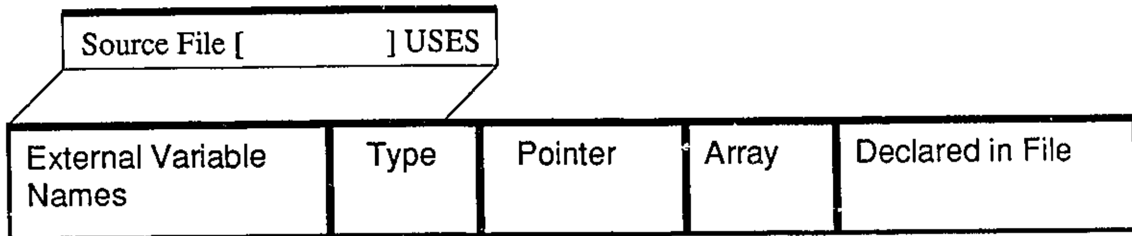


Figure 5.45

Display Template for External Variables (D5)

The display template D5 display the following information:

- External Variable Names: The first column shows a list of the external variable names in the order in which they are declared in the file.

- Type: The second column shows the type of the variable, e.g. for

```
extern int edges;
```

  D6"int" appears in the second column.

- Pointer: The third column shows the entry as either "Y" (yes) or "N" (no) for the pointer.
- Array: The fifth column shows the entry as either the array size or "N" (no), e.g. for

```
extern int edges[20][20];
```

  the entry is " 20,20 ".
- Declared in file: The sixth column shows the entry for the name of the file in which the variable is declared.

5.4.6 Abstract Data Type (a) Display Template (D6)

The abstract data type (a) template (D6) is generated from the USES relationship of the abstract data type concept template (6.1.2 of the C6) and the ISA relationship of the object concept template (7.3.1 of the C7). From the first and the last columns of the template, the abstract data type concept template C6 and the source file concept template C3 for the source file in which the abstract data type is declared are generated, respectively (Figure 5.46).

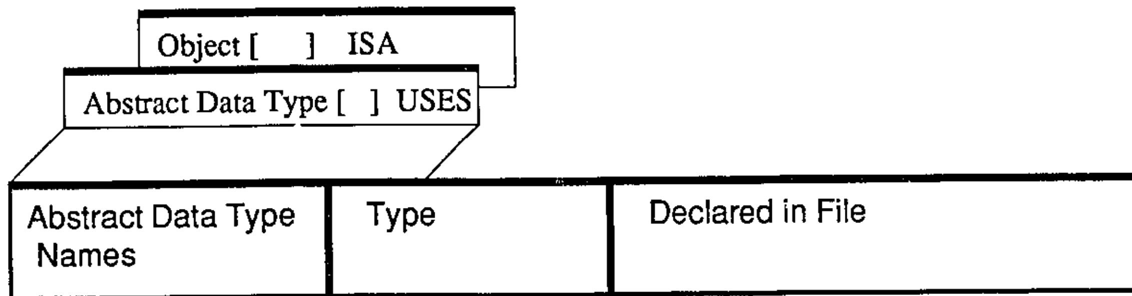


Figure 5.46

Display Template for Abstract Data Type (a) (D6)

The display template D6 displays the following information:

- Abstract Data Type Names: The first column shows a list of the abstract data type templates in the order in which they appear in the software code.

- Type: The second column shows the *type* of the abstract data type template, i.e., whether it is class, struct, or union.
- Declared in file: The third column shows the name of the file in which the abstract data type template is defined.

5.4.7 Global Objects Display Template (D7)

The global objects template (D7) is generated only from the HAS relationship of the source file concept template (3.2.1 of the C3). From the second column of the template, the concept template C6 for the abstract data type template is generated (Figure 5.47).

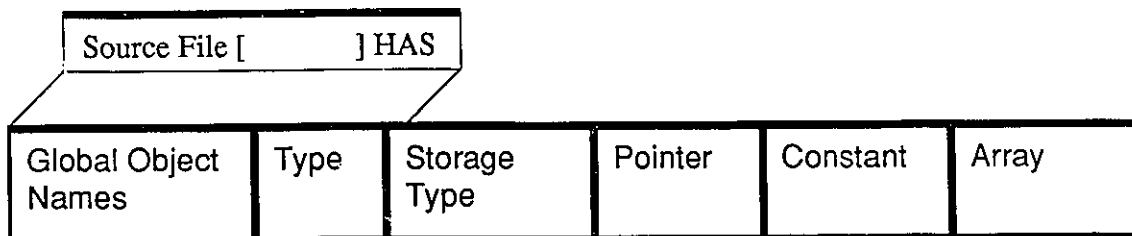


Figure 5.47

Display Template for Global Objects (D7)

The display template D7 display the following information:

- Global Object Names: The first column shows a list of the object names in the order in which they are declared in the source file.
- Type: The second column shows the instance type of the object, e.g. for

```
EdgeList edges;
```

  "EdgeList" appears in the second column.
- Storage Type: The third column shows the entry for the storage type, e.g., S for static, V for volatile.

- Pointer: The fourth column shows the entry as either "Y" (yes) or "N" (no) for the pointer.
- Constant: The fifth column shows the entry as either "Y" (yes) or "N" (no) for the constant.
- Array: The sixth column show the entry as either the array size or "N" (no), e.g. for

```
EdgeList edges[20][20];
```

the entry is " 20,20 ".

5.4.8 Global Variables Display Template (D8)

The global variables template (D8) is generated from the HAS relationship of the source file concept template (3.2.2 of the C3). No concept template is generated from this template (Figure 5.48).

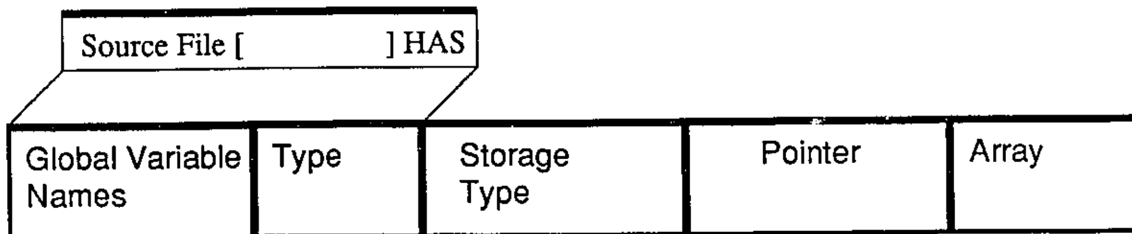


Figure 5.48

Display Template for Global Variables (D8)

The display template D8 displays the following information:

- Variable Names: The first column shows a list of the variable names in the order in which they are declared in the file.

- Type: The second column shows the type of the variable, e.g. for
```
int edges;
```
"int" appears in the second column.

- Storage Type: The third column shows the entry for the storage type, e.g., S for static, V for volatile.

- Pointer: The fourth column shows the entry as either "Y" (yes) or "N" (no) for the pointer.

- Array: The fifth column shows the entry as either the array size, the variable name which sets the array size, or "N" (no), e.g., for
```
extern int edges[20][20];
```
the entry is " 20,20 ".

5.4.9 Methods (a) Display Template (D9)

The methods (a) template D9 is generated from the HAS relationship of the source file concept template (3.2.3 of the C3) and the USES relationship of the abstract data type concept template (6.1.1 of the C6). The concept template C4 is generated from the first column of this display template. If the second column of the template has either the "A" or "O" prefix (see example below), then the concept template C6 for the abstract data type template concept and the concept template C7 for the object concept are generated, respectively (Figure 5.49).

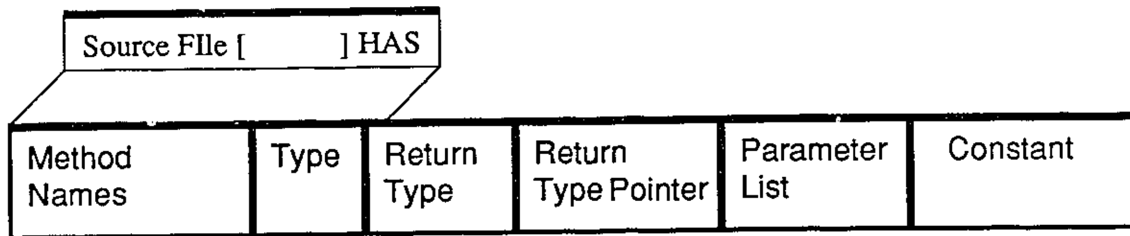


Figure 5.49

Display Template for Methods (a) (D9)

The display template D9 displays the following information:

- Method Names: The first column shows a list of method names in the order in which they are written in the source file.

