INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

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

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book. These are also available as one exposure on a standard 35mm slide or as a 17" x 23" black and white photographic print for an additional charge.

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

U·M·I

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

Order Number 8926591

Object-oriented CAD database support for software reusability in computer-aided software engineering environments

Poulin, Jeffrey Scott, Ph.D.

Rensselaer Polytechnic Institute, 1989

Copyright ©1989 by Poulin, Jeffrey Scott. All rights reserved.

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

OBJECT-ORIENTED CAD DATABASE SUPPORT FOR SOFTWARE REUSABILITY IN COMPUTER AIDED SOFTWARE ENGINEERING ENVIRONMENTS

by

Jeffrey S. Poulin

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

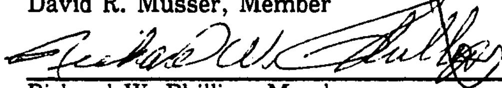
Major Subject: Computer Science

Approved by the
Examining Committee:

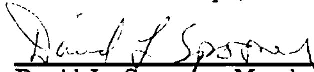
Martin Hardwick, Thesis Advisor

Mukkai S. Krishnamoorthy, Member

David R. Musser, Member



Richard W. Phillips, Member



David L. Spooner, Member

Rensselaer Polytechnic Institute
Troy, New York

May 1989

© Copyright 1989

by

Jeffrey S. Poulin

All Rights Reserved

CONTENTS

Page

ACKNOWLEDGEMENTS xiii

ABSTRACT xiv

1. INTRODUCTION AND HISTORICAL REVIEW 1
   1.1 Software Reusability 1
   1.2 Software Engineering and CASE 2
   1.3 Applying CAD Database Concepts to CASE 4
   1.4 Data Modeling in CAD and CASE 5
   1.5 Contributions 7
   1.6 Outline of the Thesis 8

2. HISTORICAL REVIEW OF SEMANTIC DATA MODELING IN CAD 11
   2.1 Introduction 11
   2.2 The Argument for Database Support of CAD and CASE 11
   2.3 Traditional Data Models 12
      2.3.1 Introduction 12
      2.3.2 The Relational Model 13
      2.3.3 The Hierarchical Model 15
      2.3.4 The Network Model 17
   2.4 Database Technology in CAD/CAM Applications 17
      2.4.1 Shortcomings of Traditional Databases for Engineering Design Data 17
   2.5 Engineering Data Models 19
      2.5.1 Complex Objects 19
      2.5.2 Molecular Objects 21
      2.5.3 Hybrid Models 22
      2.5.4 The Functional Model 23
      2.5.5 Object in a Field 23
      2.5.6 Overview of ROSE 24
   2.6 Data Model Requirements for Support of CASE and Software Reuse 26
      2.6.1 Formal Definitions 26
      2.6.2 For Semantic Modeling of CAD Data 28
      2.6.3 For Semantic Modeling of CASE Data 31
      2.6.4 For Capture of Design Data 32

2.6.5 For Classification of Design Data 32
2.6.6 For Retrieval of Design Data for Reuse 33
2.6.7 For Archive Storage of Reusable Components 33
2.6.8 Scalability 34

3. A NEW DATA MODEL FOR CASE 35
3.1 Introduction 35
3.2 Approaches to CASE Data Models 36
3.2.1 Software Module as a Static Object 36
3.2.2 The Extended Static Module Object 38
3.3 The Interactive Development Model for CASE 39
3.3.1 Introduction 39
3.3.2 Changes to the VLSI Model 40
3.4 Details of the IDM 47
3.4.1 Introduction 47
3.4.2 The Interface 47
3.4.3 The Alternative 51
3.4.4 The Call 56
3.5 Operations and Practice 58
3.5.1 Introduction 58
3.5.2 Operations on the IDM 59
3.5.3 Practice 63
3.5.3.1 Action 1: Developing a Call 63
3.5.3.2 Action 2: Filling the Call- Searching for an Interface 65
3.5.3.3 Action 3: Filling the Call- Searching for an Alternative 67
3.5.3.4 Action 4: Developing an Interface 68
3.5.3.5 Action 5: Developing an Alternative 70
3.5.3.6 Action 6: Developing Similar Modules 71
3.6 Relationship of the IDM to Object-Oriented Program Design 72

4. CAPTURING DESIGN INFORMATION IN A CASE SYSTEM 74
4.1 Introduction 74
4.2 High Level Design Methodologies 75
4.2.1 Functional Decomposition 75
4.2.2 Data Flow Design 76
4.2.3 Data Structure Design 79

4.2.3.1 The Jackson Method 79
4.2.3.2 The Warnier Method 83
4.2.4 Object-Oriented Design 85
4.2.5 IPO Charts 87
4.3 Low Level Design Methodologies 89
4.3.1 Introduction 89
4.3.2 Standard Flowcharts 89
4.3.3 Structured Flowcharts 90
4.3.4 Finite State Machines 91
4.3.5 Decision Tables 92
4.4 Mapping the Design Methodologies to Program Structure 94
4.4.1 Introduction 94
4.4.2 High Level Design Methods 95
4.4.2.1 Data Flow 95
4.4.2.2 Second-Level Factoring 97
4.4.2.3 Data Structure 97
4.4.3 Low Level Mappings 98
4.4.3.1 Flowcharts 99
4.4.3.2 Structured Flowcharts 99
4.4.3.3 Finite State Machines 100
4.4.3.4 Decision Tables 101
4.5 An Approach to Design Data Capture 101
4.5.1 Introduction 101
4.5.2 The Program Static Structure Diagram 103
4.5.3 The Program Dynamic Structure Diagram 103
4.5.4 Data Capture with the IDM 104
5. CLASSIFICATION OF SOFTWARE COMPONENTS 109
5.1 Introduction 109
5.2 Software Classification Options 110
5.2.1 The Interface Definition of a Module 110
5.2.2 Adding to the Interface Definition 111
5.2.3 Formal Semantics 112
5.3 Use of Keywords for Software Classification 113
5.4 Allowable Values for Keywords 115
5.5 Approaches to Software Classification with the IDM 117

5.5.1 Introduction 117
5.5.2 Static Classification Schedule 118
5.5.3 Variable Keyword Lists 119
6. RETRIEVAL OF SOFTWARE DESIGN DATA 122
6.1 Introduction 122
6.2 Accessing Design Data 122
6.2.1 Desired Operations 122
6.2.2 Indexing Strategy 123
6.3 Indexing Techniques 124
6.3.1 Software Catalogues 124
6.3.2 Multilists 125
6.3.3 Cluster Theory 127
6.3.4 Associative Networks 128
6.3.5 Faceted Schema 130
6.3.6 Classification Matrix 131
6.3.7 Artificial Intelligence Techniques 132
6.4 Discussion 132
6.4.1 Matching Needs with Available Components 133
6.4.2 Dependencies of the Retrieval Techniques 133
6.4.2.1 On the Classification Schema 133
6.4.2.2 On the User Interface 134
6.5 Approaches for use with the IDM 136
6.5.1 Introduction 136
6.5.2 Attribute Search 136
6.5.3 Multilist Index 137
7. ORGANIZATION OF THE SOFTWARE ARCHIVE 140
7.1 Introduction 140
7.2 Organization of Software Libraries 141
7.2.1 Application-Oriented Organization 141
7.2.2 Organization Based on Retrieval Method 142
7.2.3 Public Archives and Private Workspaces 143
7.3 Operations on the Software Archive 145
7.4 Organization of Implementation Archive 146
8. IMPLEMENTATION OF THE IDM 149

8.1 Introduction 149
8.2 About the System 149
8.3 A Sample Design Session 151
8.3.1 Introduction 151
8.3.2 Overview of the Design Process 151
8.4 The Design Session 152
9. EVALUATION OF THE IDM 162
9.1 The IDM as a Partial Solution to Reusability in CASE 162
9.2 Storage of Design Data 163
9.2.1 Advantages 163
9.2.2 Disadvantages 164
9.3 Capture of Design Data 166
9.3.1 Advantages 166
9.3.2 Disadvantages 167
9.4 Classification of Software Components 169
9.4.1 Advantages 169
9.4.2 Disadvantages 169
9.5 Retrieval of Software Components 170
9.5.1 Advantages 170
9.5.2 Disadvantages 171
9.6 Organization of the Software Archive 171
9.6.1 Advantages 171
9.6.2 Disadvantages 172
9.6.3 Economy of Scale 172
9.6.4 Levels of Abstraction 173
10. RELATED WORK 175
10.1 Design Data Management in CASE Systems 175
10.2 Existing Systems for CASE 176
10.2.1 Introduction 176
10.2.2 Software Through Pictures 176
10.2.3 Pecan 177
10.2.4 Interactive Ada Workstation 177
10.3 Semantic Data Models for Design Data 179
10.3.1 For CAD/CAM 179

10.3.2 For Software Engineering 180

11. CONTRIBUTIONS TO THE FIELD 181

11.1 Introduction 181

11.2 Contributions to Software Engineering and CAD/CAM 184

11.2.1 To Semantic Modeling of CAD Data 184

11.2.2 To Semantic Modeling of CASE Data 190

11.2.3 To Capture of Design Data 191

11.2.4 To Classification of Design Data 192

11.2.5 To Retrieval of Design Data for Reuse 193

11.2.6 To Archive Storage of Reusable Components 194

11.2.7 To Scalability 195

11.3 Contributions of the Implementation 195

11.4 Conclusion 196

12. FUTURE WORK 197

12.1 Introduction 197

12.2 Research Topics 197

13. DISCUSSION AND CONCLUSIONS 200

APPENDIX I: Prototype IDM Structure 203

APPENDIX II: Operations on the IDM 207

15.1 Introduction 207

15.2 Interface Operations 207

15.2.1 Constraints 207

15.2.2 Operations 209

15.3 Calls 211

15.3.1 Constraints 211

15.3.2 Operations 212

15.4 Alternatives 215

15.4.1 Constraints 215

15.4.2 Operations 216

15.5 Versions of Alternatives 218

15.5.1 Constraints 218

15.5.2 Operations 219

APPENDIX III: User Interface Issues 221

APPENDIX IV: The Correspondence Between DFDs and DSDs 224

LITERATURE CITED 226

Index 242

LIST OF FIGURES

		Page
Figure 1.1	Waterfall Model of the Software Lifecycle	3
Figure 2.1	An Entity-Relationship Diagram of the University Database Example	14
Figure 2.2	University Example using the Relational Model	14
Figure 2.3	University Example using the Hierarchical Model	16
Figure 2.4	University Example using the Network Model	16
Figure 2.5	A Recursive Design Object	18
Figure 2.6	A Non-Disjoint Design Object	18
Figure 2.7	Complex Object Description of a Shift Register	20
Figure 2.8	A Molecular View of a 4-input AND gate	21
Figure 2.9	A Circuit Described with QUEL as a Data Type	22
Figure 2.10	A Functional Database Example	23
Figure 2.11	The Four Types of AND/OR Trees	26
Figure 3.1	The Software Module as a Static Object	37
Figure 3.2	The Two Roles of the Software Interface	42
Figure 3.3	The Interactive Development Model	44
Figure 3.4	Object Hierarchy	44
Figure 3.4	A Partially-Defined I/O System.	64
Figure 3.5	A new Call Icon	64
Figure 3.6	The New Call After Editing	65
Figure 3.7	MVS I/O System with New Call	66
Figure 3.8	Binding an Interface to a Call	68
Figure 3.9	Binding an Alternative to a Call	68
Figure 3.10	The Resulting I/O Subsystem	69
Figure 4.1	Data Flow Diagram symbols	77

Figure 4.2 First-level factoring 77
Figure 4.3 Leveling 79
Figure 4.4 Structured Design Notation 80
Figure 4.5 Module Interface Form 81
Figure 4.6 Data Structure Diagram Notation 82
Figure 4.7 Input and Output Data Structures 82
Figure 4.8 The One-to-One Correspondence 83
Figure 4.9 The Resultant Program Structure 84
Figure 4.10 A Warnier/ Orr Diagram 85
Figure 4.11 Object-Oriented Design Symbols 86
Figure 4.12 An Object-Oriented Design Diagram 87
Figure 4.13 An IPO Chart 88
Figure 4.14 Standard Flowchart Symbols 90
Figure 4.15 Structured Flowchart Symbols 92
Figure 4.16 Structured Flowchart Example 93
Figure 4.17 A Finite State Machine 93
Figure 4.18 A Decision Table 94
Figure 4.19 Second Level Factoring 98
Figure 4.20 DSD-to-Code Conversion Example 99
Figure 4.21 Standard Flowchart Code Sequences 100
Figure 4.22 Code for a Finite State Machine 102
Figure 4.23 Code for a Decision Table 102
Figure 4.24 Symbols for Sequence, Iteration, and Selection 105
Figure 4.25 Icons for Calls, Interfaces, and Alternatives 105
Figure 4.26 Example PDS Diagram 107
Figure 5.1 The Faceted Classification Schedule 114

Figure 5.2 The RSL Classification Schedule 116
Figure 5.3 Describing a Module with a Keyword List 121
Figure 6.1 A Multilist Index 126
Figure 6.2 An Associative Tree for Software 129
Figure 6.3 A Faceted Schema Index 131
Figure 6.4 A Natural Language Query Session 135
Figure 6.5 The Relative Importance of Keywords 135
Figure 8.1 Screen Organization of the CASE Tool 150
Figure 8.2 The Development Tools in the CASE Prototype 153
Figure 8.3 An Undefined Call 154
Figure 8.4 Search/Create Calls/Interfaces 155
Figure 8.5 Searching for a Sort Function 156
Figure 8.6 IPO Chart for Integer Array Sort Interface 157
Figure 8.7 The Interface Icon for the Integer Array Sort 158
Figure 8.8 Search/Create Alternatives 159
Figure 8.9 Scan Versions 159
Figure 8.10 IPO Chart of the Integer Array Sort Module 160
Figure 8.11 Resultant PDS Diagram 161
Figure 8.12 Resultant PSS Diagram 161
Figure 10.1 IDE Data Flow Editor 178
Figure 11.1 A Recursive Call 187
Figure 11.2 A Non-Disjoint Call 188
Figure 13.1 Sample Session with the CASE Tool 202
Figure A1.1 Interface Object Structure 204
Figure A1.2 Alternative Object Structure 205
Figure A1.3 Call Object Structure 206

ACKNOWLEDGEMENTS

I gratefully acknowledge the significant contributions made by my thesis advisor, Martin Hardwick, as well as the many invaluable contributions made by David L. Spooner in the preparation of this thesis. Without the dedication and patience of these two men, this effort would not have been possible.

I would also like to acknowledge and offer thanks to all of the members of my Doctoral Committee. Particular recognition goes to Richard Phillips and Ronald Radice, who have offered wisdom and encouragement for many years.

I also acknowledge the help of Margarita Rovira in providing many thoughts and constructive criticism throughout the implementation phase of this thesis.

A special thank-you goes to my friends, family, and all those who encouraged me when I was down, offered advice when it was needed, and pointed the way when it was not clear to me.

For my grandparents, thank-you. I could not let you down.

Finally, throughout my life and through all things, there are two people who are always *there.* To them, my parents, my love and gratitude is beyond words.

Sponsorship

This research was sponsored by the Fannie and John Hertz Foundation. Their dedication to the advancement of applied physical sciences has provided educational opportunities to many students like myself, and in so doing, enhanced the defense potential and the technological stature of the United States. Without their generous contributions to this lofty goal, this research could not have taken place.

ABSTRACT

There is currently a large research effort underway to develop new techniques and methods for the efficient development of software. However, much of this effort ignores the vast sum of knowledge that has been acquired through our experiences in the field of engineering CAD, especially in the area of VLSI design. Much of what has been learned in this area centers on database support for the design process, and in particular, efficient object-oriented modeling techniques for software design data. It is believed that the data model for software is a central issue surrounding the development of CASE systems.

Recognizing that great gains in software productivity will be realized only when software developed for one application is reused in subsequent applications, it is necessary to consider ways to support reuse through the data model used in these CASE environments. However, the reuse of software components is a complex problem involving methods of capturing, classifying, storing, and retrieving the program design. Unlike CAD, these issues are made additionally complex by the relatively abstract nature of the algorithms and ideas that make up software, as well as the rather specific textual representation of the end product.

Awareness of these reusability issues has led to the development of a new object-oriented semantic data model for use in CASE environments. The proposed semantic data model for software is based on the molecular object model used in CAD, but has been enhanced to capture and support more of the software development cycle. The model differs from the molecular object model in that where the molecular model defines an object as being composed of only an interface and an implementation, this model distinguishes between the interface used for defining an object and an interface that is used to call an object.

The most important consequence of this enhancement is in supporting the reuse of software components. The comprehensive model structure incorporates a classification and retrieval mechanism designed to help map conceptual requirements to existing components in the software archive. This process is further accomplished by providing the designer with specialized operations on the model that assist him in matching an interface calling for service with an interface defined to provide that service.

1. INTRODUCTION AND HISTORICAL REVIEW

1.1 Software Reusability

The problem of reuse of existing software components is recognized as an issue of major importance [Weg84, ReA87], and is crucial to the economical development of large programs. However, software reusability is composed of a number of important subproblems, including how to capture, classify, store, and retrieve the design data. The management of software components in a CASE database and the automatic support for software reuse is the subject of this thesis.

In software engineering environments the capture, or input, of program design data is often accomplished through the use of interactive graphical design editors that are based on data flow diagrams [Mye78], data structure diagrams [Jac75], structure diagrams [You75], or similar schemas. The CASE system must extract the necessary design information from these tools and editors. A CASE system dedicated to reuse will store this design data according to various criteria, organizing the information in a manner consistent with the reuse process.

An important part of the data storage process is a mechanism for classifying design data. Classifying software modules is necessary so that the database knows how to store and access the modules comprising the program design. Much work has been done towards the classification of software components [Pri87, Bur87], and this work is often based on a keyword-style schema. However, while the classification of software is not fully understood, a data storage model can make public certain features of the design so that the program designer can readily store and access program components. Any information related to the classification of these components must be included in the database as part of the data storage model.

The classified software components are catalogued in a software library and retrieved by means of indexing structures such as associative networks [Dep83], faceted

schemas [Pri87], database joins over multilists [Wie87], or sequential catalogue listings such as in the IBM software catalogue. Again, the method used for storage of the program design will dictate the efficiency and ease with which the library components are accessed.

The premise of this work is that at the core of any CASE system there must exist a semantic model for design data that addresses these issues of reusability. A data model that meets this need and at the same time matches the user's conceptual view of the design would make any CASE system that is based on the model efficient and natural to use.

1.2 Software Engineering and CASE

Despite tremendous advances in the field of hardware design and development, the basic nature of computer programming remains unchanged. It is largely an informal, person-centered activity which results in a detailed, formal product [Bal85]. This fact is both the cause and the effect of the fundamental problem of software engineering; managing the knowledge-intensive software development process.

To meet this need, various program design methodologies have been proposed. These methodologies are meant to guide the designer from the requirements of a problem to a well-structured and documented computer program that solves it. Several models, such as the waterfall model of the software lifecycle in Figure 1.1 [Phi88], have been presented to help manage the large numbers of people and other resources involved in large software development projects. Automated tools have gradually been developed to assist programming teams and managers to access and deal with the influx of information. However, only recently is the utility and economics of combining these tools into a comprehensive and integrated workstation-type environment being realized.

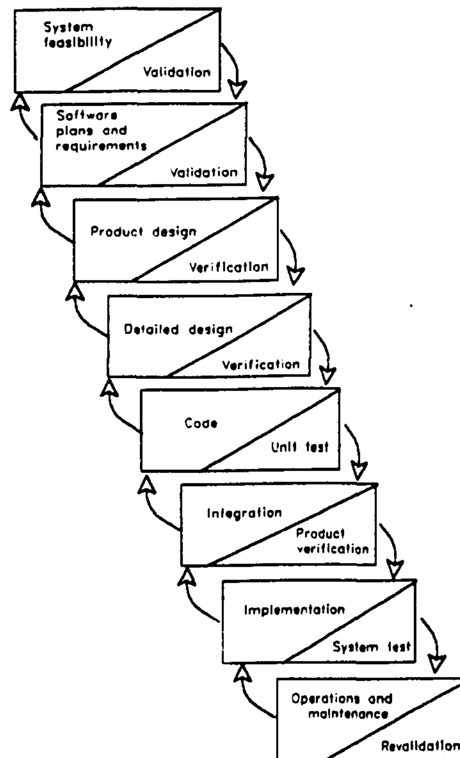


Figure 1.1. Waterfall Model of the Software Lifecycle

Software development workstations are part of what is known as Computer-Aided Software Engineering, or CASE. The aim of these environments is to reduce the administrative workload on the programmers and managers, provide tools for the efficient design and coding of programs, and act as a central library for the software that has been developed. Automatic knowledge-based assistants are often incorporated into the CASE environment as well as specialized tools for documentation and report generation. The end result is a programmer who is much more productive in

terms of both the quantity and quality of the software produced.[1]

Progress in the field of CASE has been deliberate but slow. There has been a call for more integrated systems that combine the tools available at various stages of the design process. However, a limiting factor with such systems centers on the efficient manipulation of large quantities of design data as required by large software projects. This problem is exacerbated by tools that require that design data be stored in a format specifically adapted to that tool. However serious this may appear, it is not unlike the problems faced by developers of CAD/CAM systems years ago. This realization leads us to conclude that there may be at least a partial solution to these software development problems to be found in engineering design systems.

1.3 Applying CAD Database Concepts to CASE

Engineering CAD applications and designers have at their disposal a host of tools to assist in the development of their products. The experience that these designers have with CAD/CAM systems has demonstrated that good application tools contribute greatly to the design process. With this in mind, recent work has been to develop similar tools for software engineering. Early efforts in this area indicate that many of the same advantages that were realized by the users of engineering CAD systems can be found using a CASE system for the development of software. However, since the software design process is less well understood than that of other engineering disciplines, much work remains to be done.

One thing that has been explored is the benefits to using an underlying database to organize the large quantities of data typically managed by a CAD environment. However, many CAD tools have employed conventional database systems in this role, only to discover that the relational data model on which they are based cannot

[1] An excellent review of current CASE technology is found in [Dig88].

gracefully represent engineering design data [Sid80]. It is necessary to view this data as a set of design objects, and new models for design data based on this object-oriented concept are being developed [Hel87].

In taking an object-oriented approach to model design data, it is necessary to develop and utilize specialized databases for management of these objects. If these design tools supported by object-oriented database systems are to be effective, they have to represent application data in a form that matches the user's conceptual view of the data. This requires that these tools be built around a semantic data model that is appropriate for the application. Therefore, at the core of these tools is a data model that mirrors the design of the final product and assists in the process of design. This is especially important in a CASE environment, where the abstract concepts related to software development are much more difficult to quantify than in a traditional CAD/CAM system.

1.4 Data Modeling in CAD and CASE

One data modeling method for CAD/CAM applications that is currently getting widespread attention is the molecular object model [Bat84, Buc85]. This model takes an object-oriented programming approach to design objects; each object has an interface that defines the object for the outside world, and each object has an implementation that specifies how the object is actually composed, or defined. The details of the implementation are hidden from the outside world. The molecular model is partially adaptable to CASE, but while it appeals to object-oriented design advocates, it is lacking properties needed in a dynamic design environment dedicated to reuse.

What the Batory model lacks is flexibility enough for an incremental, interactive, and evolutionary approach to design. This is because the molecular model uses the interface in two ways, both to define objects and to "call" them. This dual role makes the allowable operations on the interface overly restrictive. Furthermore, while

the reuse of existing components is recognized as an issue of major importance, there is no evidence of supporting reuse from the data modeling perspective. This is because the Batory model hides implementation information from the designer even in cases when it might affect the choice of available components. By distinguishing between interfaces used to define objects and interfaces used to call for services, the data model proposed in this thesis eliminates this problem.

An additional important function that the data model must perform in addition to modeling design data is to provide efficient support for design queries and modifications, system documentation, and any software engineering tools that need to access the data. This is especially true when the design becomes large, as is the case with large-scale applications programs such as operating systems. In this new data model, the design of large scale systems is efficiently managed through a cooperative effort between the data model and the strategies for classifying, storing, and retrieving the data. Efficient access of the design data for operations and queries are managed through the use of forward and backward reference pointers in the design objects that allow the control flow, scope issues, and data dependencies to be modeled and stored in a straightforward manner.

The data model in this thesis, termed the **Interactive Development Model,** or IDM, is composed of three parts. The first of the parts is the *interface,* which *defines* a software module. It consists of the common attributes of the alternative implementations for the interface that are visible to the outside world. These attributes not only help define the interface, but are also used to support reuse by serving to locate appropriate interfaces in the software library for a given requirement.

The second part of the IDM is the *alternative.* The alternative is the implementation of an interface, and defines a *strategy for accomplishing the function stated in the interface.* An interface may have any number of alternative

implementations; these are typically distinguished by a fundamental difference in algorithm, language, etc.

The final part of the IDM is the *call*. The call is a *request for service*, and consists of an abstract specification of a software requirement. Unlike the interface and the alternative objects, the call does not define software modules; rather, it serves to bridge the gap between the need for a function and components that are able to meet that need. Specifically, the call object and its associated operations are used to develop the abstract specification into a well-defined requirement that can be satisfied by a particular interface and alternative. The call also serves to record the specification for documentation of the design and for use during program updates and maintenance. The call, therefore, is a flexible and powerful tool for the designer.

In order to validate the theory and ideas that have been incorporated into this data model for software design, a prototype software engineering environment based on this model has been developed. This prototype demonstrates how the data model, working together with a library of design objects representing reusable software components, tackles the major issues of software reusability in a functional system.

1.5 Contributions

The major contribution of this research is the introduction of a new method for modeling design data in a software engineering environment. This new data model, the IDM, is unique in that it was developed primarily in response to an effective means for supporting software reusability in these environments. The model is innovative in that each of the major reusability issues of data capture, classification, storage, and retrieval are inherently addressed by the model.

The issue of data capture is addressed through the creation of a new graphical design tool and methodology for use with the IDM. Data classification and retrieval are studied in detail, with several techniques developed and proven to be compatible with

the IDM and a CASE system that is based on a philosophy of software reuse. Finally, data storage in both the long term archival aspect and the more immediate design database aspect are covered. An IDM-based organization for a software reuse library is presented and shown to fully support distributed CASE systems. Of course, the semantic structure of the design database is the focus of this thesis.

An additional and important contribution made by the IDM is clarifying the role that the interface plays in object-oriented programming methodology. This model stresses the fact that the implementation portion of a design object contains information critical to determining an appropriate use for the object. The model further makes this information available to the designer through the call portion of the design object.

While most semantic data models for CAD only support information related to the final product, in addition to this capability the IDM provides a facility for the development of requirement specifications through the use of call objects. This feature allows the model to be used very early in the design process. Since the call object allows modification of these requirements, it also supports system upgrades and maintenance of the program. Furthermore, the model's representation of the completed design is in a language-independent pseudocode that is useful in numerous applications.

In summary, the reusability issues of software design capture, component classification and retrieval, and information storage are integral parts of a CASE system that is dedicated to increased software productivity through reuse. The three part IDM makes a significant contribution to the field of design data modeling for CASE by addressing each of these issues in a coherent framework.

1.6 Outline of the Thesis

This thesis concentrates on data modeling requirements for supporting software reusability in a CASE environment. The major issues surrounding the reuse of software components are addressed, and a viable solution is proposed. Chapter 2 of the thesis

starts with a detailed review of traditional data models used in commercial database systems. It follows with a synopsis of the advantages of applying database technology to engineering design environments, and then provides a detailed discussion of current CAD modeling techniques. The chapter concludes with an analysis of those elements that are necessary for successfull data modeling in a CASE system for reusability.

Chapter 3 introduces the Interactive Development Model for CASE systems. This chapter then gives a detailed view of the data model, including an overview of the operations that are valid on the model and how the operations are used during the design process. The chapter concludes with some comments on how this model affects the object-oriented design methodology.

Chapter 4 is the first of several chapters devoted to the major issues surrounding software reusability. This chapter concentrates on capturing software design information in a CASE system, and reviews the major software engineering methodologies in use today. The chapter concludes by introducing a new method for design capture that meets the conceptual requirements of the software designer and also corresponds closely to the constructs of the IDM.

Chapter 5 discusses the issue of classifying software components. Several techniques are discussed, and, finally, a method based on keywords is detailed for use with the IDM. Chapter 6 takes a look at methods of retrieving software components once they have been stored in the database system. Finally, Chapter 7 addresses the physical organization of software archives and secondary storage considerations. The primary problem of these software reusability issues is how to match an abstract requirement with a component in the archive that may be able to meet the requirement. Many classification, storage, indexing, and retrieval strategies are discussed, with emphasis on selecting an appropriate approach for use with the IDM and determining the ability of the IDM to function with such techniques.

Chapter 8 gives an overview of an experimental implementation of the IDM in a prototype CASE system. The CASE system is explained, and a sample design session featuring the reusability tools built into the model is given.

Chapter 9 is a discussion and evaluation of how the IDM addresses the problem of software reusability in a CASE environment. Each of the issues of data capture, classification, storage, and retrieval is reviewed, and a conclusion is offered based on the IDM and the issues and topics involved in semantic data modeling for software engineering.

A thorough literature search is provided in Chapter 10, and is documented by the references at the end of the thesis. Chapter 11 presents the value of this research in terms of contributions to the fields of software engineering and CAD/CAM. This chapter also analyzes how the new IDM meets the data modeling and software reuse requirements introduced in Chapter 2. Chapter 12 is on future work, and looks at some areas that will require more research before a complete understanding of data management in software engineering environments is achieved. Finally, the thesis concludes with a discussion and conclusion based on the findings of this research.

The appendices that follow the thesis body detail some of the important aspects of the IDM as well as some related topics. Appendix I is a summary of the data structure used for the CASE prototype. Appendix II is a detailed description of the operations valid on the model and the semantic constraints on those operations. Appendix III discusses some findings on the related topic of user interface issues in CASE systems, and Appendix IV expands on Chapter 4 by summarizing a substantial body of work done in the preliminary phases of this research regarding the automatic conversion of design diagrams to other forms of design diagrams.

2. HISTORICAL REVIEW OF SEMANTIC DATA MODELING IN CAD

2.1 Introduction

Once the software design process begins, design data that has been entered into the CASE system must be organized and stored. Since large engineering problems typically involve enormous quantities of data, a major issue is the efficient managment of this information. This problem was first faced by developers of CAD/CAM systems, and now is being introduced into today's CASE applications. Furthermore, while databases are readily accepted as the tool to meet the data management need, there further exists the problem of semantically representing the information in a form that both the program designer and the database system could easily understand.

Before we can explore how some of these advancements can be used in CASE systems, it is necessary to understand the issues and how they were handled by the CAD/CAM designers. The following chapter provides a foundation for the topic of database issues to CAD/CAM. It is a general discussion of concepts native to database technology, and is oriented at the problem with which we are most concerned: semantic modeling of engineering design data. First, the rational for using databases for the management of CAD/CAM data is given. Second, the concept and purpose of a data model is explained, followed by a review of traditional data modeling techniques. The shortcomings of the traditional techniques are discussed and finally, current techniques used for engineering design applications are presented. The purpose of this chapter is to provide the historical and technical background for the introduction of the IDM for software design.

2.2 The Argument for Database Support of CAD and CASE

As engineering CAD and CASE systems have developed, at some point all have had to address the problem of managing large quantities of design and administrative

data. As the design and size of the project grows, so does the average response time for basic operations. This dictates the need for efficient access to the design data for the purpose of updates as well as for the generation of reports and other documentation.

Years ago, the developers of CAD systems realized the advantages of capitalizing on readily available database technology to support this need, and applied this technology in their application systems. Database features that were found particularly useful were indexing of data files, standard query language capabilities, and controls on access to data for concurrency and security reasons. Over the years, much research has gone into applying this database technology to CAD/CAM applications.

2.3 Traditional Data Models

2.3.1 Introduction

The objective of a data model is to represent, as accurately as possible, the fundamental real-world concepts that an organization or application uses. Therefore, a data model is essentially a *formalism* that expresses the logical structure of data. This formalism helps the database designer organize the problem space and then map his problem to an appropriate representation in the computer.

One of the first and most fundamental concepts used in traditional data modeling is that of the *entity*. An entity is usually an object in the real world about which information is to stored, such as a person, place, or thing. An entity has an associated collection of values, or *attributes* that describe the properties of the entity. Attributes of a person, for example, may include the person's name, address, and telephone number.

The data model must also capture how the various entities in the world interact. For this, the concept of the *relationship* is used. Typical relationships involving people are *married_to* and *is_boss_of*. By defining the entities in a system and the

relationships between these entities, the database designer is usually well on his way to understanding the nature of his problem.

In order to complete his understanding of these entities and how they interact, the database designer outlines the system using a database diagram. This diagram, or *schema*, is a structural representation of the information. Traditionally, there are three different schemas used for database diagrams. These are the *relational model*, the *hierarchical model*, and the *network model*. Each of these models has relative merits and weaknesses, and is discussed in more detail below [Dat85a, Dat85b, Tsi82, Ull82].

The example used to illustrate the three data models in the following sections is based on a familiar university database application. In the university example, students enroll for courses and receive grades for the courses taken. The entities in the example are STUDENTS and COURSES. STUDENTS has the attributes of student number (SNO), student name (SNAME), and student major (SMAJOR). COURSES has the attributes of course code (CCODE), course name (CNAME), and course credit hours (CHRS). The relationship between the entities STUDENT and COURSES is the GRADES relationship. Since a student may take many courses, and a course is taken by many students, there is a *many-to-many*, or *N-to-N*, correspondence between STUDENTS and COURSES. This relationship is shown in the *entity-relationship diagram* of Figure 2.1. In the STUDENTS-COURSES relationship, each student has one letter grade (LGRADE) for each course. This example, as well as the associated diagrams, are derived from [Pot88].

2.3.2 The Relational Model

From the user viewpoint, the relational model is composed of a set of tables called *relations*. There is typically one relation for each entity and one relation for each relationship in the model. There is one column in the entity relation for each attribute of the entity. In the table for a relationship, there are columns to uniquely identify the

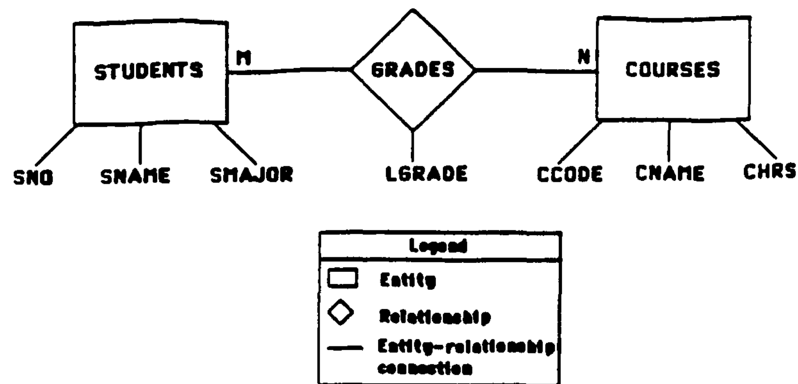


Figure 2.1. An Entity-Relationship Diagram of the University Database Example

STUDENTS

SNO	SNAME	SMAJOR
S1	Ron	Computer Science
S2	Sue	Mathematics
S3	Kim	Computer Science
S4	Bob	Electrical Engineering

COURSES

CCODE	CNAME	CHRS
CS 101	Fortran	3
M 240	Calculus	4
CS 210	Data Structure	3
EE 220	Computer Architecture	3

GRADES

SNO	CCODE	LGRADE
S1	CS101	A
S1	M 240	B
S2	CS 210	B
S2	M 240	C
S2	EE 220	A
S3	CS 210	C

Figure 2.2. University Example using the Relational Model

entities involved in the relationship and additional columns to identify the attributes of the relationship. Information is stored in the rows of the tables; each row is referred to

as a *tuple*. In the university example, each student is uniquely identified by his student number (SNO) and each course is uniquely identified by its course code number (CCODE). These columns provide a *key* for the STUDENTS relation and for the COURSES relation. Taken as a pair (SNO, CCODE), they form a key for the GRADES relation. A relational schema for the university database is shown in Figure 2.2.

Most commercial databases organize their internal data structures in a relational fashion. In this sense, the relational database has become the standard for supporting traditional applications. This is because the relational model and the data manipulation language used to query the relational database are relatively easy to understand, implement, and use. It is important to note that when the next two schemas are used in the design of database, they are often later mapped to a relational schema in order to be implemented on a relational database system. This is not because there do not exist very efficient hierarchical and network database systems, but is because of the proliferation and popularity of the relational model.

2.3.3 The Hierarchical Model

The hierarchical data model is based on the belief that much of the real world can be viewed as being organized in a hierarchical structure. A good example of a hierarchical structure is the ordering of management positions in a large corporation. Such a ranking of positions is usually depicted in a tree diagram, which quickly conveys the relative positions of the members of tree.

The hierarchical model makes extensive use of the parent-child relationship inherent in the model. This relationship is depicted by an arch in the hierarchical diagram. As shown in Figure 2.3, higher-level parent nodes in the tree are connected by arcs to lower-level child nodes. In this example, each course has two child relations. One is a relation that lists all the grades indexed by the student numbers of the students who received them. The other relation is a list of all the students who took the course.

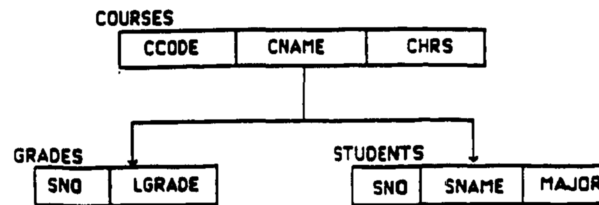


Figure 2.3. University Example using the Hierarchical Model

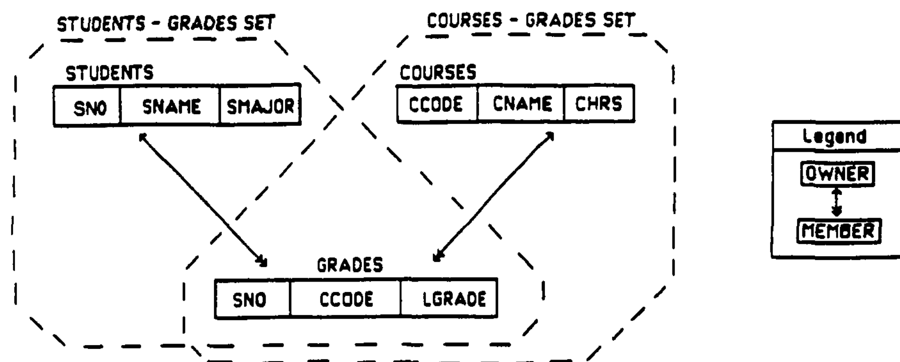


Figure 2.4. University Example using the Network Model

One shortcoming of the hierarchical model is that there is no direct way to represent many-to-many relationships in the diagram. This is because in a tree diagram, no child may have more than one parent. This would create difficulty, for example, in depicting the management structure of a company that used the matrix organization, where an employee may report to one or several managers. In an engineering application this would create difficulty if a subpart was used to construct several assemblies.

2.3.4 The Network Model

In the network model, entities are described by a record type definition, which simply defines the name and the attributes of the entity. Relationships are described by a set type definition. A set type is composed of an *owner* record type and a *member* record type. Many-to-many relationships may be formed by allowing the sets to overlap.

As shown in Figure 2.4, the letter grade of the university example is owned by the STUDENTS-GRADES and the COURSES-GRADES sets. By navigating through the network via a data manipulation language, the LGRADE for a student (SNO) in a given course (CCODE) is uniquely identified.

The primary advantage of the network model is the close correspondence between the conceptual model and the physical implementation of the model. This correspondence makes the system particularly efficient. However, it also makes the model confusing from a user's viewpoint because he must have a working knowledge about the underlying network schema in order to navigate through the data.

2.4 Database Technology in CAD/CAM Applications

2.4.1 Shortcomings of Traditional Databases for Engineering Design Data

Although many VLSI CAD systems exist and have been supported by relational databases, it is now well recognized that the standard relational database is inadequate for modeling and storing design data [Bat84, Has82, Hel87, Sid80]. The reason centers around the nature of the basic entity in an engineering design, the *object.*

The object in an engineering design is either recursive or non-recursive and disjoint or non-disjoint [Bat84, Buch85]. Recursive objects are those that may have sub-parts of the same type. As shown in Figure 2.5, an assembly is a recursive object because it may be composed of other assemblies [Hard87c]. In the diagram an IBIT is used to represent an instance of an object, and an RBIT is used to represent the

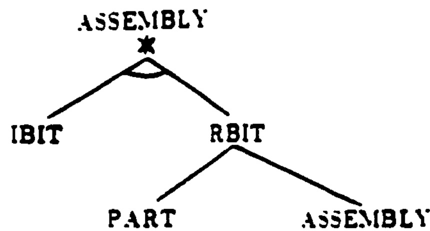


Figure 2.5. A Recursive Design Object

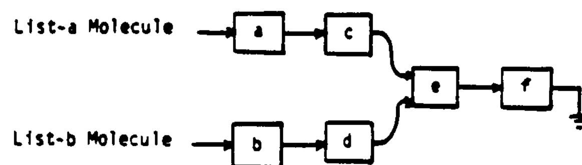


Figure 2.6. A Non-Disjoint Design Object

recursive part of the object; the AND/OR tree notation is explained in a following section. A disjoint object is one that shares no sub-part with another object. An example of a non-disjoint object is shown in Figure 2.6 [Bat84]. Representing design objects with this recursive/non-recursive, disjoint/non-disjoint structure in a relational database is extremely tedious and inefficient.

In addition to the difference between conventional business information and engineering design data, there is the difference in the interactions that take place between the designer and the computer, as well as the nature of the design process itself. One example is the duration of a design transaction. Where a conventional business database might access a record almost instantaneously, the engineer might spend hours, days, or even months to work on a design object. Additionally, the

designer quite often builds one object from a copy of another, with a desire to maintain access to all versions that have been created. The designer might also desire the ability to view different aspects of the design based on his decisions. Therefore, the design database must not only support evolution of design objects from existing objects, but maintain control over multiple versions of these objects, support multiple views of the data in a design, and remain consistent over extended, or *conversational* transactions. Conventional relational database systems do none of these things easily.

In view of all these differences, and the realization that relational databases were not designed to meet the needs of CAD/CAM applications, a new approach had to be found. What was required was a semantic model for design data that maps easily to the user's mental model of the design data and engineering process. This *ease of mapping* means you can have a system that is easy to learn and use because the user can work under the illusion that the computer actually *understands* the objects and operations that he is thinking about [Hei87].

2.5 Engineering Data Models

2.5.1 Complex Objects

To meet the need for a powerful and straightforward representation of design objects, complex objects and object-oriented database systems were developed [Lor83, Plo84, Kim87]. Complex objects are hierarchical groups of tuples consisting of a root tuple that represents a data object, and a set of dependent tuples that define the object. Figure 2.7 is an example of 2-phase shift register and its complex object representation [Kat85]. Note that the 2 Half-shift registers that compose the shift register circuit are merely referenced in the implementation definition (composition) of the shift register and are defined elsewhere. This series of pointers from parent circuit to child circuit are the foundation of complex objects. As can be seen in the figure, complex objects can

```
(TIME Mon May 2 21.43.15 CDT 1983)
(WITHIN (Register 1.0) (Register 2.0))
(INTERFACE
    (POLYGON (0 0) (0 20) (20 20) (20 0))
    (PORTS
      (LOCAL PORTNAME In DIRECTION Input TYPE 4:1 LOCATION (0 10))
      (LOCAL PORTNAME Out DIRECTION Output TYPE Gate LOCATION (10 10))
      (GLOBAL PORTNAME Phi1 DIRECTION Input TYPE 4:1 LOCATION (5 20))
      (GLOBAL PORTNAME Phi2 DIRECTION Input TYPE 4:1 LOCATION (15 20))
  )
    (DESCRIPTION 2 phase dynamic shift register cell)
    (PERFORMANCE (DELAY 5 ns) (AREA 42 um BY 42 um) (POWER 10 uw))
)
(COMPOSITION
    (INSTANCE x NAME HalfRegister TRANSLATED (0 0))
    (INSTANCE y NAME HalfRgeister TRANSLATED (10 0))
    (INTERCONNECT
      ((y In) (x Out))
      ((x In) (ShiftRegisterCell In))
      ((y Out) (ShiftRegisterCell Out))
      ((x Clk) (ShiftRegisterCell Phi1))
      ((y Clk) (ShiftRegisterCell Phi2))
  )
 )
 (REPRESENTATION)
)
```

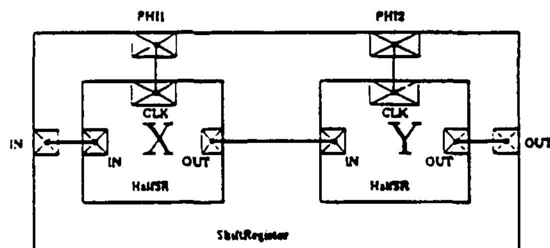


Figure 2.7. Complex Object Description of a Shift Register

succinctly represent the recursive, nondisjoint objects that the relational model cannot easily handle [Bat84, Buc85].

The primary advantage of complex objects is that they offer flexibility for implementation in that tuples may be clustered into relations or into objects. However, for complicated multi-level data structures the structure of the object is difficult to see. Also, the model contains very little semantic information. Although this makes the model extremely flexible, it also makes operations loosely typed and the entire data structure too open for interpretation.

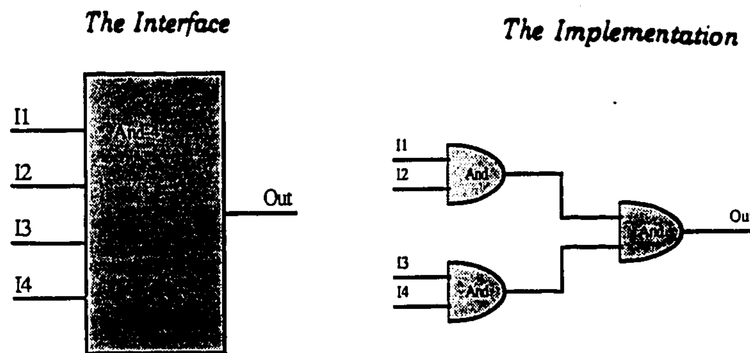


Figure 2.8. A Molecular View of a 4-input AND gate

2.5.2 Molecular Objects

An example of a semantic data model for VLSI design data that is based on complex objects is the molecular object model. In this model, circuit objects are defined to have two distinct parts, an interface and an implementation [Bat85, Kat85]. As shown in Figure 2.8 [Pou89], the interface of an object consists of *connections to the outside world*, and defines how other objects can use or access the design object. It contains the common attributes of the existing implementations for that object. The implementation of the object defines *how the object does its job*, and in VLSI is typically made up of instances of subcircuits and the wires that interconnect them.

In the VLSI molecular model, a designer may refer to a subcircuit without specifying an implementation for that subcircuit. In this case, the designer references only the interface for the object. If the designer does not bind an implementation to this interface, the interface is referred to as a *socket* in the design of the higher level circuit. This socket must then be *plugged* with an implementation for that interface before the design can be considered complete. The plug becomes an *instance* of the subcircuit in the design.

Circuit				
NAME	DESIGNER	DATE	SIMULATION	LAYOUT
adder	john	2/2/82	range s Simulation retrieve (s.all) where s.PID= 00001	range y Layout retrieve (y.all) where y.PID= 00001
I/O bus	mike	3/3/83	range s Simulation retrieve (s.all) where s.PID= 00002	range y Layout retrieve (y.all) where y.PID= 00002
ALU	paul	4/4/84	range s Simulation retrieve (s.all) where s.PID= 00003	range y Layout retrieve (y.all) where y.PID= 00003

Figure 2.9. A Circuit Described with QUEL as a Data Type

2.5.3 Hybrid Models

Hybrid data models are models that extend the relational model so that an item in a tuple can contain an unusual type of data [Har85a, Tsi82]. An example of such a data type would be pointers to other relations, as is done in the Relational/Network Model [Hay81]. The relations that are pointed to would contain further information describing the object. However, unless rules restricting the typing of these pointers are made a part of the implementation of this model, it suffers from the same lack of semantics as does the basic complex object model. Another form of hybrid model is the QUEL as a Data Type model [Sto84], in which actual queries in the data manipulation language QUEL are stored in the fields of a tuple. A conceptual example of this method is shown in Figure 2.9 [Har85b]. These queries are evaluated when the field of the tuple is referenced. However, since the results of this query are dynamic in the sense that they are not evaluated until they are accessed, the actual composition of an object is difficult to capture.

```
DECLARE Student () →→ ENTITY
DECLARE Name (Student) → STRING
DECLARE Dept (Student) → Department
DECLARE Course (Student) →→ Course

DECLARE Course () →→ ENTITY
DECLARE Title (Course) → STRING
DECLARE Dept (Course) → Department
DECLARE Instructor (Course) → Instructor

DECLARE Instructor () →→ ENTITY
DECLARE Name (Instructor) → STRING
DECLARE Rank (Instructor) → STRING
DECLARE Dept (Instructor) → Department
DECLARE Salary (Instructor) → INTEGER

DECLARE Department () →→ ENTITY
DECLARE Name (Department) → STRING
DECLARE Head (Department) → Instructor
```

Figure 2.10. A Functional Database Example

2.5.4 The Functional Model

A functional data model is a binary modeling approach to problem space [Shi81]. The designer views the object in a somewhat mathematical sense, defining, modifying, and accessing the data through a series of functions. An example of a database definition using this method is shown in Figure 2.10 [Spo86]. Once a data value is loaded into the database, it is retrieved by a query using the same functions. For example, executing *Department('John Smith')* might return *'Computer Science.'* The major disadvantage of the functional model in CAD/CAM applications is that there is no way to group related data into objects. This problem is compounded by the fact that this model makes no distinctions between functions that link objects to their attributes and functions that link objects to other objects.

2.5.5 Object in a Field

Object in a field approaches store all of the information describing an object in a field of a tuple. This tuple can be conceptually viewed as a tuple in a relational table or a tuple in a complex object. As a modeling technique, this approach is considered less

flexible than the other techniques because it incorporates a much stronger data typing mechanism. Fortunately, this is an advantage in software engineering environments, where strong data typing is considered desirable. In addition, objects in a field are particularly adept at controlling the visible complexity of a database when objects have a complicated multi-level structure. One particular object in a field approach that will be discussed in the approach used by the ROSE engineering database system.

2.5.6 Overview of ROSE

ROSE is an experimental database system that provides graphics and user interface tools for CAD applications [Har85a]. ROSE is fast, manages data clusters as objects, and provides access to the database through a combination of powerful control structures based on the 'C' programming language and database commands that are an extension of the relational algebra [Har87a].

ROSE stands for Relational Object System for Engineering. As the name implies, ROSE combines some of the features of a relational database system with those of an object-based system. As discussed above, while relational systems are efficient and easy to use, these systems do not represent objects well. The solution adopted by ROSE is to use a relational database not to store objects, but instead, to store information about objects. In effect, ROSE uses a relational database as an index into an object database. This has the advantage of employing the most effective organization for each application; design operations are efficient because design data is clustered into objects, and global operations such as searches are efficient because of the index provided by the relational database [Har87b].

An additional reason for the speed and usefulness of ROSE in real-time applications is that ROSE caches all of the information about a design session into main memory, where it can usually be found in a single search. The ROSE system divides main memory workspace into three areas. The first area is for the application program,

the second area stores object data while the objects are in main memory, and the third area is a scratchpad area used to store the results of computations.

Application programs for ROSE are written in a data manipulation language that is similar to the programming language 'C.' Constructs such as procedures, functions, while loops, and if-then-else statements are provided. Object data is accessed and modified through an extended relational algebra [Har87c]. Because of this, the language is set oriented and strongly typed. The language is extended in the sense that it provides special operators for accessing the recursive data found in engineering applications [Har87c].

One strong argument in favor of using a system such as ROSE for a CASE prototype is that ROSE features a very open architecture. Object data is saved permanently in standard operating system files and directories where it can be viewed and checked. Also, since ROSE is an interpretive system, new functions can be added and the object base can be accessed without having to continuously recompile application code.

Finally, data structures in ROSE are defined in an expressive AND/OR tree format [McL83]. An AND/OR tree is a notation for representing the data abstractions that commonly occur in design applications. Each node in an AND/OR tree defines a domain for an object or one of its sub-objects; in other words, what makes up that object. Figure 2.11 [Har87b] contains examples of these abstraction types. In the figure, the AND nodes represent aggregation abstractions, in which a point is composed of *both* an X-coordinate and a Y-coordinate. The OR nodes represent generalization abstractions [Smi77], which says that a number is *either* of type INTEGER *or* of type FLOAT. An asterisk under either type of node represents an association abstraction. In the figure, a polygon is composed of any number of X,Y coordinate pairs, and a number_collection is composed of a list of numbers, each one of which is either of type INTEGER or of type

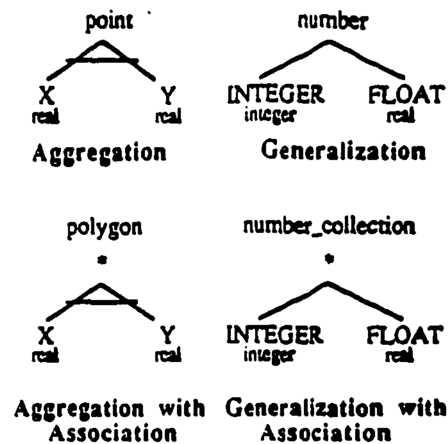


Figure 2.11. The Four Types of AND/OR Trees

FLOAT [Bro84].

2.6 Data Model Requirements for Support of CASE and Software Reuse

2.6.1 Formal Definitions

A data model is a generic concept that defines the rules according to which data is structured. This structure, however, does not provide a complete interpretation about the meaning of data and the way it will be used; operations which are permitted on the data also have to be specified. Finally, disallowed objects or relationships are excluded by defining restrictions, called *constraints*, on the data structures and the valid operations [Tsi82].

Formally, a generic data model **GDM** consists of three parts: a set of data structures **S**,[2] a set of operations **O** on those data structures, and a set of semantic constraints **C**. A generic data model can then be viewed as the three-tuple **GDM = (S,**

[2]Or, alternately, a set of generating rules **G** for the creation of data structures.

O, C). In order to be classified as a database model, each of the elements in this tuple must be addressed.

In addition to these components of a generic data model, a CAD data model **CDM** also includes a number of other criteria. Formally, a **CDM** is the six-tuple **CDM** = (**GDM**, C_{data}, C_{proc}, Obj_{cad}, **A**, **V**), where:

- C_{data} is the conceptual view of data, which in CAD is the molecular formula $C_{data} = In \times Im$, for Interface and Implementation, respectively.
- C_{proc} is the conceptual view of design, which in CAD is a top-down, structured process. Therefore, the structure **S** of each CAD object is viewed as a composition of sub-object structures, or $C_{proc} = S \in S' \times S''$, where S' and S" are complete and independent sub-objects contained in and comprising **S**.
- Obj_{cad} is the capacity to represent CAD objects, or, $Obj_{cad} = S_d + S_{nd} + S_r + S_{nr}$, where the subscripts are disjoint, non-disjoint, recursive, and non-recursive, respectively.
- A is multiple alternatives, or $A = S \in S \times S$.
- V is multiple versions, or $V = S \times T$, where **T** is time.

A data model for use in a CASE system, **CCDM**, must be a CAD data model, but must also include attributes unique to CASE environments. Therefore, a CAD data model $CDM \in CCDM$. Formally, **CCDM** = (**CDM**, **L**, **DT**), where:

- **L** is the set of structures containing the process data for the software lifecycle. Formally, $L = D_{feas} \times D_{req} \times D_{pld} \times \ldots \times D_{code} \times D_{maint}$, where each D_x is the process data **D** for a phase of the software lifecycle, from the feasability study through maintenance.
- **DT** is the representation of data types internal to the program design, $DT \in Obj_{cad}$.

The preceeding discussion has also outlined a number of criteria that are unique requirements for a data model in a CASE system supporting software reusability. Therefore, a data model for reusability in a CASE environment **RDM** can be viewed as the five-tuple **RDM = (CCDM, G, D_{class}, R, PL)**, where:

- G is the invertable mapping from design tools to objects **G: DT $\rightarrow$ S**, where **DT** is a design tool.
- D_{class} is classification data. Furthermore, for every design object **S** there exists a semantically well-defined mapping **CLASS: S $\rightarrow$ D_{class}**.
- **R** is the retrievability mapping **R: D_{req} $\rightarrow$ S.**
- **PL** is the property of programming-in-the-large. **PL** addresses concerns that arise when $\lim_{n \to \infty} \{S\}$ for any software design consisting of a set of **n** structures **S**. For large **n**, critical issues range from archival storage of design information to efficiency of database operations and implementation.

In light of this definition of a **RDM**, fifteen points are identified and presented in order to establish a basis for the evaluation of a data model. While the contents of this list is debatable, it is representative of those qualities that this research has determined to be highly desirable in CASE applications. The CASE/Reusability requirements below are further presented in terms of their relationship to the five issues of reusability that were introduced earlier. These are the issues of semantic modeling, design capture, data classification, object retrieval, and long-term data storage. The fifteen requirements will heavily influence the design of the IDM later in Chapter 3.

2.6.2 For Semantic Modeling of CAD Data

1. *Model must mirror the designer's conceptual view of data.*

Since the goal of a reusability-based CASE system is to increase software productivity by reducing the need to duplicate development efforts, the model must incorporate the concepts of reusable and interchangeable parts. Recent work in CAD has shown that this feature is supported by conceptually separating design objects into interfaces and implementations in the data model. This separation allows the designer a black-box and a white-box view of the object, and allows him to manipulate a design object without having to consider the details of implementation. In addition, through these interfaces and implementations the model must represent software concepts such as control flow, a major consideration in the design of software systems.

2. *Model must mirror the designer's conceptual view of the design process.*

   In CASE, as in CAD, design is an incremental process consisting of an initial design, followed by many product versions brought on by changing requirements. It is also a structured process, consisting of recursively reducing problems of large size into several self-contained subproblems of more manageable size. This reduction can consist of a mixture of bottom up and top down techniques. The CASE system and the data model must provide the capacity to design in either of these methods.

3. *Model must efficiently represent the object structures found in CAD.*

   For efficiency of operation, practical implementation, and conceptual elegance, the data model must be able to represent the types of design objects found in design systems. This includes the recursive, non-recursive, disjoint, and non-disjoint objects prevalent in CAD designs. Failure to do so, as is the case with the traditional data models, immediately disqualifies the candidate model for use in

a CASE environment.

4. *Model must allow multiple implementations/ configurations/ and versions of a design object.*

As discussed above, the design process is an incremental, evolutionary experience. As any design evolves, numerous changes, updates, improvements, and techniques will be tried. It is necessary to manage this history for many reasons, most importantly for documentation of the design. Therefore, the model must have a structure and semantics that support multiple alternatives and version control. Most recent CAD data models have recognized and successfully met this need.

5. *Model must allow ALL externally visible attributes of a design object to be accessible to the designer.*

This is a deviation from standard object-oriented programming process, where the internal details of implementations are universally hidden from the user. However, for a model supporting a CASE system that is dedicated to software reusability, implementation details are often critical to determining the suitability of a module in a given application. This criteria is further discussed in Section 3.3.2. Many current object-oriented data models fail to support reuse on this point.

6. *Each part of the model should have a distinct boundary.*

Distinct object boundaries are important in structured and object-oriented programming, such as Ada,[3] where "scope" rules are critical. Without differentiating between the part of the design under a module's influence and the part of the design that *should* be under the module's influence, the important

[3] Ada® is a registered trademark of the U.S. Government, Ada Joint Program Office.

concept of scoping can be violated. A scope violation occurs when a module makes a call or reference to either a subprogram or a variable that is not "visible," or accessible to the module, according to the rules of the implementation language. The distinctness of object boundaries is also critical in database systems for efficiency of implementation. Without distinct boundaries, retrieval of a database object may cause some or all of the objects that are referenced by the object to be retrieved as well; this process could proceed indefinitely. In a model with distinct boundaries, only those objects specifically needed for an operation are retrieved by the database [Har85b].

2.6.3 For Semantic Modeling of CASE Data

7. *Model must support all phases of lifecycle, from requirements through to maintenance.*

   In order to effectively support the front end of the software lifecycle, the model should separate product requirements and constraints from the definitions of existing components. This allows specifications developed during the planning and initial design phases to be maintained as requirements, and later retrieved for product documentation and maintenance. At the coding end of the lifecycle, the model should maintain design information in an implementation independent form. This allows maximum flexibility in applying the design to various source languages and environments. Most current CAD/CAM/CASE systems and data models only support the middle stages of the software lifecycle.

8. *Model must be able to represent the complex data types that are prevalent in software.*

   CAD models that are capable of representing the design structures described in requirement 3 are capable of also representing complex data object types such are found in software. However, typical CAD semantic models ignore these data type representations. This is due to the situation found in VLSI CAD,

where interfaces are typically pin lists, and are connected to other object interfaces by wires. Since there is only one type of data that travels over these wires, namely an electrical signal that can be in one of only two states, more complex data representations are not addressed. In software, interfaces must pass data in the form of records, arrays, and linked-lists, and other such complex structures. The data model should explicitly provide a mechanism for representing the exchange of this kind of data.

2.6.4 For Capture of Design Data

9. *Model must be compatible with graphical design paradigms.*

A major feature of CAD/CAM systems is the ability to reduce the amount of text with which the designer must directly deal. Numerous graphical design methodologies have been developed in order to do this. In order to do this, the CASE system must monitor the actions in the graphical editors and affect those actions with a clearly defined subsequent action in the database. For accuracy in the design process, there can be no "guessing" in the data capture algorithms. The operations used in the design process must therefore have a direct correspondence with the operations on the data model.

2.6.5 For Classification of Design Data

10. *The model must contain machine recognizable classification criteria.*

The necessity of classifying design components in a CASE system for software reusability is clear; you must be able to store and later find needed components in the database. For reasons of efficiency, it is desirable to have the classification schema included in the design information. The machine can access this classification schema when storing the design objects, and then again when comparing and locating objects during retrieval. For the widest possible application,

the method used should be readily understandable by the general user.

11. *The model must differentiate between the component definition schema and the component requirements.*

Current CAD models utilize database objects that define design components in the additional role of defining needed services. However, this has two shortcomings. First, a designer cannot change the structure of components in the public library of reusable parts whenever a local requirement changes; the integrity of numerous designs may be compromised. Therefore, the same database object cannot adequately perform both functions. Second, a semantic conflict arises during every access of a database object because the database system must first determine the role of the object before operations on it may proceed.

2.6.6 For Retrieval of Design Data for Reuse

12. *Model must support object retrieval strategies that successfully locate reusable components utilizing only abstract criteria.*

Perhaps the most critical of the reusability issues is the problem of retrieving an object when information about the desired part is incomplete or innaccurate. Although many techniques have been proposed and many more are the subject of current research, whatever strategy is desired must (1) be supportable by current database systems, and (2) be supportable by the data model.

2.6.7 For Archive Storage of Reusable Components

13. *Model must be compatible with an archiving method that supports distributed CASE environments.*

Most large-scale CASE systems are organized in a distributed fashion in order to divide responsibility for the project as well as to maximize parrallelism of effort. Normally, this kind of environment is supported by locating publically accessible components in a central location, library, or archive, with development work and partially-completed components stored at local sites or private file directories. Although generally dependent on the CASE system implementation, the structure of the data model must be conducive to this concern.

14. *Model must allow sharing of data among users in a distibuted CASE environment.*

When many users are concurrently developing portions of a large project, or are working on different versions of a design, it is necessary to allow them to exchange their work in a controlled fashion. However, when sharing partially completed designs, the model must gaurantee the integrity of approved data in the archive. Privacy of local workspaces should also be protected. While not expressly the domain of the data model, these concerns must be addressable in the context of the data model.

2.6.8 Scalability

15. *Each of the data model requirements above must be viewed in the context of being efficient for large scale applications.*

Any CASE environment must be capable of supporting a large scale application. With this in mind, each of the reusability issues of data storage, design capture, classification of components, retrieval of components, and organization of a reuse library, must be viewed from a large scale systems perspective. Any data model that cannot support a CASE system for programming-in-the-large is inadequate.

3. A NEW DATA MODEL FOR CASE

3.1 Introduction

The primary concern in choosing a data model for CASE is that it must address the reusability issues of design data capture, classification, retrievability, and it must be supportable by the underlying database system. The data model must also give consideration to the practical requirements of implementation, and any possible restrictions on access time and memory management.

This chapter starts by discussing various data modeling options that were considered for CASE; these options were implemented as part of this research and found to be inadequate. A full explanation of the modeling methods that were developed and investigated, as well as the shortcomings of each method, is given. Throughout the process of developing an adequate data model, special attention is given to increasing software productivity through the reuse of software components that have been developed for other applications.

An analysis of the data models presented in the previous chapter reveals that one model; the molecular object model, is especially promising for use in a CASE system. However, this model needs modifications and enhancements before it can be considered adequate. An explanation of why the VLSI model is of interest is given, and details of the required modifications for application in a software development environment are discussed. The conclusion is that due to the lack of an existing semantic data model that meets all of the requirements facing CASE systems, a new model must be developed for this purpose.

Following the explanation of the shortcomings of the molecular model in CASE applications, a model based on the molecular object concept demonstrating good support for the reuse process is introduced. This *Interactive Development Model* (IDM) not only addresses the software reusability issues, but also is flexible and expressive for use in a

CASE environment where program designs are constantly evolving and rapidly changing. After giving an overview of the structure and theory of the IDM, a detailed discussion of the model, allowable operations on the model, and semantic constraints on the model are given.

1.2 Approaches to CASE Data Models

1.2.1 Software Module as a Static Object

The Software Module as a Static Object data model treats the software module as a complete, autonomous entity, and provides semantics for defining the relationships between modules based on how they are *declared*. A software module is considered to be composed of a parameter list (interface), a list of the subprograms it declares, a code section, and a set of administrative *header* information. Modules are indexed by a surrogate tuple identifier (TID) that is assigned to each module and which is invisible to the user. This static object model was implemented and extensively tested in the early phases of this research, and is visually represented as an AND node in the AND/OR tree of Figure 3.1 [Pou88a]. [1]

However, while the Software Module as a Static Object model is sufficient for describing a program as it looks on *paper*, it does nothing to convey the semantics of how the program actually *works*. This dynamic program information must be derived via an examination of the code portion of the module object and then determining from this code which modules are called. The code portion of this model consists of text in a pseudocode-style format, and does not provide a means for directly identifying modules that are called, nor retrieving them from the design database by means of indices or references.

[1] AND/OR trees are described in Section 2.5.6 and the various types of AND/OR trees are shown in Figure 2.11.

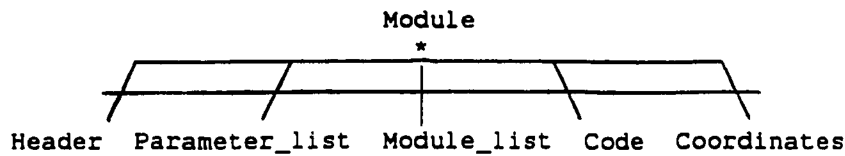


Figure 3.1. The Software Module as a Static Object

Also, the static object model necessarily restricts the nature of the design data it provides and queries that can be supported. Since information on the dynamic activities of the module is not available, valid queries are only those that are based on the declaration structure of the program. Such queries include scope rules for module calls or the number and types of parameters in the interface of the object. Queries pertaining to control flow in the program cannot be supported.

There are several other shortcomings of this model. The first is that there is no provision for version control, and no support for the evolution of one module from another. The semantics of this model do not allow for multiple instances of a module object to simultaneously exist, a criteria which would be necessary for multiple versions and development histories to be supported. An additional shortcoming of the static object model is that there is no basic functional or descriptive information stored as part of the module object, making access for reuse difficult. This information is necessary both for classifying the module as a functional unit, and then later being able to retrieve it for use in another design. Finally, there is no way to represent or document the object other than via the basic static structure that is stored in the model. Other views, especially those that aim to convey the dynamic qualities of the program, are not supported.

3.2.2 The Extended Static Module Object

The Extended Static Module Object model addresses the shortcomings in the previous Static Object model that are associated with supporting multiple views and design methods. The extended static module object does this by extending the static model with the means to store additional documentation information. This additional information describes how the modules reference each other (calling sequence) and how the modules reference data (control block/common data access). However, this information is stored in the model primarily for documentation purposes and is not stored so as to be directly related to the information describing the static structure of the program.

This enhancement to the static object model is aimed at solving only one of the deficiencies experienced by that model; in particular, the problem of supporting a wider range of views of the design data. However, this model still has no provisions for version control or reusability, and for these reasons was found to be inadequate for use in a software development environment where program designs are continuously evolving and where the reuse of program code is desired.

For the enhanced model to address the problem of version control, a meta-object for management of these versions must exist. This meta-object controls references to all existing versions of a module. But since there is no place in the module for this meta-object to be stored, it must be a self-contained entity outside the data model.

Furthermore, for the model to address the problem of classification and retrievability for reuse, a classification schema must be stored as part of the model. However, there are two problems with this approach. The first is that the semantics of the static object are meant to model as closely as possible the textual representation of a module declaration. There is no place for classification information in such a textual declaration. The second problem is that if a classification schema was included in the

model and used to find candidate modules for reuse, the *search criteria* would have no place to be recorded in the object. For example, if a designer required a sort routine at some location in his design, he would query the design database for all modules with that function. However, if the requirements later change, or if new versions of a sort routine enter the database, the model does not allow the designer to "remember" the nature of the original queries so that the query process can be repeated or modified so that the best candidate routine can be found to meet the new situation. This is especially important during the maintenance phase of the software lifecycle, and is a critical feature in a dynamic design environment where the design is constantly evolving. Therefore, it is necessary to save the the requirement specifications in the data model in order to search for reusable components and in order to maintain the design.

In order to address these important issues, it is necessary to divorce the information common to a number of related modules from the information that makes them unique. This is crucial for an efficient version control mechanism, since it reduces the redundancy caused by storing information common to a series of versions. The information common to each version further provides a central location to store descriptive information that can be used to classify and retrieve old components for possible use in new applications. As discussed below, further advantages are gained by dividing module object information into several parts.

3.3 The Interactive Development Model for CASE

3.3.1 Introduction

In light of the above approaches and issues, the Interactive Development Model, IDM, was developed and is proposed for modeling program design data in CASE systems. In this model, the software module is again considered the basic design object.

However, the module is semantically divided into three component objects that are each tailored to the roles they play in program development.

The IDM is strongly influenced by the molecular object model of Batory because of a natural correspondence between the molecular view of VLSI circuits and the object-oriented view of software modules. First, there is the similarity between the VLSI interface, which is composed of a list of pins, and the software interface, which is composed of a list of parameters. Second, there is the correspondence between the VLSI implementation, which is composed of gates, subcircuits, and wires, and the control statements, calls to subprograms, and flow of control found in the implementation of a software module. The IDM extends the two-part molecular model in order to fully support the unique requirements of the software design process.

3.3.2 Changes to the VLSI Model

The molecular model needs to be modified for use in a CASE system in the following ways. First, the concept of instantiation needs to be adapted to mean *a reference*, and not *a copy*. Second, in the molecular model the interface of the object is used in two distinct roles, one in the definition of the object and one in the definition of a call to that object. It is advantageous to divide the interface portion of the data model into two parts in order to accommodate this conflict.

Modification of the concept of instantiation is required in order to clarify the notion that in the final software product, just as in the software design, only one copy of a design object exists. Calls, or instances of the design object, are references to a software module; during execution of the program a call to a module results in the transfer of program control to the physical location of that routine. In VLSI *design*, an instance of a design object is a reference to a single copy of the design specifications for that circuit. However, in the final hardware *product*, every instance of a subcircuit in the design results in a copy of that circuit being created and placed on the chip. This

view of instantiation, which implies the creation of multiple copies of the design object, is not appropriate for software.

The second change to the VLSI model involves the dual roles of the molecular interface. The problem with the interface in the molecular model is that the interface found in the declaration and the interface found in the module call are not the same, and, in fact, have completely different functions. The difference in these two roles is that the interface found in the declaration *represents* the object. Since it may represent many alternatives and versions of that object, there are strict limitations on how the user may modify the interface. The interface in the module call, on the other hand, is a *request for service*. It differs from the declaration interface because it evolves with the design, and may not even represent a particular design object, especially early in the design process. Since the details of the request may change as the design develops, the second kind of interface should, in a sense, be more flexible, and be provided with operations that help guide the designer towards satisfying the need for service. An example of the two ways that a module interface is used is shown in Figure 3.2 [Pou89].

What typically happens is that when a designer needs a subprogram to perform some function, he first searches the database for available routines. If he finds an appropriate routine, he incorporates the interface for that routine into his design as a subprogram call (what Batory would refer to as a socket). If not, he must design his own interface, thereby creating a new object. However, since the molecular model treats definition interfaces and request-for-service interfaces the same, the designer must fully detail the required interface at this time. This is because by treating the defining interface as common to many implementations, the molecular model necessarily places a host of restrictions on how they can be modified later.

The software interface in the module declaration:

Procedure Sort(Var Variable1: Array[1..100] of integer;

Variable2: Boolean);

The software interface in the module call:

Sort(Integer_array, Error_flag);

Figure 3.2. The Two Roles of the Software Interface

What is needed is a *meta-interface* that provides a place for the designer to sketch the requirements for a subcomponent without the need to know exactly what he wants. A new object type, the *call,* is introduced to meet this need. The call not only has a much more flexible set of operations to allow the interface to evolve, but it also has an associated set of descriptive keywords and performance constraints that the designer may use to assist in selecting available components to fill the socket.

The resulting three-part model for software, termed the Interactive Development Model (IDM), is shown in Figure 3.3. The interface portion of the IDM is used exclusively to describe software modules, and plays the definition role of the molecular interface, as shown in Figure 3.2. The interface is composed of those components of the software module that are common to the various implementations of the module, and which may be needed by a program designer to select the module for use in a particular application. The alternative portion of the model is also used exclusively for describing the software module; however, the alternative defines the code and other implementation details of the module. In order to allow the existence of true alternative implementations, any number of this type of object may exist for a given interface. The final portion of the IDM is the call. This object represents a request for service, as

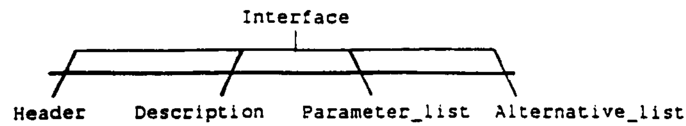
discussed above. However, in describing the request for service in a sufficiently abstract way, the call bridges the gap between requirements and any object available to meet those requirements.

Each object in the IDM is a hierarchical composition of a number of sub-elements that help the object accomplish its role in the data model and in the design process. The "Header" contains the object name as well as other administrative information about the object. [5] The "Description" in the interface and the call portion of the object consist of general comments and a set of keyword identifiers. These fields describe the object and what is needed of an object. The performance constraints in the module call correspond to the performance attributes in the module implementation and are used to describe the requirements of a given call and how well a given alternative meets those requirements. Control of versions and alternatives is via association abstractions, "Version_list" and "Alternative_list," located in the module alternative and module interface. The relationship between interfaces, alternatives, and versions is depicted as an object hierarchy in Figure 3.4. A module is defined by a single interface. This interface may have any number of alternative implementations, and each implementation may have a history consisting of several versions. Further details on the composition of these fields and the exact function of each is given in Section 3.4, Details of the IDM, as well as in later chapters.

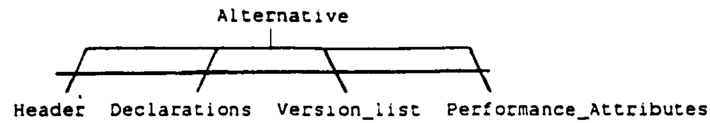
The addition of the call object to the molecular data model has an important consequence for the reuse of design objects. Because there are criteria in each portion of the IDM available to link requests for service with routines potentially able to meet them, the filling of module calls (sockets) should and can be automated. In many cases this may even be left until the design is compiled or validated. At that time, the program that compiles the design will have three sets of criteria on which to base the

[5] In some programming environments the "Header" is referred to as the module *prologue* [Fra87].

For Module Declarations:



For Module Implementations:



For Module Calls:

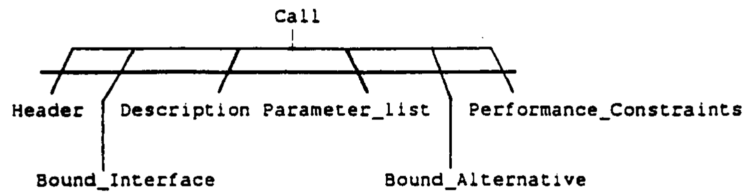


Figure 3.3. The Interactive Development Model

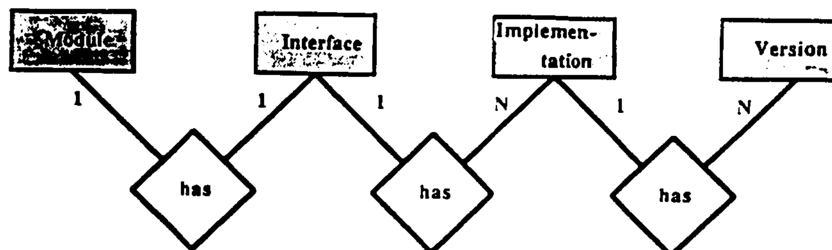


Figure 3.4. Object Hierarchy

selection of an appropriate implementation for the call. First, the initial description of the call provides keywords that partially identify an interface. Second, the parameter

list in the call is compared with the parameter lists in the potential interfaces in both number and type of parameters. At this point, if there are candidate routine(s), an interface for the call is identified, or at a minimum narrowed to a few choices that the designer may easily browse. Finally, the performance constraints specified in the call are matched against the performance attributes of the implementations for the identified interface in order to provide a final selection of implementation. A detailed analysis of the design object retrieval and how the IDM addresses this issue is found in chapter 6.

The call object has the further advantage in that it allows *software requirements* to be stored as part of the program design data. No other semantic model for CAD has this feature. By storing software requirements as part of the data model, the designer is able to develop requirement specifications using the same methods that he uses at lower levels of design. This makes the model conducive to use from the initial problem statement through to the final coded program, providing the capability for a consistent "look and feel" at all levels of design. While software requirements must be saved for documentation of the design, they can also be used during the maintenance phase of the software lifecycle to update requirements and find new reusable components to meet the evolving needs.

The matching of constraints with performance criteria and the use of dynamic binding of calls to interfaces, alternatives, and versions, makes the IDM particularly effective for an evolutionary design process. Through the IDM it is possible for objects filling a call to be referenced *explicitly* as a specific reference, or *implicitly* by dereferencing what is called a generic reference according to some specified or default criteria. For example, a software module that calls another routine may reference a specific version of the routine by number, or it may be left as an implicit reference, in which case a version will be bound to the module call at a later stage of design or implementation. The decision as to what version is to be bound will be made according

to the constraining criteria, such as memory space limitations, algorithm speed requirements, or which version is designated the current version.

The "Bound" field in the call portion of the data model allows the designer to remember an interface or implementation once he has located one in the reusable software library that will fill the call. The designer has the option to make this assignment permanent or temporary. A temporary assignment would be used, for example, if he always wanted to fill the call with the latest version of an implementation. Each time the program design is evaluated for the purposes of design updates or compilation, the last version by date of an alternative would be automatically used to fill the call. A permanent assignment would be warranted if, for example, he always wanted V1.2 of "Binary_sort." When a permanent assignment is used, no automatic binding during evaluation of the design takes place.

The advantages of an automatic binding capability were recently put forward in [Bee88]. Dynamic binding gives the designer the option of leaving an abstract specification in the design, with the assurance that it will always be filled with the most appropriate module available in the software archive. These advantages of dynamically binding design objects to sockets have also recently been recognized by Dittrich and Lorie [Dit88]. Their solution defines "environment" characteristics that are used at run time to qualify candidate objects for sockets. These characteristics are stored as global information in the database. However, it is felt that it is better for conceptual clarity to incorporate this and all other design-related information directly into the design data. This keeps all the information related to objects utilized in a design physically as well as logically in one place.

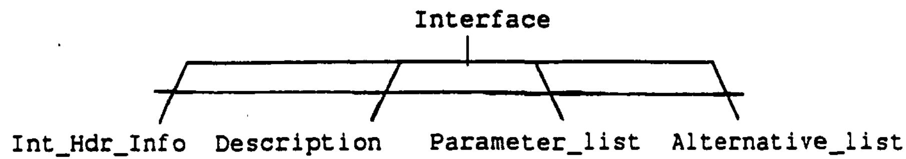
3.4 Details of the IDM

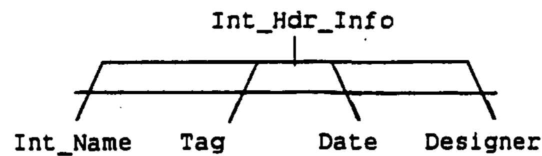
3.4.1 Introduction

This section expands on the general discussion of the IDM model found in the previous section. It is a detailed explanation of the IDM data model and the rational for the various information contained in it. Implementation-specific details are not included in this discussion, except when, for demonstration purposes, it is advantageous to provide an example of a sample schema. The notation used for the data structures in this section is the And/Or tree, as described in Figure 2.11.

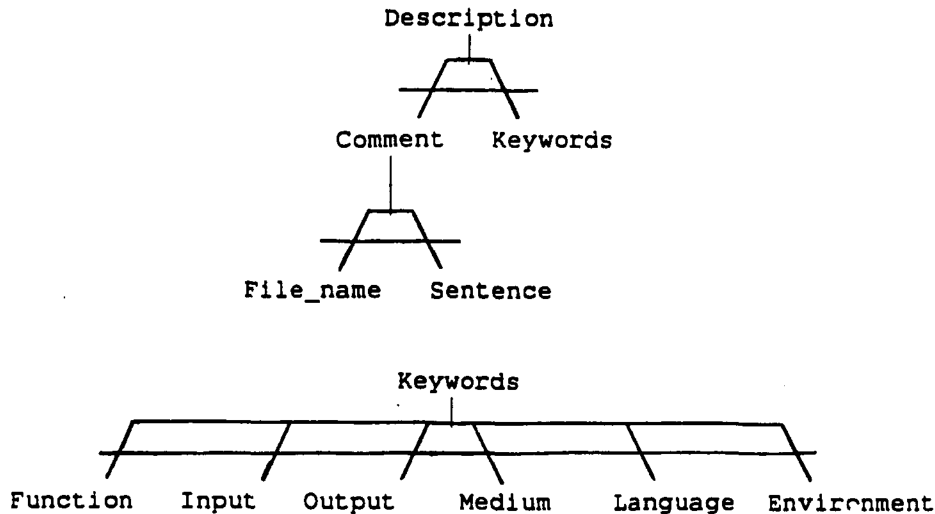
3.4.2 The Interface

The interface of the module serves to define the module object to potential users. It consists of the information common to all of the available alternative implementations. For the most part, once the interface is created, it cannot be modified, since changes to the interface would affect all of the implementations and could have unpredictable consequences. Therefore, significant modifications, including deletion, are only possible if there currently exist no implementations for the interface. A description of the valid operations on the objects that comprise the IDM and semantic constraints on these operations are discussed more fully later in this section.



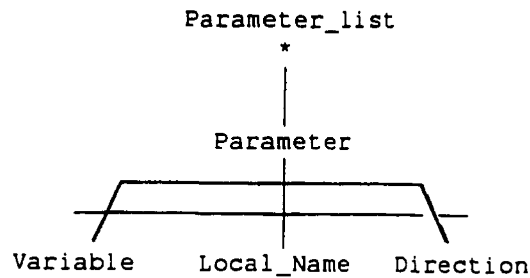


The interface header information, "Int_Hdr_Info," is a composite of administrative information about the interface. For demonstration purposes it consists of the interface name, the name of the designer who created the interface, and the date that it was created. A commercial implementation of the model would include much more detail about the administrative data of the interface. The "Tag" is included as a synonym for the object, and may be used in place of the interface name when no conflict with other objects exists. The tag field serves a similar function for other objects in the IDM.

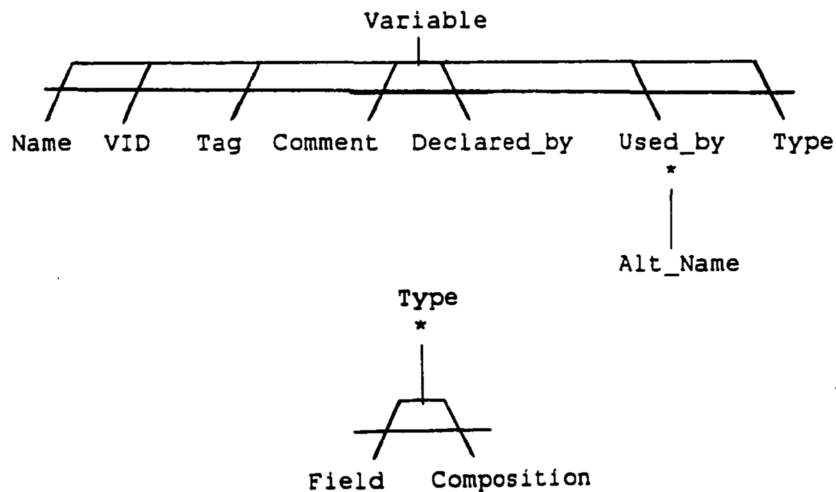


The "Description" of the interface is composed of comments and a series of keywords. These keywords are used to give a general description of the interface for classification purposes, and are used as search keys for retrieval in a reuse situation. A full discussion of the software classification issue and the use of keywords for classifying software components is found in a later chapter. For demonstration purposes, the keywords shown here are "Function," "Input," "Output," "Medium," "Language," and "Environment."

Comments are an unrestricted tool for documenting the object; every "Comment" in this example is shown as having two parts. One part is a short sentence describing the design object, and the other is a pointer to a text file where the user may store any documentation, diagrams, or data necessary.

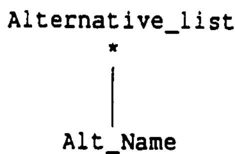


The "Parameter_list" is an association abstraction of parameters; any number of parameters may compose this list. Each parameter is a variable, and has a field for the direction that it passes information, in or out (or both in and out) of the module. The "Local_Name" for a parameter is used to record the name of the variable in the module; the name of the parameter at the point of call is the name of the variable comprising the parameter.



Variables all have a name and a comment describing the intention of the variable. Each variable that is declared in the program design is unique, and is identified by a variable identifier (VID), which is surrogate tuple identifier for variables. Variables also are of a certain data "Type." The type of data may be a basic type such as integer or Boolean, or it may be a compound type composed of one or several fields,

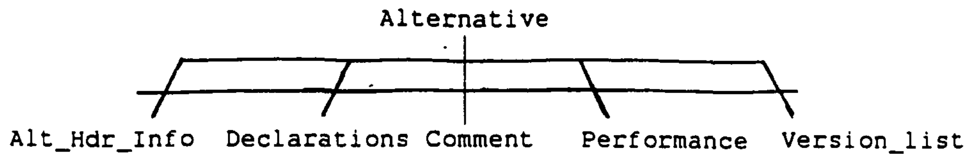
such as a control block or PASCAL record. The variable "Type" is therefore shown as an association abstraction of "Field," which is the name of the field, and "Composition," which describes the field. A field may be a basic type or another type, which creates a recursive structure in the object. There are also backward and forward references, "Declared_By" and "Used_By," that record where the variable is declared and where it is used, thereby allowing efficient queries and fast access during design operations. "Used_By" is an association abstraction, indicating that the variable can be used in a number of different alternatives of a module.

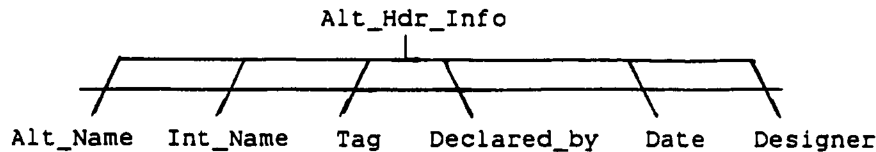


The "Alternative_list" provides links to all of the alternative implementations for this interface. These links are keys used to index the implementations for the interface. The list may be empty, for example if the interface is newly created. However, for a design to be complete, every module interface in the design must have at least one alternative implementation in existence.

3.4.3 The Alternative

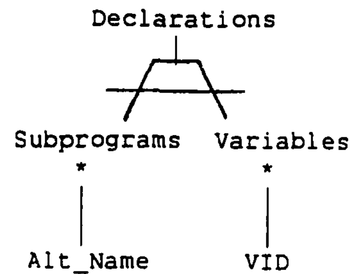
The purpose of the module alternative is to define one way of performing the function specified in the interface. The alternative implementation itself is composed of the information common to all of the versions of the interface. A version is stored via an association abstraction in the "Version_list" field of the alternative object. In order to access a full alternative, the interface name, alternative name, and version number must be specified. If no version number is given, then the version for the alternative that is labeled 'current' is retrieved.