- Type: The second column shows one of the following:

  "R" for a regular method

  "A-(Abstract data type name)" for a member method

  "O-(Object name) for a member method of the abstract data type defined without a

tag name, e.g.,

```
class
{
  void myfunc();
}myobject;
```

- Return Type: The third column shows the return type of the method; e.g., for

```
SS myfunc( );
```

  the entry "SS" appears.

- Return Type Pointer: The fourth column shows the entry as either "Y" (yes) or "N" (no) for the pointer.

- Parameter List: The fifth column shows a parameter list of the method (if applicable), e.g. for

```
void myfunc(obj1,int num,Obj obj2);
```

  the entry "obj1 , int num , Obj obj2" appears.

- Constant: The fifth column shows the entry as either "Y" (yes) or "N" (no) for the constant, e.g., for

```
void myfunc(obj1,int num,Obj obj2) const;
```

  the entry "Y" appears.

5.4.10 Abstract Data Type (b) Display Template (D10)

The abstract data type template (b) D10 is generated from the HAS relationship of the source file concept template (3.2.5 of the C3) and the abstract data type concept template (6.2.1 of the C6). From the first column of the template, the concept template C6 for the abstract data type template is generated (Figure 5.50).

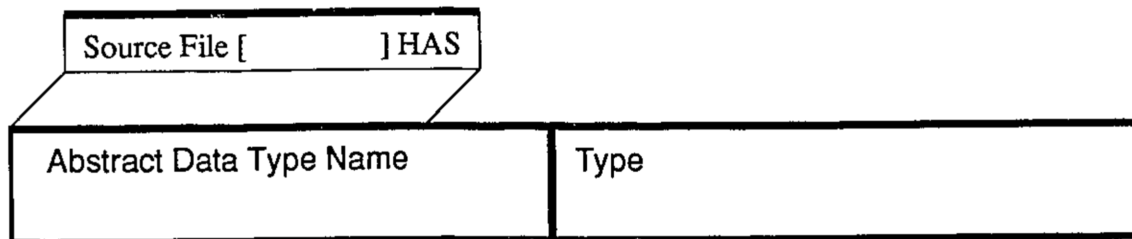


Figure 5.50

Display Template for Abstract Data Type (b) (D10)

The display template D10 display the following information:

- Abstract Data Type Names: The first column lists the abstract data type names.

- Type: The second column gives the type of the abstract data type; i.e., whether it is class, struct, or union.

5.4.11 #define Directive Display Template (D11)

The #define directive template (D11) is generated from the HAS relationship of the source file concept template (3.2.4 of the C3). No other concept template is generated from this template (Figure 5.51).

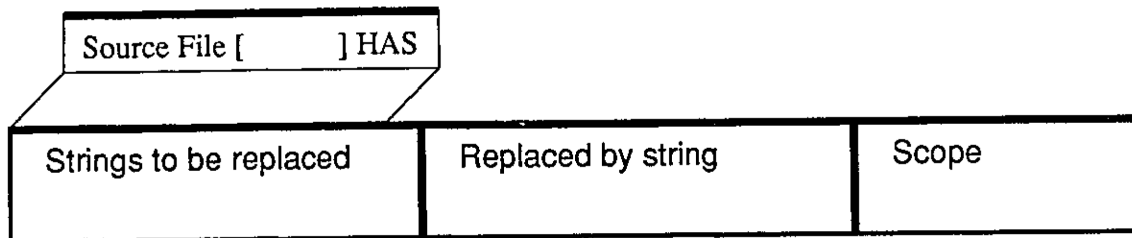


Figure 5.51

Display Template for #define Directive (D11)

The display template D11 displays the following information:

- Strings To Be Replaced: The first column shows the string to be replaced, e.g., for

  ```
  #define MAX 100
  ```

  the entry "MAX" appears.

- Replaced By String: The second column shows the string which replaces the string in first column, e.g., for

  ```
  #define MAX 100
  ```

  the entry "100" appears.

- Scope: The third column shows the entry as either "G" for global replacement, or as the "line #" below which the macro expands.

5.4.12 D12 Methods (b) Display Template (D12)

The methods (b) template (D12) is generated from the USES relationship of the method concept template (4.1.1 of the C4). From the first column of the template, the concept template C4 for another method is generated. If the method is a *member method*, then the concept template C6 is generated. The abstract data type concept template C6 is generated from the third column if the return type is an abstract data type template. From the last column of the template, the concept template C3 is generated for the source file, where the method is defined (Figure 5.52).

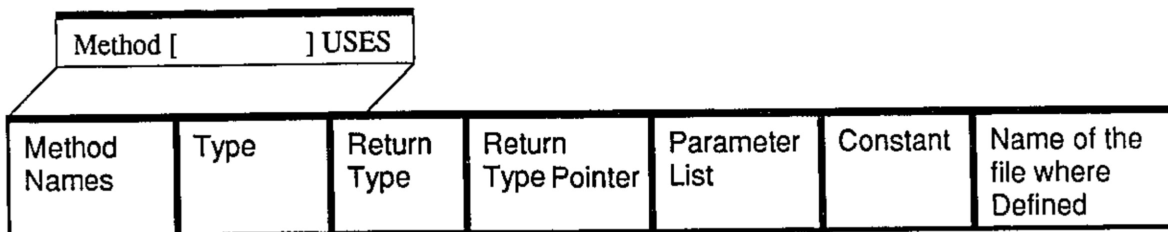


Figure 5.52

Display Template for Methods (b) (D12)

The display template D12 displays the following information:

- Method Names: The first column shows a list of the names of the methods in the order in which they are called in the method.
- Type: The second column shows one of the following entries:

  "R" for a regular method

"O-(Object name) for the member method of the object, e.g., for

```
void myfunc()
{
  edges.count();
}
/* object "edges" defined outside
 the function */
```

the entry "O-edges" appears.

- Return Type: The third column shows the return type of the method, e.g., for

  ```
  int myfunc( );
  ```

  the entry "int" appears.

- Return Type Pointer: The fourth column shows the entry as either "Y" (yes) or "N" (no) for the pointer.

- Parameter List: The fifth column shows the parameter list of the method, e.g., for

  ```
  myfunc(obj1,int num,Obj obj2);
  ```

  the entry "obj1 , int num , Obj obj2" appears.

- Constant: The sixth column shows the entry as either "Y" (yes) or "N" (no) for the constant, e.g., for

  ```
  void myfunc(obj1,int num,Obj obj2) const;
  ```

  the entry "Y" appears.

- Declared in file: The seventh column shows the file names in which the abstract data type templates are defined.

5.4.13 Objects (LHS/RHS) Display Template (D13)

The display template (D13) for objects in left hand side and right hand side of an expression (LHS/RHS) is generated from the USES relationship of the method and logical block concept template (4.1.2 and 4.1.3 of the C4 and 5.1.2 and 5.1.3 of the C5). From the first and second columns of the display template, the object concept template C7 and the abstract data type concept template C6 are generated, respectively (Figure 5.53).

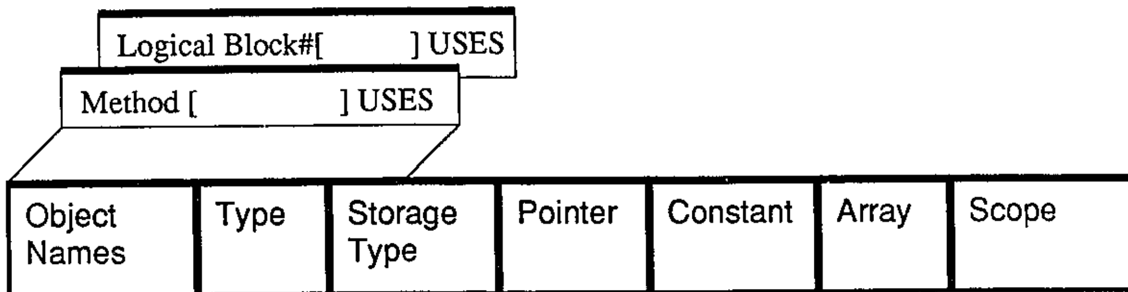


Figure 5.53

Display Template for Objects (LHS/RHS) (D13)

The display template D13 display the following information:

- Object Names: The first column shows a list of the object names in the order in which they are used in the method.
- Type: The second column shows the instance type of the object, e.g., for
  ```
  EdgeList edges;
  ```
  the entry "EdgeList" appears.
- Storage Type: The third column shows the entry for the storage type, e.g., S for static, V for volatile.