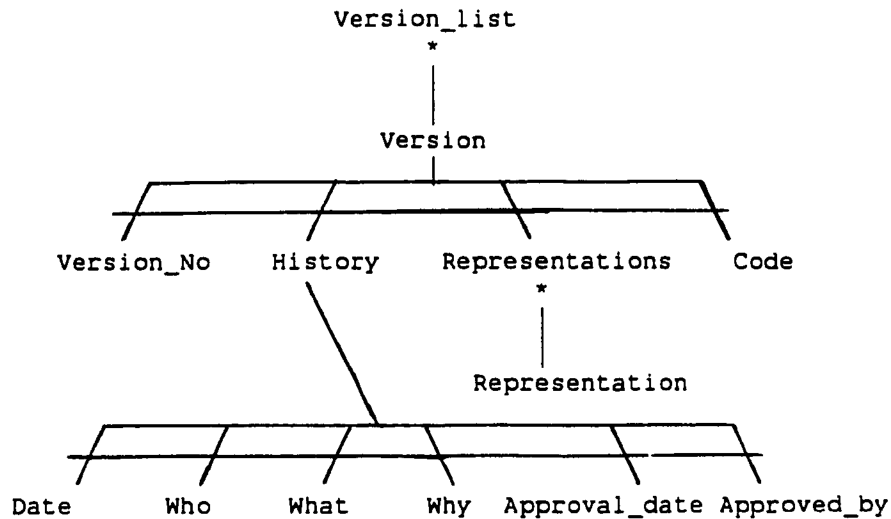


Some of the fields in the alternative object are the same as those in the interface object and will not be explained again here. Of the remaining fields, the alternative header information, "Alt_Hdr_Info," is essentially the same as that for interfaces, with two differences. First, the "Int_Name" field provides a backward pointer to the interface for the alternative. This allows the interface information to be stored in exactly one place, rather than requiring a copy of the interface be maintained in each alternative object. The "Alt_Name" is the name of the implementation for the interface. This field, together with "Int_Name," serves as the key for the alternative object in the database.

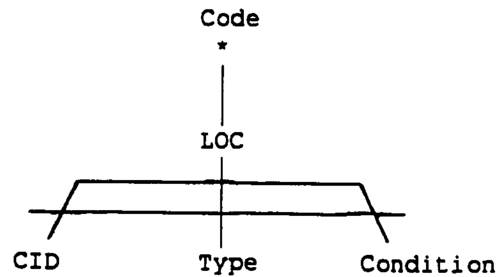
The second difference is the "Declared_By" field, which is a backward pointer to the point of declaration for the alternative, needed to make queries of the type "Where do you come from?" The field is analogous to the *with* clause of Ada [Pyl81, pp. 77-78], in that it determines the source of the object code that comprises the module.



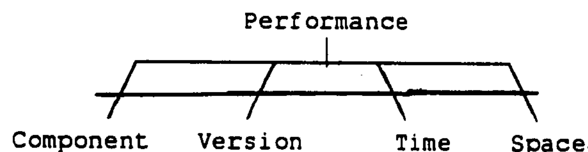
The "Declarations" consist of a list of subprograms, stored as alternative names, and a list of local variables. Alternative names are stored so that scope rules can be checked and enforced, for example, if the designer attempts to call a module that is not visible to the calling module. The declarations for variables are also used to enforce scope requirements. Notice that a variable which is local to a module is considered accessible to any of the subprograms declared by that module, according to the rules of a structured programming language. Variables that are global to the entire program are declared as local variables in the implementation of the main routine. This also provides a mechanism for the declaration of global data blocks, common data areas, and external files.



As mentioned above, version control is managed through the association abstraction of "Version_list" to the existing versions of the alternative. Each version associated with an alternative has a unique version number, code, and modification history that documents who, how, and why it was created. Through the version history the evolution of the alternative implementation can be traced. There are also "Approval_Date" and "Approved_By" fields for the validation of the design: these must be dated later than the modification date for the design to be considered consistent. The "Representations" are intended to support multiple views of the module by providing alternate documentation techniques to design or document the alternative. This field is therefore shown as an association abstraction of "Representation;" it is not further developed here.



The "Code" portion of each version captures the fundamentals of what the module does. Since the IDM seeks to be source language independent, the data structure above models only the three basic programming constructs in structured programming according to [Dij79]. In the IDM, the code of a software module consists of any number of individual lines of code, as represented by the association abstraction from "Code" to "LOC." Each code statement consists of three parts; an identifier stating the type of code construct, a call object identifier, and a condition for the call. The three basic code constructs of sequence, iteration, and selection are represented by "Type." The call object identifier, "CID," references a call object that is the abstract specification of the needed function or requirement. The "Condition" of the code statement is used only if the statement is an iteration or selection construct; it is the Boolean condition that determines if the statement is executed. In a commercial system this field would reference a variable in the database, and be subject to consistency checks on the scope and type of the variable referenced.

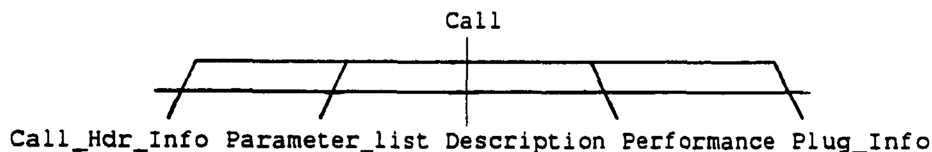


Finally, the performance attributes of the alternative are recorded to describe the object in enough detail to provide criteria for selecting an appropriate alternative

implementation for use in a call. These performance attributes correspond to the performance constraints that the designer specifies while developing the module call. For demonstration purposes these performance attributes are shown to be "Time" complexity, "Space" required, the current "Version," and the logical "Component" that the alternative is a part of in the overall program design. Examples of logical components in an operating systems environment are the Dispatcher component, the Scheduler component, and the Paging component.

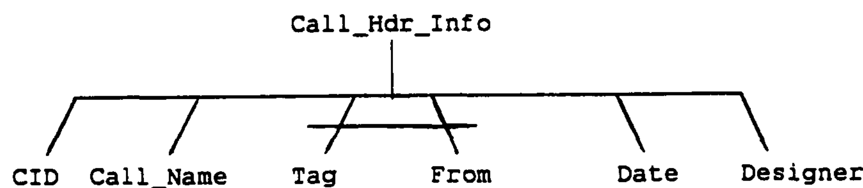
3.4.4 The Call

The purpose of the call is to provide a scratchpad area to develop an abstract service request and to provide database query facilities to assist the designer in the search for reusable components to meet that request. The operations available on calls are therefore very flexible and quite extensive. The call also records software requirements as they are defined at design-time for documentation purposes. It is important to note that the data structures shown here only outline the information that would be recorded in an industrial implementation of this model, especially considering that the call has a very important function as a requirements documentation tool.

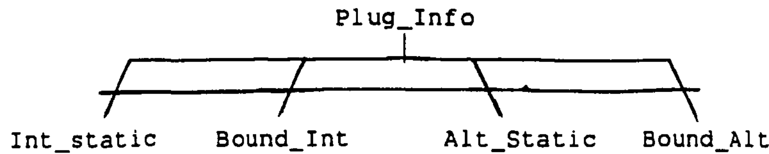


It is important to note that the call object does not *define* software components. That role is accomplished by the interface and alternative objects. The function of the call is to bridge the gap between requirements and components available to meet those requirements. With this in mind, the major parts of the call object all play an important function in the reuse process. The keywords in the call "Description" are used to tell

what is generally needed in the service, and are used by the system for an initial search of available interfaces. The process of locating candidate components then continues, using the other information comprising the call object. The needed "Parameter_list" is compared with the parameter list of the retrieved interfaces for possible matches. Finally, the designer uses the "Performance" constraints to place restrictions on the performance attributes of the alternatives that may be used to fill the call. It must be emphasized that in the call, the values of the keyword attributes *constrain* the interfaces and alternatives that may be used to fill the call.



The header information for the call object is similar in many respects to the administrative information that is maintained for the interface and alternative objects. Like those objects, the call has a name that it is known by, as well as a shorter name for the same purpose. However, since each call must remain unique in the design of a program, the database system assigns a surrogate tuple identifier for that purpose, called the call identifier (CID), to each call object. The "CID" is used to index and retrieve the calls from the database. The final addition to the call header is the "From" field. "From" is a backward pointer to the alternative that instantiated the call. This field makes it possible to efficiently retrace the flow of control sequence in a program.



With the exception of the field "Plug_Info," all the other fields that comprise the call object have been explained. "Plug_Info" is used to record the keys of the interface and alternatives used to plug the call once this information has been determined. As indicated, the user may elect to make this assignment permanent. If this is the case, the "Static" fields are set to the Boolean condition *true*, and the interface and alternative that are bound to the call are recorded in the "Bound" fields. If the "Static" fields are false, then final determination for binding these fields will be made dynamically when the program design is evaluated.

3.5 Operations and Practice

3.5.1 Introduction

This section provides a synopsis of the valid operations on the IDM and a few examples of how these operations combine to perform basic software design steps.

The purpose of this section is to illustrate how the model supports the software engineering process and software reusability. A complete and detailed discussion of the valid operations is given in Appendix II. As part of that discussion, a full analysis of the semantic constraints on the model that must be enforced throughout execution of the operations is given. The concise description of the operations shown on the next couple of pages is provided as an introduction and quick reference for the activities that follow. The activities shown represent common activities during the design process, and were carefully chosen to illustrate how many of the IDM operations are utilized to accomplish these activities.

The complete ROSE implementation of a CASE system based on this data model includes several graphical editors for design data input, a combination of pull-down menus and text-entry boxes, and mouse-driven icons. These interface features are techniques designed to assist the designer as he interacts with the system and uses the reusability features that are built into the model. A description of the prototype CASE system as well as a sample design session is given in chapter 8. The activities described below are given in an implementation-independent format with a syntax based on PASCAL to illustrate the operators for each action.

3.5.2 Operations on the IDM

The following three pages contain tables that outline valid operations on the IDM model. There is a table for each object in the IDM; first the name of the operation is given, then the purpose of the operation. Finally, the resulting action in the data model is briefly described.

Operations on Calls		
Operation	Purpose	Result
create_call:	Add a software requirement specification to the design.	Creates an instance of a new call object in the database.
copy_call:	Create a new requirement similar to an existing one.	Invokes create_call: then copies the attributes of an existing call into the new call
retrieve_call:	Prepare a call for use in an operation.	Fetches a call from the database using a surrogate tuple identifier as a key.
edit_call:	Allow unrestricted modification of a software requirement.	Call is retrieved and call editor invoked.
make_call:	Assign location for a call in the design.	Call ID is added to code of an alternative object.
unmake_call:	Remove software request from design.	Erases a call ID from the code of an alternative object.
unbind_interface:	Change interface used to fill a call.	Sets bound interface and any bound alternative in the call to "null."
unbind_alternative:	Change alternative used to fill a call.	Sets bound alternative in the call to "null."
fill_call:	Locate interfaces and alternatives to meet software requirements.	Automatically searches database and advises designer of results.
display_call:	Allow designer to view call attributes.	Call is displayed on viewing device.
delete_call:	Destroy undesired requirement specification.	Removes a call from the database.

Operations on Interfaces		
Operation	Purpose	Result
create_interface:	Define a new software module.	Creates a new interface object using the attributes of a specified call; adds the interface to the database.
copy_interface:	Create new interface that is similar to an existing interface.	Invokes create_call and copies attributes of existing interface into the new call object; then invokes edit_call
retrieve_interface:	Prepare an interface for use in an operation.	Fetches the interface from the software library using the interface name as a key.
search_for_interfaces:	Find reusable component to meet a software requirement.	Assists designer locate interfaces using keyword search and parameter matching.
bind_interface:	Use an interface to fill (plug) a call.	Associates an interface with a call in the program design.
display_interface:	Allow designer to view the attributes of an interface.	Interface is displayed on viewing device.
display_alternatives:	Allow designer to browse alternative implementations of an interface.	Alternatives are displayed on viewing device.
delete_interface:	Destroy undesired interfaces.	Removes an interface from the software library.

Operations on Alternatives		
Operation	Purpose	Result
create_alternative:	Add a new implementation for an interface.	Creates a new alternative object in the database.
copy_alternative:	Make a new implementation of an interface that is similar to an old one.	Invokes create_alternative; then copies existing alternative into new alternative object.
retrieve_alternative:	Prepare alternative for use in an operation.	Fetches an alternative from the database library using (interface_name, alternative_name) as a key.
search_for_alternatives:	Find reusable component to meet a software requirement.	Assists designer search alternatives using implementation specific data.
bind_alternative:	Designate an alternative to meet a software requirement.	Associates an alternative object with a call object.
edit_alternative:	Update alternative attributes.	Invokes alternative editor.
display_alternative:	Allow designer to view alternative attributes.	Alternative is displayed on viewing device.
display_versions:	Allow designer to browse versions of an alternative.	Show versions on viewing device.
delete_alternative:	Destroy undesired alternatives.	Removes an alternative from the database.

3.5.3 Practice

3.5.3.1 Action 1: Developing a Call

As an example of how the designer uses the IDM to develop an abstract request-for-service into a specific module instance, consider the case where the software designer has a need for special kind of sorting routine to be used in an I/O subsystem of a computer. In this scenario, the subsystem environment will be the MVS operating system. The current state of the design is shown in Figure 3.4; first the subsystem must queue incoming jobs, and then, *if* there are jobs to print, they must be written to an output device. [6] The designer now wants the I/O jobs to be sorted by priority before they are printed, and so, using the operations provided above, the following actions are performed:

1. create_call (var new_call): A generic call is added to the design with all attributes initially containing null values. A new instance of a call object is created and added to the database. This action may be portrayed by the instantiation of a call icon in one of the CASE system's graphical editors, as shown in Figure 3.5.
2. edit_call (new_call): The designer is allowed to enter desired values for the attributes of the call. These values represent the constraints that interfaces and alternatives will be subject to when it is time to fill the call. For example, the designer sets:

    new_call.tag := Sort?,

    new_call.function := sort,

    new_call.input := unsorted_integer_array,

    new_call.output := sorted_integer_array,

    new_call.language := PASCAL,

[6] The notation shown in this diagram is more fully explained in Section 4.5.

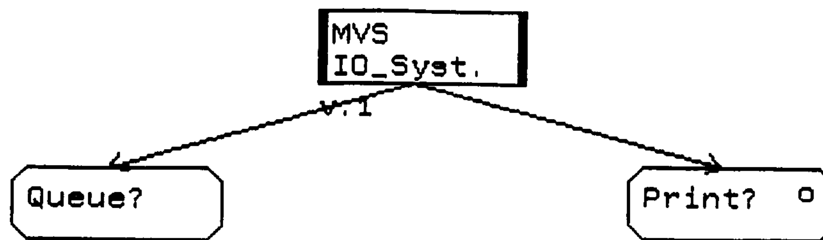


Figure 3.4. A Partially-Defined I/O System.

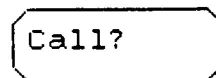


Figure 3.5. A new Call Icon

new_call.time := O(n-squared),

new_call.version := Last

The designer may also enter administrative data such as the current date, his name, and any other information stored as part of the call. The results of the edit_call operation are shown in Figure 3.6.

3. make_call (new_call, alternative, LOC_type, LOC_location, LOC_condition): The call is assigned a location in the "code" of an alternative. First, the CASE system retrieves the specified alternative from the database using the retrieve_alternative operation. Next, the current version of the alternative recieves a new line of code (LOC) of the type and in

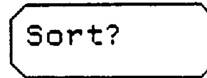


Figure 3.6. The New Call After Editing

the location specified. For example, the designer wants this call to be a line of "Sequence" code, the second statement to be executed in the "MVS" alternative of the interface "IO_System." There is no boolean condition required, since sequence statements do not have guards for iteration or selection. This operation may result in an arc, representing control flow, being drawn between the calling module and the called object, as shown in Figure 3.7.

4. display_call (new_call): The system shows the completed call to the designer. At this time, several activities may be selected. The designer may elect to continue to edit the call, in which case he returns to step 2. He may opt to search for interfaces to meet the requirements he just specified, in which case he proceeds to Action 2, below. Finally, he may ask the system to automatically attempt to fill the call with a reusable component for him by invoking the operation fill_call.

3.5.3.2 Action 2: Filling the Call- Searching for an Interface

The designer wants to use a reusable module in order to meet the requirement he outlined in the call object above. He has opted to conduct a search for a reusable interface to fill the call and consults the public software library for

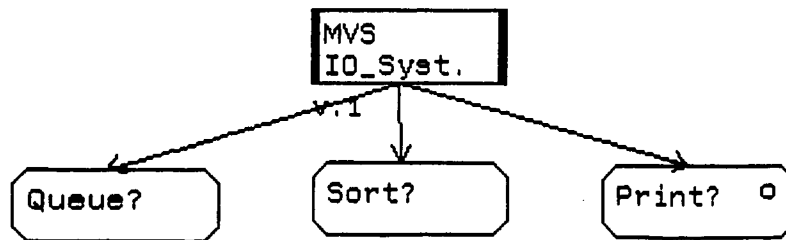


Figure 3.7. MVS I/O System with New Call

possible routines that might be available for his use. The following actions transpire:

1. search_for_interfaces: The system and designer match the call constraints specified in Action 1 with the interface definitions contained in the software library. Failure to locate a possible component normally results in the user varying the parameters of the search and trying again. This is accomplished using the call editor invoked by edit_call. When candidates for reuse are identified, the system loads the interfaces into the local workspace using the retrieve_interface operation.
2. display_interface (current_interface): After querying the software library the designer discovers an existing interface for an integer array sort that appears as if it will serve the function he requires. In order to ensure that this is the case, the interface attributes are displayed for the designer's inspection.
3. bind_interface (var new_call, current_interface): Content with the interface, the designer binds the interface, which in the example is called "Integer_Sort," to the call. In the IDM, the field *new_call.bound_int := "Integer_Sort."* The action might be portrayed by replacing the call icon in

the graphical editor with an interface icon, as shown in Figure 3.8. At this point he may proceed to investigate possible alternative implementations for this interface.

3.5.3.3 Action 3: Filling the Call- Searching for an Alternative

Now that the designer has located a suitable interface and has bound the interface to the call, he desires to browse the available alternatives for the interface, in order to determine if one of those alternatives is suitable for his needs or whether he will have to modify, or completely design, one for himself.

1. search_for_alternatives (new_call): The system and designer match the call constraints specified in Action 1 with the alternative performance criteria for each alternative of the interface "Integer_Sort," which is currently bound to the new call. As with interfaces, failure to locate a possible component normally results in the user varying the parameters of the search and trying again. This is accomplished using edit_call. When a candidate alternative is identified, it is loaded into the local workspace by the system, using the retrieve_alternative operation.
2. display_alternative (current_alternative): After querying the software library the designer discovers a quicksort implementation for the integer array sort interface that appears as if it will meet the constraints outlined in the call. This alternative is named "Quick." The alternative attributes are displayed for the designer's inspection.
3. bind_alternative (var new_call, current_alternative): If the designer is satisfied with the alternative, he binds it to the call. In the IDM the field *new_call.bound_alt := "Quick."* Graphically, the action is represented by replacing the interface icon, which represents a call with bound interface, with an alternative icon, which represents a call with a bound interface *and* a

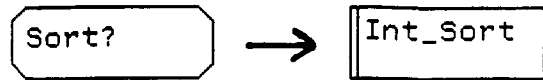


Figure 3.8. Binding an Interface to a Call

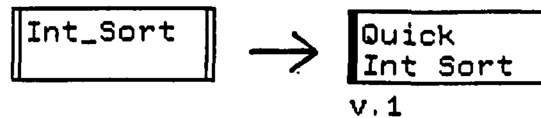


Figure 3.9. Binding an Alternative to a Call

bound alternative. This result of the binding action is depicted in Figure 3.9, and the I/O subsystem design as it now stands is depicted in Figure 3.10.

3.5.3.4 Action 4: Developing an Interface

Assuming that the designer was unable to locate an interface in the library that met his needs, he must create one that is customized for his own application. For example, consider that instead of sorting an integer array, the designer wished to search an array. The following actions are required:

1. create_interface (var new_call): An instance of a new call object is created in the database. A call object is created rather than an interface object because editing interface objects is restricted in order to guarantee the integrity of module definitions. This is discussed more fully in Appendix II.

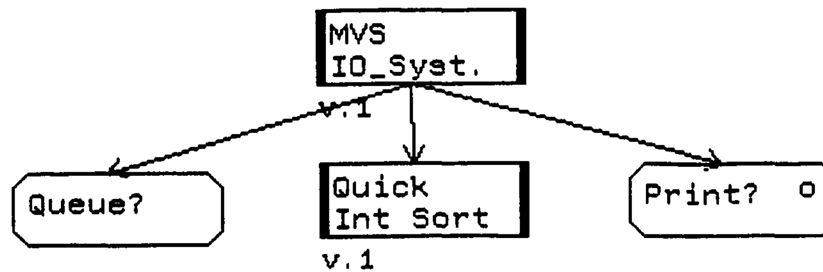


Figure 3.10. The Resulting I/O Subsystem

Initially, all of the attributes of the new call object contain null values.

2. edit_call (var new_call): The designer enters desired values for the attributes of the future interface. These values will define the interface and serve as search indices in later search_for_interfaces. For example, the designer sets:

   new_call.name := Integer_array_search,

   new_call.function := search,

   new_call.input := sorted_integer_array,

   new_call.output := index,

   new_call.language := PASCAL,

   new_call.medium := ring_buffer,

   new_call.environment := MVS,

   The designer may also enter administrative data such as the current date, his name, and any other information stored as part of the interface header.

3. display_call (new_call): In order to check the values of all the attributes, they are displayed for the designer's inspection.

4. The last step is a decision as to whether or not to actually make the call into

an interface using create_interface, or simply save the call for later modification. If the interface operation is committed, then the values of the call become locked as a new interface object and are unchangeable. Before the operation is allowed to commit, however, the interface name is checked for uniqueness. If the interface is not committed, the call is saved as a new call object in the database. If the operation is aborted, the instance of new_call is destroyed.

3.5.3.5 Action 5: Developing an Alternative

Assuming that the designer creates an new interface object, he normally proceeds to develop an alternative that will implement the function of the interface. In the IDM, the steps for creating a new alternative for an interface are similar to those taken when creating a new interface. The following actions are required:

1. create_alternative (interface, var new_alternative): An instance of a new alternative object is created in the database. The interface that defines the alternative must be specified. Initially, all of the attributes contain null values. A null version of the alternative without code, that is, a version that could be called Version #0, is logically contained in the alternative.
2. edit_alternative (var new_alternative): The designer enters the alternative editor and sets desired values for the alternative attributes. These values will define the performance attributes of the alternative and, as in for interfaces, serve as search indices during the operation search_for_alternatives. For example, the designer sets:

   new_alternative.name := Binary_method,

   new_alternative.time := O(log n),

   new_alternative.space := O(n),

   new_alternative.component := I/O manager,

The designer may also enter administrative data such as the current date, his name, and any other information stored as part of the alternative header.

3. display_alternative (new_alternative): In order to check the values of all the new alternative attributes, they are displayed for the designer's inspection. At the conclusion of this operation, the designer may commit the new alternative to the database or abort the operation, thereby destroying the newly created object.

3.5.3.6 Action 6: Developing Similar Modules

There are times when an interface or alternative is found that is very similar to the one desired, and will suffice in the current application after some small modifications. In this case, it is beneficial to be able to copy the existing objects, edit them, and then save the changes as a new object. The actions required are similar to those in Actions 4 and 5, respectively. However, the first step in each is replaced with:

1. copy_interface (current_interface, var new_call): Creates a new instance of a call object, then copies the attribute values of the current interface into the new call object. The interface name, since it must be changed, is not copied. The designer then treats the new call as in Action 4.

or...

2. copy_alternative (current_alternative, var new_alternative): Similar to above, except that the interface representing the current_alternative will also be the interface representing the new alternative. This enforces the constraint that new alternatives are defined by the same interface that defines the alternative they were copied from. The designer then treats the new

alternative as in Action 5.

3.6 Relationship of the IDM to Object-Oriented Program Design

One of the most important aspects of the object-oriented design paradigm is the encapsulation of certain parts of the design in order to hide implementation details from the user [Boo84, Pyl81, Wir85]. However, care must be taken so that during the design process certain key data remains accessible. Much of the information that the designer requires in order to (i) retrieve appropriate design objects from the database, and (ii) select the best alternative implementation for his needs, is specific to the implementation and is not available by looking solely at the interface for that object.

The data model above reflects the belief that, although one part of the object's interface is, indeed, composed of the externally visible attributes common to all its implementations, the *complete* interface must provide access to *all* the externally visible attributes of a module, even those which are implementation-specific. The primary examples of such attributes are performance characteristics that may influence a designer's decision to use that object, such as those required in order to conform to a set of design constraints. Other such attributes include memory requirements, object code size, and the time complexity of the algorithm. In the VLSI domain, such attributes include surface area, power consumption, and delay time.

In some respects this belief conflicts with the object-oriented paradigm because it extends what the user sees of an object to some aspects of the object's implementation. However, closer inspection reveals that the true difference in this approach is in what is considered to be the role of a software interface. The interface must not only provide access to the common attributes of the objects it represents, but also allow the user to distinguish among available alternatives. If some criteria that may effect a decision to use a module is only available by running the module, the interface is not completely doing its job. The key point is that such performance attributes *are* visible to the user,

although perhaps indirectly. Therefore, this definition of the role that an interface plays in the design of programs is not so much in conflict with the principle of information hiding as it is a clarification of something not previously recognized by object-oriented programming advocates.

4. CAPTURING DESIGN INFORMATION IN A CASE SYSTEM

4.1 Introduction

Current CASE systems incorporate one or several of the well-known software design methodologies into a set of tools used for the design of programs. These tools take the form of interactive graphical diagramming aids in which the designer develops the software product in a pictoral form according to the conventions of the particular methodology that he is using. It is then up to the CASE system to extract the necessary program design information from these diagrams and store this information in a meaningful internal representation.

The motivation for the use of pictures in the software process is the familiar adage that "a picture is worth a thousand words." Design diagrams help people to experiment with design ideas, helping the human reasoning process by providing an alternative view of textual specifications or programs [Buh89].

Complications arise when the internal database representation of the program design information does not easily correspond to the diagramming method used by the designer. In such cases it is necessary to infer certain information required by the model from the diagrams. However, while these deductions may help to complete the description of the design in the database, they may also lead to errors by introducing invalid assumptions about the program and intent of the designer. It is desirable, therefore, to semantically store program design data in a form that directly maps to the major software engineering methodologies and tools used to develop software.

In order to fully understand what these tools are and what design information is available from them, a survey of the major software engineering methodologies is conducted. This survey is divided into those methodologies commonly used for high-level conceptual software design and those methodologies used primarily for low-level design and programming. The goal of this study is to determine if there is some characteristic

that binds these various diagramming techniques together. If such a characteristic exists, then (1) an appropriate internal representation can be found that easily corresponds to this characteristic, and (2) an optimum software design diagramming technique can be identified or developed for the efficient capture of program data. This chapter concludes with the presentation of a new technique for the capture of design data in CASE systems that is powerful in its expressive ability and that is tied closely to the common aspects of the major software diagramming techiques.

4.2 High Level Design Methodologies

4.2.1 Functional Decomposition

Functional decomposition, modular programming, and top-down design all variations of the very strong structured programming movement that followed the landmark article by Dijkstra titled "Programming Considered as a Human Activity" [Dijk65]. This article, among other things, expounded upon the divide-and-conquer approach to solving programming problems. All of the members of this genre depend on the stepwise refinement of a problem based on functional requirements. The result is progressively smaller problems that eventually can be solved with just a few lines of code. The basic graphic tool in this design method is the tree diagram, with rectangles at the nodes representing an abstract function or problem to be solved. Children of a node are subproblems of that node, and lines to subordinate rectangles imply that the superordinate problem can be decomposed into the sub-problems below it.

The major considerations when doing Functional Decomposition are those of cohesion and coupling. Cohesion is the degree to which a module does one unique task. Modules that perform more than one task may have several types of cohesion, for example temporal (all actions that must be done at the same time), logical (all actions are of the same type) or the "worst" type, coincidental cohesion (little connection

between the actions [Stev74, p. 208-217]. As designers and programmers, we should strive for the highest possible cohesion. The second consideration is that of coupling, which is the measure of interconnection between modules. For example, modules that are related by the passing of program flow have control coupling; the worst types of coupling are those that bind modules to the environment (common coupling) or to the contents of other modules (content coupling). One of the lowest forms of coupling, data coupling, is when modules are related only by the passing of data; we should obviously strive for the lowest amount of coupling possible. A program that is the result of proper functional decomposition should achieve both of these requirements.

4.2.2 Data Flow Design

Data flow design is an analysis of the transformations made to information as it moves through a system. In its purest form it is functional decomposition applied to data. The key consideration is that the system may be thought of as being composed of information that is in a continuous "flow," undergoing a series of operations as it evolves from input to output. [Pre82, p. 178] Data flow diagrams (DFDs) are the graphical tool that depict this flow and are, for those familiar with it, basically a network representation of the system [Myer78, p.47]. The diagrams consist of arcs representing data flow and bubbles representing data transforms. See Figure 4.1.

The data flow design process, as well as several modifications to the method (structured design, composite design, SADT), have been well defined by Myers, Yourdon, and Constantine, and consists of the following [Pre82, pp. 182-192]:

1. Review the model by studying the system specification and requirements.
2. Construct and review the data flow diagrams for the software.
3. Identify the main transformation "center" of the diagram. Data incoming to this center is called "afferent;" data leaving this center is called "efferent." Delineate this transaction center on the DFD.

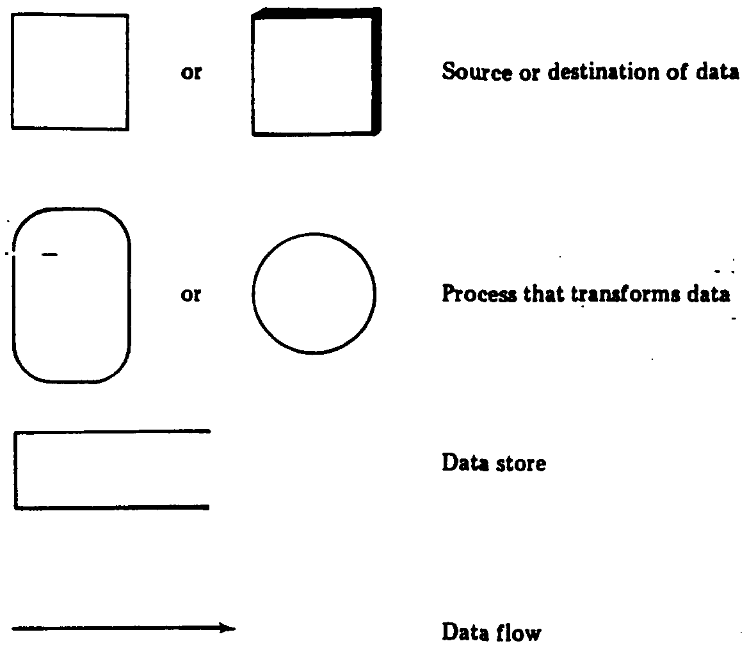


Figure 4.1. Data Flow Diagram symbols

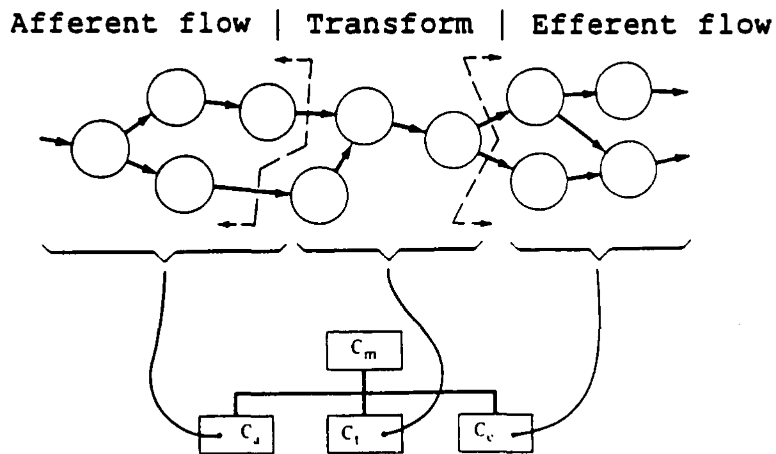


Figure 4.2. First-level factoring

4. Perform first-level factoring. That is, consider the system as being composed of the afferent flow, efferent flow, and transaction center. See Figure 4.2.
5. Perform second-level factoring. That is, decompose the bubbles in the DFD into more detailed, refined DFDs. In this manner, "levels" of diagrams are created, with each subsequent level encompassing more detail in the system. See Figure 4.3. The final system will be composed of modules that are mapped from the lowest level DFDs.

Although many systems lend themselves nicely to data flow design, it is conceded that this method loses it's usefulness as a low-level design tool since no representation for iteration or selection is provided. The DFD is powerful in its ability to quickly convey the general activities of a system, but without a means for explicitly generating loops and condition statements the diagrams cannot give the programmer the detail necessary for generating unique and correct data structures and control statements.

Stevens, Myers, and Constantine [Ste89], have developed a popular data flow technique that aims itself at getting the designer directly to a program structure. After the afferent, efferent, and transform center of the data flow diagram are identified, these are mapped into a tree diagram similar to that of the functional decomposition technique. In fact, the process they describe is, from this point on, very similar to functional decomposition. They do, however, include in their method a distinct diagramming notation for the program structure (Figure 4.4), as well as a method for including the interface information as part of the structure diagram. Each arc in the Structured Design diagram is labeled and has the parameters associated with the arc in a corresponding box elsewhere in the diagram. A sample template for a module interface form is shown in Figure 4.5.

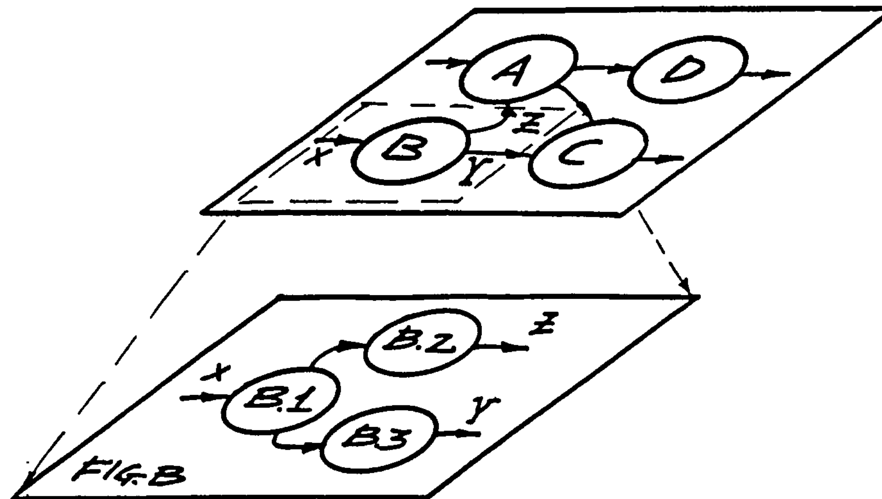


Figure 4.3. Leveling

4.2.3 Data Structure Design

4.2.3.1 The Jackson Method

Data structure design is based on structuring the program to reflect the input and output data structures. The major proponent of data structure design is Michael Jackson, and the design process and diagramming conventions that are discussed here are derived from his work [Jac75]. Data structure design is used mostly in business applications where the data structures (commonly personnel and financial records) are well defined. The concept is that "paralleling the structure of input data and output (report) data will ensure a quality design" [Pre82, p. 206]. Like the data flow design method, the data structure method has a diagramming convention to assist as a graphical tool. However, the data structure diagram (DSD) follows more closely the style of the traditional tree diagram, with rectangles representing atomic data units and lines representing the "is composed of" hierarchy in the tree.

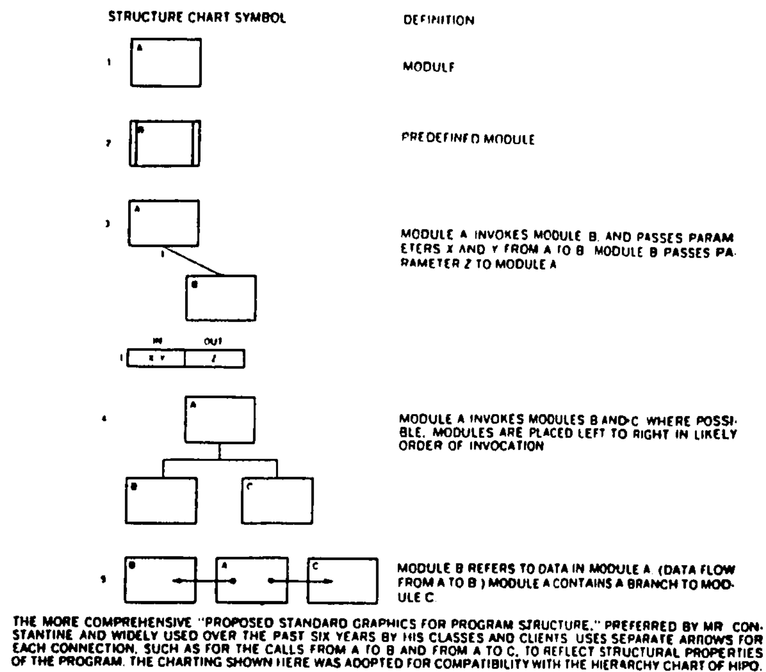


Figure 4.4. Structured Design Notation

There are two points on which the data structure method is significantly different from the data flow method. First, the data structure method moves quickly to a procedural domain; the emphasis is on producing a software framework and of outlining that framework as early as possible in the design process. The data flow approach does not have this emphasis, and as such has a more definite boundary between the stages of high-level and low-level design. Second, the data structure method has constructs to represent the three major types of program statements of sequence, selection, and iteration, as shown in Figure 4.6 [Pre82]. The circle in the upper right of a node indicates the data component is conditional, for example, an employee may or may not be married. An asterisk in the node indicates iteration, for example, the employee may have

INTERFACE CHART			Program:	Date:	Page ___ of ___		
Inter-face no.	Calling module	Called module	IN	OUT	Coup-ling	Type of call[1]	Prob of call[2]

1: I = Iterative, N = Not iterative
2: Probability that when the calling module is entered, it will invoke the called module

Figure 4.5. Module Interface Form

any number of children. An absence of either symbol indicates sequence, in that the data record is constructed of the data components as taken in a left-to-right order. This convention allows the data structure diagram to be translated nicely into high-level code.

The process of data structure design is defined as follows by Pressman, and is illustrated with the following figures from the text by Yourdon [Pre82, p. 207]:

1. Data structure characteristics are evaluated (Figure 4.7).
2. Data are represented in terms of elementary forms. such as sequence, selection, and repetition. The correspondence between the input and output data structures is noted (Figure 4.8).
3. Data structure representation is mapped into a control hierarchy for software

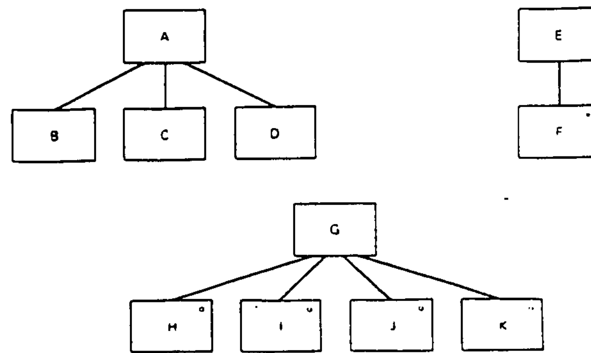


Figure 4.6. Data Structure Diagram Notation

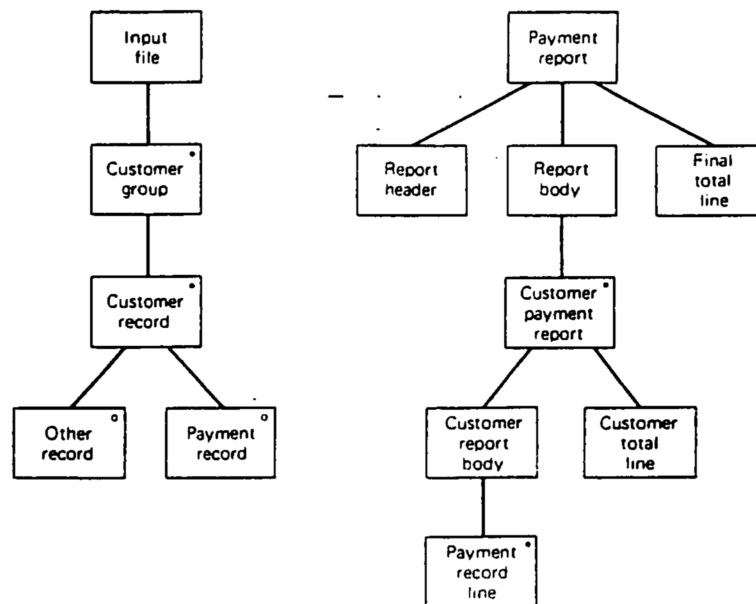


Figure 4.7. Input and Output Data Structures

(Figure 4.9).

4. The software hierarchy is refined.

5. A procedural description of the software is developed.

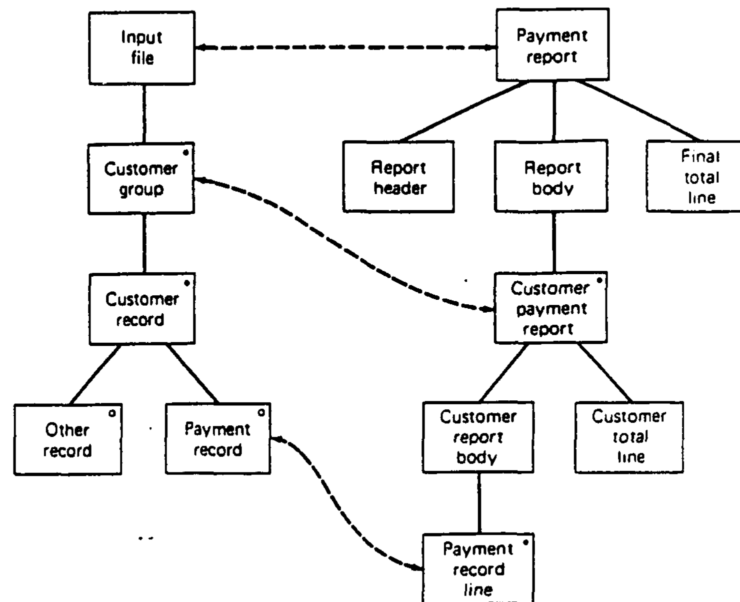


Figure 4.8. The One-to-One Correspondence

4.2.3.2 The Warnier Method

Another example of data structure design is the Warnier/Orr diagram, which is very popular in Europe, and particularly in France where it was developed [Dav83]. Shown in Figure 4.10 [Pre82], this methodology is similar to the Jackson method, differing primarily in the diagramming style. There is therefore a close correspondence between a Jackson diagram and a Warnier/ Orr diagram of the same system, and one can certainly derive either type of diagram from the other. There is also a method for mapping Warnier diagrams to flowcharts, known as the "logical construction of programs," which provides a step-by-step process to follow in going from a Warnier/Orr diagram to high-level code.

The data structure design methodology can be summed up as follows: "The process of finding one-to-one correspondences between the data structures is

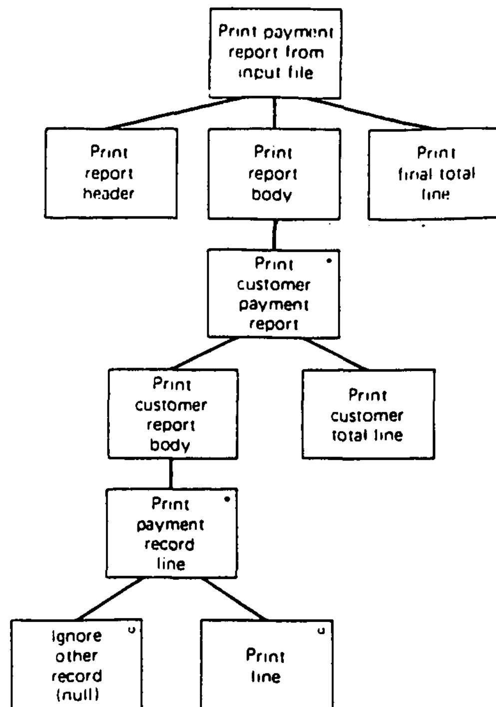


Figure 4.9. The Resultant Program Structure

fundamental to (the data structure) technique" [Myer78, p. 95]. Consequently, the major problem with this or any data structure design method is in applying it to applications where there are not well-defined data structures, or in situations where the input data and the output data are very dissimilar.