- Pointer: The fourth column shows the entry as either "Y" (yes) or "N" (no) for the pointer.

- Constant: The fifth column shows the entry as either "Y" (yes) or "N" (no) for the constant.

- Array: The sixth column shows the entry as either the array size or "N" (no), e.g., for

  ```
  EdgeList edges[20][20];
  ```

  the entry is " 20,20 ".

- Scope: The seventh column shows the entry as either "G" for global, or the "line #" beyond which the object is visible.

5.4.14 Variables (LHS/RHS) Display Template (D14)

The variables (LHS/RHS) display template (D14) is generated from the USES relationship of the method and logical block concept template (4.1.4 and 4.1.5 of the C4, and 5.1.4 and 5.1.5 of the C5). No other concept template is generated from this template (Figure 5.54).

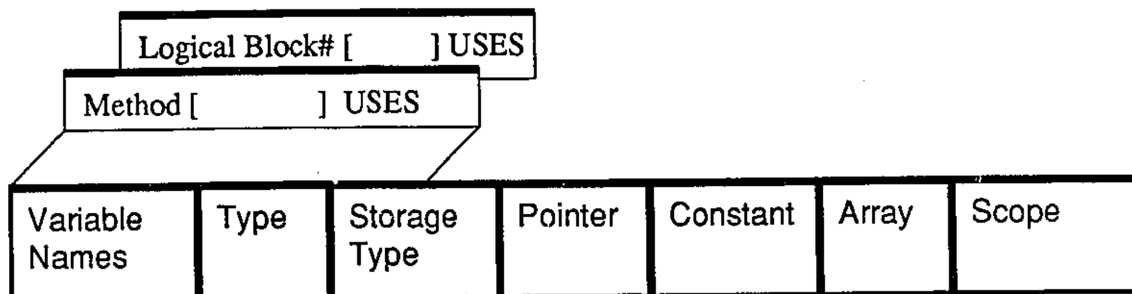


Figure 5.54

Display Template for Variables (LHS/RHS) (D14)

The display template D14 displays the following information:

- Variable Names: The first column shows a list of the variable names in the order in which they are used in the method.
- Type: The second column shows the *type* of the variable, e.g., for

  ```
  int edges;
  ```

  the entry "int" appears.
- Storage Type: The third column shows the entry for the storage type, e.g., S for static, V for volatile.
- Pointer: The fourth column shows the entry as either "Y" (yes) or "N" (no) for the pointer.
- Constant: The fifth column shows the entry as either "Y" (yes) or "N" (no) for the constant.
- Array: The sixth column show the entry as either the array size or "N" (no), e.g., for

  ```
  int edges[20][20];
  ```

  the entry will be " 20,20 ".
- Scope: The seventh column shows the entry as either "G" for global, or as the "line #" beyond which the variable is visible.

5.4.15 Logical Block Display Template (D15)

The logical block display template (D15) is generated from the HAS relationship of the method and the logical block concept templates (4.2.3 of the C4 and 5.2.3 of the C5). From first column of this template, the concept template C5 is generated for another logical block (Figure 5.55).

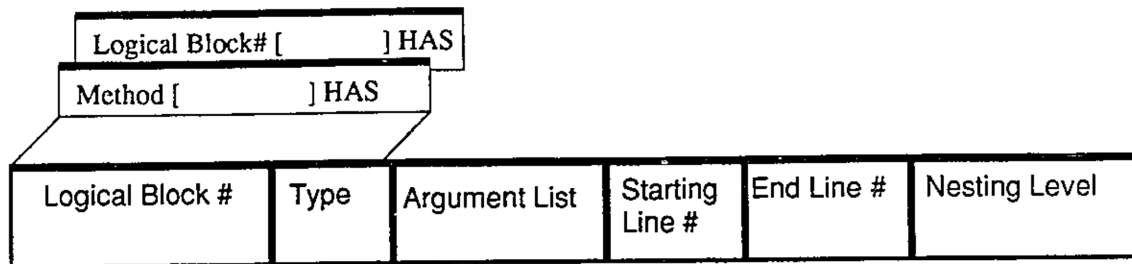


Figure 5.55

Display Template for Logical Block (D15)

The display template D15 displays the following information:

- Logical Block #: The first column shows the running serial numbers of the selection or repetitive block in the order in which they appear in the source file.
- Type: The second column gives the type of the logical block, e.g., "for", "while", or "{ }" if the block is formed using "{ }".
- Argument List: The third column shows the argument list of the logical block, e.g., for

```
void myfunc()
{
  int i;

  for (i=0;i<N;i++)
    {
      ...
    }
}
```

the third column shows the entry as "(i=0;i<N;i++)" or "N" if the block is formed using "{ }" .

- Starting Line #: The fourth column shows the line number in the source file from where the logical block begins.
- End Line#: The fifth column shows the line number in the source file where the scope of the logical block ends.
- Nesting Level: The sixth column provides the nesting level of the logical block, e.g., for

```
void myfunc()
{
  {
     ...
     while (X<10)
       {
         ...
       }
   }
}
```

the entry "2" appears.

5.4.16 Member Object Display Template (D16)

The member object display template (D16) is generated from the HAS relationship of the abstract data type template's concept template (6.2.2 of the C6). From the first and second columns of the template, the object concept template C7 and the abstract data type template concept template C6 are generated, respectively (Figure 5.56).

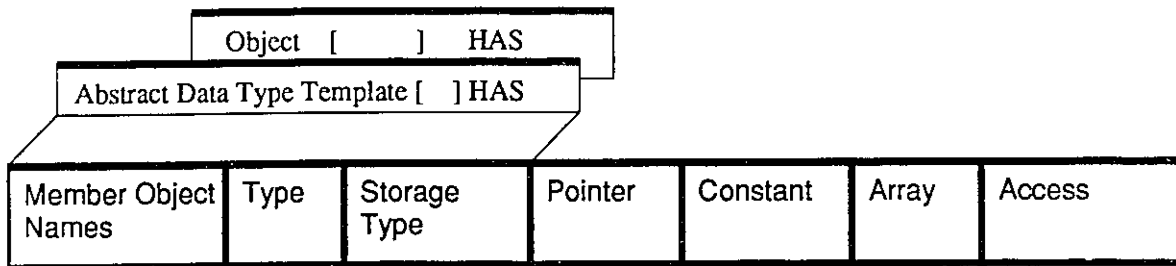


Figure 5.56

Display Template for Member Objects (D16)

The display template D16 displays the following information:

- Member Object Names: The first column shows a list of the object names declared in the abstract data type. They appear in the order in which they are declared in the abstract data type template.
- Type: The second column shows the entry for its type.
- Storage Type: The third column shows one of the following entries:

  "N" for none, the default

  "S" for static

  "V" for volatile
- Pointer: The fourth column shows the entry as either "Y" (yes) or "N" (no) for the pointer.
- Constant: The fifth column show the entry as either as "Y" (yes) or "N" (no) for the constant.

- Array: The sixth column shows the entry as either the array size or "N" (no). e.g., for

```
int edges[20][20];
```

  the entry is " 20,20 ".

- Access: The seventh column shows one of the following entries:

  "PU" if nothing is specified in the code

  "PRI" for private members

  "PRO" for protected members

  "PUB" for public members

5.4.17 Member Method (a) Display Template (D17)

The member method (a) display template (D17) is generated from the HAS relationship of the abstract data type template mode (6.2.4 of the C6). From the first column of this template and from the second column if the return type is an abstract data type, the method concept template C4 and the abstract data type template concept template C6 are generated, respectively (Figure 5.57).

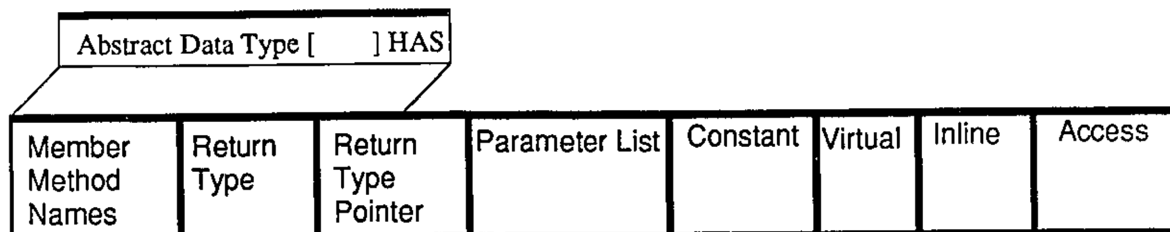


Figure 5.57

Display Template for Member Method (a) (D17)

The display template D17 displays the following information:

- Method Names: The first column shows a list of the method names in the order in which they are called in the method.

- Return Type: The second column shows the return type of the method, e.g., for

```
int myfunc( );
```

  the entry "int" appears.

- Return Type Pointer: The third column shows the entry as either "Y" (yes) or "N" (no) for the pointer.

- Parameter List: The fourth column shows the parameter list of the method, e.g., for

```
myfunc(obj1,int num,Obj obj2);
```

  the entry "obj1 , int num , Obj obj2" appears.

- Constant: The fifth column shows the entry as either "Y" (yes) or "N" (no) for the constant, e.g., for

```
void myfunc(obj1,int num,Obj obj2) const;
```

  the entry "Y" appears.

- Virtual: The sixth column shows the entry as either "Y" (yes) or "N" (no) for the *virtual* specification in the code.

- Inline: The seventh column shows the entry as either "Y" (yes) or "N" (no) for the *inline* specification in the code.

- Access: The seventh column shows one of the following entries:

  "PU" if nothing is specified in the code

  "PRI" for private members

"PRO" for protected members

"PUB" for public members

5.4.18 Abstract Data Type Inheritance Display Template (D18)

The abstract data type template inheritance display template (D18) is generated from the ISA relationship of the abstract data type template's concept template (6.3.1 of the C6). From the first column of this template, the concept template C6 is generated for another abstract data type template (Figure 5.58).

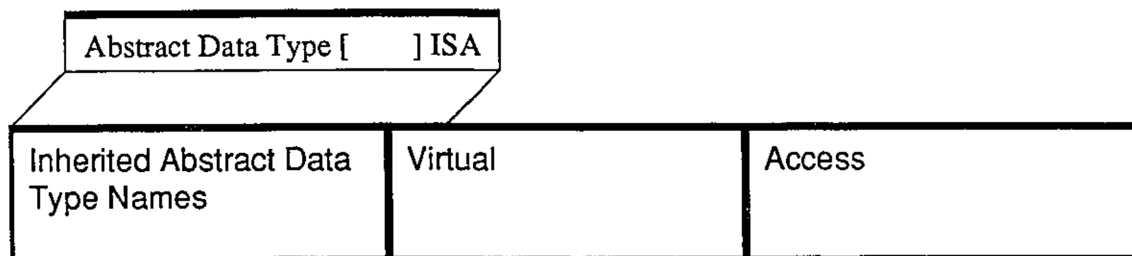


Figure 5.58

Display Template for Abstract Data Type Inheritance (D18)

The display template D18 displays the following information:

- Inherited Abstract Data Type Names: The first column shows a list of the abstract data type template names which are inherited. The names appear in the order in which they are declared in the code.
- Virtual: The second column shows the entry as either "Y" (yes) or "N" (no) for the virtual specification in the code.
- Access: The third column shows one of the following entries:

  "PU" if nothing is specified in the code

"PRI" for private members

"PRO" for protected members

"PUB" for public members

5.4.19 Parametric Abstract Data Type Display Template (D19)

The parametric abstract data type template display template (D19) is generated from the ISA relationship of the abstract data type template concept template (6.3.2 of the C6). No other concept template is generated from this template.

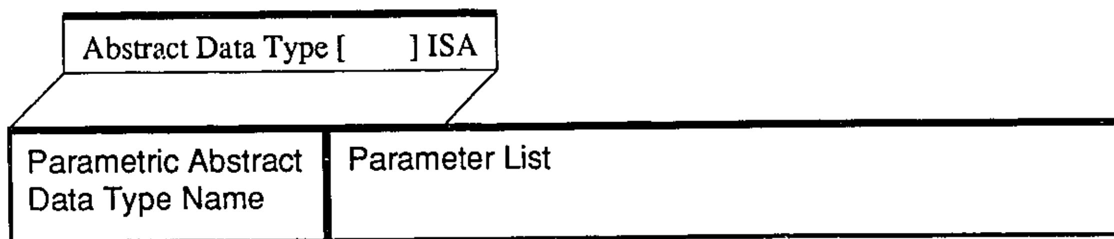


Figure 5.59

Display Template for Parametric Abstract Data Type (D19)

The display template D19 displays the following information:

- Parametric Abstract Data Type Name: The first column shows the name of the parametric abstract data type template.
- Parameter List: The second column shows a list of the parameters used to create the abstract data type template.

5.4.20 Parametric Method Display Template (D20)

The parametric method display template (D20) is generated from the ISA relationship of the method concept template (4.3.1 of the C4). No other concept template is generated from this display template.

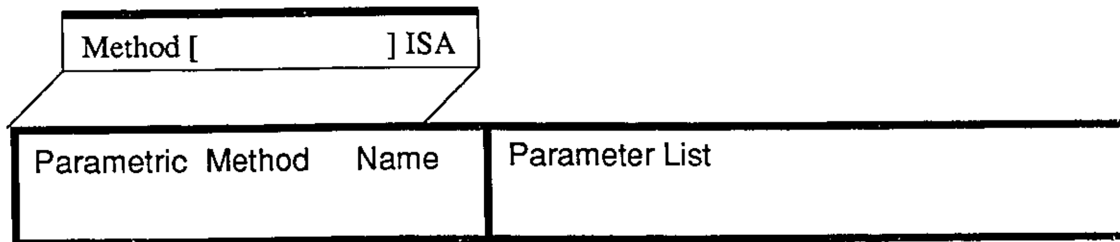


Figure 5.60

Display Template for Parametric Function (D20)

The display template D20 displays the following information:

- Parametric Method Name: The first column shows the name of the parametric method.
- Parameter List: The second column shows the parameters used to create the method.

5.4.21 Member Variables Display Template (D21)

The member variables display template (D21) is generated from the HAS relationship of the abstract data type template concept template (6.2.3 of the C6). No other concept template is generated from this template.

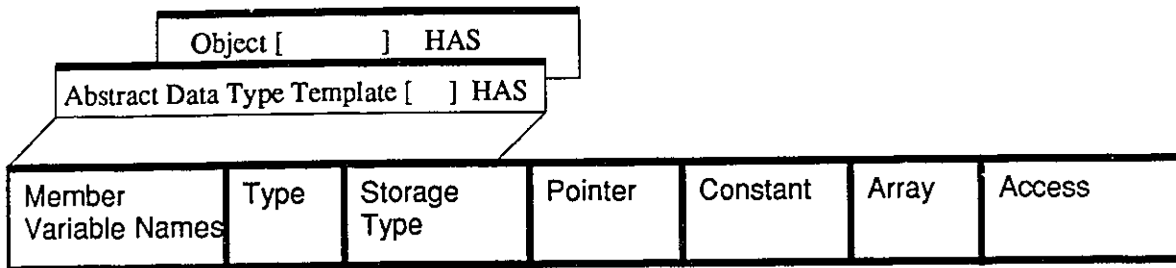


Figure 5.61

Display Template for Member Variables (D21)

The display template D21 displays the following information:

- Member Variable Names: The first column shows a list of the variable names declared in the abstract data type template. They appear in the order in which they appear in the abstract data type declaration.
- Type: The second column shows its type.
- Storage Type: The third column shows one of the following entries:

  "N" for none, the default

  "S" for static

  "V" for volatile
- Pointer: The fourth column shows the entry as either "Y" (yes) or "N" (no) for the pointer.
- Constant: The fifth column shows the entry as either "Y" (yes) or "N" (no) for the constant.

- Array: The sixth column shows the entry as either the array size or "N" (no), e.g., for
```
int edges[20][20];
```
the entry is " 20,20 " .

- Access: The seventh column shows one of the following entries:

  "PU" if nothing is specified in the code

  "PRI" for private members

  "PRO" for protected members

  "PUB" for public members

5.4.22 Member Method (b) Display Template (D22)

The member method (b) display template (D22) is generated from the USES relationship of the object concept template (7.1.1 of the C7). From the first, second, and third columns of the template, the concept templates C4, C3, and C4 are generated, respectively.