In the former case, simply starting the data structure methodology is difficult. In the latter case, we have what is known as a "structure clash." Jackson's solution to this situation is to design an intermediate data structure that can act as a bridge between the input and output data representations. Jackson then proposes to use the data structure methodology twice, once to decompose the input data to the intermediate data structure, and then to map the intermediate data to the final output data structure. It is probably best, however, to simply use

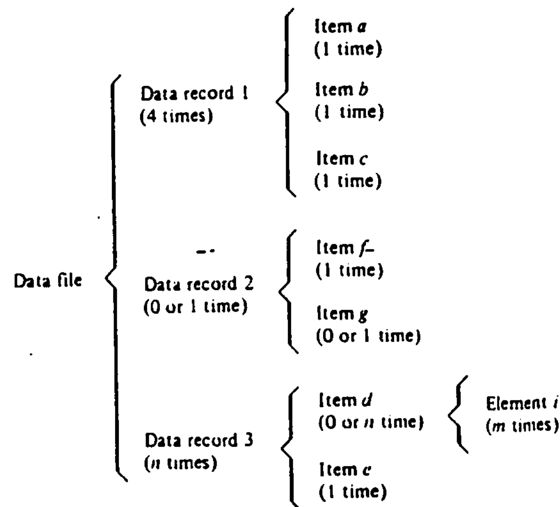


Figure 4.10. A Warnier/ Orr Diagram

a different design method when one is faced with this situation.

4.2.4 Object-Oriented Design

Object-oriented design is the paradigm that is gradually challenging functional decomposition for the premier design methodology in industry. The method considers each data structure as a separate entity, and encapsulates this entity in a separate software package along with the procedures that are allowed to operate on it. When the data entity and the procedures are taken together in this fashion, they are considered an "object." The idea is that other routines are able to use the data and the operations in a black-box fashion, but are unable to see the details of their implementation, thus achieving a high level of abstraction, maintainability, etc.

In literature by Grady Booch, who is one of the major advocates of this method and the programming language Ada, Object Oriented Design is described as follows [Boo86]:

1. Write down (in paragraph form) the problem statement.

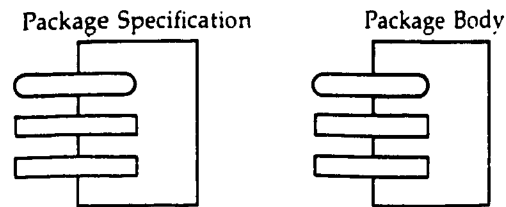


Figure 4.11. Object-Oriented Design Symbols

2. Underline the nouns in the problem statement.
3. Underline the verbs in the problem statement that are associated with each noun.
4. Map the nouns to data objects and the verbs associated with each noun to procedures. Each collection of data objects and procedures is "packaged" together.
5. Repeat the process as necessary to realize the actions of each "verb" until the appropriate level of detail is achieved.

The diagramming notation proposed by Booch is very simple and is capable of quickly conveying how the data objects and their operations are grouped as abstract data types. This grouping is called a *package* in the programming language Ada, and is shown in Figure 4.11. References between ADTs are pictorally represented by drawing lines from one package diagram to the other; however, as shown in Figure 4.12, these lines do not specify the type of interaction between the ADTs, and no provision for the three programming constructs is made. It should also be noted that in addition to the diagramming method proposed by Booch, there is the Buhr diagram, which is a proposed standard for Ada programs. Buhr diagrams, however, will not be discussed here [Buhr84].

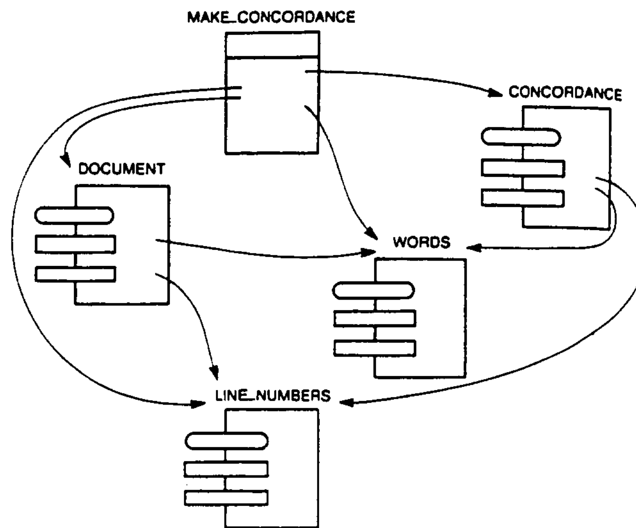
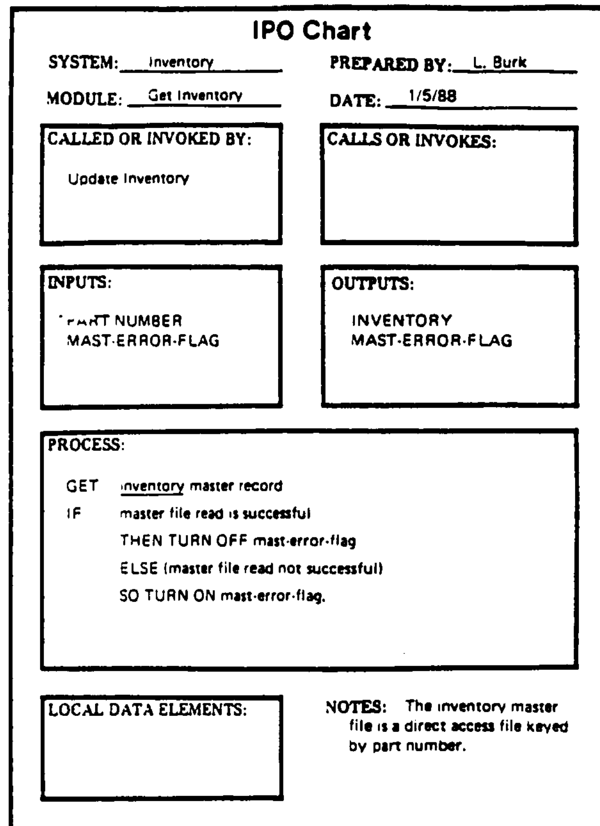


Figure 4.12. An Object-Oriented Design Diagram

4.2.5 IPO Charts

IPO (Input/ Process/ Output) charts are forms that summarize on one page the important aspects of a module, such as the data it inputs, outputs, who calls it, and who it calls. The IPO chart also provides a general description of the module as well as important management information, such as the person responsible for the module. An example IPO chart is provided in Figure 4.13 [Dav83]. By itself, the IPO chart provides an excellent summary of a single module for reports and design reviews, but is unable to quickly convey the relationship between modules. For this reason, they are often accompanied by a structure chart such as one used by the functional decomposition methodology. In this case the name IPO becomes HIPO, for Hierarchy plus Input/ Process/ Output. The IPO charts then serve to complement the functional decomposition diagram in a nice way, and perform much the same task as the module interface form does for the structured design technique.



IPO Chart

SYSTEM: Inventory
PREPARED BY: L. Burk
MODULE: Get Inventory
DATE: 1/5/88

CALLED OR INVOKED BY:
Update Inventory

CALLS OR INVOKES:

INPUTS:
PART NUMBER
MAST-ERROR-FLAG

OUTPUTS:
INVENTORY
MAST-ERROR-FLAG

PROCESS:
GET inventory master record
IF master file read is successful
THEN TURN OFF mast-error-flag
ELSE (master file read not successful)
SO TURN ON mast-error-flag.

LOCAL DATA ELEMENTS:

NOTES: The inventory master file is a direct access file keyed by part number.

Figure 4.13. An IPO Chart

One advantage of IPO charts is that they could complement any methodology just as well as they do functional decomposition. They provide a standard format for presenting information about a module, independent of the design method being used. Assuming a program design exists in a database, the IPO chart could be completed for documentation or report purposes, for the most part by only reformatting existing data or by subjecting the database to queries for the needed information.

4.3 Low Level Design Methodologies

4.3.1 Introduction

High level design methods are aimed primarily at establishing a modular structure for the software that is going to solve the problem at hand. At some point in the design process, as the refinement of the problem proceeds, it becomes necessary to address the procedural details of the modules that have been established. For this task, low level design tools are needed. Typically, high level tools lack the detail needed to express single source code statements or sequence of actions. Likewise, the converse is true of the low-level tools in that they tend to provide too much information for a general overview of a large problem.

4.3.2 Standard Flowcharts

Standard flowcharts, as shown in Figure 4.14 [Mur75], were perhaps the very first graphical tool used to assist programmers in visualizing their programs. They have the advantage that they are readily understandable, even by persons untrained in their use. The symbols are simple and translate easily to code, often with a one-to-one correspondence.

The major problem with flowcharts is that they are, in a sense, relics of a dying programming style associated with FORTRAN programs. They allow arbitrary transfer of control in a program, something common in old FORTRAN code, but considered sinful by structured programmers. They also have no means of representing recursion, a powerful programming technique not available in FORTRAN but widely used today for certain classes of problems. However, the standard flowchart is still popular because of its simplicity and its powerful way of immediately conveying to the reader a sequence of actions.

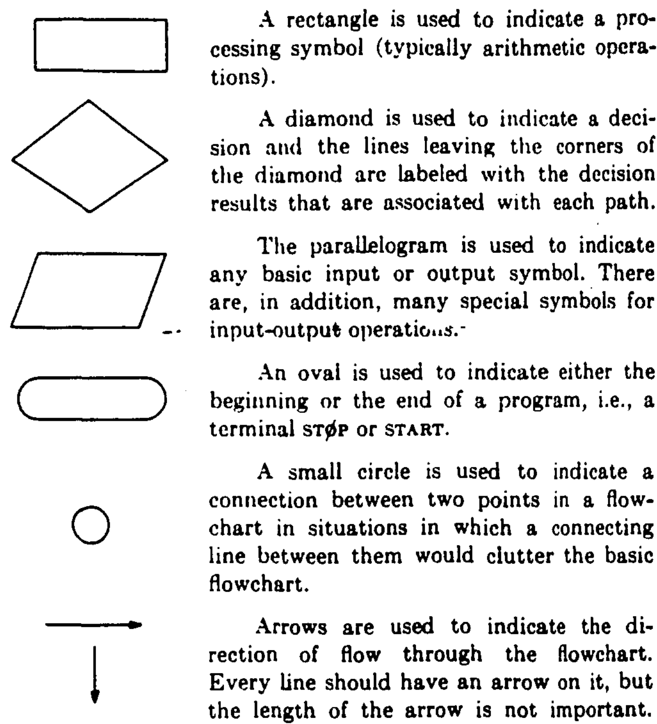


Figure 4.14. Standard Flowchart Symbols

4.3.3 Structured Flowcharts

The structured flowchart was developed to be the answer to the problems with the standard flowchart that were discussed above. Another name for structured flowcharts is the Nassi-Schneiderman diagram, named for the authors that introduced the notation in 1972. Examples of the structured flowchart's symbols are shown in Figure 4.15, and an example of how a code segment would be diagrammed in this notation is shown in Figure 4.16 [Yod83].

In the Nassi-Schneiderman diagram there are constructs for the basic operations of sequence, selection, and iteration. Each construct has exactly one entry and one exit, thereby disallowing arbitrary transfers of control. However, they have never achieved

the wide acceptance of the original flowcharts, possibly because they do not convey the flow of control in a module as quickly as do standard flowcharts. Another problem is that because each statement is represented by a box that is nested inside another box, in large sections of code, or in small sections that involve a lot of conditional branches, the space available for flowcharting may shrink very rapidly. This is a serious problem for a person working on paper, although much less of a limitation for one using a graphics computer with the ability to expand the work area by "zooming in" or windowing. Like standard flowcharts, a properly completed structured flowchart may be translated directly into code [Yode78].

4.3.4 Finite State Machines

A special class of problems may be represented by the actions of a finite state machine. For example, the lexical analyzer of a compiler may enter different states depending on whether it is scanning character symbols or numbers, or whether it has recognized a reserved word or programmer-defined symbol. Each new input character specifies a transition to a new state and possibly an action to be completed upon entering that state, such as outputting a complete token to the parser. For such problems, a finite state machine representation like that depicted in Figure 4.17 [GE87] is ideal.

In this diagram, the circles represent a state that the machine may be in, and each arc represents an allowable transition to a new state. Associated with each arc is a set of conditions that must be met for the transition to take place. This representation has the advantage that it, too, may be translated directly into code, although it has a major drawback in that the class of problems lending themselves to this type of analysis is quite limited.

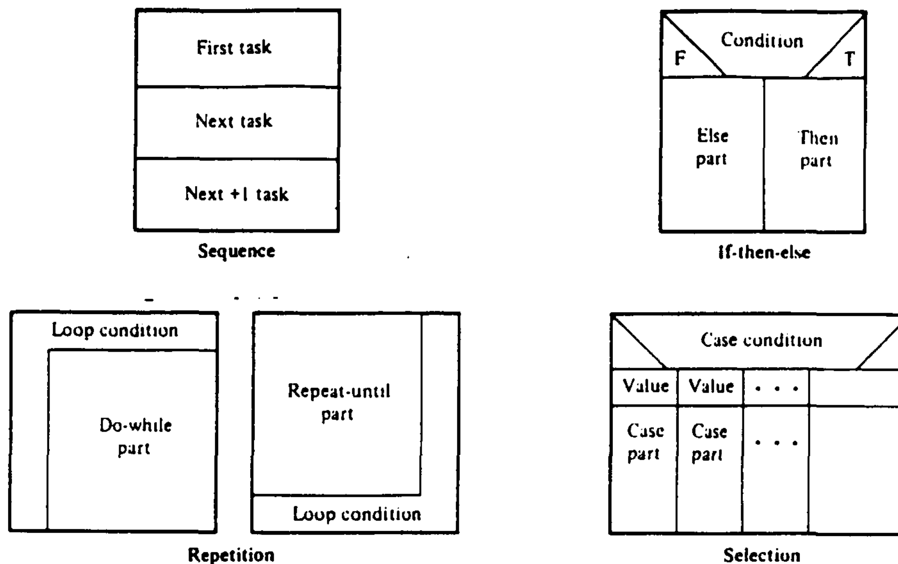


Figure 4.15. Structured Flowchart Symbols

4.3.5 Decision Tables

Certain types of problems that require multiple nested decisions are difficult to describe using the above methods. When a set of actions must be chosen based on a complex set of conditions, a tabular format as in Figure 4.18 may be easiest to understand [Lond72].

The numbers across the top of the table are rules, and represent the state that the all the conditions must be in for some action to occur. The following procedure is defined for developing a decision table [Pre82, p. 250]:

1. List all actions that can be associated with a specific procedure or module.
2. List all conditions or decisions that must be made during the execution of a procedure.
3. Associate specific sets of conditions with specific actions, or else develop every possible permutation of conditions.

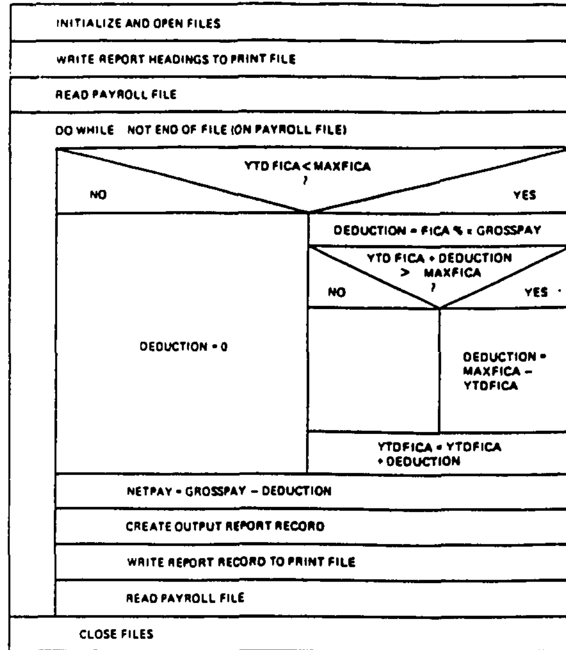


Figure 4.16. Structured Flowchart Example

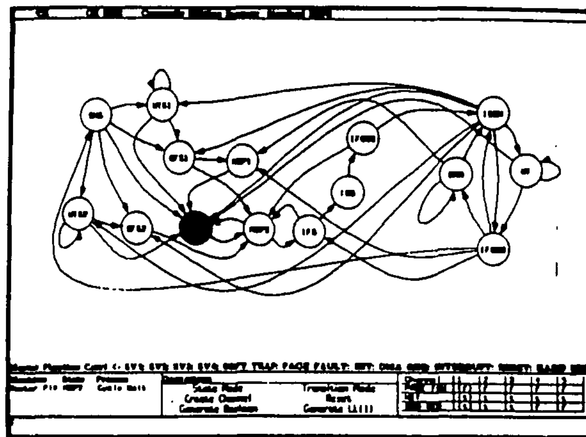


Figure 4.17. A Finite State Machine

	Rule numbers →	1	2	3	4	5
Conditions	Fixed rate account	T	T	F	F	F
	Variable rate account	F	F	T	T	F
	Consumption < 100 KWH	T	F	T	F	
	Consumption ≥ 100 KWH	F	T	F	T	
Actions	Minimum monthly charge	X				
	Schedule A billing		X	X		
	Schedule B billing				X	
	Other treatment					X

Figure 4.18. A Decision Table

4. Define the rules by indicating what action(s) occurs for a set of conditions.

The decision table may be directly coded much the same way as the other low level design representations can, since the table basically represents a giant "CASE" statement. However, the table by itself has no means for representing sequence of execution or iteration. Therefore, its usefulness is, like the finite state machine, somewhat limited.

4.4 Mapping the Design Methodologies to Program Structure

4.4.1 Introduction

The net result that each of the software engineering methodologies seeks to achieve is a viable software product. This software product consists of modularized program code depicting the flow of control in the software product. It is also important to note that, with the exception of the object-oriented design method, there is a distinct bias towards programming in a structured target language such as PASCAL or PL/I. Therefore, the following discussion details the transformations that a CASE system must make in order to capture this kind of control information from the various diagrams that may be used for program design. The goal of this discussion is three-fold.

First, this bias towards structured target languages leads us to believe that program design data should be captured in such a way that emphasizes the flow of control in programs. With this in mind, we seek to find the "best" method for capturing program design data, and eventually will present a new technique for this purpose. Second, understanding how these diagramming techniques interelate will prove important in developing a "standard" form for efficiently and effectively storing program design data. Third, by showing how each of the major methodologies maps to this standard form we can validate the usefullness of storing program design data in such a manner.

4.4.2 High Level Design Methods

In contrast to the low level design methods, where all of the diagramming conventions basically depict the flow of control, the diagram in the high level method may depict one of several things. It is important to remember, for example, that the DFD represents data flow, not necessarily any type of control mechanism. Likewise, the DSD is a pictorial representation of the input and/or output data structures, and by itself does not show how program control is handled. It is easy to loose sight of this important difference and interpret all of the diagrams as a type of flowchart, which, of course, is not correct. The following paragraphs discuss in more detail the conventions for mapping the various design methods to the flow of program control.

4.4.2.1 Data Flow

As revealed in a previous section, the data flow diagram has no convention for expressing the basic programming building blocks of selection and iteration. In addition, the idea of algorithm sequence is not well represented since the DFD may show multiple flows of data that logically could be happening in parallel or independent of one another. In problems where there happens to be some form of sequence implicit in the algorithm and this sequence is not obvious from the

direction of the data flow arrows, it is *convention* to assign the upper left of the diagram as the earliest chronological point and the lower right as the latest.

In order to derive the program structure from the DFD, it is necessary to examine each element of the diagram more closely. Since a bubble in the DFD represents a processing step, it could only map to one of two things. These are (i) a subtask (module) or (ii) a basic statement of sequence code. Since the DFD is a high level design tool, the great majority of the bubbles will represent subtasks; only in extreme circumstances would one expect the low-level construct of actual code to be expressed. [7] The arcs, which represent data in motion, are analogous to parameter lists (module interfaces), and an arc from one bubble to another is a call to that module.

The mappings for the remaining two symbols in a DFD, however, are not so well defined. The first of these symbols is the data store. Data stores are, in essence, disk files, so references to them are equivalent to file read or write statements, depending on the direction of the arc in the DFD. Note that these are a type of sequence code and could be inserted as part of the future code for the module. However, one will would have to make non-trivial assumptions as to where they would appear in the sequence of processing. A logical assumption, though not always a correct one, would be to make data reads appear before any processing is done, and data writes appear after all processing is complete.

The final DFD symbol which needs to be addressed is the data source/sink. Quite often this data source/sink is a user who is sitting at a terminal. It may also be some kind of real-time monitoring instrumentation or control device. In any case, the bubble that directly interacts with the data source/sink must have as part of its function any communication activities or protocalls that may be required.

[7] If this were not true, then the DFD would be little more than a special case of the standard flowchart.

This may involve appropriate prompts, error handling for unexpected conditions, or specialized output messages or reports. None of these activities is directly represented as part of the DFD and is probably best included as part of the functional description of the appropriate bubble.

4.4.2.2 Second-Level Factoring

One technique of deriving a program structure directly from the DFD is described in [Pre82] as "second-level factoring." The DFD is completed and the afferent data flow/ central transform/ efferent data flow boundaries are depicted. Then, "beginning at the transform center boundary and moving outward along afferent and then efferent paths, [bubbles] are mapped into subordinate levels of the software structure" [Pre82, p.187]. See Figure 4.19.

Second level factoring could be automatically performed and an initial program *skeleton* created for the user. It has the drawback, however, that the resulting program may need a lot of critical inspection and modification before it is considered "good." Since this final assessment is very difficult to formalize, it would be equally difficult to automatically enforce design decisions implied by the original DFD. Nonetheless, second level factoring is a viable technique for converting data flow diagrams into program flow.

4.4.2.3 Data Structure

The basic technique of mapping the DSD to program structure has been outlined in a previous section. The important thing to note is how the the program structure maps almost directly from either the input or output data structure diagram (in Figure 4.9 the program structure was derived from the output data structure). However, because of the recognition of iteration and selection in the DSD, there are several additional items that can be included in the code of the

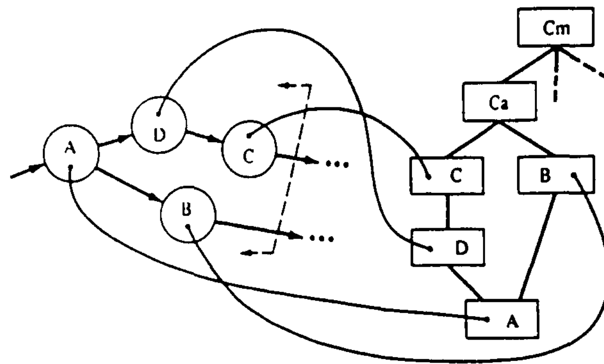


Figure 4.19. Second Level Factoring

modules.

Note the example DSD shown in Figure 4.20 [Pre82]. The DSD implies that the controlling module for PAY_REC contains code to the effect of

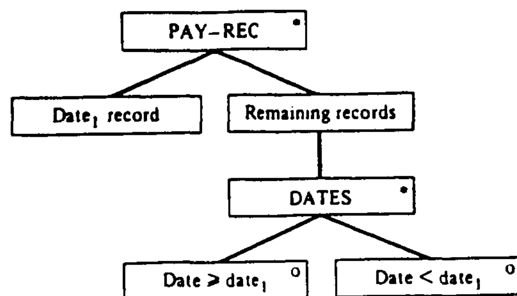
While not (condition) do PAY_REC;

because the diagram symbol for PAY_REC contains an "iteration star." By interpreting the rest of the diagram in the same manner, the remaining code sequences in the figure could be derived and added to the program design automatically.

Of course, as is necessary for second-level factoring of a DFD, a thorough inspection of the resulting software must be conducted after the above process is completed.

4.4.3 Low Level Mappings

As alluded to in the introduction to this section, the correspondence between between low level design methods are much stronger since they all directly or indirectly indicate flow of control. As such, deriving a program skeleton from these methodologies is quickly explained with a short diagram or algorithm. These transformations are well-documented and have been produced in numerous CASE systems.




```
Procedure PAY_REC;                Procedure Remaining_records;
  begin                             begin
  Datel_record;                     While not (condition) do
  Remaining_records;                   DATES;
  end;                              end;

Procedure DATES;
  begin
  if Date >= datel then AFTER;  (* some action routine *)
  if Date < datel then BEFORE;  (* another routine...   *)
  end;
```

Figure 4.20. DSD-to-Code Conversion Example

4.4.3.1 Flowcharts

The major code constructs that the flowchart must represent are shown in Figure 4.21 [Yod83]. Note that the CASE statement, although not strictly required, is included for clarity. The CASE statement is important not only for making more aesthetically pleasing flow diagrams, but greatly assists in the flowchart representation of a decision table.

4.4.3.2 Structured Flowcharts

The constructs for structured flowcharts are shown in Figure 4.15, and, like the constructs for standard flowcharts, map directly to program flow of

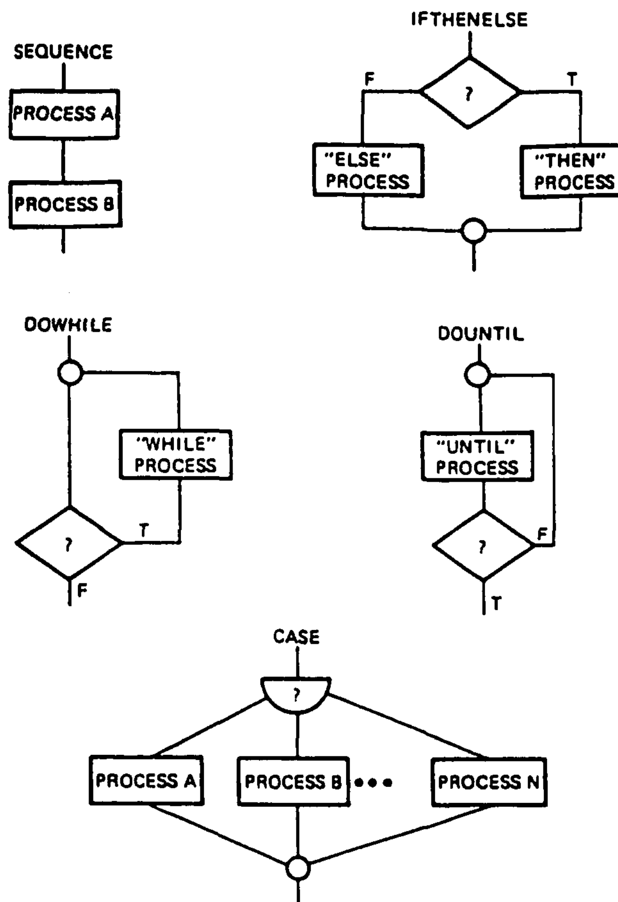


Figure 4.21. Standard Flowchart Code Sequences

control.

4.4.3.3 Finite State Machines

The finite state machine diagram is best referred to by the algorithm in Figure 4.22 [Pou88a]. This algorithm controls the state transitions of the machine based on the inputs to the machine. This algorithm assumes that the finite state machine is coded into a 2x2 array or state table. It is just as allowable to specify each transition directly as a set of nested if...elsif...elsif statements; this latter method is actually required if the state transitions are specified by conditions that

cannot serve as indices into the array. Note that the following algorithm consists solely of the basic code constructs that have been defined above.

4.4.3.4 Decision Tables

A decision table is quickly converted into an algorithm by recognizing that each rule can be associated with a clause of a CASE statement. A rule is determined to be true or false depending on the state of the condition variables in the top part of the decision table. Once this is done, a decision table is translated to into the CASE statement, just as the example in Figure 4.23 [Pou88a] has been derived from the table of Figure 4.18.

In this example, RULE is the number of the rule that was determined to be true. Note that we have set up the rules in the table so that they are mutually exclusive; ie, only one may be true at any time. If we relax this restriction then we must use the alternate if...then...elsif form of the CASE statement.

4.5 An Approach to Design Data Capture

4.5.1 Introduction

It has been shown how design diagrams used by major software engineering methodologies map to a program skeleton depicting the flow of control in a final software product. This is the common goal of the many diverse methodologies surveyed, just as it is the goal of the designer to produce a working software product from the initial problem statement. It can be concluded that this requirement for a final code representation is a common thread that binds the methodologies together. Using this common thread as a basis for the design of a data model for the capture of software design information promises to result in an efficient and conceptually elegant tool. Therefore, the premise of this chapter is that it is desirable to capture program design information in a manner as close to this representation as possible.

```
Current_state := Start_state;
While not (an accepting state) or (finished) do
     Current_state := State_Table(Current_state, Input);
     Case Current_state of :
      { perform task associated with the current state }
```

Figure 4.22. Code for a Finite State Machine

```
Case RULE of:
     1 : Minimum_monthly_charge;
     2 : Schedule_A_billing;
     3 : Schedule_A_billing;
     4 : Schedule_B_billing;
     5 : Other_treatment;
   end;
```

Figure 4.23. Code for a Decision Table

However, a conflict arises in two situations. The first is when a particular methodology, for example, the data flow method, is mandatory or prefered for use in a given application or environment. The second conflict arises when the designer has not yet developed the program design to a point that he can start to think or work with a flow-of-control representation. In these situations any of the above methodologies can be utilized. It is simply necessary to provide the tools to convert the diagramming tool that is used into a structured program format as we have shown in this chapter.

This section first discusses early work on program design diagramming techniques. As part of this presentation, the shortcomings of the different approaches are explained. Finally, a data capture technique developed for use with the IDM is introduced.

4.5.2 The Program Static Structure Diagram

Initial work on capturing program design data centered on a tree-style diagram that depicted how a program would be *declared* in a highly-scoped language such as PASCAL. This representation, called the Program Static Structure **(PSS)** Diagram, closely resembles the Functional Decomposition software engineering methodology in that it progressively reduces problems by dividing them into subproblems of manageable size. In the PSS diagram, a node in the tree corresponds to a module that solves some specific problem.

The information contained in a PSS diagram is sufficient to outline a program as it appears "at rest," so a skeleton of the module *declarations* can be automatically produced. The diagram was originally intended to be used to enforce scope rules for data types, variable declarations, and module calls. It works well with the Software Module as a Static Object data model discussed in an earlier chapter.

The shortcoming of this type of diagram is that because it only depicts the module declarations in the program, there is no way to represent the flow of control. Furthermore, it is of little use to have the ability to check the scope of module calls in a diagram when there is no way to represent such a call. Another tool is clearly needed.

4.5.3 The Program Dynamic Structure Diagram

Changing the meaning of the arcs in a tree diagram to mean *module calls* rather than *is declared by,* allows flow of control to be represented. In addition, since these module calls are basically lines of sequential code, by adding the data structure diagram notation for iteration and selection, we can allow the designer to develop a program all the way down to the pseudocode level by using this kind of diagram. However, the diagram is still very useful at a high level of design, because the lack of any symbol for iteration or selection as might be expected at such high levels does not restrict the interpretation of the diagram. The net result is called the Program Dynamic Structure

(PDS) Diagram. Using the mappings shown in Figure 4.24 [Ber85], the flow of control in the target program can easily be captured from the PDS diagram.

Like the PSS Diagram, the PDS Diagram has a shortcoming in that while it does a great job of pictorally representing the dynamic activities of the target system, it does not have the ability to represent how modules in the program are declared. Of course, a tool can be created that automatically derives an optimal declaration structure when the dynamic structure is complete, but it is often helpful to have the ability to specifically view the program's declarations during design [Rov88]. For this reason, the PSS Diagram as well as the PDS Diagram, or their equivalents, are deemed necessary for data capture in a CASE system.

4.5.4 Data Capture with the IDM

The method of data capture adopted for the CASE prototype includes both the PSS and the PDS diagrams. The diagrams are closely related in the application system in that changes made in one editor that effect the view in the other are automatically made by the CASE system. This combination of PSS and PDS diagrams gives the most powerful representation of the program design, and retains the designer's ability to customize each type of program structure to his requirements.

The IDM is structured to reflect the flow of control in the target system. The foundation of the IDM consists of linking calls made by a module to candidate modules in the archive that are capable of filling the need of that call. Using the mappings shown above from the pictoral representation of the program design to pseudocode for the target system, the IDM is able to capture control data directly from the PDS diagram and map it to the data model in the database. The pseudocode is then stored in the alternative object of the IDM as a sequence of calls. Data related to the declaration structure is likewise captured from the PSS diagram and stored in the alternative object.

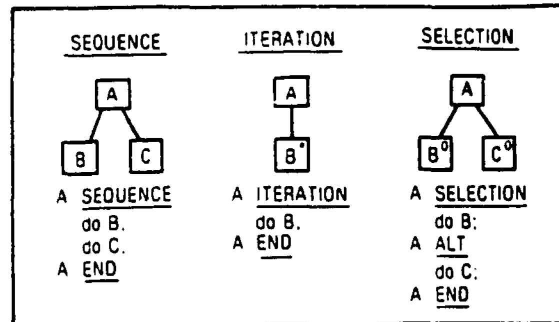


Figure 4.24. Symbols for Sequence, Iteration, and Selection

Figure 4.25. Icons for Calls, Interfaces, and Alternatives

The PDS diagram further contains an accomodation that provides for a meaningful representation for the three types of objects that comprise the IDM. In Figure 4.25, the icons representing these three object types, the call, the call with an interface bound to it, and the call with both a bound interface and bound alternative, are shown. When the program designer encounters a need for some service in a program, he represents that need by creating and placing the call icon in the PDS editor. As the abstract request for service is further developed, the designer either locates an existing interface in the archive to meet that service, or he develops an

interface of his own. At that point, he may *bind* the interface to the call; in the PDS editor this is reflected by replacing the call icon with the interface icon. Finally, when an alternative for the interface is decided upon, and, in turn, bound to the call, the PDS editor replaces the interface icon with the alternative icon. Notice that the version of the alternative that is currently in use is identified in the diagram.

An example PDS diagram representing a partially completed mail facility for the MTS operating system is shown in Figure 4.26. The actions of the call in the root node is fully defined, as version #2 of the "MTS" alternative for the "Mail" interface is bound to the call object. The are three lines of code in that alternative:

```
Call Asynchronous Input;
Call Sort?
While (not buffer_empty)
   Call Output;
```

The call in the first line of code is filled with version #0 of the "Asynchronous" alternative of the interface "Input;" the code portion of that version contains:

```
While (incoming_messages)
   Call Read?;
```

The iteration gaurd, "incoming_messages," is a Boolean variable in the scope of the alternative that has been specified by the user. The second line of code of "MTS Mail" is to a call named "Sort?" which has neither a bound interface nor bound alternative. The Kleene star marking the third line of code in "MTS Mail" indicates another

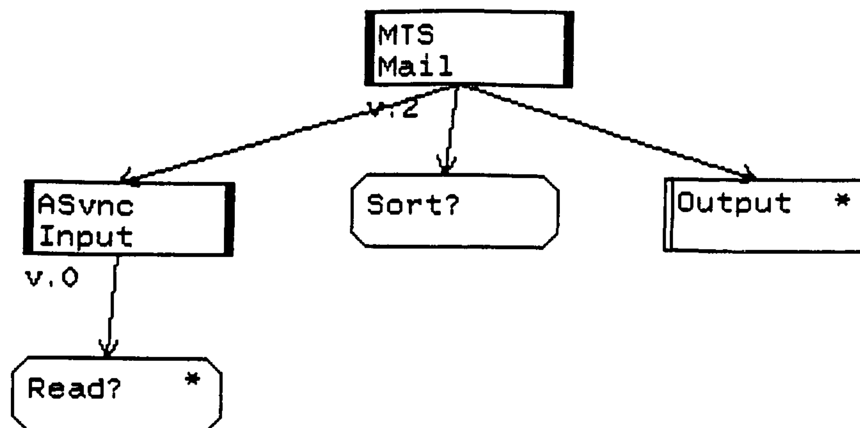


Figure 4.26. Example PDS Diagram

iteration statement. This iteration gaurd, as above, is a Boolean variable that has been specified by the user. The call made in this iteration currently has the "Output" interface bound to it.

Because the PDS and PSS diagrams so closely model the types of objects represented by the three part IDM, the process of extracting design data from these diagrams is straightforward and efficient. Furthermore, all that is required to fully update the designer as to the progress of his program design is a glance at the appropriate diagram. This ability to quickly convey large amounts of information in a single diagram is the quality of *visual depth,* and is a major distinguishing feature among visual methods [Buh89]. Visually deep methods provide most of the information through shapes and relationships in the pictures. Visually shallow methods provide most of it in related text or in such a distributed fashion that its impact is lost. A block diagram in which boxes are only used for partitioning is visually shallow; the PDS diagram (and its accompanying PSS diagram) are visually deep becuase they use

shapes and spatial relations to effectively achieve the desired affect.

5. CLASSIFICATION OF SOFTWARE COMPONENTS

5.1 Introduction

Once the software design has been captured in the CASE system, it is necessary to store the information in a manner that allows the information to be retrieved quickly and efficiently. This is especially true in a CASE system that supports the reuse of software, since the designer would like to know as early as possible in the design process whether or not the component he needs already exists. If this is so, he can substitute the component in his program without having to develop it himself.

Organizing available software routines into a library of reusable components may provide the program designer with the tools and opportunity to incorporate "old code" into his design. However, central to the problem of designing an appropriate catalog of reusable components is the problem of rigorously specifying the allowable uses of each component. That is, the capability to specify the class of contexts into which a given component can meaningfully fit, and specifying the kinds of components that can fit within a given context.

Classification is the act of grouping like things together. All members of a group, or class, share at least one characteristic that members of other classes do not possess. In this way, classifying objects displays the relationships among classes of things. A classification scheme, therefore, is a tool for the systematic ordering of things for the purpose of displaying relationships among them. These relationships determine where the things are stored and how they are later retrieved.

With this in mind, several observations and suggestions for a systematic ordering of software components have been proposed. However, The issue of software classification is particularly complex. Unlike VLSI hardware components, where the function of a component is fairly specific, in software there is an added difficulty involved in classifying a component due to the overall ambiguity and generality of

software modules. There has been success identifying simple math and I/O routines for software catalogs. But while utility functions and some ADTs are well understood, at higher levels of design a module is composed of more abstract ideas and complex algorithms, and may have many intermingled functions and side effects.

Most software retrieval techniques depend on the use of software *keywords* to classify the software component and serve as indices for retrieval. [1] These keywords provide both a broad-based description and a formal specification of the module in an attempt to define the uses of the module for an appropriate application. The identification of a software classification mechanism that is both suitable for indices in the software archive and is usable in a viable software retrieval strategy is the subject of this chapter. The following discussion is an overview of the options surrounding the classification issue and details the use of keywords for the classification of software components.

5.2 Software Classification Options

5.2.1 The Interface Definition of a Module

As part of object-oriented programming languages such as Modula-2 [Wir85] and Ada [Pyl81] the ability to separate module specifications and module definitions into separate compilable units has been forwarded. In this context the *ModSpec* and *ModDef* are analogous to the interface and implementation in the molecular view of design objects. In the IDM view of the two roles of the interface, the module specification is no more than the interface portion of the module used in the declaration role.

In keeping with the object-oriented philosophy, the ModSpec and ModDef are physically kept separate in the design of programs. The potential user of a software component is allowed to view only the part of the module that is revealed to him in the

[1] The organization of these indices and retrieval mechanisms is the subject of a later chapter.

module specification; the implementation details of the module are protected and inaccessible. Furthermore, the grammars of these languages require that little more than the syntax of the module declaration be part of this specification. Information other than this declaration, for example, on the function or constraints on the module, is not available, thereby severely restricting what can be used for classification of the module.

5.2.2 Adding to the Interface Definition

In recognizing the need for more information, there have been several proposals that expand the amount of descriptive information included in the ModSpec or interface. According to [Mat84], in addition to the module specification of each routine, there should be lists of data describing:

1. Other modules of code required to execute the software module.
2. The language, operating system, runtime utilities, and input/output devices.
3. Automatic interrupts that effect the module.
4. The amount of memory required by the software.

These identifying factors significantly increase the amount of information available for the classification of the component. Of course, the more information available for classification, the more accurate the retrieval strategy can be when searching for candidate components to meet the current requirement.

According to [Len87], a software component should be identified by a more formal *specification*. The specification should include:

1. A functional overview.
2. The syntax of the module interface.
3. A formal operations semantics describing the actions of the module.
4. All dependencies.
5. An example.

While these observations are important in that they identify additional means that may be used to uniquely identify a module, no technique for cataloging or retrieving the modules is included in their research.

In the schema developed by [Che84], each program entity has a name and a set of attributes, which are essentially (type, value, version) triples. There are some 25 distinct attribute types that range from descriptors, data types, and syntax specifications. This schema is not much unlike other techniques, except for the inclusion of a version number in the triple. This version number indicates how many times that attribute has been modified.

5.2.3 Formal Semantics

Finally, there are various formal methods that have been proposed for the classification of software. In [Lit84], there is a formal specification language for software that is based on a formal algebra. This language uses catagory theory from the field of mathematics to deal with properties characterizing classes of algebraic structures. The argument is that catagory theory can provide the formalism required for specifying the externally viewed behavior of software components.

[Gog84] and [Der85] also propose to make formal denotational semantics or predicate calculus specifications, complete with preconditions and invariant assertions, part of a program library of frequently performed tasks. These methods have an advantage in that they can be shown to be correct by well-known proof techniques. In the words of [Gog84], there is "no junk" (everything has a purpose) and there is "no confusion" (the correctness of the function and classification criteria is provable).

An advantage of these methods is the accuracy that can be attained with formal semantic nomenclature. Furthermore, the exactness of the result may permit most of the process to be completed automatically, including parsing design objects for input into the schedule and mapping formal requirement specifications to formal definitions of

reusable parts. In the future these methods may well prove to be a required part of the software engineer's toolkit.

However, [Lit84] expounds on several shortcomings of these techniques. Any method used for classification should integrate the design and management of reusable components into proven current design techniques. The axioms of catagory theory, of course, have no equivalent in programming languages. Consequently, ensuring that a component implements the semantics of the theory will require additional tools in the development environment. Furthermore, because of the conciseness, power, and accuracy of formal semantics and the predicate calculus, they are very complicated for persons untrained in their use. This is a important factor in determining the usability and wider applicability of these techniques.

5.3 Use of Keywords for Software Classification

The most widely accepted means for classifying software that can be understood by the general user is the keyword list. However, due to the natural ambiguity and generality of software there is no accepted agreement on a standard technique or set of keywords for this purpose.

There are two general variations on the keyword option. These are to have a fixed number of keywords that take the role of attributes, each attribute assuming a value that describes the software component. An example would be to have *Function* as an attribute of the component, with a possible value of *Sort*. The other option is to have a variable length list of single keywords that are totally up to the designer's discretion. A routine might then be classified by a list of keywords such as *Sort, Quick, Integer_Array, Pascal, MVS*. Such a method is similar to the one that authors use to classify their journal publications. The choice of method used depends on the retrieval strategy and interface mechanism the library designer wishes to use.

Function	Objects	Medium	System type	Functional area	Setting
add	arguments	array	assembler	accounts payable	advertising
append	arrays	buffer	code generation	accounts receivable	appliance repair
close	backspaces	cards	code optimization	analysis structural	appliance store
compare	blanks	disk	compiler	auditing	association
complement	buffers	file	DB management	batch job control	auto repair
compress	characters	keyboard	expression evaluator	billing	barbershop
create	descriptors	line	file handler	bookkeeping	broadcast station
decode	digits	list	hierarchical DB	budgeting	cable station
delete	directories	mouse	hybrid DB	capacity planning	car dealer
divide	expressions	printer	interpreter	CAD	catalog sales
evaluate	files	screen	lexical analyzer	cost accounting	cemetery
exchange	functions	sensor	line editor	cost control	circulation
expand	instructions	stack	network DB	customer information	classified ads
format	integers	table	pattern matcher	DB analysis	cleaning
input	lines	tape	predictive parsing	DB design	clothing store
insert	lists	tree	relational DB	DB management	composition
join	macros	.	retriever	.	computer store
measure	pages	.	scheduler	.	.
modify	.	.	.	.	.
move	.	.	.	.	.
.	.	.	.	.	.

Figure 5.1. The Faceted Classification Schedule

Recent work by [Pri87] and [Bur87] has been to identify candidate keywords for software classification. The listing of six keyword categories in Figure 5.1 is part of the *faceted* classification and retrieval technique proposed in [Pri87, Pri88]. Along with each of the six classification categories is a list of values that the user selects as the most applicable entry for the piece of software that he is classifying. The software component is then identified by the six-tuple comprised of the values for these six categories. However, since the selection of values for these keywords is subjective, the same component may be classified in different ways by different people. Without some means to group synonomous values of keyword attributes, a reusable component may not be retrievable in a given situation. In fact, an on-line thesaraus is used for this purpose. The authors further studied this problem by asking a group of graduate students to classify a set of modules, and experienced from 100% agreement on the keyword for *function* to a 60% correlation on the keyword for medium. This finding substantiates the

problem of trying to rigorously classify relatively abstract objects.