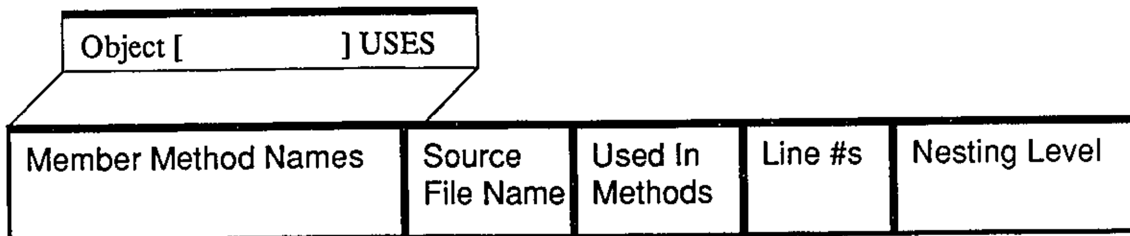


Figure 5.62

Display Template for Member Method (D22)

EXAMPLE:

```
/* source file "example.CC" */

Edgelist   edges;// edges is an object of type
          // Edgelist

void myfunc()
{
    int i=100;
    edges.num();        // line 46
    while (i < 100)     // line 47
    {             // line 48
      if (i > 50)     // line 49
        edges.mes(i) ;//line 50
    }
}
```

The display template D22 displays the following information:

- Member Method Names: The first column shows a list of the member method names called by the object. For the example above, this column shows **"num"** and **"mes"**
- Source File Name: The second column shows the entry for the source file names from which the object makes the call to its member methods. For the example above, the column shows **"example.cc"**.
- Used In Methods: The third column shows the method names in the source file from which the object makes the call. For the example above, the column shows **"myfunc"**.
- Line #s: The fourth column shows the line number of the source file from which the object makes a call to a method. It may show the entries repetitively if the object calls the same member method more than once from within the method. For the example above, the column shows **"46"** and **"50"**.

- Nesting Level: The last column shows the numeral indicating the level of nesting of the logical blocks from which the object makes the call. For the example above, the column shows "0" for line# 46 and "2" for line# 50.

5.4.23 Another Object's Member Methods (RHS) Display Template (D23)

The another object's member methods (RHS) display template (D23) is generated from the USES relationship of the object concept template (7.1.2 of the C7). From the first, second, third, and fourth columns, the concept templates C7, C4, C3, and C4 are generated, respectively.

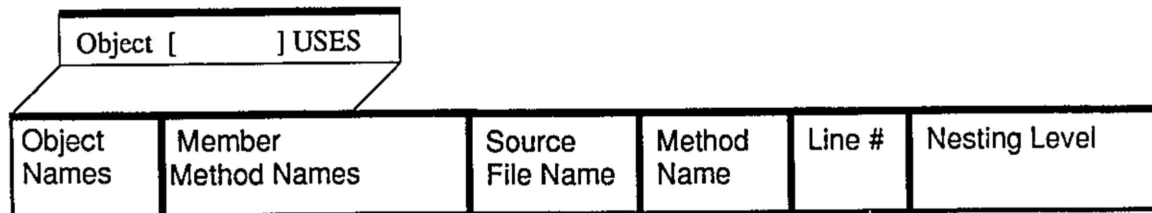


Figure 5.63

Display Template for Another Objects's Member Method (D23)

The display template D23 displays the following information:

- Object Names: The first column shows the names of other objects whose member methods are used by the object.

For details of the other columns, see the discussion of the Member Method (b) Template (D22).

5.4.24 Methods (RHS) Display Template (D24)

The methods (RHS) display template (D24) is generated from the USES relationship of the object concept template (7.1.3 of the C7). From the first, second, and third columns, the concept templates C4, C3, and C4 are generated, respectively.

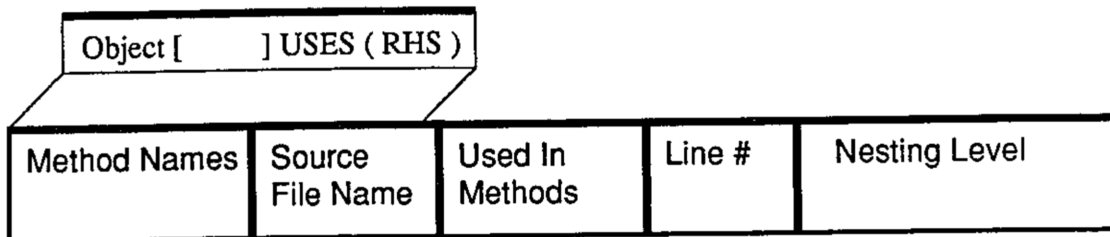


Figure 5.64

Display Template for Methods (RHS) (D24)

The display template D24 displays the following information:

- Method Names: The first column provides a list of the methods used by the object.

For details of the other columns, see the discussion of the Member Method (b) Template (D22).

5.4.25 Objects (RHS) Display Template (D25)

The objects (RHS) display template (25) is generated from the USES relationship of the object concept template (7.1.4 of the C7). From the first, second, and third columns, the concept templates C7, C3, and C4 are generated, respectively.

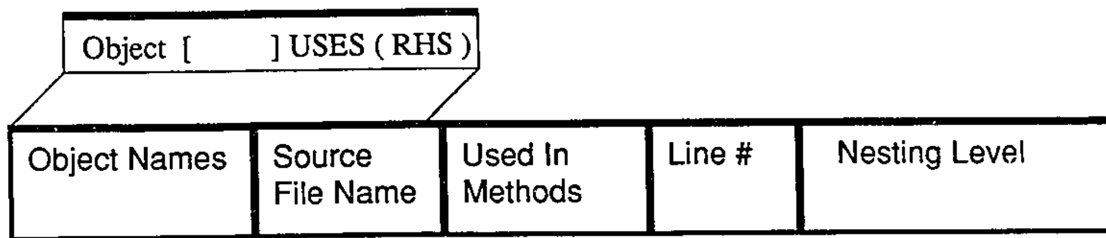


Figure 5.65

Display Template for Objects (RHS) (D25)

- Object Names: The first column provides a list of object names.

For details of the other columns, see the discussion of the Member Method (b) Template (D22).

5.4.26 Variables (RHS) Display Template (D26)

The variables (RHS) display template (D26) is generated from the USES relationship of the object concept template (7.1.5 of the C7). From the second and third columns of the display template, the concept templates C3 and C4 are generated, respectively.

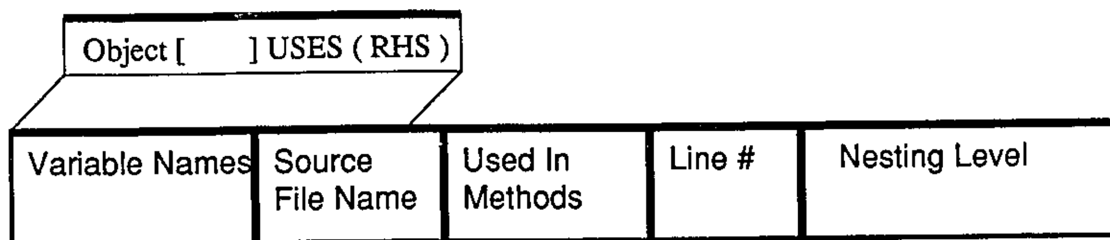


Figure 5.66

Display Template for Variables (RHS) (D26)

- Variable Names: The first column provides a list of variable names.

For details of the other columns, see the discussion of the Member Method (b) Template (D22).

Chapter 6 Conclusion

This work has shown both a scheme and general specifications for a reverse engineering tool for organizing C++ programs. The proposed schema is an attempt to improve understanding of existing C++ programs. The work on the proposed schema may also be viewed as specifications for creating an environment for object-oriented software development. The objective of the thesis was to prepare design specifications for a program-understanding tool. Based on these specifications, we carried out some initial work to explore the feasibility of such a tool. The creation of a fully working tool was not part of the original plan. The details of our initial work in implementing such a tool are given below. A plan for future work on the proposed schema is presented in section 6.2.

6.1 Implementation

As discussed in Chapter 1, we need the following components in order to implement a tool based on the proposed schema:

1) A parser and a semantic analyzer to perform lexical, syntactic, and semantic analysis of the software code.

2) An information base for storing the information in an internal representation; from this view, a composer composes the views.

3) A view composer to create views for understanding purposes.

A description of what has been achieved as regards implementation is given below.

Parser and Semantic Analyzer: We have developed an analyzer to parse and analyze the following:

1) Abstract Data Type Templates, analyzed for

nesting: any level of nesting of abstract data type

multiple inheritance: forward declaration

member methods: inline, virtual

data members: pointers, basic type, storage type, typequalifier

accessibility: public, private or protected

2) Variables, analyzed for

type: int, float, char, struct, union, double, class

storage type: extern, typedef, static, auto, register

pointer: whether the variable is a pointer

typequalifier: constant, volatile

3) Methods, analyzed for

parameter: basic type, pointer, abstract data type declaration, typedef

return type: basic type (void included), pointer, typedef, abstract data type

parametric: whether the method is a parametric method or not

4) Member Methods, analyzed for the items mentioned in 3), and also for

storage type: inline, virtual

Information Base: The information stored after analysis is kept in linked lists; however, it is flattened in tabular form for storage purposes. We are maintaining three linked lists

for the information base. Each of these lists is described below.