Figure 5.2 displays the desirable attributes about reusable components according to [Bur87]. The method suggested in their research is to combine two alternate mechanisms. The first is to permit the user to identify up to five descriptive keywords for each component. The actual length and contents of this keyword list is not restricted. The second mechanism is a category code system similar that used by libraries and publications such as *Computing Reviews*. Details of this category code schema are not given, and the authors admit that they have not been able to standardize the types of reuse information required to document components in a Reusable Software Library (RSL). They also suggest that standardizable techniques, such as objective metrics, are also needed to help system librarians rate all component's attributes on an equal basis. This would assist in the correlation statistics experienced by [Pri87].

5.4 Allowable Values for Keywords

The problem of allowable values for keywords is primarily one of vocabulary control. Without some restrictions on allowable values of attributes or keywords, the search space can become quite large and the time required for searching the lists of these keywords unacceptably slow. In the faceted schema, the user must choose from a list of allowable values for each of the six attributes. This process is made easier by assistance from a thesaurus, and effectively restricts the search space for each attribute.

Without a thesaurus or other automated assistant, it has been pointed out that the use of controlled vocabulary can actually be *less efficient* than with an uncontrolled vocabulary [Fra87]. The reason is that the user of the system must be familiar with the classification schedule and retrieval mechanism in order use them effectively. The use of a somewhat artificial controlled vocabulary, where conventions must be learned, may be

Attributes	Description
UNITNAME	The unitname is the name of the procedure, package, or subroutine.
CATEGORY CODE	The catcode is a predefined code that describes the functionality of the component.
MACHINE	The machine signifies the computer on which the component was programmed.
COMPILER	The compiler signifies the compiler used during development of the component.
KEYWORDS	Keywords are programmer-defined words that describe the functionality of the component.
AUTHOR	The author is the person who wrote the component.
DATE CREATED	The date created is the date the component was completed.
LAST UPDATE	The last update is the date the component was last updated.
VERSION	The version is the version number of the component.
REQUIREMENTS	The requires field contains information about any special requirements of the component (eg. other components that must be available).
OVERVIEW	the overview of the component contains a brief textual description of the component.
ERRORS	The errors field contains information about any error handling or exceptions raised in the component.
ALGORITHM	The algorithm field contains the algorithm used in the design of the component.
DOCUMENTATION AND TESTING	The documentation and testing field contains a description of available documentation about the component and a description of test cases.

Figure 5.2. The RSL Classification Schedule

a barrier to the effective use of a library retrieval system by anyone who is not an information specialist. The automated assistants help break down this barrier by providing a certain portion of the expert knowledge that normally requires the presence of a full-time librarian.

The allowable values of keyword attributes are not the only concern. The format and data types of the attributes are also critical in determining the types of searches that may be supported as well as the subsequent efficiency of these searches. For

example, retrieving all module interfaces with *Function=Sort* is a straightforward search easily implemented in any database system. However, locating all interfaces designed after a given date may require a mathematical comparison of values of type *Date,* a value that may be comprised of the three fields *Day, Month,* and *year.* While this option may be complex to implement, choosing to store this data type in a Julian format can reduce complexity of the search algorithm significantly. The allowable search operations and retrieval methods should be evaluated before the allowable values of the keyword attributes are determined.

5.5 Approaches to Software Classification with the IDM

5.5.1 Introduction

There are two complete software classification techniques that are implemented as part of this research. The first technique discussed is the static classification schedule, which consists of a set of pre-determined keyword attributes, the values of which provide a broad description of the software module. This method has the advantage in that it encourages the designer to provide a description of the module in each of the chosen areas. This method is also somewhat more straightforward to implement, particularly with regards to a consistent user interface to the classification schema.

The second method that was implemented is based on a variable length list of keywords, similar to that used by journal authors when giving subject keywords for their articles. Each module has a list of single keywords that describe it; this list can be of any length. This method has the advantage in that it is much more powerful than a predetermined list, but implementing an efficient search mechanism and interface to this schedule is correspondingly more complicated. This technique also has the potential to be abused by a designer, who may simply fail to provide any keywords, or may give

several that are so closely related to each other as to be worthless.

5.5.2 Static Classification Schedule

The Static Classification Schedule used in this research is based on the work done by [Pri87, Bur87, Rei87], as well as personal experience. This method uses the two-tuple technique of *(attribute, keyword)* to describe various qualities of each software module. There are six keywords used for describing interfaces and for making constraints in calls. These are: function, major input, major output, medium, environment, and language. "Function" is the most important of these, and describes the general action of the module. "Medium" relates to the larger data structure that the routine acts on; for example, a ring buffer or sparse matrix. "Environment" is wher the routine works; for example, a specific operating system. "Language" refers to the source code language.

The use of the keywords in this implementation is not in any way constrained by the system, in the sense that the user must make specific entries or choose valid attribute values from some list. It is also important to remember that these keywords were derived from a number of different sources and are not conclusive. These particular keywords were chosen only to be representative of a classification schema that might be used in a functional CASE system and demonstrate that the IDM model can operate, in conjunction with a software library, with such a schema.

For the classification of alternatives, three keywords are used. These are time complexity, space complexity, and component. Object code size could also be used, but this was not implemented. The keywords, for time and space complexity are meant to contain values such as O(log N) and O(n-squared), but like the keywords for interfaces, there is no restriction that enforces this intention. The keyword "Component" refers to the logical part of the system that the routine works in, as opposed to a physical part; for example, the Output Manager. Unlike the keywords used for interfaces, these

keywords were derived solely from personal experience.

The advantages of this selection of keywords are many and diverse. First, these keywords are some of the most widely accepted classification attributes for retrievability. Second, they represent a broad description of the module rather than perhaps concentrating on one area, such as function. Third, the classification attributes can be incorporated into the user interface for the prototype CASE system in an elegant way. How the prototype implementation incorporates this schema into a practical interface is discussed in Chapter 8 and shown extensively in the Figures of that chapter.

The IDM is designed to accomodate the classification schema directly in the object structure. The interface and call objects both employ the six descriptive keywords described above; the alternative and the call both employ the three keyword scheme. The difference is in how the call uses the keywords; in the call the values of these keywords serve to *constrain* the interface and alternative objects that may be used to meet the call, where in the other two objects the keywords are exclusively descriptive. The call has the further responsibility of recording the software requirements for documentation purposes, a task adequately accomplished by the nine keywords and comments stored in the call object.

5.5.3 Variable Keyword Lists

The second option implemented for the classification schema is based on the variable length list of keywords technique. In this technique, the designer assigns any number of single descriptive words to each object. These keywords can relate to any attribute of the software; an example of such a list is shown in Figure 5.3. As in the static schedule above, the list of keywords is recorded in the IDM as part of the model. Each of the interface, call, and alternative objects contain a keyword list; it is up to the retrieval mechanism to match those keywords in the call to those in the interfaces and

alternatives when the designer is attempting to fill a call with a reusable component.

This technique was implemented as part of the research but was not chosen for the final implementation because it has the disadvantage of not encouraging or enforcing a broad classification of the module. Furthermore, a consistent interface providing uniform access to such a list is difficult with available software tools. It was important, however, to determine that the IDM could function in conjunction with this method.

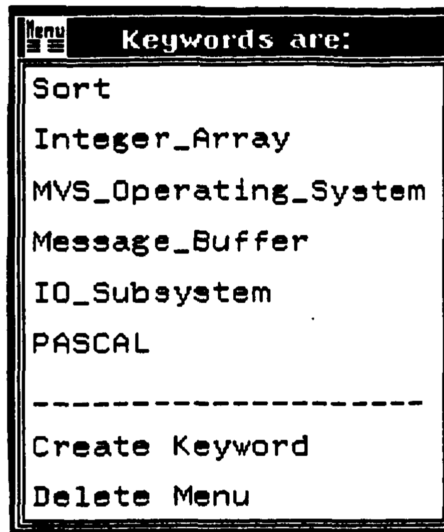


Figure 5.3. Describing a Module with a Keyword List

6. RETRIEVAL OF SOFTWARE DESIGN DATA

6.1 Introduction

Retrievability is the degree to which a software module can be stored, selected, and used by users who have no prior knowledge of its existence. The retrievabiltiy of software in a CASE system supporting reuse further includes the mapping of some conceptual or abstract specification of what is to be accomplished into a very specific data representation and algorithm that can be located in the software archive to accomplish the task at hand.

In short, in order to use a software module, you must be able to find it.

The issue of fast and efficient data retrieval is a major consideration in the design and analysis of database systems [Dat85a, Dat85b, Haw84, Ull82]. Consequently, the combination of indexing methods and data retrieval techniques in traditional applications is well understood. These applications, however, generally presuppose that the search space is well-defined and exact. On the contrary, the designer of a software system in a reuse environment seeks to retrieve software components based only on a vague understanding of what is needed and with no knowledge of what is available to meet that need. This chapter addresses techniques of indexing reusable components so that the conceptual mapping from requirements to availability can be made a reality.

6.2 Accessing Design Data

6.2.1 Desired Operations

The retrieval of design objects is required for two purposes; supporting the design process and supporting interactive queries about the design. These queries may be of an ad hoc nature to meet a variety of requirements, or may be systematically made as part of the automatic generation of design documentation and reports.

Retrieval of design objects during the design process is most often hidden from the user by the graphical editor or design tool. Since these operations repeatedly access closely related data, these retrieval operations are most efficient when the design data is clustered, or semantically stored in an object-oriented manner. Administrative and ad hoc queries are more often of a global nature, typically entailing questions about the entire design. Such queries are more efficiently handled by a relational database, in which efficient algorithms and operators are available for accessing large quantities of information. For most of the standard data retrieval situations in a CASE system, the retrieval issue is similar to the arguments found with the data modeling issue; there is a trade-off between grouping data as objects for design operations versus grouping data as relations for global operations and queries.

In a CASE system supporting reuse, however, there is the unique problem of having to retrieve design objects without necessarily knowing what is being sought, nor what is available. There exists a need for an object retrieval strategy that is flexible enough to be used early in the design process by providing the program designer with hints and other assistence that may lead him to candidate reusable components. The retrieval mechanisms discussed below are therefore intended to meet not only the usual retrieval needs of the design process, but are specially intended for use in a reuse environment in situations where program requirements are ill-defined.

6.2.2 Indexing Strategy

One possible indexing strategy is to have the supporting database maintain an index for every classification category possible. This allows for the most speed and flexibility over the widest range of queries. For example, in the faceted classification schedule of [Pri87], a total of six indices would exist, one for every entry in the classification tuple. This allows very rapid access for queries specialized to only one entry in the tuple, assuming wild-card values are valid for the other entries.

However, such a technique is not without its disadvantages. In this case, the problem is essentially one of a time/space tradeoff; for the luxury of fast response times for specialized queries, the cost is the memory and disk space required for the indices as well as other overhead required to maintain the indices. For classification schedules with considerably more classification attributes, there must exist a point where maintaining an index for the additional attributes is no longer practical.

At the other extreme, there is the option of not maintaining an index for each attribute, and using sequential searchs to locate the information. Clearly, for large search spaces or for queries that are commonly executed, this option is not desirable because of the time that is required for these linear searches.

Interestingly, several research efforts that have discussed the software classification and storage issues have neglected to address the issue of retrieval. For example, while [Len87] is quite concerned about the classification of the reusable building blocks in an operating system environment, no means for the retrieval of these blocks during the design process is discussed. Of course, some indexing or retrieval technique must be included as part of the complete CASE system. The following section introduces candidate retrieval techniques and outlines some of the advantages and disadvantages of each.

6.3 Indexing Techniques

6.3.1 Software Catalogues

The simplest indexing technique is perhaps those used by the published catalogs of software utility routines. Such catalogs are found as part of the user manuals of large computer systems, or are published in book form as an accessory to the computer system. Examples are the math packages available on the mainframe computers at most research centers [DeB85, Cor87], and program libraries such as for the Apple or

IBM series of personal computers [Rug86].

These software catalogs use a table of contents to list the available routines in the library. Such routines are usually grouped into "chapters" by function, such as placing all trigometric utilities together, all sorting functions together, all graphics utilities together, etc.

The method of retrieval in software catalogs consists of having the user scan the table of contents for the routine that meets his need. The grouping of related routines into chapters makes this process fairly straightforward, and since the user is accustomed to using the table of contents in books, he is generally comfortable with the method and able to use it with a minimal amount of instruction.

The primary disadvantage with the software catalog is that it is difficult to automate and incorporate into a CASE environment. In addition, this method only works well with a small number of routines. It is not practical to expect a user to manualy browse through a library consisting of thousands of candidate components, nor is is reasonable to expect to give him enough information in the table of contents to distinguish between a large number of closely related components.

6.3.2 Multilists

One technique which is exceptionally flexible and straightforward to implement within the framework of conventional database systems is the multilist indexing structure. A multilist index, as shown in Figure 6.1 [Wie87], consists of an index record for each value of the attribute that describes the software component. From that index record there is a chain of pointers to records, each containing the address of a software component with that value for the attribute. [9] Queries of the type

---

[9] Again, a tradeoff exists between the size of the directory (multilists) and the length of the search in the main file. The opposite of the multilist structure is the *inverted file*, in which there is one index record for each value of the keyword, and the length of each multilist is one record.

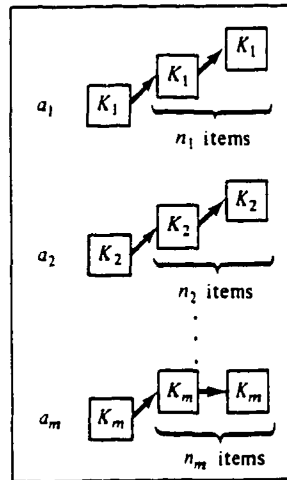


Figure 6.1. A Multilist Index

Select Interface_Names

From Interfaces

Where Function=Sort

result in traversing the chain of records for *Function=sort* in order to quickly provide the keys for the required interfaces.

When two or more search criteria are specified, such as in queries of the type

Select Interface_Names

From Interfaces

Where Function=Sort

And Algorithm=Quick

a *join* operation is used by the database to merge the two resulting lists of candidate components. The join operation compares the two lists and extracts the keys of the

components that are common to both lists.

This technique has limitations when the number of possible values for each attribute is large and is allowed to grow. In this case the amount of space and overhead for maintaining the multilist structure becomes unacceptably excessive. For this reason, the maximum length of the multilist in applications systems is often controled. However, because the capability for implementing the multilist index is often a one of the features of the supporting database system, and because the method is well understood, it is a popular technique.

6.3.3 Cluster Theory

Cluster theory is a file organization technique for document libraries that has had an important influence on several of the software library retrieval strategies to be discussed below. In cluster theory, documents carrying similar content descriptions are grouped into *clusters* [Sal75]. These clusters are identified by a representative cluster profile, or *centroid.* The centroid is a weighted set of terms derived from the *descriptive vectors* from the documents included in the cluster.

The descriptive vectors of a document come from the classification method used for the documents. The vector is the result of some algorithm that compares lists of keywords, catalog numbers, or some other criteria. In software, many of the classification methods classify the component by a fixed set of attributed keywords. By imposing an ordering on these keywords and considering the values for the keywords as an n-tuple, the result is a document vector as required for this and other classification and retrieval strategies.

A search in a clustered file proceeds as follows. First, the target vector is compared with the index file of centroid vectors. Second, documents within the candidate clusters are ranked in decreasing order according to their closeness to the target vector. Finally, individual documents are retrieved and examined. It is clear that the "depth" of

the search can be easily controlled in a clustered file because it is possible to search only the "best" cluster, or, if desired, the top two clusters, or the top ten, as necessary.

In traditional cluster theory the clusters are automatically generated by the database system, and a document is allowed to occupy a place in more than one cluster. Furthermore, documents may be moved from one cluster to another if it should prove useful. The physical grouping of the documents in secondary storage is also assumed to be managed by the database system in order to optimize the number of disk accesses required to retrieve all of the documents in a cluster. However, in some of the retrieval mechanisms utilizing clustering techniques that are discussed below, these requirements may be relaxed.

6.3.4 Associative Networks

An associative network is a tree structure in which the internal nodes of the tree form the index for the components [Dep83]. This method requires a set of "features," or *(attribute, value)* pairs, that can uniquely characterize each component. If we think of the set of these features as basis vectors of n-dimensional space, then each element of library can be viewed as a point in the space. The vector that describes a given element is called the *pattern vector.*

Each software component in the n-dimensional feature space having a similiar descriptive vector is grouped into clusters; the clusters are represented as a hierarchical tree. The root of the tree determines the search node at the first level down the tree by comparing the pattern vector of the desired element to each of the first level nodes and taking the child that most closely matches the target. Further levels of the tree are traversed by sequentially accessing the "closeness" of the next descriptor in the pattern vector with each of the immediate successors of the current tree node. Part of a sample associative tree index for software is shown in Figure 6.2. The goal of the index traversal is to identify the cluster of library components that has a feature vector most

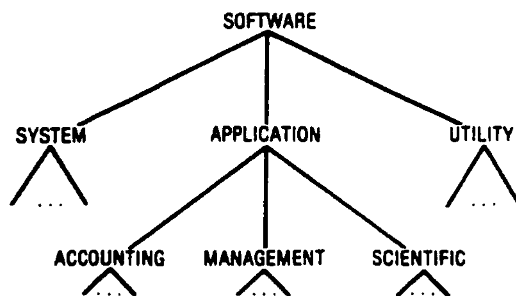


Figure 6.2. An Associative Tree for Software

closely matching the target pattern vector.

The fundamental problem with this technique lies in that the values assigned to the feature vectors in the associative network must be orthagonal. Orthogonal values occur when only one value for each attribute could be considered to accurately describe a component. If the values for an attribute are orthogonal then classification of the components and their subsequent grouping into clusters is easy. If this is not so, several problems arise. The first problem is that if more than one value could be considered to describe the component, then the suggested assciative tree technique does not work because a component may "belong" in several leaves of the tree. The second problem lies in the retrieval process. If, for any feature more than one value may be "close" to the target value for that feature, then all those subtrees in the index must be searched. While this can be done, and is actually allowed in traditional cluster theory, within the associative network framework the associative tree becomes a relatively unimportant part of the whole process. Furthermore, some heuristic to define the notion of "closeness" of feature vectors needs to be defined. While several known measures may well serve this role, none is identified as being appropriate for the associative tree technique.

6.3.5 Faceted Schema

The faceted schema proposed by [Pri87] and shown in Figure 6.3 is based on the assumption that collections of reusable components are very large and growing continuously, and that there are large groups of similar components. As in the associative tree index above, the faceted schema requires a set of descriptive *(attribute, value)* pairs that describe each software component. A *facet* is the term given to these attributes. The schema also includes, however, a metric for conceptual distances between terms in each facet that is used to help select between closely related components.

The classification mechanism proposed by [Pri87], consisting of a six-tuple of (attribute,value) pairs, is described in a previous chapter. The faceted index is represented as a conceptual graph that measures the closeness among *terms* in the facet. Nodes in the directed acyclic graph represent general *concepts* related to the software facet. Leaves in the graph are terms for the general concept. Arcs connecting the general concepts and the concept terms have a weight assigned to them that represents the "closeness" of a concept to a particular term. Unlike the associative tree technique, where the closeness metric is left undefined, in the faceted implementation the weights relating the closeness of two modules are user-assigned.

One practical application of a closeness measurement occurs during retrieval. If a particular term in an entry does not match any available description in the collection, the system tries the next most closely related term to retrieve descriptions of closely related items. One major drawback to this technique is that constructing conceptual graphs for more than a few concepts and terms is very time consuming. However, because the conceptual graph allows for synonyms of concepts to be systematically accessed, there is no requirement for the terms describing each concept to be orthagonal.

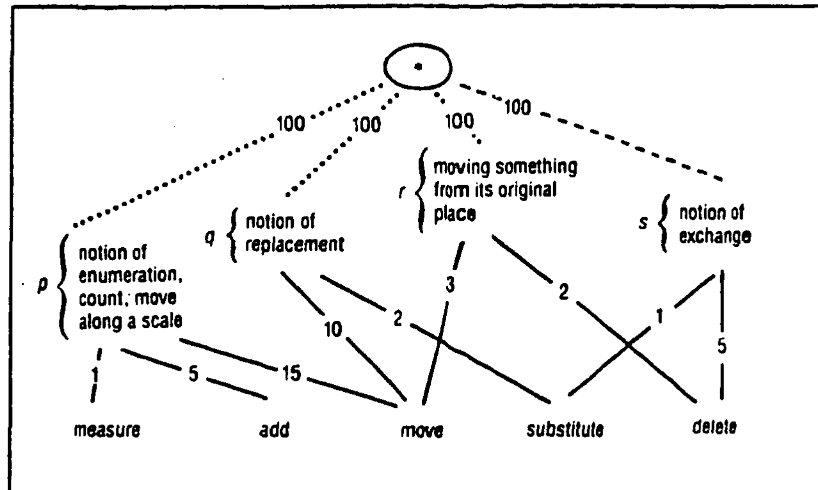


Figure 6.3. A Faceted Schema Index

6.3.6 Classification Matrix

[Mit87] states that software can be classified in two ways:

1. The *applications* in which they are used.
2. The *task* they perform.

Applications which are alike, such as accounting programs, can be combined into a common *application category*. Programs which are alike according to the task they perform constitute a *task category*.

In [Mit84] a matrix organization based on these two criteria is suggested for the index. Columns in the matrix represent task categories, and rows in the index represent application catagories. Each software component in the archive is classified by application and task, and the appropriate entry for it is made in the matrix. Candidate reusable components are then retrieved by identifying the task and application area required, and looking up the components in the matrix.

This matrix organization for the index can clearly be viewed as a table, and organized into database relations in a relational database system. This is a practical as long as the software components can be sufficiently identified by the two stated criteria. For large software archives, however, the number of components occupying a bucket in the table can become quite large, and searching the buckets no longer practical. To address this problem, and to accomodate the classification mechanisms that rely on more than two classification criteria, n-dimensional matrices will have to be formed [Mit87]. However, this organization is non-trivial to implement in a relational system.

6.3.7 Artificial Intelligence Techniques

Recognizing that retrieval of software components is a key problem in software reusability, some efforts to apply techniques from other fields have been tried. While such efforts are only partially related to this research, it is worth recognizing that work is currently ongoing in this area. For example, one retrieval technique combines the the artificial intelligence [Cha86] and database [Tsi82] concepts of a association/generalization [Mit87].

In such a system, "associations" are formed between software modules based on whether or not they are similar to each other in some predefined way. The details of this similarity metric are not given. However, if there is an association between two modules, then they are considered to have a common ancestor at the next higher level of generalization. Candidate components are located by navigating down the generalization hierarchy, progressively and interactively refining the requirements of the target component until the candidate modules in the database are located.

6.4 Discussion

6.4.1 Matching Needs with Available Components

In the retrieval strategies discussed above, the common goal of matching a high-level, possibly incomplete description of a needed target component with candidate modules from the software archive is approached. This issue is at the crux of the reusability problem. Ultimately, the usefullnes of the retrievability strategy is a function of the criteria used to specify modules and requirements.

As can be noted from the previous section, the retrieval strategy is dependent on many other factors. The first of these comes from the problem of determining the amount of resources in the database to dedicate to indices and retrieval algorithms. When evaluating this decision, the types and frequency of queries and design operations must be considered. The desired flexiblity of the retrieval process must also be determined; some methods will retrieve many related modules for the designer by using indices based on conceptual networks, while other methods require a more precise definition of the required component and more interaction from the user. Some of these dependencies are further discussed below.

6.4.2 Dependencies of the Retrieval Techniques

6.4.2.1 On the Classification Schema

Each of the retrieval techniques discussed above depends in one form or another on the method used to classify the software components. The matrix index, for example, presupposed a two dimensional classification schema, whereas most of the techniques worked with a n-dimensional vectors of descriptive keywords. It is clear that some retrieval techniques are not capable of functioning with classification mechanisms other than those for which they where designed. It is desirable, of course, to employ a retrieval technique that is not only efficient, but flexible enough to operate under a variety of conditions.

6.4.2.2 On the User Interface

The ease and efficiency with which a reusable software component can be included in a new application depends in many ways on how the user interacts with the CASE system via the interface to the software library. Obviously, an interface that helps the user retrieve candidate components based on keywords will be quite different from one that retrieves components by formal mathematical methods.

Methods based on keywords may help control the size of the attribute-value vocabulary and make data entry easier by providing pull-down menus from which values for the keywords may be selected. Text entry boxes and automated thesaurus-based assistants are also possible [Arn87, Iso87]. Some interface techniques based on artificial intelligence are also being used; the user may query the database through a natural English dialog with a query processor [Bur87]. A sample session with such a system is shown in Figure 6.4.

One interesting approach in systems that utilize weighted networks or descriptive vectors is to assign the weights to attributes by means of a bar graphs [Bur87]. The user indicates which arcs or attributes are most important by increasing the "barometric" scale in the bar graph associated with that attribute, as shown in Figure 6.5.

The key point of the discussion of these dependencies is that the retrieval method is often very visible to the user of the CASE system through the interface, and it is often this interface that actually determines the usability of the system. Developing a user friendly and practical interface, therefore, is a major concern when considering potential methods of software retrieval.

Request	Response	Action
I need a stak package.	by "stak" do you mean "stack"? Current series of requests: I need a stack package.	The unitnames and overviews of all stack packages are displayed.
Only display those that implement garbage collection	Current series of requests: I need a stack package. Only display those which implement garbage collection.	The above search is pruned so that only those stacks that are managed are displayed.
When were they written, and what version are they?	Current series of requests: I need a stack package. Only display those that implement garbage collection. When were they written, and what version are they?	In addition to unitnames and overviews, the dates and version numbers of the packages from the previous search are displayed.

Figure 6.4. A Natural Language Query Session

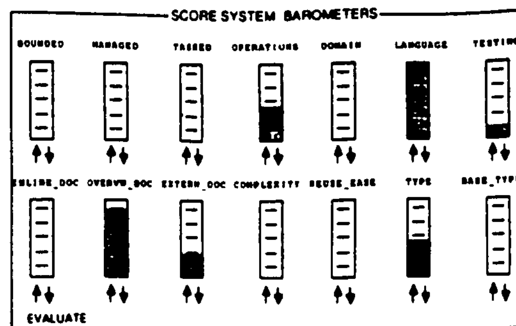


Figure 6.5. The Relative Importance of Keywords

6.5 Approaches for use with the IDM

6.5.1 Introduction

There are two different retrieval techniques that were implemented as part of the prototype system. The first of these two options is a linear search on the keywords that describe the objects. There is no separate index; the keywords used for the search are stored within the data object representing the software module. This method has the advantage of saving the space and overhead associated with maintaining a separate index, but has the disadvantage in that it is costly in terms of CPU time to extract the keyword values from the software objects and then to conduct a linear search on these values.

The second retrieval option in the prototype CASE system is based on the multilist architecture for database indices. There are several reasons for this. One is that the method works well with both of the keyword classification schema that have been incorporated into the data model. Another reason is that the multilist indices cooperate nicely with the user interface that has been implemented for data entry and retrieval. Both of these options are discussed below.

6.5.2 Attribute Search

The attribute search method was implemented for use with the static classification schedule and works by extracting the value of a particular attribute out of each object in the database, and comparing it with the value that is desired. This is effective because the set of keywords is known in advance, and the action is made to operate very quickly by pre-programming the set of possible queries. For large databases, however, the cost of extracting a given attribute from every object can be prohibitive. In such cases it is necessary to follow a heuristic search strategy, or resort to an indexing solution such as that used in the multilist technique.

A good heuristic search strategy for this option is to first choose for the first search operation the keyword that is most likely to yield a minimal positive result, and

then to initialize the search on the next keyword in the descriptive vector with the result of the last search. [10] This is similar to the strategy used in commercial automated library systems, and optimization strategies used in large database systems. This speeds up the searching process significantly, and performs best when the first searches restrict the search space as much as possible.

6.5.3 Multilist Index

The second of the two options implemented was for use with the variable length list classification method. This retrieval technique maintains a multilist index of all the keywords and the modules having those keywords as attributes. This index is automatically system generated and maintained, and can be regenerated on command [Rov89].

In addition to to the multilist index for keywords, there exists a multilist index for each attribute of the module that is a potential search key. The group of attribute fields that are identified as possible search qualifiers and for which currently exist multilist indices are:

1. Version
2. Keywords
3. Parameters
4. Designer
5. Interface name
6. Interface tag
7. Alternative name
8. Alternative tag

[10] The search for library components is done by string comparison of the target attribute value with the candidate attribute values. A technique of searching that was insensitive to blanks and case was tried in order to increase hit probability, but implementation-specific technical problems were encountered due to the fact that ROSE indices are case sensitive.

9. Performance attributes

There are also indices for the various data types and variables in the design.

The cost in CPU time to maintain these indices is neglible during the design process, since individual modifications can be made cheaply. The cost in terms of memory, however, is much greater, as there is a requirement to store the multilists in the main memory workspace, and there is no restriction on the size of the indices nor the number of indices that may ultimately exist. However, at search time the database can access the multilist containing the indices of candidate components in one fast search operation.

The search strategy for the multilist option is to retrieve the multilists for all keywords in the descriptive vector and evaluate them in parallel. The lists are combined via a series of *join* operations, with the result (if any) giving the addresses of candidate components. While the join operation is expensive for large multilists, in practice, this is moderated because as the join operations execute, the length of the lists of candidate components shrinks rapidly.

Each of the two above options has corresponding advantages and disadvantages. The first method is much more flexible in terms of allowing the user to control the search on specific attributes; the second method takes more of a "shotgun" approach in this regard. The first method avoids the overhead associated with a potentially enourmous number of multilist indices; the second avoids the overhead associated with extracting the descriptive vectors from the design data and conducting a relatively slow linear search on the result. The choice of a final implementation technique depends ultimately on whether the major concern is one of space, in which case option 1 is prefered, or is one of time, in which case option 2 is preferred.

It should also be noted that a combination of these two techniques may prove to be the optimal solution. Such a technique would provide multilist indices for the static classification schedule. This option would restrict the number of indices from having no upper bound to a total of nine, and be indexing each attribute would save the cost of extracting attribute values from the database for every query.

7. ORGANIZATION OF THE SOFTWARE ARCHIVE

7.1 Introduction

The physical organization of software libraries determines not only the efficiency of the CASE system but often also has a direct effect on the user's view of the software development environment. The physical organization of the library is, however, much less discussed in the current literature than other reusability issues such as classification and retrievability. This is for two basic reasons. First, most research tends to concentrate on matching the high level description of the needed target to the available components. After this most difficult task is accomplished, the actual, physical retrieval of the target component is considered relatively straightforward. Second, the organization of a permanent archive is a consideration that is often not faced until late in the process of researching other reusability issues. Most of the current research and publications, therefore, have not progressed to the stage where library organization has had to be addressed. Nonetheless, careful attention needs to be directed to software archive organization so that the reusable library can properly supplement the design process.

In VLSI CAD, once a part is designed and tested it can be "plugged" into any circuit and used as needed. The same should be true for software; programs used in different applications should be truly identical. Obviously, this is the ideal case. The goal of a CASE system dedicated to reuse should be to have application programs consist solely of a sequence of references to the software library. The organization of such a library should reflect that goal.

The issue of software archive organization is the topic of this chapter. First, several candidate library organizations from the literature are discussed. Next, required operations on the library based on the library organization and the design process are presented. The chapter concludes with an analysis of the library organization

incorporated in the current CASE prototype.

7.2 Organization of Software Libraries

7.2.1 Application-Oriented Organization

Application-oriented libraries are those archives that contain routines dedicated to one type of problem, or alternately, group the routines in the library based on the type of function they perform.

There are many examples of libraries dedicated to a special class of problems. Based on the belief that user interface routines are the most "reusable" of the components in their applications, [And88] has a dedicated library of these components. Their argument is that by maintaining a consistent "look and feel" to their products by means of similar interfaces and screen organizations, their products are more usable and easily learned by their customers.

Another approach to application-oriented libraries is given in [Nei84]. Each application is thoroughly analyzed by experts in that field, and reusable routines are written in a domain-specific language. Programs in that area of application can then be assembled from these routines by writing a problem statement in the domain language. However, only about 10 or 12 fully usable application domains have been built because domain analysis and design is *very hard* [Nei84]. This technique also has the disadvantage that routines in one domain area cannot be applied in another domain area. This method, due to the specific nature of the domain analysis process, does not apply to this research.

Examples of program libraries organized by the function of the routines are likewise numerous. Subroutine libraries, such as for FORTRAN [DeB85], or PASCAL [Rug86], are organized in such a manner. Trigometric functions, date/time functions, sorting, and searching routines are all grouped by type. This organization is suitable for

these collections of basic routines for several reasons. First, there are typically a limited number of such functions, and each is small and well-enough defined so as to be identified by name. Second, the retrieval mechanism for such libraries is often a table of contents in a software catalog or user's manual, where grouping related items into chapters is a well-known concept to users. However, for larger libraries, such an index is impractical.

By encouraging the use of the programming language Ada, the Defense Department of the United States has also sparked a lot of research in how to best take advantage of the software reusability features that are part of the language [Con87,Gag87,Onu87]. The Ada Software Repository has been created on the Defense Data computer Network as a central library for reusable Ada components. The Repository is organized by dividing it into several subdirectories which represent topic areas. Some of the topics are educational information, software development, graphics, and communication message handling. Within the general topic areas, however, little is done to further classify the Ada components in the library. This is primarily due to a lack of classification schedule; taxonomies for that purpose are currently under consideration, with the most likely candidates based on a keyword-style schema [Con87].

7.2.2 Organization Based on Retrieval Method

A number of library organizations are dictated by the method of retrieval employed by the supporting CASE system. Not all retrieval methods, however, are specific as to where the boundary between the physical organization of the library and the index to the library is located.

For example, in the category theory work done by [Lit84], he suggests that the library of reusable components should be structured to reflect the dependence among the associated theory morphisms. He also suggests that the theory morphisms should serve

as the criteria for selection of components from the library, and that a candidate organization for these morphisms is an acyclic graph. Such a graph could be stored in a relational database, but he does not specify whether the actual design information, or just the name of a operating system file where the information can be found, is to be stored in the database.

Likewise, [Iso87] suggests storing the requirements/design schema of his system in a network organization. He does not say whether the design data or the actual index should be placed in such a network, nor whether the network should be managed in main memory by a database or in secondary storage by the operating system. In light of this, the remainder of this chapter will assume that the term *software archive* refers to the physical organization of the design data on disk, and not the indices to this information.

The retrieval method and archive structure are also both influenced by the type of internal organization used in the data files. It is possible to alleviate all questions of archive organization by simply storing all of the archive in one file. There exist tradeoffs, however, as to the granularity of the file structure and the capacity of the file server and the ability of the operating system to manage the files. A common approach is to store one design object per file; all the design information contained in the file is *used* when the file is initially read by the database or CASE system [Gog84, Har86]. In the context of this chapter, the basic unit in the archive is the operating system file, and the information contained in a file corresponds to the design information for one software module.

7.2.3 Public Archives and Private Workspaces

Presumably, a lot of programs in the software archive can be reused in different applications, sometimes after changing only a few parameters. If the archive is organized on an application basis, the search for available components in other parts of

the archive has a conceptual barrier, even if the actual search for them is physically possible. For this reason it is desirable to remove this barrier and allow all available routines to mingle, thereby giving them all an equal opportunity to become candidates during a search for reusable code.

An additional requirement, however, is to maintain the integrity of the information in the archive. During the extended, conversational transactions typical in a design environment, partially modified information in the public archive could corrupt other users and routines dependent on those partially completed components. One solution is to separate the library into public and private workspaces, and place certain constraints on the information in the public workspace [Rov88].

In such an organization, the public archive serves as an repository of approved routines, and are read-accesible to client users. However, routines cannot be added to this archive unless they are fully tested and approved by the library administrator or project manager. The private library, on the other hand, is a local workspace area where the individual designer keeps his current project information and partially completed code. Access to the private workspace is limited to the owner of the workspace; however, if access rights are granted to others it is with the understanding that the routines within the workspace may not meet the standards required in the archive.

A further addition to this organization is a third type of work area, and is supported by [Kat86]. Rather than specifically grant access to routines in the private workspace to designers requesting such access, a semi-private workspace is created. The contents of this workspace is read-accessible by members of the group working with the owner of that workspace. This organization, while adding some complexity to the archiving process of the CASE system, has the advantage that it provides an explicit location for common code to be shared in a distributed environment while further

modifications to other versions of the code are being made.

The separation of software libraries into public, private, and possibly semi-private workspaces has an added advantage in that the organization is immediately transferable to a distributed database and design network. Since many CAD systems operate on private workstations that are networked to a central computer, all of the personnel working on a large project can easily port and maintain there private workspaces on their personal machines, leaving the public archive on the central computer for all to access. Given that this is a common situation in CAD and CASE environments, it is a desirable feature in software library organization.

7.3 Operations on the Software Archive

The intent of the software archive is, of course, to provide a common repository of approved routines for general use. As such, the archive must provide *read* access to those approved for its use. However, in addition to simply adding new routines to the archive, there must exist a set of operations based on the semantics or constraints that the library administrator wishes to enforce in the archive.

The first such constraint is on the type of routines that are allowed in the library. A realistic constraint in this case is only to allow approved and tested routines to be archived. Therefore, before a routine may undergo a *write* to the library, it must pass a formal testing, inspection, documentation, and approval process. Such a process may be part of the software engineering methodology employed at the site or made a function of the database.

Another constraint might be on whether or not to *delete* past versions of an object. Considerations are the preservation of disk space versus keeping past releases of code for maintenance, documentation, or legal reasons. While methods of economically storing versions, such as the "delta" method of the UNIX SCCS [SUN86] are viable options, the issue of deleting old versions must be addressed.

Should deletion be allowed, further constraints must be considered. Deletion should not be allowed for components that have dependencies that may propogate throughout the library. For example, if Routine A is used by another Routine B, this dependency should be removed before Routine A may be deleted. The maintenance of these constraints is the responsibility of the library administrator.

7.4 Organization of Implementation Archive

The software archive in the prototype CASE system is organized by the type of objects used in the IDM model. It is further divided into public and private libraries in order to fully support conversational transactions, distibuted design environments, and database integrity as discussed above.

The software archive is a public directory consisting of four sub-directories; a directory for calls, one for interfaces, one for alternatives, and one for data. The separate directory for data exists for efficiency of queries about the use of variables and parameters. There is also a local workspace in the user's directory that contains a directory for each design currently under development. Each of these local design directories is divided, like the public archive, into directories for each object in the IDM. This allows the designer to freely develop his designs without concern for side effects caused by changes to the archive.

Operations on the public library are limited. Because it is desirable to retain old code for documentation purposes, deletion of objects is restricted. In keeping with the semantic constraint that all components in the public archive must be validated and tested, write access is restricted to approved objects. Operations on the private workspace, however, are quite liberal. By using "hooks" from ROSE to the VMS operating system, entire directories and subdirectories representing designs and subdesigns are created, updated, and destroyed at the designer's option.

When the designer has completed the work on a design or sub-design in his local workspace, and the work has been tested and approved, the work is archived in the public library. In order to limit side effects of any actions to other modules, modifications to the archive are limited. In terms of interfaces, the names of new alternative implementations may be added. In terms of alternatives, entire new alternative implementations may be specified. Calls are always unique and are added or updated as necessary in the archive. The only exception to these rules is that the version of an alternative that is to be considered "current" may be redesignated.

In keeping with the operation of the ROSE database system, each design object is stored as an operating system file. This is practical and efficient because the size of the data objects has been carefully designed to correspond to the size of an operating system file. The indices for the design objects in the database become the names of the operating system files where they are stored. Interfaces are accessed by name, alternatives are accessed by the two-tuple *(interface name, alternative name)*. Calls and data objects are accessed by a system generated surrogate identifier. Finally, while the operating system performs the actual manipulation of the disk files, all database activities are performed by ROSE, thereby maintaining the speed of the CASE system.

In a commercial implementation of this system, it is anticipated that the public archive will be located on a central computer that hosts several workstations or design terminals. Each of the private workspaces will be located either in private account directories or on the private workstations. Supporting distributed design environments is a significant feature of this library organization.

When the program design is complete, the modules that comprise the design are added to the database as individual design objects. Once entered into the database, each module becomes a permanent retrievable resource. Executing a program consists of making a reference to the appropriate component. That component, in turn, calls the

required subcomponents. This organization is close to realizing the goal of having programs consist only as a sequence of references to the software archive.

8. IMPLEMENTATION OF THE IDM

8.1 Introduction

In order to validate the ideas set forth in this thesis a prototype CASE system has been developed. This prototype CASE system is based on the IDM and is serving an invaluable role by identifying and clarifying many important issues about design data management and software reuse.

The prototype is implemented on a VAXstation in the Center for Interactive Graphics (CICG), and runs the VMS operating system with the UIS graphics package. The CASE system makes heavy use of the graphics and multi-processing capabilities of the workstation through the ROSE User Interface to UIS (RUF), with almost all input asynchronously driven (AST) by a mouse.

The prototype CASE system includes graphical design editors for displaying program flow of control and program declarations. The library search functions for the software archive are also implemented. In addition, a great many user interface issues were encountered and addressed; a full discussion of these is found in an appendix.

8.2 About the System

The prototype system divides the workstation screen into several regions which are consistently used for the same set of functions. These regions are depicted in Figure 8.1 and are described below.

1. This region is for the main menu bar. This bar is always present on the screen and provides access (via pull-down menus) to the most general functions provided by the CASE System. The first two of these functions are access to the design library and the local workspace of the designer. Next are debugging and toolkit development operators for use of the ROSE database system in the interactive mode. There are also a full set of predefined queries about the system. Finally,

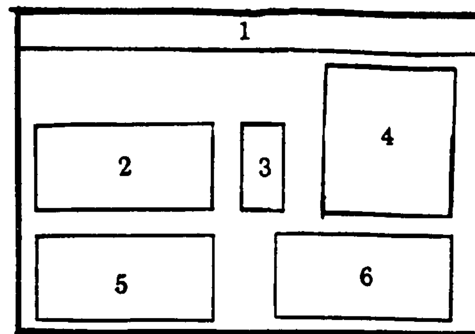


Figure 8.1. Screen Organization of the CASE Tool

there is a standard set of menus for managing the editing windows, with operators for resizing and moving windows, and operators for panning and zooming within the windows. Special variants of the some of these are also provided, such as centering on an object and a pan-to-home function.

2. This region contains the program dynamic structure (PDS) editor. It provides the ability to describe *the flow of program control* in an enhanced type of structure diagram as originaly proposed by [Mye78] and [You75]. The enhancements allow for the pseudocode constructs of sequence, selection, and iteration as explained in [Pou88b] and in chapter 4. This is the primary editor for interactive design in the prototype.
3. This is a function "palette" providing the operations for the PDS editor.
4. This region contains an IPO-style chart. A full discussion of this chart is given in the appendix; in short, it allows the user to view the attributes of a module "on one piece of paper."
5. This is a dialogue box. It is used to output messages and prompts to the user.
6. This is the Program Static Structure (PSS) editor. This provides a top-down depiction in tree form of how the modules in the program are *declared.* For the

purpose of this editor, a module is considered to be an interface and the alternative for that interface that is being considered for use in the current design. The intent is to show and enforce scope constraints in the design. Operations in this window are provided by a pull-down menu.

8.3 A Sample Design Session

8.3.1 Introduction

Here we see how the program designer uses the CASE system to locate existing code to perform a required sort of an integer array. This section seeks to illustrate how the IDM and the software library support the interactive design process by showing how they work together in the CASE system. To do this, a brief overview of the process is first given. This is followed by a more detailed walk-through of a short design session.

8.3.2 Overview of the Design Process

A detailed version of the design process is given in the following section; this synopsis is only an introduction to the philosophy behind IDM and software reuse.

When the program designer encounters a need for some service in a program, he meets that need by calling a subroutine or function. Using the terminology of the IDM, he creates an instance of a call. This call represents an abstract request for service. In further defining this call, the supporting CASE system provides tools that help locate pre-defined interfaces existing in a library of software components that may meet the software requirement. The tools in the CASE system also help the designer locate alternative implementations for specific interfaces. The designer has the option of whether to actually use the library components in his application or to design his own. If he chooses to use the library component, he is said to *bind* that component to the call. He may also make new interface and alternative objects by copying the library routines

and then modifying the new objects in order to customize them for his own application. In any case, the call remains a non-binding request for service independent of the actions the designer has made.

8.4 The Design Session

In this sample design session, the designer chooses to start the design process by making all of the tools visible and accessible. (Since in this example we are only seeking one component, none of these tools are strictly necessary, and all work could be done by simply browsing the library. However, this section also seeks to show how the tools in the CASE system work together throughout program development.) At this point the workstation screen appears as in Figure 8.2.

The designer starts by adding a call icon to the PDS editor by clicking on the **Add Call** icon in the Dynamic Editing palette. By doing this all he has done is stated that he has some undefined software requirement. The PDS editor now draws the undefined call icon on the screen, as shown in Figure 8.3.