1) **"Derivation" linked list:** contains the inheritance information of the abstract data type templates. For an abstract data type, this provides the information about from which abstract data type template the abstract data type inherits, and from what part.

2) **"Member" linked list:** contains information about the abstract data type template declarations of objects and variables, as well as their attributes, such as storage type and basic type.

3) **"Parameter" linked list:** contains information about the parameters of ordinary and member methods declared in the source files.

View Composer: The user interface of the view composer has been designed and implemented using X-View.

6.2 Future Work

Some of the immediate future work on this schema involves the creation of an environment to implement the schema.

Further work is required to incorporate semantics into the schema which will enable the schema to synthesize higher abstractions and fully facilitate reuse.

The proposed work presents schemata for representing C++ key concepts using concept templates, displaying information using display templates, and navigating among the concept templates and the display templates in order to understand C++ software. The program understanding schema presented in the thesis combines both object-oriented and structured programming. A tool based on the proposed schemata can also be enhanced to meet other program-understanding objectives as well: we wish to manage complexity, generate alternate

views, recover lost information, and carry out impact analysis, validation and verification, reverse documentation, redocumentation, analysis, design recovery, and natural language processing.

Manage Complexity: In the schemata presented, complexity is managed by providing a systematic classification of the useful information within a framework of the seven concept templates and three relationships. The seven concept templates, namely executable file, object file, source file, method, logical block, abstract data type, and object, are used to represent the key concepts of a C++ program. The three relationships, USES, HAS, and ISA, define how these concepts interact with each other. For a complex software system implemented in C++, the information is classified into one of the 21 categories shown in Figure 4.1. The schema provides more than 150 different types of information, using 26 display templates. The templates are designed so that the context-sensitive information of the previous level is always available, facilitating comprehension of the software. Further comprehension can be achieved by grouping the templates according to the information in them.

Generate Alternate View: The proposed schema provides a very easy path along which alternate views, such as class diagrams and object diagrams, can be generated. The proposed schema can be easily represented in icon form. The entire schema can be viewed as a graph consisting of collapsible and uncollapsible nodes, while the relationships can be viewed as the edges connecting these nodes.

Recover Lost Information: By providing ways to create alternate views using the proposed schema, forward engineering design documents such as class diagrams and object diagrams can be generated. This document can provide lost information by comparing it with the original documentation.

Impact Analysis: Impact analysis can be directly seen by observing the USES relationship

at different abstraction levels. For example, if a modification is carried out in an abstract data type template, then its impact can be easily seen by viewing the ISA relationship in an object concept template. It is also possible to discover how an object of that data type is affected and where (e.g., at the source file level, the method level, or at a logical block). Furthermore, it is possible to see how many other objects this object affects, and where (e.g. information display template D25, Object USES Object (RHS)). Future work needs to be done to incorporate query-based impact analysis.

Validation and Verification: By generating a specification-oriented suitable view, the validation and verification can be carried out. The specification of an object-oriented design such as a class diagram or persistence can be verified by observing the appropriate display templates of the schema.

Design Recovery: The object-oriented design of the system can be easily extracted from the schema. The abstract data type ISA relationship can be used to create an inheritance chart. The HAS relationship of the abstract data type can provide the class diagram. An object USES and HAS relationship diagram can provide the persistence or scope of the object.

Currently, the schema presents only static aspects of a C++ program. Future work could involve incorporating the dynamic aspects of C++ and exploring how the present schema can be used for a real time system (if possible). It could also involved investigating how the schema can be adapted as a forward engineering methodology, independent of any programming languages. In this case, the methodology itself will have the inherent advantage of all the reverse engineering aspects presently pursued for different software programs developed using different languages.

References

[1] Ackroyd, M. and Daum, D. (1991). *"Graphical Notation for Object-Oriented Design and Programing,"* Journal of Object-Oriented Programming, 3:5, 18–28.

[2] ANSI / IEEE Std 729–198.

[3] Arnistead, M. and Burnham, J. (1990). *"HP C++/Softbench: a development environment for C++,"* Journal of Object-Oriented Programming, 3:4, 42–60.

[4] Arnold, P., Bodoff, S., Coleman, D., Gilchirst, H., and Hayes, F. (1991). *"An Evaluation of Five Object-Oriented Development Methods,"* Journal of Object-Oriented Programming, 3:5, 107–121.

[5] Bedi, F. and King, D.G. (1992) *"GLAY: Graph Layout,"* Technical Report (Prepared for Bell Canada), Simulation and Software Engineering Quality Assurance Research Group, Computer Science Department, University of Ottawa, Ottawa, Ontario.

[6] Birta, L.G., Abou-Rabia, O., Ören, T.I., King, D., and Wendt, N.R. (1992). *"Reverse Engineering in the Simulation Life Cycle,"* SAMS 9, 69–89.

[7] Booch, G. (1991)., *"Object-Oriented Design: with Applications,"* NY: Benjamin-Cummings.

[8] Chikofsky, E.J. and Cross II J.H. (1990). *"Reverse Engineering and Design Recovery: A Taxonomy,"* IEEE Software, 90, 13–17.

[9] Collefellow, J.S. and Bortman, S. (1986) *"An Analysis of the Technical Information Necessary to Perform Effective Software Maintenance,"* Proceedings of the Phoenix

Conference on Computers and Communications, Scottsdale, Arizona, March 1986, 420–424.

[10] Ghezzi, C., Jazayeri, M., and Mandrioli, D. (19??)., "*Fundamentals of Software Engineering*," NJ: Prentice Hall.

[11] (1991.) "*Product News*," Journal of Object-Oriented Programming, 3:6, 65.

[12] (1991.) "*Product News*," Journal of Object-Oriented Programming, 4:2, 73.

[13] (1991). "*Product News*," Journal of Object-Oriented Programming, 4:2, 74.

[14] (1991). "*Product News*," Journal of Object-Oriented Programming, 4:3, 78.

[15] (1991). "*Product News*," Journal of Object-Oriented Programming, 4:7, 78.

[16] (1992). "*Product News*," Journal of Object-Oriented Programming, 4:8, 6.

[17] King, D.G. and Bedi, P. (1992). "*Repository-based Integrative Software Environment*," Technical Report (Prepared for Bell Canada), Simulation and Software Engineering Quality Assurance Research Group, Computer Science Department, University of Ottawa, Ottawa, Ontario.

[18] Kozaczynski, W., Letovsky, S., and Ning, J. (1991). "*A Knowledge-Based Approach to Software System Understanding*," The 6th Annual Knowledge-Based Software Engineering Conference, Syracuse, NY, Sept 22–25, 162–170.

[19] Lawrence, M., Jhonson, L., Ning, J.Q., Quilici, A., and Devanbu, P. (1992). Panel: "*Program-understanding - Does It Offer Hope for Aging Software?*" The Seventh Knowledge-Based Software Engineering Conference, McLean, Virginia, Sept. 20–23, 238–242.

[20] Littman, D., Pinto, J., Letovsky, S., and Soloway, E. (1987). "*Mental Models and Software Maintenance*," Empirical Studies of Programmers: Second Workshop, Soloway and Iyengar (eds.), Norwood, NJ: Ablex Publishing Corp.

[21] Letovsky, S., Pinto, J., Lampert, R., and Soloway, E. (1986). "*A cognitive analysis of a code inspection*," Empirical Studies of Programmers: Second Workshop, Olson, Sheppard, and Soloway (eds.), Norwood, NJ: Ablex Publishing Corp.

[22] Lowry M.R. (1991). "*Software Engineering in the Twenty-First Century*," Automated Software Design, Lowery, M.R. and McCartney, R.D. (eds.), AAAI Press / The MIT Press.

[23] Ören, T.I., Birta, L.G., and Wendt, N.R. (1989). "*Knowledge Base for SLAM II Understanding System: Templates*," Technical Report TR-89–54, Computer Science Department, University of Ottawa, Ottawa, Ontario.

[24] Oman, P. (May 1990). "*Case Analysis and Design Tools*," IEEE Software, 37–41.

[25] Rumabugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenson, W. (1991)., "*Object-Oriented Modeling and Design*," Englewood Cliffs, NJ: Prentice Hall.

[26] Simon, H.A. (1981). , "*The Sciences of the Artificial*," Cambridge, MA: MIT Press.

[27] Whitehead, A.N. (1968). , "*Modes of Thought*," NY: Free Press (original publication, 1938).

[28] Wirfs-Brock, R., Wilkerson, B., and Wiener L. (1990)., "*Designing Object-Oriented Software*," Englewood Cliffs, NJ: Prentice Hall.