In order to define the call *and simultaneously search for existing components that meet the need* of the call, the designer invokes **Edit Call** . The "Search/Create Calls/Interfaces" text entry box is now displayed and activated. This box is shown in Figure 8.4. Since many of the concepts critical to the IDM are implemented as functions from this box, it is worth explaining the many options available at this time.

The reason why this box is so critical, and the title of the box is so long, is that all of the library search functions, the copying functions, and the creation of interfaces, are all done from here. This box also launches the designer into searches for alternative implementations of an interface, as will be seen below.

The evolution of a software requirement and the reuse of software components is the philosophy of IDM and is the underlying motivation while working in this box. As the designer defines his call, he also searches for interfaces that meet or partially meet

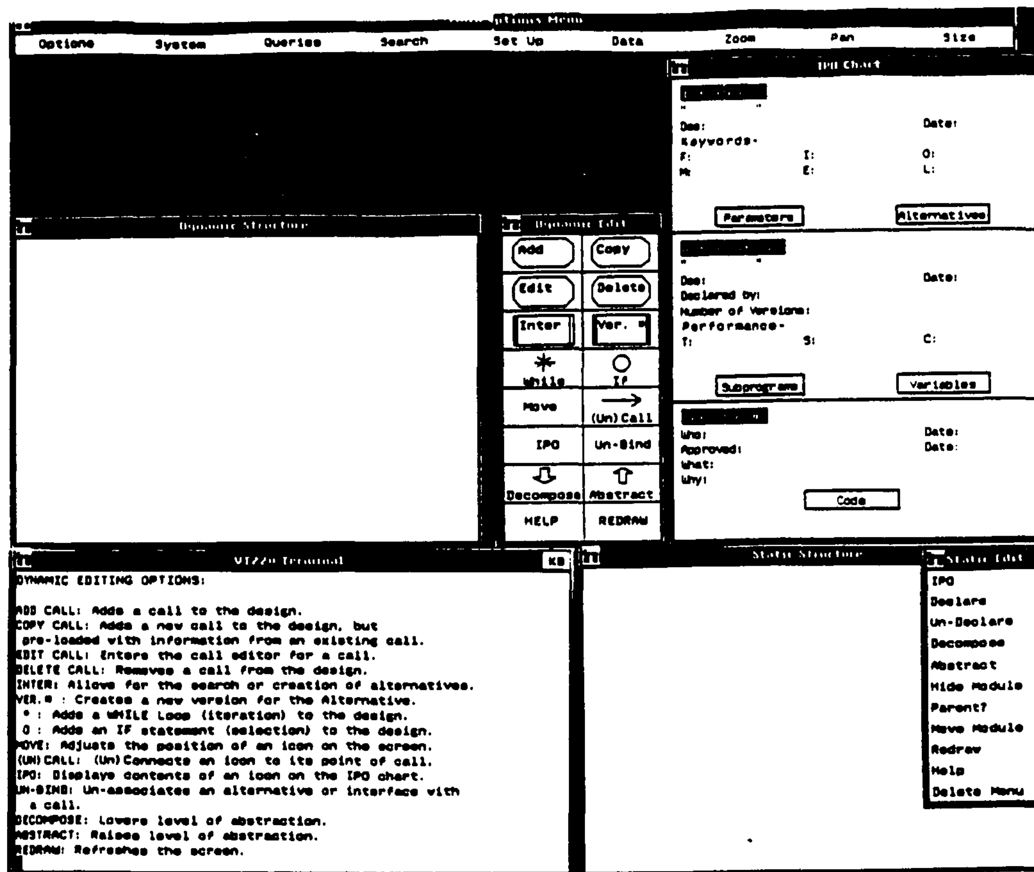


Figure 8.2. The Development Tools in the CASE Prototype

the requirements he is defining. He does this by comparing the entries he makes for the call, which are viewed as *constraints*, against the corresponding entries in the library of module interfaces, which are considered *definitions* of available components. The designer starts (or restarts) his search by selecting the **Search** icon. The single letter icon S next to each attribute box invokes a search on all interfaces over that attribute. The results of this search are used to initialize the next search, or if the result is zero, the last search result (LSR) is automatically retained. The designer may also backup by manually selecting **Last Result.** He may scan through the interfaces found in the last

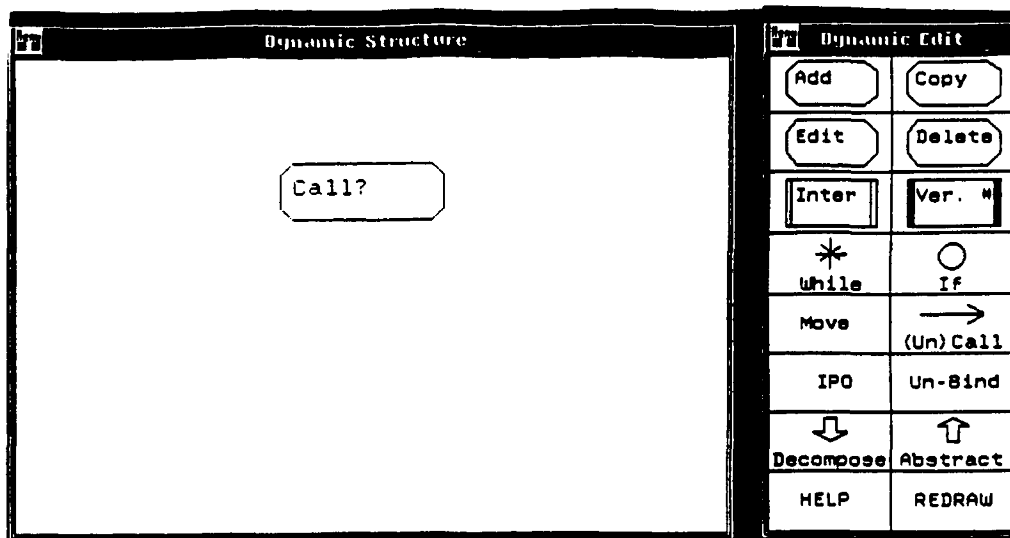


Figure 8.3. An Undefined Call

search by using the **Browse LSR** option; interfaces will appear on the IPO chart. Of course, if at any time the designer needs an explanation of these operations, there is assistance provided by **Help**, and if he needs suggestions for search keys for any particular attribute he may select the single letter **H** next to that attribute.

To illustrate the search facility, we pick up the design after several entries have been made. The designer names the call "Sort_an_integer_array" with "I_Sort?" as it's short name, or *tag*. He also has filled some other administrative information. He has selected **Search** to initialize the search for interfaces that might sort an integer array. Right now the search space contains all the interfaces in the software library. He then makes the entry "Sort" in the Function box for the call and clicks **S** as shown in Figure 8.5. The dialogue box advises him:

There were 4 interfaces with that function.

Encouraged, he requests help on interface inputs and clicks the **H** next to the "Input" box. The dialogue box advises him that the four interfaces found in the last

Figure 8.4. Search/Create Calls/Interfaces

search have the inputs:

Real Array

Integer Array

Linked List

Matrix

Since "Integer Array" most closely matches his need, the designer makes that entry in the "Input" box, and clicks **S.** The system advises him that there is only one interface in the library that has that input (and is a sort function). He displays this interface on the IPO chart by clicking **Browse LSR,** as is shown in Figure 8.6. Note that because no alternative has yet been specified, this part of the IPO chart is left blank. The designer may also view the parameters and names of existing alternatives for the

Figure 8.5. Searching for a Sort Function

interface by clicking on the appropriate icons on the IPO chart. The results are listed in the dialogue box.

At this point, the designer has several options. He may elect to copy the interface information (**Copy**)for the integer array sort into the call he is editing. If he chooses to do this he finishes defining the call, by giving the call the attributes of the interface that was found. This would also allow him to edit the call information and parameters any way he likes, and then later make a new, custom interface for his application (using **Make Int**). Furthermore, he could bind the interface to the call using **Bind**, thereby telling the system that all future references to this call should automatically reference this interface. This action would be reflected in the PDS editor by replacing the call icon with an interface icon, as shown in Figure 8.7. The key point

IPO Chart

Integer_Array_Sort
"Int Sort "
Des: Jeffrey S. Poulin Date: 21Sept88
Keywords-
F: Sort I: Integer Array O: Integer Array
M: Buffer E: MVS L: PL/I
This routine sorts an array of integers from low

[Parameters] [Alternatives]

" "
Des: Date:
Declared by:
Number of Versions:
Performance-
T: S: C:

[Subprograms] [Variables]

Who: Date:
Approved: Date:
What:
Why:

[Code]

Figure 8.6. IPO Chart for Integer Array Sort Interface

is the designer may define as little or as much as he likes, knowing that the call will remember the definitions and also allow any future modifications should he change his mind. In our case the designer elects to investigate alternatives for this interface and invokes **Alternatives,** resulting in the tool shown in Figure 8.8.

Since this box looks and operates in a manner similar to the box for calls and interfaces, a detailed explanation here is spared. The designer is satisfied with the "Quick_Sort_Method" alternative of the "Integer_Array_Sort," and now scans the versions of this alternative using **Versions.** Here he has the option of viewing the history and pseudocode of each version and selecting one to be a current version. The tool for these actions is shown in Figure 8.9. Note that this step is also not necessary,

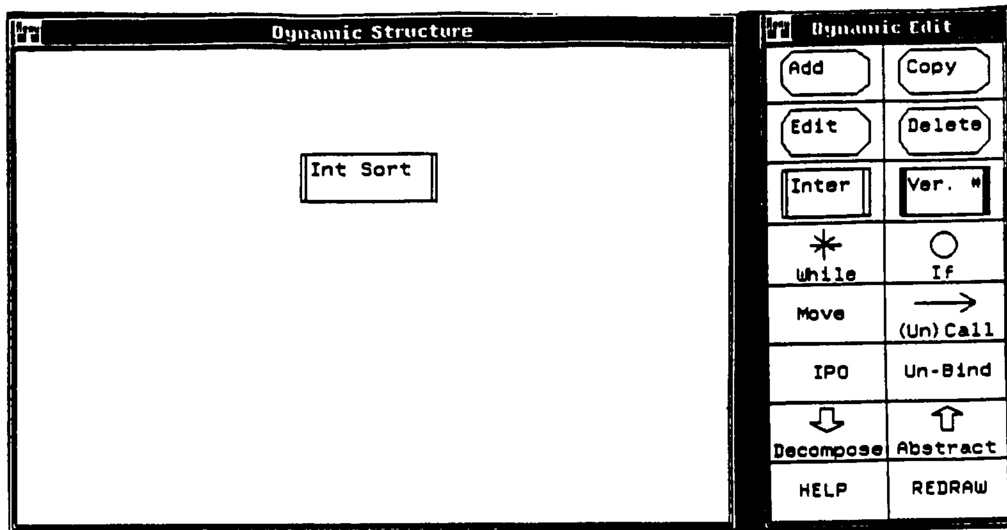


Figure 8.7. The Interface Icon for the Integer Array Sort

and if skipped will result in the version now identified as the current version to be used in the application. Also note that in the future a large variety of documentation schemas will be provided to document the actions of the version.

To complete this example, the designer has clicked **Bind, IPO Call,** and has exited back to the PDS editor. This combination of actions is reflected in Figures 8.10 and 8.11. Note that once the interface and alternative for this module have been added to the design, the Program Static Structure editor automatically updates itself, as shown in Figure 8.12.

What the designer has successfully done is to interactively develop a requirement for a software service into a completely defined piece of code using existing components from a reusable software library. The IDM is central to his ability to do this. It allows him the ability to evolve an idea without having to be exact, secure with the knowledge that he can define as little or as much as he likes, with the ability to

Figure 8.8. Search/Create Alternatives

Figure 8.9. Scan Versions

change the requirements at any time without any penalty. The prototype implementation of this philosophy is demonstrating that the IDM and software library

IPO Chart

Integer_Array_Sort
"Int Sort "
Des: Jeffrey S. Poulin Date: 21Sept88
Keywords-
F: Sort I: Integer Array O: Integer Array
M: Buffer E: MVS L: PL/I
This routine sorts an array of integers from low
Parameters Alternatives

Quick_Sort_Method
"Quick "
Des: Jeffrey Poulin Date: 26Sept88
Declared by: Some_Kind_Of_Sort_Package_Array_
Number of Versions: 1
Performance-
T: Very Fast S: Pretty Much C: Print Mngr
This sorts an integer array using the quicksort a
Subprograms Variables

1
Who: Jeffrey S. Poulin Date: 12Oct88
Approved: M. Hardwick Date: 13Oct88
What: This first version is the result of the prog
Why: Completed first design review.
Code

Figure 8.10. IPO Chart of the Integer Array Sort Module

provide a viable method for supporting reuse and software design in a CASE system.

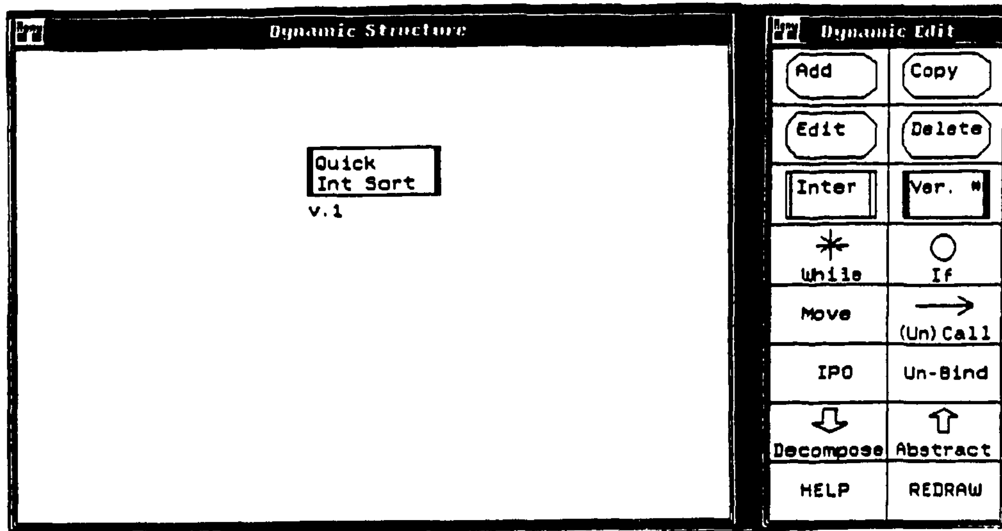


Figure 8.11. Resultant PDS Diagram

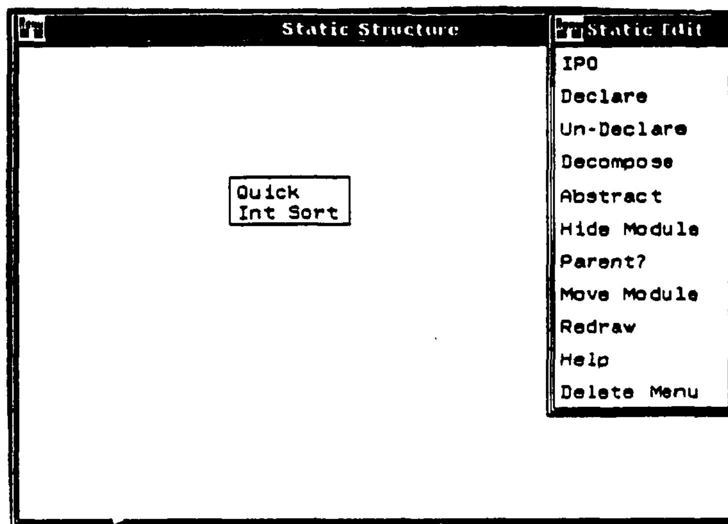


Figure 8.12. Resultant PSS Diagram

9. EVALUATION OF THE IDM

9.1 The IDM as a Partial Solution to Reusability in CASE

The IDM addresses several design object modeling problems that current conventional and CAD data models fail to address. The first of these is the ability to "see" inside implementations of design objects; in traditional object-oriented programming this is not permitted. However, in order for a designer to use a module with confidence, some of this information *must* be accessible. This is an important philosophical addition to an object-oriented world.

By dividing the molecular interface into a requirements and a definition portion, the IDM permits a level of flexibility during the design process that is not possible with the molecular model. Since molecular interfaces define modules, they cannot be modified, thereby unnecessarily tying the designer's hands as he seeks to develop his program. However, the IDM is intended for just this type of interactive, dynamic approach, and allows requirement specifications to grow as the project grows.

Unlike other CAD data models, the IDM provides support for all stages of the software engineering lifecycle. This is possible at high levels of design and during product maintenance because the IDM inherently stores requirement specifications as part of the model. At the middle stages of design, the three IDM constructs reflect software modules and how they interact, both from a control flow and a declaration viewpoint. At the code level, the language-independent pseudocode representation reflects the source code constructs that are required in any structured programming language.

Finally, no solution as simple as a semantic model for design data will solve all of the problems of software engineering and CASE. The IDM seeks only to make a small step towards more efficient software development and the production of a higher quality software product. No panacea or greater claim is made. However, it is strongly

believed that in those areas of software engineering that the IDM addresses, the IDM does an exceptional job in meeting those tasks.

Due to the lack of a numerical evaluation method for comparing software engineering processes, this kind of evaluation of the IDM is not provided. While every attempt at objectivity is maintained, a mathematical approach to evaluation of this research is not possible due to the difficulty in quantifying software engineering results [Duf89]. The same is true when comparing the IDM with current CAD data models. For this reason, any evaluation of this type of research will contain a significant subjective element.

An extensive review of research done in the fields of data modeling, software engineering, and database techniques has shown that other authors are often faced with this same lack of a quantitative evaluation method. The approach commonly taken is to undertake a thorough discussion of the advantages and disadvantages of the new technique. For this reason, this chapter uses this approach. First, the major issue of semantic storage of design data is discussed. Next, the issues of design capture, data classification, component retrieval, and software library organization are presented. In each case, the strengths and weaknesses of the new IDM's approaches are forwarded.

9.2 Storage of Design Data

9.2.1 Advantages

Until recently, the dominant tendency in database systems has been to work with available techniques and to ignore data semantics, although the object-based technologies and particularly the object-oriented technologies have attempted to close the semantic gap beteeen data and reality. Most of the software-oriented database work has been based on models that can be viewed as simple extensions of the relational and entity-relationship models. The inadequacy of these models for software design support

has been broadly acknowledged [Bat84, Has82, Hel87, Sid80, Web88].

The IDM takes the object-oriented approach to data modeling, adding a set of semantic rules that mirror the software product that is being created. Unlike other data models, the IDM grants the designer a great amount of flexibility during the design process by providing a place for the evolution of product requirements and constraints. These requirements and constraints are stored as part of the IDM, thereby providing a record of the software development process for documentation purposes as well as providing a means through which to retrieve candidate components for reuse.

The storage model has good performance expectations for large-scale systems. Each object in the IDM is relatively small, allowing it to be manipulated by the database system and the operating system with a mimimum of operations and disk accesses. This is in keeping with the ROSE philosophy in the performance optimizations of an object-oriented database system [Har87a]. Furthermore, the boundary of each object in the IDM is well-defined. Retrieval of one design object does not require the automatic retrieval of all objects referenced by it; this action is postponed until specifically required. This prevents a proliferation of I/O requests for relatively minor operations.

9.2.2 Disadvantages

The primary drawback to this model centers on the semantics regarding the modification of existing interfaces. The valid operations on the model do not include the ability to edit an interface once it has been created. This is because changing the definition of a module invalidates the implementations of the module, and compromises all places where that module might be used. This is traditionally a major concern in database systems. In the IDM, object integrity is guaranteed by preventing such an action altogether. The process required to modify an existing interface is to copy the old interface into a new call object, and then to edit the call as desired. At this point, the

new call may be made into an interface object, or saved as a call object for even further modification at a later time. However, this is not so much a disadvantage as it is an inconvenience.

The positive side of this restriction on the user's modification of interfaces is that the process explicitly requires the designer to ensure that no type conflicts are created by his action. In fact, no type conflict *can* arise. Now consider the case where a routine that is currently in use throughout a design requires a change to its interface. The semantics of the IDM prevent modification of the interface because the consequences of such a change made uniformly in so many locations cannot be predicted. However, design engineers are often much less concerned about immediate integrity constraints, and may wish to do exactly this kind of universal modification. If this is the intention of the designer, he can proceed in the following fashion: First, construct the new interface according to the semantics of the IDM, as described in Chapter 3. Second, retrieve all call objects in the database that have the old interface bound to them. Finally, set the Bound_int fields of these call objects to the new interface name.

A variation of this situation is discussed in Appendix II, Section 15.2.1. It is important to note that the prototype implementation strictly implements the valid IDM operations as described in Chapter 3 and in Appendix II. Without entering the ROSE database system in the interpretive mode, it is impossible to operate on a program design other than in a semantically approved fashion. Note that in the situation described in the appendix, the prototype implementation is based on a design decision that is model independent. In order to circumvent the "inconvenience" that this disadvantage creates, a commercial implementation of a CASE system based on this model might add a *replace* operation to the valid operations on interfaces. Once again, this is a design decision based on database concerns and not engineering practice.

Finally, the IDM was developed with a heavy orientation towards structured programming and the major software engineering methodologies, all of which support structured programming. The data model is not intended for the design of parrallel computing algorithms, concurrent computing algorithms, non-sequential imbedded systems controllers, and object-oriented programming with methods, such as in Smalltalk.

9.3 Capture of Design Data

9.3.1 Advantages

The method developed for data capture with the IDM is a direct reflection of the major top-down, structured programming methodologies as well as the semantic objects that comprise the IDM. In the Program Dynamic Structure editor, the system incorporates a "who-calls-who" orientation in a tree-style format that quickly shows control flow dependencies. In the Program Static Structure editor, the system further shows the declaration scheme of the program, and is useful for answering questions about scoping rules.

In addition to these kinds of information, the various IDM icon shapes and the spatial relationships between the icons provide a very fast and effective method for conveying the status of the program design. A quick glance at the diagram reveals which software requests are currently unsatisfied, and which modules are currently undeclared. This desirable characteristic is an attribute of *visually deep* diagramming methods.

The PDS and PSS diagrams not only provide a methodology for software design that matches the design process, but they also provide a tool that is flexible enough for use throughout the software lifecycle. The abstract representation of a request-for-service that exists in the call is suitable for the development of product

requirements, whereas the pseudocode representation of the code statements present in the alternative object relieve the model of any language dependence.

An additional problem with classical representation schemes is that they tend to promote understanding and communication among the development team up to the implementation stage of a project, but at that time all the design information is effectively discarded. This is the result of the unfortunate consequence that there is, for the most part, currently no way to integrate design information with the code, or to otherwise effectively maintain, manage, or reuse it. Moreover, the fact remains that many large, complex computer systems have been designed without any specific methodology to guide the software development process [Web88].

The IDM eliminates this problem in a natural way. The model is by itself a direct reflection of the PDS and PSS design diagrams that created it. The design documentation is therefore always available. The model also has provisions for incorporating other documentation data into the object schema, so it is possible for other types of diagrams to be included in the database for design or documentation purposes.

Large scale systems can be visually represented in a satisfying manner through the abstraction and generalization mechanisms built into the diagram editors. The abstraction of a part of a design results in the hiding of unnecessary clutter and detail, and increases the response time of most global operations by restricting the traversal of the diagram to unabstracted objects. Generalization of all or a portion of the diagram reverses the abstraction operation, thereby revealing the details of a design. This concept is further developed in the section "Economy of Scale," below.

9.3.2 Disadvantages

As with any design tool that is forced to use a finite space to represent large designs, there are problems concerning the amount of information and detail that can be usefully displayed at one time. While window resizing, zooming, panning, and the

abstraction and generalization operations help in this regard, the PDS and PSS diagrams do not entirely solve this problem.

There are several features that could be added to the PDS diagram in the future. The most notable of these is to enhance the diagram with a visual indication of the suitability of a call and the objects that are bound to it. For example, if the interface currently bound to the call only partially satisfies the constraints of the call, this should be expressed in the diagram. The visual indicator should also indicate, at a glance, the level of conflict currently existing between the call and bound objects. In this way, bound objects violating five constraints would be stressed over those violating only one constraint.

Furthermore, the design diagrams only provide a limited bottom-up design capability. This is because of the dichotomy the reusability problem creates with top-down programming advocates. The top-down designers suggest that all programming problems should be progressively subdivided until the problems become so small that they can be easily performed by one software module. The idea of reusability, however, is to incorporate as much existing code as possible into the design in order to increase productivity and decrease effort. This is inherently a bottom-up activity. The issue becomes one of deciding at what point the top-downers should start making design decisions based on the availability of reusable components, and of how much the bottom-uppers should force existing parts into a design. Although both methods are supported by the IDM and the graphical tools, the strength of the model is in supporting a top-down approach to programming.

Perhaps the greatest criticism of the PDS diagram is that is fails to remove the designer from the semantic structure of the database, which is a drawback for those unfamiliar with data modeling concepts and/or database systems. The intention is to forward a design method based on the IDM that has a high emphasis on reuse.

However, the graphical tools should be as general as possible in order to appeal to those unaccustomed to the technique or who do not wish to learn it. This problem can be partly addressed by integrating other design methods into the system.

9.4 Classification of Software Components

9.4.1 Advantages

While there are many classification techniques currently in use and under study for various kinds of libraries, there is significant difficulty surrounding the classification of objects that are as abstract as software. The mainstream in research and the tendency in application systems is to classify reusable software components using some form of descriptive vector that is composed of a finite set of keywords. The length of this vector can be fixed or of any length.

The IDM has been shown to fully support these keyword classification schema and do so efficiently. By storing the classification criteria in the IDM objects, no additional overhead is required in the library system beyond the management of the design objects themselves.

9.4.2 Disadvantages

The use of a keyword based schema for the classification of software has been fully addressed in Chapter 5. The largest problem with these methods is the lack of precision; the choice of keywords and their attributes is a subjective decision made by the designer or librarian. Part of this problem is due to the abstract nature of the design objects, but part of the problem is due to the manual method of selecting keywords and their values. Incorporating an automatic or machine-assisted tool to classify the objects might result in more precise definitions.

The most common tool for this purpose that has proven to be very effective in a CASE system is a thesaurus or "valid word" list [Fra87, Iso87]. The lack of such a

thesaurus in the IDM prototype makes it difficult for a designer to find objects that have been classified with similar, but not compatible keywords. The lack of this tool also increases the number of similar values that an attribute might contain, making searches more difficult from a designer's point of view and less efficient from a database point of view. Some form of vocabulary control mechanism is clearly needed.

9.5 Retrieval of Software Components

9.5.1 Advantages

Matching vague and abstract requirement statements to those components in a reusable software library that are able to fill those needs is one of the most challenging aspects of software reuse. Any retrieval technique must use all of the information available to it, such as parameter lists and module performance attributes. However, in order to do this, the technique is dependent on the method used to store and classify the design data. In the IDM, the product requirements and constraints are stored in the call, where they can be modified, developed, and saved. These requirements are later matched by a retrieval algorithm with interfaces and alternatives that have the same values for corresponding attributes.

There have been two retrieval algorithms implemented as part of this research. These methods involve accessing the keywords in the object classification schema and comparing their values against those in the software requirements. The primary difference between the two techniques is the use of multilist indices in one of the methods in order to provide faster access to large data sets. These indices are external to the model and are independent of the IDM. Both methods, however, demonstrate that the IDM is capable of supporting an effective retrieval mechanism in a CASE system. Furthermore, by creating and maintaining multilist indices for search keys, the model is capable of supporting large-scale queries at relational database speeds. It is believed

that this can be a tremendous asset in an industrial strength version of a CASE system based on the IDM.

9.5.2 Disadvantages

The issue of design object retrieval is by itself a major research topic, critical to the reuse problem. In many ways it is the most important of the reusability issues, because in one way or another retrieval depends on all of the other issues; on how the information is presented to the user, how the information is classified, and how it is stored, both in the immediate database workspace and in longer term public libraries.

The retrieval techniques used with the IDM are database standards, but lack extensive heuristics that might be found with a knowledge-based search assistant or more "intelligent" system. The use of these advanced searching concepts could greatly increase the response and usefulness of a CASE system. Although it is not possible to predict performance improvements, even the incorporation of a thesaurus as discussed above could make object retrieval significantly more efficient.

An additional shortcoming and possible enhancement to the current retrieval algorithm would be to incorporate a partial matching mechanism for automatically retrieving those objects that meet a subset of the call's constraints. The retrieved objects could then be ranked in order of their suitability. Currently, the implemented retrieval algorithm finds only those objects that meet *all* the call constraints. As above, such an enhancement would make object retrieval a more intelligent and user-friendly process.

9.6 Organization of the Software Archive

9.6.1 Advantages

The design objects in the archive represent all of the complete and functional software components developed to date. While these objects are related through references known as *calls*, from an external viewpoint the archive is comprised of a

collection of "equal opportunity" components. There is no boundary, physical or conceptual, that prevents the selection of any component. This means that a component that might have been developed for a totally unrelated application has the same chance of being accessed as any other component while the designer is conducting a search for reusable modules.

The IDM software library is not only divided by object type, but also into a public archive and a private workspace. The public archive contains approved and active design objects, while the private area contains modules under development. This division of the software library has several advantages. First, by separating the modules under development from those in active use, the integrity of the data in the public archive is guaranteed. Second, this division is readily adaptable for use in a distributed design environment. Not only is this the most common form of CASE environment, but is particularly advantageous when developing large scale software systems where hundreds and perhaps thousands of programmers are involved in the design and/or maintenance of a program.

9.6.2 Disadvantages

The organization of the public and private workspaces assumes that the host operating system has no practical upper limit on the number of files that can be maintained in a file system directory. As designs grow, and especially as the public archive grows, the number of files in these directories can become enourmous. The library organization and the database depend on the operating system to manage this potential growth.

9.6.3 Economy of Scale

One major concern in any CASE system must be how the system will perform when extremely large quantities of information are being accessed and manipulated.

Specifically, will the system response time unacceptably deteriorate? The organization of the prototype library allows a heuristic that is aimed at keeping system response times fast.

This heuristic is to keep the contents of main memory to a bare minimum. The strategy is to have something in memory *only if absolutely necessary.* Normally, when designing a large system, the user will only work on a small part of the actual design. With the IDM library organization, only this small part of the design will exist in memory; the rest remains in secondary storage until needed. The designer can save this small part in his workspace and recall it at will. He may also request that other parts of the design be loaded, either explicitly (by name) or automatically (by navigating the structures in the PSS and PDS editors). When a call, interface, or alternative object is required for the first time, it is read from disk. Using this strategy, the amount of information typically in active use will be restricted to dozens of objects rather than thousands, with a corresponding speedup in response time. This strategy is termed *the economy of scale,* and is possible through this library organization by direct access and retrieval of the required object in the appropriate directory.

9.6.4 Levels of Abstraction

An additional feature which is possible primarily because of this library organization centers on the desire to view the program from multiple levels of abstraction [Rov88]. A high level of abstraction is to view a program as if it were just an interface or a call, without considering the subroutines it uses to implement the action. For example, in the prototype CASE system, abstracting a node in either of the two tree diagrams has the effect of hiding the children (and grandchildren, recursively) from view. In contrast, a low level of abstraction is to see how the module does it's job, in other words, who it calls and who it declares. In the CASE system, decomposing a node in one of the tree diagrams causes the immediate children of that node to be

displayed. If the information about these nodes is not in memory, it is first read from secondary storage.

This feature has several advantages. First, and most importantly, it supports the IDM, molecular object, and object-oriented concepts of viewing modules in a black-box and in a white-box fashion. The second advantage is that it supports the top-down structured design methodology. And finally, in keeping with the above stategy of "economy of scale," it keeps the CASE system performing at optimal efficiency. By organizing the library by object type, and storing the objects by primary key index, these object can remain in the archive on secondary storage until needed, and yet still be located at main memory speeds.

10. RELATED WORK

10.1 Design Data Management in CASE Systems

There has been a lot of interest in database system support for the software development life-cycle [Blu87, Bro87, Cam83, Day83, Mat87, MHS86, Olu83, Onu87]. Most of the articles addressing this issue give extensive consideration to the requirements for a CASE system, but do not explore detailed solutions to these requirements [Ber87, Nes86, Rom87, Sid80, Yau87]. The following paragraphs provide a summary of recent work in this area.

Numerous commercial systems are currently available through CASE vendors [Dig88, Bal85]. These products can be grouped into four categories, depending on the amount of support they provide to the software engineering process. The first group allows the designer to design a program using one or several of the major diagramming techniques for design. They incorporate some semantic checking of the diagrams that have been created, but are little more than customized drawing routines [Rou83]. The second, smaller, group advertises some level of semantic checking and data dictionary support of the diagrams that have been created. This group typically uses a database to manage the design data, and this database is almost always relational [Cad87, GlL86, Gut82, IDE87, Was87]. The third group, which comprises only a few products, adds to the functionality of the second group by producing pseudocode skeletons from the design diagrams, or some other form of output that may be useful to later stages in the design process. The final group is made up of highly specialized products, such as those that automatically generate "structured COBOL" programs from unstructured ones, and therefore are outside the scope of this work.

Many of the applications related to software development opt to use an abstract syntax tree representation of the program to manage the design data [Alb84, ReS85]. There are several advantages to this. First, there is nearly a one-to-one mapping

between this representation and code, so automatic code generation is generally possible. Second, incremental parsing of the program design is possible after nearly every syntax change. However, databases are not employed to support abstract syntax trees, as it is considered much more efficient to use linked-lists of records to simulate the tree [Pow83], or alternately, to use the list management facilities in LISP [GE87]. However, because the data structure so closely models the Backus-Naur description of the programming language, this class of systems is generally oriented at and limited to low-level design and program development.

10.2 Existing Systems for CASE

10.2.1 Introduction

Not only has there been an intense research effort in CASE, but a strong commercial interest as well. From a business perspective, a company that is paying a programmer a fairly substantial salary every year is making a sound investment in any product that has the potential to multiply his productivity. For this reason there are numerous CASE packages available on the market and described in technical literature. Journals such as *IEEE Software* commonly carry advertisements for CASE systems that specialize in areas ranging from source code version management to graphical design. In place of a survey of the state of the art in this area, I describe several representative systems below.

10.2.2 Software Through Pictures

Interactive Development Environments (IDE) is a company founded by a pioneer of this field, Anthony Wasserman. IDE markets the Software through Pictures system, which supplies editors for data flow, program structure, data structure, entity relationship and transition (finite state machine) diagrams. The system runs on several major workstations and uses a relational database for data management. It is

advertised as being flexible and easily extendible for custom situations, as a result of it's "open architecture" design. An example of an IDE editing window is shown in Figure 10.1.

Although the IDE system provides numerous graphical editing capabilities, any one part of the target program can be designed using only one of the views. Several schemes may complement each other, however, such as a text description of a module designed with the data flow editor. Each part of the overall program design is stored in it's own database relation, and consistency checks with the rest of the program are done at the express order of the designer. IDE is a "high level" tool aimed more at programming in the large and does not provide a means to work directly with source code [IDE87].

10.2.3 Pecan

The Pecan family of program development systems is the research product of Steven Reiss at Brown University. It is, in contrast to Software through Pictures, a low level design and programming environment. Therefore the graphical tools provided are more code-oriented; Pecan supports standard flow charts, Nassi-Schneiderman diagrams, a structure chart, and a syntax tree representation. The latter is due to the representation of the program internal to Pecan. There is no database supporting the design. Rather, the program is stored as an abstract syntax tree. This allows for much more syntax directed checking, as well as incremental parsing of the code. Pecan also includes a text editor for source code and run-time facilities for visual program debugging are provided [Reis85].

10.2.4 Interactive Ada Workstation

The Interactive Ada Workstation is under development at the GE research and development center in Schenectady, New York, for the U.S. Department of Defense.

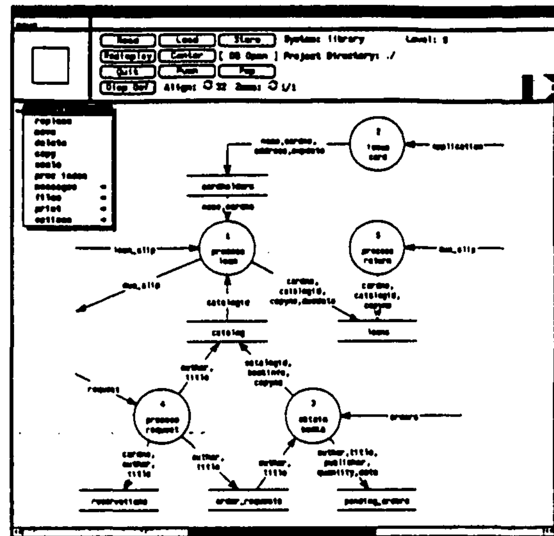


Figure 10.1. IDE Data Flow Editor

The primary high level tool that the Workstation supports is the Buhr diagram, which is a diagramming convention developed specifically for the Ada language and has the ability to represent concurrent Ada tasks. The Workstation supports or will support several low level design tools including a finite state machine editor, decision table editor, truth tables (a special case of the decision table) and Nassi-Schneiderman diagrams. The Workstation was programmed in LISP and uses that language's resident list processing facilities instead of a database to manage the design data. The internal representation of the design is, like the Pecan system, a syntax tree. The Workstation allows the programmer to edit source text and run Ada programs. It supports incremental compilation and can automatically generate code from diagrams such as the finite state machine [GE87].

10.3 Semantic Data Models for Design Data

10.3.1 For CAD/CAM

There is no shortage of literature on data modeling problems in general CAD/CAM or VLSI applications. As discussed in Chapter 2, many of these applications use the molecular object approach to model design objects [Bat84, Bat85, Buc85, Has82, Hel87, Kat85, Sto87]. Recently, however, many of these researchers have turned their attention to issues such as version control, choosing to remain with the molecular object data model. [Bha87, Dit88, Kat86, Kat87, McL83].

Two novel approaches to design data modeling come from recent doctoral dissertations. One is the Design Object Model (DOM) by [Bap86]. In the DOM, almost every concept related to the design is encapsulated into an object along with a set of allowable operations. Objects related to the design in DOM include interfaces, implementations, views, components, interconnections, evolutions, schemas, instances, copies, definitions, and for anything not included in the above list, a generic object. The model in this proposal is conceptually much simpler than the DOM approach because it encapsulates most of the design issues into the module object itself, thereby allowing the designer and database to deal with the issues in a unified manner.

The second project is by Stephanie Cammarata, and is oriented towards a mechanical design, engineering, and manufacturing application where she works. The data model in her thesis concentrates on storing the product definition data that is generated in the initial design phases. Her model is constructed from four basic components; *intentions*, *instances*, *descriptions*, and *extensions*. An intention corresponds to a generic, prototype object. An instance represents a real world object, and is a copy, or instantiation, of the intention. The description contains the values of the attributes of the instance. The entire design is comprised of the set of all design object instances, and is referred to as the extension. The Cammarata model differs from the model in this

proposal in that it is heavily based on set theory and predicate logic.

10.3.2 For Software Engineering

One major research project in the area of CASE is an on-going effort at the University of Colorado called Cactis [Hud87, Hud88]. Cactis is designed to support the construction of objects and type/subtype hierarchies, which are useful for managing the complex data found in software environments. In fact, Cactis is a complex object model that references other objects via the *relationship*. This relationship is a very simple interface, consisting of only the number, type, and direction of the values in what is essentially a parameter list. Cactis concentrates on functionally defined and derived information, and considers the multiple references between objects as creating an object base similar to an attributed graph. Through the use of well-known graph algorithms, this representation allows the Cactis system to support the design environment while retaining good performance characteristics. Unlike the model in this proposal, Cactis allows some procedurally defined data which gives an object local behavior. Like the interface in the molecular object model, the simplicity of the Cactis interface allows a lot of flexibility in filling sockets in the design; however, no support for this action is addressed. Finally, Cactis admittedly is not meant to support real-time graphic editing and checking efficiently.

Recent work at Brown University has investigated object-oriented database support for "conceptual" programming [ReS86, ReS87, Van84]. This system, called GARDEN, is a programming rather than a design tool and utilizes a Smalltalk-style approach for its objects.

11. CONTRIBUTIONS TO THE FIELD

11.1 Introduction

The field of CASE is relatively young and constantly in search of new ideas. Of the many issues currently being investigated in CASE, several important ones are being addressed here. These include not only the primary concern of data modeling, but the integration of major software design techniques, a friendly and functional user interface, and, especially, a strong reusability capability. Most of these topics are discussed in detail as part of the evaluation of the IDM in Chapter 9.

The primary contribution to the field of software engineering made by this model is the high emphasis and support it provides for the reuse of software components. By specializing the interface to handle the two roles it assumes in the design process, and by opening parts of the implementation of a design object to the user, the IDM model assists designers seeking a flexible way to mate requirements with availability. With the pending crisis facing the supply and demand for software [Weg84], CASE systems that follow the CAD/CAM approach of assembling new products from existing components will be greatly needed.

This chapter analyzes the contributions of the IDM to the field of CASE from the perspective of the definition of a data model for CASE reusability

$$\mathbf{RDM} = (\mathbf{CCDM}, \mathbf{G}, \mathbf{S}_{\mathbf{class}}, \mathbf{R}, \mathbf{P}_{\mathbf{big}})$$

introduced in Section 2.6. The three elements of the generic data model **GDM** are extensively discussed in the Appendices; Appendix I details the data model structure **S**, Appendix II details the allowable operations **O** as well as the constraints **C** on the data structure and the operations. The remaining elements of the **RDM** represent the special attributes of CAD, CASE and software reuse. They are comprised of fifteen requirements, also introduced in Section 2.6, and are summarized in the table below. In this table, the IDM is rated on a three-point scale according to the level of support it

gives to each requirement; either weak, average, or strong. Also included in the table are examples of other data models that do particularly well, or particularly poorly, in meeting the requirement. A justification for these classifications is given as part of the evaluation of the IDM in the sections that follow.

The analysis of each reusability requirement identifies how the IDM contributes to CAD, CASE, and software engineering by addressing that issue. Since most of the contributions that the IDM makes to the field of software engineering apply equally well to the field of CAD/CAM, these contributions are included in the discussion where appropriate.

IDM Evaluation Summary

Criteria for CASE / Reusability	*IDM*			*Other Models that do Well*	*Other Models that do Poorly*
	W	A	S		
1. Provides Conceptual View of Design Data			X	Molecular	Functional
2. Provides Conceptual View of Engineering Process			X		Complex Objects
3. Supports Multiple Implementations and Versions		X		Complex Objects	Relational
4. Efficiently Models All CAD Structures			X	Complex Objects	Traditional Data Models
5. Permits Access to Implementation Attributes			X		Molecular
6. Has Distinct Object Boundaries			X	Object in Field	Functional, Complex Objects
7. Models Complete Lifecycle		X		Complex Objects	Relational
8. Represents Complex Data Types		X		Complex Objects	Functional
9. Suitable for Graphical Design			X	Molecular	QUEL as a Data Type
10. Contains Classification Criteria			X		Molecular
11. Separates Object Requirements and Object Capabilities			X		Molecular
12. Retrieves Objects using Abstract Criteria			X		
13. Can Organize Archive for Distributed Systems			X		
14. Can Organize Archive for Data Sharing and Integrity Constraints		X			
15. Suitable for Large Scale Applications			X	Relational	QUEL as a Data Type

KEY: *W=Weak, A=Average, S=Strong*

NOTE: The contents of this table are justified in the following sections.

11.2 Contributions to Software Engineering and CAD/CAM

11.2.1 To Semantic Modeling of CAD Data

1. *Model must mirror the designer's conceptual view of data.*

Evidence in recent research reveals that this feature is supported by logically separating design objects into interfaces and implementations in the data model, as is done in the molecular data model. This separation follows the object-oriented design paradigm where the program modules are viewed from two perspectives; the overall functional viewpoint that the user of the module sees, and the detailed specification viewpoint that the implementor of the module sees. The IDM takes this same approach to modeling design objects, allowing the designer to manipulate only the functional definition of a module as an interface object, or to manipulate the implementation of the function as an alternative object. An example of a model that does not do this well is the functional model, which, because of its mathematical "argument-in," "argument-out" orientation, makes it difficult to visualize the composition of the design objects.

In order to represent flow of control in the program design, interfaces and implementations in the design reference each other through the call object, which represents a subprogram call. However, this call has a very abstract constitution, allowing the designer the flexibility to change and develop requirements at any point in the program, as well as providing the operations to help him locate existing components to meet the needs that are specified. This approach to software design concepts is unique to the IDM.

For CAD/CAM, the contribution of the IDM in this thesis is to provide a new approach to modeling molecular objects. The traditional CAD/CAM approach of using the interface portion of the object in two separate roles semantically limits the model. By separating the dual functions of the interface into the declaration

role and the call role, the model is made much more powerful and flexible in the manner that each part is represented.

This new approach to data modeling is a major contribution to the field of CAD and CAD, as will be evident in design systems based on the IDM. Designers will no longer be required to instantiate an instance of an interface object to represent a "socket" in a CAD design. In this way, designers will become aware of the distinction between interfaces used to represent existing reusable parts and interfaces that merely hold a place in the design to "call" some sub-part. The design process will reflect this distinction conceptually as well as graphically. As the designer adapts to this notion and learns to fully develop his requirements and constraints interactively before attempting to implement them, the number of new interfaces and alternatives created during a design will be greatly reduced. This will not only save space and effort, but will also hopefully lead to greater accuracy in the final software product.

2. *Model must mirror the designer's conceptual view of the design process.*

In both CASE and CAD, the design process is incremental and evolutionary. It is a structured procedure that repeatedly reduces large problems into several smaller subproblems. The IDM supports this approach through the semantic structure of the alternative object, where the implementation of a module is modeled as a series of calls to subprograms. The call object further supports the concept of interchangeable parts by providing a variable-shape socket which can adapt to components available to fill it. This contributes to CASE and CAD by providing object structure and operational support for design, unlike models such as complex objects that generally lack the semantics that give meaning and constraints to this process.

Since the division of large problems into smaller ones can also take advantage of both bottom-up and top-down techniques, the IDM supports both of these approaches. In fact, any reuse of existing components is inherently a bottom-up activity. In the IDM, modeling the alternatives of modules as subprogram call encourages the top-down design approach. The binding of existing components from a reusable software library to the call objects encourages the bottom-up approach.

3. *Model must efficiently represent the object structures found in CAD.*

Any CASE or CAD data model must be able to effectively represent recursive, non-recursive, disjoint, and non-disjoint objects. As discussed in Chapter 2, the traditional relational, hierarchical, and network data models universally lack this ability. On the other hand, CAD data models such as complex objects are designed expressly for this purpose and can model all of these object types. It remains to show only that the IDM can also model these object types, as they appear in software.

Any simple program design is an example of a disjoint, non-recursive object, and can be easily represented in the IDM structure. As an example of a recursive structure modeled in the IDM, consider a design for a factorial function, as shown in Figure 11.1. The first call to "Fact!" has bound to it a interface for the factorial function and an alternative that implements the function. The alternative, in turn, has code that includes a recursive call. This call has bound to it the same interface and alternative as the first call. In the graphic representation of this situation note that a more intelligent routing algorithm would draw the line representing the call around the module box rather than through it.

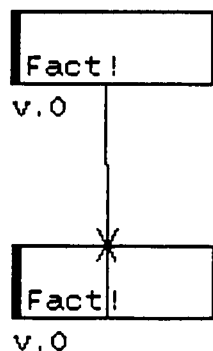


Figure 11.1. A Recursive Call

As an example of a non-disjoint structure, consider two routines, say a mouse manager and a keyboard manager, that need to know the position of the keyboard cursor. They both make calls to a subroutine, "Cursor Position," that supplies this information. Since the original calls from "Mouse Manager" and "Keyboard Manager" are unique, they are represented by two call objects. However, since the same interface and implementation are bound to these calls, all subsequent actions are non-disjoint. Therefore, the calls made by "Cursor Position," namely "Mouse On," "Get X," and "Get Y," are all represented only once.

4. *Model must allow multiple implementations/ configurations/ and versions of a design object.*

The IDM models multiple alternative implementations and versions of modules through the association abstractions described in Chapter 3 and Appendix I. These abstractions are "Alternative_list" and "Version_list." Configurations are considered alternative implementations, since they typically consist of minor variations of an alternative made in order to conform to local site requirements.

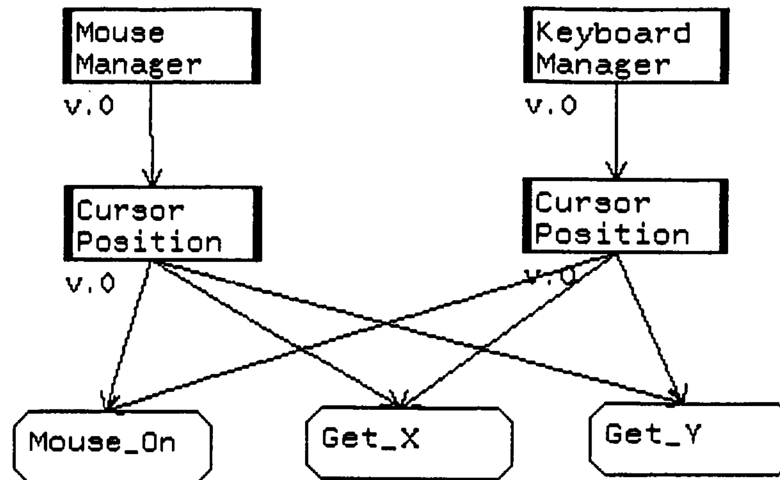


Figure 11.2. A Non-Disjoint Call

The IDM fully represents these concepts, although not in a particularly novel or unique fashion. The technique is similar to that in molecular and complex objects, but is much better than a relational implementation, which is not able to implement version control mechanisms without significant difficulty.

5. *Model must allow ALL externally visible attributes of a design object to be accessible to the designer.*

While most current CAD models, and especially the molecular model, fail on this point, the IDM is an important contribution to the field of software engineering because it clarifies the role that a software interface plays in object-oriented programming methodology. In languages that support separate compilation of module definitions (ModDefs) and module specifications (ModSpecs), the ModDef portion is the only part of the module the designer is allowed to see.

This section of the program may contain only the parameter list of a module and whatever other comments the original programmer chose to include; all details contained in the ModSpec are inaccessible. However, it is likely that more information, particularly about the performance characteristics of the module, must be known before the module can be used. Unfortunately, this information is contained in the ModSpec.

The IDM data model stresses the fact that certain implementation-specific attributes of modules are visible across this interface and should be readily available to the designer. This information is contained in the performance attributes section of the alternative object. These attributes allow the designer to make a more informed decision about the usefulness of a reusable part in a given application.

6. *Each part of the model should have a distinct boundary.*

The IDM has a strict sense of object boundaries in the model, both for the enforcement of structured programming scope rules as well as for database system efficiency. The scope rules are dependent on the declaration structure of the design, which is viewed and manipulated through the PSS diagram editor, and effects the "Declarations" and "Declared_by" fields in the model. Efficiency of database operations is also a major concern, and for this reason the distinctness of object boundaries is strictly adhered to. This issue is fully discussed in Section 9.6.3; the important point is that only those objects *specifically* needed for an operation are retrieved by the database.

The IDM is therefore considered to be very strong with regard to this modeling requirement. Another model that does well in this area is the object-in-a-field model, in which object boundaries are clearly delineated by storing

all the subparts of an object in a single abstraction. Objects that do poorly defining object boundaries are (1) the functional model, in which it is difficult to separate functions which define attributes and functions that define relationships to other objects, and (2) the complex object model, in which one experiences the same problem as with the functional model, except with the tuples that comprise the object.

11.2.2 To Semantic Modeling of CASE Data

7. *Model must support all phases of lifecycle, from requirements through to maintenance.*

The IDM, while being particularly effective in the middle stages of the software lifecyle, depicted as "Product Design" to "Code" in the waterfall model of Figure 1.1, is also capable of supporting earlier and later stages of the design process. Since product requirements are developed and stored as part of the IDM call object, the model can be used during early stages of design to specify product requirements in a top-down fashion. Also, since the call object has a liberal policy regarding the modification of its attributes, the model can be used for developing upgrades to the software as well as during the maintenance phase of programming as the original product requirements change.

Other CAD models, such as complex objects, can be used to manage software lifecycle data, assuming they are given appropriate semantics. Traditional models, however, fail to perform well in this role for the same reasons they do not handle version control well; the database schema is not flexible enough to adapt to the dynamic requirements and design objects of CASE and CAD. The IDM makes a contribution in this area by addressing each of the lifecycle stages in some degree of detail. The planning and requirements phase is the responsibility of the call, the high level design the responsibility of the interface, and low level design the realm of alternatives. Each of these objects plays a part in the maintenance of the

design.

The "code" of the program design is modeled as a series of subprogram calls. Each call is stored in a pseudocode style format that is independent of any implementation. This provides an uniform input to a post-processor or source code generator for conversion to any one of various source languages and environments.

8. *Model must be able to represent the complex data types that are prevalent in software.*

As discussed in the requirement 4, like most CAD models, the IDM can model all four of the basic CAD objects. Since in CASE, interfaces must be able to pass data in the form of records, arrays, and linked-lists, and other such complex structures, the data model should explicitly provide a mechanism for representing the exchange of this information. The data model allows for the construction of these complex data types through the "Declarations" of the alternative object. The full implementation of this feature allows variables and parameters to be declared as any of these user-defined types, as well as any of the basic types more commonly used in programming, such as integer, real, etc. Models that do poorly representing complex data types are all of the traditional models, for the same reasons as they do not represent basic CAD objects well, and the functional model, in which complex compositions are difficult to construct using the binary relationships on which the model is based.

11.2.3 To Capture of Design Data

9. *Model must be compatible with graphical design paradigms.*

The IDM is a model with a strong sense of program structure and flow of program control. This orientation is based on extensive research on the nature of the major software engineering paradigms and the goals of each methodology. Chapter 4 details these findings and the rational behind the new Program Dynamic

Structure and Program Static Structure diagrams that resulted from this study. The IDM and these diagrams are very closely related, based on this research. Therefore, it is no surprise that the editing operations used in the PDS and PSS diagrams have a direct one-to-one correspondence with the operations on the data model. [11]

Another data model that does well in graphical form is the molecular model, especially for VLSI applications. The interface and implementation views are ideal for visual representation, as shown in Figure 2.8. However, a model such as QUEL as a data type lacks any meaningful graphical representation, since it is comprised of nested database queries.

11.2.4 To Classification of Design Data

10. *The model must contain machine recognizable classification criteria.*

The IDM has a keyword-based object classification schema built into the structure of each of the objects that comprise the model. This schema is located in the "Description" and "Performance" fields. The values of the keywords are set by the user, then extracted by the system for the purpose of identifying, comparing, and retrieving the objects in the design database. While the same information can be extracted from any model used in a design environment, only the IDM explicitly provides this information to the database system. No intelligent extraction mechanism or traversal of the design data is required in order to determine search parameters, since the IDM declares these parameters in advance.

Since the IDM is the only model that has this feature, it is deemed to be strong on this point. The molecular model, which does not address the classification issue, and actually hides certain information, is rated poor at meeting this requirement.

[11] This one-to-one correspondence was strictly enforced in the prototype system.

11. *The model must differentiate between the component definition schema and the component requirements.*

The IDM utilizes the classification schema in two roles that are distinctly separated by the objects of the IDM that contain them. Unlike the molecular model, which uses the interface object to define components in the library and also to serve as "sockets" in the design, the IDM declares a call object to specifically perform the latter role. Therefore, unlike in a molecular-based design environment, where the designer has necessary restrictions on changes he can make to sockets, the IDM allows calls, and therefore the entire program design, to interactively develop [12] and freely evolve. And unlike the molecular model, since the IDM keeps the software definition and requirements separated, the IDM further guarantees the integrity of the components in the reusable archive. Because of this separation, a semantic conflict cannot arise over the role of an object in the design. In the IDM, interfaces and alternatives are *always* used to define software components, and calls are *always* used to reference them. Therefore, the IDM is considered strong on this point, whereas the molecular model is considered poor.

11.2.5 To Retrieval of Design Data for Reuse

12. *Model must support object retrieval strategies that successfully locate reusable components utilizing only abstract criteria.*

Two separate object retrieval strategies that meet this requirement have been discussed and implemented for use with the IDM. The contribution that the IDM makes in this area is the incorporation of a major reusability issue in the design of a new data model and the application of retrieval algorithms in a prototype system. The prototype implementation has demonstrated that the IDM is strong meeting this requirement.

[12] Hence, the *Interactive Development* Model.

11.2.6 To Archive Storage of Reusable Components

13. *Model must be compatible with an archiving method that supports distributed CASE environments.*

A library organization that supports a distributed CASE environment and is compatible with the IDM model has been shown and implemented. This archive organization places publically accessible components in a central archive, and places components under development in private directories that can be physically located on individual workstations or private directory filespace. While individual library organizations are primarily site-dependent, this model stresses the need for support of distributed CASE environments from the semantic perspective. The library organization for this model, in providing a division between public and private storage, is designed to meet this criteria, and for this reason, the IDM is rated strong for use in distributed systems.

14. *Model must allow sharing of data among users in a distibuted CASE environment.*

The library organization presented allows several users to develop different portions of a software design in parallel. In the IDM, the partially completed designs can be shared by checking them in and out of the public archive. This technique is used in order to guarantee the integrity of partially-completed designs. However, while allowing a means for data sharing, it does not provide support for truly merging parallel efforts. When several designers operate on the same design, the last design checked into the public library will take precedence. Therefore, the IDM is only rated average for its ability to promote data sharing. Nonetheless, database integrity in these situations has been extensively studied, and a locking mechanism can be incorporated into an implementation system.

11.2.7 To Scalability

15. *Each of the data model requirements above must be viewed in the context of being efficient for large scale applications.*

    The contributions of the IDM to requirements of scalability are in the design of the model, and the attention it gives to the reusability issues of data storage, design capture, classification of components, retrieval of components, and organization of a reuse library. In the chapters of this thesis that address each of these issues, the problem of making the issue effective for programming in the large is addressed. This takes the form of a consistent concern for time and space trade-offs in the classification and retrieval algorithms and strategies, to the "economy of scale," levels of abstraction, and object boundary considerations given to the graphical representations and the software archive. The IDM is intended to be a large scale systems model.

11.3 Contributions of the Implementation

The implementation of the data model on the ROSE engineering database system is also innovative and noteworthy. Since ROSE provides many features of a relational database, such as a query language based on a relational algebra, and many features of an object-oriented database, such as clustering related data into objects, the system prototype performs well in two ways. First, the relational orientation provides a fast and efficient search facility for queries and updates. Second, the object-oriented facility provides the semantic qualities necessary for the data model, as well as efficient handling of the design and graphics objects.

Finally, one particularly unique feature of the system prototype presented in this thesis is that the database not only serves to organize the design data, but the database also manages all of the information related to the menus, editing windows, and graphics. This is significant in that it demonstrates the flexibility and power of

supporting a CASE environment with a database system. Additional details on implementation and interface issues can be found in Appendix III.

11.4 Conclusion

The IDM was developed with the above fifteen requirements in mind, as they apply to the definition of a data model for use in a CASE system for software reuse. It has been shown how the IDM addresses each of these points, and therefore qualifies as a candidate data model for these applications.

The major contributions of the model are in specialization of database objects to perform the two roles of a software interface, and in the subsequent flexibility throughout the design process that this enhancement gives the designer. This also has an impact on the traditional view of object-oriented design, pointing out that some implementation details must be included as part of the externally visible attributes of a module traditionally included in the module interface.

A significant effort has also been dedicated to studying various issues surrounding the reusability problem and how the IDM addresses these issues. For the capture of software design data, a new type of Structured Design editor was developed that uniquely corresponds to structured programming and the new data model. Numerous software classification and object retrieval techniques were studied, with several of these implemented in order to study and demonstrate the effectiveness of the IDM with these techniques. Finally, a software library organization that supports the three types of objects in the IDM *and* supports distributed software development environments was developed. Never before considered as an integrated package, the research on these issues highlights the importance and complexity of reusability in computer aided software engineering and proposes a viable solution.

12. FUTURE WORK

12.1 Introduction

Although the data model in this thesis addresses many issues surrounding data management and storage requirements for software engineering, not all of the problems in this field have been identified, much less solved. To date, there remains several areas in which there are open questions and in which more work needs to be done. Some of these areas are discussed below.

12.2 Research Topics

While it has been discussed how the IDM supports the entire software development process, the prototype implementation primarily addresses the middle of the software lifecycle, ie, the component and module levels of design as described in [Phi86, Phi88]. While the model is capable of modeling design data across the entire lifecycle, this particular prototype is not meant to give much support to the early requirements phases and the later testing and maintenance phases of design. This decision was made for several practical considerations and real problems.

The decision was made not to concentrate on analyzing problem statements because this area is very nebulous, difficult to quantify, and still seems to require large amounts of direct human participation. The keyword classification schedules in the call that depict software requirements are minimal, and a study of what kinds of information are necessary to complete a requirements statement in this context is worthwhile. The coding, testing, and maintenance phases are not specifically addressed because they tend to rely on a different set of software tools than the database supplies, for example, compilers and test case generators. However, the model is capable of storing a source-language independent form of pseudocode that can be interpreted by an additional tool or post-processor. The data model should adequately provide the input

and support for these tools.

Furthermore, since the call object is a very dynamic structure, it may be desirable to maintain a log or history of the development of the call. This would be a particularly usefull feature during maintenance of the product. Some form of version control for call objects that is perhaps similar to that used for alternative objects may be required.

A desirable feature of a CASE system is to allow the user to view or design a program using one of several techniques or levels of abstraction. While earlier research [Pou88a] identified the difficulties surrounding automatically generating multiple representations of a design from a common store of data, it is possible to provide the tools for the user to do these things himself. For this reason, the prototype has been developed with the capability to add any number of graphic design and documentation editors as the system evolves. Future work would be to investigate the interaction of these tools within the context of a CASE system based on the IDM.

An additional goal is to expand on the ability to allow the user to specify a level of abstraction from which he wishes to view the design. Currently, levels of abstraction are supported by showing and hiding levels in the calling of declaration structures of the program. There is good reason to allow the user to further define his own levels of abstraction, possibily based on a conceptual rather than physical organization [Rov88]. At a high level, for example, an operating system would be viewed as a set of logical components such as the I/O component and the Memory Manager Component; in the actual design these have no direct physical counterpart. At the low level, these components are specific functions composed of variable declarations and code. While an actual implementation for this kind of abstraction mechanism needs to be researched further, it is believed that logical abstractions can be provided by how the application software interprets the data stored in the model.

One major remaining concern involves accommodating the major role that data structures play in the design of programs and software systems. Most software design methodologies concentrate on the functional requirements of the system and almost completely ignore the design of the underlying data structures. Although it is believed that the software design process is a parallel development of functions and data, the full extent of this relationship and how the data model supports it needs further study.

There are several additional practical considerations surrounding the implementation of the prototype. These arise out of the trade-off that must be made between the desire to make the system as realistic and usable as possible, and the need to be able to actually code the desired features. One such problem is providing interfaces for the definition and querying of the highly recursive code and data structures. While the data model is designed to accomodate these requirements, a compromise in the implementation of these issues has been made. Further time and effort in this area can add to the findings of this research.

13. DISCUSSION AND CONCLUSIONS

In light of the numerous advantages of database support for the engineering design process, much research has gone into applying this technology to CAD, especially with respect to VLSI design. Little, however, has been done to use this knowledge in the design of software. This research has identified several of the important similarities as well as some significant differences in the two domains. These include the question of reuse of program modules, version control requirements, and storage representations. Reusability of software, while a complex melding of the issues of capturing, classifying, storing, and retrieving software design data, may prove to be the most productive approach to software engineering. This thesis demonstrates a data model specifically for use in CASE that addresses these issues, and identifies valid operations on the model.

This three-part data model separates the interface portion of the molecular object model into two distinct portions. These two parts reflect the two different roles that an interface has in the semantic representation of design data. First, the interface defines and represents the object. Second, the call is used to a request service from the object. Finally, the implementation of the object is contained in the third part of the data model. Operations on this model are identified and are adapted to the roles each part of the model plays in representing the software module.

The major contributions made by this model to the fields of software engineering and engineering CAD center on the explicit effort this model makes to support reuse, and the new approach the model takes to representing molecular objects. The separation of the interface into the module call and module definition portions more closely describes the actual roles of these entities in a program design. This separation also allows the model to customize operations on each part, in particular, to assist the designer in locating previously defined interfaces that meet the requirements of a current subprogram call. Finally, by providing a location in the model for recording

design constraints, software requirements have been made an integral part of the design process. No other CAD data model addresses this relationship between software requirements and the final product.

A prototype implementation of a language-independent CASE environment based on these modeling ideas has been completed. Several graphical editors are provided, including a type of Structured Diagram editor, a program structure editor, and a variety of formatted text entry tools. A comprehensive data retrieval mechanism is provided and is based on the software classification schema built into the IDM. The prototype stores design objects in a software library that is organized both to support the three types of data objects in the IDM as well as the special requirements of a distributed design environment. All design tools are incorporated into design editors and activated through a series of pull-down menus. A sample terminal session in this CASE environment is shown in Figure 13.1. Special emphasis has been placed on providing flexible access to the reusability characteristics built into the model.

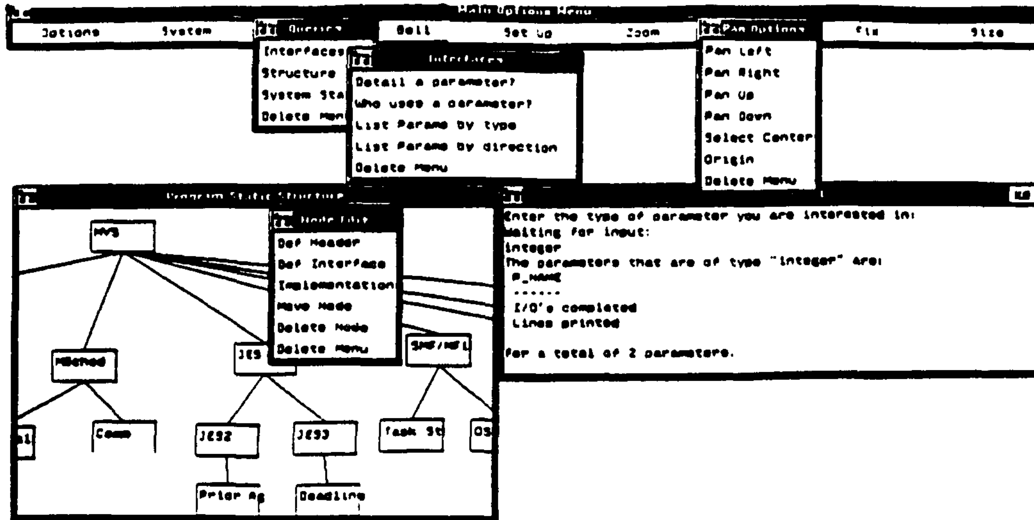


Figure 13.1. Sample Session with the CASE Tool

APPENDIX I: Prototype IDM Structure

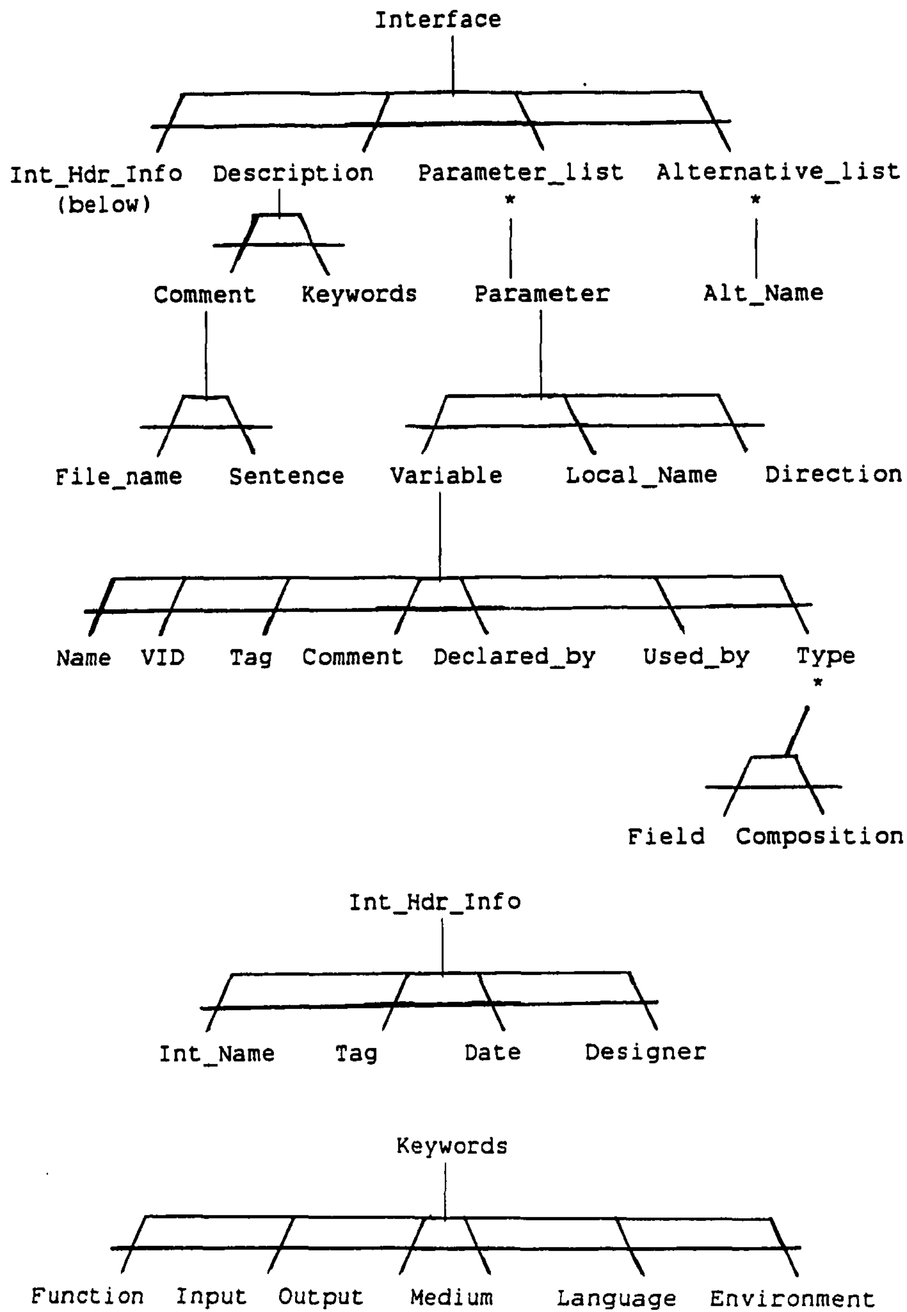


Figure A1.1. Interface Object Structure

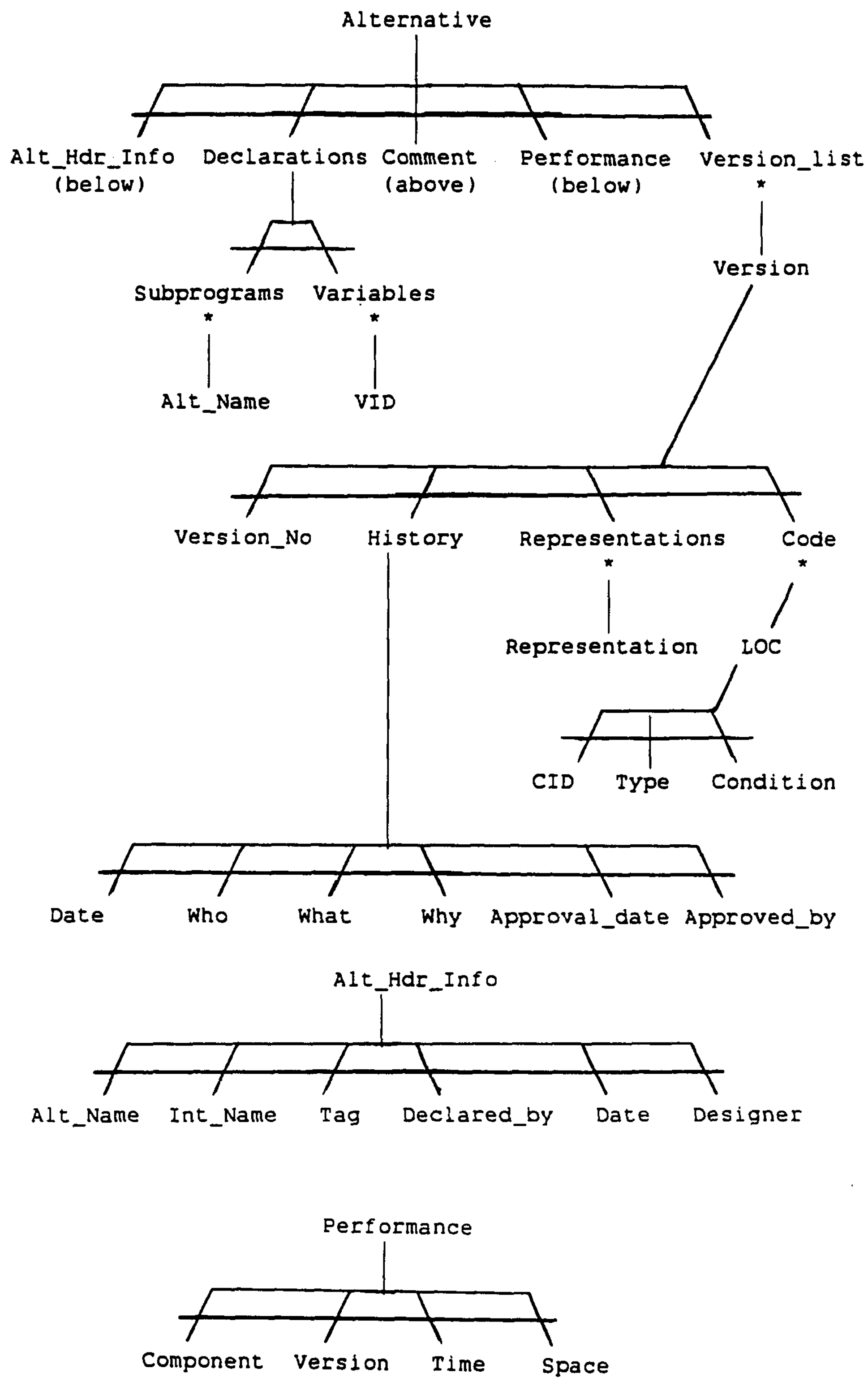


Figure A1.2. Alternative Object Structure

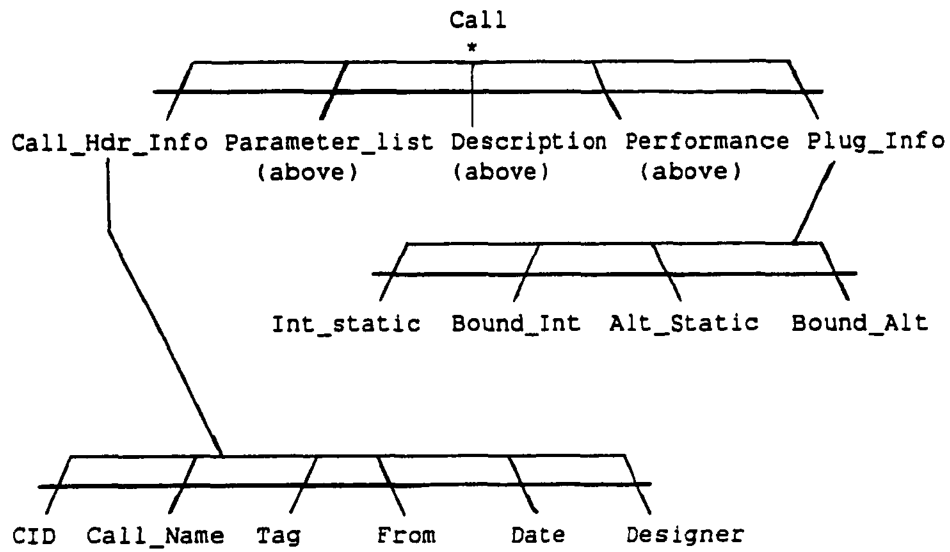


Figure A1.3. Call Object Structure

APPENDIX II: Operations on the IDM

15.1 Introduction

This appendix gives a detailed analysis of the allowable operations on the Interactive Development Model and the constraints that must be enforced during the execution of those operations. The purpose of this appendix is to provide the detail necessary to guide any implementation of the model, as well as to substantiate the IDM from a theoretical perspective.

All of the operations presented here have been implemented as part of the IDM-based CASE prototype discussed in chapter 8.

15.2 Interface Operations

15.2.1 Constraints

In order to allow direct access to interfaces by name, the interface name is used as an index. This places the restriction on all interfaces that the names of each interface be unique. In order to enforce this constraint, a check of the interface archive is completed before the create_interface operation is allowed to complete.

The attributes of an interface are considered non-modifiable. This is because the interface, along with a group of alternative objects, *defines* a software module. If it were permissable to modify the interface, then it would be possible for. the designer to change the interface in ways that are not supported by some or all of the alternatives that implement the interface. If this were so, the interface would no longer accurately represent the module that it serves to define. For this reason, there is no edit_interface operation. The correct course of action when a change to an interface is desired is to copy the interface into a call object, make the desired modifications using the edit_call operation, and then create a new interface from the call. This method has the added advantage of allowing the designer to leave the interface in a partially-defined state for

as long as is deemed necessary by storing the information as a call object.

In order to ensure that no "orphan" alternative objects are created in the software archive, the deletion of interfaces is not allowed when there exist alternative implementations for the interface in the library. However, removal of interfaces having no alternative objects implementing a function is permitted. A note of particular interest is that, in general, it is desirable to indefinitely maintain all interfaces in the software archive for a variety of documentation and legal reasons. While it is not practical to absolutely deny deletion rights to the designer, it may prove worthwhile in a production environment to supplement or replace the deletion operation with a *sleep* operation, which would have the effect of moving the module objects to a long term archive or storage medium such as tape.

The only remaining matter for consideration is the viability of copying alternatives for an interface when the interface is copied, and likewise whether to copy versions along with the alternative when an alternative is copied. The semantics of the IDM allow some flexibility on this point, although it is preferred *not* to make these copies. The reason is that there is no way to enforce whether the newly created interface or alternative will perform the same action or in the same manner as the original; it must in fact be presumed that this is not the case. Therefore, copying such information would guarantee that at least for a time there will be inconsistent information in the database.

However, it is conceivable that such a copy *should* occur. Assume that a complete, tested interface and alternative implementations for that interface exist. Suppose that the represented object is a routine that sorts an integer array. Now assume that the designer wishes to create similiar routines to sort a real array. Normally he would create a copy of the integer routines and simply use the find/replace function in his text editor to make all occurrences of the keyword "integer" into the

keyword "real." Including such an operation is viable, as long as these issues are understood. Note that if this function is provided it is necessary to differentiate the new objects from the old by making copies of the calls in each alternative and assign new surrogate identifiers to each.

15.2.2 Operations

16. *create_interface:* this creates a template for a new interface in the workspace and allows unrestricted modification of the interface attributes. This operation provides the designer with the ability to generate completely new, as yet undefined, interfaces. As in the previous operation, when this process is complete, the interface may be frozen and become the definition for a new object, or it may be saved as a call for later modification. Therefore, the result of this operation is either a new, complete interface object or a new call object.
17. *copy_interface:* this copies an existing interface into the workspace as a call object and allows unrestricted modification of the new call's attributes. This allows the designer to save time and effort when creating a new interface by using an existing interface that may be similar to the one desired. All of the fields in the new call object are given the values in the fields of the copied interface. When all of the attributes have been assigned desired values using the edit_call operation, and the modifications are complete, the interface may be frozen using create_interface and it becomes the definition for a new object. If this action is chosen, the designer will, of course, be required to choose a new name for the interface so as to differentiate it from the interface from which it was created. However, if the interface definition is not considered complete, the information contained in the partially completed interface may be saved as a call object. Therefore, the result of this operation is either a new, complete interface object or a new call object.

18. *retrieve_interface:* fetches a specific interface from the database library using the interface name as the key. The purpose of this operation is to provide access to the interfaces in the software library and in the local workspace. The operation is executed either implicitely by the system as part of a design operation, or explicitely by the designer as part of the reuse process. The result of this operation is find the desired interface in the appropriate location and load it into the main memory work area for general access.

19. *search_for_interfaces:* the search operation assists the designer in locating interfaces in the database library that may meet a given call by using full or partial matches on the interface keywords and parameter list. In practice, any number of search and retrieval strategies may be used; this is the topic of a later chapter. The purpose is to provide the user with a database operation that directly supports the software reuse capability of the CASE system. The result of the search operation is the identification and retrieval into main memory of one module or a group of modules that meet the search criteria. If no modules meet the desired criteria, a null result is obtained. Any modules identified as part of the search operation can be used as the source interface for most of the other interface operations.

20. *bind_interface:* associates the interface with a call in the program design. The purpose is cause the bound interface to be invoked whenever the associated call object is executed. During program design, invoking the interface from a call implies that the designer has located and approved of the use of that interface to meet the need specified in the call object. The binding association may be permanent, or subject to constraints that are dynamically evaluated when the call is evaluated. The effects and advantages of dynamic binding are discussed above. The result of this operation is the storing of a reference pointer in the call to the

bound interface.

21. *display_interface:* shows all the attributes of the interface to the designer. The purpose is to allow the designer to view the attributes for his information. The attributes to be displayed include any administrative information, descriptive keywords, and parameters. The result of the operation is to output the attributes in a specified format in a design window or on some other working surface.
22. *display_alternatives:* shows all the alternative implementations of the interface to the designer. The purpose of the operation is normally to allow the designer to choose among the available alternatives a possible candidate for use in the current application. The alternatives may be browsed, during which time they are subject to the operations on alternatives below. Therefore, the result of this operation is dependent on the operations conducted during the browsing process.
23. *delete_interface:* removes the interface from the library. The purpose of this operation is to explicitely remove from the library any interfaces that are no longer desired. As discussed above, in order to prevent the creation of orphan alternatives, this operation is disabled if there are alternative implementations for the interface remaining in the database. The result of the operation is to remove all evidence of the interface from the public archive and private workspaces.

15.3 Calls

15.3.1 Constraints

In the IDM the call is always considered to be unique. In other words, every request for service is somehow special, and every call object in the design represents exactly one such request. A further explanation of this uniqueness follows.

When copying a call, what does the designer seek to do? In the IDM it is determined that he is developing a similar (or, for that matter, exactly the same) call

for use somewhere else. Still, because it is made from a new location it must be *different*. In database terminology, it is an *instance* of a request. As an example, consider a fictional routine "get_char {c}" as it might be called from the two routines "read_int {I}" and "read_string {S}." There are, in fact, *two* calls made. What is the same in this case is that the same interface, namely the one for "get_char," is bound to both calls. Still, only one copy of the interface and alternatives for "get_char" exist in the database.

This uniqueness of the call object forces the constraint that every call be identified, and therefore indexed, by a surrogate system-generated identifier. When operations on the model might effect the uniqueness of the call, care must be taken to ensure that these identifiers are not duplicated.

A call may or may not an interface and/or an alternative bound to it. However, calls may not have an alternative bound to the call without an interface also being bound to the call. The further restriction implied here is that the interface that is bound to the call is the interface that defines the alternative that is bound to the call. This prevents attempts to mix interfaces with alternatives that are defined by other interfaces, or attempts to leave an alternative without a defining interface. Of course, it is possible to bind an interface to a call without binding an alternative for the interface to the call.

15.3.2 Operations

1. *create_call:* creates a template for a new call in the workspace. The purpose of the operation is to generate module calls whenever and wherever they are required. Upon creation, the call template is assigned an unique surrogate identifier for indexing purposes. This operation has the effect of generating a generic, abstract request for service, which can be referenced from any point in the current program design. The operation results in adding the call object to the database and caching

the call for subsequent operations.

2. *copy_call:* retrieves an old call using the retrieve option and copies the call into the workspace. This has the effect of creating a new call with all the attributes of the original, except for the surrogate identifier used to identify and index the call. Instead, a new surrogate identifier is created and used. The purpose of the operation is similar to that of the similar operation for interfaces; it saves the designer time when creating multiple instances of similar calls. However, in the case of the call object, the copied call may have all of its visible attributes identical to the source call, such is the case when a given routine is called from multiple locations. The names of any interfaces and alternatives that are associated (bound) to the call are also copied, since it is presumed that the same request will be filled with the same routines. As a matter of administration, the designer's name and date are not copied, and should be entered by the person requesting the operation. The result of the operation is the addition of a new call object to the database and the caching of the call for subsequent operations.

3. *retrieve_call:* is used by the system to locate calls in the database library. The purpose of the operation is to provide fast and efficient access to the information about software requirements at a given location in the program design. Because the search key is a hidden surrogate identifier, this operation is indirectly accessible to the designer. After completing the search for the call, the system caches the identifier of the selected call for subsequent operations.

4. *edit_call:* allows unrestricted modification of the call. This operation allows the program design to evolve and change without comprising the integrity of the interface and alternative objects that define the design modules. All of the constraining keywords, comments, and administrative information may be edited. The identifying surrogate key is, however, immutable. The result is the

replacement of the old values of the call attributes with the new values specified by the designer.

5. *make_call:* gives a call object a point of call in the code of an alternative. The create_call operation makes a new call and allows the request for service to be developed; this operation assigns the call to a location in the code of the program design. The operation results in the addition of the surrogate identifier for the call being added to the code for the alternative that is specified as part of the operation. The identifiers for all calls made within an alternative are logically stored in the order in which they will be made during execution of the program.
6. *unmake_call:* removes a call object from a point of call in the code of an alternative. The purpose of this operation is to remove a service request if it is no longer needed, or to allow the designer to change the order of the requests. Of course, the call must be actually made from the specified alternative for the operation to successfully complete. The operation results in the surrogate identifier for the call being removed from the code of the specified alternative object.
7. *unbind_interface:* manually removes an existing association between a call and an interface, if such an association exists. If none exists, an error occurs and no action is performed. Then this operation removes any existing association between the call and an alternative. This prevents the existence of a call with a bound alternative and no bound interface. The purpose is to disassociate modules from a point in the program where they are called, an action which may be done at the designer's discretion any time during the design process. The resulting action in the data model is the removal of the key identifier for the bound object from the "Bound" field of the call and replacing the identifier with a null value.
8. *unbind_alternative:* manually removes an existing association between a call and an alternative, if such an association exists. If none exists, an error occurs and no

action is performed. The purpose is to disassociate alternatives previously thought appropriate for use in a call. Like the previous operation, this disassociation is done at the designers discretion. The resulting action in the data model is the removal of the identifier for the bound object from the "Bound_alternative" field of the call object and replacing the identifier with a null value.

9. *fill_call:* automatically finds all interfaces and alternative implementations that meet the requirements of the call and chooses one. The purpose of this operation is to provide automatic database support for the dynamic binding of design objects to calls as discussed above. If none are available, the system advises the designer. The result of the operation is that, if an appropriate interface and alternative are located within the constraints of the call, the identifiers for those objects are stored in the "Bound" fields of the call object.

10. *display_call:* shows a call from the design database to the designer. This operation allows the designer to view the attributes of requests for service. The operation results in the call being output to the terminal in a predetermined format.

11. *delete_call:* removes a call from the database. This operation allows to eliminate requests for service when they are no longer needed, or when the designer changes his mind. Any alternatives or interfaces bound to the call are not effected. The operation results in all evidence of the call being removed from the designer's workspace.

15.4 Alternatives

15.4.1 Constraints

An alternative cannot come into existence without an interface to represent it. Therefore, before creation of any instance of an alternative object, an interface for the alternative must be specified. The interface that the user specified for the new

alternative is confirmed by the system.

As is the case for interfaces, alternative objects are indexed by name. This provides the user with a direct method to retrieve specific objects by the same name that he assigns them. In the case of alternatives, however, the alternative is dependent on its defining interface for its identity. Therefore, the complete key for an alternative is a two-tuple consisting of *(interface_name, alternative_name)*, and therefore the name of every alternative for a given interface must be unique.

15.4.2 Operations

1. *create_alternative:* creates a template in the workspace for a new alternative. The purpose is to allow the designer to develop a new implementation method for an existing interface. The result of the operation is the addition of a new alternative object in the database, where it is cached for further operations. The name of the defining interface is stored in the alternative object as "Int_Name" and the identifier of the newly created alternative is added to the alternative list in the defining interface.
2. *copy_alternative:* copies an existing alternative implementation into the workspace using the retrieve_alternative operation. This makes a copy of the alternative in order to serve as the basis for another alternative. Like the similar operations for interfaces and calls, this operation exists to save the designer time. By default, the new alternative will be for the same interface as the original. The result is the creation of a template for a new alternative, with the values of the attributes of the copied alternative copied into the new template. A new alternative is then added to the database, where it is cached.
3. *retrieve_alternative:* fetches an existing alternative from the database library using (interface_name, alternative_name) as a key. The purpose is to provide access to the alternative for other operations. If the alternative is not in the local workspace

then it is read from the public archive, as required. The result of this operation is first ensure the specified alternative is available, and then to cache the alternative for access by subsequent operations.

4. *search_for_alternatives:* assists the designer to locate alternative implementations in the library for use in a given call. This operation works with the search_for_interfaces operation in supporting the reuse of software components. The search is based on alternative specific data such as performance attributes, and may use any of the search and retrieval strategies discussed in later chapters. The operation functions on alternatives for a previously specified interface. The result is to cache an alternative of the interface for input into subsequent operations.