[29] (August 1991). Advertisement, "*Byte*," pp. 117.

[30] (August 1991). Advertisement, "*Byte*," pp. 229.

Appendix A Program Listing

Here we have provided the listing of the following files: makefile, glay.cc, graph.h and graph.c These files are used as an example for presenting a C++ programs based on the proposed schemata.

Listing A.1 "makefile"

```
#SccsId[] = "@(#)makefile 1.1\t11/27/92 DGK";

#==============================================================
# Makefile to build glay
# -- the example repository tool
# -- does graph layout
#--------------------------------------------------------------
.KEEP_STATE:

RISEDIR= ../rise

#GPPFLAGS=
#GPPFLAGS= -DDEBUG
GPP= g++

#==============================================================
# the example tool which does graph layout
#--------------------------------------------------------------
glay: glay.o IML.o IMLtool_glay.o IM.o general.o ctools.o
 $(GPP) $(GPPFLAGS) glay.o IML.o IMLtool_glay.o IM.o general.o
  ctools.o -lnsl -o glay

glay.o: glay.cc IML.h general.h IM.h ctools.h args.h
 $(GPP) $(GPPFLAGS) -c glay.cc

#==============================================================
# the interface manager module
# -- to link with server
#--------------------------------------------------------------
IM.o: IM.cc general.h IML.h IM.h ctools.h
 $(GPP) $(GPPFLAGS) -c IM.cc

#==============================================================
# the interface manager LINK module
# -- to link with tools
```

```
#-------------------------------------------------------------
IML.o: IML.cc general.h IML.h IMLtool.h IM.h ctools.h
 $(GPP) $(GPPFLAGS) -c IML.cc

#=============================================================
# the tool-specific interface LINK modules
# -- should be generated during installation
#-------------------------------------------------------------
IMLtool_glay.o: IMLtool_glay.cc general.h IMLtool.h IM.h ctools.h
 $(GPP) $(GPPFLAGS) -c IMLtool_glay.cc

#=============================================================
# the general repository access modules
# -- all tools link with
#-------------------------------------------------------------
general.o: general.cc general.h
 $(GPP) $(GPPFLAGS) -c general.cc

ctools.o: ctools.c ctools.h
 cc $(GPPFLAGS) -c ctools.c

#=============================================================
# fetch the source files from $(RISEDIR)
# -- all tools link with
#-------------------------------------------------------------
general.h:
 sccs -d$(RISEDIR) get general.h
general.cc:
 sccs -d$(RISEDIR) get general.cc
ctools.h:
 sccs -d$(RISEDIR) get ctools.h
ctools.c:
 sccs -d$(RISEDIR) get ctools.c
IM.h:
 sccs -d$(RISEDIR) get IM.h
IM.cc:
 sccs -d$(RISEDIR) get IM.cc
IML.cc:
 sccs -d$(RISEDIR) get IML.cc
IML.h:
 sccs -d$(RISEDIR) get IML.h
IMLtool.h:

 sccs -d$(RISEDIR) get IMLtool.h
```

Listing A.2 "glay.cc"

```
static char SccsId_C[] = "%W%\t%G% DGK";

#include <iostream.h>
#include <fstream.h>
```

```
#include <strings.h>
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include "graph.c"
#define YES 1
#define SEG 99
#define NO 0


 int  nodeFlg;
 int  edgeFlg;
 int  gridX,gridY,c;

main (int argc, char *argv[])
{
 fstream  inFile,outFile;
 extern char *optarg;
 extern int optind;
 char *inf,*outf,*extFile,*depFile;
 int  aflg;
 char  *lclv;

 int  xFlg = 0, yFlg = 0,
  deposFlg =0,extrFlg = 0,localinFlg = 0,
  localoutFlg = 0;

 outf = inf = depFile = extFile = "";
 Graph  cmg;


 while ((c=getopt(argc,argv,"l:x:y:i:o:e:d:"))!=-1)
 switch(c)
 {
  case 'l':

  if (optarg ="al")
   aflg = YES;
  break;
  case 'x':
  gridX = atoi(optarg);
  xFlg = gridX;
  break;
  case 'y':
  gridY = atoi(optarg);
  yFlg = gridY;
  break;
  case 'i':
  inf = optarg;
  localinFlg = YES;
  break;
  case 'o':
  outf = optarg;
  localoutFlg =YES ;
  break;
```

```
case 'e':
extFile = optarg;
extrFlg = YES;
break;
case 'd':
depFile = optarg;
deposFlg = YES;
break;
case'?':
cout << "Options not Valid";
}
if (localinFlg == YES)
{
inFile.open(inf,ios::in);
if (!inFile)
{ // Check for opening of file

cerr << "Cannot open file " << inf << " for input";

exit (1);
}
}
if (localoutFlg == YES)
{
outFile.open(outf,ios::out);
if (!outFile)

{ // Check for opening of file

cerr << "Cannot open file " << outf << " for output";

exit (2);
}
}


Repository * r;
if (deposFlg || extrFlg)
{
char * server = getenv("RISE_HOST");
// Internet address of RISE host
if (!server) {
 cerr << "Environment variable RISE_HOST is not set.\n";
 exit (3);
} else {
 r = new Repository(server); // server address
}
}
 Graph *g = &cmg ;

if (extrFlg == YES && localinFlg ==YES)
{
cerr << " Location for input data is both repository and ";
cerr << "local Choose either of them\n";
exit (3);
```

```
}

if (localinFlg == YES)
{
inFile >> *g;
inFile.close();
}

if (extrFlg == YES)
{

r->extract (g , extFile);
}

if (extrFlg != YES && localinFlg !=YES)
{ cerr << " Location for input data is UNKNOWN\n ";
exit (3);
}

if (!xFlg || !yFlg)
{ cerr << " Invalid grid size option\n ";
exit (3);
}
int nListSz = cmg.nodes.size(); // Get Node list size
int eListSz = cmg.edges.size(); // Get Edge list size

Node np[nListSz],snp[nListSz];
Edge ep[eListSz];

for (int i=0; i < nListSz; ++i)
{ // Conversion of Linked list to Array
np[i] = cmg.nodes[i];
};

for ( i=0; i < eListSz; ++i)
{ // Conversion of Linked list to Array
ep[i] = cmg.edges[i];
};

cmg.sortNodes (np, nListSz); // sort edges & nodes X-Y
if (aflg == YES)
cmg.layout (np, nListSz,gridX,gridY);
else
{
cerr << "Algorithm for Graph Layout not specified\n";
exit (4);
}

for ( i=0; i < eListSz; ++i)
{ // Conversion of Array to Linked list
cmg.edges[i] = ep[i];

};

for ( i=0; i < nListSz; ++i)
```

```
{ // Conversion of Array to Linked list
cmg.nodes[i] = np[i];

};

if (localoutFlg != YES && deposFlg != YES)
{
cerr << "Location for output data not specified\n";
exit (5);
}

if ( deposFlg == YES )
{
r->deposit(g , depFile);
}

if (localoutFlg == YES )
{
outFile << *g;
outFile.close();
}


return(0);

}



/**************End Of File*******************/
```