5. *bind_alternative:* associates the alternative with a call in the program design. The purpose and results are similar to those for the bind_interface operation. The association between a call and alternative, like that for the interface, may be permanent, or subject to constraints that are dynamically evaluated when the call is evaluated. When the binding is made, it is reflected in the data model by storing the identifier of the bound alternative in the "Bound" field of the appropriate call object.

6. *edit_alternative:* allows modification of the alternative. This provides the designer to change comments and performance information about the alternative when it changes as part of the program design. All performance data and administration information is modifiable, with the exception of the alternative name and the name of the representing interface. The alternative name cannot be changed because it serves as the key for the object and its uniqueness must be guaranteed. The name is also used in any call object that currently binds the alternative. Normally, when alternative modifications are complete, a new version number is assigned and the

modification log updated; this process is explained below as part of the version constraints and operations. This operation replaces the old attribute values with the new values in the database.

7. *display_alternative:* shows all the attributes of the alternative and its interface to the designer. The purpose is to assist the designer to select an alternative for his application as he browses the alternatives available for a given interface. The displayed attributes include any administrative information, performance attributes, and version data. The display operation results in the attributes being shown in a predetermined format in a window on the workstation screen.
8. *display_versions:* shows all the versions of the alternative to the designer. This operation makes it possible to select among the various versions of an alternative, and to trace the development of the code for an alternative. As versions of the alternative are browsed, they are subject to the constraints and operations below.
9. *delete_alternative:* removes an alternative from the database. As with interfaces, it is normal for unused or old alternatives to be kept for documentation and legal reasons. The purpose of this operation, therefore, is to allow the designer to manually remove alternatives when they are no longer needed. All evidence of the alternative is removed from both the local and public databases.

15.5 Versions of Alternatives

15.5.1 Constraints

Although versions are not a separate object in the IDM schema, they are an important enough part of the alternative object to deserve special attention and their own set of constraints and operations. These constraints and operations serve an important function in version control in the IDM.

Each alternative may have any number of versions to implement its function. One of these versions is designated as the current version, and is used for editing in the current design. In addition, each call may constrain which version is to be used for the call. Valid constraints are *Current, Last,* and specific version numbers. Since this is done dynamically as the designed is "compiled," or evaluated, it is necessary to keep the constrained version and the current version separate.

Once a version is created and assigned a number, it cannot be edited. It is modified by making a copy of the version, and then editing the copy. The result may later become a new version of the alternative. Numbered versions can be approved, just as the current version can be approved. The difference is that after approval, any editing done on the current version nullifies the approval. The program is not considered validated, or consistent, unless all of the versions bound for use in the design show approval dates later than the version creation time.

Creation of versions is left to the designer. This allows maximum flexibility for version control. Another option is to create a new version automatically whenever certain changes are made to the alternative. However, this would entail classifying the types of edits into those that justify a new version and those that do not, and would undoubtably lead to a proliferation of versions.

15.5.2 Operations

1. *view_version:* shows the designer the code and the derivation history of a single version. As with the other display operations on the three parts of the IDM, the purpose of this operation is to allow the designer to view what is available to him. The result of the operation is the display of the version on the appropriate viewing surface on the worstation screen.
2. *scan_versions:* allows the designer to browse the available versions of an object. The code and the derivation history of each version are made visible through the

view_version operation.

3. *approve_version:* the version passes site requirements and is validated by an approving authority. The standards for this operation are dependent on the user, and will vary with the implementation. In the prototype, the result of this operation is to set the fields "Approved_by" and "Approval_date" to the values provided by the approving authority.

4. *copy_version:* This creates a new version of an alternative using the current version as a template. The purpose is to make changes to the code of the alternative, but in the process saving the previous status of the code as a distinct step in the development of the alternative. This may be required for documentation, incremental development, prototyping, or legal reasons. Actions on the version portion of the data model are as follows. The newly created version is set equal to the current version. This makes the new version ready for editing. Call identifiers within the versions are not modified to make the calls within each version unique; this is unnecessary since only one version of the alternative is active at any time. This preserves and ensures the uniqueness of the call, as discussed above.

5. *set_current_version:* designates the version currently being viewed as the current version for all future references to the alternative. The current version is the version used for editing, and is the version bound to any requesting call that constrains the version to "current." Therefore, this operation controls how the design is evaluated whenever calls constrain versions in this manner. The result of the operation is to store a currency pointer in the alternative to the version newly designated as "current."

APPENDIX III: User Interface Issues

1. Introduction

User Interface considerations, although really not a part of this research, are important for many reasons. The most important of these is that the usability of any software tool is most directly effected by what the user sees, and particularly how he interfaces with the tool. Another reason that the interface is important is that it forms more of an initial impression about the system than what is actually going on behind the scenes.

Nonetheless, it is necessary to develop *some kind of interface* in order to demonstrate how the IDM functions in a prototype CASE environment. With this in mind, it seems worthwhile to put some effort into the interface so that useful observations made and added to our techniques for other applications. Finally, it is important to demonstrate how the IDM might appear if packaged in a somewhat reasonable fashion. However, it must be noted that the user interface for this prototype is not a research issue, and for this reason this discussion has been relegated to an appendix.

2. Text Entry Boxes

The text entry box is an effective technique for string input that provides a fixed format, prompts, and a variety of text fonts. One big advantage is that the user sees all of the entries he should make at once, and has the option of making them in any order. This is highly preferred over many sequential prompts in the dialogue box. The order of entry also has important consequences for the searching functions. The particular text boxes seen in the chapter describing the prototype are, however, uniquely ugly and somewhat large. The size is limited by the minimum font size under the UIS graphics system, and while the aesthetics are not so good they do not effect overall performance and functionality. All in all, the

text box presents all the tools in one place in a standard manner. This advantage outweighs all disadvantages.

3. **Editing Palette vs. Pull-Down Menus**

The editing palette used for the PDS is something learned from and first developed as part of the CB SketchPad [Pou87]. The advantages are similar to those experienced by Apple computer users; the human mind assimilates and relates to icons faster than it can to text. However, such a palette is much more difficult to create, change, and expand. In this regard the pull-down menu is preferred over the palette, especially early in system development or during prototyping. The palette also takes up a lot of screen space, which in this application is at a premium.

4. **IPO Window**

The concept of the IPO Chart is to display everything about a module in one place in a standard format. Unfortunately, there is a lot of information that we wish to display in the IPO, and not enough space to do it. Part of the problem is due to the minimum font size in UIS. An additional problem is the nature of some of the design information. Some information is of fixed length (designer's name) while other information can be of any length (parameter lists). How much space does one leave on the chart to display such information? Many experiments with these two problems was conducted, including writing information microscopically small and using the zoom function to read it. Sending all information to the dialogue window was also considered; that solution, although much easier than filling the IPO chart, failed because the dialogue window was also too small for all the required information. The best solution was to put fixed length data on the chart, and list the variable length data on request to the dialogue window. This not only cleaned up the IPO chart but also made the display

operation much faster. This is method currently in use.

APPENDIX IV: The Correspondence Between DFDs and DSDs

Earlier research concentrated on the automatic conversion of one type of design diagram to any other type. During this study, it was discovered that for the most part the methodologies *could* be mixed, within certain guidelines [Pou88a].

The advantage of such an ability is that, assuming the design data was stored in some standard form, any representation, or view, could be automatically generated depending on the wishes of the user. Such a capability would be invaluable for documentation and report purposes, or in cases where several designers or managers wished to work on or view the program design using the diagrams of their own "favorite" methodolgy. Since the views are created from a common store, the views are always guaranteed to be consistent.

The major problem with such an ability comes in the creation of a DSD from DFD-generated design data, and the converse. Both of the methodologies lead the designer from a problem statement to a software structure. Ideally, both methods would result in the exact same software (because there is only one "best" solution), although in reality this is quite rare (because the methods are very different). The problem becomes a matter of how much information one is allowed to infer from the database. [13]

Allow the discussion to take a slightly mathematical turn for a moment. Consider a problem that has been analyzed using both the data flow and data structure techniques. Intuitively, if a program structure (PS) for this problem is derived from the DFD, ie, f(DFD) = PS_from_DFD, then f-inverse(PS_from_DFD) = DFD. The same follows if g(DSD) = PS_from_DSD, then g-inverse(PS_from_DSD) = DSD. The next question is, does g-inverse(PS_from_DFD) allow you to infer anything about the data structures of the program? Does f-inverse(PS_from_DSD) give you a good data flow perspective of the problem? Is such an inference valid? Could one, in general, construct

[13] An excellent discussion on how the various methodologies can be used on the same problem and yield various solutions can be found in [Yau87].

a DFD directly from a DSD or the reverse, a DSD from a DFD? This is a matter for further research, although I conjecture the following two points:

1. This is not a possible transformation since an analysis of the problem from one perspective is completely independent of the other.
2. The intuitive argument regarding the validity of the inverse function above is not entirely correct because the function itself is not guaranteed to be unique.

LITERATURE CITED

[Alb84] Alberga, C.N., A.L. Brown, G.B. Leeman, Jr., M. Mikelsons, and M.N. Wegman. "A Program Development Tool," *IBM Journal Research and Development,* Vol. 28, No. 1, January 1984, pp. 60-73.

[And88] Anderson, Kathryn J., Roger P. Peck, and Thomas E. Buonanno, "Reuse of Software Modules," in *AT&T Technical Journal,* Volume 67, Number 4, July/August 1988, pp. 71-76.

[Arn87] Arnold, Susan P. and Stephen L. Stepoway, "The REUSE System: Cataloging and Retrieval of Reusable Software," in *Proceedings of COMPCON S'87,* 1987, pp. 376-379.

[Bal85] Balzer, Robert. "A 15 Year Perspective on Automatic Programming," in *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 11, November 1985, pp. 1257-1268.

[Bap86] Bapa Rao, K.V. "An Object-Oriented Framework for Modeling Design Data," position paper in *Proceedings of the International Workshop on Object-Oriented Database Systems,* Pacific Grove, California, September, 1986, p. 232.

[Bat84] Batory, D.S. and Alejandro Buchmann, "Molecular Objects, Abstract Data Types, and Data Models: A Framework," in *Proceedings of the 10th International Conference on Very Large DataBases,* Signapore, August, 1984, pp. 172-184.

[Bat85] Batory, D.S. and Won Kim, "Modeling Concepts for VLSI CAD Objects," in *ACM Transactions on Database Systems,* Vol. 10, No. 3, September 1985, pp. 322-346.

[Bee88] Beech, David and Brom Mahbod, "Generalized Version Control in an Object-Oriented Database, in *Proceedings of the 4th International Conference on Data Engineering,* IEEE Computer Society, 1988.

[Ber85] Bergland, G.D. "A Guided Tour of Program Design Methodologies," in *Chow, Tsun S.* IEEE Tutorial on Software Quality Assurance, IEEE Computer Society Press, Silver Spring, Maryland, 1985, pp. 219-243.

[Ber87] Bernstein, Philip A., "Database System Support for Software Engineering: An Extended Abstract," in *Proceedings of the 9th International Conference on Software Engineering,* Monterey, California, April, 1987, pp. 166-178.

[Bha87] Bhateja, Rajiv and Randy H. Katz, "Valkyrie: A Validation Subsystem of a Version Server for Computer-Aided Design Data," in *Proceedings of the 24th Design Automation Conference,* Los Vegas, Nevada, 1987, pp. 321-327.

[Blu87] Blum, Bruce I. "A Paradigm for Developing Information Systems." in *IEEE Transactions on Software Engineering*, Vol. SE-13, No. April 1987, pp. 432-439.

[Boo84] Booch, Grady. Software Engineering with Ada, Second edition. The Benjamin/Cummings Publishing Company, Inc, Menlo Park, California, 1984.

[Boo86] Booch, Grady, "Object-Oriented Development," in *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 2, February 1986, pp. 211-221.

[Bro84] Brodie, M., B. Blaustein, U. Dayal, F. Manola, and A. Rosenthal, "CAD/CAM Database Management," in *Database Engineering,* Volume 7, Number 2, 1984.

[Bro87] Brooks, Frederick P., Jr. "No Silver Bullet: Essence and Accidents of Software Engineering," in *IEEE Computer,* Vol. 20, Number 4, April 1987, pp. 10-19.

[Bro85] Brown, Gretchen P., Richard T. Carling, Christopher F. Herot, David A. Kramlich, and Paul Souza. "Program Visualization: Graphic Support for Software Development," in *IEEE Computer,* Vol. 18, Number 8, August 1985, pp. 27-35.

[Buc85] Buchmann, Alejandro P. and Concepcion Perez de Celis, "An Architecture and Data Model for CAD Databases, in *Proceedings of the 11th International Conference on Very Large Data Bases*, Stockholm, 1985, pp. 105-114.

[Buh84] Buhr, R.J.A. System Design with Ada. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Buh89] Buhr, R.J.A., et al. "Software CAD: A Revolutionary Approach," in *IEEE Transactions on Software Engineering,* Vol. 15, No. 3, March 1989, pp. 235-249.

[Bur87] Burton, Bruce A., Rhonda Wienk Aragon, Stephen A. Bailey, Kenneth D. Koehler, and Lauren A. Mayes, "The Reusable Software Library," in *IEEE Software,* July, 1987, pp. 25-33.

[Cad87] Cadre. "Teamwork: A Management Overview," *Cadre Technologies, Inc.,* 1987.

[Cam86] Cammarata, Stephanie Jo, "An Object-Oriented Data Model for Managing Computer-Aided Design and Computer-Aided Manufacturing Databases," *Doctoral Dissertation,* University of California, Los Angeles, 1986.

[Cam88] Cammarata, Stephanie, et al., "Panel: Is 'Object-Oriented" The Final Solution to DBMS Problems?" at *Fourth International Conference on Data Engineering,* Los Angeles, California, February, 1988.

[Cam83] Campos, Ivan M. and Gerald Estrin, "Concurrent Software System Design Supported by SARA at the Age of One," in *Tutorial on Software Design Techniques,* 4th Edition, ed. Peter Freeman and Anthony I. Wasserman, IEEE Computer Society Press, Silver Springs, Maryland, 1983, pp. 353-365.

[Cha85] Charniak, Eugene and Drew McDermott. Introduction to Artificial Intelligence. Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

[Che84] Cheatham, Thomas E. Jr., "Reusability Through Program Transformations," in *IEEE Transactions on Software Engineering,* Vol. SE-10, No. 5, September 1984, pp. 589-594.

[Chu87] Chung, Moon-Jung, Ephraim P. Glinert, Martin Hardwick, Edwin H. Rogers and Kenneth Rose. "Toward an Object-Oriented Iconic Environment for Computer Assisted VLSI Design," *Department of Computer Science Technical Report Number 87-3,* Rensselaer Polytechnic Institute, Troy, New York, February, 1987.

[Con87] Conn, Richard, "The Ada Software Repository and Software Reusability," in *Proceedings of the Fifth Annual Joint Conference on Ada Technology,* Washington Ada Symposium, 1987, pp. 45-53.

[Cor87] Corliss, George F., "Design of an Ada Library of Elementary Functions with Error Handling," in *J. Pascal, Ada, Modula-2,* Vol. 6, No.3, May-June 1987, pp.17-31.

[Dat85a] Date, C.J., An Introduction to Database Systems, Volume 1. Addison-Wesley, Reading, Massachusetts, 1985.

[Dat85b] Date, C.J., An Introduction to Database Systems, Volume 2. Addison-Wesley, Reading, Massachusetts, 1985.

[Dav83] Davis, William S. Tools and Techniques for Structured System Analysis and Design. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[Day83] Day, F.W. "Computer Aided Software Engineering," in *Proceedings of the ACM IEEE 20th Design Automation Conference,* Miami Beach, Florida, 1983, pp. 129-136.

[DeB85] De Bruin, R. and Van Der Laan, C.G., "The Creation of a Virtual NAG-ALGOL 68 Program Library," in *Software Practical Experience, Vol. 15, No. 10, October 1985, pp.963-972.*

[DEC84] DEC. "User's Introduction to VAX DEC/CMS," *Digital Equipment Corporation,* Maynard, Massachusetts, 1984.

[Der85] Dershowitz, Nachum, "Program Abstraction and Instantiation," in *ACM Transactions on Programming Language Systems,* Vol. 7, No. 3, July 1985, pp.446-477.

[Dei84] Deitel, Harvey M. An Introduction to Operating Systems. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[DeM78] DeMarco, Tom. Structured Analysis and System Specification. Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Dep83] Depree, Robert W., "Pattern Recognition in Software Engineering, in *IEEE Computer,* Vol. 16, No. 5, May 1983, pp. 48-53.

[Dic81] Dickover, M.E., C.L. McGowan and D.T. Ross, "Software Design Using SADT," in *"Tutorial on Software Design Strategies,"* Second Edition, Glenn D. Bergland and Ronald D. Gordon, eds., IEEE Computer Society Press, Los Angeles, California, 1981, pp. 397-409.

[Dig88] Digital Consulting, Inc. *Brochure for CASES, a Computer-Aided Software Engineering Symposium,* Boston, Massachusetts, April 25-27, 1988.

[Dij79] Dijkstra, E, "Programming Considered as a Human Activity," in Classics in Software Engineering, ed. Nash Yourdon, Yourdon Press, New York, 1979, pp. 3-12.

[Dit88] Dittrich, Klaus R. and Raymond A. Lorie, "Version Support for Engineering Database Systems," in *IEEE Transactions on Software Engineering,* Vol. 14, No. 4, April 1988.

[Duf89] Duffrin, David. *Private conversation,* International Business Machines Corporation, Myers Corners, New York, 10 February 1989.

[Fer88] Ferucci, David, "OOCADE VLSI Representation Issues Figures (Draft)," 16 March 1988.

[Fis87] Fischer, Gerhard, "Cognitive View of Reuse and Redesign," in *IEEE Software,* July, 1987, pp. 60-72.

[Fra87] Frakes, W.B. and B.A. Nejmeh, "An Information System for Software Reuse," in *Proceedings of the 10th Minnowbrook Workshop on Software Reuse,* 1987, pp. 142-151.

[Fri82] Friedell, Mark, Jane Barnett and David Kramlich. "Context Sensitive, Graphic Presentation of Information." *Computer Graphics,* Vol. 16, No. 3, July 1982, pp. 181-188.

[Gar87] Gargaro, Anthony, and T.L. Pappas, "Reusability Issues and Ada," in *IEEE Software,* July, 1987, pp. 43-51.

[GE87] GE. "Interactive Ada Workstation," *Research Brochure,* General Electric Company Corporate Research and Development, Niskayuna, New York, 1987.

[Gli86a] Glinert, Ephraim P. "Interactive, Graphical Programming Environments: Six Open Problems and a Possible Solution," *Department of Computer Science Technical Report Number 86-13*, Rensselaer Polytechnic Institute, Troy, New York, July, 1986.

[Gli86b] Glinert, Ephraim P. "Towards 'Second Generation' Interactive, Graphical Programming Environments," *Department of Computer Science Technical Report Number 86-12*, Rensselaer Polytechnic Institute, Troy, New York, July, 1986.

[Gli86c] Glinert, Ephraim P. and Craig D. Smith. "PC-TILES: A Visual Programming Environment for Personal Computers Based on the BLOX Methodology," *Department of Computer Science Technical Report Number 86-21*, Rensselaer Polytechnic Institute, Troy, New York, October, 1986.

[Gli87] Glinert, Ephraim P., Martin Hardwick, M.S. Krishnamoorthy, and David Spooner. "Visual Programming Environments and CAD Databases: A Synthesis of Emerging Concepts for Software Engineering," *Department of Computer Science*, Rensselaer Polytechnic Institute. Troy, New York, 1987, pp. 1-10.

[GlL86] Glintz, Martin and Jochen Ludewig, "SEED- A DBMS for Software Engineering Applications Based on the Entity-Relationship Aprroach," in *Proceedings of the International Conference on Data Engineering,* Los Angeles, California, February 1986, pp. 654-660.

[Gog84] Goguen, Joseph A., "Parameterized Programming," in *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 528-543.

[Gut82] Guttman, Antonin and Michael Stonebraker, "Using a Relational Database Management System for Computer Aided Design Data," in *Database Engineering*, Vol. 5, No. 2, June 1982, pp. 21-28.

[Har85a] Hardwick, Martin. "Design and Implementation of a Data Manager for Design Objects," *Department of Computer Science Technical Report Number 85-34*, Rensselaer Polytechnic Institute, Troy, New York, 1985.

[Har85b] Hardwick, Martin and David L Spooner. "Comparison of Data Models for CAD Objects," *Department of Computer Science Technical Report Number 85-36*, Rensselaer Polytechnic Institute, Troy, New York, 1985.

[Har86] Hardwick, Martin. "User Manual for ROSE: A CAD/CAM Database System," *Department of Computer Science Technical Report Number 86-24,* Rensselaer Polytechnic Institute, Troy, New York, October, 1986.

[Har87a] Hardwick, Martin. "Why ROSE is Fast: Five Optimizations in the Design of an Experimental Database System for CAD/CAM Applications," in *Proceedings of ACM SIGMOD,* San Francisco, California, May, 1987, pp. 292-298.

[Har87b] Hardwick, Martin and David Spooner. "The ROSE Object-Oriented Database System: The Advantages of an Open Architecture," *Department of Computer Science Technical Report Number 87-30,* Rensselaer Polytechnic Institute, Troy, New York, December, 1987.

[Har87c] Hardwick, Martin, George Samaras and David Spooner. "Evaluating Recursive Queries in CAD Using an Extended Projection Function," in *Proceedings of the 3rd International Conference on Data Engineering,* Los Angeles, California, February, 1987, pp. 138-148.

[Has82] Haskin, Roger and Raymond Lorie, "Using a Relational Database System for Circuit Design," in *Database Engineering,* Vol. 5, No. 2, June 1982, pp. 10-14.

[Haw84] Hawryszkiewycz, I.T. Database Analysis and Design. Science Research Associates, Inc., Chicago, 1984.

[Hay81] Haynie, M., "The Relational/Network Hybrid Data Model for Design Automation Databases," in *Proceedings of the 18th Design Automation Conference*, 1981, pp. 646-652.

[Hel87] Helier, Sandra, Umeshwar Dayal, Jack Orenstein, and Susan Radke-Sproull, "An Object-Oriented Approach to Data Management: Why Design Databases Need it," in *Proceedings of the 24th Design Automation Conference*, Los Vegas, Nevada, 1987, pp. 335-340.

[Hig83] Higgins, David. Designing Structured Programs. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1983.

[Hud87] Hudson, Scott E. and Roger King, "Object-Oriented Database Support for Software Environments," in *Proceedings of ACM SIGMOD,* San Francisco, California, May, 1987, pp. 491-503.

[Hud88] Hudson, Scott E. and Roger King, "The Cactis Project: Database Support for Software Environments," in *IEEE Transactions on Software Engineering,* Volume 14, Number 6, June, 1988, pp. 709-719.

[IDE87] IDE. "Software Through Pictures," *Sales Brochure,* Interactive Development Environments, San Francisco, CA, 1987.

[Iso87] Isoda, Sadahiro, "SoftDA: A Computer Aided Software Engineering System," in *Proceedings of the 1987 Fall Joint Computer Conference,* October 25-29, 1987, Dallas, Texas, pp.147-151.

[Jac75] Jackson, Michael. Principles of Program Design. Academic Press, 1975.

[Kat85] Katz, R. H. Information Management for Engineering Design. Springer-Verlag, Berlin, FRG, 1985.

[Kat86] Katz, R. H., M. Anwarrudin, and E. Chang, "A Version Server for Computer-Aided Design Data," in *Proceedings of the 23rd Design Automation Conference,* Los Vegas, Nevada, 1986, pp. 27-33.

[Kat87] Katz, Randy H., Rajiv Bhteja, Ellis E-Li Chang, David Gedye, and Vony Trijanto. "Design Version Management," in *IEEE Design and Test,* 1987, pp. 12-22.

[Kai87] Kaiser, Gail E., and David Garlan, "Melding Software from Reusable Building Blocks," in *IEEE Software,* July, 1987, pp. 17-32.

[Kim87] Kim, Won, Hong-Tai Chou and Jay Banerjee, "Operations and Implementation of Complex Objects," in *Proceedings of the 3rd International Conference on Data Engineering,* Los Angeles, California, 1987, pp. 626-633.

[Koz87] Kozol, Micheal. "An Automated Software Engineering Project Notebook for the IBM-PC," *Master's Project,* Rensselaer Polytechnic Institute, Troy, New York, April, 1987.

[Kra83] Kramlich, David, and Gretchen P. Brown, Richard T. Carling, Christopher F. Herot. "Program Visualization: Graphics Support for Software Development," in *Proceedings ACM IEEE 20th Design Automation Conference,* Miami Beach, Florida, 1983, pp. 143-149.

[Kur75] Kurzban, Stanley A, Thomas S. Heines and Anthony P. Sayers. Operating Systems Principles. Petrocelli/Charter, New York, 1975.

[Len87] Lenz, Manfred, Hans Albrecht Schmid, and Peter F. Wolf, "Software Reuse Through Building Blocks," in *IEEE Software*, July, 1987, pp. 34-42.

[Lit84] Litvintchouk, Steven D. and Allen S. Matsumoto, "Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specification," in *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 544-551.

[Lon72] London, Keith R. Decision Tables. Auerbach, Princeton Jersey, 1972.

[Lor83] Lorie, Raymond and Wilfred Plouffe, "Complex Objects and Their Use in Design Transactions," in *Proceedings of Annual Meeting of Engineering Design Applications*, San Jose, California, May 1983, pp. 115-121.

[Mar85a] Martin, James and Carma McClure. Action Diagrams: Clearly Structured Program Design. Prentice Hall, Inc, Englewood Cliffs, New Jersey, 1985.

[Mar85b] Martin, James and Carma McClure. Diagramming Techniques for Analysts and Programmers. Prentice Hall, Inc, Englewood Cliffs, New Jersey, 1985.

[Mar85c] Martin, James and Carma McClure. Structured Techniques for Computing. Prentice Hall, Inc, Englewood Cliffs, New Jersey, 1985.

[Mar87] Martin, James. Recommended Diagramming Standards for Analyst and Programmers. Prentice Hall, Inc, Englewood Cliffs, New Jersey, 1987.

[Mat84], Matsumoto, Yoshihiro, "Some Experiences in Promoting Reusable Software Presentation in Higher Abstraction Levels," in *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984, pp. 502-513.

[Mat87] Matsumura, Kazuo, Hiroyuki Mizutani and Masahiko Arai. " An Application of Structural Modeling to Software Requirements Analysis and Design." *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 4, April 1987, pp. 461-471.

[MHS86] Matthews, T.J, R.M. Hollett and C.M. Smith. "Advanced Graphical Workstations for Software Development," in *British Telecommunications Technology Journal,* Vol. 4, Number 3, July 1986, pp. 92-101.

[McL83] McLeod, Dennis, K. Narayanaswamy, and K. V. Bapa Rao, "An Approach to Information Management for CAD/VLSI Applications," in *Proceedings of the Annual Meeting to Engineering Design Applications,* San Jose, California, May, 1983, pp. 39-50.

[Mit87a] Mittermeir, Roland T. and Wilhelm Rossack, "Software Bases and Software Archives: Alternatives to Support Software Reuse," in Proceedings of the 1987 Fall Joint Computer Conference, October 25-29, 1987, Dallas, Texas, pp. 21-28.

[Mit87b] Mittermeir, Roland T. and Marcus Oppitz, "Software Bases for the Flexible Composition of Application Systems," in IEEE Transactions on Software Engineering, Vol. SE-13, No. 4, April 1987, pp. 440-460.

[Mur75] Murrill, Paul W. and Cecil L. Smith. An Introduction to Fortran IV Programming: A General Approach, Second edition. Intext Educational Publishers, New York, 1975.

[Mye83] Myers, Brad A. "INCENSE: A System for Displaying Data Structures." *Computer Graphics,* Vol. 17, No. 3, July 1983, pp. 115-125.

[Mye78] Myers, Glenford J. Composite/Stuctured Design. Van Nostrand Reinhold Company, New York, 1978.

[Nes86] Nestor, John R. "Re-creation and Evolution in a Programming Environment," in *Proceedings of the International Workshop on Object-Oriented Systems,* Pacific Grove, California, September 1986, p. 230.

[Nei84] Neighbors, James M., "The Draco Approach to Constructing Software from Reusable Components," in *IEEE Transactions on Software Engineering,* Vol. SE-10, No. 5, September 1984, pp. 564-574.

[Olu83] Olumi, Mohammad, et al, "Software Project Databases," in *Proceedings of Annual Meeting of Engineering Design Applications,* San Jose, California, May 1983, pp. 124-134.

[Onu87a] Onuegbe, Emmanuael O., "Database Management System Requirements for Software Engineering Environments," in *Proceedings of the 3rd International Conference on Data Engineering,* Los Angeles, California, 1987, pp. 501-509.

[Onu87b] Onuegbe, Emmanuael O., "Software Classification as an Aid to Reuse: Initial Use as Part of a Rapid Prototyping System," in *Proceedings of the 20th Annual Hawaii International Conference on System Sciences,* 1987, pp. 521-529.

[Pet81] Peters, Lawrence J. Software Design: Methods and Techniques. Yourdon Press, New York, 1981.

[Phi86] Phillips, Richard and Ron Radice. "Software Engineering I and II," *Class Notes,* Rensselaer Polytechnic Institute, Troy, New York, September 1985- May 1986.

[Phi88] Phillips, Richard and Ron Radice. Software Engineering: An Industrial Approach., Vol. 1. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1988.

[Pid85] Pidgeon, Christopher W. and Peter A. Freeman. "Development Concerns for a Software Design Quality Expert System," in *Proceedings, 22nd ACM/IEEE Design Automation Conference,* Las Vegas, Nevada, 1985, pp.562-568.

[Plo84] Plouffe, Wil, Won Kim, Raymond Lorie, and Dan McNabb, "A Database System for Engineering Design," in *Database Engineering,* Vol. 7, No. 2, June 1984, pp. 48-55.

[Pot88] Potter, Walter D. and Robert P. Trueblood, "Traditional, Semantic, and Hyper-Semantic Approaches to Data Modeling," in *Computer,* Volume 21, Number 6, June, 1988, pp. 53-63.

[Pou87] Poulin, Jeffrey S., "Summary of Research," *Research Summary,* Department of Computer Science, Rensselaer Polytechnic Institute, August, 1987.

[Pou88a] Poulin, Jeffrey S., "Object-Oriented Database Support for Computer Aided Software Engineering," *Research Summary,* Department of Computer Science, Rensselaer Polytechnic Institute, February, 1988.

[Pou88b] Poulin, Jeffrey S., "Object-Oriented Database Support for Computer Aided Software Engineering," *Research Proposal,* Department of Computer Science, Rensselaer Polytechnic Institute, 3 August, 1988.

[Pou89] Poulin, Jeffrey S., and Martin Hardwick, "Adapting Object-Oriented Database Concepts for Computer Aided Software Engineering (CASE)," to appear in *Proceedings of the International Symposium on Database Systems for Advanced Applications,* Seoul, Korea, April, 1989.

[Pow83] Powell, Michael L. and Mark A. Linton, "Database Support for Programming Environments," in *Proceedings of Annual Meeting of Engineering Design Applications,* San Jose, California, May 1983, pp. 63-70.

[Pre82] Pressman, Roger S. Software Engineering: A Practitioner's Approach. McGraw-Hill Book Company, New York, 1982.

[Pri87] Prieto-Diaz, Ruben, and Peter Freeman, "Classifying Software for Reusability," in *IEEE Software,* Los Alomitos, California, January 1987, pp. 6-16.

[Pri88] Prieto-Diaz, Ruben, and Gerald A. Jones, "Breathing New Life into Old Software," in *GTE Journal of Sciences and Technology,* Vol. 1, 1988, pp. 152-160.

[Pyl81] Pyle, I.C. The Ada Programming Language. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[Rae85] Raeder, Georg. "A Survey of Current Graphical Programming Techniques," in *IEEE Computer,* Vol. 18, Number 8, August 1985, pp. 11-24.

[Raj87] Rajlich, Vaclav. "Refinement Methodology for Ada," in *IEEE Transactions on Software Engineering,* Vol. SE-13, No. 4, April 1987, pp. 472-478.

[Ram86] Ramamoorthy C.V., Vijay Garg and Atul Prakash. "Programming in the Large," in *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 7, July 1986, pp. 769-783.

[ReA87] Reilly, Angela. "Roots of Reuse," in *IEEE Software,* Los Alomitos, California, January 1987, pp. 4-5.

[ReS85] Reiss, Steven P. "PECAN: Program Development Systems that Support Multiple Views." in *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 3, March 1985, pp. 276-285.

[ReS86] Reiss, Steven P. "An Object-Oriented Framework for Graphical Programming (Summary Paper)." *SIGPLAN Notices,* Vol. 21, No. 10, October 1986, pp. 49-57.

[ReS87] Reiss, Steven P. "Working in the Garden Environment for Conceptual Programming," in *IEEE Software,* November 1987, pp. 16-27.

[Rom87] Roman, Gruia-Catalin, "Data Engineering in Software Development Environments," in *Proceedings of the 3rd International Conference on Data Engineering,* Los Angeles, California, 1987, pp. 85-86.

[Rou83] Roussopoulos, Nick and Stephen Kelly, "A Relational Database to Support Graphical Design and Documentation," in *Proceedings of Annual Meeting of Engineering Design Applications,* San Jose, California, May 1983, pp. 135-149.

[Rov88] Rovira, Margarita, *Private Conversations,* July-October 1988.

[Rov89] Rovira, Margarita, "Adapting Object-Oriented CAD Database Concepts for Computer Aided Software Engineering," *Fall 1988 Research Summary,* March, 1989.

[Rug86] Rugg, Tom, and Phil Feldman, Turbo Pascal Program Library. Que Corporation, Indianapolis, Indiana, 1986.

[Sal75] Salton, Gerard. Dynamic Information and Library Processing. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1975.

[Sam87] Samaras, George and Martin Hardwick. "User Manual for a VLSI CAD System Developed on ROSE," *Department of Computer Science,* Rensselaer Polytechnic Institute, Troy, New York, 1987, pp. 1-24.

[Sha87] Shatz, Sol M. and Jia-Ping Wang, "Introduction to DistributedSoftware Engineering," in *Computer,* Vol. 20, No. 10, October, 1987, pp. 23-31.

[Shi81] Shipman, D., "The Functional Data Model and the Data Language Daplex," in *ACM Transactions on Database Systems*, Volume 6, Number 1, March, 1981, pp. 140-173.

[Smi77] Smith, J. and D. Smith, "Data Abstractions: Aggregation and Generalization," *ACM Transactions on Database Systems,* Volume 3, Number 3, 1977, pp. 105-133.

[Spo86] Spooner, David. "Advanced Database Management Topics," *Class Notes*, Rensselaer Polytechnic Institute, Troy, New York, September 1986-December 1986.

[Spo87] Spooner, David, and Martin Hardwick. "Using CAD Database Technology for Software Engineering: Research Plan," *Research Plan for IBM project,* Rensselaer Polytechnic Institute, Troy, New York, June, 1987.

[Sid80] Sidle, Thomas W., "Weaknesses of Commercial Data Base Management Systems in Engineering Applications," in *Proceedings of the 17th Design Automation Conference*, New York, 1980, pp. 57-61.

[Sim86] Simon, Herbert A. "Whether Software Engineering Needs to be Artificially Intelligent," in *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 7, July, 1986, pp. 726-732.

[Ste79] Stevens, W, G. Meyers and L. Constantine, "Structured Design," in Classics in Software Engineering, ed. Nash Yourdon, Yourdon Press, New York, 1979, pp. 207-231.

[Sto84] Stonebraker, M., et al. "QUEL as a Data Type," in *Proceedings of the International SIGMOD Conference,* Boston, June, 1984, pp. 208-214.

[Sto87] Stovsky, Michael P. and Bruce W. Weide. "STILE: A Graphical Design and Development Environment," in *Proceedings of the 22nd ACM/IEEE Design Automation Conference,* Las Vegas, Nevada, 1987, pp. 247-250.

[SUN86] SCCS, "Source Code Control System," in Programming Utilities for the SUN Workstation," *SUN Microsystems, Inc.*, Mountain View, California, February, 1986, pp. 71-90.

[Swa87] Swaminathau, Ramesh , James Loy, M.S. Krishnamoorthy, and Patrick Harubin. "On Animation Programs," *Department of Computer Science Technical Report Number 87-9,* Rensselaer Polytechnic Institute, Troy, New York, March, 1987.

[Tay85] Taylor, Richard N. and Thomas A. Standish. "Steps to and Advanced Ada Programming Environment." in *IEEE Transactions on Software Engineering,* Vol. SE-11, No. 3, March 1985, pp. 302310.

[Tsi82] Tsichritzis, Dionysios C. and Frederick H. Lochovsky. Data Models. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

[Tur84] Turner, Ray. Software Engineering Methodology. Reston Publishing Company, Inc, Reston, Virginia, 1984.

[Ull82] Ullman, Jeffrey D. Principles of Database Systems, 2nd. Edition. Computer Science Press, Rockville, Maryland, 1982.

[Van84] Van Dam, Andreis. "The Electronic Classroom: Workstations for Teaching." in *Int. Journal of Man-Machine Studies* (1984) 21, pp.353-363.

[Was86] Wasserman, Anthony I., Peter A. Pircher, Davud T. Shewmake, and Martin L. Kirsten. "Developing Interactive Information Systems with the User Software Engineering Methodology." in *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 2, February 1986, pp. 326-345.

[Was87] Wasserman, Anthony L. and Peter A. Pircher. "A Graphical, Extendible Integrated Environment for Software Development." in *SIGPLAN Notices*, Vol. 22, No. 1, January 1987, pp. 131-142.

[Web88] Webster, Dallas E., "Mapping the Design Information Representation Terrain," in *IEEE Computer,* Vol. 21, Number 12, December 1988, pp. 8-23.

[Weg84] Wegner, Peter, "Capital Intensive Software Technology," in *Special Issue of IEEE Software,* Vol. 1, Number 3, July 1984, pp. 7-45.

[Wie87] Wiederhold, Gio. File Organization for Database Design. McGraw-Hill Book Company, New York, 1987.

[Wir85] Wirth, Nicklaus. Programming in Modula-2. Springer-Verlag, New York, 1985.

[Yau86] Yau, Stephen S. and Jeffrey J.-P Tsai. "A Survey of Software Design Techniques." in *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 6, June 1986, pp. 713-721.

[Yau87] Yau, Stephen S. "Relationship Between Data Engineering and Software Engineering," in *Proceedings, 3rd IEEE International Conference on Data Engineering,* Los Angeles, California, 1987, pp. 84.

[Yod83] Yoder, M. and Marilyn L. Schrag, "Nassi-Schneiderman Charts: An Alternative to Flowcharts for Design," in Tutorial on Software Design Techniques, 4th Edition, ed. Peter Freeman and Anthony I. Wasserman, IEEE Computer Society Press, Silver Springs, Maryland, 1983, pp. 506-514.

[You75] Yourdon, Edward. Techniques of Program Structure and Design. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1975.

[You79] Yourdon, Edward and Larry L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1979.

[Zel87] Zelkowitz, Marvin V. "An Editor for Program Design," in *U. S. Government Publication, Institute for Computer Science and Technology,* National Bureau of Standards, Gaithersburg, Md, 1987, pp. 242-246.

Index

A NEW DATA MODEL FOR CASE, 35
A Sample Design Session, 151
About the System, 149
Accessing Design Data, 122
Adding to the Interface Definition, 111
Advantages, 163, 166, 169, 170, 171
Allowable Values for Keywords, 115
Alternatives, 215
An Approach to Design Data Capture, 101
APPENDIX I: Prototype IDM Structure, 203
APPENDIX II: Operations on the IDM, 207
APPENDIX III: User Interface Issues, 221
APPENDIX IV: The Correspondence Between DFDs and DSDs, 224
Application-Oriented Organization, 141
Applying CAD Database Concepts to CASE, 4
Approaches for use with the IDM, 136
Approaches to CASE Data Models, 36
Approaches to Software Classification with the IDM, 117
Artificial Intelligence Techniques, 132
Associative Networks, 128
Attribute Search, 136
Calls, 211
Capture of Design Data, 166
CAPTURING DESIGN INFORMATION IN A CASE SYSTEM, 74
Changes to the VLSI Model, 40
Classification Matrix, 131
CLASSIFICATION OF SOFTWARE COMPONENTS, 109, 169
Cluster Theory, 127
Complex Objects, 19
Conclusion, 196
Constraints, 207, 211, 215, 218
Contributions, 7
Contributions of the Implementation, 195
Contributions to Software Engineering and CAD/CAM, 184
CONTRIBUTIONS TO THE FIELD, 181
Data Capture with the IDM, 104
Data Flow Design, 76
Data Model Requirements for Support of CASE and Software Reuse, 26
Data Modeling in CAD and CASE, 5
Data Structure Design, 79
Database Technology in CAD/CAM Applications, 17
Decision Tables, 92
Dependencies of the Retrieval Techniques, 133
Design Data Management in CASE Systems, 175
Desired Operations, 122
Details of the IDM, 47
Disadvantages, 164, 167, 169, 171, 172
Discussion, 132
DISCUSSION AND CONCLUSIONS, 200
Economy of Scale, 172
Engineering Data Models, 19

EVALUATION OF THE IDM, 162
Existing Systems for CASE, 176
Faceted Schema, 130
Finite State Machines, 91
For Archive Storage of Reusable Components, 33
For CAD/CAM, 179
For Capture of Design Data, 32
For Classification of Design Data, 32
For Retrieval of Design Data for Reuse, 33
For Semantic Modeling of CAD Data, 28
For Semantic Modeling of CASE Data, 31
For Software Engineering, 180
Formal Definitions, 26
Formal Semantics, 112
Functional Decomposition, 75
FUTURE WORK, 197
High Level Design Methodologies, 75
High Level Design Methods, 95
HISTORICAL REVIEW OF SEMANTIC DATA MODELING IN CAD, 11
Hybrid Models, 22
IMPLEMENTATION OF THE IDM, 149
Indexing Strategy, 123
Indexing Techniques, 124
Interactive Ada Workstation, 177
Interface Operations, 207
Introduction, 11, 12, 35, 39, 47, 58, 74, 89, 94, 101, 109, 117, 122, 136, 140, 149, 151, 176, 181, 197, 207
INTRODUCTION AND HISTORICAL REVIEW, 1
IPO Charts, 87
Levels of Abstraction, 173
LITERATURE CITED, 226
Low Level Design Methodologies, 89
Low Level Mappings, 98
Mapping the Design Methodologies to Program Structure, 94
Matching Needs with Available Components, 133
Molecular Objects, 21
Multilist Index, 137
Multilists, 125
Object in a Field, 23
Object-Oriented Design, 85
Operations, 209, 212, 216, 219
Operations and Practice, 58
Operations on the IDM, 59
Operations on the Software Archive, 145
Organization Based on Retrieval Method, 142
Organization of Implementation Archive, 146
Organization of Software Libraries, 141
ORGANIZATION OF THE SOFTWARE ARCHIVE, 140, 171
Outline of the Thesis, 8
Overview of ROSE, 24
Overview of the Design Process, 151
Pecan, 177

Practice, 63
Public Archives and Private Workspaces, 143
RELATED WORK, 175
Relationship of the IDM to Object-Oriented Program Design, 72
Research Topics, 197
Retrieval of Software Components, 170
RETRIEVAL OF SOFTWARE DESIGN DATA, 122
Scalability, 34
Semantic Data Models for Design Data, 179
Shortcomings of Traditional Databases for Engineering Design Data, 17
Software Catalogues, 124
Software Classification Options, 110
Software Engineering and CASE, 2
Software Module as a Static Object, 36
Software Reusability, 1
Software Through Pictures, 176
Standard Flowcharts, 89
Static Classification Schedule, 118
Storage of Design Data, 163
Structured Flowcharts, 90
The Alternative, 51
The Argument for Database Support of CAD and CASE, 11
The Call, 56
The Design Session, 152
The Extended Static Module Object, 38
The Functional Model, 23
The Hierarchical Model, 15
The IDM as a Partial Solution to Reusability in CASE, 162
The Interactive Development Model for CASE, 39
The Interface, 47
The Interface Definition of a Module, 110
The Network Model, 17
The Program Dynamic Structure Diagram, 103
The Program Static Structure Diagram, 103
The Relational Model, 13
To Archive Storage of Reusable Components, 194
To Capture of Design Data, 191
To Classification of Design Data, 192
To Retrieval of Design Data for Reuse, 193
To Scalability, 195
To Semantic Modeling of CAD Data, 184
To Semantic Modeling of CASE Data, 190
Traditional Data Models, 12
Use of Keywords for Software Classification, 113
Variable Keyword Lists, 119
Versions of Alternatives, 218