Listing A.3 "graph.c"

```
static char SccsId_c[] = "@(#)graph.c 1.1\t11/27/92 DGK";
#include "graph.h"
#include "IML.h"
istream& operator >> (istream& ins, IO_type& a)
{
// Write as per matched definition of a variable
 a.read(ins);
 return (ins);
}

ostream& operator << (ostream& outs, IO_type& a)
{
// Write as per matched definition of a variable
 a.write(outs);
 return (outs);
}
```

```
void Plabel::read (istream& ins)
{
/*****Read Edge/Node label and X-Y Positions*******/
 char delim,nextchar;

 int done,i;

 ins >> done;

 if (!done)
 {
 ins >> lx >> ly ;
 ins >> delim;
 ins >> nextchar; // skip whitespace too
 for ( i = 0 ; nextchar != '~' ; ins.get(nextchar) , i++ )
 {
  label[i] = nextchar;
 }
 label[i-1] = '\0'; // at end of string on top of blank


 }
 else
 {
 lx =0;
 ly = 0;
 label[0] = '\0';
 }


}

void Plabel::write (ostream& outs)
{
/*****Write Edge/Node label and X-Y Positions*******/
 if (label[0] == '\0')
 outs << 1 << "\n";
 else
 { outs << 0 << ' ' << lx << ' ' << ly;
 outs << ' ' <<" ~ " << label << " ~\n"; // NEED the last blank
 }
}

void Edge::read (istream& ins)
{
/*****Read Edge from Node to Node and X-Y Positions*******/
 ins >>arcNum >> src >> dst ;
}

void Edge::write (ostream& outs)
{
/*****Write Edge from Node to Node and X-Y Positions*******/

 outs << arcNum << ' ' << src << ' ' << dst << "\n";
```

```
}

void Node::read (istream& ins)
{
/*****Read Node type and X-Y Positions*******/
 ins >> nodeNum >> type >> x >> y >> label;


}

void Node::write (ostream& outs)
{
/*****Write Node type and X-Y Positions*******/

    outs << nodeNum << ' '<< type << ' ' << x << ' ' ;
 outs << y << ' ' << label;

}

void EdgeList ::read(istream& ins)
{
/*****Read Edges as linked list *******/
 extern int edgeFlg;

 EdgeList *nptr = this;

 int done;

 ins >> done; // 0 at start of record, 1 at end of any list
 if (done !=1)
 edgeFlg = 1;

 while (done==1)
 {
 nptr->edge = new Edge; // allocate record

 ins >> (*nptr->edge); // read into it

  ins >> done;
 if (done!=-1)
 {
  nptr->nxt = new EdgeList;
  nptr = nptr->nxt;
 }
 }
 }


void EdgeList ::write(ostream& outs)
{
/*****Write Edges as linked list *******/
 Edge *nptr = edge;
 EdgeList *curr = this;
```

```
 while (nptr) {
 outs << 1 << "\n"; // 1 at start of record
 outs << (*nptr);
 curr = curr->nxt;
 if (curr)
  nptr = curr->edge;
 else
  break;
 }
 outs << -1 << "\n";  // -1 at end of list

}



void NodeList ::read(istream& ins)
{
// Read the Nodes as NodeList of a Graph

 extern int nodeFlg;
 NodeList *nptr = this;

 int done;

 ins >> done; // 1 at start of record
 if (done!=1)
 nodeFlg = 1;

  while (done==1)
 {
 nptr->node = new Node; // allocate record

 ins >> (*nptr->node); // read into it


 ins >> done;
 if (done!=-1)
 {
  nptr->nxt = new NodeList;
  nptr = nptr->nxt;
 }

 }
 }

 void NodeList ::write(ostream& outs)
 {
 // Write the Nodes as NodeList of a Graph
  Node *nptr = node;
  NodeList *curr = this;
```

```
while (nptr) {
outs << 1 << "\n"; // 1 at start of record
outs << (*nptr) ;
curr = curr->nxt;
if (curr)
 nptr = curr->node;
else
 break;
}
outs << -1;  // -1 at end of list

}

void Graph ::read(istream& ins)
{
// Read the Edges and Nodes of a Graph

 ins >> gtype >> edges >> nodes;

}

void Graph ::write(ostream& outs)
{
// Write the Edges and Nodes of a Graph

 outs << gtype << "\n"<< edges << nodes;
}

int Graph:: write_to_stream(ostream &outs)
{
// Use to RUN with Repository
 outs << gtype << ' ' << edges << nodes;
 return (0);
}

int Graph:: read_frm_stream(istream &ins)
{
// Use to RUN with Repository

 ins >> gtype >> edges >> nodes;
 return (0);
}
Graph::layout (Node *np,int nListSz,int gridX, int gridY)
{
    /****** Algorithm for Laying out the X-Y Co-ordinates***/


 int remndrX,remndrY,j,k,Y=0;
 int pointx = 0;



 for (int i=0;i<= nListSz;i++)

 {
```

```
 remndrX = np[i].x % gridX;

 if (remndrX != 0 )
 {

 j=0;
 while ((np[i].x - (gridX*j)) > 0)
 {
  j++;
 };
 if (np[i].x <= gridX*(j-1)+.5*gridX)
  np[i].x = gridX*(j-1);
 else
  np[i].x = gridX*(j);


 };
};
for (i=0; i<= nListSz; i++)

{

 remndrY = np[i].y % gridY;
 if (remndrY != 0 )
 {

 j=0;
 while ((np[i].y - (gridY*j)) > 0)
 {
  j++;
 };
 if (np[i].y <= gridY*(j-1)+.5*gridY)
  np[i].y = gridY*(j-1);
 else
  np[i].y = gridY*(j);


 };
};
/***** Return the Aligned X-Y Co-ordinates ****/

return (np,nListSz);


}
Graph::sortNodes (Node *np, int nListSz)
{
 /***** Sorting of Node on X and Y co-ordinates***/

 for ( int i=0; i<= nListSz; i++)
 for ( int j=i; j<nListSz ; j++)
  if (np[i].x > np[j].x)
  nSwap ( np ,i,j);

 for (i=0; i<=nListSz; i++)
```

```
for (j=i; j<nListSz ; j++)
 if ((np[i].y > np[j].y) && (np[i].x == np[j].x))
 nSwap ( np,i,j);



 /***** Return the Sorted X-Y Co-ordinates ****/
 return (np,nListSz);
}

/****Helping Function for sorting of Nodes & Edges****/
void Graph:: nSwap (Node *na, int i, int j)
{
// Heping function for sorting of nodes on X-Y
 Node tmp;
 tmp = na[i];
 na[i] = na[j];
 na[j] = tmp;

 }
```

Listing A.4 "graph.h"

```
static char SccsId_h[] = "@(#)graph.h 1.1\t11/27/92 DGK";
#include "IML.h"
#define LENGTH 60
class IO_type {
public:
 friend istream& operator >> (istream& ins, IO_type& a);
 friend ostream& operator << (ostream& outs, IO_type& a);
protected:
 virtual void read(istream& ins) = 0;
 virtual void write(ostream& outs) = 0;
};

class Plabel : virtual public IO_type
{
public:
 void  read(istream& ins);
 void  write(ostream& outs);


  int lx;
  int ly;
  char  label[LENGTH];

 };
```

```
/*************************************************************/
class Edge : virtual public IO_type
{
public:
 void  read(istream& ins);
 void  write(ostream& outs);

 int  arcNum;
 int src;
 int dst;


};

/*************************************************************/

class Node : virtual public IO_type
{
public:
 void  read(istream& ins);
 void  write(ostream& outs);


 int  nodeNum;
 int type;
 int x;
 int y;

 Plabel label;


};



/*************************************************************/

class EdgeList : virtual public IO_type
{
public:
 void  read(istream& ins);
 void  write(ostream& outs);


 Edge  *edge;
 EdgeList *nxt;

 Edge& operator[] (int einx)
 {
 if (einx == 0)
  return (*edge);

 EdgeList * e = nxt;
 for (int i = 1;i < einx; ++i)
```

```
 e = e->nxt;
return (*e->edge);
}

int size()
{ int i=0;
EdgeList *e = nxt;
while (e)
{ i++;
 e = e->nxt;
};
return (i);
}


EdgeList() {nxt = (EdgeList *)NULL; edge = (Edge *)NULL ; }

~EdgeList() {reset();}

void reset()
{
EdgeList *pl;
if (edge) {
 delete edge;
 edge = (Edge *)NULL;
 pl = nxt;
 nxt = (EdgeList *)NULL;
 delete pl;
 }
}


};



/*********************************************************/


class NodeList : virtual public IO_type
{
public:
 void  read(istream& ins);
 void  write(ostream& outs);
public:
 Node  *node;
 NodeList *nxt;


 Node& operator [] (int ninx)
 {
 if (ninx == 0)
  return (*node);
```

```
NodeList * n = nxt;
for (int i = 1 ; i < ninx ; ++i )
 n = n->nxt;
return (*n->node);
}
int size ()
{
int i = 0;
NodeList * n = nxt;
 while(n)
{ i++;
 n = n->nxt;

};
 return (i+1);

}

NodeList() { node = (Node *)NULL ; nxt = (NodeList *)NULL; }

~NodeList() { reset (); }

void reset()
{
NodeList *pl;

  if (node) {
  delete node;
  node = (Node *)NULL;
  pl = nxt;
  nxt = (NodeList *)NULL;
  delete pl;
  }
 }

};


/*****************************************************************/
class RObject;

class Graph : virtual public IO_type,virtual public RObject
{

public:
 Graph() : RObject("MGRAPH") {}

 layout ( Node *, int nListSz,int gridX,int gridY);
 void  read(istream& ins);
 void  write(ostream& outs);
 sortNodes (Node *np, int nListSz);
 void nSwap (Node *na, int i, int j);
 int read_frm_stream(istream &ins);
 int write_to_stream(ostream &outs);
```

```
 public:
 int  gtype;
 EdgeList edges;
 NodeList nodes;

};


/*******************************************************/
```